Hardware Synthesis using SAFL and Application to Processor Design*

Alan Mycroft and Richard Sharp

Computer Laboratory, Cambridge University

`http://www.cl.cam.ac.uk/users/am/`

7 September 2001

*Work done in collaboration with AT&T Laboratories Cambridge.

This talk

- What's wrong with VHDL/Verilog?
- SAFL: features, pros and cons
- Processor Design and Transformation
- Partial Evaluation
- Conclusions

VHDL and Verilog

- Large languages; multiple subsets/interpretations
  - Structural (netlist) subset NB
  - RTL subset
  - Behavioural subset NB
  - Simulation interpretation
- Complicated semantics (still PhD theses on their semantics)
- Industry standard, mature tools

What's wrong with VHDL/Verilog

UNIVERSITY OF CAMBRIDGE

- Complicated languages; hard to analyse and manipulate.
- No real abstraction mechanism ("MODULE" is not enough); a proper interface would specify (e.g.) timing constraints.
- No "late binding"; various decisions frozen and propagated through a design from day one—would prefer to:
  - specify the *logical* structure, but choose its realisation as *physical* structure at late stages of design;
  - select between synchronous and asynchronous design or rework the exact number of functional units, how they are shared and even the interconnection protocol (e.g. changing parallel-on-ready-signal to serial-starting-on-next-clock-tick) at any stage in the design process.

What's right with VHDL/Verilog

- RTL VHDL/Verilog is a good compiler target language
- analogy: like C or even .NET for programming languages
- analogy: lots of good compilers
- analogy:

netlist	assembler
RTL	C
SAFL	ML/Haskell

- SAFL is just one attempt ("keep it simple" methodology).
- Also: Lustre, Esterel, Hydra, Lava, Hawk, System/C, Hardware/C ...

This talk

- What's wrong with VHDL/Verilog?
- SAFL: features, pros and cons
- Processor Design and Transformation
- Partial Evaluation
- Conclusions

SAFL

UNIVERSITY OF CAMBRIDGE

- Language of the FLaSH project (Functional Languages for Synthesising Hardware) we have developed at AT&T Laboratories Cambridge.
- First-order call-by-value functional language
- Syntactically restricted to be statically allocatable (recursive loops must only involve tail-calls)—well-suited to hardware synthesis
- Parallel evaluation of tupled expressions (arguments)
- Shared resources protected by semaphores (arbiters); recursion restrictions imply no deadlock.

Why good for hardware?

UNIVERSITY OF CAMBRIDGE

- Static allocation is exactly what hardware does (any form of dynamic addressing is hard—"von Neumann bottleneck").
- Parallelism (from call-by-value) is what hardware provides lots of.
- Resource aware: the *compiler* treats functions as hardware units; two calls to one function represent a shared unit (protected by semaphore); we duplicate a function for two unshared units.
- Separate *transformer* to map within area-time spectrum.
- We have built a hardware compiler which translates SAFL to circuit designs (hierarchical RTL Verilog).

  Promising results: 3-4 page spec. of a commercial processor.

SAFL isn't Hydra/Lava/Hawk

- Lava is structural—program evaluation yields hardware.
- SAFL is behavioural—programs (under-) specify hardware computations, a compiler completes the specification;

More concretely

- Lava provides a higher-order 'macro' language; the user can specify silicon precisely at the netlist level and still make (some) late-bound decisions
- SAFL aims at "having the system fill in netlist-level details as appropriate"; therefore these can be optimised (and automatically track user-level transformations on SAFL)
- SAFL is eager, Lava is lazy.

Language

- c constants,
- x variables (`let` and formal parameters)
- a primitive functions
- f user-defined functions

Terms e and programs p given by:

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \mid \\
 & & \texttt{let } \vec{x} = (e_1, \ldots, e_k) \texttt{ in } e_0 \mid \\
 & & a(e_1, \ldots, e_{arity(a)}) \mid f(e_1, \ldots, e_{arity(f)}) \\
p & ::= & \texttt{fun } f^1(\vec{x}) = e_1; \ldots; \texttt{fun } f^n(\vec{x}) = e_n;
\end{array}
$$

Calls in recursive cycles must all be tail-calls.

Data flow, static allocation and concurrency

- SAFL evaluation is concurrent, limited by data flow (and control flow at user-function call and conditional).

