AHIR: A Hardware Intermediate Representation for Hardware Generation from High-level Programs

Sameer D. Sahasrabuddhe
IIT Bombay
sameerds@it.iitb.ac.in

Hakim Raja
IIT Bombay
huckym@iitb.ac.in

Kavi Arya
IIT Bombay
kavi@it.iitb.ac.in

Madhav P. Desai
IIT Bombay
madhav@ee.iitb.ac.in

Abstract

We present AHIR, an intermediate representation (IR), that acts as a transition layer between software compilation and hardware synthesis. Such a transition layer is intended to take advantage of optimisations available in the software compiler flow, and also to provide freedom to the low-level synthesiser, to explore options for application-specific implementations. Two operations become possible — reuse of computational resources across different modules in the design, and generation of an application-specific memory subsystem for faster data accesses.

AHIR presents a decoupled view of the program, in terms of control flow, data flow and memory accesses. Each module in AHIR is a triplet consisting of a control-path, data-path and a symbolic association between the two. Memory is represented only by load-store operators, while the memory subsystem is separately designed by the implementor.

In the program-to-hardware flow, a module in AHIR corresponds to a function in C. A complete program is a call-graph of functions, which is translated to a set of modules. The call-graph is restricted to be a DAG; recursion is not allowed. The representation is generated by a back-end in the software compiler, which runs after all source-level optimisations have been performed by relevant passes.

1. Introduction

Digital VLSI platforms form a spectrum of products, that offer a trade-off between good performance and rapid deployment. Application Specific Integrated Circuits (ASICs) represent the extreme in terms of performance, while microprocessors provide a generic platform that allows rapid development and deployment of applications.

A number of platforms have emerged, that fall between these extremes, such as Field Programmable Gate Arrays (FPGAs), Structured ASICs[13], Field Programmable Functional Arrays (FPFAs)[10], FPGA-ASIC hybrids[14] and reconfigurable processor cores[12].

The availability of such platforms has created interesting alternatives to microprocessors in digital applications. But simple and automated design flows are needed to utilise the potential of these platforms. An attractive avenue is a high-level synthesis flow that translates programming languages into hardware descriptions. This allows existing software programmers to design application specific digital hardware.

1.1. Related work

A number of attempts have been made to create a path from high level programming languages to hardware specifications. Some approaches either extend or restrict the C language, in order to allow programmers to write synthesisable C code.

SA-C[1] disallows pointers, and extends the syntax for arrays, designed specifically for DSP algorithms. Handel-C[2] provides a timing guarantee of one clock cycle per program statement, and constructs such as parallel execution, variable width data-types, semaphores, channels, etc. But the compiler cannot optimise the code, since any rearrangement in the statements will violate the timing model.

Another approach is to introduce a hardware intermediate representation in a software compiler, and send this to a hardware back-end. The representation is based on data flow graphs[4], augmented to handle side-effects in external memory. The Phoenix project uses an intermediate representation called PEGASUS for a compiler flow from C to hardware[5]. PEGASUS is a data flow representation, that uses handshakes to exchange data between operators designed as micropipeline stages[11].

Synchronous data flow[6] is a subset of data flow graphs, that is well-suited for DSP applications. Here, the number of tokens consumed or produced by each node on execution is specified *a priori*, independent of the data. This disallows conditional execution of nodes, and hence cannot be used for general applications.

1.2. Our approach

Like PEGASUS, AHIR uses handshakes to transfer data across operators. But where PEGASUS assumes unlimited hardware, AHIR includes hardware reuse as one of its goals. This is achieved mainly by decoupling the control and data flows in the program.

AHIR is an attempt to provide a synthesis path that builds on existing high-level compilers. The representation is generated by a compiler back-end, independent of the input language used. The program may have been written in any kind of programming environment, such as sequential or concurrent execution.

The design is expressed only in terms of operations, and mandates minimal timing constraints on the implementation. The synthesiser has freedom in exploring various implementation options within these constraints. AHIR thus positions itself as a convenient transition point that glues together high-level compilation and low-level synthesis flows.

