The FLaSH Compiler: Efficient Circuits from Functional Specifications

Richard Sharp[1]
rws@uk.research.att.com

Alan Mycroft[1,2]
am@cl.cam.ac.uk

[1]AT&T Laboratories Cambridge
24a Trumpington Street
Cambridge CB2 1QA, UK

[2]Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street
Cambridge CB2 3QG, UK

June 23, 2000

Abstract

In previous work we have outlined the design of a functional language, SAFL, and argued that it is well suited to hardware description and synthesis. Unlike conventional high-level synthesis languages, SAFL specifications capture explicitly resource allocation, variable binding and scheduling. This paper is concerned with the details of the FLaSH compiler: an optimising silicon compiler which translates SAFL specifications to RTL Verilog suitable for simulation or synthesis. We describe a number of high-level optimisation and analysis techniques which find novel application in the field of hardware-synthesis. In particular, we believe our approach to compiling function definitions into shared resources could be applied advantageously in existing industrial silicon compilers.

1 Introduction

The last few decades have seen significant advances in programming language design and implementation. Although much of the research in these areas has been directly applied to software design, it seems that many of the established results and techniques have not yet found their way into the world of hardware synthesis. At AT&T Laboratories Cambridge we are in the process of building an advanced hardware synthesis system, combining new techniques with those that have currently only been applied to software compilation.

The FLaSH (Functional Languages for Synthesising Hardware) system allows a designer to map a high level functional language, SAFL (Statically Allocated Functional Language), into hardware. The system has two phases:

1. We *transform* SAFL programs using meaning-preserving transformations to choose the area-time position (e.g. resource allocation, binding and schedule) while remaining a high-level specification.
2. The resulting specification is *compiled* into hardware in a *resource-aware* manner, that is we map separate functions to separate hardware functional units; functions which are called twice now become shared functional units—accessed by multiplexers and possibly arbiters.

This paper concerns the internals of the FLaSH compiler, outlining novel analysis and optimisation techniques which allow us to compile SAFL to efficient hardware whilst respecting resource-awareness. A more formal presentation of SAFL and examples of source-to-source program transformations appear in [9]; hardware/software partitioning using SAFL is considered in [11] and a brief overview of the whole project can be found in [10].

1.1 A brief overview of the SAFL language

SAFL is a language of first order recurrence equations with an ML-like syntax [7]; a user program consists of a number of function definitions (declared using the `fun` keyword) and a single *initialising expression* declared with the `do` keyword. The initialising expression is invoked as soon as the program is executed and is thus analogous to C's `main()` function. Other constructs include `let-in-end` which declares variable bindings; `if-then-else` for conditional evaluation and a set of primitive operations for use in arithmetic and boolean expressions (e.g. `+`, `*`, `<`, `>=` etc.). SAFL has a call-by-value semantics since strict evaluation naturally facilitates parallel execution which is ideal for hardware implementation. Some of the interesting features of the FLaSH system are outlined below:

Parallelism

SAFL is a functional language and therefore enjoys the property of referential transparency. This enables us to synthesise designs in which many expressions are computed simultaneously. More precisely the FLaSH compiler produces designs in which all function call arguments and `let` declarations are evaluated in parallel. This kind of fine-grained parallelism is ideal for a hardware implementation where, in contrast to software, performing operations in parallel is easier than forcing their sequentialisation.

Resource Awareness

One of the novel features of the FLaSH compiler is *resource awareness*. We enforce the rule that a single SAFL function definition, f, synthesises to a *single* block of hardware, H_f. In this context, multiple calls to f at the source level correspond directly to multiple accesses to the shared unit, H_f, at the hardware level. H_f supports a function-style call-return interface and has a *single* data output. We use the terms *resource* and *functional-unit* interchangeably to refer to hardware blocks like H_f.

Our approach can be illustrated by considering the compilation of the following SAFL code:

```
fun mult(x, y, acc) =
  if (x=0 | y=0) then acc
     else mult(x<<1, y>>1, if y.bit0 then acc+x else acc)

fun cube(x) = mult(mult(x, x, 0), x, 0)
```

From this specification, the FLaSH compiler generates two hardware resources: a circuit, $H_{\texttt{mult}}$, corresponding to `mult`[1] and a circuit, $H_{\texttt{cube}}$, corresponding to `cube`. The two calls to `mult` are *not* inlined: at the hardware level there is only one shared resource, $H_{\texttt{mult}}$, which is invoked twice by $H_{\texttt{cube}}$.

Adopting a technique such as this allows SAFL to capture both the intensional semantics of a specification *and* the structure of the synthesised circuit (*extensional semantics*). In this framework, source-to-source program transformation becomes a very powerful technique allowing a designer to explore a wide range of possible implementations by repeatedly transforming some initial specification [9, 11].

Static Allocation

In order to make SAFL well suited to hardware description and synthesis we impose syntactic restrictions on SAFL source:

1. a function can only call previously defined functions; and

2. all recursive calls must be in tail-context.

[1]The tail-recursive call is synthesised into a feedback loop at the circuit level.

Restriction (1) prohibits mutual recursion which, if treated naïvely, can create cycles in our call graph, leading to deadlock in our hardware implementations where functions represent shared resources.[2] Restriction (2) ensures that the amount of storage (e.g. number of registers) needed for a program's execution can be calculated at compile time, a property that we call *static allocability*.

We argue that *static allocation* is ideal for hardware synthesis since it means our designs do not have to rely on a global addressable store to model stacks and heaps. This prevents us from inhibiting parallelism by creating a *Von Neumann bottleneck*.

Architecture Independence

Although we try to make SAFL well-suited to describing hardware in general, we are careful not to favour the description of any particular circuit design paradigm. We say that SAFL is *architecture neutral* to mean that it abstracts a number of implementation styles. For example, a single SAFL specification could be compiled into either synchronous or asynchronous hardware.

In particular, resource-awareness provides a useful abstraction since we can compile different source-level function definitions into different design styles (e.g. synchronous or asynchronous) and automatically synthesise suitable inter-resource interfaces. In [11] we use this idea to show how a source-to-source transformation can arbitrarily partition a SAFL specification into hardware and software parts.

1.2 Relation to Other Work

We are not the first to observe that the mathematical properties of functional languages are desirable for hardware description and synthesis. A number of synchronous dataflow languages, the most notable being LUSTRE [4], have been used to synthesise hardware from declarative specifications. However, whereas LUSTRE is designed to specify reactive systems SAFL describes interactive systems (this taxonomy is introduced in [1]). Furthermore LUSTRE is inherently synchronous: specifications rely on the explicit definition of clock signals. This is in contrast to SAFL which could, for example, be compiled into either synchronous or asynchronous circuits.

The ELLA HDL is often described as functional. However, although constructs exist to define and use functions the language semantics forbid a resource-aware compilation strategy. This is illustrated by the following extract from the ELLA manual [8]:

> Once you have created a named function, you can use instances of it as required in other functions ... [each] instance of a function represents a distinct copy of the block of circuitry.

ELLA contains low-level constructs such as `DELAY` to create feedback loops, restricting high-level analysis. SAFL uses tail-recursion to represent loops; this strategy makes high-level analysis a more powerful technique.

Previous work on compiling declarative specifications to hardware has centred on how functional languages themselves can be used as tools to aid the design of circuits. Sheeran's et al. muFP [13] and Lava [2] systems use functional programming techniques (such as higher order functions) to express concisely the repeating structures that often appear in hardware circuits. In this framework, using different interpretations of primitive functions corresponds to various operations including behavioural simulation and netlist generation. Our approach takes SAFL constructs (rather than gates) as primitive. Although this restricts the class of circuits we can describe to those which satisfy certain high-level properties, it permits high-level analysis and optimisation yielding efficient hardware.

Conventional HDLs such as Verilog and VHDL allow a user to specify a design at various levels of abstraction. Although the behavioural subsets of these languages support function definitions and calls, we are not aware of any existing high-level synthesis tools which provide explicit support for treating functions as shared resources (e.g. automatic support for sharing, insertion of arbiters

[2] In fact the formal semantics of SAFL presented in [9] permits a form of mutual recursion by stratifying function definitions into (potentially mutually recursive) *groups*.

and temporary registers). Typically, behavioural Verilog and VHDL synthesisers inline function calls as a preprocessing stage and then apply traditional high-level scheduling techniques to share the large amount of duplicated logic which arises as a result. Silicon compilers for other high-level languages [3, 6, 12] tend to use a similar approach leading to a dangerous exponential increase in code-size which our method avoids.

2 Compiling SAFL

Translating SAFL to parallel hardware whilst respecting *resource awareness* leads to some interesting issues which the FLaSH compiler needs to deal with. In order to whet the reader's appetite (and to provide some real examples of SAFL code) two such issues are presented here:

2.1 Parallel Sharing Conflicts

Parallel sharing conflicts arise because, at the hardware level, we are dealing with multiple threads all trying to access a shared set of resources. Consider the following SAFL code fragment taken from the specification of a processor:

```
fun add(x,y) = x+y
...
fun ALU(op, arg1, arg2, ...) =
     if op=1 then add(arg1,arg2)
             else ...
...
fun calculate_new_PC(current_PC,offset,condition) =
     if condition then add(current_PC,offset)
                  else current_PC
```

Since both the `ALU` and `calculate_new_PC` functions call `add`, the FLaSH compiler will synthesise a circuit containing a single `add`-unit shared between the `ALU` and `calculate_new_PC` units. One interesting question is, although the `add` circuit is shared, is it subject to multiple concurrent accesses—i.e. is there a scenario where both the `calculate_new_PC` and `ALU` functions may try to call the `add` circuit simultaneously? If so we say that the calls to `add` have a *parallel sharing conflict* and automatically synthesise an arbiter to protect $H_{\texttt{add}}$ from multiple concurrent accesses.[3]

The FLaSH compiler performs *parallel conflict analysis* to infer which hardware resources are subject to sharing conflicts. This enables us to synthesise arbiters only where necessary—even though a functional-unit is shared our compiler is often able to infer from the program structure that an arbiter is not required. Parallel conflict analysis is described in detail in Section 4.

