A competitive pathway from high-level programs to hardware specifications

A thesis submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

by

Sameer D. Sahasrabuddhe
(Roll No. 02429002)

Under the guidance of
Prof. Kavi Arya
and
Prof. Madhav P. Desai

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY–BOMBAY
2009

Thesis Approval

The thesis entitled

A competitive pathway from high-level programs to hardware specifications

by

Sameer D. Sahasrabuddhe
(Roll No. 02429002)

is approved for the degree of

Doctor of Philosophy

Examiner	Examiner
Guide	Co Guide

Chairman

Date:

Place:

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea or data or fact or source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Sameer D. Sahasrabuddhe
(Roll No. 02429002)

Date: ______________

Place: ______________

Abstract

High-level synthesis is the process of generating hardware from high-level programs. An effective high-level synthesis flow can reduce the cost of designing hardware systems in two ways: eliminating the need for verification of the circuits, and making hardware design accessible to a large set of people.

We present a high-level synthesis flow that supports a large class of high-level programming languages while providing a verifiable and scalable path to a hardware implementation. The flow introduces an intermediate step between software and hardware compilation in the form of an intermediate representation called AHIR[1]. This decouples the high-level issues encountered when writing programs, from low-level issues in the hardware implementation. We describe a translation method that always produces a correct AHIR specification from a high-level program, thus eliminating the need for verification.

An AHIR specification is factorised into three components: control-flow, data-flow and storage. The three components can be analysed and transformed separately without affecting each other, enabling optimisations that can scale with the size of the circuit. We demonstrate this with an optimisation that improves resource utilisation in the data-path using a static analysis of the control-path. The specification is independent of implementation delays and can be routinely mapped to hardware. Timing correctness can be guaranteed by satisfying a set of single-sided delay constraints.

We conclude by comparing the performance of circuits generated from high-level programs using AHIR, with two extremes: the same programs running on a microprocessor, and hand-crafted circuits that implement the equivalent behaviour. The generated circuits are competitive with microprocessor implementations, but the performance is less than that of hand-crafted circuits. This gap can be bridged by future work that leverages the factorisation introduced in AHIR.

Key-words: High-level Synthesis, Electronic Design Automation, Intermediate Representation, Behavioural Synthesis, Systems Design

Contents

List of Tables

List of Figures

“The Unix-nature is simple and empty. Because it is simple and empty, it is more powerful than a typhoon.”
“Moving in accordance with the law of nature, it unfolds inexorably in the minds of programmers, assimilating designs to its own nature. All software that would compete with it must become like to it; empty, empty, profoundly empty, perfectly void, hail!”

Master Foo Discourses on the Unix-Nature

Chapter 1

Introduction

Digital VLSI platforms form a spectrum of products that offer a trade-off between good performance and rapid deployment. On the one hand, Application Specific Integrated Circuits (ASICs) are used for creating specialised high-performance implementations, and on the other hand, microprocessors are used as a generic platform that allows rapid development and deployment of applications at the cost of performance. Other platforms fall between these extremes, such as Field Programmable Gate Arrays (FPGAs), Structured ASICs[2], Field Programmable Functional Arrays (FPFAs)[3], FPGA-ASIC hybrids[4] and reconfigurable processor cores[5].

This diversity in hardware platforms has created interesting opportunities for implementing complex digital VLSI applications. But the design of large hardware systems is an expensive venture due to the need for highly trained manpower, and the need to verify the system at every step of the process. These costs can be reduced through high-level synthesis, the process of generating hardware from high-level programs.

High-level synthesis starts with an executable specification, i.e., a representation of the desired functionality in a machine-readable and simulatable form[6]. Such an executable specification can be in the form of a program written in a high-level programming language. The executable specification is translated into a circuit implementation by a compiler flow that lowers the level of abstraction in stages.

The use of high-level programs as the starting point makes hardware design accessible to a very large set of users. If the compiler is guaranteed to produce a correct implementation of the input specification, the resulting hardware does not have to be verified. Only the behaviour of the input specification has to be verified, which can be done using existing software verification practises. This approach also fits in with the existing hardware design flow that starts with an executable specifications that is translated manually to a hardware implementation.

1.1 Our work

Current efforts in high-level synthesis have at least one of two important shortcomings: restrictions are imposed on the input language, or the abstraction is not sufficient enough to hide hardware details. As a result, programmers are unable to use standard practises in writing programs and the compiler is restricted in the scope of optimisations used on the generated hardware.

The main aim of our work is to explore the design of a high-level synthesis process that preserves common practises in software programming while providing a verifiable and optimisable path to a hardware implementation. This requires a compiler flow that can generate efficient circuits from complex high-level programs. Such a compiler flow must have the following features:

1. The flow should be independent of the programming language used.
2. The flow should guarantee a correct implementation of the specified behaviour.
3. The flow should support optimisations that can scale to very large systems.

We present a compiler flow that achieves this goal by introducing an intermediate step between software and hardware compilation in the form of an intermediate representation called AHIR[1]. This decouples the high-level issues encountered when writing programs from low-level issues in the hardware implementation. We describe a translation method that always produces a correct AHIR specification from a high-level program, thus eliminating the need for verification.

An AHIR specification is factorised into three components: control-flow, data-flow and storage. The three components can be analysed and transformed separately without affecting each other, enabling optimisations that can scale with the size of the circuit. We demonstrate this with an optimisation that improves resource utilisation in the data-path using a static analysis of the control-path. The specification is independent of implementation delays and can be routinely mapped to hardware. Timing correctness can be guaranteed by satisfying a set of single-sided delay constraints.

1.1.1 Programming language independence

Software programming is an evolutionary process — typical software projects start off with an aim to first correctly implement the intended application, and later evolve in terms of optimisations for performance. But these optimisations in the program body are always constrained by the need for maintainability. This requires that the program should reflect higher-level abstractions so that the meaning is clear even at the cost of performance.

AHIR helps in this respect since it does not introduce any modifications or constraints in the programming language itself. No assumptions are made about the underlying hardware when mapping the high-level language to AHIR. Programming style is not affected by low-level considerations such as impact on performance, costs, etc., other than the ones that are commonly applicable in software programming.

Since the input language is not reinterpreted in any way, all optimisations normally performed by the source level compiler remain available. This leverages all the existing work in software compilation which is usually independent of the target platform. Such optimisations can lead to significant improvements in performance since they are implemented at a higher level where more information is available.

1.1.2 Correctness

Correctness is one of the cornerstones of effective design automation. If it can be shown that a given high-level synthesis flow always produces a correct implementation, then the effort required for verifying the implementation is eliminated. The only effort required is for the functional verification of the input specification.

We show that our method produces an AHIR specification that is correct *by construction*. The compiler translates an input C program to an AHIR specification using a CDFG as an intermediate step. The first step of translating the program to a CDFG is a routine one that does not need to be verified separately. We show that in the second step, the generated AHIR specification is equivalent to the CDFG and hence it is a correct implementation of the input program. If the input C program correctly implements the intended algorithm, then the generated AHIR specification also correctly implements that algorithm. The resulting hardware is also correct if it satisfies the delay constraints defined in AHIR.

Any optimisation defined on AHIR must ensure that every transformation preserves the behaviour specified by the input program. Since the original AHIR description is shown to be correct *by construction*, every transformation must only ensure that the output description correctly implements the input description. Such transformations can then be composed to achieve desired optimisations, with the guarantee that the final circuit is also correct.

1.1.3 Easily verifiable implementations

AHIR specifies a very basic set of delay constraints that ensure validity of data at every event in the execution, as described in Section 2.5.1. When the delays in an implementation satisfy these constraints, the implementation is guaranteed to be *correct*. In practise, it is easy to satisfy these constraints by sufficiently padding the delays when implementing the control-data handshakes.

1.1.4 Scalability

Due to the factorised nature of AHIR, different components can be analysed and modified separately in order to optimise the resulting implementation. This factorisation makes it possible to design efficient analyses and transformations that can scale with the size of the circuit. For example, in Chapter 4, we describe a native optimisation in AHIR that uses a static analysis of the control-path to improve the utilisation of resources in the data-path. The analysis used is shown to be almost linear in complexity with respect to the size of the program.

1.2 Related work

A number of attempts have been made to create a path from high-level programming languages to hardware descriptions. These can be loosely categorised as follows:

1.2.1 Improvements over RTL

Some efforts attempt to raise the abstraction in an RTL description in order to support higher-level constructs. For example, Bluespec[7] describes the behaviour of a system in terms of explicit state entities and *guarded atomic actions*. The control logic for the interaction of these actions is synthesised automatically. The resulting hardware has been shown to be competitive with hand-coded RTL.

But this is a tool meant for a hardware designer, not a programmer. An effort that proposes a new language for hardware design introduces new requirements in the skills needed to use that language. The user has to be aware of the components involved, and the low-level state that is being manipulated by operations in the system. The language can be quite powerful in expressing the architecture of the hardware, but the target user is a hardware designer who can effectively utilise this expressive power.

1.2.2 Modified high-level languages

Some efforts extend existing programming languages and/or interpret them as hardware descriptions. These systems essentially propose new programming languages that may be superficially similar to the parent languages. But it becomes difficult to retain standard programming practises in this setup — some of the common knowledge gained by programmers has to be revaluated in this new use of the language.

The language SA-C[8] for example is a purely functional subset of the C programming language. SA-C declares that variables represent values and not storage. Thus, pointers are not allowed in SA-C, and information is passed by value. The language introduces new syntax to work with arrays, which is especially powerful in expressing DSP algorithms.

But SA-C makes an *a priori* choice with respect to the programming paradigm used. Although the syntax superficially represents an imperative language, the program must be designed in a functional style. SA-C effectively creates a new language, and the programmer is forced to revaluate existing programming practises within the scope of this modified language.

On the other hand, Handel-C[9] is a language that guarantees complete ISO-C compatibility and further extends it to include features such as arbitrary-width data types, primitives that specify parallel or sequential execution, thread synchronisation mechanisms, etc. These features can prove very effective in describing the intended hardware.

An important property of Handel-C is that the compiler guarantees a cycle-accurate implementation of every program, which facilitates verification. But this cycle-accurate nature restricts the scope of optimisation, since the compiler cannot reorder or remove instructions. As a result, the programmer must manually optimise sequences of instructions. This can be an error-prone process, and opportunities could easily be missed simply because they were not obvious to the programmer.

In both these examples, the focus is on the language used by the designer. The designer is encouraged to use specific language features to describe the intended hardware, instead of the compiler inferring hardware from the behaviour of the program. Thus, although both these languages claim a relationship with C, they in fact represent completely different programming styles, hidden behind a syntax that resembles C.

1.2.3 High-level programs as hardware specifications

Some efforts simply use standard programming languages as a starting point that provides a hardware specification rather than a description. This approach makes it easy for programmers to write a hardware specification like any other program, while the compiler is free to automatically infer a suitable implementation. Such an approach usually employs an intermediate representation as a transition step from software to hardware.

For example, the Phoenix[10] project uses an intermediate representation called Pegasus[11] for a compiler flow from C to hardware[12]. Pegasus has a data-flow architecture, where operations use asynchronous handshakes to exchange data along data-flow edges. Control-flow is represented as predicates associated with the operations. Special token flow edges are used to represent dependences between memory operations. A description in Pegasus can be implemented directly by translating each operation to a micropipeline stage. The compact nature of Pegasus allows the compiler to natively implement a number of high-level transformations.

The SPARK[13] project uses an internal representation based on hierarchical task graphs (HTG). Statements that have no control-flow between them are aggregated together into basic blocks. These basic-blocks are used in HTG nodes, that form hierarchical structures such as branching and loops. This hierarchy captures all the information that is useful for high-level transformations such as code motion and speculation. The compiler uses a heuristic that combines these transformations with scheduling and resource binding in order to improve the schedule length as well as resource utilisation in a single implementation phase.

Our work involving AHIR is similar to these efforts, since the goal is to transparently compile software programs into hardware. AHIR differs from both Pegasus and SPARK in the fact that a circuit specification in AHIR is factorised into three separate components: control flow, data flow and memory. This factorisation is the key to a compiler flow that can scale to very large systems. The components can be optimised and implemented separately as long as specified constraints are satisfied.

An AHIR description always represents a circuit that can be routinely translated to a hardware implementation. The decoupled control and data paths respectively represent operation sequences and operators bound to them. The compiler can use this information to perform delay-insensitive low-level optimisations natively in AHIR itself.

1.3 Organisation of the thesis

In Chapter 2, we present AHIR, the proposed intermediate representation for high-level synthesis. We describe the structure and the execution model of an AHIR specification. We also describe the specific class of petri-nets used in AHIR, called “Type-2 petri-nets”, along with a linear-time algorithm to recognise a Type-2 petri-net.

In Chapter 3, we describe the process of mapping an imperative program to a circuit specification in AHIR using a CDFG as an intermediate step. Each element in the CDFG is translated to a circuit fragment that implements the corresponding behaviour. We prove that the construction is correct by recovering a CDFG from the circuit specification, and demonstrating that it is equivalent to the original CDFG.

In Chapter 4, we demonstrate the ease of analysing an AHIR specification, with an optimisation that reuses operators in the AHIR data-path, using a static analysis of the associated control-path. The complexity of the algorithms and auxiliary representations used is close to linear with respect to the size of the petri-net.

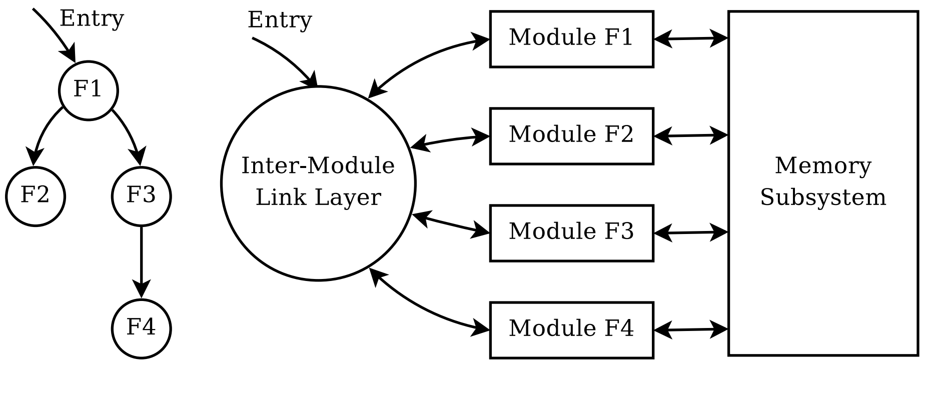
In Chapter 5, we provide an overview of our compiler that translates a C program to a synthesisable VHDL descriptions, using AHIR as an intermediate form. The compiler employs a greedy algorithm to share operators in each data-path, based on the analysis presented in Chapter 4. We use this compiler to generate VHDL descriptions for a number of input programs representing diverse applications. The area occupied and throughput delivered by these circuits are presented at the end of Chapter 5.

Finally, we conclude the thesis in Chapter 6 with a summary of the thesis and a look at directions for improvement as well as future research.

Chapter 2

AHIR

A system in AHIR consists of a number of modules, each consisting of a control-path and a data-path. Typically, each function in an input program is translated to a module. Function calls are implemented by passing the call request along with the relevant arguments through an inter-module link layer. The modules can also share data through an external memory subsystem.



(a) Callgraph (b) Modules

Figure 2.1: Call-graph of a program and the resulting AHIR system.

2.1 A module in AHIR

Two flows are described in an AHIR module — control-path and data-path. The control-path is a petri-net that specifies the ordering of events in the module. The data-path is a pool of hardware resources connected by wires. These two paths communicate with each other via the *intra-module link layer*.

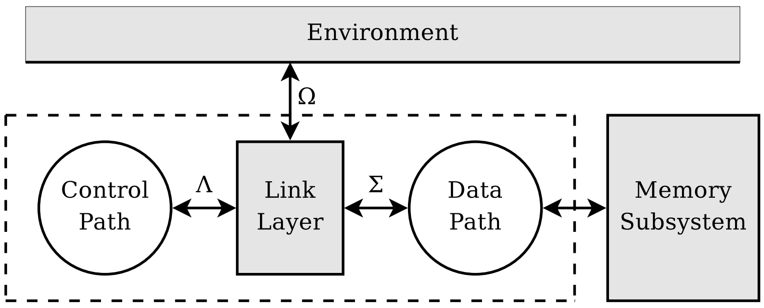


Figure 2.2: An AHIR module.

The *environment* of a module includes all the other components of the system, that are not part of the module. The module interacts with its environment (essentially other modules) in two ways — by passing messages through the inter-module link layer and passing data through the memory subsystem.

Communication using symbols

Communication of control through the link layers is specified in terms of the emission and reception 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 the alphabet Ω.

2.2 Data-path

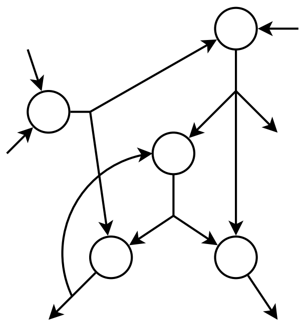


Figure 2.3: An AHIR Data-path.

The data-path is a directed hypergraph (N, E), where E is the set of hyperedges $\{e_0, e_1, \ldots\}$ that represent values flowing in the data-path, N is the set of nodes $\{n_0, n_1, \ldots\}$ that represent operations on these values.

A hyperedge $e_i \in E$ is a tuple (d_i, L_i) where the node $d_i \in N$ is the single *tail* or *driver* of the edge and the nodes in set $L_i \subset N$ are the *heads* or *loads* on the edge. These can be accessed through the functions $\mathrm{driver}(e_i)$ and $\mathrm{loads}(e_i)$ respectively. The value of a hyperedge e_i at time t is given by the function $\mathrm{value}(e_i, t)$. The value is driven by the driver, and the new value reaches all the loads instantaneously.

For a node n, outgoing edges are represented by the set $\mathrm{Out}(n) = \{e_j | e_j \in E \textit{ and } n = \mathrm{driver}(e_j)\}$, while incoming edges are represented by the set $\mathrm{In}(n) = \{e_j | e_j \in E \textit{ and } n \in \mathrm{loads}(e_j)\}$. Each node n declares a set of ports, corresponding to the hyperedges incident on the node. The ports serve as connection points for the hyperedges, and the mapping of ports to data-edges is defined by a bijection $\mathrm{portmap}(n) : \mathrm{ports}(n) \rightarrow \mathrm{In}(n) \cup \mathrm{Out}(n)$. This function allows indirect reference to a data-path edge using the port to which it is connected.

2.2.1 External access

The data-path uses specialised load and store operators to communicate with external memory. When a load or store operator is triggered by a request symbol from the control-path, it presents a request to the memory subsystem. The memory subsystem is expected to service this request *eventually*. When the request is executed, the operator indicates completion by emitting an acknowledge symbol.

Additionally, each data-path has input and output ports connected to the inter-module link layer. An input port is a node that drives a single data-edge, whose value is determined by the inter-module link layer. An output port is a data-path node with a single incoming data-edge incident on it. The value of this edge is made available to the inter-module link layer through the port. Pairs of such input/output ports are used to implement function calls through the inter-module link layer.

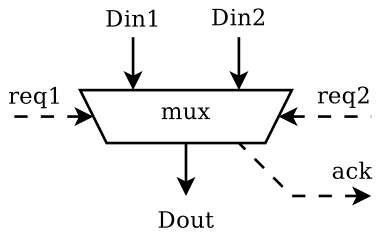
2.2.2 Behaviour of a data-path node

The possible states of a data-path node include a single idle state, and one or more busy states. Events in an idle node are initiated by the arrival of a request symbol and the node changes its state to the corresponding busy state. A busy state is associated with each request symbol that is accepted by the node.

Let $\mathrm{reqs}(n) = \{\mathrm{req}_1, \mathrm{req}_2, \ldots, \mathrm{req}_{p(n)}\}$ be the symbols in Σ accepted at a node n. Then the set of states that the node can take is $S(n) = \{\mathrm{idle}, \mathrm{busy}_1, \mathrm{busy}_, \ldots, \mathrm{busy}_{p(n)}\}$. At a time t, the state of the node is given by the function $\mathrm{state}(n, t) \in S(n)$. When a request symbol req_i arrives at an idle node, the data-path node changes state to the corresponding state busy_i, and samples the input edges. Let E_{sampled} be the set of sampled values $\hat{e}_i$ for each incoming edge e_i. This set is defined in terms of the ports to which the incoming edges are connected: $E_{\mathrm{sampled}}(n) = \{\hat{p}_i | \, \mathrm{portmap}(p_i) \in \mathrm{In}(n)\}$. At a time t, each value in E_{sampled} is the last sampled value of that edge if a sampling had occurred in the past, or else it is undefined.

Computations in a busy node use the sampled values of the relevant edges. On completion, the node updates the relevant outgoing edges and emits an appropriate acknowledge symbol before returning to the idle state. Let $\mathrm{acks}(n) = \{\mathrm{ack}_1, \mathrm{ack}_2, \ldots, \mathrm{ack}_{q(n)}\}$ be the set of all symbols in alphabet Σ that may be emitted by the node n. The relation between request and acknowledge symbols is specific to each node, but the arrival of a request and the eventual emission of an acknowledge together constitute a *handshake* in AHIR.

Example: A multiplexer node



$$\begin{aligned}
\mathrm{ports}(\mathrm{mux}) &= [D_{\mathrm{in}_1}, D_{\mathrm{in}_2}, D_{\mathrm{out}}] \\
\mathrm{In}(\mathrm{mux}) &= \{\mathrm{portmap}(D_{\mathrm{in}_1}), \mathrm{portmap}(D_{\mathrm{in}_2})\} \\
\mathrm{Out}(\mathrm{mux}) &= \{\mathrm{portmap}(D_{\mathrm{out}})\} \\
\mathrm{reqs}(\mathrm{mux}) &= \{\mathrm{req}_1, \mathrm{req}_2\} \\
\mathrm{acks}(\mathrm{mux}) &= \{\mathrm{ack}\} \\
S &= \{\mathrm{idle}, \mathrm{busy}_1, \mathrm{busy}_2\} \\
\mathrm{state}(\mathrm{mux}, 0) &= \mathrm{idle} \\
E_{\mathrm{sampled}}(\mathrm{mux}) &= [\hat{D}_{\mathrm{in}_1}, \hat{D}_{\mathrm{in}_2}]
\end{aligned}$$

Figure 2.4: Data-path node for a multiplexer.

As an example, consider a multiplexer node as shown in Figure 2.4. The node has two incoming data-edges, and reacts to two request symbols — req_1 and req_2. When one of these is received, it forwards the value on the corresponding data-edge to the single outgoing data-edge and then emits the single acknowledge symbol ack. The state table describing the behaviour of a multiplexer node is listed in Table 2.1.

$\mathrm{state}(\mathrm{mux}, t)$	event	$\mathrm{state}(\mathrm{mux}, t^+)$	sequence of actions
idle	req_1 arrives	busy_1	sample incoming edges consume req_1 start computation
idle	req_2 arrives	busy_2	sample incoming edges consume req_2 start computation
idle	req_1 and req_2 arrive	undefined	undefined
busy_1	computation done	idle	$\mathrm{state}(D_{\mathrm{out}}, t^+) := \hat{D}_{\mathrm{in}_1}$ emit ack
busy_2	computation done	idle	$\mathrm{state}(D_{\mathrm{out}}, t^+) := \hat{D}_{\mathrm{in}_2}$ emit ack

Table 2.1: State table for a data-path multiplexer node.

2.3 Control-path

The control-path is a petri-net[14] represented by a four-tuple (P, T, E, M), where P is the set of places, T is the set of transitions and E is a set of directed edges connecting places and transitions. The state of the petri-net is its *marking*, represented by a function M, that maps places to non-negative integers.

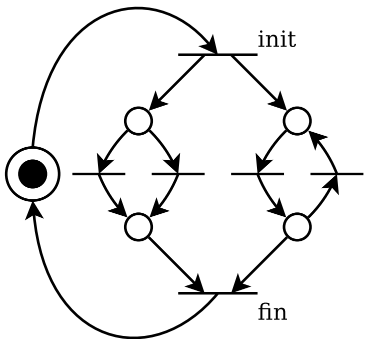


Figure 2.5: An AHIR Control-path.

The control-path interacts with other components using the alphabet Λ. Each transition in the control-path is associated with at most one symbol from Λ. The set of transitions can be partitioned into three disjoint sets, $T = T_I \cup T_O \cup T_H$, where T_I is the set of input transitions, T_O is the set of output transitions and T_H is the set of hidden transitions. An *input* transition is gated on the arrival of the corresponding symbol: an enabled input transition fires only when the symbol arrives. An *output* transition emits the corresponding symbol when it fires. A *hidden* transition is not associated with any symbol.

Let the function $\mathrm{symbol}(t) : T_I \cup T_O \rightarrow \Lambda$ represent the symbol associated with a particular input or output transition. Symbols associated with distinct transitions are distinct, so that $t_1 = t_2 \iff \mathrm{symbol}(t_1) = \mathrm{symbol}(t_2)$. The alphabet Λ can be divided into two disjoint subsets: the set Λ_I of symbols that are consumed by the input transitions and the set Λ_O of symbols that are produced by the output transitions.

The petri-net that specifies the control-path is required to be live and safe. In particular, it must belong to a class of petri-nets called "Type-2 Petri-nets", defined in Section 2.8.2. Every Type-2 petri-net has a single marked place in the initial marking. This place enables a single transition called the init transition, which is an input transition that responds to a symbol received from the environment. The arrival of this symbol indicates the start of execution for the control-path. Similarly, the Type-2 petri-net has a single output transition called fin, that indicates the end of execution of the control-path. This transition marks the same initial place when fired, and emits a symbol expected by the environment.

2.4 The Intra-module Link Layer

The intra-module link layer translates symbols in Λ produced by the control-path, to symbols used by the data-path (Σ) or the inter-module link layer (Ω), and *vice versa*. These translations are represented as the forward function f and the reverse function r.

$$\begin{array}{cccccc} \mathrm{LN} & = & (f, r) & & \\ f & : & \Lambda_O & \rightarrow & \Sigma_I \cup \Omega_I \\ r & : & \Sigma_O \cup \Omega_O & \rightarrow & \Lambda_I \end{array}$$

Table 2.2: The intra-module link layer.

The state of the link layer LN at a time t is composed of the symbols that are currently waiting to be consumed. It is given by the function $\mathrm{state}(LN, t) = (\lambda, \sigma, \omega)$, where $\lambda \subset \Lambda$, $\sigma \subset \Sigma$, and $\omega \subset \Omega$. At the start of execution, no symbols are present in these subsets. When the link layer is executed at some time t, it instantaneously consumes all the waiting symbols and emits the corresponding symbols as defined by the translation functions.

2.5 Symbol Handshakes

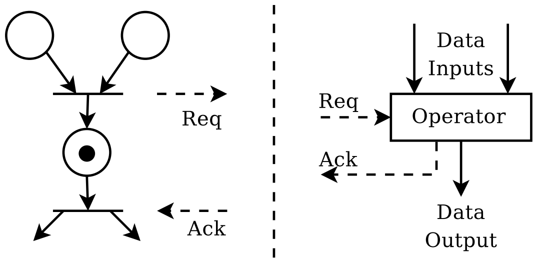


Figure 2.6: A symbol handshake.

Operations in an AHIR module are managed by the exchange of symbol handshakes. Each operator in the data-path is associated with a number of input and output transitions in the control-path. When one of the output transitions fires, it emits a *request* symbol in Λ. This is translated by the intra-module link layer to a symbol in Σ, that triggers the corresponding operator in the data-path. When the operator finishes execution, it emits an *acknowledge* symbol in Σ, that is translated to a symbol in Λ. The arrival of this symbol at the control-path triggers the corresponding input transition indicating completion of the operation.

This request-acknowledge handshake encapsulates any delays in the implementation. Thus, the specification remains independent of hardware details. But for an implementation to be correct, it must still satisfy some constraints as described below. These constraints ensure that the data in the data-path is valid at all times.

2.5.1 Delay Constraints

In Figure 2.7, we show a hypothetical example with associated delays. The data-path consists of a single operator with its output connected back to its input. The control-path is a simple cycle that triggers the data-path operator in an endless loop. The numbered delays d_0 to d_5 are not individual values, but representatives of their respective class of delays.