- static allocation and concurrency conflict

```
let f(a,b) = ...
let g(x) = ...
let main() = ... f(g(x),g(y)) ...
```

  use semaphores (arbiters) to control entry to critical region of `f()`; here this could be done statically, but for

```
let main() = ... f(g(h(x)),g(k(y))) ...
```

  dynamic scheduling can win. Optimise: *soft scheduling*.

Hardware Synthesis using SAFL (1)

In principle, the SAFL compiler compiles each function definition

$$\texttt{fun}\ f(\vec{x}) = e$$

to a functional unit H_f which contains:

- an input register file, r, clocked by *clk*, of width given by $\vec{x}$;
- a *request* signal which causes data to be latched into r and computation of e to start;
- an output port P, here just specification of output wires to hold the value of e;
- a *ready* signal which is asserted when P is valid.

Exception: zero-cycle functions (see later)

Hardware Synthesis using SAFL (2)

The SAFL primitives are compiled each having an output port and *request* and *ready* control signals; they may use the output ports of their subcomponents:

constants c: the constant value forms the output port and the *request* signal is merely copied to the *ready* signal.

variables x: since variables have been clocked into a register then they can be treated as constants, save that the output port is connected to the register.

if e_1 **then** e_2 **else** e_3: first e_1 is computed; when it is ready its boolean output routes a *request* signal either to e_2 or to e_3 and also routes the *ready* signal and output port of e_2 or e_3 to the result.

Hardware Synthesis using SAFL (3)

let $\vec{x} = \vec{e}$ **in** e_0: the $\vec{e}$ are computed concurrently (all their *request*s are activated); when all are complete e_0 is activated. Note that the results of the $\vec{e}$ are not latched.

built-in function calls $a(\vec{e})$: these are expanded in-line; the $\vec{e}$ are computed concurrently and their outputs placed as input to the logic for a; when all are complete this logic is activated;

user function calls $g(\vec{e})$: in the case that the called function only has one call (being non-recursive) the effect is similar to a built-in call—the $\vec{e}$ are computed concurrently and their outputs connected as to the input register file for g; when all are complete the *request* for g is activated; its output port and *ready* signal form those for $g(\vec{e})$.

Hardware Synthesis using SAFL (4)

- Tail recursive self-calls map into loops.
- Shared functional units (two or more callers) have multiplexers to route values from multiple call sites (and arbiters too, unless analysis can show there are no concurrent calls).
- Returned values are just distributed to all callers (if the implementation uses a completion signal then this is returned to the actual caller).

Our current implementation:

- "Activate and wait for result" or "Clocked asynchronous" model: global clock with completion signal after possibly computation-dependent delay. (Compositional.)

Hardware Synthesis using SAFL (5)—permanisors

UNIVERSITY OF CAMBRIDGE

Subtlety: values of functions invoked at multiple call sites. Consider:

```
f(x,y) = g(h(x+1),k(x+1),k(y))
```

where there are no other calls to `h` and `k` in the program. Then,

- Assuming `h` clocks `x+1` into its input register; then $P_{\mathtt{h}}$ will continue to hold its value needed by `g`.
- But, consider calls to `k`, suppose `k(y)` done first. Its result on $P_{\mathtt{k}}$ will be lost when `k(x+1)` is called.

Therefore:

- insert a clocked *permanisor* register inside $H_{\mathtt{f}}$ to hold value of `k(y)`. (cf. expression temporary).

This trick gives much better compilation and explains our ability to avoid latching `let` results (good for exploiting concurrency).

Optimisating Compilation

Compiler optimisations based on whole-program analysis:

Removing arbiters : not all shared functions can be activated concurrently, e.g. `f(f(x))`.

Register placement : not all values returned from shared functions need to be latched in permanisors.

Cycle counting : (for synchronous targets) aids the above two optimisations by exploiting cycle count information of called functions.

Zero cycle functions : small functions can produce better code sometimes without latching arguments (cf. callee-save/caller-save).

Transformations in SAFL

UNIVERSITY OF CAMBRIDGE

Transformations move in the area-time spectrum:

- Fold-Unfold (Burstall & Darlington).
- Instruction ROMs for specialised processors.
- Specialised processors are themselves SAFL programs.

Important point: source-to-source transformation allied with resource-aware compilation means understandable transformations.