2.2 Register Placement

Consider the following SAFL expression:

```
let var x = f(4)
 in let var y = f(5)
     in x + y
    end
end
```

In this example `x` is bound to the result of computing `f(4)` whilst `y` is bound to the result of computing `f(5)`. However, since `f` represents a shared resource, H_f, with a single output we see that, if translated naïvely, the second call to `f` will invalidate the first (since both `x` and `y` are bound to H_f's shared output). This is an instance of a *sequential sharing conflict*, the result of

[3]Note the similarity between parallel sharing conflicts and structural hazards [5] in pipelined processor design.

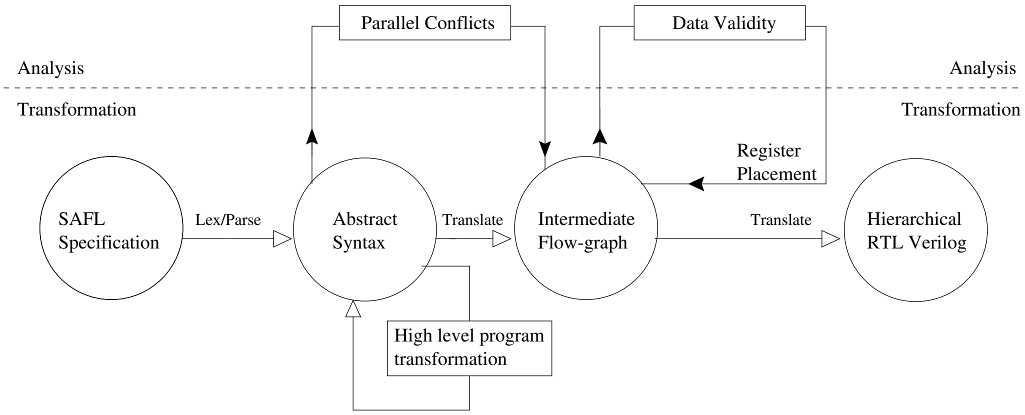


Figure 1: Structure of the FLaSH Compiler

which is that we must synthesise a register to latch the value of `f(4)` before it is corrupted. We call these latches *permanising registers* since they make the result of computing an expression permanent, decoupling the caller from the callee.

Similarly, consider this code fragment in which the calls to `f` have a parallel sharing conflict (see Section 2.1):

```
let var x = f(4)
    var y = f(5)
 in x+y
end
```

Here, `f(4)` will be evaluated in parallel with `f(5)` and both `x` and `y` will be bound to the shared output of resource H_f. Although an arbiter will be synthesised to dynamically schedule the concurrent accesses to H_f, we cannot determine statically[4] which of the two `let` declarations will be evaluated first. Thus, in this case, we require two permanising registers: one to latch the value of computing `f(4)`, the other to latch the value of `f(5)`.

The problem of register placement involves the insertion of as few registers as possible to deal both parallel and sequential conflicts. The method used to place registers in the FLaSH compiler is documented later in this paper.

3 An overview of the FLaSH Compiler

The structure of the FLaSH compiler can be seen in Figure 1 and is summarised below:

- SAFL source is lexed and parsed into an abstract syntax tree. We check that the source complies with the restrictions described in Section 1.1; invalid SAFL is rejected.
- We perform *parallel conflict analysis* at the abstract syntax level which allows us to infer which functions may be subject to multiple concurrent calls. The results of this analysis are used to place arbiters at the hardware level.
- The abstract syntax tree is translated into intermediate code. Our intermediate representation is based on a control/data-flow graph model. At this level we perform data-validity analysis to detect *sequential conflicts* and place *permanising registers*—latches used to store temporary values (see Section 2.2).

[4]This simple example is chosen for expository purposes only; in reality, for this trivial case, it would be beneficial to transform the SAFL into a specification where the order of access to H_f is specified statically (as in the previous example). However, in more complex cases (e.g. where the two calls to f are preceded by operations with data-dependent timings) synthesising an arbiter may lead to better performance.

- Finally we translate the intermediate graph into a hierarchical RTL Verilog design.

The remainder of this paper outlines the phases of the compiler in more depth, explaining both the technical details and motivations behind the analyses and transformation steps.

3.1 Abstract Syntax Level

As SAFL is a small language, the abstract syntax is straightforward, containing the following nodes (where, here and throughout this section, we use the notation $\vec{x}$ to abbreviate $(x_1, \ldots, x_k)$ and similarly $\vec{e}$ to abbreviate $(e_1, \ldots, e_k)$):

- variables: x; and constants: c
- user function calls: $f(e_1, \ldots, e_k)$
- primitive function calls: $a(e_1, \ldots, e_k)$, where a ranges over operators such as `+`, `-`, `<` etc.
- conditionals: `if` e_1 `then` e_2 `else` e_3
- let bindings: `let` $\vec{x} = \vec{e}$ `in` e_0 `end`

It is important to differentiate between function definitions and function calls. In order to distinguish distinct calls we assume that each abstract-syntax node is labelled with a unique identifier, α, writing $f^\alpha(e_1, \ldots, e_k)$ to indicate a call to function f at abstract-syntax node α.

4 Parallel Conflict Analysis

Parallel conflict analysis allows us to determine the set of function calls subject to parallel sharing conflicts (as outlined in Section 2.1). This analysis is performed at the abstract-syntax level and is described below.

We define a *call set* to be a set of calls. The result of Parallel Conflict Analysis is a *conflict set*: a call set containing the calls which require arbiters. For example, if the resulting conflict set is $\{f^1, f^2, f^5, g^{10}, g^{14}\}$ then we would synthesise two arbiters: one for calls $\{f^1, f^2, f^5\}$ and one for calls $\{g^{10}, g^{14}\}$.

Let e_f represent the body of function f. Let the predicate $\mathrm{RecursiveCall}(f^\alpha)$ hold iff f^α is a recursive call (i.e. occurs within e_f). $\mathcal{C}[\![e]\!]$ returns the set of non-recursive calls which may occur as a result of evaluating expression e:

$$\begin{array}{rcl}
\mathcal{C}[\![x]\!] & = & \emptyset \\
\mathcal{C}[\![c]\!] & = & \emptyset \\
\mathcal{C}[\![a(e_1, \ldots, e_k)]\!] & = & \bigcup_{1 \le i \le k} \mathcal{C}[\![e_i]\!] \\
\mathcal{C}[\![f^\alpha(e_1, \ldots, e_k)]\!] & = & (\bigcup_{1 \le i \le k} \mathcal{C}[\![e_i]\!]) \cup \left\{ \begin{array}{ll} \emptyset & \text{if RecursiveCall}(f^\alpha) \\ \{f^\alpha\} \cup \mathcal{C}[\![e_f]\!] & \text{otherwise} \end{array} \right. \\
\mathcal{C}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] & = & \bigcup_{1 \le i \le 3} \mathcal{C}[\![e_i]\!] \\
\mathcal{C}[\![\texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0]\!] & = & \bigcup_{0 \le i \le k} \mathcal{C}[\![e_i]\!]
\end{array}$$

$PC(\mathcal{S}_1, \ldots, \mathcal{S}_n)$ takes call sets, $(\mathcal{S}_1, \ldots, \mathcal{S}_n)$, and returns the conflict set resulting from the assumption that calls in each $\mathcal{S}_i$ are evaluated in parallel with calls in each $\mathcal{S}_j$ $(j \neq i)$:

$$PC(\mathcal{S}_1, \ldots, \mathcal{S}_n) = \bigcup_{i \neq j} \{f^\alpha \in \mathcal{S}_i \mid \exists \beta.\ f^\beta \in \mathcal{S}_j\}$$

We are now able to define $\mathcal{A}[\![e]\!]$ which returns the conflict set due to expression e:

$$\begin{array}{rcl}
\mathcal{A}[\![x]\!] & = & \emptyset \\
\mathcal{A}[\![c]\!] & = & \emptyset \\
\mathcal{A}[\![a(e_1,\ldots,e_k)]\!] & = & PC(\mathcal{C}[\![e_1]\!],\ldots,\mathcal{C}[\![e_k]\!]) \cup \bigcup_{1\leq i\leq k} \mathcal{A}[\![e_i]\!] \\
\mathcal{A}[\![f(e_1,\ldots,e_k)]\!] & = & PC(\mathcal{C}[\![e_1]\!],\ldots,\mathcal{C}[\![e_k]\!]) \cup \bigcup_{1\leq i\leq k} \mathcal{A}[\![e_i]\!] \\
\mathcal{A}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] & = & \bigcup_{1\leq i\leq 3} \mathcal{A}[\![e_i]\!] \\
\mathcal{A}[\![\texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0]\!] & = & PC(\mathcal{C}[\![e_1]\!],\ldots,\mathcal{C}[\![e_k]\!]) \cup \bigcup_{0\leq i\leq k} \mathcal{A}[\![e_i]\!]
\end{array}$$

Finally, for a program, p, consisting of:

- a sequence of user-function definitions: $\texttt{fun } f_1(\ldots) = e_1;\ \ldots;\ \texttt{fun } f_n(\ldots) = e_n$ and
- an initial expression, e_0

$\mathcal{A}[\![p]\!]$ returns the conflict set resulting from program, p:

$$\mathcal{A}[\![p]\!] = \bigcup_{0\leq i\leq n} \mathcal{A}[\![e_i]\!]$$

(The letter $\mathcal{A}$ is used since $\mathcal{A}[\![p]\!]$ represents the calls which require arbiters.)