The arrival of an input symbol Ack at an input transition is an error if the transition is not enabled. Hence the implementation must ensure that the necessary tokens arrive before the symbol arrives, and the transition is activated in time. Hence we have:

$$d_5 \leq d_0 + d_1 + d_3 \tag{2.1}$$

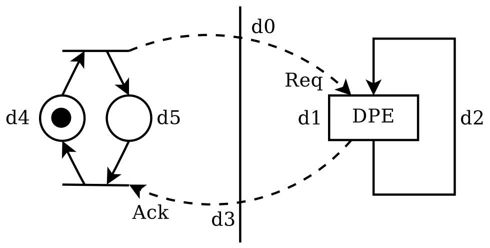


Figure 2.7: Delays in an AHIR specification.

The dual of this fork occurs when a data-path element produces a result and emits the corresponding *acknowledge* 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* arrives. Hence we have:

$$d_2 \leq d_3 + d_4 + d_0 \tag{2.2}$$

Note that the term $d_0 + d_3$ is common to both expressions. It corresponds to the delays in the intra-module link-layer. An implementation can always guarantee timing correctness by sufficiently padding these delays so that the inequalities are satisfied.

In Section A.2.1, we describe a synchronous VHDL implementation of the AHIR specification. The control-path and the link layer in the resulting circuit are asynchronous, while the data-path operators are synchronous. The delays in the above expressions are assigned the following values as multiples of the clock cycle.

$$d_0 = d_2 = d_3 = d_4 = d_5 = 0$$

$$d_1 \geq 1$$

Clearly, these values satisfy the delay constraints specified above, and hence the resulting synchronous circuit is guaranteed to be a correct implementation of the AHIR specification.

2.6 The Inter-module Link Layer

The inter-module link layer is used for communication between modules. For example, when translating C programs to AHIR, function calls are routed through the inter-module link layer. Each data-path is connected to the inter-module link layer through pairs of input and output ports as described in Section 2.2.1.

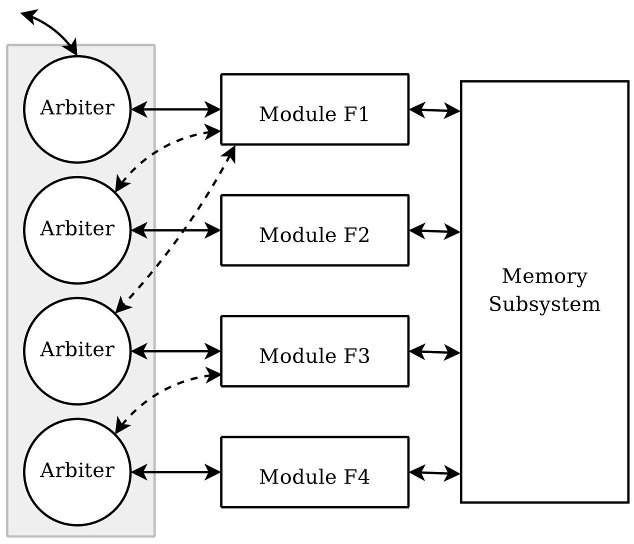


Figure 2.8: Arbiters in the inter-module link layer.

One pair represents the *formal* arguments and return value used when this module is called by another module. The remaining pairs represent the *actual* arguments and return value used when this module calls other modules instead. The module has a separate pair of input/output ports for each call it makes, even if multiple calls are made to the same destination module. The inter-module link layer has a separate arbiter assigned to the *formal* ports of each module. This arbiter manages the transfer of control and data from the *actual* ports of the calling modules to the *formal* ports of the called module.

A function call is initiated when the caller emits the corresponding request symbol. The arguments must be available on the *actual* port of the caller, and these are latched by the arbiter. The arbiter uses a suitable arbitration mechanism to forward this request to the callee. The callee indicates completion by emitting an acknowledge symbol and provides the return value on its formal return port. The arbiter latches this data and forwards it to the caller along with the acknowledge symbol.

2.7 Execution model

AHIR uses a synchronous execution model, where synchronisation is enforced by the data-path. The control-path is a Mealy machine, while the data-path is a Moore machine. All elements in a module execute instantaneously and their results are also instantaneously propagated to the destinations. Only the computations within a data-path node take a finite amount of time to complete. This execution model satisfies the delay constraints since all delays are zero, except the computation delay d_1 which is finite. The execution can be represented as an endless loop, as shown in Table 2.3.

```
For{ever}
  Repeat
    Execute control-paths and link layers
  Until {no input symbols are available for the link-layers
         and no transition in the control-path can fire}
  Execute data-paths
EndFor
```

Table 2.3: Synchronous execution of an AHIR module.

2.8 Petri-nets in the control-path

The control-path of a module in AHIR is described as a petri-net. A petri-net[14] is a directed bipartite graph with two kinds of nodes — transitions and places. A *marking* assigns a non-negative number to each place, indicating the number of tokens in the place.

$$\begin{aligned} PN &= (P, T, E) \\ P &= \text{set of places} \\ T &= \text{set of transitions} \\ E &\subseteq (P \times T) \cup (T \times P) \\ M &: P \rightarrow \mathbb{N}_0 \end{aligned}$$

A petri-net is a powerful representation of control-flow that can be used to represent a very large class of behaviours. We propose a class of petri-nets called *Type-2*, defined in terms of constraints specified on the structure of the petri-net. The constraints make it easy to design analyses that are scalable with the size of the petri-net. At the same time, the class is powerful enough to express useful sequencing concepts such as parallelism, pipelining, etc.

Terminology

A petri-net represents branching and parallelism in terms of the number of edges incident on a transition or a place. We need a consistent manner of referring to such elements that can occur in a *live and safe* petri-net, as shown in Figure 2.9. We define the following terms to refer to these structures:

Simple place: A place with one incoming edge and one outgoing edge.

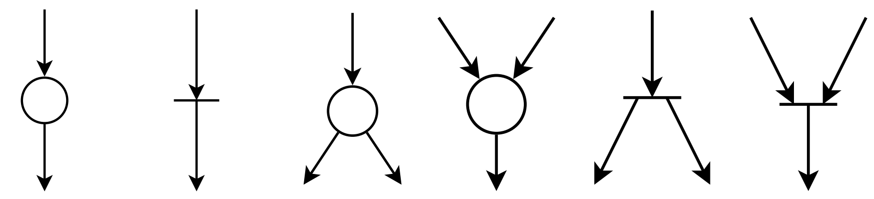
Simple transition: A transition with one incoming edge and one outgoing edge.

Fork: A transition with one incoming edge and multiple outgoing edges.

Join: A transition with multiple incoming edges and one outgoing edge.

Branch: A place with one incoming edge and multiple outgoing edges.

Merge: A place with multiple incoming edges and one outgoing edge.



(a) Simple Place (b) Simple Transition (c) Branch (d) Merge (e) Fork (f) Join

Figure 2.9: Building blocks in a petri-net.

Canonical form

Note that the terms defined above exclude two other structures that can occur in a live and safe petri-net — a transition (or a place) with multiple incoming and outgoing edges. These can be easily replaced by a combination of a fork and join (or a branch and merge) without affecting the behaviour of the petri-net. These replacements do not affect the execution trace in terms of externally visible events, or the safety and liveness properties of the petri-net.

A transition with multiple incoming and outgoing edges: This is replaced with a pair of transitions t_0 and t_1, connected to a simple place p by directed edges (t_0, p) and (p, t_1). All the incoming edges on the original transition are transferred to t_0, while all the outgoing edges are transferred to t_1.

A place with multiple incoming and outgoing edges: This is equivalently replaced with a pair of places connected by a simple transition.

2.8.1 Type-1 petri-nets and Token Preserving Regions (TPRs)

We first define a class of petri-nets called *Type-1*, which forms the basis of defining the *Type-2* class of petri-nets. At the same time, we introduce the notion of a *Token Preserving Region* (TPR), which is instrumental in constructing Type-2 petri-nets.

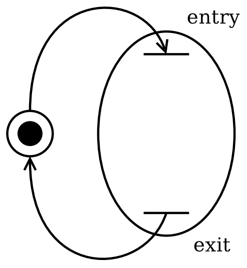


Figure 2.10: A TPR and the corresponding Type-1 petri-net

Definition 2.8.1 *A* **Type-1 Petri-net** *is a live and safe petri-net that marks exactly one simple place in the initial marking.*

Definition 2.8.2 *A* **token-preserving region (TPR)** *is a petri-net P that can be augmented with one simple place $\hat{p}$ and a sufficient number of simple transitions and edges, to produce a live and safe petri-net P' such that $\hat{p}$ is the only marked place in the initial marking.*

Clearly, a Type-1 petri-net is constructed by augmenting a TPR as shown in Figure 2.10. In every TPR, there is one place or transition at which an incoming edge is introduced when completing the corresponding Type-1 petri-net. This place or transition is called the *entry* of the TPR. Similarly, there is one place or transition with an outgoing edge, called the *exit* of the TPR. There are four classes of TPRs as shown in Figure 2.11, differentiated by the nature of their entries and exits.

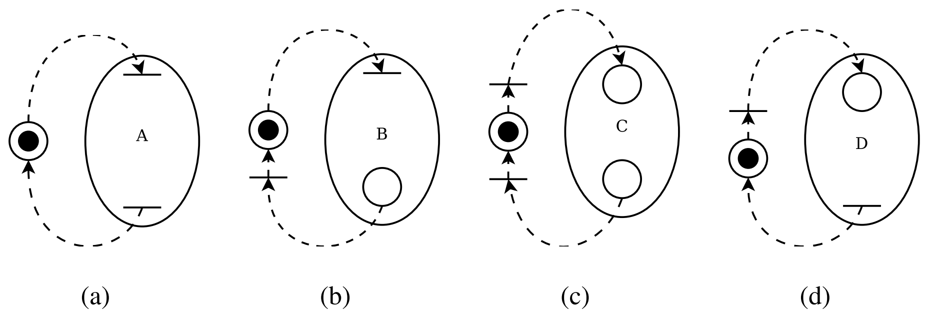


Figure 2.11: Classification of Token Preserving Regions (TPRs).

Class A: A region with a transition at the entry as well as exit.

Class B: A region with a transition at the entry and a place at the exit.

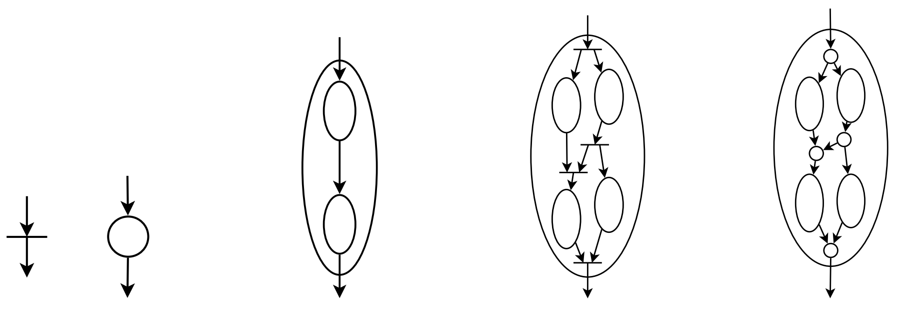
Class C: A region with a place at the entry as well as exit.

Class D: A region with a place at the entry and a transition at the exit.

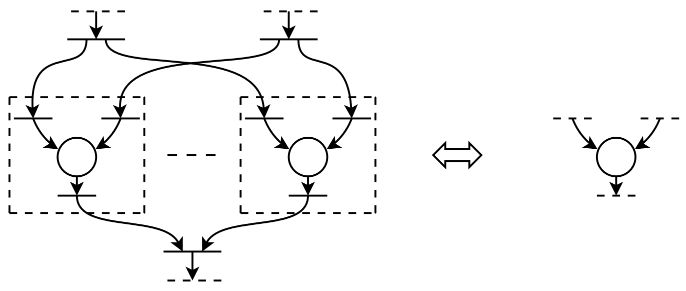
2.8.2 Type-2 Petri-nets and Standard TPRs

The Type-1 class admits a large number of petri-nets. We define a subclass of Type-1 called *Type-2 petri-nets*, which is used to describe a control-path in AHIR. A Type-2 petri-net is a Type-1 petri-net created from a TPR that is itself constructed using a standard set of rules described below. Such a TPR is referred to as a *standard TPR*(STPR).

In Figure 2.12, we show the five different types of standard TPRs that are admitted in a Type-2 petri-net. Note that we choose to omit transitions and places whose existence can be inferred from the context, in order to simplify the representation. This omission is also used in other figures in the rest of this document wherever it does not introduce any ambiguity in the petri-net being represented.



(a) Primitive Regions (b) Series Region (c) Fork Region (d) Branch Region



(e) Parallel merges

Figure 2.12: Type-2 construction rules.

Type-2 construction rules:

1. A simple place or transition is a *primitive* STPR. For convenience, we refer to a primitive STPR as simply "a place" or "a transition" respectively.
2. A *series region* is an STPR formed by joining two standard STPRs in series. Not all combinations are possible since the petri-net is a bipartite graph. The exit of the first region and the entry of the next region should not have the same type. The following table shows the series combinations that are allowed. Each row in the table represents the first region in the series, while each column represents the second region. Each cell shows the class of the resulting region if the corresponding series construction is allowed.

	A	B	C	D
A	A	B	$-$	$-$
B	$-$	$-$	B	A
C	$-$	$-$	C	D
D	D	C	$-$	$-$

3. A connected *acyclic* subgraph made of Class C STPRs, forks and joins is a Class A STPR called a *fork region*.
4. A connected (possibly cyclic) subgraph made of Class A STPRs, branches and merges is a Class C STPR called a *branch region*.
5. Replacing a merge place in a branch region with parallel merges as shown in Figure 2.12(e) also results in a standard TPR. The set of parallel merges introduced by this replacement is called a *parallel-merge region*, described in Section 2.8.3. The elements introduced by this replacement cannot be used for further application of the standard construction rules, as explained in that section.

The construction of a branch region allows cycles so that it can express arbitrary branch as well as loop structures. The fork region expresses parallelism, but its construction does not allow cycles. If a cycle is present in a fork region, then every back-edge in the region must have a marked place in the initial marking for the petri-net to be live. This increases the complexity of analysing the petri-net. Hence we choose to disallow cycles in the fork region. This restriction does not significantly reduce the expressive power of Type-2 petri-nets. For example, a pipeline can be expressed in a Type-2 petri-net as a fork region inside a loop as shown in Figure 2.13.

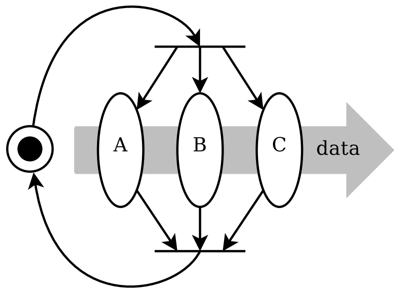


Figure 2.13: Synchronous pipeline in a Type-2 petri-net.

Type-2 replacements

The simplest Type-2 petri-net is a cycle that consists of one marked simple place p and one simple transition t. The STPR in this petri-net is a primitive transition region that consists of the transition t. Replacing this transition with a Class A STPR results in a larger Type-2 petri-net. This process can be continued further to construct large and complex Type-2 petri-nets.

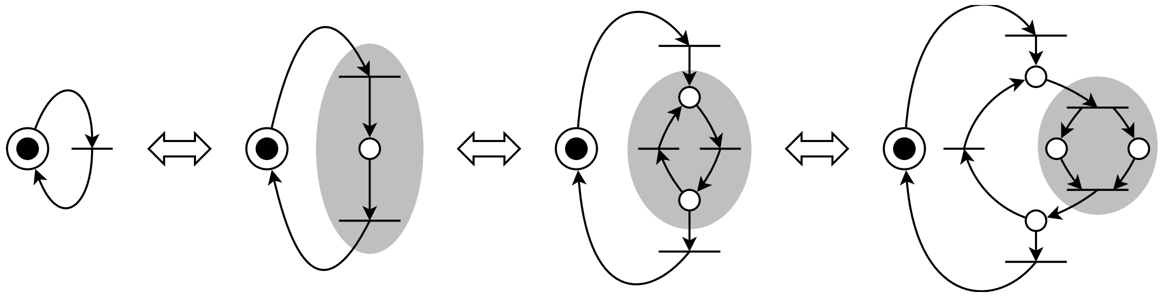


Figure 2.14: A sequence of Type-2 replacements.

In Figure 2.14, we show the construction of a Type-2 petri-net through a series of replacements starting with a simple cycle. In general, if P is a Type-2 petri-net, then replacing a simple transition in P with a Class A standard TPR results in another Type-2 petri-net. Similarly, replacing a simple place in P with a Class C standard TPR also results in a Type-2 petri-net.

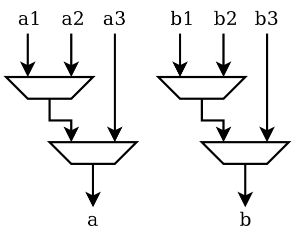
2.8.3 Parallel-merge region

The parallel-merge region is a special construct required to implement the flow of values at the exit of a branch. When a branch occurs in a program, a variable may be assigned different values along each path in the branch. The actual value that is applicable at this variable beyond the exit of the branch depends on the path chosen at run-time.

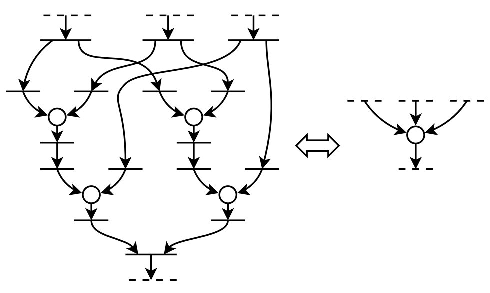
In Figure 2.15(a), we show a code snippet where this run-time selection of values is represented as a special operator called the ϕ-function. This operator is used in the Static Single Assignment (SSA) form, described further in Section 3.1. In AHIR, this corresponds to a tree of multiplexers in the data-path and a tree of merge places in the control-path as shown in Figure 2.15.

```
if ()
  a1 = ...; b1 = ...
else if ()
  a2 = ...; b2 = ...
else
  a3 = ...; b3 = ...
a = ϕ(a1, a2, a3);
b = ϕ(b1, b2, b3);
```

(a) Code snippet.



(b) Data-path



(c) Control-path

Figure 2.15: Multiple ϕ-functions and the parallel-merge region.

The dotted transitions at the top of the control-path fragment represent exits from the three paths of the branch in the code. A token will arrive from only one of the three dotted transitions, so that only one fork is activated. That fork sends one token to each merge tree, effectively executing the two ϕ-functions in parallel. The two tokens exit the merge trees at the join, which marks the end of the merge operation. In the absence of ϕ-functions, this entire subgraph of the control path is equivalent to a single merge place with three input edges.

We define the parallel-merge region as follows:

1. A *merge fragment* is an acyclic subgraph made of a single place with two input edges and one output edge, with a simple transition connected to each edge.

2. An m*-way merge tree* is a connected acyclic subgraph made of $m - 1$ merge fragments. The output edge of every merge fragment except one is connected to the input edge of some other merge fragment through a simple place. The subgraph has m input edges and a single output edge. A 2-way merge tree is a merge fragment.

3. A $k \times m$ *parallel-merge region* is a connected acyclic subgraph made of the following components:
   (a) k m-way merge trees
   (b) m forks with fan-out k, such that each output edge of a fork is connected to an input edge of a distinct merge tree through a simple place
   (c) a single join with fan-in k, such that each input edge is connected to the single output edge of a distinct merge tree through a simple place

The control-path in Figure 2.15 represents a 2×3 parallel-merge region made of two 3-way merge trees. The entire region is equivalent to a merge place with three incoming edges and a single outgoing edge.

Restrictions on the parallel-merge region

The parallel-merge region represents the run-time selection of values in the data-path at the exit of a branch. It is an atomic operation whose intermediate state is not relevant to the rest of the program. Hence, the elements in the parallel-merge regions are not used for further replacements in the construction of a Type-2 petri-net. As a convenience, any general reference to a "Type-2 petri-net" in the rest of this document actually means a Type-2 petri-net in which all parallel-merge regions have been replaced with equivalent merge places.

Relaxation of the construction rules

The construction rules for Type-2 petri-nets do not allow the replacement of an element within the parallel-merge region. It is possible to define a larger class of petri-nets by relaxing this restriction. The resulting petri-net is also live and safe, since all the replacements defined in Type-2 petri-nets preserve both liveness and safety. Note that three other replacements are possible, that are structurally similar to the parallel-merge region — parallel-branch, parallel-fork and parallel-join.

But the strict class of Type-2 petri-nets is itself sufficient for supporting a large class of programming languages. The structure of a strict Type-2 petri-net is also easy to analyse, as seen in Chapter 4. The larger class defined by relaxing the construction rules will require new analyses that are applicable to that class.

2.8.4 Hierarchical representation of a standard TPR

An STPR is constructed by the recursive application of the construction rules described in Section 2.8.2. If R is an STPR, then it is made of a set of smaller STPRs and so on, ending in primitive regions. This leads to a hierarchy that represents how an intermediate STPR is made from smaller STPRs. Note that this hierarchy is only a representation although the Type-2 petri-net itself is flat. The hierarchical representation is useful for analysing the structure of the Type-2 petri-net, as described in later sections.

Let S be the set of all the regions that correspond to intermediate steps in the construction of R. If a region $r \in S$ is constructed from a set of regions $T \subseteq S$, then every region $q \in T$ is said to be a *child* of r, written as $q \sqsubset r$, and r is said to be the *parent* of q. Every region $r \in S$ is used in exactly one construction step, unless $r = R$, which is not used in any construction step. Hence, every region $r \in S$ has at most one parent. The transitive closure of the child relation is the *descendant* relation $(\sqsubset^*)$. If $q \sqsubset^* r$, then q is a descendant of r, while r is called an *ancestor* of q. Note that the child and descendant relations are not reflexive.

The child relation $(\sqsubset)$ can be represented as a single-rooted directed tree where nodes represent regions and edges represent the child relation. A directed edge (u, v) exists in the tree if and only if $v \sqsubset u$. For a set $T \subseteq S$, the root of the smallest subtree T' that contains all the regions in T is called the **nearest common ancestor** of T. It represents the smallest region that contains all the regions in T. The NCA is not defined if the root of T' is in T, since a region cannot be its own ancestor.

2.8.5 Algorithm to identify a Type-2 petri-net

We identify a Type-2 petri-net P by checking whether it was created by augmenting an STPR. By definition, the initial marking of the petri-net must mark a single simple place p_0. The petri-net $T_c = P - \{p_0\}$ represents a candidate STPR with a single entry and exit point.

First, we replace all the *parallel merge* regions in T_c by simple merge places. The corresponding reduced petri-net is termed $T_c{}'$. Clearly, T_c is an STPR if and only if $T_c{}'$ is an STPR. We construct the following directed graph $G = (N, E)$ for the candidate STPR $T_c{}'$:

1. The set of nodes N in the directed graph consists of all the places and transitions in $T_c{}'$. A node $u \in V$ is said to be of a *place-type* (respectively *transition-type*) if it corresponds to a place (respectively transition) in $T_c{}'$.

2. For a place p and a transition q in $T_c{}'$, there is an edge from p to q in E, if q is a successor of p in $T_c{}'$. Similarly, there is an edge from q to p in E, if p is a successor of q in $T_c{}'$.

Now, the graph G is subjected to the following reduction algorithm:

1. If there is a subgraph of G which is a directed simple path from u to v such that the only connections to the path are at u and v, then the path is replaced by a simple edge from u to v (and all other edges and nodes in the path are deleted).

2. If there is a connected subgraph of G induced by *place-type* nodes such that the connected subgraph has a unique entry point u and a unique exit point v, then this subgraph is replaced by a single edge between u and v (and all other nodes and edges in the subgraph are deleted).

3. If there is a connected *acyclic* subgraph of G induced by *transition-type* nodes such that the connected subgraph has a unique entry point u and a unique exit point v, then this subgraph is replaced by a single edge between u and v (and all other nodes and edges in the subgraph are deleted).

These reductions are repeated until no further reduction is possible. If the remaining graph at the end of these reductions is a single edge, then the candidate region $T_c{}'$ is in fact an STPR.

This algorithm demonstrates how the hierarchy in a Type-2 petri-net can be exploited by a divide-and-conquer strategy when analysing it. The algorithm verifies the petri-net in a bottom-up manner by first verifying the smallest regions and then moving upwards to larger regions composed from them. The reductions in each iteration have equivalent steps in the construction of the original Type-2 petri-net.

2.9 Summary

In this chapter, we introduced AHIR as an intermediate representation for hardware circuits. The representation factorises the circuit into three components: control, data and storage.

The data-path is a simple pool of hardware resources, that consists of operators with well-defined behaviours. The control-path is a petri-net whose structure is restricted to a class of petri-nets that we introduce as the Type-2 petri-net. The structure of the Type-2 is defined by a small set of construction rules. This structure ensures that the control-path is easy to analyse but also expressive enough to specify complex behaviours.

The ease of analysing a Type-2 petri-net is evident from the algorithm we use for checking whether a given petri-net is Type-2. This simplicity is further demonstrated by the static analysis that we introduce in Chapter 4.

Chapter 3

The Compilation Process

Our approach to high-level synthesis introduces a well-defined intermediate step in the compilation of software programs to hardware circuits. We implement this intermediate step in the form of the intermediate representation called AHIR. We demonstrate the practicality of this approach with a compiler flow that translates a C program to a hardware circuit using AHIR.

The choice of C as the first language to be supported on our high-level synthesis flow is based on two important facts about C. Firstly, C represents the most common set of features that are expected by the user of an imperative language. And secondly, C is a very simple language, with no built-in syntax for special features such as parallelism, array manipulation, streams, etc. In this respect, it represents the worst starting point for hardware design, while covering the most common programming features expected by the users. If AHIR works well as an intermediate step from C to hardware, then it is likely to work better for "better" languages.

The compiler flow uses the LLVM compiler framework to translate the input C program to a circuit specification in AHIR. The use of an existing compiler framework allows us to save the effort of building a language front-end to parse the input source code. It also allows us to leverage all the optimisation tools that are already available in the framework.

The conversion from C to AHIR is a sequence of steps, with a different representation at each step. The input C program is first converted internally by the LLVM framework to LLVM bytecode. The LLVM bytecode is an implementation of the SSA form used in software compilers, described in Section 3.1. The LLVM bytecode is then converted to a CDFG, described in Section 3.2. Finally, the CDFG is factorised into the control and data paths of an AHIR specification as described in Section 3.4. In Section 1.1.2, we show that our method of implementing a given CDFG as an AHIR specification always produces a correct implementation.

3.1 Static Single Assignment (SSA)

```
                  d1 = m + n;
d = m + n;        b = m - n;
b = m - n;        if (b > 0)
if (b > 0)        {
{                   a = b + c;
  a = b + c;        d2 = e + a;
  d = e + a;      }
}                 d3 = ϕ(d1,d2);
x = d + 2;        x = d3 + 2;
```

(a) C code. (b) SSA form.

Figure 3.1: A code fragment and its SSA form.

The LLVM internal representation is based on the SSA form[15]. The SSA form is a purely functional representation, that removes the notion of individual *variables* from a program. Every statement that assigns a value to a variable is represented by an instruction that defines a unique *version* of that variable instead.

The SSA is convenient for symbolic analysis of programs and provides the right information for generating a hardware data-path. Since variable names are no longer important, there is no explicit location or register for a value. Instead, the value is always available at the output of the operator that created it.

In Figure 3.1, we show a code fragment in C along with its SSA version. The value of the variable d after exiting the `if`-statement cannot be known at compile time. d may be assigned one of two possible values depending on the value of the condition $(b > 0)$ This run-time information is captured by a special operator called the ϕ-function in SSA.