Transformations in SAFL, example (1)

Fold/unfold transformation can trade area against time. E.g.

```
fun f x = ...
fun main(x,y) = g(f(x),f(y))
```

Unfold one call to `f` and fold as a new definition `f'`:

```
fun f  x = ...
fun f' x = ...
fun main(x,y) = g(f(x),f'(y))
```

The second program has more area (extra `f'`) but (here) runs more quickly as `f(x)` and `f'(y)` execute in parallel.

Transformations in SAFL, example (2)

Trading work done per clock cycle against clock speed. Consider

```
fun mult(x, y, acc) =
  if (x=0 or y=0) then acc
     else mult(x<<1, y>>1, if y[0:0] then acc+x else acc)
```

[These 3 lines of SAFL produce over 150 lines of RTL Verilog (1146 2-input equivalent gates).] Can fold/unfold to:

```
fun mult(x, y, acc) =
  if (x=0 or y=0) then acc
  else let (x',y',acc') = (x<<1, y>>1,
                           if y[0:0] then acc+x else acc) in
    if (x'=0 or y'=0) then acc'
    else mult(x'<<1, y'>>1, if y'[0:0] then acc'+x' else acc')
```

gives: almost twice the area but half as many clock cycles.

This talk

- What's wrong with VHDL/Verilog?
- SAFL: features, pros and cons
- Processor Design and Transformation
- Partial Evaluation
- Conclusions

A simple (Hennessy&Patterson) CPU

```
enum { OPhalt, OPj, OPbz, OPst, OPld, OPadd };
fun cpu(pc, regs) =
  (let I = imem(pc)
   let (op,m,rd,ra) = (I[31:27], I[26], I[21:25], I[16:20])
   let (rb,imm) = (I[0:4], sext32(I[0:15]))
   let A = regs[ra]
   let B = m==0 ? imm : regs[rb]
   let C = alu(op, A, B)
   let D = case op of OPld => dmem(C,0,0)
                    | OPst => dmem(C,regs[rd],1)
                    | _    => C
   let regs' = case op of OPld => regs[D @ rd]
                        | OPadd => regs[D @ rd]
                        | _    => regs
   let pc' = case op of OPj => B
                      | OPbz => pc+4 + (A==0 ? imm : 0)
                      | _ => pc+4
   in (op==OPhalt ? regs'[0] : cpu(pc', regs')));
```

Features

- CPU specified very simply
- `regs[v @ i]` is functional array update (OK with respect to static allocation).

Issues:

- instruction stages not pipelined
- `imem()` and `dmem()` accesses serialised
- no multiple issue

The CPU re-folded

```
fun cpu1a(pc, op1, regs1, C1, rd1) =
  (let D = case op1 of OPld => dmem(C1,0,0)
                     | OPst => dmem(C1,regs1[rd1],1)
                     | _    => C1
   let regs = case op1 of OPld => regs1[D @ rd1]
                        | OPadd => regs1[D @ rd1]
                        | _     => regs1
   in (op1==OPhalt ? regs[0] :          (* note this line *)
      let I = imem(pc)                  (* note this line *)
      let (op,m,rd,ra) = (I[31:27], I[26], I[21:25], I[16:20])
      let (rb,imm) = (I[0:4], sext32(I[0:15]))
      let A = regs[ra]
      let B = m==0 ? imm : regs[rb]
      let C = alu(op, A, B)
      let pc' = case op of OPj => B
                         | OPbz => pc+4 + (A==0 ? imm : 0)
                         | _ => pc+4
      in cpu1a(pc', op, regs, C, rd)));
```

Same semantics, but still serialises `imem()` and `dmem()`, so:

And a little tweak

```
fun cpu2(pc, op1, regs1, C1, rd1) =
  (let D = case op1 of OPld => dmem(C1,0,0)
                     | OPst => dmem(C1,regs1[rd1],1)
                     | _    => C1
   let regs = case op1 of OPld => regs1[D @ rd1]
                        | OPadd => regs1[D @ rd1]
                        | _     => regs1
   let I = imem(pc)                        (* note this line *)
   in (op1==OPhalt ? regs[0] :             (* note this line *)
      let (op,m,rd,ra) = (I[31:27], I[26], I[21:25], I[16:20])
      let (rb,imm) = (I[0:4], sext32(I[0:15]))
      let A = regs[ra]
      let B = m==0 ? imm : regs[rb]
      let C = alu(op, A, B)
      let pc' = case op of OPj => B
                         | OPbz => pc+4 + (A==0 ? imm : 0)
                         | _ => pc+4
      in cpu1a(pc', op, regs, C, rd)));
```

Semantic change (pre-fetch instruction following `OPhalt`) but now initialisation of `I` and `D` are concurrent.