2. Formal specification for AHIR

Two different, but closely related, flows are exposed in AHIR: control-path and data-path. The data-path consists of hardware elements connected by wires. The control-path is a state machine, that defines the sequence in which different operations are triggered. Events in the data-path trigger state changes in the control-path, which in turn trigger more events in the data-path.

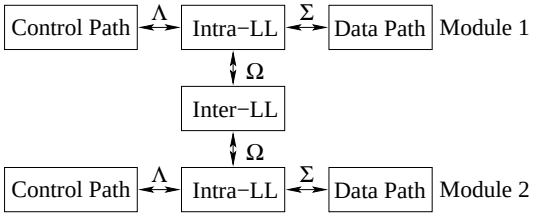


Figure 1. Communicating IR modules

A program expressed in AHIR is a collection of modules. A module is a tuple consisting of three entities — data-path, control-path and a link-layer that provides a symbolic association between the two. A module communicates with other modules using two mechanisms — a link-layer that passes control information, and external memory for passing data values.

Table 1. The Intermediate Representation

IR program :	({IR module}, Inter-LL, Ω)
IR module :	(DP, CP, Intra-LL, Σ, Λ)
DP :	data-path
CP :	control-path
Inter-LL :	inter-module link-layer
Ω :	alphabet for inter-module signalling
Intra-LL :	intra-module link-layer
Σ, Λ :	alphabets used by the DP and CP respectively

The different paths and link-layers interact through the exchange of symbols. The set of symbols associated with a component is called its alphabet. The data-path uses alphabet Σ, while the control-path uses alphabet Λ. Similarly, the interaction between modules is represented by symbols from the alphabet Ω.

The two paths within a module communicate through symbolic handshakes. The control-path emits a symbol to initiate an operation in the data-path, while the data-path eventually emits a symbol to signal completion. This mechanism ensures delay-insensitive synchronisation between the two paths.

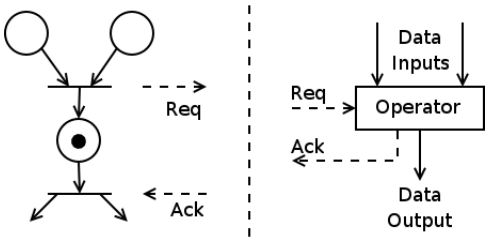


Figure 2. Symbolic handshakes

2.1. Data-path

The data-path is a collection of operators that accept multiple inputs and may produce multiple outputs. Each output is connected to a wire, that transmits the generated value to the inputs of other operators. An output produced by the operator is registered, and remains valid on the wire, until it is updated by a subsequent re-invocation.

Table 2. Data-Path

DP :	(DE, W, MA, Σ)
DE :	$O \cup D \cup M$
O :	operator nodes with a sequence of operands
D :	decoders that translate boolean values to symbols
M :	multiplexers
W :	wires with a single driver and multiple loads
MA :	memory access elements
Σ :	symbolic alphabet for the data-path

Each operator in the data-path is triggered by an input symbol $Req \in \Sigma$. The operator produces a symbol $Ack_j \in \Sigma$ for each output $result_j$ of its m data outputs. Thus the availability of each data output is signalled independently.

Table 3. Data-path operator

$operator$:	$(operation, [opnd_k], Req, \{(result_j, Ack_j)\})$
where $opnd_k$:	input data transmitted by a wire $w_k \in W$
$Req \in \Sigma$:	input symbol that triggers the operator
$result_j$:	one of possibly multiple data results
$Ack_j \in \Sigma$:	output symbol indicating availability of $result_j$

The data-path incorporates “memory access units” that

accept address and data for memory operations. These units expect to present a load/store request to the memory subsystem, that is guaranteed to be serviced *eventually*. The addresses are generated by ordinary operators in the data-path. Mechanisms for external storage can be chosen independently by the implementor.

2.2. Control-path

The control-path is a Petri net[8], that expresses the sequence of operations in the data-path. Transitions in the Petri net represent initiation and completion of various operations in the data-path.