3.1.1 The LLVM internal representation

The LLVM framework uses an internal representation based on a combination of control-flow graphs and the SSA form. A program in LLVM is a collection of basic blocks as shown in Figure 3.2(b). A basic block[15] is a maximal sequence of instructions that is not interrupted by control-flow (branch or merge). The basic block is never empty. Every basic block must end with a *terminator* instruction, which belongs to one of two kinds: a *branch* instruction that points to other blocks, or a *return* instruction that exits from the function.

```
d1 = m + n;
b = m - n;
if (b > 0)
{
  a = b + c;
  d2 = e + a;
}
d3 = φ(d1,d2);
x = d3 + 2;
```

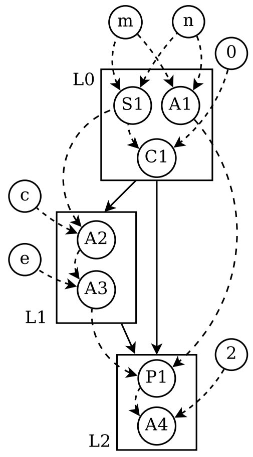
(a) SSA form.

```
L0:
  A1 = add m, n
  S1 = sub m, n
  C1 = cmpgt S1, 0
       br C1, L1, L2

L1:
  A2 = add S1, c
  A3 = add e, A2
       br L2

L2:
  P1 = φ((A1,L0),(A3,L1))
  A4 = add P1, 2
```

(b) LLVM Instructions.



(c) CFG + DFG

Figure 3.2: The LLVM internal representation.

The entire program is a control-flow graph (CFG). A CFG is a directed graph where the nodes are basic blocks and edges represent the control flow specified by the branches. In Figure 3.2(c), we represent the CFG nodes as boxes and control flow as solid edges.

The control-flow graph is super-imposed on a data flow graph that represents the def-use chains in the program[15, 16]. The data flow graph is a directed graph where the nodes represent instructions and edges represent values. The tail of an edge corresponds to the instruction that defines the value, and the head represents the instruction that use that value for further computations. In Figure 3.2(c), we represent these nodes as circles and data flow as dotted edges respectively.

3.2 The CDFG representation

When generating an AHIR specification, we first convert the LLVM IR into a control data flow graph (CDFG) as shown in Figure 3.2. The CDFG has been frequently used as an intermediate representation for high-level synthesis in various forms [17]-[21]. A generalisation of the usual notion of a CDFG is a hypergraph, where a hyperedge connects the definition of a value with all its uses. The hyperedge can be replaced by a set of equivalent simple edges, and hence its use does not alter the meaning of the CDFG. The hyperedge can be used as a convenient representation of a wire, which simplifies the translation of the CDFG into an AHIR specification.

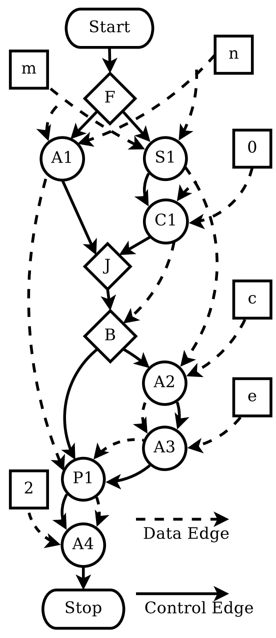


Figure 3.3: A control data flow graph.

We use the following definition of the CDFG when translating the LLVM IR to AHIR.

The CDFG is a connected directed hypergraph $G = (N, E)$. N is the set of nodes in the graph, while E is the set of edges. Edges represent values flowing through the CDFG, while nodes represent operations on the values of incident edges. Each edge e_i is a tuple (u, s_v) where $u \in N$ is a single tail and $s_v \subset N$ is a set of heads. The members of the edge $e = (u, s_v)$ are accessed by the functions $\mathrm{tail}(e) = u$ and $\mathrm{heads}(e) = s_v$. The set E is partitioned into two disjoint subsets: the control edges E_C and the data edges E_D.

Every CDFG must have two special nodes: start and stop. The start node is the only node in the CDFG that has no incoming edges incident on it, while the stop is the only node with no outgoing edges incident on it.

3.2.1 Control edges

A control-edge $e_i^c \in E_C$ has a single head. At any instant of time t, a control edge can have one of two states: $\mathrm{enabled}$ or $\mathrm{disabled}$. The state of the edge is accessed through the function $\mathrm{state} : E_C \times T \rightarrow \{\mathrm{enabled}, \mathrm{disabled}\}$.

The initial state of a control edge is $\mathrm{disabled}$. During execution, the tail may set the state of a control-edge to $\mathrm{enabled}$ and the head may set it to $\mathrm{disabled}$. The new value is available at both ends of the control edge instantaneously. While a control edge is $\mathrm{enabled}$, the head is expected to fire and set the edge to $\mathrm{disabled}$ before the tail may fire again. If the CDFG is such that both the nodes on a control-edge are ready to fire at some instant of time, the result is undefined — the structure of the CDFG must ensure that such a state is not reachable.

3.2.2 Data edges

A data-edge $e_j^d \in E_D$ is a directed hyperedge, with one tail called the driver and multiple heads called the loads. At any instant of time t, the state of a data-edge is the value being driven on it. Its type is defined as the set of values that can be assigned to it, such as int, float, bool, etc. The state of a data-edge $e \in E_D$ is accessed through the function $\mathrm{state}(e, t) \in \mathrm{type}(e)$.

Only the driver can modify the value of a data-edge, and the new value reaches all the loads instantaneously. The initial state of any data-edge is undefined; the structure of the CDFG must ensure that the value of data-edge is not accessed before it is defined.

3.2.3 Nodes

A node represents operations performed on the values of the input and output edges incident on it. For a node n, the edges for which it is the tail are called output edges, represented by the set $\mathrm{Out}(n) = \{e_j | e_j \in E \textit{ and } n = \mathrm{tail}(e_j)\}$. The edges for which the node is a head are called input edges, represented by the set $\mathrm{In}(n) = \{e_j | e_j \in E \textit{ and } n \in \mathrm{heads}(e_j)\}$.

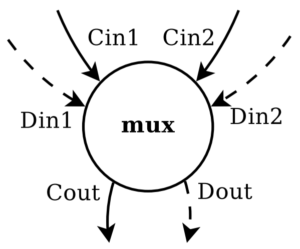
Each node n declares a set of ports corresponding to the edges incident on the node. The ports serve as connection points for the edges, and the mapping of ports to edges is defined by a bijection $\mathrm{portmap}(n) : \mathrm{ports}(n) \rightarrow \mathrm{In}(n) \cup \mathrm{Out}(n)$. The node also declares a set $E_{\mathrm{sampled}}(n) = \{\hat{p}_i | \, \mathrm{portmap}(p_i) \in \mathrm{In}(n)\}$, that contains the sampled value $\hat{p}_i$ for each port p_i associated with an incoming edge.

The state of a node n at a time instant t is given by the function $state : N \times T \rightarrow S$ where $S = \{\mathrm{idle}, \mathrm{busy}_1, \mathrm{busy}_2, \ldots, \mathrm{busy}_m\}$. The initial state of a CDFG node is idle, except the start node, whose initial state is busy. An idle node *fires* when sufficient incoming control edges are enabled to trigger some transition from idle to a busy state. The node first samples the incoming data edges, disables the incoming control edges and then begins computation using the sampled values. On completion, it updates the outgoing edges and returns to the idle state.

Atomic execution The firing of a CDFG node starts a sequence of events that ends with the node returning to the idle state. The response of the node to external events during this time is undefined. The exact trace of events is invisible to the external world; the only observable effects are the changes in the values of the outgoing edges when the node becomes idle again. In this respect, the execution of the CDFG node is said to be *atomic*, and the corresponding execution trace can be considered a single event in the CDFG itself.

3.2.4 The multiplexer node as an example

The behaviour of each node is specific to the operation that it performs on its edges. Typical CDFG nodes are forks, joins, operators, constants, branches, multiplexers, etc. We describe the multiplexer node as an example; other nodes can be described by similar state machines.



$$\begin{aligned}
\mathrm{In}(\mathrm{mux}) &= [C_{\mathrm{in}_1}, C_{\mathrm{in}_2}, D_{\mathrm{in}_1}, D_{\mathrm{in}_2}] \\
\mathrm{Out}(\mathrm{mux}) &= [C_{\mathrm{out}}, D_{\mathrm{out}}] \\
S &= \{\mathrm{idle}, \mathrm{busy}_1, \mathrm{busy}_2\} \\
\mathrm{state}(\mathrm{mux}, 0) &= \mathrm{idle} \\
E_{\mathrm{sampled}} &= [\hat{C}_{\mathrm{in}_1}, \hat{C}_{\mathrm{in}_2}, \hat{D}_{\mathrm{in}_1}, \hat{D}_{\mathrm{in}_2}]
\end{aligned}$$

Figure 3.4: CDFG node for a multiplexer.

A multiplexer node (commonly known as a "mux") has two incoming control-edges, and two incoming data-edges associated with them. When control reaches the mux along one control-edge, the value on the corresponding data-edge is forwarded to the outgoing data-edge. This behaviour is listed in Table 3.1.

3.2.5 `start` and `stop` nodes

Every CDFG has exactly one start node and one stop node. Execution of the CDFG begins by firing the start node and control is passed on to other nodes in the graph. Execution is said to have stopped when control reaches the stop node.

The state table for a start node is listed in Table 3.2. It has only one transition — from the state busy to the state idle. The initial state for the start node is busy, so that it fires once

$\text{state}(\text{mux}, t)$	condition	$\text{state}(\text{mux}, t^+)$	sequence of actions
idle	$\text{state}(C_{\text{in}_1}, t) = \text{enabled}$ and $\text{state}(C_{\text{in}_2}, t) = \text{disabled}$	busy_1	sample incoming edges $\text{state}(C_{\text{in}_1}, t^+) := \text{disabled}$ start computation
idle	$\text{state}(C_{\text{in}_1}, t) = \text{disabled}$ and $\text{state}(C_{\text{in}_2}, t) = \text{enabled}$	busy_2	sample incoming edges $\text{state}(C_{\text{in}_2}, t^+) := \text{disabled}$ start computation
idle	$\text{state}(C_{\text{in}_1}, t) = \text{enabled}$ and $\text{state}(C_{\text{in}_2}, t) = \text{enabled}$	undefined	undefined
busy_1	computation done	idle	$\text{state}(D_{\text{out}}, t^+) := \hat{D}_{\text{in}_1}$ $\text{state}(C_{\text{out}}, t^+) := \text{enabled}$
busy_2	computation done	idle	$\text{state}(D_{\text{out}}, t^+) := \hat{D}_{\text{in}_2}$ $\text{state}(C_{\text{out}}, t^+) := \text{enabled}$

Table 3.1: State table for the CDFG multiplexer node.

when execution begins. The effect of this firing is to enable the single outgoing control-edge. The node cannot fire again, since there is no transition defined from idle to the busy state.

$$\begin{aligned}
\text{In}(\text{start}) &= \{\} \\
\text{Out}(\text{start}) &= \{C_{\text{out}}\} \\
S_{\text{start}} &= \{\text{idle}, \text{busy}\} \\
\text{state}(\text{start}, 0) &= \text{busy}
\end{aligned}$$

$\text{state}(\text{start}, t)$	condition	$\text{state}(\text{start}, t^+)$	sequence of actions
busy	true	idle	$\text{state}(C_{\text{out}}, t^+) := \text{enabled}$

Table 3.2: State table for the start node.

The state table for the stop node is listed in Table 3.3, which is a dual of the start node. There is no transition defined from busy to idle. Hence once the stop node fires, it never returns to the idle state.

$$\begin{aligned}
\text{In(stop)} &= \{C_{\text{in}}\} \\
\text{Out(stop)} &= \{\} \\
S_{\text{stop}} &= \{\text{idle}, \text{busy}\} \\
\text{state}(\text{stop}, 0) &= \{\text{idle}\}
\end{aligned}$$

$\text{state}(\text{stop}, t)$	condition	$\text{state}(\text{stop}, t^+)$	sequence of actions
idle	$\text{state}(C_{in}, t) = \text{enabled}$	busy	$\text{state}(C_{in}, t^+) := \text{disabled}$

Table 3.3: State table for the stop node.

3.3 Translating the LLVM IR to a CDFG

The creation of a CDFG from the internal representation of a software compiler is a routine procedure. Since an exhaustive description of the process is not practical, we summarise the important steps in generating a CDFG from the LLVM IR. The basic building block of the LLVM IR is the basic block. Each basic block is flattened to create a fragment of the CDFG as shown in Figure 3.5. A node in the CDFG fragment represents an instruction in the LLVM basic block. Control and data edges are created between CDFG nodes that enforce the dependences between instructions in the LLVM IR. These edges may join nodes that represent instructions within the same basic block, or in different basic block. The result is a complete CDFG as shown in Figure 3.6.

3.3.1 CDFG Nodes

Each instruction in the LLVM IR is translated to a node in the CDFG with equivalent behaviour. An example is the multiplexer node described in Section 3.2.4. Most nodes in the CDFG are simple operations that operate on the value of incoming edges to generate new values on outgoing edges. Two kinds of nodes have side-effects outside the CDFG — memory load/store operators and function calls. Load/store operators interact with external memory when invoked using address and data values available on incident data edges. The function call node receives arguments along incoming data edges, which are forwarded to the called CDFG. The return value if any is available on an outgoing data edge.

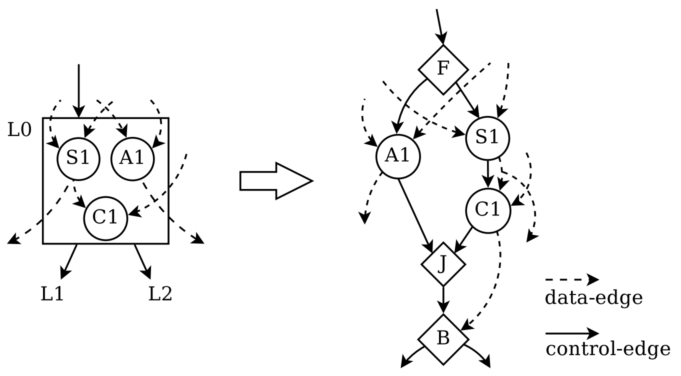


Figure 3.5: Translating an LLVM basic block to a CDFG fragment.

3.3.2 Data edges

The data edges in the CDFG represent the flow of values from nodes that compute them to nodes that use them. Consider an instruction I in the program that defines a value used by a set of instructions $S_I = \{I_0, I_1, \ldots\}$ in the program. This is replaced by a data-flow edge (u, s_v) in the CDFG, where u is the CDFG node for instruction I while $s_v = \{v_0, v_1, \ldots\}$ is the set of CDFG nodes that correspond to instructions in the set S_I. The heads of the data edge may correspond to instructions in the same basic block or other blocks. For example, in Figure 3.5, the dotted edge starting from node S_1 has two heads — one is node C_1 which is in the same fragment, and another is a node in some other fragment.

3.3.3 Control edges

The control edges in the CDFG arise from three components:

1. The data dependence DAG within a basic block.
2. The external dependences between operations within a basic block.
3. Control flow across basic blocks.

Control flow due to data dependences

The instructions in a basic block can be represented as a DAG where the nodes represent instructions and edges represent data dependences. When control reaches the basic block, the order of execution is specified by these data dependence edges. An instruction can execute only when its data dependences are satisfied, i.e., as soon as its predecessors in the DAG have executed. This data dependence is enforced by control-edges between the corresponding nodes in the CDFG. For example, in Figure 3.5, the solid edge from node S_1 to C_1 is a control-edge arising from such a data dependence.

An instruction which has no predecessors in the DAG — a root of the DAG — can execute as soon as the basic block receives control. The entry of control into a basic block is represented by an *entry fork* in the CDFG. The execution of the DAG is triggered by control-edges from this entry to each root. Similarly, an *exit join* represents the completion of execution of the entire DAG, with a control-edge from each leaf of the DAG to the join. For example, in Figure 3.5, F and J are the entry fork and exit join respectively for the block L_0.

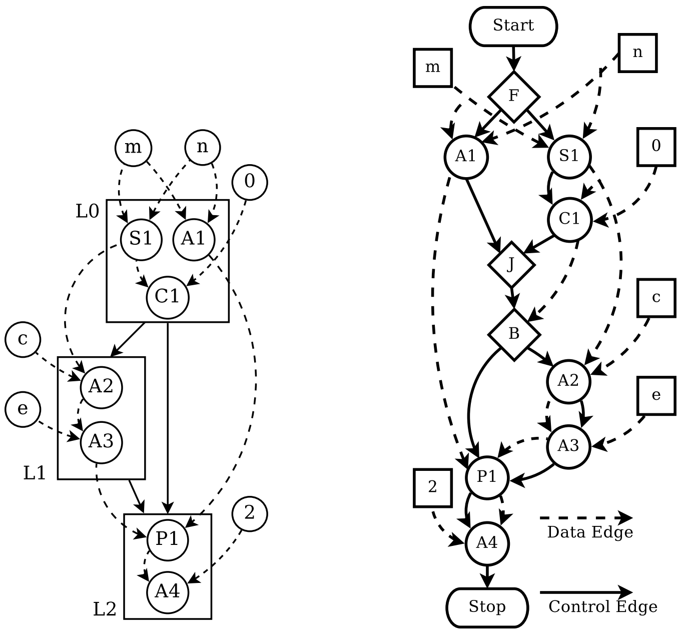
It is possible that an instruction I in the LLVM IR has multiple predecessors within the DAG. The corresponding CDFG node n must be executed only when the nodes for all the predecessors have executed. This is represented as a join node preceding the CDFG node. Control-edges from the predecessor nodes are connected to this join node instead of n itself. Similarly, a fork node is introduced if an instruction has multiple successors in the DAG.

Control flow due to external dependences

An additional data dependence is introduced by the occurrence of memory access operators. Ideally, two memory operators can be executed in parallel if they do not access the same memory location. But a memory reference analysis of the input program is required to determine this relation between two memory accesses. The current implementation takes the most conservative approach in this respect. In the absence of a memory reference analysis, the sequence in which memory operations occur within a basic block is preserved. This is represented by additional control-edges between memory access operators. This implies a restriction that only one memory operation may be active at any time. An improved implementation may use reference analysis in the future to parallelise memory accesses where possible.

Control flow across basic blocks

Control flow in the LLVM IR is described by terminator instructions that occur at the end of basic blocks. An unconditional branch instruction in a block BB represents control flow from BB to the destination block mentioned in the branch. Other instructions such as conditional branch, switch-case, etc. can result in control flow from BB to multiple destination blocks. Only one of these blocks receives control at runtime, depending on the associated condition.



(a) CFG + DFG (b) CDFG

Figure 3.6: LLVM IR and the corresponding CDFG.

For example, in Figure 3.6(a), the conditional branch C_1 at the exit of block L_0 specifies control flow from block L_0 to blocks L_1 and L_2, depending on the condition evaluated at C_1.

The control-flow in the LLVM IR is translated to control-edges between the entry and exit nodes in the CDFG fragments. In case of a conditional branch at the end of a basic block, the exit join of the corresponding fragment is followed by a branch node which represents the actual exit of the fragment. For example, in Figure 3.6(b), the node B represents the exit of the block L_0. Note that the DAGs within blocks L_1 and L_2 do not have multiple roots or leaves, and hence their entry fork and exit join respectively are optimised away. Instead, A_2 and P_1 represent the entries of fragments for L_1 and L_2. Control flow from L_0 to L_1 and L_2 is implemented as control edges from B to A_2 and P_1.

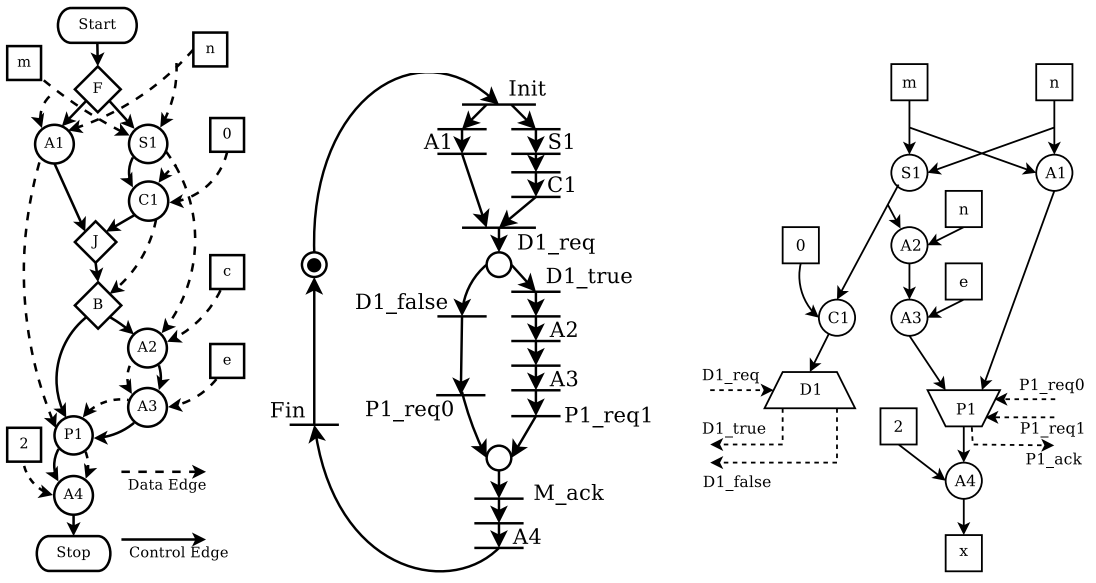
3.3.4 *start* and *stop* nodes

Every function in the LLVM representation has one basic block that does not have any incoming control-flow edge. This represents the entry of the function body itself. We create a start node for the CDFG and connect it to the entry fork for this basic block. Similarly, for the block that represents the return from the function, we connect the exit join by a control-edge to a stop node. This completes the construction, to provide a CDFG version of the input program.

3.4 Generating AHIR from a CDFG

AHIR generation proceeds by translating every element in a CDFG to *AHIR fragments* that implement the same behaviour as the CDFG element. The complete description is obtained by joining all the AHIR fragments together according to the connections between the corresponding CDFG elements.

In Figure 3.7, we show the CDFG obtained from our example along with control and data-paths in AHIR that implement the CDFG. A number of details have been omitted from the figure, in order to keep it readable. The symbols for the control-path transitions and data-path operators are not shown. Every edge in the control-path that joins two transitions is actually a place with edges connecting it to the transitions.



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

Figure 3.7: Translating a CDFG to AHIR.

The data-path uses an element called a *decoder* that implements a conditional branch. Since the control-path cannot have direct access to values in the data-path, the decoder element $D1$ is used to examine the value of the condition $C1$. When it receives a request from the control-path, it emits one of two symbols depending on the boolean value produced by $C1$. This symbol selects the correct branch in the control-path. This is a dual of the multiplexer node M in the data-path, which chooses a value based on the state of the control-path.

$$\begin{array}{rcl}
\Delta & = & (\mathrm{CP}, \mathrm{DP}, \mathrm{LN}, \Lambda, \Sigma) \\
\mathrm{CP} & = & (P, T, E, M) \\
T & = & \{t_1, t_2, ...\} \text{ ... transitions} \\
P & = & \{p_1, p_2, ...\} \text{ ... places} \\
E & = & \{e_1, e_2, ...\} \text{ ... petri-net edges} \\
M & : & P \rightarrow \{0, 1\} \\
\mathrm{DP} & = & \{N, W\} \\
N & = & \{d_1, d_2, ...\} \text{ ... data-path elements} \\
W & = & \{w_1, w_2, ...\} \text{ ... data-path edges within the fragment} \\
\mathrm{LN} & = & (f, r) \\
f & : & \Lambda \rightarrow \Sigma \\
r & : & \Sigma \rightarrow \Lambda \\
\Lambda & = & \{\lambda_1, \lambda_2, ...\} \\
\Sigma & = & \{\sigma_1, \sigma_1, ...\}
\end{array}$$

Figure 3.8: An AHIR fragment.

3.4.1 An AHIR fragment

An AHIR fragment is the set of all the AHIR elements that together implement the behaviour of a particular CDFG element. We define a distinct fragment for every kind of a node or edge in the CDFG. An AHIR fragment is listed as a tuple Δ that consists of fragments from the control-path, data-path and the link layer. In Figure 3.8, we list a general AHIR fragment. In particular, Δ^{op} denotes a fragment that implements the CDFG node for an operation op. Δ^C and Δ^D denote fragments that implement control and data edges respectively. For example, the AHIR fragment Δ^{mux} in Figure 3.9(b) AHIR fragment implements a CDFG multiplexer node.

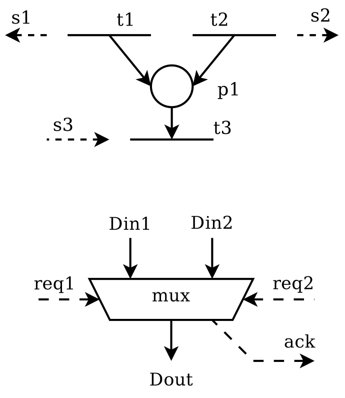
The interface of a fragment

The edges incident on a CDFG node represent the *interface* through which local events in the node interact with the rest of the CDFG. An AHIR fragment has a similar interface for interacting with the rest of the AHIR module. In the control-path, there are some transitions that initiate activity within the fragment, when they fire. These are termed as *entry transitions* of the fragment. Similarly, there are *exit transitions* in the fragment, whose firing marks the end of activity within the fragment. These transitions correspond to the incoming and outgoing control-edges respectively, that are incident on the corresponding CDFG node. A similar bijection exists between the data-ports on a CDFG node and the data-ports in the data-path fragment.

Atomic execution

When an AHIR fragment is triggered by the activation of an entry transition, it exhibits a trace of events, that results in a change in the state of the data-edges. The trace ends with the firing of an output transition. This trace of events is *atomic* in the sense that it is undisturbed by further events happening outside the fragment. These atomic traces of a particular fragment can be considered equivalent to the execution of the corresponding CDFG node.

3.4.2 The multiplexer node



(a) CP and DP elements.

$$\begin{aligned}
\Delta^{\text{mux}} &= (\text{CP}, \text{LN}, \text{DP}, \Lambda, \Sigma)\\
\ldots &\quad \ldots\\
T &= \{t_1, t_2, t_3\}\\
P &= \{p_1\}\\
E &= \{(t_1, p_1), (t_2, p_1), (p_1, t_3)\}\\
M &= \{(p_1 \rightarrow 0)\}\\
N &= \{\text{mux'}\}\\
\Lambda &= \{\text{req}_1, \text{req}_2, \text{ack}\}\\
\Sigma &= \{s_0, s_1, s_2\}\\
f &= \{s_1 \rightarrow \text{req}_1, s_2 \rightarrow \text{req}_2\}\\
r &= \{\text{ack} \rightarrow s_3\}
\end{aligned}$$

(b) The AHIR fragment.

Figure 3.9: AHIR fragment for a CDFG multiplexer node.

A CDFG multiplexer node implements the ϕ-function in the LLVM representation. It is implemented in AHIR as a data-path multiplexer node along with a control-path fragment that interacts with it through symbol handshakes, as shown in Figure 3.9. The petri-net consists of three transitions — two output transitions that trigger the multiplexer node through requests s_1 and s_2, and one input transition that waits for an acknowledge symbol s_3. The multiplexer begins execution in response to request symbols req_1 and req_2, and emits an acknowledge symbol

ack on completion. The symbols declared by the two fragments are translated in the link layer by the functions f and r.

An execution of the multiplexer fragment

The transitions t_1 and t_2 in Figure 3.9 are the entry transitions through which control enters the control-path fragment for a multiplexer node. A marking of the control-path where one of these two transitions is enabled represents the state in which the multiplexer is triggered. If both transitions are enabled at the same time, the result is undefined. But this cannot occur in a Type-2 petri-net since it is safe.

Consider a marking where transition t_1 is enabled. The transition fires, emitting symbol $s_1 \in \Lambda$, and also enabling the transition t_3. The symbol s_1 is translated to the request symbol $\mathrm{req}_1 = f(s_1)$, which triggers the multiplexer node in the data-path. The multiplexer samples the value on data-edge D'_{in_1} and drives it onto data-edge D'_{out}. It then emits the acknowledge symbol ack, which is translated by the link layer to the symbol $s_3 = r(\mathrm{ack})$, which in turn triggers transition t_3. A similar trace of events occurs when transition t_2 is enabled in the control-path fragment, where the value on data-edge D'_{in_2} is driven on the data-edge D'_{out}.