5 FLaSH Intermediate Code

There are many analysis and optimisation techniques which are more suited to a lower level of representation. For this reason the FLaSH compiler translates designs into intermediate code. The intermediate code was was designed with the following aims:

- to map well onto hardware (many of the intermediate operations can be represented directly by simple circuit templates);
- to make all control and data dependencies explicit; and
- to facilitate analysis and transformation.

As in many compilers, the intermediate representation is a graph-like structure best modeled as sets of nodes and edges.

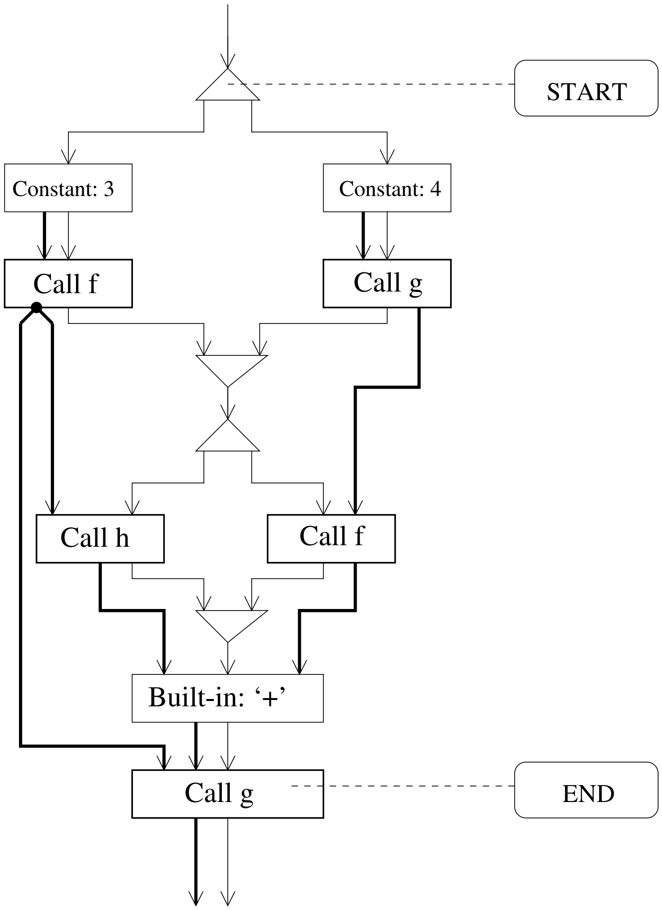
5.1 The structure of intermediate graphs

An intermediate graph can be represented as $(\mathcal{N}, E_c, E_d)$ where:

$\mathcal{N}$ is a set of nodes
$E_c \subseteq \mathcal{N}\times\mathcal{N}$ is a set of *control edges* i.e. $(n,n') \in E_c \Leftrightarrow$ control flows out of n into n'
$E_d \subseteq \mathcal{N}\times\mathcal{N}$ is a set of *data edges* i.e. $(n,n') \in E_d \Leftrightarrow$ data flows out of n into n'

We define functions to compute successors/predecessors as follows:

$$\begin{array}{rcl}
Succ_c(n) & = & \{n' \mid (n,n') \in E_c\} \\
Succ_d(n) & = & \{n' \mid (n,n') \in E_d\} \\
Pred_c(n) & = & \{n' \mid (n',n) \in E_c\} \\
Pred_d(n) & = & \{n' \mid (n',n) \in E_d\}
\end{array}$$



This intermediate graph represents the following expression:

```
let var x = f(3)
    var y = g(4)
 in g(x, h(x) + f(y))
end
```

Figure 2: Example intermediate graph

Node Type	*Number of Control*		*Number of Data*	
	Inputs	*Outputs*	*Inputs*	*Outputs*
CONTROL_FORK	1	≥ 2	0	0
CONTROL_JOIN	≥ 2	1	0	0
CONSTANT	1	1	0	1
BUILT_IN	1	1	≥ 1	1
CALL	1	1	≥ 0	1
JUMP	1	1	≥ 0	1
READ_FORMAL	1	1	0	1
CONDITIONAL_SPLIT	1	2	1	0
CONDITIONAL_JOIN	2	1	3	1

Figure 3: Nodes used in intermediate graphs

Intermediate graphs are best viewed pictorially. We adopt the convention that thin lines represent control edges and thick lines represent data edges. Figure 2 gives an example of an intermediate graph and the SAFL expression it represents. The types of node used in intermediate graphs are summarised in Figure 3. Given a node n, we define the formula (n : CALL f) to hold iff n is a call node (and similarly for other node forms).

As can be seen from Figure 3, nodes have at most one data output-port. We say that a node n is a *data-producer* if it has a data output-port. If n is a data-producer then we define $n.DataOut$ to refer to n's (single) data output-port. We define R^+ to be the transitive closure of relation R; similarly $R^\star$ is the reflexive-transitive closure. For example, $Succ_c^+(n)$ is the set of nodes which occur after n on the control path.

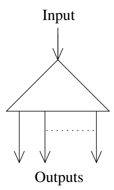
When compiling an expression to an intermediate graph, we mark two distinguished intermediate nodes: *start* and *end*. The node marked *start* represents an expression's entry point and the node marked *end* is a data-producer whose value will ultimately yield the result of the expression.

The meanings of nodes (and their pictorial representations) are outlined informally in the following sections. In order to describe the semantics of nodes we often talk of *control* and the act of *propagating control*. This will become clearer when we outline the translation to hardware, but until then it may help the reader to imagine control edges propagating events cf. asynchronous circuit design.

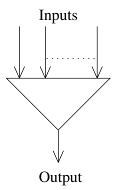
5.1.1 Fork/Join Parallelism

Parallelism is made explicit through control fork/join nodes:

CONTROL_FORK



CONTROL_JOIN



- When control reaches a CONTROL_FORK node's single input then control is propagated to its many control outputs.
- Conversely, a CONTROL_JOIN node waits for control to arrive at all its inputs, before propagating control to its single output.

Examples of the use of CONTROL_FORK and CONTROL_JOIN can be seen in Figure 2 where they are used to facilitate the parallel evaluation of `let`-declarations and function arguments.

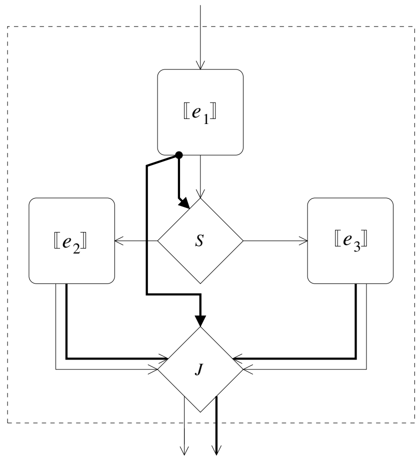
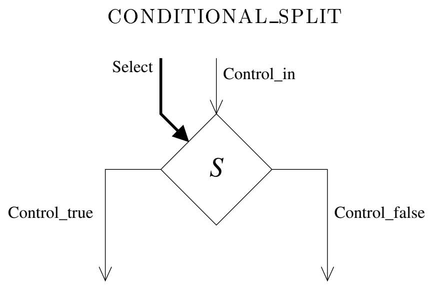
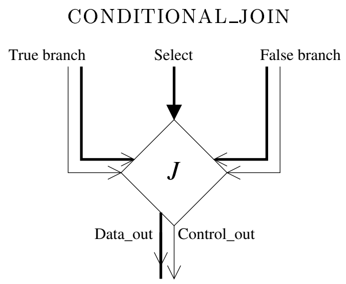


Figure 4: Conditional translation: ⟦`if` e_1 `then` e_2 `else` e_3⟧

5.1.2 Conditionals

We represent conditional execution using two nodes: CONDITIONAL_SPLIT and CONDITIONAL_JOIN. The style of these nodes may seem unusual to people familiar with software compilers, but they map well onto hardware. In particular the CONDITIONAL_JOIN node mimics a multiplexor.



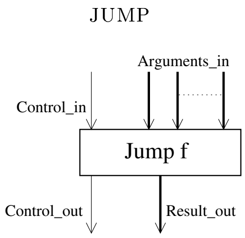


- A CONDITIONAL_SPLIT node channels control from *control_in* to either *control_true* or *control_false* output depending on the boolean data value on the *select* input.
- The CONDITIONAL_JOIN node has two control/data input pairs corresponding to the true and false branches of a corresponding CONDITIONAL_SPLIT. When control arrives at either control input it is propagated to *control_out*. Similarly, data on one of the two data-inputs is propagated to *data_out*. The boolean value on *select* is used to determine which of the data values to forward.

Figure 4 shows how conditionals are translated into intermediate structures.