Table 4. Control-Path

$CP : (T, P, E, M_0, \Lambda)$
$T : \{transition\}$
$P : \{place\}$
$E :$ edges $(u, v) \in (T \times P) \cup (P \times T)$
$M : P \rightarrow \{0, 1, 2, ...\}$ labelling function
$M_0 :$ initial labelling
$\Lambda :$ symbolic alphabet for the control-path

A transition is termed an *input* or *output* transition, based on whether it receives or emits a symbol when fired. When an output transition is enabled, it immediately fires, producing an output symbol for the environment. When an input transition is enabled, it waits for the environment to produce the associated input symbol. An input symbol that arrives when the corresponding transition is not enabled, is considered an *error*.

2.3. Link-layers

Events in the modules need to be associated in a way that represents the behaviour of the original program. This association is provided by the link-layer. There are two kinds of link-layers — the inter-module link-layer (Inter-LL) and the intra-module link-layer (Intra-LL).

Table 5. Link-Layers

Intra-LL : $((\Lambda \rightarrow \Sigma, \Sigma \rightarrow \Lambda), (\Lambda \rightarrow \Omega, \Omega \rightarrow \Lambda))$
Inter-LL : $\{(\text{Server}, \{\text{Client}\})\}$
Client : (Lock, Unlock, Grant, Release, $\{$(Req, Ack)$\}$)
Server : $\{$(Req, Ack)$\}$

The Inter-LL translates symbols exchanged between control-paths of multiple modules. The Intra-LL manages the symbols exchanged by the control-path, data-path and the Inter-LL.

The Inter-LL also captures the arbitration signals used by modules to gain access to other modules. When a server module is accessed by one of multiple client modules, the client must acquire access using the `Lock` symbol. A arbitration mechanism responds by generating a `Grant`. Similarly the client releases access using `Unlock` which is acknowledged by the corresponding `Release`. These symbols are defined in the alphabet Ω. The client then communicates with the server using `Req-Ack` handshakes translated through the Inter-LL.

2.4. Delay constraints

In Fig. 3 we show a hypothetical example, with associated delays. The numbered delays d_0 to d_5 in this figure are not individual values, but representatives of their respective class of delays.

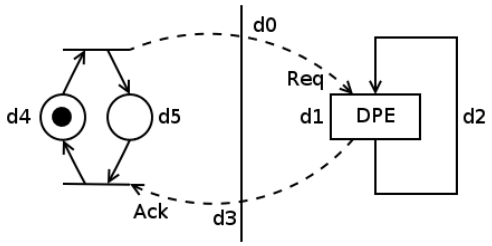


Figure 3. Delays

Delays are captured by handshakes between the control and data-paths in AHIR. In order to guarantee correct execution, AHIR imposes delay constraints on the *implementation* of a specification.

The arrival of an input symbol $Ack \in \Sigma$ at an input transition, when it is not enabled, is considered an error. For this constraint to hold, the implementation must ensure that tokens to the enabling places must arrive before the symbol arrives, so that the transition is activated in time. This is captured by the following expression:

$$d_5 \leq d_0 + d_1 + d_3$$

The dual of this fork occurs, when a data-path element produces a result and emits the corresponding output symbol. The control-path will eventually trigger some other data-path element that uses this result. Data must arrive at the second element before the request $Req \in \Lambda$.

$$d_2 \leq d_3 + d_4 + d_0$$

The term $d_0 + d_3$, made of delays in the link-layer, is common to both expressions. An implementation can always guarantee faithfulness by sufficiently padding either or both of these delays, to satisfy the inequalities.

3. Simulation

We describe a simulation scheme for AHIR that forms a reference implementation, modelling the fastest possible execution, without violating the timing constraints. We start by setting the interconnect and link-layer delays to zero.

$$d_2 = d_3 = 0; \;\; d_5 = d_0 = 0$$

The above expressions satisfy the inequalities stated earlier. One of the remaining delays d_4 and d_1, must be set to δ(delta), so that the notion of a "simulation cycle" is defined. Setting d_1 as δ is a natural choice, since it corresponds to combinational delays in the data-path.

$$d_4 = 0; \;\; d_1 = \delta$$

In the simulation, the translation of symbols across the link-layer is instantaneous. The response of the control-path to input tokens is also instantaneous. But the response of the data-path is available only in the next simulation cycle. The simulation can now be represented by a simple loop:

for ever **do**
  repeat
    execute CP, Intra-LL, Inter-LL
  until all signals stabilise
  execute DP
end for

4. Back-end for a C compiler

We describe a compiler back-end that translates C programs to their AHIR representation. The input is a C program in the Static Single Assignment (SSA) form. The output is a formally correct translation of the program to an IR form. The back-end assumes that all source-level optimisations have been already performed on the program. Recursive functions, function pointers and variable arguments are currently not supported.

