Generic Architecture Description for Retargetable Compilation and Synthesis of Application-Specific Pipelined IPs

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

Abstract—**Constraints of embedded systems and the shrinking time-to-market have elevated the importance of designer productivity and design predictability more than ever. To improve productivity, in ASIP approaches the system is designed with software and executed on a customized processor. In ASIP design flow, the processor is described in an Architecture Description Language (ADL) and the toolset is generated from that ADL automatically. However, in these approaches design predictability is low because the designer has little or no control over the quality of the final implementation.**

In this paper, we present a new design approach where the target processor or Intellectual Property (IP) does not have any predefined instruction-set and its datapath component netlist is described in a Generic Netlist Representation (GNR). The GNR is used by the toolset to generate the controller of the IP and the RTL of the design. 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. We have also developed a web-based interface for our toolset, so that users can upload and evaluate new IPs described in GNR.

I. Introduction

Tight constraints of embedded systems require careful design exploration and fine tuning of quality metrics such as performance, power consumption, area, or manufacturability. Shrinking time-to-market and increasing complexity of these systems has made *designer productivity* a vital factor for success. The logical way of gaining productivity is to increase abstraction level by designing the systems using software (high-level languages such as C) rather than directly implementing them in RTL. However, *design predictability* must not be sacrificed by increased abstraction level and as before the designers must be able to control the quality of the final implementation.

One increasingly popular option for implementing software is using Application-Specific Instruction-set Processors (ASIPs). An ASIP is customized for the application to meet the design constrains. Due to low volume and short life-span of ASIPs, automatic generation of toolset (e.g. compiler, simulator) and automatic implementation of processor are very important. In an ASIP design flow, an Architecture Description Language (ADL) is used to capture the processor behavior and to generate the toolset.

Over the past years many ADLs have been proposed. However the focus of majority of these ADLs has been either compilation or simulation of the programs but not synthesis of the processor. Only a few approaches have offered automated or semi-automated RTL synthesis of the processor. However, these approaches require very complex description of the processor and also do not provide enough control for the designer over the quality of the final implementation. We believe the main source of such limitations is that all ADL-based design flows always assume that the processor has a predefined instruction-set.

The ADLs can be categorized to *behavioral* and *structural*. The *behavioral* ADLs describe the functionality of instructions and synthesize the architecture from that behavioral description. Since all possible formats of instructions in the instruction-set (e.g. all operations with all possible addressing modes) must be described explicitly in these ADLs, they are typically very lengthy even for simple processors. On the other hand, since such ADLs only focus on the functionality of the processor, the implementation related information that may not change the functionality cannot be captured by them. Therefore, the designer has little or no control over the quality of the final implementation. For example consider Figure 1(a) that shows a possible datapath for implementing addition (ADD), multiplication (MUL), and multiply-and-accumulate (MAC) operations. When performing MUL, although the adder is not used, it still consumes power due to the activities on its input signals. To reduce the power consumption of the design, we can add input registers as shown in Figure 1(b). However, this design has two disadvantages: (1) even if clock period is longer than the accumulated delay of ADD and MULL, still MAC will take two cycles; (2) adding registers increases the load on the clock tree and hence can increase clock power consumption. An alternative approach for reducing the power consumption of this design is to use signal gating as shown in Figure 1(c). In this solution, whenever the functional units are not used, their inputs are locked using a gate signal. This example shows that meeting tight constraints may require fine-grained architecture adjustments. However, behavioral ADLs only capture functionality and timing of the operations. As a result, the design predictability is very low in these approaches.

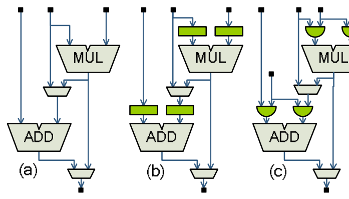


Figure 1- (a) A simple datapath, power optimization (b) using registers, (c) using gates.