All symbols emitted during the execution of the fragment are consumed before control is passed on to subsequent fragments. The overall effect of this flow of control is that the requested value — D'_{in_1} or D'_{in_2} — is driven on the output data-edge D'_{out}.

Relation with the CDFG multiplexer node

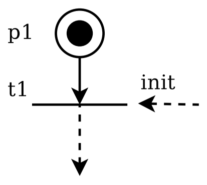
$$\begin{aligned}
C_{\mathrm{in}_1} &\rightarrow t_1 \\
C_{\mathrm{in}_2} &\rightarrow t_2 \\
C_{\mathrm{out}} &\rightarrow t_3 \\
D_{\mathrm{in}_1} &\rightarrow D'_{\mathrm{in}_1} \\
D_{\mathrm{in}_2} &\rightarrow D'_{\mathrm{in}_2} \\
D_{\mathrm{out}} &\rightarrow D'_{\mathrm{out}}
\end{aligned}$$

Figure 3.10: The interface of a multiplexer node

The AHIR fragment described above implements the behaviour of a CDFG multiplexer node. This can be demonstrated by comparing the traces of events in the behaviour of the

CDFG node described in Section 3.2.4 with the behaviour of the AHIR fragment. For example, the firing of transition t_1 in the AHIR fragment initiates a sequence of events which is equivalent to that in the CDFG multiplexer node when the control edge C_{in_1} is enabled. A similar trace of events can be listed for the transition t_2 in the AHIR fragment, which is equivalent to the control-edge C_{in_2} being enabled. In Figure 3.10, we show the mapping from the interface of a CDFG multiplexer node to that of the corresponding AHIR fragment.

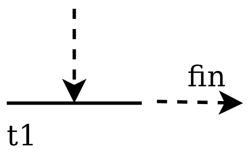
3.4.3 `start` and `stop` nodes



$$
\begin{aligned}
\Delta^{\mathrm{start}} &= (\mathrm{CP}, \mathrm{LN}, \mathrm{DP}, \Lambda, \Sigma)\\
\dots &\quad \dots\\
T &= \{t_1\}\\
P &= \{p_1\}\\
E &= \{(p_1, t_1)\}\\
M &= \{(p_1 \to 1)\}\\
\Lambda &= \{\mathrm{init}\}
\end{aligned}
$$

Figure 3.11: AHIR fragment for the CDFG start node.

The CDFG start node is translated to a control-path fragment as shown in Figure 3.11, made of a place followed by a single input transition that waits for the init symbol. The place is initially marked so that the transition is enabled when the AHIR module is initialised.

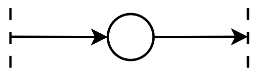


$$
\begin{aligned}
\Delta^{\mathrm{stop}} &= (\mathrm{CP}, \mathrm{LN}, \mathrm{DP}, \Lambda, \Sigma)\\
\dots &\quad \dots\\
T &= \{t_1\}\\
\Lambda &= \{\mathrm{fin}\}
\end{aligned}
$$

Figure 3.12: AHIR fragment for the CDFG stop node.

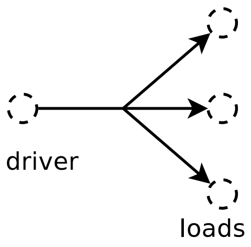
The CDFG stop node is equivalent to a single transition in the petri-net that has no output places. The transition is an output transition that emits the fin symbol for the environment, as shown in Figure 3.12.

3.4.4 CDFG edges



$$\begin{aligned} \Delta^C &= (\mathrm{CP}, \mathrm{LN}, \mathrm{DP}, \Lambda, \Sigma) \\ \ldots &\quad \ldots \\ P &= \{p_1\} \\ M &= \{(p_1 \to 0)\} \\ E &= \{(t_x, p_1), (p_1, t_y)\} \end{aligned}$$

(a) Control Edge



$$\begin{aligned} \Delta^D &= (\mathrm{CP}, \mathrm{LN}, \mathrm{DP}, \Lambda, \Sigma) \\ \ldots &\quad \ldots \\ W &= \{(u, \{v_0, v_1, \ldots\})\} \end{aligned}$$

(b) Data Edge

Figure 3.13: AHIR fragments for CDFG edges.

A CDFG control edge is mapped to a single place, with one incoming and one outgoing edge. The transitions corresponding to these edges lie in other fragments. These are identified by place-holders in the fragment listing.

A CDFG data edge is mapped to a single hyperedge in the data-path, with no elements in the control-path. There are no symbols in these fragments, nor any functions for the link layer. The single hyperedge is defined in terms of place-holders that indicate nodes in other data-path fragments.

3.4.5 Labelling scheme

While constructing the AHIR specification, we use a labelling scheme that associates each AHIR fragment with the CDFG element that it represents. Every node and edge in the CDFG is first assigned a unique label. When an AHIR fragment is created for a node or edge, each element in the fragment is assigned a label derived from the label of that node or edge.

This labelling serves two purposes. First, it is used to identify AHIR elements when connecting fragments as described later in Section 3.4.6. Second, the labelling scheme allows us to recover the CDFG that corresponds to a given AHIR specification. This is used in Section 1.1.2, where we prove that our construction method always produces an AHIR specification that is faithful to the original CDFG.

Labels for CDFG elements

Let $G = (N, E)$ be the CDFG, and $\mathcal{L}$ be the set of labels used on the CDFG. The type of the labels is not important, as long as they are unique. We have the following labelling for the CDFG:

$$\begin{aligned} \text{label} &: N \cup E \to \mathcal{L} \\ \text{label}(x_i) &= \text{label}(x_j) \iff x_i = x_j \\ &\ \forall x_i, x_j \in N \cup E \end{aligned}$$

Labels for AHIR elements

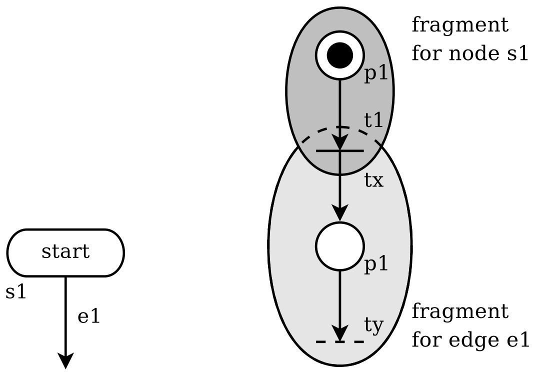
Consider a CDFG element with label $a \in \mathcal{L}$. When it is translated to an AHIR fragment, each element in the fragment is assigned a label of the form (a, z), where z identifies the element within the fragment. For example, elements in the AHIR fragment that corresponds to a CDFG multiplexer node with label a are labelled as follows:

$$\begin{aligned} t_1 &\to (a, t_1) \\ p_1 &\to (a, p_1) \\ s_1 &\to (a, s_1) \\ \text{mux'} &\to (a, \text{mux'}) \\ \text{req}_1 &\to (a, \text{req}_1) \\ &\ldots \end{aligned}$$

3.4.6 Connecting fragments

Once the compiler generates fragments for individual CDFG element, the complete AHIR specification is obtained by connecting the fragments together. A connection is established between a node fragment and a control-edge fragment by assigning the relevant transition in the node fragment to the placeholder in the control-edge fragment.

For example, consider an outgoing control-edge with label e_1, incident on the start node with label s_1, as shown in Figure 3.14. The start node is mapped to a fragment containing a marked place p_1, a transition t_1, and an edge (p_1, t_1). The control-edge is mapped to a fragment with a single unmarked place p_1, and two incomplete edges that include place-holders for transitions in other fragments. In particular, the place-holder t_x represents the transition t_1 in the fragment for the start node.



(a) CDFG. (b) AHIR.

Figure 3.14: Connecting fragments

The members of the two fragments are assigned labels derived from the original CDFG elements, as shown in Figure 3.15. The place-holder t_x refers to the relevant transition using its label, (s_1, t_1). For simplicity, we assume that labels can be interchanged with the elements they represent, so that the edge (t_x, p_1) can be rewritten as $\big((s_1, t_1), p_1\big)$.

$$\begin{aligned} t_1 &\rightarrow (s_1, t_1) \\ p_1 &\rightarrow (s_1, p_1) \\ (p_1, t_1) &\rightarrow \big(s_1, (p_1, t_1)\big) \end{aligned}$$

(a) Start node fragment $\Delta^{\mathrm{start}}_{s_1}$.

$$\begin{aligned} p_1 &\rightarrow (e_1, p_1) \\ t_x &\simeq (s_1, t_1) \\ (t_x, p_1) &\rightarrow \big(e_1, (t_x, p_1)\big) \\ (p_1, t_y) &\rightarrow \big(e_1, (p_1, t_y)\big) \end{aligned}$$

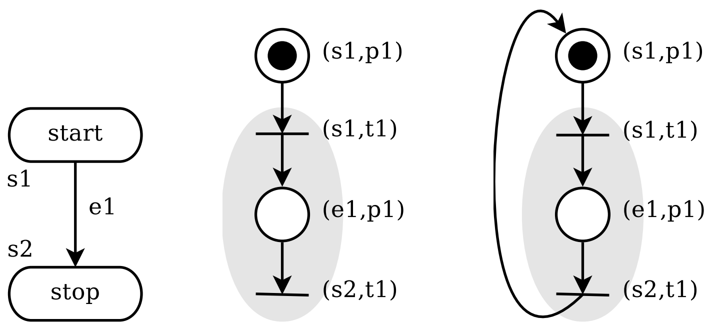
(b) Control edge fragment $\Delta^{C}_{e_1}$.

Figure 3.15: Labelling in AHIR fragments.

Note that for every edge that is incident on a node in the CDFG, exactly one connection is established between the corresponding AHIR fragments. This establishes a bijection that maps the incidence of CDFG edges on nodes to the connections between AHIR fragments.

3.4.7 Creating a Type-2 petri-net

The piecewise translation of the CDFG results in a standard TPR in the control-path, as defined in Section 2.8. This is because a C program does not contain any constructs for parallel execu-



(a) CDFG. (b) Class-A STPR. (c) Type-2 Petri-net.

Figure 3.16: Completing the Type-2 petri-net.

tion. The CDFG derived from the C program introduces forks and joins, but these are restricted to the body of each basic block. Each basic block thus represents a fork region, used in the construction of a branch region that represents the entire program.

This branch region is part of a series region that includes the transitions introduced by the start and stop nodes. The resulting series region is always a Class A standard TPR. We construct a Type-2 petri-net from this standard TPR by connecting the exit to the entry. As an example, the smallest possible CDFG is shown in Figure 3.16(a). It consists of a start node and a stop node connected with a control-edge. The resulting standard TPR is shown in Figure 3.16(b). We complete the Type-2 petri-net with a single edge as shown in Figure 3.16(c).

3.4.8 Piece-wise translation from CDFG to AHIR

The process of translating a CDFG to an AHIR specification can be summarised as the sequence of steps enumerated below. This process automatically handles loops in the CDFG, since the connections between the nodes and edges of the CDFG are preserved in the resulting AHIR specification. This property is used in Section 3.5 to show an equivalence between the CDFG and the resulting AHIR specification.

1. For each node n in the CDFG instantiate the corresponding AHIR template Δ^{op} where op is the operation performed by that node (Section 3.4.1).

2. For each data-edge (u, v) in the CDFG, instantiate the AHIR template Δ^{D} (Section 3.4.4).

3. For each control-edge (u, v) in the CDFG, instantiate the AHIR template Δ^{C}.

4. For the start and stop nodes in the CDFG, instantiate the AHIR templates Δ^{start} and Δ^{stop} respectively (Section 3.4.3).

5. For each edge e incident on a node n, connect the AHIR fragments that correspond to that node and edge (Section 3.4.6).

6. The AHIR fragments for the start and stop nodes contain the init and fin transitions respectively. The fragment for the start node also contains a marked place p. Connect the place to the two transitions using edges (p, init) and (fin, p). This completes the Type-2 petri-net for the control-path (Section 3.4.7).

3.5 Equivalence

The synthesis process generates an AHIR specification from the input CDFG by replacing each node and edge in the CDFG with an *equivalent* fragment. In order to show that this AHIR specification is a correct implementation of the input program we show that it is equivalent to the intermediate CDFG. In general, an implementation can be shown to be equivalent to the input specification in two ways:

Black-box equivalence: When the system is treated as a black box, only the outcomes of an execution of the system are available for examination, since the actual events in the system cannot be traced. For each initial state, a system specification provides the set of possible final states that can be reached by an execution of the system. An implementation is *equivalent* if the set of outcomes obtained by executing it has a one-to-one correspondence with the specified outcomes.

Trace equivalence: Another way to demonstrate equivalence is to trace the operations that occur for a given input, and show that these traces are equivalent. In general there can be multiple such traces for a given initial state of the system. The implementation is equivalent to the specification if there is a one-to-one correspondence between the traces possible in the implementation, and those in the specification for the same initial state.

When generating AHIR from the CDFG, we demonstrate equivalence in terms of the operation traces. This is stronger than simply showing that the results are equivalent. Trace equivalence implies that there is no loss of information when translating the CDFG to AHIR.

The AHIR implementation is closer to hardware since it separates operations in the control path from operators in the data-path that implement them. But every operation in the CDFG is correctly represented in the AHIR implementation. Thus every analysis and transformation that is possible in the CDFG is also possible in the AHIR implementation.

Let G be the CDFG derived from an input program P. We assume that the CDFG G correctly implements the program P. Let A be the AHIR specification constructed from the CDFG G. Let $S_G(X)$ be the set of traces in G for an input X and $S_A(X)$ be the corresponding set of traces in A. We prove that there is a trace equivalence between A and G:

$$\forall X, S_A(X) \equiv S_G(X) \tag{3.1}$$

Outline of the proof:

It is impractical to enumerate the behaviour of the CDFG and its AHIR implementation for an arbitrary program. Instead, we exploit the piece-wise nature of the construction method to prove that the structure of the implementation is itself equivalent to the input CDFG. The AHIR specification is built up by substituting an AHIR fragment for each element in the CDFG and then connecting those fragments together. We use the following assumption about the CDFG elements and the corresponding AHIR fragments:

> *For a CDFG element (node or edge) e, the corresponding AHIR fragment Δ^e has a behaviour that is* equivalent *to that of the element e — there is a one-to-one correspondence between the sets of sequences observable in e and Δ^e.*

Using the equivalence between the CDFG elements and the corresponding AHIR fragments, we can inductively show that the AHIR implementation A correctly implements the CDFG G. Note that this is a subset relationship — every trace in $S_A(X)$ has an equivalent trace in $S_G(X)$, but the opposite is not necessarily true.

$$\forall X, S_A(X) \subseteq S_G(X) \tag{3.2}$$

We now translate the AHIR implementation A to a CDFG G' which is itself a correct implementation of A:

$$\forall X, S_{G'}(X) \subseteq S_A(X) \tag{3.3}$$

In general, there can be many such graphs derived from the implementation. The method we use results in a graph G' which is isomorphic to G — there is a one-to-one correspondence between the nodes and edges in G and G', and also between the sets of sequences observable in G and G'. Thus, the behaviour of G' is equivalent to the behaviour of G:

$$\forall X, S_{G'}(X) \equiv S_G(X) \tag{3.4}$$

Equations 3.2, 3.3 and 3.4 together imply Equation 3.1, hence proving that the AHIR specification is equivalent to the input CDFG.

Deriving a CDFG from AHIR

The labelling scheme described in Section 3.4.5 allows us to segregate AHIR elements into groups using the first component a of their label. Each such group is a fragment Δ_a, that was created for a particular element x with label a in the original CDFG.

$$x \simeq \Delta_a \text{ where } \mathrm{label}(x) = a \tag{3.5}$$

From each fragment Δ_a, we can infer a new CDFG element x'. Clearly, this new element x' is *equivalent* to the original element x, and we assign it the same label a.

$$\Delta_a \simeq x' \text{ where } \mathrm{label}(x') = a \tag{3.6}$$

Using the labels to identify fragments, we generate a new CDFG $G' = (N', E')$ as follows:

- For each CDFG node n in G, there exists one fragment in the AHIR specification A identified by the label assigned to n. We map this fragment to a new CDFG node n' of the same kind as n, and assign it the same label.
- For each control or data edge e in G, there is one fragment in the AHIR specification identified by the label assigned to e. We map this fragment to a new CDFG control or data edge e' respectively and assign it the same label.
- For an edge e incident on a node n in the CDFG G, the corresponding fragments in the AHIR specification are connected. The portmap for the corresponding node n' is updated to indicate the corresponding edge e'.

The result of this construction is a new CDFG, with labelled nodes and edges. These labels can then be used to demonstrate an isomorphism with the original CDFG.

Proving Equation 3.2 and Equation 3.3

Due to the manner in which A is constructed from G, we see that for every node or edge in G, there is an equivalent fragment in A. From this we can make the following observations:

1. Any operation performed by a fragment in A has an equivalent operation performed by some node or edge in G.
2. Any sequence of operations across fragments in A has an equivalent sequence of operations across the corresponding nodes in G.

Thus, the behaviour of A is included in the behaviour specified by G. And hence, A is a correct implementation of G. This proves Equation 3.2 in the outline of the proof. We can similarly prove Equation 3.3 — the behaviour of the new CDFG G' is included in the behaviour specified by the AHIR specification A.

Proving Equation 3.4

From the manner in which A is constructed from G, and G' is in turn constructed from A, we can show that every element x in G can be mapped to an element in x' in G', such that:

1. The mapping is one-to-one and onto.
2. If $n \in N$, then it is mapped to $n' \in N'$ such that n and n' are behaviour equivalent. In particular, the start and stop nodes in N are mapped respectively to the start and stop nodes in N'.
3. Every edge $e \in E$ is mapped to an edge $e' \in E'$ such that $\mathrm{driver}(e)$ is mapped to $\mathrm{driver}(e')$ and the set $\mathrm{loads}(e)$ is mapped in a one-to-one fashion on the set $\mathrm{loads}(e')$.

The CDFG G' is isomorphic to the CDFG G and is in fact *indistinguishable* from G. Every sequence of operations that occurs in G' is also specified in G, and *vice versa*. This proves Equation 3.4 in the outline of the proof. Thus it is proved that Equation 3.1 is true. The AHIR specification A derived by our construction method is equivalent to the input CDFG G.

3.6 Summary

A program written in C can be easily translated to an AHIR specification using a CDFG as an intermediate step. We use a piece-wise construction to obtain the AHIR specification from the CDFG, as described in Section 3.4. This method is proven to be correct — the behaviour of an AHIR specification created by this process always implements the behaviour specified by the input CDFG. The resulting AHIR specification is well-formed, since the control-path is always a Type-2 petri-net as seen in Section 3.4.7.

Chapter 4

Contention-free Reuse of Hardware

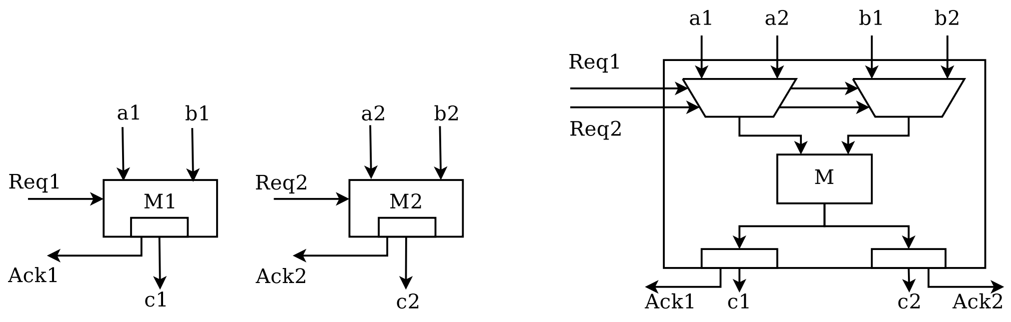
The translation from a high-level program to AHIR results in a circuit specification that decouples control and data components of the behaviour. Every element in the specification has a clearly described behaviour, so that the entire specification can be routinely translated to hardware. We have implemented a simple mapping from AHIR to synthesisable VHDL that uses a pre-defined library of RTL descriptions for the building blocks. But the AHIR specification assigns one operator to every operation in the input CDFG. No optimisation is done at generation time, since it is desirable to allow the implementer to choose an appropriate strategy for optimising resource utilisation. Utilisation can be improved by sharing resources, so that the same operator is allocated to multiple operations.

In general, such a sharing of operators may cause contention, if the corresponding operations are active at the same time. One way to tackle this contention is to use an arbiter that controls access to the shared operator. But arbitration introduces unpredictability in the behaviour of the system. It also involves the overhead of dealing with issues such as fairness, deadlock and starvation. Instead we propose a scheme that avoids this cost by sharing an operator between only those operations that are never active simultaneously.

The scheme is based on a static analysis of the control-path that provides exhaustive information about such opportunities for sharing hardware. The analysis exploits the structure of a Type-2 petri-net to identify all the pairs of operations that are guaranteed not to be active at the time. We use a simple greedy approach to choose pairs of operations from these candidates to share hardware. Synthesis results show that this simple scheme itself is quite effective in reducing hardware costs. This optimisation effectively demonstrates how the factorisation in AHIR makes it easy to analyse and transform an AHIR specification.

We first introduce the notion of "compatible operations" — operations that can share hardware without contention. Then in Section 4.1.1, we define compatibility in a Type-2 petri-net in terms of paths in the petri-net. We introduce a labelling scheme in Section 4.2, that encodes these paths as labels associated with the petri-net elements and in Section 4.4, we provide an equivalent definition of compatibility based on labels. Finally in Section 4.5, we present a compact graphical representation of the labels which makes it possible to determine compatibility using an algorithm that is almost linear in complexity, as described in Section 4.7.

4.1 Compatible operations



(a) Individual Operators. (b) Shared Operator.

Figure 4.1: Sharing a data-path operator.

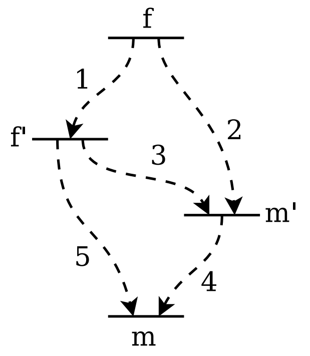
Consider two operators M1 and M2, that respond to requests $\mathrm{req1}$ and $\mathrm{req2}$ respectively as show in Figure 4.1. They generate acknowledgements $\mathrm{ack1}$ and $\mathrm{ack2}$ respectively at the end of execution. The operators can be replaced with a single shared operator if their activity does not overlap in the control-path — if $\mathrm{req1}$ is emitted, then $\mathrm{req2}$ should not be emitted before $\mathrm{ack1}$ is received, and *vice versa*. Pairs of operations that satisfy this property are said to be *compatible* with each other. The compatibility relation $(\simeq)$ is symmetric, but not transitive.

$$\begin{aligned} a, b, c \quad &: \quad \text{operations scheduled in the control-path} \\ a \simeq b \quad &\Longleftrightarrow \quad b \simeq a \\ a \simeq b \text{ and } b \simeq c \quad &\not\Longrightarrow \quad a \simeq c \end{aligned}$$

We propose static analysis that exploits the structure of a Type-2 petri-net to determine whether a given pair of operations is compatible. The resulting compatibility information is independent of actual delays in the implementation.

4.1.1 Compatibility in a Type-2 petri-net

Two operations in a petri-net are *incompatible* if and only if there exists a marking of the petri-net where both the operations are enabled. Such a marking arises in a Type-2 petri-net (or a safe petri-net in general) when a fork receives a token and generates tokens that flow to the two operations. The operations are activated concurrently, making them incompatible. But the actual condition that determines incompatibility is slightly more complex, because there may be other structures in the petri-net that force these operations to execute in sequence.



(a) Path segments in a fork region.

	1	2	3	4	5
1	Y	-	Y	Y	Y
2	-	Y	-	Y	-
3	Y	-	Y	Y	-
4	Y	Y	Y	Y	-
5	Y	-	-	-	Y

(b) Compatibility for different segments.

Figure 4.2: Compatibility in a petri-net.

In Figure 4.2(a), we show a fork region with different segments identified by labels. The fork f gives rise to two tokens that travel concurrently along segments 1 and 2. Operations that lie on one of these segments are incompatible with those that lie on the other.

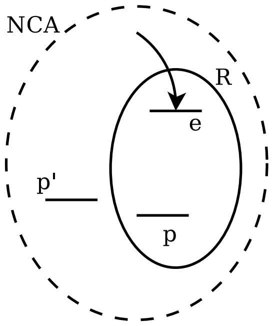
But the presence of f' and m' creates additional copies of the token along segment 1, one of which continues to segments 3 and 4. As a result, operations in segment 1 are compatible with operations in segments 3 and 4, as well as those in segment 5. The compatibility table for operations lying on different combinations of segments is listed in Figure 4.2(b).

Incompatibilities arise only between different path segments in a fork region. Branch and serial regions do not affect compatibility, since tokens do not get replicated in these regions. It is clear that in order to determine compatibility between two elements, we only need to examine the region that contains both of them, i.e., their NCA as defined in Section 2.8.4. Based on this discussion, we have the following definition of compatibility in a Type-2 petri-net:

Definition 4.1.1 *Two elements p_1 and p_2 in a Type-2 petri-net are compatible, if and only if $\mathrm{NCA}(p_1, p_2)$ is not a fork region or there is a path between the regions within the NCA.*

The following theorem establishes the relationship between compatible operations and standard TPRs in the Type-2 petri-net. This allows us to discuss the compatibility of arbitrary elements in terms of the standard TPRs that contain these elements.

Theorem 4.1.1 *In a Type-2 petri-net, an element p inside a standard region R is compatible with an element p' outside R if and only if the entry e of the region R is compatible with p'.*



Proof:

Case 1: If p is compatible with p', one of the following is true:

1. $\mathrm{NCA}(p, p')$ is not a fork region
2. There is a path within the NCA, joining p and p'

The element p' is outside the region R while both p and e are inside it. Hence, we have: $\mathrm{NCA}(p, p') = \mathrm{NCA}(e, p')$. If this is not a fork region, then p' is compatible with e. If it is a fork region, there must exist a path joining p and p'. If p' occurs first on this path, then it must pass through the entry e before reaching p. If p occurs first on the path, it is possible to construct a path from e to p', since there must exist a path from e to p within the region R. In either case, p' is also compatible with e.

Case 2: If p' is compatible with e, we can similarly show that p' is compatible with p — either the NCA is not a fork region, or it is possible to construct a path joining p' and p.

4.2 Labels to indicate compatibility

The compatibility relation in Definition 4.1.1 is based on paths in a Type-2 petri-net. We encode information about these paths as labels assigned to the elements in the petri-net. Two operations can be checked for compatibility by comparing the labels assigned to them. Since compatibility is only affected by forks and joins, the labels are designed to indicate only the forks and joins that occur along each path that reaches an element.

In Figure 4.3, we show a petri-net fragment with only the transitions visible. Class-C standard TPRs (such as simple places, series regions or branch regions) that occur between these transitions are suppressed to save space. The label L assigned to the transition at the top is propagated along petri-net edges. When a fork is encountered, new labels are created to identify tokens generated at the fork (such as L_1, L_2, etc.) Eventually when all the tokens reach a join, the original label is restored (such as the join at the bottom of the figure.)

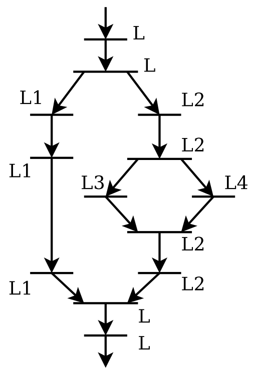


Figure 4.3: A labelled petri-net fragment.

A label is a set $L = \{x_0, x_1, \ldots\}$ where each member is a sequence of label elements $x = [a_0, a_1, \ldots, a_n]$. A label element is a 3-tuple $a = (f, k, i)$, containing a fork identifier f, the fan-out k of the fork, and an index i into the fan-out of the fork. A label element $a = (f, k, i)$ is said to *indicate* the fork f. The length of a sequence is the number of label elements in the sequence, represented as function $l : \text{label} \to N$. The empty sequence is denoted by ϕ, so that $l(\phi) = 0$. The empty label is represented by Φ.