4.1. Static Single Assignment (SSA)

The SSA form[7] is designed to remove the notion of a "variable" from a program. Every statement that defines a variable, provides a unique "version" of that variable. At the exit of a control structure such as a branch or a loop, multiple definitions of the variable may be available, only one of which is valid. This is captured by the ϕ-function in SSA, as seen in Fig. 4.

4.2. Control Data Flow Graphs (CDFG)

The SSA program is translated into a Control Data Flow Graph (CDFG) as shown in Fig. 5(a). In a CDFG, instructions are represented as nodes, connected by control and data edges. The data edges represent the flow of values from their definition to their use, while control edges represent the sequence of operations to be enforced. The edges in the CDFG arise from three kinds of dependences in the original program:

- data dependences between instructions, that create data as well as control flow

```
S0: d1 = m1 + n1
S1: b1 = m1 - n1
S2: if (b1 > 0)
{
S3: a1 = b1 + c1
S4: d2 = e1 + a1
}
S5: d3 = φ(d2,d1)
S6: x1 = d3 + 2
```

(a) SSA form

```
A1 = add m1, n1
S1 = sub m1, n1
C1 = cmpgt S1, 0
Br = br C1,L1,L2
L1 = label
A2 = add S1, c1
A3 = add e1, A2
L2 = label
P1 = phi A3, A1
A4 = add P1, 2
```

(b) Instructions

Figure 4. A sample input program

- control dependences in loops and branches, that create control flow
- external dependences for memory references, that create control flow

The current implementation takes the most conservative approach towards external data dependences. Control edges are introduced between memory operations, that preserve the order in which they occur in the original program. This allows only one memory operation at a time, but a smarter implementation may use reference analysis to parallelise memory accesses where possible.

4.3. Control and Data Paths

The control and data-paths in AHIR are obtained from the control and data edges in the CDFG respectively, as shown in Fig 5. A number of details have been omitted from the figure, in order to keep it readable. The symbols in alphabet Σ for each data-path element are not shown. Every pair of transitions connected by an edge also has a place along that edge. Each transition is associated with an incoming or outgoing symbol.