The *Structural* ADLs describe the detailed component netlist of the processor (similar to HDLs) and then extract the instructions from the structural description of the controller and instruction decoder. These ADLs provide better design predictability than behavioral ADLs. However, designer productivity is very low because describing the controller and instruction decoder is very time-consuming and tedious. Also, extracting instructions from instruction decoder is very complex and usually limits the architectural features that these ADLs can support.

In addition to the aforementioned issues, using technology-dependent third-party cores is not supported in any ADL. Furthermore, all ASIP approaches always impose the overhead of the instruction decoder on the design. This overhead is not acceptable in many embedded systems where a dedicated hardware is designed for a fixed application.

To address these issues, in this paper we present a design approach for generating programmable and dedicated custom pipelined IPs from high level C description of the application. In contrast to ASIP approaches, our target architecture does not have a predefined instruction-set. In our approach, the accurate netlist of the datapath components is described in a Generic Netlist Representation (GNR). Using this GNR, a *cycle-accurate compiler* compiles C code of the application directly on the input datapath and generates the control words of each clock cycle. The result of this compiler and the input GNR is used to generate the controller and the simulatable/synthesizable RTL codes of the IP. 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 is formal, supports use of third-party cores, and the same GNR description is used for compilation, simulation and synthesis. Since the designer does not describe the controller in our approach, the GNR descriptions are much shorter than other ADLs. For example, the GNR of a MIPS like datapath is less than 300 lines. We have developed a web-based interface for our toolset, so that users can upload and evaluate new IPs 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 [11] and [12] respectively. In this paper we focus on how to model the architecture in the GNR and will explain its use for compilation and synthesis. The rest of the paper is organized as follows. Section II presents related works. Section III and IV explain our modeling approach and the syntax of GNR. Section V discusses the details of GNR using three examples. Section VI presents the flow of our toolset, followed by experimental results in Section VII. Section VIII concludes the paper.

II. Related works

Over the past decade, a few 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 [10].

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.

In our approach only the netlist of the datapath is captured in a format that is very close to HDL but includes additional properties for ports, components, and connections. Such properties enable automatic netlist completion and design rule checking. While GNR is low level enough for capturing all implementation details, it has enough high-level information for the compilation.

III. GNR modeling approach

In our approach, a custom processor or IP is composed of a set of input/output ports, and a netlist of components and connections. The components have a type, and may have input, output and control ports. The components may hierarchically contain other components and connections as well. Components and custom IPs have especial properties, called aspects, targeted for different tools such as compiler, and RTL generator. In this section, we describe our approach for modeling components and custom IPs.

A. Component model

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:

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

Where, *NiscArchitecture* is our top-level custom IP. Among these components, *NiscArchitecture, module*, and *controller* contain an internal netlist of components, while others are basic RTL components with no internal netlist.

The set of ports P_x is defined as follows:

$P_x = QP_x \cup IP_x \cup OP_x \cup CP_x$

Where QP_x is the set of clock ports, which may have zero or one element; IP_x is the set of input ports, OP_x is the set of output ports, and CP_x is the set of control ports. Each port p has a bit-width or size denoted by β_p. For example a register has one clock port, one input port, one output port, and one control port (i.e. load enable). In general, we call IP_x and OP_x *data ports*. We have separated clock ports and control ports from data ports to enable automatic netlist completion and validation.

Set of connections L_x is defined as follows:

$$L_x = QL_x \cup CL_x \cup DL_x \quad (1)$$

Where QL_x is the set of clock connections, CL_x is the set of control connections, and DL_x is the set of data connections defined as follows:

$$QL_x = \{(p_1, p_2) \mid p_1 \in QP_x \text{ and } p_2 \in (\bigcup_{y \in C_x} QP_y)\}$$

$$CL_x = \{(p_1, p_2) \mid p_1 \in CP_x \text{ and } p_2 \in (\bigcup_{y \in C_x} CP_y)\}$$

$$DL_x = \{(p_1, p_2) \mid p_1 \in (IP_x \cup (\bigcup_{y \in C_x} OP_y)), \text{ and } p_2 \in (OP_x \cup (\bigcup_{y \in C_x} IP_y))\}$$