1. **Equality:** Two label elements $a = (f_a, k_a, i_a)$ and $b = (f_b, k_b, i_b)$ are equal if and only if the corresponding members are equal.

   $$a = b \iff f_a = f_b \text{ and } k_a = k_b \text{ and } i_a = i_b$$

   Two sequences $x = [a_0, a_1, \ldots, a_{l(x)-1}]$ and $y = [b_0, b_1, \ldots, b_{l(y)-1}]$ are equal if and only if corresponding label elements are equal.

   $$x = y \iff l(x) = l(y) \text{ and } a_i = b_i \text{ for } 0 \le i < l(x)$$

2. **Concatenation:** The concatenation operator joins two sequences to create a longer sequence. Given two sequences $x = [a_0, a_1, \ldots, a_{l(x)-1}]$ and $y = [b_0, b_1, \ldots, b_{l(y)-1}]$, the term $x.y$ represents the sequence $[a_0, a_1, \ldots, a_{l(x)-1}, b_0, b_1, \ldots, b_{l(y)-1}]$.

   The following properties are true for concatenation:

$$\begin{aligned} l(x.y) &= l(x) + l(y) \\ \phi.x &= x.\phi = x \\ x.(y.z) &= (x.y).z \end{aligned}$$

   For convenience, the same operator is used to denote the concatenation of a single label element b to a sequence x. In this case, the single element is considered a sequence of unit length.

$$x.b = x.[b]$$

3. **Body and tail:** For a sequence $x = [a_0, a_1, \ldots, a_{l-1}]$, the function $\mathrm{tail}(x) = a_{l-1}$ returns the last element in the sequence, while the function $\mathrm{body}(x) = [a_1, \ldots, a_{l-2}]$ returns the entire sequence except the last element.

4. **Product:** The product of two labels X and Y is given by:

$$\begin{aligned} X * Y &= Y \text{ if } X = \Phi, \\ &\quad X \text{ if } Y = \Phi, \\ &\quad \{x_i.y_j | \forall x_i \in X, y_j \in Y\} \text{ otherwise} \end{aligned}$$

   For convenience this operator is also overloaded to allow the product of a label X with a label element a:

$$X * a = X * \{[a]\}$$

   The product operator is distributive over the label union:

$$X * (A \cup B) = (X * A) \cup (X * B)$$

5. **Prefix relation:** Consider two sequences $x = [a_0, a_1, \ldots, a_{l(x)-1}]$ and $y = [b_0, b_1, \ldots, b_{l(y)-1}]$. The sequence x is a prefix of sequence y, if and only if there exists a possibly empty sequence w, such that $y = x.w$. The prefix relation $(\leq)$ imposes a partial order on the set

of sequences.

$$x \leq y \iff l(x) \leq l(y) \text{ and }$$
$$a_i = b_i \text{ for } 0 \leq i \leq l(x) - 1, \text{ if } l(x) > 0$$

The following properties are true for the prefix relation:

$$x = y \iff x \leq y \text{ and } y \leq x$$

$$\phi \leq x, \forall x$$

6. **Longest Common Prefix:** The Longest Common Prefix (LCP) of a set of label sequences S is the longest label sequence that is a prefix of every sequence in S. It is defined in terms of the concatenation operator $(.)$ as follows.

   A non-empty label sequence x can be represented as $y.a$ where y is a possibly empty label sequence and a is a label element. The sequence y is said to be a *parent* of sequence x, while x is said to be a *child* of y.

   Clearly, every non-empty sequence has exactly one parent. The set of all label sequences can be represented as a single-rooted directed tree T where the nodes represent label sequences while edges represent parent-child relationships. A directed edge (y, x) exists in the tree if and only if y is the parent of x, i.e., $x = y.a$. The single root of this tree is the empty sequence (ϕ).

   In this tree, $\mathrm{LCP}(S)$ is the root of the smallest subtree T' that contains every sequence in S. Since the prefix relation is reflexive, $\mathrm{LCP}(S)$ can itself be a member of S.

4.2.1 Labelling scheme

Before labelling, the parallel-merge regions described in Section 2.8.3 are reduced to simple merges. The labelling scheme begins by assigning an empty label to the init transition and then traverses the petri-net by visiting each successor of a recently labelled element. A petri-net element p is assigned the same label as its predecessor(s) in two cases:

1. p has a single predecessor p' which is either a branch or has a single successor p.

2. p is a merge, in which case, all its predecessors have the same label.

Besides this, there are two special cases — when the predecessor p' is a fork, and when p is a join. These are handled as follows.

4.2.2 Labelling successors of a fork

A fork is a transition with a single incoming edge and multiple outgoing edges. Let f be a fork with $k(f)$ successors and L be the label assigned to it. Each successor s_i is assigned the label $L * \big(f, k(f), i\big)$, as shown in Figure 4.4.

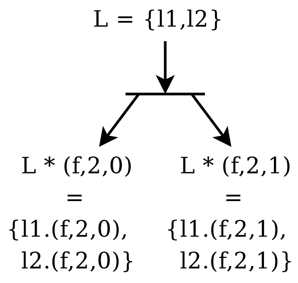


Figure 4.4: Labelling successors of a fork.

4.2.3 Labelling a join

A join is a transition with multiple incoming edges and a single outgoing edge. It is assigned the union of the labels assigned to all its predecessors as shown in Figure 4.5(a). In some cases, the label assigned to a join is also *reduced* as shown in Figure 4.5(b) and 4.5(c).

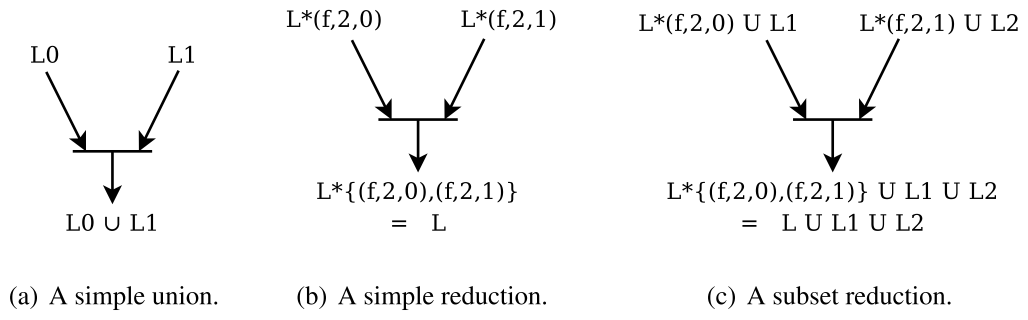


Figure 4.5: Labelling at a join.

Reducible Subsets

The union of labels computed at a join is reduced by identifying *reducible subsets*. A subset J of a label L is said to be *reducible* with respect to some fork f, if it can be rewritten as a product expression as follows:

$$\begin{aligned}
J &= U * V, \text{ for some } U, V \text{ such that} \\
V &= \{v_0, v_1, \ldots, v_{k(f)-1}\} \\
v_i &= \big(f, k(f), i\big)
\end{aligned}$$

In the above expression, the set U is in fact the label assigned to the fork f, and the set V represents the elements introduced at the successors of fork f.

Each reducible subset in a label is replaced by the function reduce. Let $\mathrm{reducibles}(L)$ be the set of reducible subsets in some label L. The function $\mathrm{reduce} : \mathrm{label} \to \mathrm{label}$ is defined as:

$$\begin{aligned}
\mathrm{reduce}(L) &= L - J + U \\
&\quad \text{for } J = U * V \in \mathrm{reducibles}(L)
\end{aligned}$$

This reduction is performed recursively until a fixed point is reached, represented by the function *reduce**.

$$\begin{aligned}
\mathrm{reduce^*}(L) &= L \text{ if } \mathrm{reduce}(L) = L \\
&\quad \mathrm{reduce^*}\big(\mathrm{reduce}(L)\big) \text{ otherwise}
\end{aligned}$$

Note that reducible subsets in a label can be reduced independently. The reduction of a reducible subset in a label does not break other reducible subsets that may exist in the label. This follows from the fact that for any subset J that is reducible with respect to some fork f, every sequence in J ends with an element of the form $\big(f, k(f), i\big)$. Consider another reducible subset J' such that some sequence $j \in J$ is also in J'. It follows that J' is also reducible with respect to the same fork f. Since both J and J' are reducible, the presence of j implies that there is a subset Q in both J and J' such that:

$$Q = \mathrm{body}(j) * \{v_0, \ldots, v_{k(f)-1}\}$$

Q is itself a reducible subset and when J is reduced, the subset Q vanishes entirely and is replaced by the sequence $\mathrm{body}(j)$ in set J', and the resulting set $J' - Q$ is also reducible. This is true for every sequence that occurs in both sets J and J'. In the special case where $J' \subseteq J$, J' vanishes entirely when J is reduced.

4.3 Concurrency encoded in labels

The labelling scheme is a symbolic execution of the Type-2 petri-net that tracks the parallelism introduced by forks. The labels are assigned to petri-net elements, but they actually record the flow of tokens in the petri-net. A label for a petri-net element contains a label element indicating fork f if and only if it receives a token from f which has not yet merged with all the other tokens produced by f. This is a result of the interaction between forks and joins that occur within a fork region. In this section, we define a number of terms to describe this interaction and describe the relationship between labels and concurrency in the form of Theorem 4.3.2.

Definition 4.3.1 *For a fork f in a fork region R, an element that occurs on every path from f to the exit of the region is called a* ***post-dominator*** *of the fork f.*

Clearly, the exit of the region is itself a post-dominator of every fork in the region. The following properties are true for two post-dominators p and p' of a fork f in region R:

1. *Both p and p' must lie on every path from f to the exit of the region.* This automatically follows from the definition of a post-dominator.

2. *If p occurs earlier than p' on some path P from f to the exit of the region, then it does so on every such path.* If this were not true, then there is a path P' such that p' occurs before p. This implies that there is a cycle in the fork region, passing through p and p', which is not allowed by the definition of a Type-2 petri-net. Hence p must occur before p' on every path from f to the exit of region R.

This relationship between multiple post-dominators allows us to identify a unique element called the *immediate post-dominator* of a fork f.

Definition 4.3.2 *For a fork f in a fork region R, the post-dominator of f that occurs before any other post-dominator on every path from f to the exit of the region is called the* ***immediate post-dominator*** *of f.*

The element where two paths meet in a fork region is always a join, and hence the immediate post-dominator is a join. This join j is said to be the **associate join** of the fork f. It represents the boundary of the influence of fork f — all tokens starting from a fork must meet at its associate join and the concurrency introduced by the fork collapses into a single thread beyond that join.

Similarly, we can define the notion of a **pre-dominator** of a join j, which is a fork f that must occur on every path from the entry of the region R to that join. The **immediate pre-dominator** of a join is called the **associate fork** of that join.

Note that the $\mathrm{associate}$ relationship is not symmetric. If a fork f is the associate of a join j, it is not necessary that the join j is also the associate of the fork f.

Theorem 4.3.1 *In a fork region R, consider a fork f with associate join j and another fork f' that lies on a path from f to j. If j' is the associate join of f', then one of the following is true:*

1. *j' is j itself.*
2. *j' is encountered on every path from f' to j.*

The behaviour implied by this property is that if a concurrent thread starting from a fork f reaches another fork f', then the concurrent threads further started by f' must converge before those started by f, or along with them.

The first case is only a special case. If it is not true, we need to prove that the second case is true. Consider a path from f' that goes out of region R. Since j' is the associate join, it lies on such a path. If j is encountered before j' on this path, then there exists a path from f' to j' that does not pass through j, but reaches the exit of the region R. This implies that there is a path from the fork f (through f' and j', that does not pass through j but still reaches the exit of the region R. This cannot be true since j is the associate join of the fork. Hence, j must lie after j' on every path from f' to the exit of the region.

4.3.1 Canonical form of a fork region

The Type-2 construction rule for a fork region allows subgraphs that are themselves fork regions. These subgraphs can be identified from their entries and exits — if a fork f and join j in a fork region R are associates of each other, then the set of elements that lie on every path from f to j is itself a fork region. A fork region that does not contain such smaller fork regions is said to be in the canonical form. Note that the canonical form does not affect the actual petri-net itself, but only its representation in terms of the standard TPRs. The canonical representation allows us to simplify the treatment of fork regions in later sections.

Definition 4.3.3 *A* ***canonical fork region*** *is a fork region where a fork f and a join j in the region are associates of each other if and only if they are the entry and the exit of the region respectively.*

Two types of subgraphs occurring in a non-canonical fork region can be identified as smaller fork regions:

Case 1: If the exit j of a fork region is not the associate of the entry f, then there exists an associate join j' for the entry fork f and similarly an associate fork f' for the exit join j. The fork region can be replaced by a series region made of three regions:

1. The fork region defined by f and j'.
2. The set of elements lying on every path from j' to f'.
3. The fork region defined by f' and j.

Case 2: If some fork f' and join j' other than the entry and exit are associates of each other, then they form a fork region that in turn forms a series region along with the region preceding f' and the region succeeding j'.

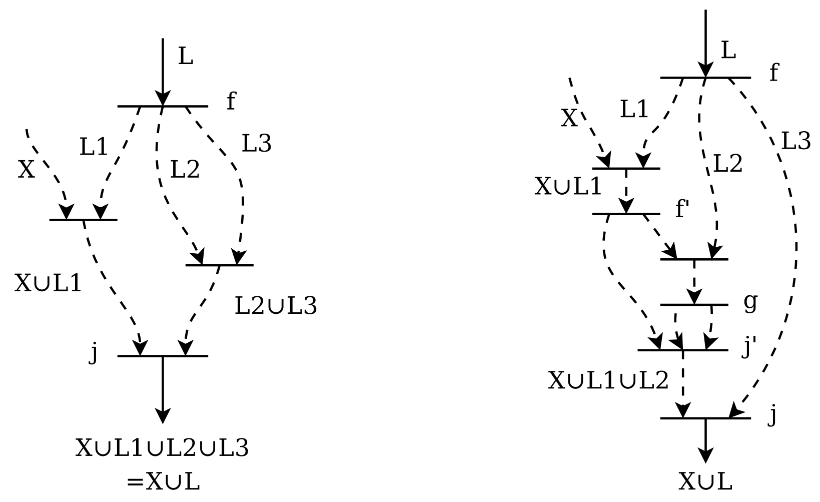
4.3.2 Labelling in a canonical fork region

Theorem 4.3.2 *If j is the associate join j of a fork f in region R, then there is no sequence in the label assigned to j that contains a label element indicating fork f.*

We prove this property by induction on the structure of a canonical fork region. For the child regions of this region, we make the following general assumption, which we prove as a theorem in the next section: *The label assigned to the exit of a region is the same as the label assigned to its entry.*

Base case: A region where joins occur on any path from f to j (but not forks) as shown in Figure 4.6(a).

Let L be the label assigned to fork f. Each successor s_i of the fork f is assigned the label $L_i = L * \big(f, k(f), i\big)$, which is propagated along all paths starting from s_i. Since there are no other forks in the region, this label is not modified by further product operations. A path reaching j from the entry of the region R may or may not have passed through the



(a) Base case: No intermediate forks. (b) General case: With intermediate forks.

Figure 4.6: Associate join for a fork.

fork f. The label assigned to j is computed from the union of all the labels arriving along all such paths. The entire union of labels at j can be decomposed into two terms as shown below. The second term is reducible with respect to the fork f, and can be replaced by the label L:

$$\begin{aligned} L^j &= X \cup (L_1 \cup \ldots \cup L_{k(f)-1}) \\ &= X \cup L \\ X &: \text{the union of labels arriving along paths that do not pass through } f. \end{aligned}$$

In the above expression, neither of the terms contains a label element indicating fork f. Hence, the theorem is true for the base case.

General case: In the general case we allow the presence of other forks on any path from f to j as shown in Figure 4.6(b). Consider a fork f' on a path from f to j, such that no other fork occurs on any path from f to f'. Since this is the first fork encountered, the label assigned to it has the following form:

$$\begin{aligned} L^{f'} &= X \cup L' \\ X &: \text{the union of labels arriving along paths that do not pass through } f. \\ L' &= L_{i_1} \cup L_{i_2} \cup \ldots \\ &\ \text{the union of labels assigned to successors of fork } f \text{ on paths from } f \text{ to } f'. \end{aligned}$$

We assume that Theorem 4.3.2 is true for every such fork f', i.e., the label assigned to the associate join j' does not contain any label element indicating fork f'. This label has the following form:

$$\begin{aligned} L^{j'} &= Y \cup L^{f'} \\ Y &: \text{the union of labels arriving along paths that do not pass through } f'. \end{aligned}$$

The paths reaching j' may or may not pass through the outer fork f. Hence the label at join j' can be decomposed into two components similar to the label at fork $L^{f'}$.

$$\begin{aligned} L^{j'} &= X \cup L' \\ X &: \text{the union of labels arriving along paths that do not pass through } f. \\ L' &= L_{i_1} \cup L_{i_2} \cup \ldots \\ &\ \text{the union of labels assigned to successors of fork } f \text{ on paths from } f \text{ to } j'. \end{aligned}$$

From Theorem 4.3.1, j' is either the join j itself or it lies on every path from f' to j. If j' is the join j itself, then $L^{j'}$ is in fact only a subset of the label L^j assigned to join j, which is a union of all the labels arriving at j. The presence of fork f' did not produce any terms indicating fork f in label L^j.

If j' is not the join j, then further forks may occur along the path from j' to j. But in each case we can make the same argument as above, and eventually conclude that the presence of any fork along a path from f to j is not reflected in the label L^j. This implies that the labels assigned to the successors of fork f are unmodified when they eventually reach the join j, so that the label at j can be written as:

$$\begin{aligned} L^j &= X \cup (L_1 \cup \ldots \cup L_{k(f)-1}) \\ &= X \cup L \end{aligned}$$

Hence proved that the label at the associate join of a fork does not contain any label elements indicating that fork.

Theorem 4.3.3 *If f is the entry of a fork region R, and j is the exit of the region, then the label assigned to j is the same as the label assigned to f.*

The entry f and the exit j are associates of each other in a canonical fork region. From Theorem 4.3.2, if L is the label assigned to the fork f, then the label assigned to the join j can be written as:

$$\begin{aligned} L^j &= X \cup (L_1 \cup \ldots \cup L_{k(f)-1}) \\ &= X \cup L \\ X &: \text{the union of labels arriving along paths that do not pass through } f. \end{aligned}$$

But since f and j are associates of each other, there is no path that passes through j but does not pass through f. Hence the term X in the above expression is empty, and the label assigned to the join is the same as the label assigned to the fork.

4.3.3 Labelling in a Type-2 petri-net

Theorem 4.3.4 *The label assigned to the exit of a standard TPR is the same as the label assigned to its entry.*

We prove this theorem by induction over the structure of the different standard TPRs defined in a Type-2 petri-net.

Primitive regions: A primitive region consists of a simple place or transition, which is the entry as well as the exit of the region. Hence the theorem is trivially true for a primitive region. This provides the base case for the induction.

Series region: Let A and B be the regions that occur in a series region R. If A occurs first, then the entry of A is the entry of R, while the exit of B is the exit of R. The exit of A is connected to the entry of B by a simple edge in the petri-net, and the labelling scheme assigns the same label to both these elements. If the theorem is true for regions A and B, then the label assigned to the entries and exits of all the regions A, B and R are the same.

Branch region: The label assigned to every successor of a branch is the same as the branch itself, and the label assigned to a merge is the same as all its predecessors. Thus, if the theorem is true for all the child regions of the branch region, then it follows that the exit (branch or merge) is assigned the same label as the entry (branch or merge).

Fork region: We have already proven in Theorem 4.3.3 that the entry and the exit of a fork region have the same label, assuming that Theorem 4.3.4 is true for all regions. This results in a "recursive induction", which terminates at a fork region such that none of its descendants are fork regions.

Theorem 4.3.5 *The label assigned to the entry of a region is a prefix of all labels assigned to elements within that region.*

Proof:

We prove the theorem by induction on the construction of Type-2 standard regions as follows:

Primitive region: Since there is only one element in the region, which is the entry as well as the exit, the theorem is trivially true.

Branch or Series regions: In a branch or series region, the entries of the child regions are assigned the same label as the entry of the region itself. If the theorem is true for the constituent regions, then L is the prefix of any labels created in each of these regions.

Fork regions: Let L be the label assigned to the entry fork. The successors of this fork are assigned new labels created by extending L as described in the labelling scheme. These extensions disappear only at the exit of the region. Thus, the label assigned to the entry of a constituent region is an extension of L. If the theorem is true for the constituent regions, then all labels assigned within the fork region are extensions of the label L.

4.4 Testing labels for compatibility

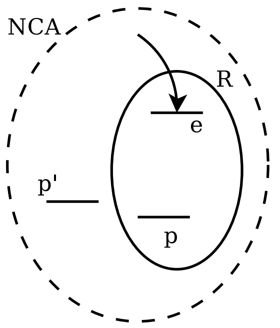
Given a labelled Type-2 petri-net, it is possible to exhaustively identify pairs and even sets of compatible operations. The label of a petri-net element records the effect of forks along paths reaching that element from the init transition. This information is sufficient to infer the nature of the NCA of two elements and the presence of paths passing through both the elements. This allows us to determine whether the two elements are compatible.

We first propose a definition of compatibility for labels, and in Theorem 4.4.1, we establish a relation between this and the hierarchy of regions in a Type-2 petri-net. Then in Theorem 4.4.2, we prove that the compatibility of two elements is identical to the compatibility of the labels assigned to them.

Definition 4.4.1 *Two labels L_1 and L_2 are compatible if and only if one of the following is true:*

1. $L_1 = \Phi$ *or* $L_2 = \Phi$
2. $\exists\, l_1 \in L_1, l_2 \in L_2$ *such that one of the following is true:*
   - (a) $l_1 \leq l_2$ *or* $l_2 \leq l_1$
   - (b) *If* $x = \mathrm{LCP}(l_1, l_2)$*, then there exist label elements* e_1 *and* e_2 *such that following statements are true:*
     - i. $x.e_1 \leq l_1$ *and* $x.e_2 \leq l_2$
     - ii. $e_1 = (f_1, k_1, i_1), e_2 = (f_2, k_2, i_2)$ *where* $f_1 \neq f_2$

Theorem 4.4.1 *Let p be an element in a region R in a Type-2 petri-net. Let e be the entry to the region, and p' be an element outside the region. The label assigned to p' is compatible with the label assigned to p, if and only if it is compatible with the label assigned to e.*



Proof:

Let L_p, L_e and $L_{p'}$ be the labels assigned to the three elements p, e and p' respectively. The label L_p is an extension of the label L_e, which can be written as $L_p = L_e * U$ for some set U that is a function of forks occurring in the region R. The set U is empty when all ancestors of p upto and including R are non-fork regions. If U is not empty, then sequences in U are unique to the region R and cannot occur in any label outside of R.

If L_p and L_e are not empty, then there exist sequences $l_e \in L_e$ and $l_p \in L_p$, such that $l_p = l_e.u$, for some u defined inside region R. Thus we have, $\text{LCP}(l_p, l_{p'}) = \text{LCP}(l_e, l_{p'})$.

Case 1: If $L_{p'}$ is compatible with L_p, either one of the two labels is empty, or there exists a sequence $l_{p'} \in L_{p'}$ that satisfies Definition 4.4.1. We have the following possibilities, along with their implications:

$$\begin{aligned} &L_{p'} = \Phi && \\ &L_p = \Phi &\implies& L_e = \Phi \\ &l_{p'} \leq l_p &\implies& l_{p'} \leq l_e \\ &l_p \leq l_{p'} &\implies& l_p = l_e \leq l_{p'} \end{aligned}$$

In each of these cases, it follows that $L_{p'}$ is also compatible with L_e.

Case 2: Similarly, if $L_{p'}$ is compatible with L_e, then we have the following possibilities:

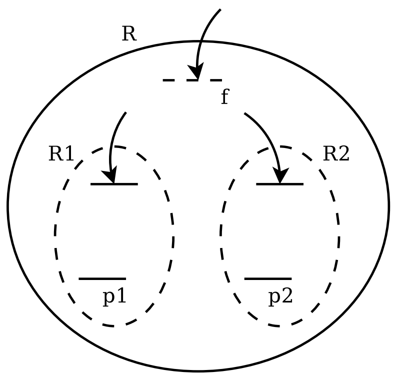
$$\begin{aligned} &L_{p'} = \Phi && \\ &L_e = \Phi &\implies& \text{LCP}(l_p, l_{p'}) = \phi \\ &l_{p'} \leq l_e &\implies& l_{p'} \leq l_p \\ &l_e \leq l_{p'} &\implies& \text{LCP}(l_p, l_{p'}) = l_e \end{aligned}$$

In each case, it follows that $L_{p'}$ is also compatible with L_p.

Theorem 4.4.2 *Two elements in a Type-2 petri-net are compatible if and only if the labels assigned to them are compatible.*

Proof:

Consider two elements p_1 and p_2 in a Type-2 petri-net, with labels L_1 and L_2 respectively. Let the region R be the NCA of the two elements. If p_1 is not a child of R, then there exists a region



R_1 such that $p_1 \sqsubset^* R_1 \sqsubset R$. This is the largest region inside R that contains p_1 but not p_2. From Theorem 4.1.1, p_2 is compatible with p_1, if and only if it is compatible with the entry of R_1. Similarly from Theorem 4.4.1, L_2 is compatible with L_1, if and only if it is compatible with the label assigned to the entry of R_1.

Hence, if R_1 exists, we can use the entry of R_1 in the place of p_1 in our proof, without affecting the outcome. We simply use the same name p_1 to refer to this entry, while L_1 refers to the label assigned to it. We also replace p_2 with the entry of a similar region R_2 if it exists.

Part 1: If the two elements are compatible, then from Definition 4.1.1, one of the following must be true:

1. R is not a fork region.
2. There is a path in R that passes through both p_1 and p_2.

Case 1: If R is not a fork region, then $L_1 = L_2$. Either $L_1 = L_2 = \Phi$, or there are pairs of sequences $l_1 \in L_1$ and $l_2 \in L_2$, such that $l_1 = l_2$. In either case, the labels assigned to the two elements are compatible.

Case 2: If R is a fork region, then there is a path P that reaches both elements. Without loss of generality, let p_1 be the element that occurs earlier on the path. Let f be the fork at the entry of region R, with fanout $k(f)$. Each sequence $x \in L$ is extended by the fork f as $x.\big(f, k(f), i\big)$. A number of forks may have occurred on the path P before reaching p_1, introducing additional elements in the label. If these forks are denoted as $g_1, g_2, \ldots$, then any sequence $l_1 \in L_1$ can be expressed as:

$$l_1 = x.\big(f, k(f), i_f\big).\big(g_1, k(g_1), i_{g_1}\big) \ldots$$

If p_1 is the entry of R_1, then the label L_1 is also assigned to the exit of R_1. Irrespective of the existence of R_1, the same label L_1 eventually reaches p_2 along path

P. But it may be modified by forks and joins that may occur on path P, before it reaches p_2 as follows:

1. Intervening forks extend the label, denoted by h_j.
2. Intervening joins reduce the label, removing elements from forks that may have occurred before p_1 (denoted by g_j) or after it (denoted by h_j).

A sequence l_2 that reaches p_2 from p_1 along the path P has the form:

$$l_2 = x.\big(f, k(f), i_f\big).\big(g_1, k(g_1), i_{g_1}\big) \dots \big(h_1, k(h_1), i_{h_1}\big) \dots$$

Note that the label element indicating fork f cannot be reduced by any join inside the region R, since the associate join occurs at the exit of the region. If l_2 is written as $l_2 = a.\big(h_1, k(h_1), i_{h_1}\big) \dots$, then a is a prefix of l_1 as well as l_2, such that the conditions for compatibility described in Definition 4.4.1 are satisfied. Hence, the two labels L_1 and L_2 are compatible.