5.1.3 Recursive calls

We use the JUMP node to represent recursive calls. (Recall that we can always implement recursive calls as jumps because all recursive calls are restricted to tail-calls.)



A JUMP returns no data and never propagates control—hence its outputs are ignored; we can treat the data output as some random undefined value and the control output as never asserted. In practice, control and data outputs are not realised at the hardware level. (We only include them at the intermediate level because it allows us to maintain the invariant that a closed expression is represented by an intermediate graph with a control input, a control output and a data output).

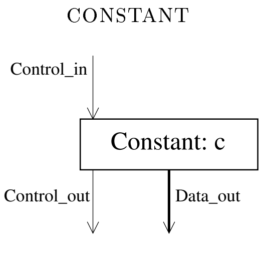
5.1.4 Primitive and User-defined Function Calls

Although they are treated very differently at the hardware level, primitive function calls and user-defined function calls look similar at the intermediate level:

Both these nodes read their data-inputs when control reaches their single control-input, perform their operation and then return their result on the single data-output, propagating control forwards when the operation is complete. (Note that the CALL node is only used to represent non-recursive calls to external functional-units. We have already shown how we use JUMP to represent recursive calls).

5.1.5 Constants

The CONSTANT node simply propagates control whilst continually writing its value, c, onto its data-output.



5.1.6 Variables

Variables can be subdivided into two separate categories:

- `let`-bound variables
- formal parameters

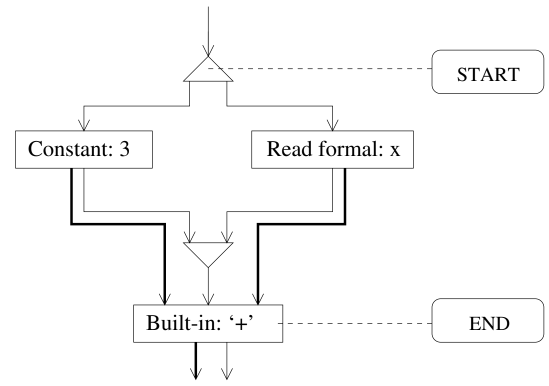
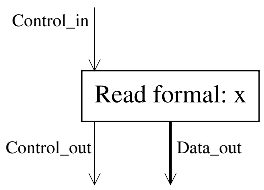


Figure 5: Intermediate graph representing the body of `fun f(x) = x+4`

Although `let`-bound variables are represented implicitly by sharing a node's data-output (see Figure 2) we require a special node to deal with formal parameters:

READ_FORMAL



This node propagates control whilst writing the value of the local function's formal parameter, x, to its data-output. Figure 5 shows an example of the READ_FORMAL node by translating the body of `fun f(x) = x+4`.

5.2 Translation to Intermediate Code

The main translation phase is implemented by a recursive function which walks the abstract-syntax tree, constructing parts of intermediate graph and then gluing them together. Although the technicalities of the translation procedure are omitted there are some issues worth mentioning:

1. Intermediate graphs represent expressions rather than programs. Thus, in order to compile a SAFL program into intermediate form we compile each of the function bodies separately and maintain a list of (function name, intermediate expression) pairs. The *start* and *end* nodes of the intermediate expression then correspond to the entry and exit points of the function.

2. We translate abstract syntax expressions into intermediate graphs where all `let`-declarations, primitive function call arguments and user-defined function call arguments are evaluated in parallel. (This parallelism is made explicit through the use of CONTROL_SPLIT and CONTROL_JOIN nodes.)

3. We perform a simple reachability analysis and dead-code elimination phase to remove redundant nodes from the graph before any analysis takes place. For example consider the intermediate graph corresponding to the body of:

```
fun f(x) = if ... then f(x+1)
                  else f(x-1)
```

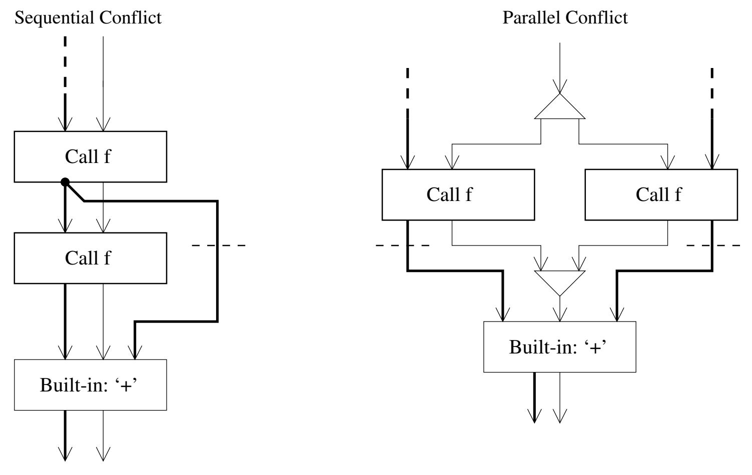


Figure 6: A sequential conflict (left) and a parallel conflict (right). The horizontal dotted lines show the points where data may become invalid. These are the points where permanising registers are required

In this case, since both of the conditional branches contain tail recursive calls (represented as JUMP nodes) we know that the corresponding CONDITIONAL_JOIN node will never be reached and is hence unnecessary.

6 Analysis at the Intermediate Code Level

The intermediate level analyses described here concern the placement of *permanising registers* as outlined in Section 2.2. Recall that there are two types of conflict which require the insertion of permanising registers:

- *Sequential conflicts.* These arise when the output of a call to f is read after a following call to f may have already occurred. The problem is that since both calls to f access the same shared resource, H_f, the second call changes the value of H_f's data output thus invalidating the result of the first call.
- *Parallel conflicts.* These arise when there are multiple parallel calls to the same function. Since we do not know the order in which the calls will occur we have to place permanising registers after both call nodes since either may be corrupted by the other.

An example of both a sequential conflict and a parallel conflict is shown in Figure 6. The horizontal dotted lines show the points where data may become invalid. It is at these points that permanising registers must be placed. In order to represent permanising registers at the intermediate level we introduce a new node which models a latch:

Node Type	*No. Control*		*No. Data*	
	Inputs	*Output*	*Inputs*	*Outputs*
PERMANISOR	1	1	1	1

On receiving control, this node latches its data input, propagating control once the data output (read directly from the latch) is valid. Once we have determined which data-edges require permanising we can insert PERMANISOR nodes using the transformation shown in Figure 7. The transformation is based on the observation that multiple data-edges originating at a single node can all share a permanisor if necessary.

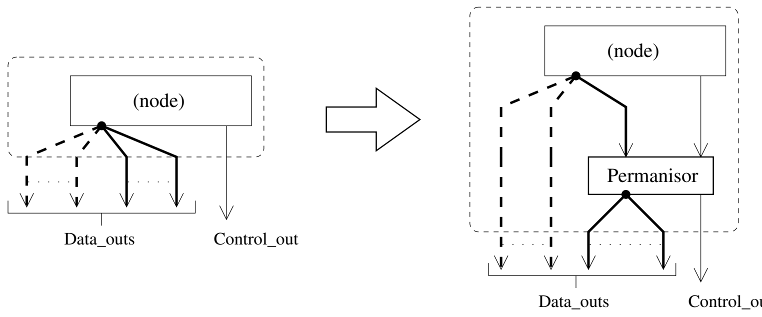


Figure 7: We insert permanisors on data-edges using this transformation. The dashed data-edges represent those which do not require permanisors; the solid data-edges represent those which do require permanisors.

Section 4 has already described an analysis for detecting parallel conflicts; now we present a data-flow style analysis over intermediate graphs which allows us to infer which data-edges require permanising due to *sequential conflicts*.

The analysis presented here assumes that hardware-level functional-units respect the following invariants:

Invariant 1 *After a call to a functional-unit, H_f, the data on H_f's (shared) output remains valid until another call to H_f occurs.*

Invariant 2 *Functions latch their arguments when called.*

Invariant 2 (analogous to *callee-save* in software compilers) means that the caller does not have to worry about keeping arguments valid *throughout* the duration of a call; arguments need only be valid at the point of call. (In Section 7 we show how to modify the analysis to deal with a hybrid of both *callee-save* and *caller-save* policies.)

The register placement analyses are presented in three stages:

Resource Dependency Analysis tags each data-producing node, n, with the set of functional-units that $n.DataOut$ *depends* upon. (We say that a node, n, *depends* on a functional-unit, H_f, iff changing the value on H_f's (shared) output may invalidate the value of $n.DataOut$.)

Validity Analysis tags each node, n, with the set of nodes whose data output is guaranteed to be valid when control reaches n.

Sequential Conflict Register Placement uses validity information to decide which data-edges require permanising registers to resolve sequential conflicts.

These analyses assume that permanisors have already been inserted to ensure the validity of function calls subject to parallel sharing conflicts (see Section 4). The equations below require information about parallel sharing conflicts and the structure of the call-graph; thus we make the following definitions:

Definition 1 *Given a* CALL *node, n, predicate $HasParConflict(n)$ holds iff n has a parallel sharing conflict.*

Definition 2 *CG is the call-graph relation of the program being translated. Thus, $CG(f)$, returns the functions that f calls directly and $CG^{\star}(f)$ returns all the functions which may be invoked as a result of invoking f.*

It is also worth mentioning that whilst parallel conflict analysis is an *inter*-procedural analysis, dealing with the structure of the whole program, sequential conflict analysis is *intra*-procedural, concerned with placing registers within each function separately.