The clock connections are between the clock port of component x and the clock of its children components. Similarly, control connections are between control signals of x and the control signals of its children. Data connections are between the input ports of x and the input ports of the children, input port of x and output ports of x, output ports of the children and input ports of the children, and output ports of the children and output ports of x. Furthermore, we limit the number of connections to input, clock and control ports to one connection. The only exception is the bus component's input port, which can be driven by multiple sources. However, all the sources must go through a tri-state buffer to ensure correct values on the bus. The limitation on number of connections is described as follows:

$$\forall\ (p_1, p_2), (p_3, p_4) \in L_x, \text{ if } p_1 \neq p_3 \text{ and } p_2 = p_4, \text{ then } p_2 \in IP_y \text{ and } \tau_y = bus$$

By distinguishing between the types of components and ports, we can (a) perform many static analysis and validation, and (b) generate many parts of the description automatically (see also Section V.C.1))

A_x is a list of aspects required by different tools for processing the component x. 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 generation of 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 be generated automatically from their internal components or, they can be explicitly specified by the designer. In this way, third party cores and pre-laid-out components, that may have special technology or manufacturability considerations, can also be modeled and used in the GNR (refer to Section V.B).

The components in a *NiscArchitecture* may represent a proxy to a component outside of the IP block. For example, a memory proxy represents a memory or cache hierarchy connected to the ports 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 outside component.

B. NiscArchitecture model

A *NiscArchitecture* is a component with additional properties. A *NiscArchitecture* ξ is modeled by (P_ξ, C_ξ, L_ξ, $CNST_\xi$, A_ξ), where P_ξ is the set of the ports, C_ξ is the set of internal components, L_ξ is the set of connections, $CNST_\xi$ is the set of constant fields in the control word used for constant and jump operations, and A_ξ is a collection of ξ's compilation, simulation and synthesis aspects denoted by CA_ξ, MA_ξ, and NA_ξ, respectively. Set P_ξ includes one clock port qp_ξ, input ports IP_ξ, and output ports OP_ξ:

$$P_\xi = \{qp_\xi\} \cup IP_\xi \cup OP_\xi$$

Note that a *NiscArchitecture* does not have any control ports. Since control ports are introduced to facilitate compilation and binary generation, they are used only inside the *NiscArchitecture*. The set of all control ports of the components inside the IP is defined as follows:

$$ICP_\xi = \{p \mid p \in \bigcup_{x \in C_\xi} CP_x\}$$

The set of components C_ξ includes one (and only one) component of type controller, and at least one memory proxy and one register file:

$$C_\xi = \{x \mid x \text{ is a component, and } \exists!\, x1 \in C_\xi \text{ where } \tau_{x1}=controller,$$
$$\text{and } \exists x2, x3 \in C_\xi, \text{ where } \tau_{x2}=memory\text{-}proxy, \text{ and } \tau_{x3}=register\text{-}file\}$$

The controller is a special component that drives the control signals of all other components at every cycle. The controller has an output port called *cwPort* that its bit-width is equal to the sum of the bit-widths of all control ports in the IP, plus the bit-widths of the constant fields in $CNST_\xi$:

$$\beta_{cwPort} = \sum_{p \in ICP_\xi} \beta_p + \sum_{f \in CNST_\xi} \beta_f$$

The set of connections L_ξ is defined similar to Equation (1). The only difference is that the set of control connections is defined only between the *cwPort* of the controller and the control ports of the components:

$$CL_\xi = \{(cwPort, p, s, e) \mid p \in ICP_\xi, \text{ and } s, e \in [0, \beta_{cwPort} - 1],$$
$$\text{and } e - s = \beta_p\}$$

Where s and e are start and end indices for connecting part of *cwPort* to the control port p. The start and end indices of every two connection in CL_ξ are not allowed to overlap in order to maintain the correct functionality.

Compilation aspect CA_ξ is modeled by (Γ_ξ, sPt_ξ, fPt_ξ, dPt_ξ). The Γ_ξ is a function that defines the ordering of the constant and control fields in the control word. The ordering must match the connection indices in CL_ξ. The sPt_ξ, fPt_ξ, dPt_ξ are storage components used for stack pointer, frame pointer and data segment pointer. The storage components can be separate registers or registers in a register file.