Part 2: If the two labels L_1 and L_2 are compatible, then the theorem claims that the elements p_1 and p_2 are also compatible. To prove that, we prove the contra-positive instead: *If two elements are not compatible, then their labels are also not compatible.*

If p_1 and p_2 are not compatible, then R is a fork region, and there is no path from the entry of the region R, that reaches both elements. Since R is a fork region, neither label is empty. Let P_1 and P_2 be a pair of paths from the entry, that reach p_1 and p_2 respectively. The two paths must have diverged at some fork g (which could possibly be the entry of the region R). Hence any two sequences $x_1 \in L_1$ and $x_2 \in L_2$ can be written with a (possibly empty) common prefix a indicating the forks that occur along the common path upto g, as follows:

$$\begin{aligned} x_1 &= a.\big(g, k(g), i_1\big) \dots \\ x_2 &= a.\big(g, k(g), i_2\big) \dots \end{aligned}$$

Clearly, the sequences x_1 and x_2 do not satisfy the conditions in Definition 4.4.1. This is true for any pair of sequences taken from the sets L_1 and L_2, and hence the two labels are not compatible.

4.5 A compact graph-based representation of labels

The compatibility label is a record of every path reaching that element from the $init$ transition, which results in an exponential size. Comparing two labels for compatibility also has exponential complexity, since every sequence in one label has to be compared with every sequence in the other label. This complexity makes it infeasible to implement labels as just sets of sequences. Instead, we propose a graphical representation, where each label is represented by a node in the graph, as shown in Figure 4.7. Edges in the graph represent the construction of labels from other labels. Compatibility between two nodes is determined by the existence of a node that satisfies a specific condition. This representation has the following advantages:

1. The representation is compact, since the number of nodes corresponds to the number of distinct labels, which is less than the number of elements in the petri-net.

2. The complexity of the test for compatibility for two labels is close to linear, based on a depth first search that locates a node satisfying a specific condition.

3. If a node that satisfies the compatibility condition for one pair of labels, then it also identifies many more pairs of compatible labels. This reduces the effort needed for an exhaustive search for compatible pairs.

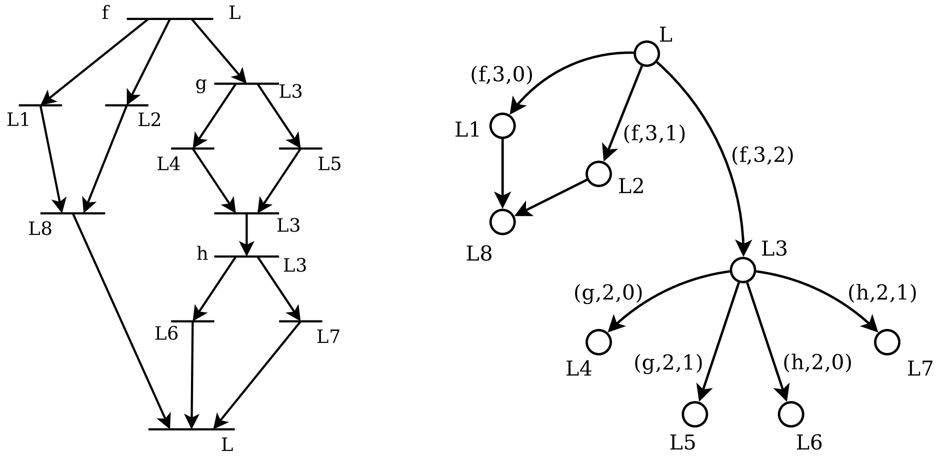


Figure 4.7: Label Representation Graph.

4.5.1 The label representation graph (LRG)

The label representation graph (LRG) is a directed acyclic graph $G = (N, E, r)$, where N is the set of nodes and E is the set of directed edges. The graph has a single root node $r \in N$.

The nodes in the graph represent labels assigned to petri-net elements. The root node represents the empty label Φ. Edges are associated with label elements used in the construction of labels. Let $l(n)$ be the label represented by a node $n \in N$, and $l(e)$ be a label element associated with an edge $e \in E$. An edge $e = (u, v) \in E$ in the graph indicates that the label $l(v)$ is derived from the label $l(u)$ in some manner, using $l(e)$.

The label element at an edge can be empty, represented by the symbol ϕ. An edge $e \in E$ is said to be *labelled* if and only if $l(e) \neq \phi$. A labelled edge in the graph represents the product operation. For a labelled edge $e = (u, v)$ in the LRG, $l(v) = l(u) * l(e)$. On the other hand, if multiple incoming edges are incident on a node $n \in N$, then $l(n)$ is the union of the labels represented by the tails of those edges. In a *well-formed* LRG, all the incoming edges at a node are unlabelled, if and only if there are multiple incoming edges at that node. An LRG may represent a valid labelling scheme only if it is well-formed.

The LRG serves as a compact representation of all the labels created by labelling a Type-2 petri-net. Any path in the LRG starting from the root node r to a node $n \in N$ contributes one sequence in the label $l(n)$. The actual label $l(n)$ is the set of sequences corresponding to all paths that reach n from r.

4.6 Construction of the LRG

Initially, the LRG for a Type-2 petri-net is empty. The labelling of a Type-2 petri-net begins by assigning the empty label (Φ) to the init transition. This label is represented as a new node in the LRG, which becomes the root of the graph. As labelling proceeds, if an element is assigned the same label as its predecessor, that label is already represented in the LRG. New nodes and edges are introduced in the LRG for two cases as follows:

4.6.1 Labelling the successors of a fork

For a fork f with label L, each of its successors is assigned a new label $L_i = L * \big(f, k(f), i\big)$ where $0 < i \leq k(f)$. Let u be the node in the LRG, that corresponds to the label L. Then for

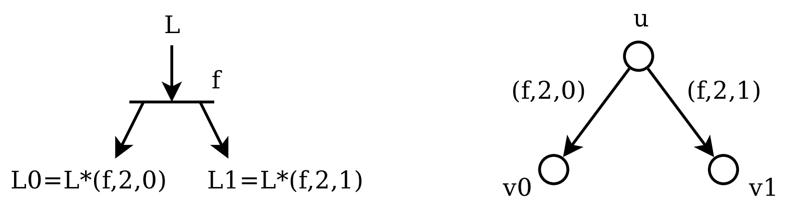


Figure 4.8: Labelling at a fork.

each new label L_i derived from L, a new node v_i is added to the graph. Correspondingly, a new labelled edge (u, v_i) is added to the graph, labelled with the element $\big(f, k(f), i\big)$.

4.6.2 Labelling a join

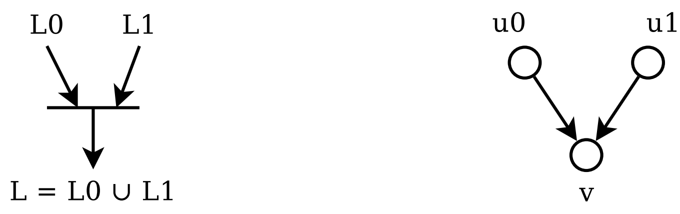


Figure 4.9: Labelling at a join.

When a join is encountered in the petri-net, it is assigned a label L that is the union of the labels assigned to its predecessors. Correspondingly, a new node v is created in the LRG. Let L_i be the label assigned to the i^{th} predecessor, represented as node u_i in the LRG. For each such node, a new unlabelled edge (u_i, v) is created in the graph.

Sequences of unlabelled edges

The construction of a join is the only step that introduces unlabelled edges in the graph. This may result in sequences of unlabelled edges, which are inconvenient to handle in operations that are described in later sections. Hence the introduction of a new join in the LRG is followed by an operation that eliminates sequences of unlabelled edges of length two or more.

At the newly created node v, let (u, v) be an unlabelled edge incident on v. If the node u is a join, then the incoming edges at u are unlabelled. For each edge (t_i, u), we create an unlabelled edge (t_i, v) and then remove the edge (u, v). Since this elimination is performed for every new node, the node t_i cannot have an incoming unlabelled edge. Such an unlabelled sequence of length two would have been eliminated when the join u was created.

Reductions at a join

If a new join is the associate join of some fork f, then label assigned to it must be reduced by eliminating label elements that indicate the fork f. This reduction is implemented in the LRG as follows.

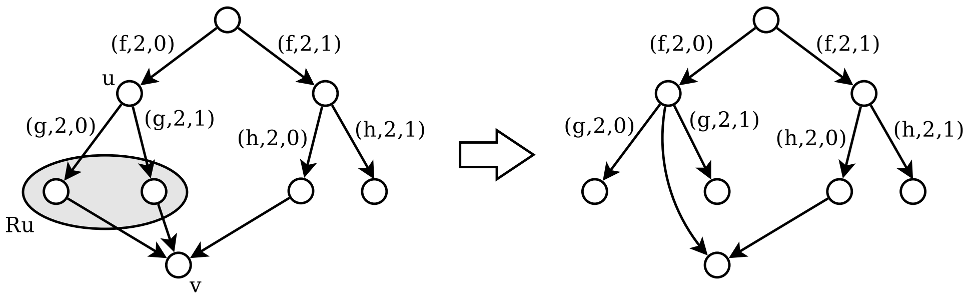


Figure 4.10: Reductions at a join.

Let L be the label assigned to the join, and represented by node v in the LRG. If $W = U{*}V$ is a reducible subset of L, then L is reduced by replacing the subset W with U. W is actually the union of labels represented by a subset R_u of the predecessors of the join. U corresponds to a node u, such that for each node $w_i \in R$, (u, w_i) is a labelled edge in the LRG. The reduction is equivalent to replacing these labelled edges with a single unlabelled edge (u, v).

The reduction at a new join node is implemented by partitioning its predecessors into subsets that have a common parent, such as set R_u in Figure 4.10. Every member w_i of such a subset has a single incoming labelled edge (u, w_i) from the common parent node u. The incoming edge cannot be unlabelled since edge (w_i, v) is unlabelled, and the construction of the LRG does not allow a sequence of unlabelled edges. The labels on the edges are of the form $\big(f, k(f), i\big)$, for some fork f. If $|R(u)| = k(f)$, then the labels corresponding to the nodes in $R(u)$ form a reducible subset in L. We reduce this set as follows:

1. for each $r_i \in R(u)$, remove the edge (r_i, v)
2. create an unlabelled edge (u, v) in E

After each reduction, we recompute the set of predecessor for the node v, and attempt further reduction. If at the end of all reductions, a single unlabelled incoming edge (u, v) remains incident on the new label v, then we eliminate the label v itself and assign u as the label of the join.

4.6.3 Uniqueness of nodes

Theorem 4.6.1 *There exists a one-to-one correspondence between the set of labels used in labelling the elements of a Type-2 petri-net and the set of nodes in the corresponding LRG.*

Proof:

Labelling at a fork: The labelling scheme creates a new node in the LRG whenever a fork is encountered. Each fork is associated with a unique identifier when its successors are labelled. Thus, the labels created by extending the label at a fork are guaranteed to be unique.

Labelling at a join: A new node may be created when a join is encountered, unless the node is removed due to a reduction. But a join in the petri-net is not associated with any unique identifier. Labelling at a join can potentially generate a label that has already been assigned to some other join. If a new node is created in the graph for this join, then the graph contains two nodes that represent the same label. We show that this is not possible, as follows.

Consider two joins j_1 and j_2 that are assigned the same label. We consider the two cases where the NCA region $R = \mathrm{NCA}(j_1, j_2)$ is either a fork region or not.

Case 1: If R is not a fork region, then each join lies in a distinct fork region. Consider the two regions $R_1 \sqsubset^* R$ and $R_2 \sqsubset^* R$, that are the parents of j_1 and j_2 respectively. All the labels assigned in a fork region contain label elements that are unique to that region. Only the entry fork and the exit join have labels that do not contain elements unique to that fork region. Thus, for j_1 and j_2 to have the same label, they must be the exit joins of their respective regions.

Let f_1 and f_2 be the entry forks of these two fork regions. Clearly, $\mathrm{NCA}(f_1, f_2) = \mathrm{NCA}(j_1, j_2) = R$. The forks f_1 and f_2 are assigned the same label as j_1 and j_2, and in fact, this is the label assigned to the entry of R. The node in the LRG that represents this label is reused when labelling the two forks. Since j_1 and j_2 are their associate joins respectively, no new node is created when labelling them, as seen in Section 4.6.2.

Case 2: If R is a fork region, the two joins j_1 and j_2 must lie on the same path from the entry of the NCA. To prove that, assume that no such path exists. Consider a pair of paths P_1 and P_2 that reach j_1 and j_2 respectively. Since the two paths must have diverged at some fork in R, each path has labels extended by a unique label element. Hence if the assumption is true, the two joins cannot have the same label.

Let P be the path from the entry fork that passes through both j_1 and j_2 and let j_1 be the join that occurs first on the path. If there was a path P' from the entry that passed through j_2 but not through j_1, then the label assigned to j_2 would contain sequences that are not present in the label assigned to j_1. Thus every path that reaches j_2 must pass through j_1 or, in other words, the all the paths that converge at j_2 must have diverged from path P itself at forks that occurs after j_1. There exists a fork f' on this path which is the associate fork of join j_2. This fork is assigned the same label as j_1, reusing the corresponding node in the LRG. Similarly, j_2 is assigned the same label as f', using the same node in the LRG. Thus no new node is created in this case as well.

4.7 Compatibility using the LRG

Definition 4.7.1 *Two nodes u and v in the LRG are said to be compatible, if only if one of the following is true:*

1. *There is a path from u to v or* vice versa.
2. *There exists a node a in the LRG, from which u and v are reachable along non-intersecting paths such that one of the following is true:*
   (a) *One or both paths begin with an unlabelled edge.*
   (b) *The labels on the first edges in the paths indicate different forks.*

Theorem 4.7.1 *Two labels L_1 and L_2 are compatible if and only if the corresponding nodes u and v in the LRG are compatible.*

Proof:

Part 1: If the two labels are compatible, then there exist sequences $l_1 \in L_1$ and $l_2 \in L_2$, that satisfy the conditions for compatibility of labels, stated in Definition 4.4.1.

Case 1: $L_1 = \Phi$ or $L_2 = \Phi$

If either label is empty, then the corresponding node is the root node, and there must be a path that reaches the other node, and hence they are compatible.

Case 2: $l_1 \leq l_2$

Let n be the node at which l_1 was created as an extension due to a fork. If the entire label L_1 was created at n itself, then $u = n$. Else, u is a union node such that there is an unlabelled edge (n, u) in the LRG. If $l_1 = l_2$, then by the same logic, either $v = n$, or there exists an unlabelled edge (n, v) in the LRG. If $l_1 < l_2$, then a path with at least one labelled edge exists from n to v. If $u = n$, then there is a path from u to v, else n is the node a described in Definition 4.7.1. A similar deduction can be made in the case where $l_2 \leq l_1$. In either case, the two nodes are compatible.

Case 3: $x.e_1 \leq l_1$ and $x.e_2 \leq l_2$, where e_1 and e_2 indicate different forks.

Let n represent the node where x is created, possibly as the extension of some existing sequence. If x is empty, then n is the root node. The label elements e_1 and e_2 represent distinct labelled outgoing edges at node n. The nodes u and v respectively are reachable from n along paths that start with these edges. Thus, n is the node a described in Definition 4.7.1, and the two nodes are compatible.

Part 2: If the two nodes u and v are compatible, we show that the corresponding labels L_1 and L_2 are also compatible.

Case 1: v is reachable from u along some path in the LRG. If u is the root node, then L_1 and L_2 are compatible.

If there is any labelled edge on the path, then for every sequence $l_1 \in L_1$, there exists a label $l_2 \in L_2$ which is an extension of l_1. In the absence of any labelled edges, l_2 is the same as l_1. In either case, the two labels are also compatible.

Case 2: There exists a node a from which u and v are reachable along non-intersecting paths as defined in Definition 4.7.1. If L is the label at a, for every sequence $l \in L$, there exist sequences $l_1 \in L_1$ and $l_2 \in L_2$, such that $l \leq l_1$ and $l \leq l_2$.

If u is reachable from a along an unlabelled path, then that path consists of a single unlabelled edge. The corresponding label L_1 is a union that contains L, and hence $l_1 = l$. Similarly, if v is reachable along an unlabelled path, then $l_2 = l$.

If the two paths start with edges labelled e_1 and e_2, we have $l_1 = l.e_1$ and $l_2 = l.e_2$, where e_1 and e_2 indicate different forks.

Thus, in all cases, the labels are compatible.

4.7.1 Testing for compatibility

Given a node u and a node v in the LRG, they are trivially compatible if $u = v$. Otherwise, we determine compatibility by traversing the graph from each node towards the root in a breadth-first manner using a queue. Let the traversal begin with u. For each node m, we maintain a list $e_u(m)$ that contains outgoing edges at m, such that there is a path from m to u along each edge in $e_u(m)$. Initially, all the nodes are white. The queue is initialised by adding node u, and colouring it black. While the queue is not empty, we remove one node m from the queue, and perform the following steps at each predecessor p:

1. If $p = v$, the two nodes are compatible. Return this result.
2. Add the edge (p, m) to the list $e_u(p)$.
3. If p is white, add it to the queue, and colour it black.

When the queue is empty, we restart the breadth-first traversal from the node v. The colour of all nodes is reset to white and node v is coloured black. While the queue is not empty, we remove one node m from the queue, and perform the following steps at each predecessor p:

1. If $p = u$, the two nodes are compatible. Return this result.
2. Compare the edge (p, m) with each edge in $e_u(p)$. If one or both edges are unlabelled, or the labels indicate different forks, then p is the ancestor node a mentioned in Definition 4.7.1 and the two nodes are compatible. Return this result.
3. If p is white, add it to the queue, and colour it black.
4. If the traversal continues until the queue is empty, then the two nodes are not compatible. Return this result.

4.7.2 Identifying sets of compatible operations

The labelling scheme is a complete representation of the compatibility of operations in a Type-2 petri-net. A label L represents a set of operations that are all compatible with each other, and it is possible to find labels that represent other sets of operations that are compatible with this set.

The compatibility of labels can be represented as edges in a compatibility graph where nodes represent labels. A clique in this compatibility graph represents a set of labels that are mutually compatible. But an exhaustive search of the LRG to build such a compatibility graph is not feasible. Even after building such a graph, the implementation must also build a clique cover for it. Instead, we use the structure of the LRG in a simple greedy algorithm to generate a clique cover.

The definition of compatibility in the LRG is based on the existence of a node with outgoing edges, such that at least one of the edges is unlabelled, or the labels indicate different forks. All the nodes reachable from one edge are pair-wise incompatible with all the nodes reachable from the other edge. In addition, all the nodes reachable from a node are compatible with that node. This property can be used to build up a clique cover.

We perform a postorder traversal of the LRG, possibly creating a new clique when visiting a node. Let $S(n)$ represent the set of candidate cliques at a node n. For every node n, $S(n)$ is initially empty. We visit each n as follows:

1. If n is a leaf, create a new clique C, and insert n in it. Insert C in the set $S(n)$, and return.

2. For an internal node n, create a set of candidate nodes containing all the successors of n. Extract pairs of candidates m_1 and m_2, such that the edges (n, m_1) and (n, m_2) satisfy the conditions for compatibility. Each clique in $S(m_1)$ is compatible with each clique in $S(m_2)$. Extract pairs of cliques $C_1 \in S(m_1)$ and $C_2 \in S(m_2)$, and create a new clique $C = C_1 \cup C_2$ for each pair. Insert this clique in $S(n)$. Extract all the remaining cliques in $S(m_1)$ and $S(m_2)$ and insert them in $S(n)$.

3. When all candidate pairs are exhausted, extract the remaining candidates one at a time. For each candidate m, extract all the cliques in $S(m)$ and insert them in $S(n)$.

   At this point, the set $S(m)$ for each successor m is empty. Promoting all the cliques from $S(m)$ to $S(n)$ ensures that these cliques are considered as candidates when visiting a predecessor of n. The cliques are removed from $S(m)$ to ensure that they are not reused when m is visited by some other predecessor.

4. Select a clique $C \in S(n)$, and insert n in it. If $S(n)$ is empty, then create a new clique C and insert n in it.

In Figure 4.11, we show a possible cover that would be generated for a given LRG. Note that the algorithm does not mention any specific criteria when choosing cliques in step (2).

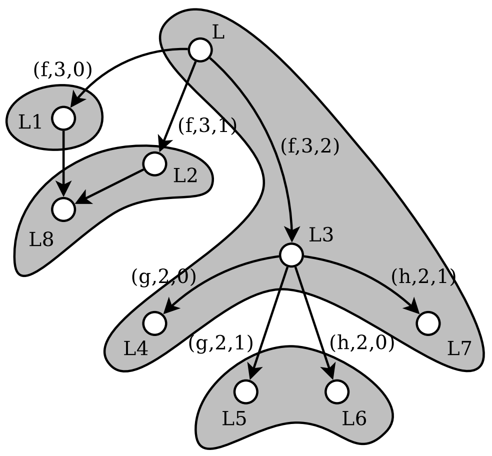


Figure 4.11: A cover generated for the example LRG.

These criteria can be based on external inputs, possibly including feedback from the later implementation stages. The algorithm utilises every available opportunity to create larger cliques from existing cliques. External factors can also influence this behaviour, since in some cases it is possible that limiting the size of a clique could lead to a better implementation.

4.8 Summary

This chapter presents an effective demonstration of the ease with which an AHIR specification can be analysed. The optimisation presented exploits the structure of the Type-2 petri-net to exhaustively identify opportunities for sharing data-path operators without the need for arbitration. Such pairs of operations are termed as "compatible operations".

We define compatibility in terms of paths in the Type-2 petri-net, combined with the hierarchy of regions. A direct use of this definition to determine compatibility results in exponential complexity. Instead we introduce a labelling scheme built around a structure called the label representation graph (LRG). This structure supports a linear-time algorithm for identifying compatible operations. We use a greedy algorithm that assigns hardware resources to sets of mutually compatible operations based on the label representation graph.

Chapter 5

Implementation and Results

We have built an integrated flow that translates C programs to synthesisable VHDL using AHIR as an intermediate step. This combined flow can be divided into three stages — software optimisation, generation of an AHIR specification, and implementation in hardware. The software optimisation is handled by the LLVM compiler framework. The input C program is compiled to the internal LLVM representation, and then optimised. The optimised program is then translated to an AHIR specification by a customised LLVM back-end. In Appendix A, we describe the practical details of implementing such a flow.

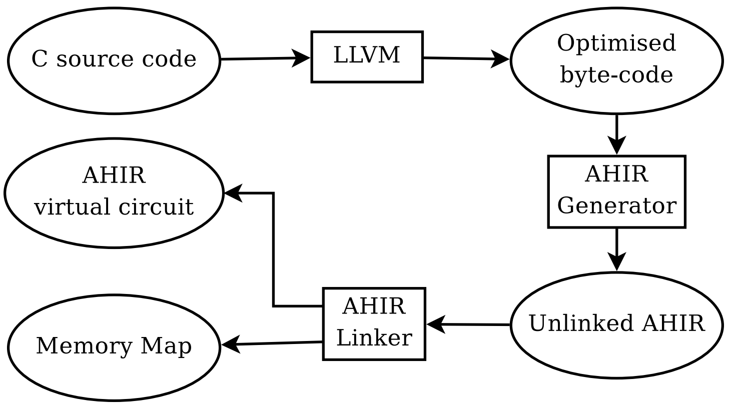


Figure 5.1: Generating AHIR from a C program.

In Figure 5.1, we show the translation process from a C program to a complete AHIR specification. The LLVM back-end generates an *unlinked* circuit in AHIR. This circuit contains one module for each function in the C program, but inter-module communication is absent. The AHIR linker then generates an inter-module link layer based on the function calls in the program along with a memory map that describes memory locations hard-coded in the circuit.

5.1 Support for the C language

The current implementation supports a practical subset of the C language, although there is no fundamental restriction on supporting the entire C language. The following high-level features are available in the tool-chain:

Scalar data-types: Scalar data-types that have a width upto 32 bits are fully supported. This includes bool (single bit), char (8 bits), unsigned char, int (16 bits), unsigned int, long (32-bits), unsigned long and single precision floats (32 bits). Larger data-types can be supported, but the current tool-chain cannot produce the multiple loads and stores required to access these data-types from external memory.

Pointers, structures and arrays: The AHIR tool-chain simply treats pointers as 32-bit unsigned integer values. As a result, address generation instructions in the LLVM IR are mapped to simple arithmetic operators in an AHIR data-path. Arrays, structures and their combinations are all completely supported.

Dynamic memory allocation: Dynamic memory allocation is usually represented by the malloc() system call. Since there is no external "operating system" in AHIR, memory allocation is directly handled by the program itself. The program can use a suitable implementation of malloc() as along as it guarantees that if the function returns a non-NULL address then the address returned is actually usable for storage in the external memory subsystem.

The following features were not implemented purely for practical reasons:

Recursion: Recursion requires a stack to store the current state of a function. This stack is limited by two factors: the amount of storage available for the stack, and the ease of accessing the state of the function. If the stack is implemented in external memory, its size is only limited by the width of the memory addresses, which is the same in AHIR as in a microprocessor. But the state of an AHIR module is distributed throughout the elements in the data-path and the control-path, which makes it particularly inaccessible. As a result, the current implementation does not support recursion.

Variable arguments: Most programming languages allow a program to have a function call where the number and type of arguments is not known at compile time. AHIR can support this with an appropriate mechanism for packaging arguments when passed through the inter-module link layer, but this is not implemented yet.

Function pointers: Function pointers allow a program to have a function call where the identity of the invoked function is not known at compile time. The function pointer can be used as a unique identifier of the function when a call is routed through the inter-module link layer. The current implementation does not support this due to practical limitations, although function pointers play an important role in creating large programs.

Library calls: Most applications are built on existing libraries that provide commonly used features in the form of an API. Software programs use these libraries in two forms: as statically linked, where the relevant functions are copied into the body of the program, or dynamically linked, where the library functions are made available when required. The current implementation can only support static libraries since the compiler must have access to every function in the program when generating the AHIR specification.

I/O ports: The concept of I/O ports in a program is usually implemented as a set of functions that have side-effects that are hidden from the program. An example is the standard input-output library in C which provides functions such as printf() and scanf() for exchanging data with the outside world. Similar I/O facilities can be made available in an AHIR circuit by mapping the standard I/O functions to specialised operators that provide a suitable implementation of the interaction specified by these functions.

5.2 Simulation and Synthesis

The AHIR tool-chain can generate two different implementations of the input specification: a SystemC model and a synthesisable VHDL circuit description as shown in Figure 5.2. The output of the SystemC simulation is a trace file that tracks all events in the circuits including request and acknowledge symbols, and load/store operations. This memory access trace has been used in a related project[22] that investigates the nature of memory accesses generated by different programs.

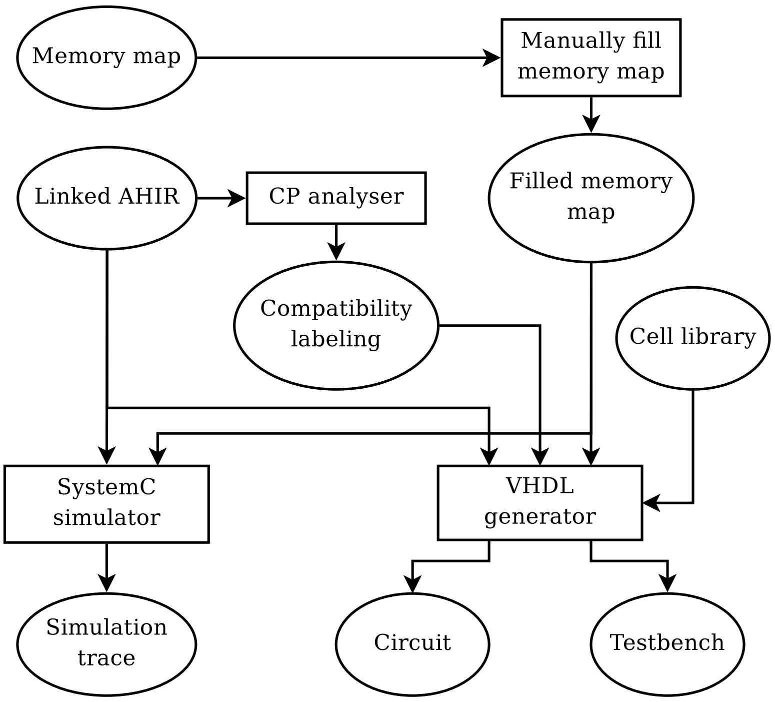


Figure 5.2: Generating SystemC and VHDL descriptions.