Dataflow equations for the register placement analysis are summarised in Figure 10. The following sections clarify some of the terminology and describe the intuition behind the equations:

6.1 Resource Dependency Analysis

Recall that a data-producing node, n, *depends* on a functional-unit, H_f, iff changing the value on H_f's (shared) output may invalidate the value of $n.DataOut$. The resource dependency equations map a node, n, onto the set of functional-units on which n depends.

Definition 3 *Given a node, n, $\mathcal{D}_{out}(n)$ is the set of (names of) functional-units that n depends on.*

$$\mathcal{D}_{out} : \mathit{Node} \to \mathcal{P}(\mathit{Functional\text{-}unit\ name})$$

$$\mathcal{D}_{out}(n) = \begin{cases} \emptyset & \text{if } (n : \textsc{jump}) \vee [(n : \textsc{call}\ f) \wedge \mathit{HasParConflict}(n)] \\ \mathit{CG}^{\star}(f) & \text{if } (n : \textsc{call}\ f) \wedge \neg \mathit{HasParConflict}(n) \\ \bigcup_{p \in \mathit{Pred}_d(n)} \mathcal{D}_{out}(p) & \text{otherwise} \end{cases}$$

The resource dependency equations reflect the following observations:

1. If n is a CALL node (n : CALL f) subject to a parallel sharing conflict then we assume that a permanising register has already been inserted. Hence $n.DataOut$ has been decoupled from H_f and is not dependent on any functional-units.

2. If n is a CALL node (n : CALL f) and n is not subject to any parallel sharing conflicts then $n.DataOut$ is dependent only on f and the functional-units that f may access. We know that $n.DataOut$ is not dependent on any of its data-predecessors since H_f latches these values at the beginning of the call decoupling n from all $n' \in \mathit{Pred}_d(n)$.

3. If n is not a CALL node then it is dependent on the same functional-units as its data-predecessors since changes to the data-outputs of any $n' \in \mathit{Pred}_d(n)$ may be propagated through to $n.DataOut$.

6.2 Data Validity Analysis

The data validity equations form the core of the register placement analysis. Defined mutually recursively, $\mathcal{V}_{in}$ and $\mathcal{V}_{out}$ map a node, n, onto the set of nodes whose data-output is *guaranteed* to be valid when control respectively reaches and leaves n. However, before launching into the data validity equations we must first introduce some auxiliary definitions. Definitions 4 and 5 formalise the notion of a *thread* allowing us to reason precisely about parallelism (they are shown diagrammatically in Figure 8). Definition 6 defines *kill* which maps a node, n, onto the set of nodes whose data outputs are invalidated as a result of control passing through n.

Definition 4 *Given a* CONTROL_SPLIT *node, n, $Join(n)$ is the corresponding* CONTROL_JOIN *node.*[5]

[5]Due to the properties of the translation to intermediate code each CONTROL_SPLIT node has a corresponding CONTROL_JOIN node (cf. *bras* and *kets* in a well-bracketed string).

Definition 5 *Given a* CONTROL_SPLIT *node, n, such that $Succ_c(n) = \{s_1, \ldots, s_k\}$, let $thread_i = Succ_c^\star(s_i) \cap Pred_c^+[Join(n)]$ (for $1 \leq i \leq k$). Then, $\pi(n, s_i)$ is the set of nodes in each of n's threads except the thread containing node s_i:*

$$\pi(n, s_i) = \bigcup_{j \neq i} thread_j$$

Definition 6 *Given a node n, $kill(n)$ is the set of nodes whose data outputs are invalidated as a result of control passing through n:*

$$kill : Node \to \mathcal{P}(Node)$$

$$kill(n) = \begin{cases} \{n' \neq n \mid \mathcal{D}_{out}(n') \cap CG^\star(f) \neq \emptyset\} & \text{if } (n : \text{CALL } f) \\ \emptyset & \text{otherwise} \end{cases}$$

The equations for *kill* reflect the following observations:

- The only way a node's data output can be invalidated by executing n is if n invokes some shared resource. Thus if n is not a CALL node then nothing can be invalidated.
- If n is a call node (n:CALL f) then every node which is dependent on something which n may invoke (either directly or indirectly) is invalidated.

The data validity equations are now presented:

Definition 7 *Given a node, n, $\mathcal{V}_{in}(n)$ is the set of nodes whose data-output is guaranteed to be valid when control* reaches *n.*

Definition 8 *Given nodes n and s, $\mathcal{V}^s_{out}(n)$ is the set of nodes which are guaranteed to be valid when control* leaves *n along edge $(n, s) \in E_c$*

For the sake of clarity, in cases where we know that n has only one control successor (i.e. $Succ_c(n) = \{n'\}$), we write $\mathcal{V}^\diamond_{out}(n)$ to mean $\mathcal{V}^{n'}_{out}(n)$.

$$\begin{aligned} \mathcal{V}_{in} &: Node \to \mathcal{P}(Node) \\ \mathcal{V}^s_{out} &: Node \to \mathcal{P}(Node) \end{aligned}$$

$$\mathcal{V}_{in}(n) = \begin{cases} \displaystyle\bigcap_{p \in Pred_c(n)} \mathcal{V}^\diamond_{out}(p) & \text{if } (n : \text{CONDITIONAL\_JOIN}) \\ \displaystyle\bigcup_{p \in Pred_c(n)} \mathcal{V}^n_{out}(p) & \text{otherwise} \end{cases}$$

$$\mathcal{V}^s_{out}(n) = \mathcal{V}_{in}(n) \cup \{n\} \setminus \begin{cases} \displaystyle\bigcup_{n' \in \pi(n,s)} kill(n') & \text{if } (n : \text{CONTROL\_SPLIT}) \\ kill(n) & \text{otherwise} \end{cases}$$

The intuition behind $\mathcal{V}_{in}(n)$ is as follows:

1. If n is a CONDITIONAL_JOIN node then at run-time control will arrive at n from *either* its true-branch *or* its false-branch. Thus the nodes guaranteed to be valid when control reaches n are those that are guaranteed to be valid at *both* the end of the true-branch *and* the end of the false-branch.
2. If n is not a CONDITIONAL_JOIN node then the nodes that are guaranteed to be valid when control reaches n are those that were guaranteed to be valid just after n's control-predecessors have been executed.

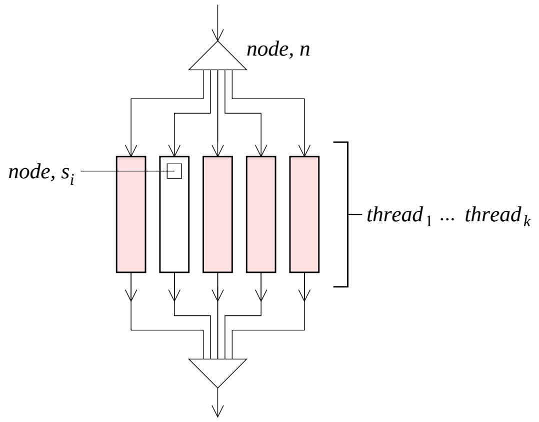


Figure 8: The nodes contained in the highlighted threads are those returned by $\pi(n, s_i)$

$\mathcal{V}^s_{out}(n)$ reflects the following intuition:

- If n is not a CONTROL_SPLIT node then the nodes guaranteed to be valid when control leaves n are precisely:
  - those nodes which were valid when control arrived at n;
  - plus n itself;
  - minus the nodes that were invalidated as a result of executing n. i.e. those nodes in $\mathit{kill}(n)$
- If n is a CONTROL_SPLIT node then things are a little more complicated since we have to cope with the parallelism that n introduces. As we do not know statically which interleaving of parallel operations will occur at run-time we are forced to make a *safe* approximation: when analysing a particular thread (the one containing s_i—see Figure 8) we assume that every other parallel thread has already been executed (i.e. we assume that any nodes whose data *may* be invalidated *has* been invalidated!).

6.3 Sequential Conflict Register Placement

Sequential conflict register placement is the process where we decide which data-edges $(n, n') \in E_d$ require registers to resolve sequential conflicts. We define a predicate $\mathit{Perm}(n, n')$ which holds iff data-edge (n, n') requires a permanisor. As a first approximation, we simply observe that if a node n is not guaranteed to be valid at n' then we must place a permanising register on data-edge (n, n'):

$$\forall (n, n') \in E_d.\ \mathit{Perm}(n, n') \Leftrightarrow n \notin \mathcal{V}_{in}(n')$$

Although this works, we can improve the accuracy of our model (i.e. make it insert considerably fewer permanisors) by giving CONDITIONAL_JOIN nodes a special treatment:

Definition 9 *Given a* CONDITIONAL_JOIN *node, n', and a node, n, which occurs before n' on the control path,*[6] *$\mathcal{RV}_{in}(n, n')$ 'Relative-$\mathcal{V}_{in}$' is the set of nodes guaranteed to be valid on entry to n' given the extra information that control passes through node n.*

$$\mathcal{RV}_{in}(n, n') = \bigcap_{n'' \in [\mathit{Succ}^\star_c(n) \cap \mathit{Pred}_c(n')]} \mathcal{V}^\diamond_{out}(n'')$$

[6] I.e. $n' \in \mathit{Succ}^+_c(n)$.

The vertically hatched nodes are those in $Succ_c^\star(n)$; the horizontally hatched nodes correspond to $Pred_c(n')$. Only control flow is shown.