IV. GNR SYNTAX

We use XML language [14] to describe IP models in GNR. We define GNR syntax in XML Schema [15] 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.

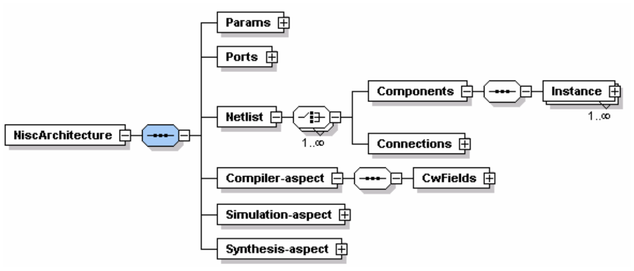


Figure 2- Block diagram of GNR schema for *NiscArchitecture*.

V. GNR MODELING EXAMPLES

In this section, we discuss modeling IPs in more details using three examples. We first explain how a simple component, namely a custom ALU, is defined in GNR. Next, we discuss modeling of complex third-party cores. Finally, we explain how components are integrated to form a simple IP that can execute C code.

A. Modeling a custom ALU

ALU is a component of type *functional-unit* (FU). 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 due to space limitations). For some components, it is also possible to generate the HDL description automatically from the component entity information and compiler aspect. 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 either 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 component of type *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>.

```
- <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.

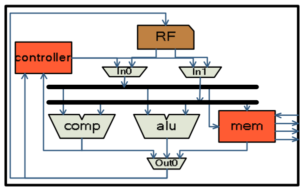


Figure 4- Block diagram of a simple IP.

B. Modeling third-party cores in GNR

To improve the area, power consumption, and performance of the generated IPs, designers may desire to include hardwired third-party cores. This requires proper support in ADL and RTL code generator. To the best of our knowledge, no ADL-based ASIP design approach is able to incorporate hardwired cores in a systematic way.

In the synthesis aspect of GNR components, we allow calling *core-translator* programs to generate proper input files for third-party core generators. For example, for Xilinx FPGAs, the core generator (i.e. LogiCore) requires a .xco file that describes the properties of the core. For other cores, additional information may be required in specific formats. The information of the core is extracted from GNR and formatted by the translator program to match the requirements of the target platform. For example, in GNR, the synthesis aspect of memory components calls an external program to generate proper .xco and .coe (memory content) files that LogiCore needs for generating memory cores. The *translator* programs have different implementations for different target platforms.

Another example is integer divider unit that is usually implemented by a hard or soft core. Dividers are costly functional units that are usually designed in pipelined fashion to improve the performance. Suppose that we have a four-stage pipelined divider core where each stage has a delay of two cycles. The goal is to describe such divider in ADL both for compilation and synthesis. From compiler point of view, to perform a divide operation, the inputs of the divider must be preserved for two cycles, and the division result is ready after eight cycles. Also, new inputs can be loaded every two cycles. The GNR model of such divider is a component of type *module* that contains a two-cycle *functional-unit* and three *registers*. The registers are connected to the output of the functional unit serially. The compiler-aspect of the functional unit contains the divide operation and the delays of the functional-unit and all of the registers are parameterized to two cycles. This way, the compiler detects that it can use the functional unit every two cycles and the results will be ready after eight cycles. To guarantee correct control generation, the registers should not have any control port and the control port of the module must be connected to the control port of the functional unit. To prevent the toolset from generating RTL for this module, we add proper commands to the simulation-aspect and synthesis-aspect of the module to call a core translator program. For example, if Xilinx FPGA is chosen as target platform, then the translator program generates proper .xco file based on the parameters passed to the divider component.

C. Modeling a simple IP

Figure 4 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.

```
- <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.

In this IP, suppose that a constant field of 10 bits is used for constants and offsets of jump operations. 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 4, 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.

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%. We also observed that such automation reduces the design and validation time by more than two times because it eliminates the unavoidable control connection errors.