5.2.1 Synthesisable VHDL implementation

The VHDL generator converts an AHIR specification into a synthesisable VHDL description. The generator creates one entity for every control path, data path and link layer in the AHIR specification. These entities instantiate components described in a separate cell library. The generator also creates a test-bench using the memory map provided by the linker. In Appendix B, we list an example that starts with a simple C program and results in a set of VHDL files that describe a hardware implementation of the input C program.

The VHDL implementation is a clocked circuit, where the control-path is a Mealy machine and the data-path is a Moore machine. Each operator in the data-path is mapped to a clocked entity provided by the cell library. Transitions in the control-path are converted to combinational logic, while places are translated to clocked latches. These latches use a bypass mechanism to implement a zero-delay change in state.

5.2.2 Evaluation of generated circuits

The high-level compiler generates an RTL description in VHDL, that can either be mapped to an FPGA or an ASIC. We have tested the generated circuits on both these platforms, using a number of C program as representatives of various application domains:

- Linpack (high-performance computing)
- Red-Black Trees (high-level data-structures)
- FFT (digital signal processing)
- A5/1 (cryptography - stream cipher)
- AES (cryptography - block cipher)

5.3 FPGA implementation

In the first set of experiments, we compare the performance of FPGA implementations generated by the compiler with two extremes — equivalent programs running on a microprocessor (Intel Pentium IV running at 2.4GHz), and hand-crafted circuits that implement the same behaviour (results reported in the literature). The VHDL descriptions were mapped to the Xilinx Virtex-II (part number xc2v8000-5ff1152) using the Xilinx ISE software kit.

5.3.1 Performance

We use the following metric to compare the performance of the three different platforms for each program:

$$\mathrm{performance} = \mathrm{throughput} \,/\, \mathrm{area}$$

Throughput is defined as the number of tasks completed in one second. The notion of a task is specific to each program. For example, in Linpack, the factorisation and solution of one matrix is considered one task. Hence we define the throughput of a Linpack implementation as the number of matrices factorised in one second. Area is measured in terms of the number of equivalent gates. The circuits used in the experiments are generated after enabling arbiterless sharing in order to reduce the area.

Note: The following approximations were used when comparing the generated circuits with the Pentium IV processor:

1. The reported size of the Intel Pentium IV is 42 million transistors[23]. Assuming an average ratio of 4 transistors per gate, this number is considered to be equivalent to 10 million gates.

2. The size of an FPGA implementation is measured in terms of the slices used. We have assumed an average ratio of 5 gates per slice[24] in estimating the size of the circuit in terms of gates.

5.3.2 Static sharing of hardware

The experiments also demonstrate the effect of sharing hardware using the optimisation described in Chapter 4. We use the same metric described in the previous section for evaluating the performance of the circuit. Since we are comparing two versions of the same circuit, a simpler definition of the metric can be used:

$$\text{performance} = \text{frequency} \,/\, \text{slices}$$

5.3.3 A note on the implementation

The hardware circuits generated by our synthesiser are constructed from a predefined cell library. The library incorporates a number of simplifications as follows:

1. Every operation in the data-path takes one clock cycle. As a result, a floating-point multiplication takes the same time as a boolean function.

2. Operators for the integer type have been implemented with a fixed width of 32 bits. Types with smaller widths are padded, while types with larger widths are not supported.

3. The memory subsystem is single-ported. All the load-store operators defined by the circuit are connected to the memory through an arbiter. Also, the current compilation process serialises all memory operations, as described in Section 3.3.3.

These simplifications partially account for the suboptimal results in our experiments. A better implementation of the library components can improve the results in three areas: operating frequency, latency in terms of clock cycles, and the accuracy of the computations.

5.3.4 Results

We present the measurements for each experiment in the form of two tables. The first table shows the effect of arbiterless sharing on the size and operating frequency of the circuit. The second table compares the performance of the AHIR circuit with the Pentium-IV, and also with hand-crafted circuits in some cases. In the first table, the following observations are noteworthy:

1. When sharing is enabled, the actual size of the circuit is always larger than the estimated size of the circuit. This indicates the increased complexity in routing since the number of wires connected to an operator increases when it is shared.

2. For some programs, the operating frequency of the circuit improves considerably when sharing is enabled. This is likely to be the result of a reduction in the number of load/store operators, which reduces the delays in the memory access arbiter.

Linpack The Linpack benchmark[25] factorises a matrix using Gauss elimination and then solves a system of equations using those factors. We measure performance by factorising a 100×100 single-precision matrix. The throughput is the inverse of the time taken to factorise and solve one matrix. The Linpack program is designed to maximise the use of the processor cache. Hence the performance results for Linpack are especially relevant for a comparison between an AHIR circuit and the Pentium-IV.

	Slices		Frequency	Performance
	Post Synthesis	Post PAR	(MHz)	(Hz/slice)
w/o sharing	17710	17300	20.861	1205
w/ sharing	10459	12999	21.365	1643
gain (%)	41	25	2	36

Table 5.1: Synthesis results for Linpack.

	AHIR	P-IV
cycles	4006k	6870k
time (ms)	188	2.86
gates	65k	10M
throughput	5.319	350
throughput / area	8.18×10^{-5}	3.50×10^{-5}
AHIR is	-	2.34× better

Table 5.2: Performance comparison for Linpack.

Red-Black Trees The source for implementing a Red-Black tree was obtained from the project `libredblack` [26] available under the LGPL. The version used supports deletion, insertion and searching in a Red-Black tree. The source has been modified to use a simple built-in memory manager instead of the `malloc()` system call. New nodes are assigned from a pre-allocated array of nodes, whose size can be modified at compile time through a macro. Performance was measured by inserting 1000 distinct nodes into an empty tree. The throughput is the inverse of the total time taken for these 1000 insertions.

	Slices		Frequency	Performance
	Post Synthesis	Post PAR	(MHz)	(Hz/slice)
w/o sharing	9969	10659	20.360	1910
w/ sharing	6098	7958	44.189	5552
gain (%)	39	25	117	191

Table 5.3: Synthesis results for Red-Black Trees.

	AHIR	P-IV
cycles	389k	590k
time (ms)	8.8	0.246
gates	40k	10M
throughput	114	4065
throughput / area	2.85×10^{-3}	0.4065×10^{-3}
AHIR is	-	7× better

Table 5.4: Performance comparison for Red-Black Trees.

FFT The FFT implementation was based on a program provided in *Numerical Recipes in C*[27]. The program is independent of the radix of the FFT, which is supplied as an argument. We tested this circuit for a 64-point FFT performed on a complex signal. The throughput measured is the inverse of the time taken to complete the 64-point FFT.

	Slices		Frequency	Performance
	Post Synthesis	Post PAR	(MHz)	(Hz/slice)
w/o sharing	15794	14960	21.429	1432
w/ sharing	6906	7466	21.046	2818
gain (%)	56	50	-2	97

Table 5.5: Synthesis results for FFT.

	AHIR	P-IV
cycles	8k	41k
time (μs)	380	17
gates	37k	10M
throughput	2631	58824
throughput / area	0.0711	0.00588
AHIR is	-	12× better

Table 5.6: Performance comparison for 64-point FFT.

A5/1 Stream Cipher The A5/1 stream cipher was implemented by Prakalp et al.[28]. The throughput in this case is the number of bits produced per second in the keystream. This is equal to the inverse of the time required for generating one bit of the keystream. The results for the A5/1 implementation are especially relevant for a comparison with a hand-crafted circuit. There is no scope for architectural variations in the hand-coded implementation, since it consists of just three LFSRs along with some combinational logic. This provides a clear target for comparison and makes it easy to identify space and time overheads incurred by the automated design flow.

	Slices		Frequency	Performance
	Post Synthesis	Post PAR	(MHz)	(Hz/slice)
w/o sharing	1684	1701	65.351	38419
w/ sharing	1516	1926	85.346	44312
gain (%)	10	-13	31	15

Table 5.7: Synthesis results for A5/1 stream cipher.

	AHIR	RTL[29]	P-IV
cycles	21	1	234
time (μs)	0.246	0.0053	0.0975
gates	9630	160	10M
throughput (Mbps)	4.065	188.3	10.26
throughput / area	0.422×10^3	1.177×10^6	1.026
AHIR is	-	2800× worse	520× better

Table 5.8: Performance comparison for A5/1 stream cipher.

Note that the literature for the hand-crafted implementation reports only the hardware required for the computational core. The hardware involved in moving data to and from the core is not mentioned. Assuming a 3× overhead for this, AHIR is approximately 900× worse than the hand-coded version.

The number of cycles used by the microprocessor is larger than those required by the AHIR circuit by an order of magnitude. The circuit was able to execute a number of operations in parallel during most calculations. This also accounts for the relatively small gains obtained when sharing hardware.

AES The AES encryption program was implemented by Prakalp et al[28]. It uses a round-based architecture with an 8-bit data-path. The results only report the hardware required for encryption and not for decryption.

	Slices		Frequency	Performance
	Post Synthesis	Post PAR	(MHz)	(Hz/slice)
w/o sharing	4906	5760	29.249	5077
w/ sharing	4302	5348	61.372	11475
gain (%)	12	7	110	126

Table 5.9: Synthesis results for AES block cipher.

	AHIR	RTL[30]	P-IV
cycles	30.6k	-	32k
time (ms)	0.5	-	0.0133
throughput (Mbps)	0.256	2.2	9.6
gates	26k	620	10M
throughput / area	9.85	3550	0.96
AHIR is	-	360× worse	10× better

Table 5.10: Performance comparison for AES block cipher.

Performance comparison

In Figure 5.3, we compare the three implementations — a desktop processor, automatically generated circuits and hand-crafted circuits — in terms of throughput per area. For each program, the automatically generated circuit is better than the desktop processor by an order of magnitude. But the performance is less than that of the hand-crafted circuits by two or three orders of magnitude.

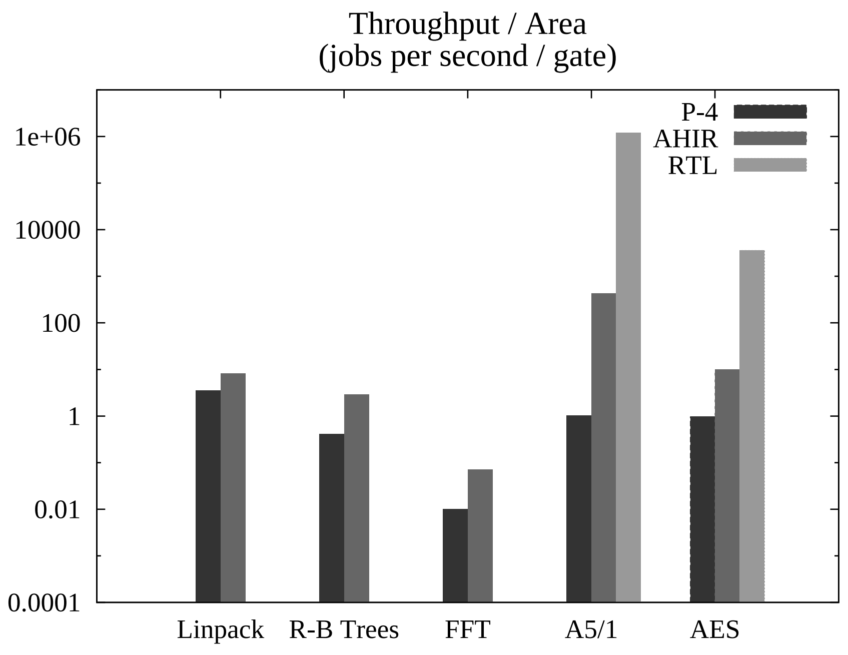


Figure 5.3: Performance comparison for the FPGA implementation

5.4 ASIC implementation

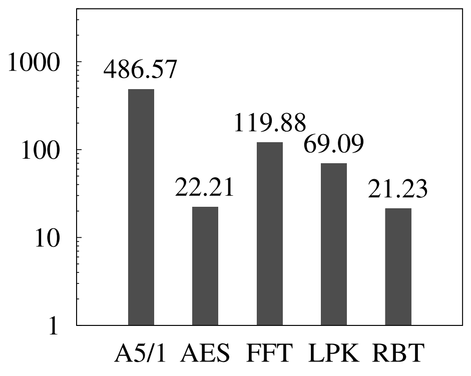
In the second set of experiments, we compare an ASIC implementation of each program with an embedded low-power microprocessor — the Intel Atom N270. The ASIC implementation is produced by a completely automated C-to-ASIC tool-flow — the VHDL description is first compiled to a netlist by Synopsys Design Compiler, which is then translated to a layout using Cadence Encounter. The flow targets the TSMC 180nm technology, using the OSU standard cell library and CACTI models for SRAM[31]. The synthesis flow has the following enhancements compared to the one used in the previous set of experiments:

- A memory subsystem that supports multiple load/store ports.
- Load/store operations may take multiple clock cycles once they are accepted by the memory subsystem.
- The ability to schedule concurrent load/store operations in the control-path based on a static memory reference analysis of the input program.

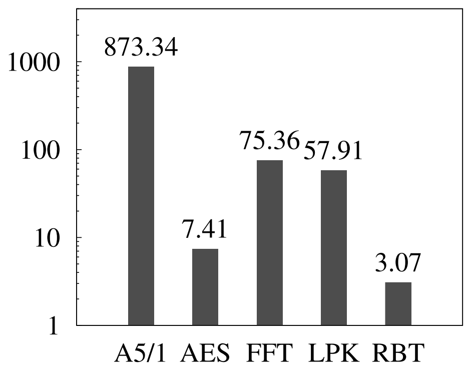
We present a comparison in terms of two performance parameters — energy dissipated in completing one task, and the product of the energy dissipated and the time taken to complete one task. The same set of C programs was run on the Intel Atom, and also translated to hardware: A5/1 stream cipher, AES block cipher, 64-point FFT, Linpack and Red-Black Trees. The values for area, frequency and power dissipation for the Atom processor were obtained from the corresponding datasheet. The measurements for the AHIR circuits were scaled from the 180nm data to match the 45nm technology used in the Atom processor.

Table 5.11: Comparison with the Intel Atom N270

	Area (mm^2)	Freq (MHz)	Delay (ms)	Power (mW)	Energy (μJ)	E$\times$D
A5/1-Atom	25	1600	0.12μs	2500	298.44 nJ	35.63
A5/1-AHIR	0.10	285	0.07μs	9.22	0.61 nJ	0.04
AES-Atom	25	1600	0.036	2500	89.362	3.194
AES-AHIR	0.41	285	0.107	37.56	4.023	0.431
FFT-Atom	25	1600	0.022	2500	55.64	1.238
FFT-AHIR	0.32	180	0.035	13.11	0.464	0.016
LPK-Atom	25	1600	7.90	2500	19740	155875
LPK-AHIR	1.69	165	9.42	30.33	285	2691
RBT-Atom	25	1600	0.36	2500	891.89	318.19
RBT-AHIR	1.13	165	2.47	17.00	42.01	103.80



(a) Energy $\left(\frac{\text{Atom}}{\text{AHIR}}\right)$



(b) Energy $\times$ Delay $\left(\frac{\text{Atom}}{\text{AHIR}}\right)$

Figure 5.4: Comparison of AHIR circuits with the Intel Atom N270

In Table 5.11, we show the results of experiments that compared the generated circuits with the Intel Atom N270 processor. The energy used for completing a job is equivalent to the throughput achieved for each watt of power supplied, commonly termed as "performance per watt". The graphs in Fig. 5.4 show the ratio of the performance of the AHIR circuit to that of the Intel Atom for each program. From the results, it is clear that the AHIR circuits are better than the Intel Atom by at least an order of magnitude in terms of energy dissipation.

5.5 Inference

The experiments demonstrate that AHIR can be used in translating programs from diverse application domains to hardware circuits. The use of arbiterless sharing is effective in reducing circuit area, inspite of a simple greedy algorithm that does not consider hardware overheads. Circuits generated in AHIR are competitive with the same programs running on a microprocessor, but the performance is less than that of hand-crafted circuits by two to three orders of magnitude. This is accounted for by three factors — the lack of support for expressing parallelism in C, bottlenecks in the memory subsystem and inefficiencies in the cell library used. Future work should be directed towards resolving these issues. Native optimisations in AHIR will also lead to significant improvements.

Chapter 6

Conclusion and Future Work

Our work on high-level synthesis was aimed at exploring the design of a compiler flow that meets the following requirements:

1. Allow programmers to create executable specifications using existing software practises.

2. Guarantee a correct implementation.

3. Enable the design of optimisations that can scale with the size of the system.

We have presented an approach that achieves these goals by decoupling the software compilation phase from the hardware compilation phase in the hardware synthesis flow. Our approach uses an intermediate representation called AHIR to implement this decoupling.

AHIR sufficiently encapsulates the low-level details of hardware implementations to be easily targeted by a software compiler phase. At the same time, an AHIR specification is always a single step away from a hardware implementation. AHIR specifies a set of constraints on delays in order to guarantee a correct implementation. It is easy to meet the constraints by suitably padding the delays in the implementation.

A large class of programming languages can be easily translated to a CDFG form. We provide a straight-forward method of translating this CDFG to an AHIR specification, and prove that the translation process itself is correct, i.e., an AHIR specification generated from a CDFG by this process always implements the behaviour specified by the CDFG.

AHIR factorises a circuit into separate control and data paths. This decoupling is the key to implementing optimisations that can scale with the size of the system. In particular, the Type-2 petri-net used for the control-path has a regular structure that is easy to analyse. This is demonstrated by our work on an optimisation that reuses operators in the data-path using a static analysis of the control-path. The complexity of the auxiliary structures and algorithms used by the analysis is close to linear with respect to the size of the control-path, and hence this optimisation is scalable to large systems.

AHIR also decouples the implementation of a memory subsystem from the actual circuit. A circuit can have an arbitrary number of load and store operators, and the memory subsystem is required to ensure that each request is eventually serviced. The implementation of the memory subsystem itself can be explored independently, without affecting the design of the circuit.

Summary

We have established a high-level synthesis flow with the following features:

1. An intermediate representation that is easy to analyse and transform using methods that can scale with the size of the circuit. The scalability is made possible by factorising the intermediate representation into three components: control, data and storage.

2. A verifiable path from high-level programs to the abstract representation that is built on existing software compilation techniques.

3. A set of constraints to ensure correctness when translating the intermediate representation to a hardware implementation. These constraints are easy to satisfy in practise and allow considerable freedom to the hardware implementer.

These features ensure that the entire path is verifiable and scalable, thus providing a competitive option for high-level synthesis.

6.1 Looking forward

Future research work in AHIR needs to target two goals in order to provide a viable alternative for hardware design. One is to extend the reach of the compiler in terms of programming languages and design paradigms, and the other is to improve the quality of hardware generated.

6.1.1 A universal design platform

AHIR can potentially be used as a universal design platform that unifies *software compilation* (translating programs to executables) with *hardware compilation* (synthesising circuits from programs). The expressive power of AHIR is evident at two levels: as a framework for exploring hardware architectures, and as a target for the design and implementation of high-level languages. A combination of these two aspects will result in a compiler flow that can map various classes of high-level languages to different low-level architectures.

Hardware Architectures

An AHIR specification is a primitive representation of the intended task, in terms of the sequence of operations and the resources required for them. The representation itself does not contain low-level details about the implementation. Hence, the synthesis flow can be targeted at any architecture by mapping the AHIR specification to suitable building blocks available in the target architecture. In addition, a synthesiser can also generate application specific architectures that are suitable for a given specification.

Support for various languages

The combination of a data-path and control-path in AHIR that interacts with a memory subsystem can be used to describe a microprocessor. The Type-2 petri-net used in AHIR is powerful enough to express any sequence of instructions that can be executed on a processor.

Our experiments demonstrate that AHIR can be used to compile imperative languages, with suitable implementations for specific features. Functional programming languages also can be implemented in AHIR using techniques similar to those used in microprocessors. The implementation must tackle issues similar to those seen in a microprocessor, such as emulating a stack in the memory address space, managing communication of function arguments, managing state for closures and co-routines, etc. AHIR can also support synchronous languages such as Esterel. The Type-2 petri-net is sufficient to express Esterel programs, with a suitable convention for implementing synchronous operations.

Further research should be aimed at exploring the practical aspects of implementing these languages and at providing the theoretical background for verifying their implementations.

6.1.2 Hardware optimisations

The current implementation has only been tried on small programs that consist of a few functions. AHIR can be used to generate hardware descriptions for large systems starting from high-level programs. The compiler must be able to generate implementations that deliver good throughput while occupying acceptable amounts of resources. These two parameters are usually traded-off with each other, since faster computations come at the cost of more hardware, and *vice versa*. AHIR provides many opportunities for creating transformations that improve the final circuit. We describe two of these as an example here.

Pipelining

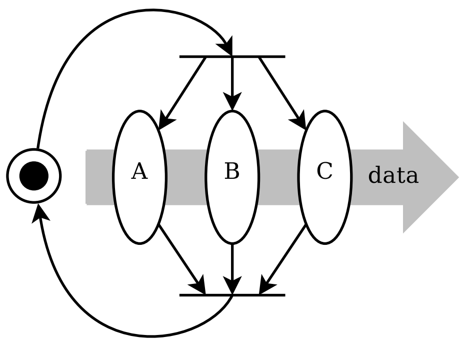


Figure 6.1: Synchronous pipeline in a Type-2 petri-net.

In the current implementation, parallelism is limited to the instructions within each basic block in the original program. Small sets of operations are triggered in parallel, when there are no dependences between them. But the parallelism allowed by a Type-2 petri-net is richer than this instruction-level parallelism. A control path can include a synchronous pipeline, as shown in Figure 6.1. The three regions A, B and C represent stages of the synchronous pipeline, which is managed by the fork and join.

Shared data-paths

The presence of the internal link layer makes it possible to share data-paths across different modules, as shown in Figure 6.2. In a statically shared data-path, the control-paths using it are guaranteed to be mutually exclusive. All the resources available in the data-path can be shared with no contention.

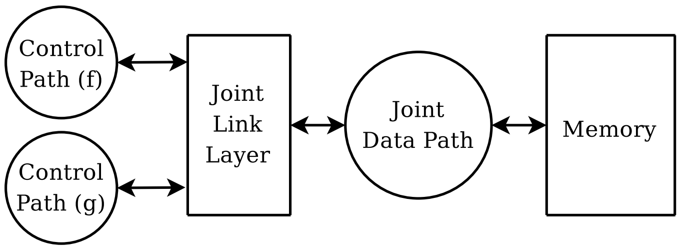


Figure 6.2: A shared data-path.

In a dynamically shared data-path, multiple control-paths can be active at the same time. The values associated with each control-path must be preserved, which constrains the sharing of registers. In addition, contention between control-paths for a shared resource will result in arbitration overheads. Due to these overheads, dynamic sharing is less attractive than static sharing. But dynamic sharing is more common in practise, in the sense that multi-processing is equivalent to a single data-path (the microprocessor) being shared by multiple control-paths (the processes running on it).

6.1.3 Memory subsystems

In addition to optimised hardware, the performance delivered by the memory subsystem is critical to the performance of the system. The high-level synthesis has to be coupled with an automated memory design flow, that generates application-specific memory subsystems that can keep up with the hardware system.

Some work has been done by related projects, towards developing simple performance models[32] for memory subsystems. Such models will support an integrated memory subsystem design procedure that can explore large areas of the design space in a feasible manner. For a given application, the design process may be parametrised by the properties of the expected memory access trace[22].

Appendix A

Implementation of a High-level Synthesis Flow using AHIR

We have implemented a straight-forward hardware synthesis flow that first translates a C program to AHIR using a CDFG as an intermediate step, and then translates the AHIR specification to synthesisable VHDL. This flow serves as a demonstration of how to build a competitive high-level synthesis flow based on AHIR. There cannot be a single definitive flow that represents the best way to translate high-level programs to hardware using AHIR. This is because any such flow includes a number of design decisions that are not part of the AHIR specification itself.

The implementation was guided by the simple need to quickly develop a working compiler flow without expending too much effort on optimality. It may be possible to replace various components of the implementation with better designs based on platform-specific and application-specific choices. In the following sections, we describe the most important choices faced when designing the flow, and the respective decisions made in the implementation.

A.1 Translating C to AHIR

AHIR is a transition step that divides the synthesis flow into two stages — generation of an AHIR specification, and implementation in hardware. In Figure 5.1, we show the translation process from a C program to a complete AHIR specification. The steps involved are as follows:

1. **C to LLVM IR:** The software optimisation is handled by the LLVM compiler framework. The input C program is compiled to the internal LLVM representation, and then optimised using tools available in the LLVM framework.

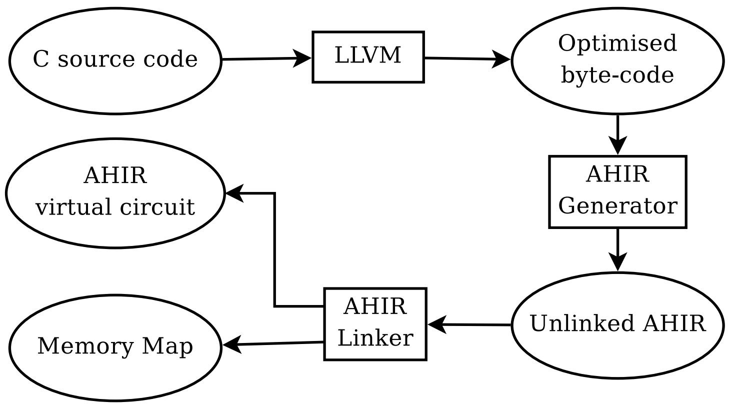


Figure A.1: Generating AHIR from a C program.

2. **LLVM IR to AHIR:** A specialised back-end implemented using the LLVM framework generates AHIR from the optimised LLVM form. The body of each function is represented as a CDFG, which is then translated to an AHIR module. This step produces two outputs — a collection of AHIR modules and a memory map that provides information about storage such as global variables and static allocation. This is not yet a complete AHIR system, although all the important components are present.

3. **AHIR linker:** The output produced by the AHIR back-end is termed as "unlinked AHIR" since it does not describe the interaction between the different modules. The system is completed by the AHIR linker that takes a set of AHIR modules and produces an inter-module link layer connecting them. In addition, the linker also assigns numerical addresses to the variables described in the memory map and updates the symbolic addresses in the AHIR data-paths with these numerical addresses. The result is a "linked" AHIR specification that describes a complete working system.

A.1.1 C to LLVM IR

The initial software phase of the compiler flow builds on the LLVM framework. The LLVM front-end parses the input C program and converts it into the LLVM intermediate representation. High-level information about the behaviour of the program is available at this stage, such as instruction-level parallelism (ILP), inter-procedure analysis (IPA) and memory reference analysis. Optimisations based on this information can have a significant impact on the performance of the generated system. In the current implementation, we have enabled a set of simple optimisations that are known to provide substantial improvements without any significant trade-offs to consider. This includes register promotion, strength reduction, constant propagation and common sub-expression elimination. Support for more complex optimisations will require an analysis of their effect on performance as well as hardware cost. A different set of optimisations may be applicable to every program being processed by the compiler.

A.1.2 LLVM IR to AHIR

The optimised LLVM IR is translated to AHIR using a CDFG as an intermediate step, as described in Section 3.3. Nodes in the CDFG represent computations happening in the program, but do not provide any details about the implementation. This ensures that a suitable implementation may be chosen either when translating the CDFG to AHIR, or when synthesising AHIR to actual hardware. For example, the CDFG represents a function call as a single "`call`" node, with data-edges which represent the arguments to be passed. The mechanism with which arguments are actually passed to the called CDFG is not specified, allowing later stages to choose a suitable implementation.

The CDFG is translated in a straight-forward manner to an AHIR representation. Each node and edge in the CDFG is replaced with the corresponding fragment in AHIR to obtain a complete AHIR module, as described in Section 3.4. The result of this construction is a set of AHIR modules that correspond to the functions in the input program. This is termed as an *unlinked* AHIR specification, which must now be linked to obtain a complete specification.

The AHIR template for a CDFG node consists of control and data-path fragments that implement the behaviour of the node. This essentially decouples the control and data flow present in the behaviour. The data-path fragment contains an abstract representation of an operator that performs the same operation as the CDFG node. It is possible to keep this representation abstract enough to hide all implementation details, or to lower the abstraction to specify the actual mechanisms used. For example, the CDFG "`call`" node can be implemented in two different ways in AHIR as described below.