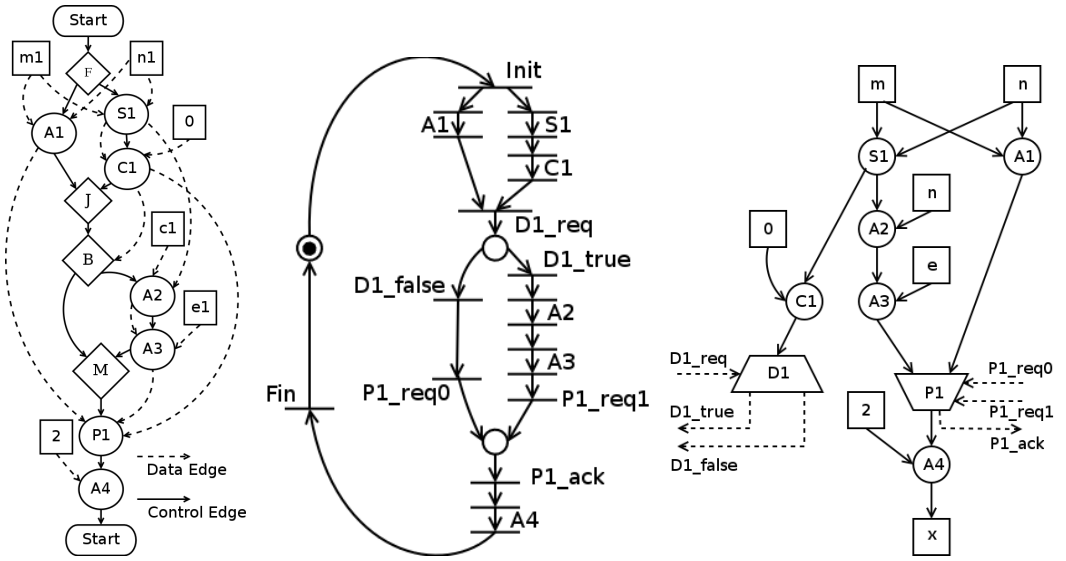
5. Specific structures in AHIR

Two structures are visible in AHIR, that represent points of execution where the control and data-paths directly influence each other. These points occur in the form of branches and ϕ-functions.

5.1. Conditional branches

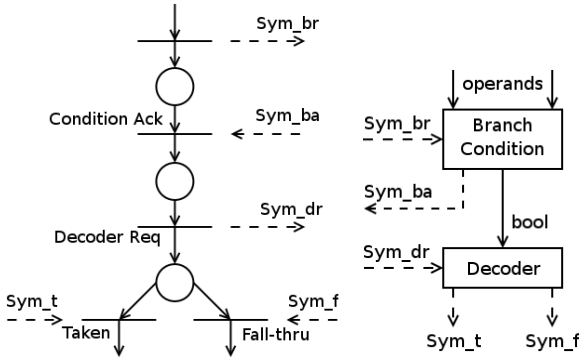
At a branch instruction, a boolean value generated in the data-path is used by the control-path to decide the next operation. AHIR uses a *decoder* element in the data-path to translate the boolean value into symbols. This element is activated by an input Req, but emits one of two output symbols — $true$ or $false$, based on a single boolean input.

The control-path creates a place with two output transitions, called as a "choice" in a Petri net. The symbol from the decoder causes one of the two transitions to fire, executing the appropriate branch in the control-path.



(a) CDFG (b) Control-Path (c) Data-Path

Figure 5. Deriving the hardware specification from a CDFG



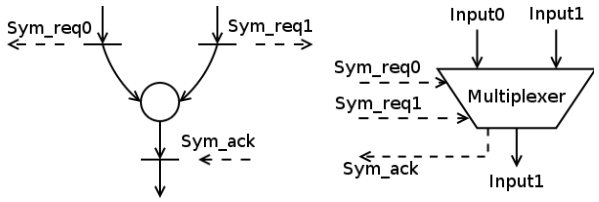
(a) Petri net Choice (b) Condition Decoder

Figure 6. A conditional branch

5.2. The multiplexer as a ϕ-function

The ϕ-function in SSA represents a point where the control-path influences the flow of values in the data-path. Two control edges reach a ϕ-function, from two different preceding basic blocks. Associated with each control edge is a data edge whose value is forwarded by the ϕ-function if that control edge is followed.

The ϕ-function is replaced by a multiplexer in the data-path, with two data-inputs and corresponding two request symbols. When it receives one of the two symbols, the multiplexer forwards the corresponding data-input to the output, and generates an acknowledge. The schematic for a multiplexer and the corresponding structure in the control-path are shown in Fig. 7.



(a) Control-Path (b) Data-Path

Figure 7. A ϕ-function in AHIR

6. Representing a complete C program

A C program consists of a call-graph, where nodes represent functions, and an edge (f, g) implies that function f includes a call to function g. The function-calls involve transfer of data as arguments and return values, along with a transfer of control from the caller to the callee. AHIR specifies a post-box model for passing of arguments between such functions, and uses the *inter-module link-layer* to signal function calls and returns.

The formal parameters of a function form its input address space, while its return values form the output address space. Every caller function must read from or write to these address spaces when calling a function. The linker assigns memory locations to these address spaces for each function, and propagates these locations to all its caller functions.

The caller function f first acquires access to the called function g. It then writes to the specified input locations and emits a request symbol in Ω, that is passed through the Inter-LL, to the function g. The function g indicates completion using another symbol in Ω. The caller can now read the return values from the corresponding output locations.