The designer may also choose to explicitly specify the synthesis aspect of a *NiscArchitecture* to meet special constraints such as manufacturability. In that case, the connections and control fields must be specified manually.

VI. GENERATING RTL FROM GNR

Figure 6 shows the block diagram of our toolset. The inputs of the toolset are GNR description of the custom IP and the application C code. Currently, 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 V.C. The semantic correctness of the completed netlist is verified afterwards, and proper warning and error messages are reported bye *Pre-Processor*. The netlist checker reports unconnected ports, invalid connectivity, and non-existing component and port names. GNR modeling enables additional checking that is not possible using HDL-based structural descriptions. 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 result in an incorrect design behavior. Using such simple checking in GNR, most architecture problems are quickly determined.

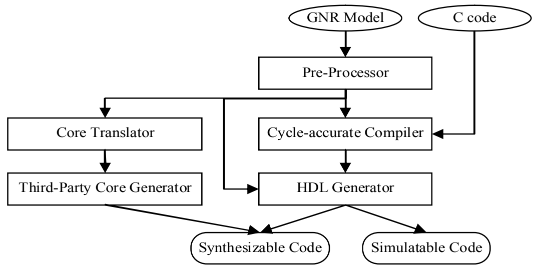


Figure 6-The flow of our toolset.

The *Cycle-accurate compiler* compiles the C code on the given GNR using the algorithm presented in [11]. If a specific operation required by C code is not supported by the given datapath, then compiler displays proper error messages. After compilation, the compiler generates the contents of data memory as well as the control words for every clock cycle of the execution.

The *HDL Generator* uses the GNR and the output of the compiler to produce the output 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 final synthesizable code.

Our IP design toolset can be used for creating new IPs or reprogramming existing IPs. An online version of the toolset is available at [13].

VII. EXPERIMENTAL RESULTS

We conducted two sets of experiments using our toolset and GNR. First, we compiled a fixed-point MP3 decoder (13000 lines of C code downloaded from [17]) on three processors and compared the results. As a second set of experiments, we explored different custom designs for implementing a 2D DCT. 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.

In the first experiment, we compiled MP3 code on three processors: (1) Xilinx MicroBlaze, (2) a NISC-style MIPS processor (DMIPS), and (3) a NISC customized for MP3 application (GD). All processors had an integer divider unit, and the compiler options were set to maximize optimization. Table 1 compares these processors in terms of cost and performance. The second column shows that clock frequency of 100MHz, 70MHz, and 95MHz was achieved for MicroBlaze, DMIPS, and GD, respectively. The third column shows the area of the three processors in terms of the percentage of utilized logic on the chip. As shown in the fourth column, MicroBlaze takes 2.7 million cycles to run the MP3 code, while DMIPS and GD take less than a million cycles. Considering both number of cycles and the clock frequency, DMIPS and GD run 2.04 and 3.1 times faster than MicroBlaze as shown in column fifth. Column sixth, seventh, and eight show number of lines of GNR code, simulatable code and synthesizable code, respectively. The size of GNR codes is significantly smaller than their corresponding simulatable and synthesizable code. Since synthesizable Verilog files include the structural design of the Xilinx cores, they are much larger than their simulatable counterparts. Although our GNR is XML-based (and hence relatively verbose), the description of the DMIPS and GD are very short compared to typical LISA descriptions, which are usually in the range of several thousands lines of code [8][9][10].

Table 1- Comparing MicroBlaze with two NISC processors.

Processor	clock freq (MHz)	Area (%)	#Cycles (million)	Speedup	Lines of GNR	Lines of Simulatable RTL	Lines of Synthesizable Code
MicroBlaze	100	11	2.7	1	-	-	-
DMIPS	70	13	0.92	2.04	301	1981	22300
GD	95	17	0.83	3.1	432	2490	23500