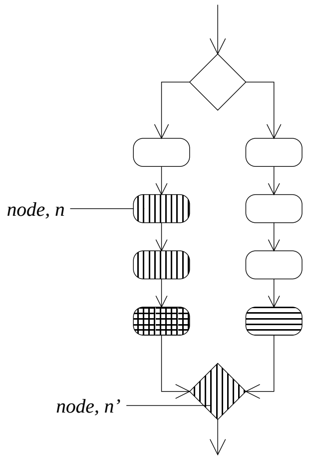


Figure 9: Diagrammatic explanation of $Succ_c^\star(n) \cap Pred_c(n')$

This is based on the observation that if we know which way a CONDITIONAL_SPLIT has branched, we can do much better at predicting the nodes that are going to be valid at the corresponding CONDITIONAL_JOIN: the nodes which are valid at n' are those which are valid at the end of the conditional branch in which n occurs. We use the equation $Succ_c^\star(n) \cap Pred_c(n')$ to calculate the final node in the conditional branch containing n (shown graphically in Figure 9). Note that if n is not in either of the conditional branches joining at n' then (since n occurs before n' on the control path) $Succ_c^\star(n) \cap Pred_c(n') = Pred_c(n')$. Thus, in this case $\mathcal{RV}_{in}(n, n')$ reduces to $\mathcal{V}_{in}(n')$ (intuitively this is what we expect: if n is not in either of the conditional branches joining at n' then we have not gained any extra information by stating that control passes through n.)

Now we can use $\mathcal{RV}_{in}$ to define a more accurate version of $Perm$ as follows:

Definition 10 *$Perm(n, n')$ holds iff data-edge (n, n') requires* permanising*:*

$$\forall (n, n') \in E_d.\ Perm(n, n') \Leftrightarrow \begin{cases} n \notin \mathcal{RV}_{in}(n, n') & \text{if } (n' : \text{CONDITIONAL\_JOIN}) \\ n \notin \mathcal{V}_{in}(n') & \text{otherwise} \end{cases}$$

7 Extending the model: Calling conventions

The equations presented in Figure 10 assume a *callee-save* model. In a hardware implementation this corresponds to every functional-unit latching its arguments on a call. Often these latches are unnecessary and substantial savings can be made by adopting a *caller-save* model, where functional-units do not latch their arguments but make the assumption that the caller will keep the arguments valid throughout the duration of the call.

In this section we show how we can adjust our data-flow analyses to cope with a combination of both callee-save and caller-save conventions. Let predicate $CalleeSave(f)$ hold for a function, f, iff f adopts a callee-save model. In this way we can specify on a per-resource basis which functional-units latch their arguments and which functional-units require their arguments to remain valid throughout a call (we can use a suitable pragma to convey this information to the compiler). Adopting this policy has proved very useful since selecting the right combination of callee- and caller-save conventions allows one to considerably reduce the area of a final circuit. Thus, for caller-save functional-units, our invariants (see Section 6) become:

Invariant 3 *After a call to a (caller-save) functional-unit, H_f, the data on H_f's (shared) output remains valid until either (i) another call to H_f occurs or (ii) the values on H_f's inputs change.*

Resource Dependency Analysis

$$\mathcal{D}_{out} : Node \rightarrow \mathcal{P}(Function\ name)$$

$$\mathcal{D}_{out}(n) = \begin{cases} \emptyset & \text{if } (n : \textsc{jump}) \vee [(n : \textsc{call}\ f) \wedge \mathit{HasParConflict}(n)] \\ CG^{\star}(f) & \text{if } (n : \textsc{call}\ f) \wedge \neg \mathit{HasParConflict}(n) \\ \bigcup_{p \in Pred_d(n)} \mathcal{D}_{out}(p) & \text{otherwise} \end{cases}$$

Data Validity Analysis

$$\begin{array}{ll} \mathcal{V}_{in} & : Node \rightarrow \mathcal{P}(Node) \\ \mathcal{V}^{s}_{out} & : Node \rightarrow \mathcal{P}(Node) \end{array}$$

$$\mathcal{V}_{in}(n) = \begin{cases} \bigcap_{p \in Pred_c(n)} \mathcal{V}^{\diamond}_{out}(p) & \text{if } (n : \textsc{conditional\_join}) \\ \bigcup_{p \in Pred_c(n)} \mathcal{V}^{n}_{out}(p) & \text{otherwise} \end{cases}$$

$$\mathcal{V}^{s}_{out}(n) = \mathcal{V}_{in}(n) \cup \{n\} \setminus \begin{cases} \bigcup_{n' \in \pi(n,s)} \mathit{kill}(n') & \text{if } (n : \textsc{control\_split}) \\ \mathit{kill}(n) & \text{otherwise} \end{cases}$$

where $\pi(n, s)$ (see Definition 5 and Figure 8) is the set of nodes in each of CONTROL_SPLIT node n's threads except the thread containing node s.

and $\mathit{kill}(n)$ is the set of nodes whose data outputs are invalidated as a result of control passing through n:

$$\mathit{kill} : Node \rightarrow \mathcal{P}(Node)$$

$$\mathit{kill}(n) = \begin{cases} \{n' \neq n \mid \mathcal{D}_{out}(n') \cap CG^{\star}(f) \neq \emptyset\} & \text{if } (n : \textsc{call}\ f) \\ \emptyset & \text{otherwise} \end{cases}$$

$\mathcal{RV}_{in}(n, n')$ is the set of nodes valid on entry to the CONDITIONAL_JOIN node n' given the extra information that control passes through node n.

$$\mathcal{RV}_{in}(n, n') = \bigcap_{n'' \in [Succ^{\star}_c(n) \cap Pred_c(n')]} \mathcal{V}^{\diamond}_{out}(n'')$$

Sequential Conflict Register Placement

$$\forall (n, n') \in E_d.\ Perm(n, n') \Leftrightarrow \begin{cases} n \notin \mathcal{RV}_{in}(n, n') & \text{if } (n' : \textsc{conditional\_join}) \\ n \notin \mathcal{V}_{in}(n') & \text{otherwise} \end{cases}$$

Figure 10: Summary: Register Placement for Sequential Conflicts

Invariant 4 *The caller keeps the arguments valid throughout the duration of the call.*

It turns out that we only have to modify the resource dependency analysis and the permanisation analysis; validity analysis remains unchanged.

7.1 Extended Resource Dependency Analysis

We add an extra clause to the definition of $\mathcal{D}_{out}$ (see Section 6.1) reflecting the observation that for a node n : CALL f, where H_f does not latch its arguments, n is dependent upon:

- the functional-units that n's data predecessors are dependent upon; and
- all the functional-units that H_f may invoke

$$\mathcal{D}_{out}(n) = \begin{cases} \emptyset & \text{if } (n : \text{JUMP}) \vee [(n : \text{CALL } f) \wedge \mathit{HasParConflict}(n)] \\ \mathit{CG}^\star(f) & \text{if } (n : \text{CALL } f) \wedge \neg \mathit{HasParConflict}(n) \wedge \mathit{CalleeSave}(f) \\ \mathit{CG}^\star(f) \cup \bigcup_{p \in \mathit{Pred}_d(n)} \mathcal{D}_{out}(p) & \text{if } (n : \text{CALL } f) \wedge \neg \mathit{HasParConflict}(n) \wedge \neg \mathit{CalleeSave}(f) \\ \bigcup_{p \in \mathit{Pred}_d(n)} \mathcal{D}_{out}(p) & \text{otherwise} \end{cases}$$

7.2 Extended Permanisation Analysis

We update our definition of *Perm* to reflect the observation that if we are not dealing with a callee-save function, it is the duty of the caller to keep the function arguments valid until after the call.

$$\forall (n, n') \in E_d.\ \mathit{Perm}(n, n') \Leftrightarrow \begin{cases} n \notin \mathcal{RV}_{in}(n, n') & \text{if } (n' : \text{CONDITIONAL\_JOIN}) \\ n \notin \mathcal{V}^\diamond_{out}(n') & \text{if } (n' : \text{CALL } f) \wedge \neg \mathit{CalleeSave}(f) \\ n \notin \mathcal{V}_{in}(n') & \text{otherwise} \end{cases}$$

8 Translation to Hardware

Having performed analyses and transformation at the intermediate level, we compile intermediate graphs into hierarchical RTL Verilog suitable for simulation or synthesis using existing tools. This section describes how the FLaSH compiler maps intermediate graphs onto hardware. We outline the particular design style currently adopted by our compiler; targeting the system to produce different circuit styles is the topic of future work.

The generated hardware is synchronous and based on a matched-delay protocol where each group of data-wires is bundled with a control wire. Control wires propagate events which, in this framework, are represented as one-cycle pulses. The circuits are designed so that control events propagate at the same speed as valid data. When the control wire is high the corresponding data wires are guaranteed to be valid; how long they remain valid for after the control wire falls is determined by validity analysis (see Section 6.2 and Figure 11).

Firstly we discuss how expressions are compiled; Section 8.2 explains how function definitions are compiled into hardware functional-units.

Data is always guaranteed to be valid when the control pulse (one cycle) is high. Validity analysis allows us to infer how long the data remains valid after the control pulse falls (time t').