A CPU with memory controller

```
fun memctrl(pc, a, d, r, w) =
  (let iv = is_dmem(pc) ? dmem(pc,0,0)
                         : imem(pc)
   let dv = (r or w) ? (is_dmem(a) ? dmem(a,d,w)
                                    : imem(a))
                      : a
   in (iv,dv))
fun cpu3(pc, op0, regs0, C0, rd0) =
  (let (I,D) = memctrl(pc, C0, regs[rd0],
                       op==OPld, op==OPst)
   in ...)
```

Allows instruction and data-fetch from both `imem` (ROM) and `dmem` (RAM) and reads both concurrently if possible (SAFL soft scheduling will insert arbiters to delay a cycle if conflict).

Pipelining ALU and memory access

```
fun cpu4(pc, op1, A1, B1, rd1, op2, regs2, C2, rd2) =
  (let C = alu(op1, A1, B1)
   let D = case op2 of OPld => dmem(C2,0,0)
                     | OPst => dmem(C2,regs2[rd2],1)
                     | _    => C2
   let regs = case op2 of OPld => regs2[D @ rd2]
                        | OPadd => regs2[D @ rd2]
                        | _     => regs2
   let I = imem(pc)
   in (op2==OPhalt ? regs[0] :
      let (op,m,rd,ra) = (I[31:27], I[26], I[21:25], I[16:20])
      let (rb,imm) = (I[0:4], sext32(I[0:15]))
      (* forwarding (a.k.a. by-passing) would go here *)
      let A = regs[ra]
      let B = m==0 ? imm : regs[rb]
      let pc' = case op of OPj => B
                         | OPbz => pc+4 + (A==0 ? imm : 0)
                         | _ => pc+4
      in cpu4(pc', op, A, B, rd,  op1, regs, C, rd1)));
```

Faster clock, instruction overlap.

Simple Superscalar Processor

```
fun cpu5(pc, regs) =
  (let (I1,I2) = (imem(pc), imem(pc+4))
   let (op1,m1,rd1,ra1) = (I1[31:27], I1[26], I1[21:25], I1[16:20])
   let (rb1,imm1) = (I1[0:4], sext32(I1[0:15]))
   let (op2,m2,rd2,ra2) = (I2[31:27], I2[26], I2[21:25], I2[16:20])
   let (rb2,imm2) = (I2[0:4], sext32(I2[0:15]))
   if ((op1 == OPld or op1 == OPadd) and
       (rd1 == ra2 or (m1==1 and rd1 == rb2)
                   or (op2 == OPst and rd1 == rd2))) then
     <I2 reads from a register written by I1 -- serialise>
   else
     <do I1 and I2 concurrently>
     let (A1,A2) = (regs[ra1], regs[ra2])
     let (B1,B2) = ((m1==0 ? imm1 : regs[rb1]),
                    (m2==0 ? imm2 : regs[rb2]))
     let (C1,C2) = (alu(op1, A1, B1), alu(op2, A2, B2))
     let D1 = case op1 of ...
     let D2 = case op2 of ...
     ...
     in (op1==OPhalt ? regs'[0]
         op2==OPhalt ? regs''[0] : cpu5(pc', regs'')));
```

This talk

- What's wrong with VHDL/Verilog?
- SAFL: features, pros and cons
- Processor Design and Transformation
- Partial Evaluation
- Conclusions

Compile-time and Run-time operations

Recall:

- Lava: recursion specifies nested or iterated structure
- SAFL: recursion specifies (behavioural) iteration

Just like *compile-time* versus *run-time* data in Partial Evaluation.

Formalise via types.

Types for run-time/compile time

SAFL types:

$$\begin{array}{ll} \text{(values)} & bit_n \\ \text{(functions)} & bit_m \mapsto bit_n. \end{array}$$

Lava types:

$$\tau ::= int \mid circuit \mid \tau \to \tau \mid \tau \times \tau \mid list(\tau)$$

Can combine using $circuit = bit_m \mapsto bit_n$.

Nielson&Neilson's *Two-level metalanguage*: '$\to$' represents compile-time, or structural, composition (cf. Verilog `module` instantiation) and '$\mapsto$' representing run-time, or behavioural, computation (i.e. data movement in silicon).