A. DCT implementation using GNR

In the second set of experiments, we start from a MIPS architecture (without the divider unit), called NMIPS, and customize it for the 2D DCT application. The Discrete Cosine Transform (DCT) and Inverse Discrete Cosine Transform (IDCT) are important parts of JPEG and MPEG standards. The definition of DCT for a 2-D *N*×*N* matrix of pixels is as follows:

$$F[u,v]=\frac{1}{N^2}\sum_{m=0}^{N-1}\sum_{n=0}^{N-1}f[m,n]\cos\frac{(2m+1)u\pi}{2N}\cos\frac{(2n+1)v\pi}{2N}$$

Where u, v are discrete frequency variables ($0 \le u, v \le N-1$), $f[i, j]$ gray level of pixel at position (i, j), and $F[u,v]$ coefficients of point (u, v) in spatial frequency. Assuming $N=8$, matrix C is defined as follows:

$$C[u][n] = \frac{1}{8}\cos\frac{(2n+1)u\pi}{16}$$

Based on matrix C, an integer matrix $C1$ is defined as follows: $C1$ = round($factor \times C$). The $C1$ matrix is used in calculation of DCT and IDCT: $F = C1 \times f \times C2$, where, $C2 = C1^T$. As a result, DCT can be calculated using two consecutive matrix multiplications. Figure 7(a) shows the C code of multiplying two given matrix A and B using three nested loops. In general, customization of a design involves both software and hardware transformations. To increase the parallelism in code, we unroll the inner-most loop of the matrix multiplication code, merge the two outer loops, and convert some of the costly operations such as addition and multiplication to OR and AND. In DCT, the operation conversions are possible because of the special values of the constants and variables. The transformed code is shown in Figure 7(b).

```
for(int i=0; i<8; i++)
   for(int j=0; j<8; j++){
      sum=0;
      for(int k=0; k<8; k++)
         sum= sum+A[i][k]×B[k][j];
      C[i][j]= sum;
   }
```

(a)

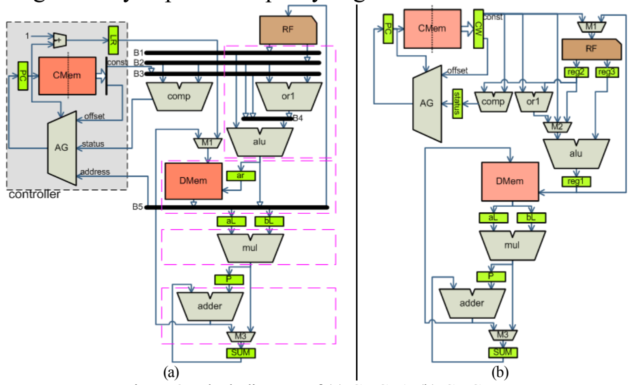
```
ij=0;
do {
    i8 = ij & 0xF8;
    j = ij & 0x7;
    aL= *(A+(i8|0) ); bL= *(B + (0|j) );  sum =  aL × bL;
    aL= *(A+(i8|1) ); bL= *(B + (8|j) );  sum+= aL × bL;
    aL= *(A+(i8|2) ); bL= *(B + (16|j) ); sum+= aL × bL;
    aL= *(A+(i8|3) ); bL= *(B + (24|j) ); sum+= aL × bL;
    aL= *(A+(i8|4) ); bL= *(B + (32|j) ); sum+= aL × bL;
    aL= *(A+(i8|5) ); bL= *(B + (40|j) ); sum+= aL × bL;
    aL= *(A+(i8|6) ); bL= *(B + (48|j) ); sum+= aL × bL;
    aL= *(A+(i8|7) ); bL= *(B + (56|j) );
    *(C + ij) = sum + (aL × bL);
} while(++ij!=64);
```

(b)

Figure 7. (a) Original and (b) Transformed matrix multiplication

By looking at the body of loop, four steps of computation can be identified: (1) calculation of the memory addresses of the matrix elements; (2) loading the values from data memory; (3) multiplying the two values; (4) accumulating the multiplication results. We design our custom datapath in a way that each of these steps is a pipeline stage. Figure 8(a) shows the proposed custom pipelined datapath called CDCT1. The datapath includes four major pipeline stages that are marked in the figure. Table 2 shows the summary of the customizations applied to architectures. It also shows the size of corresponding GNR files and the amount of code that has been modified to implement the customizations. After applying all customizations, we get to the CDCT7 that is shown in Figure 8(b). We capture all of these architectures in GNR and synthesize them using Xilinx ISE tool.