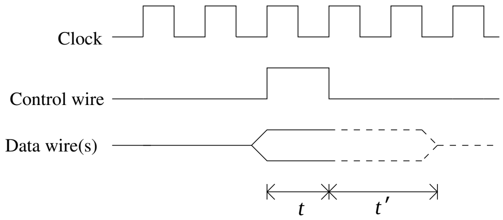


Figure 11: A lower-level view of validity analysis

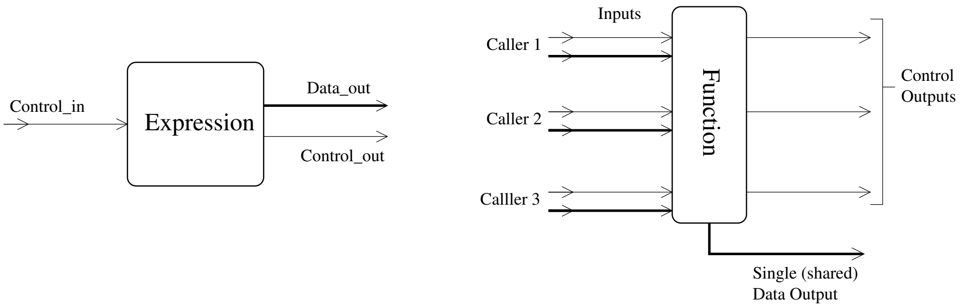


Figure 12: Expressions and Functions

8.1 Compiling Expressions

A closed expression is compiled into a hardware-block with one control input, one control output and one data output—see Figure 12 (left). Signalling an event on the control input triggers the expression which computes its value and places its result on the data output, signalling an event on its control output when the data output becomes valid.

CALL nodes are synthesised into connections to other hardware level functional-units; JUMP nodes (representing recursive calls) are synthesised into a connection back into the current functional unit; all other intermediate nodes are translated into a corresponding circuit template. Figure 14 shows the circuit templates corresponding to conditional split and join nodes; Figure 15 shows the hardware block corresponding to a CONTROL_JOIN node.[7] In many cases we can optimise CONTROL_JOINs away by performing timing analysis to infer that control will arrive at each of the control inputs simultaneously. Similarly, if we can infer which control input will be asserted last, the CONTROL_JOIN circuit reduces to a wire!

Other nodes have their obvious translations. For example a CONTROL_SPLIT node is just a wire connecting its control input to all its control outputs, and a BUILT_IN: `<op>` node contains combinatorial logic to perform operation `<op>`.

8.2 Compiling Functions

A function *definition* is compiled into a single hardware-block (functional-unit) with multiple control and data inputs: one control/data input-pair for each call—see Figure 12 (right). There are multiple control outputs (one to return control to each caller), but only a single data output (which is shared between all callers). Each function contains logic to compute its body expression.

8.2.1 Calling Protocol

Figure 16 shows how functional-units are composed to form larger structures. In this example functional-unit H_f is shared between H_g and H_h. Notice how H_f's data output is shared, but the

[7]Some of our schematics use synchronous reset-dominant SR flip-flops. Figure 13 shows how these can be constructed from the more familiar D-type flip-flop.

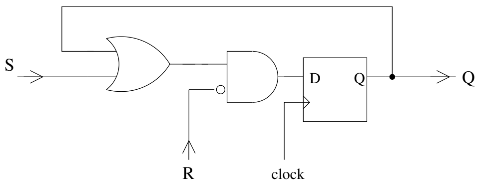


Figure 13: How to build a synchronous reset-dominant SR flip-flop from a D-type

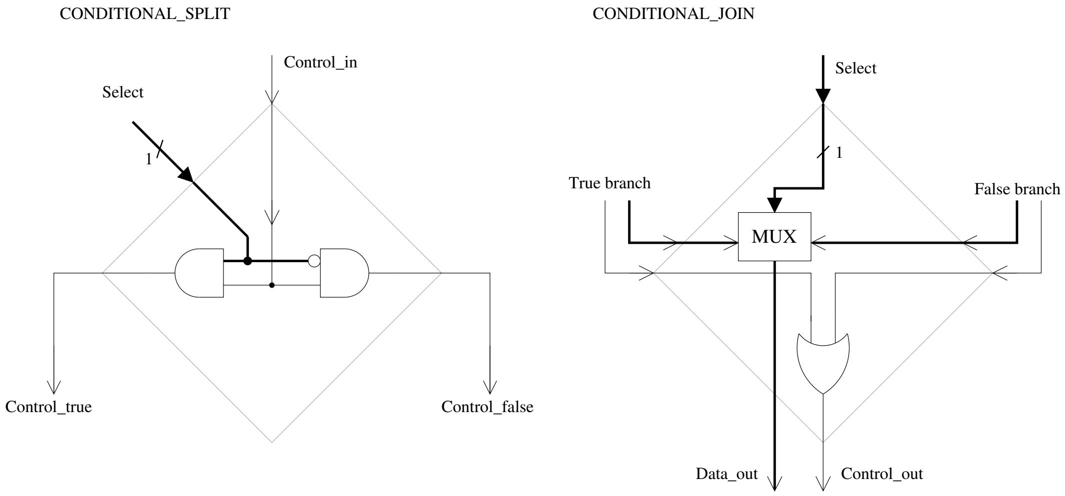


Figure 14: Hardware blocks corresponding to CONDITIONAL_SPLIT (left) and CONDITIONAL_JOIN (right) nodes

control structure is duplicated on a per call basis.

To perform a call to resource H_f the caller places the argument values on its data input into H_f before triggering a call event on the corresponding control input. Some point later, when H_f has finished computing, the result of the call is placed on H_f's shared data-output and an event is generated on the corresponding control output.

8.2.2 Function Unit Internals

The internals of a functional-unit are shown in Figure 17. To simplify the presentation we show a functional-unit that contains registers which latch the incoming arguments on a call. (Recall that this corresponds to the *callee-save* policy discussed in Section 7). The functional-unit's operation is discussed below.

First consider the control path. Each control/data input-pair to a functional-unit, H_f, corresponds to a single call. If any of the control inputs may trigger calls in parallel (as inferred by the analysis in Section 4) then these control wires are passed through an arbiter (priority encoder with arbitrary priority in the synchronous case) which ensures that only one call is dealt with at a time.

Having been through the arbiter, the control inputs are fed into the *External Call Control Unit* (ECCU—see Figure 18), which:

1. remembers which of the control inputs triggered the call;