6.1. Current state of the AHIR compiler

A compiler flow that utilises this IR has been implemented using LLVM[3]. The first step is an LLVM backend that produces a hardware specification in AHIR, as an XML file. This description is made of one module for each function in the input program, with arguments and static allocations assigned to memory locations from a global address space. The function calls are translated to control and data communication as described.

This XML specification is used by stand-alone programs to generate a SystemC simulation model, and synthesisable VHDL code. The SystemC simulator can generate a trace of events occurring in the modules. The VHDL is meant to be a proof of concept, with an initial focus on correct and complete execution. Hence, the XML specification is mapped directly to VHDL, with one entity per control or data-path, and one signal per symbol in the entire description.

7. Conclusion

We have established a complete flow from a high-level C program to a hardware description using AHIR. AHIR provides a glue layer that combines software compilers with hardware synthesis to enable a novel hardware design flow. Such a flow makes "soft" architectures feasible, that increase flexibility in design, at acceptable hardware overheads.

The development process begins with an executable source program, that is verified at a software level. The IR presents a complete specification of the hardware, while the simulator provides information about memory accesses and operator utilisation to aid in hardware optimisation. Hardware created in this manner may not compete with hand-designed implementations in performance, but the savings in the overall design cycle are large enough to offset the performance overheads.

8. Opportunities

The decoupled nature of the specification allows transformations that optimise the resulting hardware. For example, computing resources can be shared across modules, by generating a joint specification of multiple data-paths.

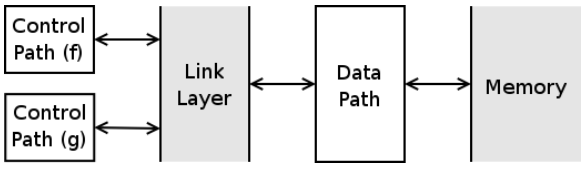


Figure 8. Joint implementation of modules

The intra-module link-layer provides the necessary flexibility for such a reuse. The joint specification can reuse resources, since AHIR allows arbitrarily complex elements in the data-path (Section 2.1). A transformation can map operations across the original data-paths to such elements in the combined data-path, provided the symbolic handshakes are correctly interpreted.

The memory system can also be implemented independently, and made arbitrarily complex in order to improve access times and reduce bottlenecks.

8.1. Equivalence

The generated AHIR specification is equivalent to the source program. This can be shown by demonstrating an isomorphism between the CDFG derived from the source, and the AHIR specification[9]. A similar equivalence must be maintained by optimisations performed on the specification, and by the hardware description generated from the specification.

References

[1] Cameron Project and Single Assignment C (SA-C). http://www.cs.colostate.edu/cameron/.

[2] Celoxica: Software-Compiled Systems Design. http://www.celoxica.com/.

[3] The LLVM Compiler Infrastructure. http://llvm.org/.

[4] A. L. Davis and R. M. Keller. Data Flow Program Graphs. *Computer*, 15(2):26–41, Feb 1982.

[5] T. C. Girish Venkataramani, Mihai Budiu and S. Goldstein. C to Asynchronous Dataflow Circuits: An End-to-End Toolflow. In *International Workshop on Logic and Synthesis*, 2004.

[6] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245, Sep 1987.

[7] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[8] T. Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77, 1989.

[9] S. D. Sahasrabuddhe. Equivalence between an input CDFG and the generated AHIR specification. Technical report, IIT Bombay, October 2006.

[10] G. J. Smit, P. J. Havinga, L. T. Smit, P. M. Heysters, and M. A. Rosien. Dynamic Reconfiguration in Mobile Systems. In *Lecture Notes in Computer Science*, volume 2438, page 171. Springer Berlin / Heidelberg, Jan 2002.

[11] I. E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, 1989.

[12] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 225–235, New York, NY, USA, 2000. ACM Press.

[13] B. Zahiri. Structured ASICs: opportunities and challenges. In *21st International Conference on Computer Design*, pages 404–409, Oct 2003.

[14] P. S. Zuchowski, C. B. Reynolds, R. J. Grupp, S. G. Davis, B. Cremen, and B. Troxel. A hybrid ASIC and FPGA architecture. In *IEEE/ACM International Conference on Computer Aided Design*, pages 187–194, Nov 2002.