Table 3 compares the performance, power, energy, and area of the all the DCT implementations after synthesis. Compared to NMIPS, CDCT7 runs 10 times faster, consumes 1.3 times less power and 12.8 times less energy. Also, it occupies 2.9 times less area than NMIPS. All of the above experiments took less than 7 man-days. Using GNR and the toolset, we can quickly explore different architecture alternatives and significantly improve the quality of generated IPs.



(a) (b)

Figure 8- Block diagram of (a) CDCT1, (b) CDCT7.

Table 2- Summary of customizations and GNR changes.

	Customization	#lines in GNR	#modified lines in GNR
NMIPS	Initial generic architecture	247	-
CDCT1	Custom pipeline design	199	150
CDCT2	Optimizing interconnects	160	50
CDCT3	removing unused ALU and Comparator operations	160	10
CDCT4	controller pipelining 1	162	5
CDCT5	controller pipelining 2	164	5
CDCT6	bit-width reduction	164	10
CDCT7	multi-cycle multiplier, additional pipelined registers at the outputs of the RF	173	15

Table 3- Performance, dynamic power, energy, and area of DCTs.

	No. of cycles	Clock Freq	DCT exec. time(us)	Power (mW)	Enegy (uJ)	Normalized area
NMIPS	10772	78.3	137.57	177.33	24.40	1.00
CDCT1	3080	85.7	35.94	120.52	4.33	0.81
CDCT2	2952	90.0	32.80	111.27	3.65	0.71
CDCT3	2952	114.4	25.80	82.82	2.14	0.40
CDCT4	3080	147.0	20.95	125.00	2.62	0.46
CDCT5	3208	169.5	18.93	106.00	2.01	0.43
CDCT6	3208	171.5	18.71	104.00	1.95	0.34
CDCT7	3460	250.0	13.84	137.00	1.90	0.35

VIII. CONCLUSION

To improve the productivity of the designers while maintaining the predictability or the design, we presented GNR modeling approach. GNR captures programmable custom IPs at structural level and contains enough information for compilation, simulation and synthesis of IPs, yet it is much shorter than other ADL languages with similar capabilities. The new semantics in GNR significantly improve productivity by freeing designer from describing error-prone and tedious parts of the IP netlist. Using GNR, we could quickly explore different datapath architectures to gain ten and three times performance improvements for a DCT algorithm and a complete MP3 application, respectively.

REFERENCES

[1] P. Mishra and N. Dutt, "Architecture Description Languages for Programmable Embedded Systems", *IEE Proceedings on Computers and Digital Techniques (CDT), Special issue on Embedded Microelectronic Systems: Status and Trends*, volume 152, no 3, pages 285--297, May 2005.
[2] W. Qin and S. Malik, "Architecture Description Languages for Retargetable Compilation", in *The Compiler Design Handbook: Optimizations & Machine Code Generation*. 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*, January, 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, Jan 1998.
[7] R. Leupers, P. Marwedel, "Retargetable Generation of Code Selectors from HDL Processor Models", *European Design and Test*, 1997.
[8] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun, "C Compiler Retargeting Based on Instruction Semantics Models", *DATE*, 2005.
[9] K. Karuri, R. Leupers, G. Ascheid, H. Meyr, and M. Kedia, "Design and implementation of a modular and portable IEEE 754 compliant floating-point unit". *DATE* 2006.
[10] 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.
[11] M. Reshadi, D. Gajski, "A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths", CODES+ISSS, 2005.
[12] B. Gorjiara, D. Gajski, "Custom Processor Design Using NISC: A Case-Study on DCT algorithm", ESTIMEDIA, 2005.
[13] http://www.cecs.uci.edu/~nisc
[14] XML: http://www.w3.org/XML/
[15] XML Schema: http://www.w3.org/XML/Schema
[16] MIPS32® M4K™ Core, http://www.mips.com
[17] http://www.underbit.com/products/mad/