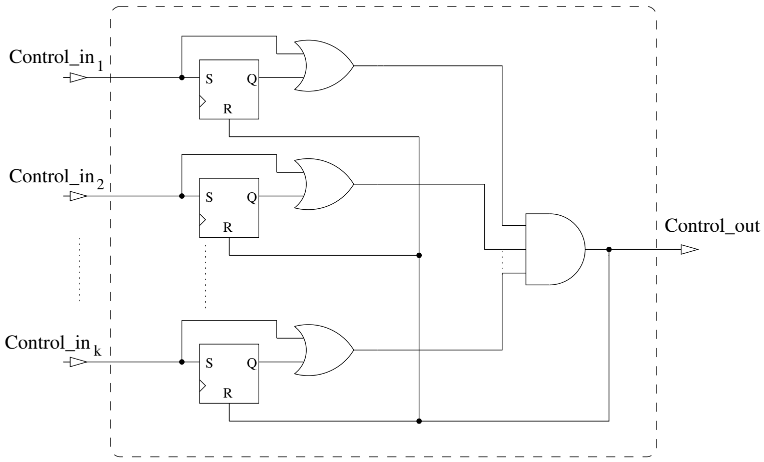


Figure 15: Hardware block corresponding to a CONTROL_JOIN node

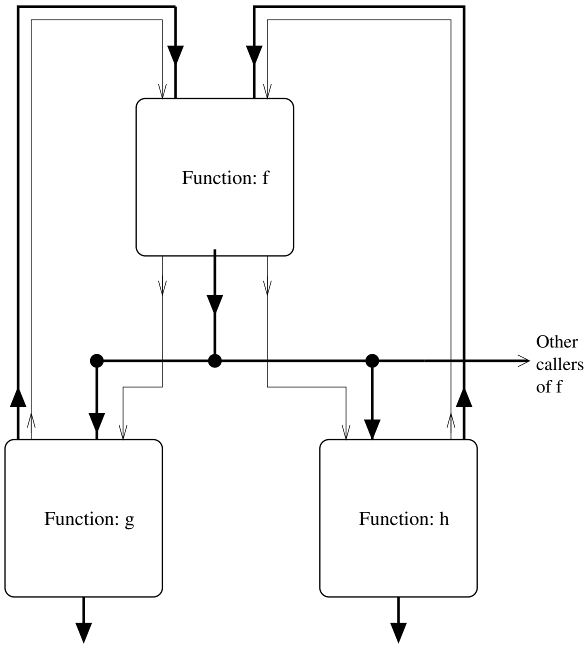


Figure 16: Function Connectivity

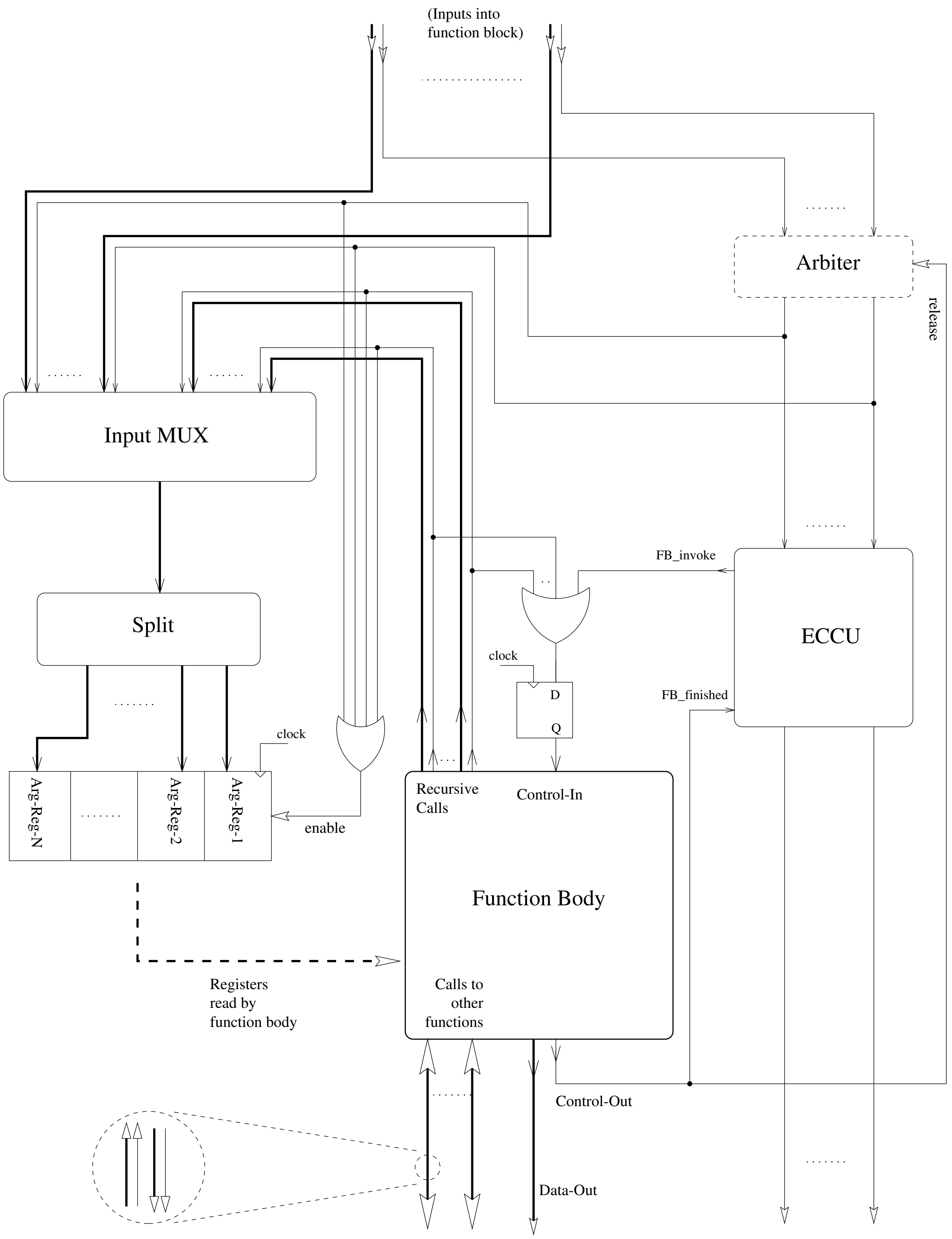


Figure 17: A Block Diagram of a Hardware Functional-Unit

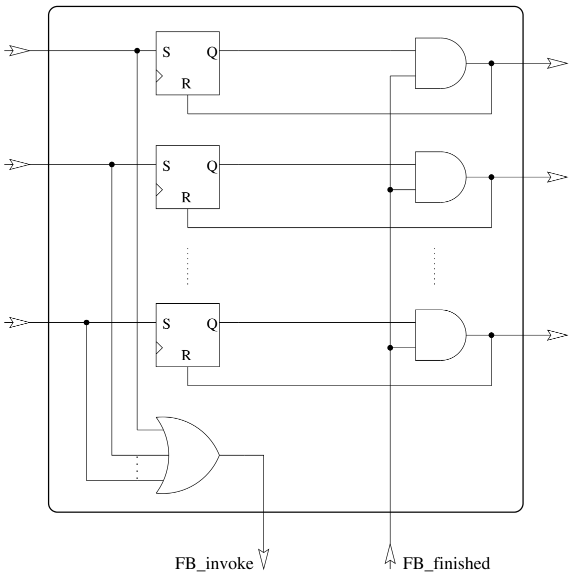


Figure 18: The Design of the External Call Control Unit (ECCU)

2. invokes the function body expression (by generating an event on the `FB_invoke` wire);
3. waits for completion (signalled by the `FB_finished` wire); and finally
4. generates an event on the corresponding control output, signalling to the caller that the result is waiting on H_f's shared data output.

Now let us consider the data-path. The data inputs are fed into a multiplexor which uses the corresponding control inputs as select lines. The selected data input is latched into the argument registers. (Obviously, the *splitter* is somewhat arbitrary; it is included in the diagram to emphasise that multiple arguments are all placed on the single data-input). Note that recursive calls feed back into the multiplexor to create loops, re-triggering the body expression as they do so. The D-type flip-flop is used to insert a 1-cycle delay onto the control path to match the 1-cycle delay of latching the arguments into the registers on the data-path.

The function body expression contains connections to the other functional-units that it calls. These connections are the ones marked "calls to other functions" in Figure 17 and are seen in context in Figure 16.

8.3 Generated Verilog

The FLaSH compiler produces RTL Verilog to express the generated hardware. The Verilog is hierarchical in the sense that each *function* definition in the SAFL source has a corresponding *module* definition in the generated Verilog.

This is useful for two reasons:

- it makes the Verilog more readable, since the high-level circuit structure is explicit; and
- a hierarchical design is useful when performing technology mapping, since typical RTL compilers operate more efficiently when provided with modular designs. Furthermore, typical logic synthesisers often allow users to specify optimisation priorities (e.g. area vs time) on a per-module basis.

Data is always guaranteed to be valid between request and acknowledge events. Validity analysis allows us to determine how long the data remains for after the control pulse falls (time t').

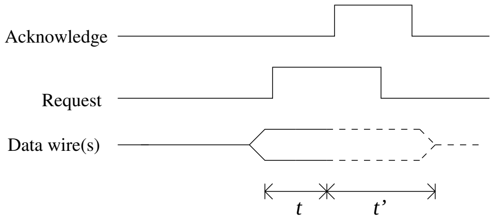


Figure 19: Applying validity analysis to a 4-phase request/acknowledge handshake protocol

9 Conclusions and Future Directions

We have outlined the design of a silicon compiler for producing hardware directly from functional specifications. Early experiments show that the techniques presented here can produce efficient hardware both in terms of time and area. However, all the examples we have tried so far are quite small, more work is needed to investigate how our compiler scales to larger design projects. We also intend to explore other hardware styles by retargetting the FLaSH compiler to produce different types of design. In particular we would like to produce a compiler that generates asynchronous hardware. This would require functional-units to communicate using some kind of request/acknowledge protocol. Figure 19 shows how validity analysis could be applied to a 4-phase request/acknowledge protocol.

In other work we have demonstrated that SAFL can represent a wide range of hardware designs using only its simple functional primitives. This makes source-to-source transformation a very powerful technique for investigating multiple designs from a single specification. Although SAFL is very good at expressing many types of design, it seems that there are some types of hardware for which SAFL is not an ideal specification language: in particular we are currently unsure how best to represent input/output. More work is needed to investigate how to make the SAFL language applicable to a wider range of designs. We are currently investigating a number of ideas including the controlled incorporation of imperative features (cf. ML [7]) and an approach based on CSP-like channel communication.

References

[1] Berry, G. Real-time programming: General purpose or special-purpose languages. In G. Ritter (ed.), Information Processing 1989, pages 11-17. Elsevier Science Publishers B.V. (North Holland), 1989.

[2] Bjesse, P., Claessen, K., Sheeran, M. and Singh, S. Lava: Hardware Description in Haskell. Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, 1998.

[3] Greaves, D.J. An Approach Based on Decidability to Compiling C and Similar, Block-Structured, Imperative Languages for Hardware Software Codesign, Unpublished memo, 1999.

[4] Halbwachs, N., Caspi, P., Raymond, P. and Pilaud D. The Synchronous Dataflow Programming Language LUSTRE. Proc. IEEE, vol. 79(9). September 1991.

[5] Hennessy, J.L. and Patterson, D.A. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 1990.

[6] The LavaLogic "Forge" software-to-hardware compiler. See `www.lavalogic.com`

[7] Milner, R., Tofte, M., Harper, R. and MacQueen, D. The Definition of Standard ML (Revised). MIT Press, 1997.

[8] Morison, J.D. and Clarke, A.S. ELLA 2000: A Language for Electronic System Design. Cambridge University Press 1994.

[9] Mycroft, A. and Sharp, R.W. A Statically Allocated Parallel Functional Language. To appear: Proc. ICALP 2000, Springer-Verlag LNCS, July 2000.

[10] Mycroft, A. and Sharp, R.W. The FLaSH Project: Resource-aware Synthesis of Declarative Specifications. Proceedings of the International Workshop on Logic Synthesis 2000. Also available as `http://www.cl.cam.ac.uk/users/am/papers/iwls00.ps.gz`

[11] Mycroft, A. and Sharp, R.W. Hardware/Software Co-Design using a Functional Language. Submitted for publication.

[12] Page, I. and Luk, W. Compiling Occam into Field-Programmable Gate Arrays. In Moore and Luk (eds.) FPGAs, pages 271-283. Abingdon EE&CS Books, 1991.

[13] Sheeran, M. muFP, a Language for VLSI Design. Proc. ACM Symp. on LISP and Functional Programming, 1984.