A.1.3 Function calls

LLVM represents a function call as a single "call" instruction that lists the arguments that are to be passed in the call. This representation is independent of low-level details such as the use of registers or stack locations to pass the actual arguments to the called function. In a software compiler, these decisions are taken by the back-end that generates platform specific assembly code. In our synthesis flow, we have implemented two different methods for passing arguments:

Direct communication: A function call can be implemented as a call operator in the data-path that directly communicates with the called function. The operator has a number of input data-edges that correspond to the arguments that are to be passed in the function call. These arguments are driven on input/output ports connected to the inter-module link layer, where an arbiter forwards the call arguments to the appropriate module.

Postboxes: A function call can also be implemented indirectly using reserved memory locations or *postboxes*. The linker assigns such locations to each module — the module expects input arguments in these locations when it is called. During a function call, the calling module first stores the relevant arguments in these postboxes using normal store operators, and then sends a request symbol to the called module. When invoked, the called module loads the argument values using normal load operators and then begins execution. Finally, the return value is transferred using a similar postbox.

Either of these approaches is sufficient to implement a function call in AHIR. The former results in more hardware, but reduces the time required for transferring a call, while the latter manages to save on hardware, but takes longer to transfer the arguments. The high-level synthesis flow can choose either of these approaches or even a mixture, depending on the nature of the functions involved and the limitations of the target platform. Other mechanisms can also be devised for passing arguments without affecting the correctness of the AHIR specification.

I/O Ports

The call operator is in fact just one use of a general input/output operator that allows the data-path to communicate data (or cause side-effects) with the outside world. The load/store operator is also a special version of this operator, while a computation is a further specialisation that simply has no side effects. The input/output operator can also be used to implement other features such as streams, channels, file I/O, etc.

A.1.4 AHIR Linker

The LLVM backend produces an unlinked AHIR specification which essentially consists of a collection of modules, with no interaction specified between them. The AHIR linker completes the system specification by producing two pieces of information — an inter-module link layer that enables communication between the module, and an address assignment that maps all the storage locations to actual addresses in the memory subsystem.

The Inter-module Link Layer

The inter-module link layer is used for communication between modules. It provides an arbiter for each module, which is responsible for forwarding control and data messages to that module. Each module has a single port on which it receives a call from the inter-module link layer. The linker connects this port to the corresponding arbiter in the link layer.

A "call" operator in the AHIR data-path declares input and output ports for the call arguments. When a call operator c in a module A calls a module B, it drives the required arguments on its output port. The corresponding arbiter samples this port and then forwards the request to the module B when it is free. When module B signals completion, the arbiter samples the return value and forwards it to module A, thus completing the call. The arbiter uses a bypass mechanism so that the arguments are available in the same clock cycle when they are sampled. Hence the call is forwarded from A to B instantaneously if module B is free.

The arbitration policy used by the arbiter is not specified in AHIR — the current implementation uses a fixed priority scheme. Note that for a system generated from a C program, there is no contention for a called module, since only one module is active at a time. As a result, arbitration is unnecessary in the inter-module link layer and can be optimised away.

Memory Locations

The LLVM IR is an SSA representation that replaces *variables* in the program with *values* produced by operations. These values are implemented in AHIR as registers at the outputs of data-path operators. But the following kinds of variables result in actual locations in the memory address space:

- Postboxes used for function calls.
- Global variables.
- Global or local arrays and structures.
- Local scalar variables whose address is assigned to pointers in the function — the LLVM framework removes such variables from the SSA graph and replaces any access to them with load/store operators.

The unlinked AHIR produced by the LLVM back-end represents the addresses of these variables as symbolic constants in the data-paths. The linker maps these variables to locations in the memory address space, and updates the symbolic constants with the numerical values of these locations. Note that typical compilers allocate local variables onto the stack. But the current implementation does not have the notion of a stack. Instead, local variables are also assigned permanent locations in the memory address space.

A.2 Synthesising AHIR

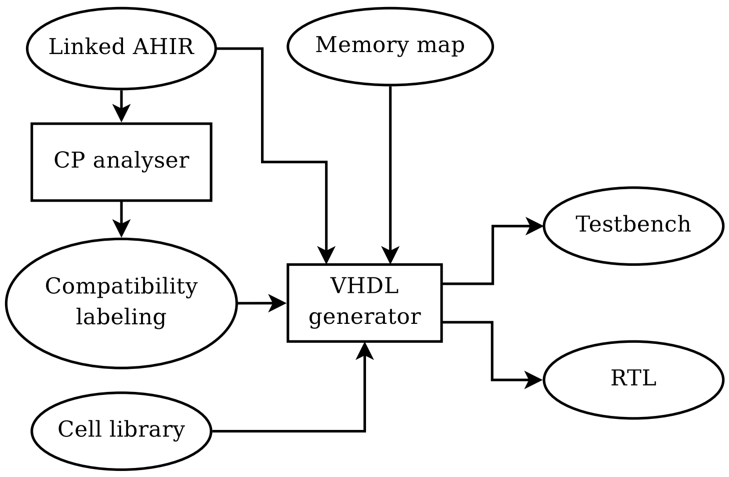


Figure A.2: Translating AHIR to synthesisable VHDL RTL.

The second phase of the high-level synthesis flow consists of translating the AHIR specification to a synthesisable VHDL RTL description. The VHDL circuit is created by translating each element in AHIR — such as a control-path place or transition or a data-path operator — to an instance of the corresponding entity from a pre-defined cell library. This is a straightforward replacement that does not target constraints such as timing, power and area in the resulting circuit. The VHDL RTL is later synthesised to the target platform (such as ASIC or FPGA) using an automated flow based on vendor-supplied tools.

A.2.1 Synchronous VHDL

We have implemented a VHDL cell library that provides the necessary building blocks in the form of VHDL entities. These are used to create a synchronous implementation of the AHIR specification where the control-path is a Mealy machine while the data-path is a Moore machine.

Data-path operator

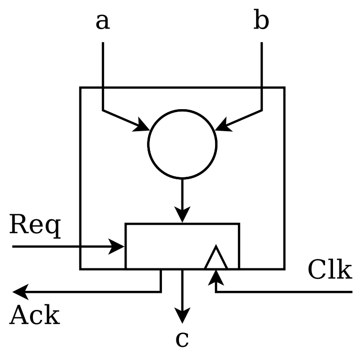


Figure A.3: Data-path operator.

Each data-path operator is made of two parts as shown in Figure A.3 — a combinational element for the intended operation, and an output register that interacts with the symbolic handshake. The request-acknowledge signalling is implemented as pulses with a width of one clock cycle. The data-path operator expects a pulse on its request input to begin execution, and emits a pulse on its acknowledgement output on completion.

Control-path place

A control-path place is implemented as a flip-flop with bypass logic as shown in Figure A.4. The place is set when any one of its input transitions fire, and reset when any one of its output transitions fires. The implementation assumes that the petri-net is safe, and hence no check is made for multiple input or output transitions firing.

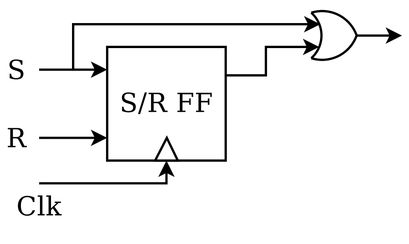


Figure A.4: Control-path place.

The bypass logic ensures that the value at the input of the place is available at the output in the same cycle when the flip-flop is enabled. Thus when an input transition fires, a pulse arrives at the set input of the place and is immediately forwarded to the output transitions. If no output transition fires, the flip-flop is set and the output remains high after the pulse disappears from the input. It will be reset when any of the output transitions fire. If an output transition fires immediately, then a pulse arrives simultaneously on the reset input of the place, and the flip-flop is never set.

Control-path transition

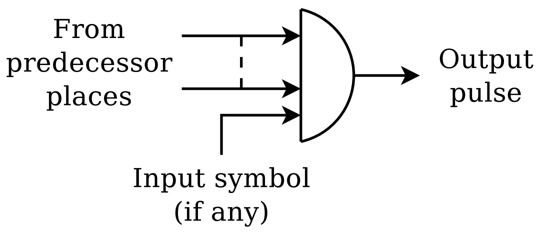


Figure A.5: Control-path transition.

The control-path transition is implemented as combinational logic that computes the logical AND of all the signals from input places, and the acknowledge symbol if it exists. For a safe petri-net, we can show the following:

1. The output of an idle transition is low when it is not enabled.

2. When the transition is enabled (all the input places are set) and the acknowledge symbol arrives (if any), the output of the transition is high for exactly one cycle, and it resets all its input places.

A complete AHIR fragment

In Figure A.6, we show the implementation of an AHIR fragment for a simple binary operator. Place P_0 enables the transition T_0, which produces a pulse on the Req signal for the operator, and also marks place P_1. When the operator completes execution, it generates a pulse on the Ack signal, which enables the transition T_1.

Note that transition T_0 is trivial since it has only one input place. It fires as soon as a pulse arrives at the input of place P_0, and asserts the reset signal. As a result, place P_0 is never set, and can be optimised away by the low-level synthesis flow.

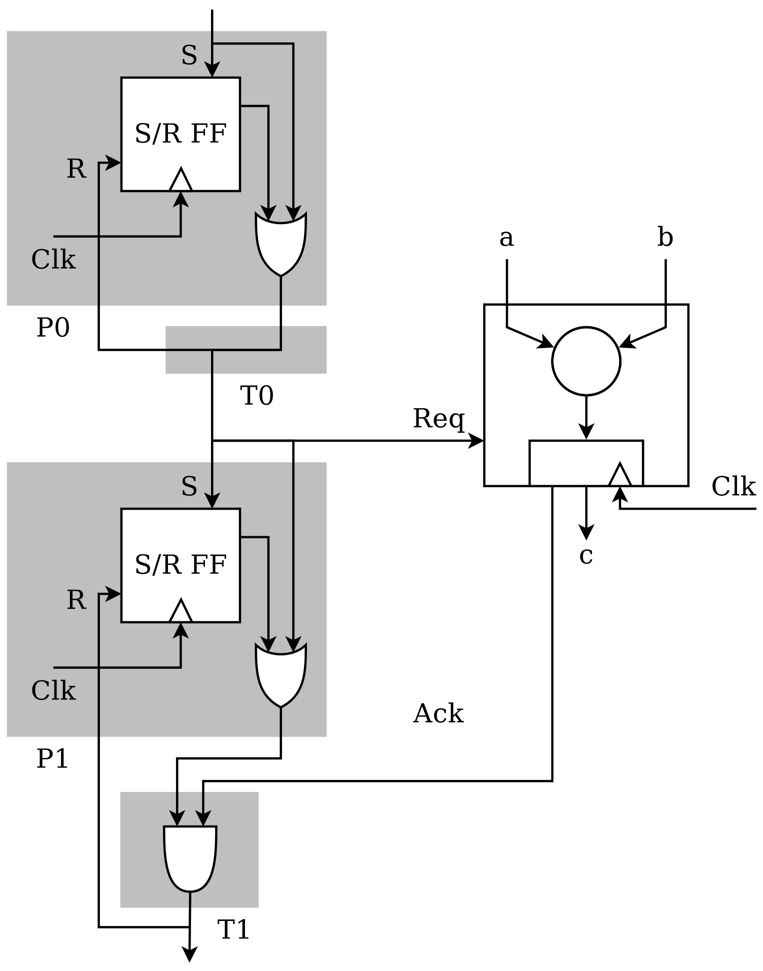


Figure A.6: A complete AHIR fragment.

A.2.2 Operator reuse

The VHDL generator also optimises the AHIR representation using the arbiterless sharing scheme described in Chapter 4. This optimisation maps multiple request-acknowledge handshakes in the control-path to the same shared operator in the data-path. The cell library provides a wrapper element that manages the incoming data and updates the appropriate output registers for each handshake.

A.2.3 Scheduling and Allocation

There is no explicit scheduling step in the current implementation of the high-level synthesis flow. When the CDFG is created from the LLVM IR, control-edges are introduced that represent data dependences between operations, as described in Section 3.3. This implicitly schedules the CDFG operations in an “as soon as possible” manner. The same scheduling is seen in the generated AHIR specification, since the translation process preserves the control-flow in the CDFG — operators in the data-path are triggered as soon as their dependences are satisfied.

Also, the synthesis flow does not have a separate allocation or binding step when mapping the AHIR specification to hardware. Each operator in a data-path is mapped to a new instance from the cell library. The data-path operator may itself represent a shared operator created as a result of arbiterless sharing. But the actual operator instantiated for this data-path operator is not shared with any other operator.

Note that this is not the only way to schedule operations or to utilise available hardware resources. Further improvements in the synthesis flow may use the results of research that has already been done towards the problem of scheduling and allocation during synthesis. It may be possible to modify the implicit ASAP scheduling scheme to use information available about the actual delays between operations. Such a schedule can result in improved sharing of operators since additional run-time information is available besides the existing static analysis of the control-path.

Appendix B

An End-to-end Example

The high-level synthesis flow is implemented as a sequence of utilities that starts with a text file containing the input C program and ends with a set of VHDL files describing the hardware implementation:

`llvm-gcc` — The C front-end provided by LLVM for parsing the input C program.

`irgen` — A command implemented in the LLVM framework that optimises the LLVM byte-code and generates an unlinked AHIR specification.

`irlink` — A linker that links the modules in the AHIR specification to produce an inter-module link layer and a memory map.

`irsyn` — A VHDL generator that translates the linked AHIR specification to an RTL description, and also generates a test-bench based on the memory map.

B.1 Expected input

The tool-chain works with a single C file that contains all the functions of a program. The program accepted by the tool-chain is subject to two restrictions:

1. The complete program must be present in a single file.

2. The complete DAG must contain a function called `start`, which denotes the root of the call-graph. The function `main` cannot be used, since it triggers internal modifications by the C front-end that are incompatible with the AHIR tools.

For the following sections, we use an example file named `mul.c` containing the C program listed in Figure B.1. The call-graph of the program consists of a single function call from start to mul. The program accesses two integer global variables and returns their product.

$$\texttt{start} \longrightarrow \texttt{mul}$$

B.2 Source compiler

The input C source is parsed by the LLVM front-end, `llvm-gcc`. This produces a bytecode file that contains the input program in LLVM's native IR, which can be viewed using a utility called `llvm-dis` provided by the LLVM framework. This utility displays the bytecode in the form of a virtual assembly code as shown in Figure B.2.

B.3 AHIR-XML generator

The command `irgen` is an LLVM-based tool that reads the bytecode, converts it to AHIR, and generates an XML file describing the AHIR virtual circuit. This is the unlinked form of the AHIR specification which lacks two pieces of information — the interaction between the modules, and the addresses assigned to various memory locations. In our example, the XML file contains the call-graph of the program, a description of the global variables and descriptions of the functions `start` and `mul` as shown in Figures B.3 and B.4.

B.4 AHIR linker

The AHIR linker produces a linked version of the input AHIR specification in a file called `mul_linked.xml`. This file contains a description of the inter-module link layer as listed in Figure B.5 (referred to as "omega"), besides the module descriptions copied from the unlinked XML. Additionally, a memory map is created in a file named `mul_map.xml` as shown in Figure B.6. The memory map is divided in two sections:

`init` — The initial state of the memory when the system is invoked.

`fin` — The expected state of the memory after execution.

These sections are used to generate the test-bench for the VHDL implementation. The test-bench initialises the memory subsystem with values specified in the init section and begins execution of the system. The fin section specifies the final state of the memory, which is used by the test-bench to verify the results after execution.

B.5 VHDL generator

The VHDL generator produces a hardware implementation by mapping each element in the AHIR specification to a hardware entity. These entities are assumed to be available in an independent library. The VHDL description consists of a number of files as listed below:

start_cp.vhdl, etc — One control-path for each function.

start_dp.vhdl, etc — Data-path.

start_ln.vhdl, etc — Intra-module link layer.

system.vhdl — The final circuit, built using instances of all the above components.

testbench.vhdl — Used for testing the system.

```
int a, b;
int start(void)
{
  return mul(a,b);
}
int mul(int a, int b)
{
  return a * b;
}
```

Figure B.1: Input C code.

```
; ModuleID = 'mul.bc'
target endian = little
target pointersize = 32
target triple = "i686-pc-linux-gnu"
%b = weak global int 0          ; <int*> [#uses=1]
%a = weak global int 0          ; <int*> [#uses=1]

implementation   ; Functions:

int %start()
entry:
        %tmp = load int* %b             ; <int> [#uses=1]
        %tmp1 = load int* %a            ; <int> [#uses=1]
        %tmp21 = call int %mul( int %tmp1, int %tmp ) ; <int> [#uses=1]
        %unnamed_0 = ret int %tmp21


int %mul(int %a, int %b)
entry:
        %tmp2 = mul int %a, %b          ; <int> [#uses=1]
        %unnamed_1 = ret int %tmp2
```

Figure B.2: LLVM byte-code.

```
<callgraph id="mul" start="start">
  <function id="mul" cp="mul_cp" dp="mul_dp" ln="mul_ln"
            req="1" ack="1">
  </function>
  <function id="start" cp="start_cp" dp="start_dp" ln="start_ln"
            req="1" ack="1">
    <callsite id="call_mul_0" callee="mul" req="6" ack="6"
              args_width="64" retval_width="32">
    </callsite>
  </function>
</callgraph>

<globals>
  <global id="a" size="1" type="int">
    <scalar type="int" size="1">0</scalar>
    <address id="location_start_dp_global_a_1"/>
  </global>

  <global id="b" size="1" type="int">
    <scalar type="int" size="1">0</scalar>
    <address id="location_start_dp_global_b_0"/>
  </global>
</globals>
```

Figure B.3: XML format: Call-graph and Global variables

```
<cp id="mul_cp">
  <trans id="entry.entry">
    <src>place_1</src>
    <snk>place_4</snk>
  </trans>
  ...
</cp>
<dp id="mul_dp">
  <wire id="oper_tmp2_int.wire" type="int"/>
  <dpe id="oper_tmp2_int" operation="mul" type="int">
    ...
    <smap>
      <req id="req0">1</req>
      <ack id="ack0">1</ack>
    </smap>
  </dpe>
  ...
</dp>
<ln id="mul_ln">
  <map>
    <from iface="mul_cp" sym="2"/>
    <to iface="mul_dp" sym="1"/>
  </map>
  ...
</ln>
```

Figure B.4: XML format: Components of an AHIR module.

```
<omega id="omega">
<server id="mul" req="1" ack="1"
        args_width="64" retval_width="32">
  <call id="start_call_mul_0" client="start" callsite="call_mul_0"
        req="2" ack="2"/>
</server>
<server id="start" req="1" ack="1"
        args_width="0" retval_width="32">
  <call id="env_call_start" client="env" callsite="call_start"
        req="1" ack="1"/>
</server>
</omega>
```

Figure B.5: XML format: Inter-module link layer (Omega)

```
<map id="mul" size="3">
<init>
<location addr="1" id="a" type="int" size="1">
  <scalar type="int" size="1">0</scalar>
</location>
...
</init>
<fin>
</fin>
</map>
```

Figure B.6: XML format: Memory map

```
library ahir;
use ahir.dpath.all;

entity mul_dp is
  port(
    ip : in std_logic_vector(1 downto 1);
    op : out std_logic_vector(1 downto 1) := (others => '0');
    args : in std_logic_vector(63 downto 0);
    retval : out std_logic_vector(31 downto 0);
    reset : in std_logic;
    clk   : in std_logic);
end mul_dp;

architecture default_arch of mul_dp is
  signal formal_0_d_wire : std_logic_vector(31 downto 0);
  signal oper_tmp2_int_d_wire : std_logic_vector(31 downto 0);
  ...
begin
  ...
  int_mul_0 : binary_operator -- MUL
  port map(
    clk => clk
    , ip0 => int_mul_0_ip0 -- in
    , ip1 => int_mul_0_ip1 -- in
    , op => int_mul_0_op -- out
    , req0 => int_mul_0_req0
    , ack0 => int_mul_0_ack0
    , reset => reset);
  ...
end default_arch;
```

Figure B.7: VHDL format: A data-path entity.

```
library ahir;
use ahir.cpath.all;

entity mul_cp is
  port (
    ip    : in std_logic_vector(2 downto 1);
    op    : out std_logic_vector(2 downto 1) := (others => '0');
    reset : in std_logic;
    clk   : in std_logic);
end mul_cp;

architecture default_arch of mul_cp is
  signal entry_d_entry_tip : std_logic_vector(0 downto 0);
  signal entry_d_entry_ge : std_logic;
  signal entry_d_entry_top : std_logic;
  ...
begin
  ...
  entry_d_entry : transition
  generic map(1)
  port map(entry_d_entry_tip, entry_d_entry_ge, entry_d_entry_top);

  fin : transition
  generic map(1)
  port map(fin_tip, fin_ge, fin_top);
  ...
  entry_d_entry_tip(0) <= init_top;
  oper_tmp2_int_d_req_tip(0) <= entry_d_entry_top;
  fin_tip(0) <= oper_tmp2_int_d_ack_top;
  ...
end default_arch;
```

Figure B.8: VHDL format: A control-path entity.

```
entity system is
  port (
    env_reqs : in  std_logic_vector(1 downto 1);
    env_acks : out std_logic_vector(1 downto 1);
    ...;
end system;
architecture default_arch of system is
...
begin
...
  memory_inst : memory_subsystem
    generic map (
        num_loads       => 3
      , num_stores      => 1
      , data_width      => memory_data_width
      , addr_width      => memory_address_width)
    port map (
      ... load ports
      ... store ports);
  mul_dp_inst : mul_dp
    port map (...);
  mul_cp_inst : mul_cp
    port map (...);
  omega_mul : omega_amux -- arbiter in the inter-module link layer
    generic map (
      args_width => 64,
      retval_width => 32,
      num_clients => 1)
    port map (...);
  ...
end default_arch;
```

Figure B.9: VHDL format: The system entity.

Bibliography

[1] S. D. Sahasrabuddhe, H. Raja, K. Arya, and M. P. Desai, "AHIR: A Hardware Intermediate Representation for Hardware Generation from High-level Programs," in *20th International Conference on VLSI Design*, pp. 245–250, January 2007.

[2] B. Zahiri, "Structured ASICs: Opportunities and Challenges," in *21st International Conference on Computer Design*, pp. 404–409, Oct 2003.

[3] 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*, vol. 2438, p. 171, Springer Berlin / Heidelberg, Jan 2002.

[4] 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*, pp. 187–194, Nov 2002.

[5] 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*, (New York, NY, USA), pp. 225–235, ACM Press, 2000.

[6] D. D. Gajski, F. Vahid, and S. Narayan, "A System-Design Methodology: Executable-Specification Refinement," in *European Design and Test Conference (ED&TC) 94*, pp. 458–463, February 1994.

[7] Arvind, R. Nikhil, D. Rosenband, and N. Dave, "High-level synthesis: An Essential Ingredient for Designing Complex ASICs," in *International Conference on Computer Aided Design (ICCAD 2004)*, November 2004.

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

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

[10] "The Phoenix Project." http://www.cs.cmu.edu/ phoenix/compiler.html.

[11] M. Budiu and S. C. Goldstein, "Pegasus: An Efficient Intermediate Representation," tech. rep., School of Computer Science, Carnegie Mellon University, April 2002.

[12] G. Venkataramani, M. Budiu, T. Chelcea, and S. Goldstein, "C to Asynchronous Dataflow Circuits: An End-to-End Toolflow," in *International Workshop on Logic & Synthesis*, (Temecula, CA), pp. 501–508, June 2004.

[13] "SPARK: High-Level Synthesis using Parallelizing Compiler Techniques." http://mesl.ucsd.edu/spark/.

[14] T. Murata, "Petri Nets: Properties, Analysis and Applications," in *Proceedings of the IEEE*, vol. 77, 1989.

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

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

[17] M. Rim and R. Jain, "Representing Conditional Branches for High-Level Synthesis Applications," in *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 106–111, 1992.

[18] G. G. Jong, "Data Flow Graphs: System Specification With the Most Unrestricted Semantics," in *Proceedings of the European Conference on Design Automation*, pp. 401–405, IEEE, 1991.

[19] J. T. van Eijndhoven and L. Stok, "A Data Flow Graph Exchange Standard," in *Proceedings of the 3rd European Conference on Design Automation*, pp. 193–199, IEEE, 1992.

[20] D. D. Gajzki and A. Orailoglu, "Flow Graph Representation," in *Proceedings of the 23rd Design Automation Conference*, pp. 503–509, IEEE, 1986.

[21] S. Amellal and B. Kaminska, "Functional Synthesis of Digital Systems with TASS," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 537–552, IEEE, May 1994.

[22] A. Singla, "Memory Access Pattern Analysis," Master's thesis, Department of Electrical Engineering, IIT Bombay, June 2008.

[23] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the pentium 4 processor," in *Intel Technology Journal* (L. Chao, ed.), vol. 5, Intel Corporation, February 2001.

[24] M. Xu and F. J. Kurdahi, "Accurate Prediction of Quality Metrics for Logic Level Designs Targeted Toward Lookup-Table-Based FPGA's," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, pp. 411–418, December 1999.

[25] J. J. Dongarra, P. Luszczek, and A. Petitet, “The LINPACK Benchmark: past, present and future,” December 2001. Available online: http://www.netlib.org/utk/people/JackDongarra/PAPERS/hpl.pdf.

[26] D. Ivereigh, “RedBlack Balanced Tree Searching and Sorting Library.” http://libredblack.sourceforge.net/.

[27] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, 1992.

[28] P. Somawanshi, “A Hardware Implementation of Cryptographic Ciphers,” Master’s thesis, Dept. of Electrical Engineering, IIT Bombay, 2008.

[29] G. Kostopoulos, N. Sklavos, M. D. Galanis, and O. Koufopavlou, “VLSI Implementation of GSM Security: A5/1 and W7 Ciphers,” in *International IEEE Workshop on Wireless Circuits and Systems*, May 2004.

[30] T. Good and M. Benaissa, “AES on FPGA from the Fastest to the Smallest,” in *Cryptographic Hardware and Embedded Systems - CHES 2005*, vol. 3659/2005 of *Lecture Notes in Computer Science*, pp. 427–440, Springer Berlin / Heidelberg, September 2005.

[31] K. P. Ghosh and S. Subramanian, “Power / Delay Estimation of Auto Generated Circuits,” tech. rep., Department of Electrical Engineering, IIT Bombay, April 2009.

[32] G. Hazari, *Bottleneck Analysis and Performance Modeling of VLSI Memory Sub-systems*. PhD thesis, Department of Electrical Engineering, IIT Bombay. Submitted in 2009.

Acknowledgments

My family — father Dilip, mother Pushpalata, brothers Mandar and Mahesh, and my wife Meenal — for allowing me the luxury of being insulated from reality while I followed my path, and for being there with me at every step.

My guides — Prof. Kavi Arya and Prof. Madhav P. Desai — for the inspiration and guidance I received from them during my work, and for pointing me in the right direction every time I strayed from the one true path. It is in their presence that I have begun to appreciate the meaning of clarity in thought and action.

The office staff at the erstwhile KReSIT, at the Department of CSE and at the VLSI Lab in the Department of EE, who have been instrumental in smoothening my interaction with the administrative processes at IIT Bombay. They have always strived to cushion students from the mundane workings of academic life at the institute, truly personifying the popular marketing slogan, “Hum hain na!”.

Last but not at all the least, are all the friends that have *made* my stay at IIT Bombay. The list is too long to be exhaustively written down, but I mention a few, roughly in the order of appearance — Shantanu, Raghu, Srinath, Vikram, Abhinay, Deepanshu, Ajay, Abhishek, Janaki, Paddy, Jaju, Yogi, Chetan, Arun, Shweta, Nitin, Vipul, Jatin, Tejaswi and Ramanand. Thanks for all the fish, guys! And the beer, and the Rock, the movies, the SIGFood outings, the late-night stints at SP, the early morning excursions to Maddu’s, the treks, the Foundation Lab assignments, the amazing adventures as department sysads, the chai, the road trips, and everything else in between, too!

Date: ____________________ Sameer D. Sahasrabuddhe