Payback: circuit specialisation

Can define a merged-feature language, 2SAFL with abstract syntax

$$
\begin{array}{rcll}
e & ::= & x \mid c \mid & e\,e' \\
 & & \mid \lambda^c\, \vec{x}.e & \mid \lambda^r\, \vec{x}.e \\
 & & \mid \mathit{if}^c\ e\ \mathit{then}\ e\ \mathit{else}\ e & \mid \mathit{if}^r\ e\ \mathit{then}\ e\ \mathit{else}\ e \\
 & & \mid \mathit{let}^c\ \vec{x} = e\ \mathit{in}\ e & \mid \mathit{let}^r\ \vec{x} = e\ \mathit{in}\ e
\end{array}
$$

with compile-time and run-time variants of operations (NB various constraints on nesting), and then transform between them.

Example

Using $\texttt{fun}^r$ etc. instead of λ^r etc.

```
fun^c multiplier(n) =
  local fun^r f(x:bit(n), y:bit(n), acc:bit(n)) :  bit(n) =
    if y=0 then acc
    else f(x<<1, y>>1, if y[0:0] then acc+x else acc)
  in f end
fun^r m1 = multiplier(16); fun^r m2 = m1;
fun^r m3 = multiplier(16);
fun^r main(...)  = ...  m1 ...  m2 ...  m3 ...
```

Resource awareness: `m1` and `m2` are synonyms; `m3` is distinct 16-bit multiplier. The type of `multiplier` is

$$(n \in int) \to (bit_n * bit_n * bit_n \mapsto bit_n).$$

Specialisation (from run-time to compile-time)

SAFL

```
fun mult = λʳ(x, y, acc).
  if y=0 then acc
  else mult(x<<1, y>>1, if y[0:0] then acc+x else acc)
  fun mult13(x,acc) = mult(x,13,acc).
```

Now curry it:

```
fun mult' = λᶜy.  λʳ(x, acc).
  if y=0 then acc
  else mult'(y>>1)(x<<1, if y[0:0] then acc+x else acc).
```

Not valid 2SAFL (`mult'` applied insider a λ^r) but expand `mult'(13)` to get ...

Specialisation(2)

```
fun mult13 = λr(x, acc).  if 13=0 then acc
  else mult6(x<<1, if 13[0:0] then acc+x else acc)
fun mult6 = λr(x, acc).  if 6=0 then acc
  else mult3(x<<1, if 6[0:0] then acc+x else acc)
fun mult3 = λr(x, acc).  if 3=0 then acc
  else mult1(x<<1, if 3[0:0] then acc+x else acc)
fun mult1 = λr(x, acc).  if 1=0 then acc
  else mult0(x<<1, if 1[0:0] then acc+x else acc)
fun mult0 = λr(x, acc).  if 0=0 then acc
  else mult0(x<<1, if 0[0:0] then acc+x else acc).
```

which simplifies to:

Specialisation(3)

```
fun mult13 = λʳ(x, acc).  mult6(x<<1, acc+x)
fun mult6 = λʳ(x, acc).  mult3(x<<1, acc)
fun mult3 = λʳ(x, acc).  mult1(x<<1, acc+x)
fun mult1 = λʳ(x, acc).  mult0(x<<1, acc+x)
fun mult0 = λʳ(x, acc).  acc
```

and hence (by unfolding, or just by compiling the used-once functions `mul6` to `mul0` into zero-clock functions) to:

```
fun mult13 = λʳ(x, acc).
  acc + x + ((x<<1)<<1) + (((x<<1)<<1)<<1)
```

This talk

- What's wrong with VHDL/Verilog?
- SAFL: features, pros and cons
- Processor Design and Transformation
- Partial Evaluation
- Conclusions

Conclusions

SAFL is a prototype language which attempts

- to separate function correctness from detailed implementation; Burstall/Darlington write simple code and then optimise in a semantics-preserving manner;
- to allow transparent manipulations via a *transformer* with the *compiler* filling in fine details. Resource-awareness not "this is what the black-box tool does";
- late binding, even for software/hardware partitioning.

Main problems:

- not industry standard
- "invoke and wait for return" model, but see Sharp&Mycroft paper (yesterday).

Where to find more

- ICALP'00: Theory of static allocatability/resource duplication
- AT&T report tr.2000.3: Compiler details
- TACAS'01: Hardware software co-design
- SAS'01: Soft scheduling
- CHARME'01: Adding statically allocatable process-like primitives.

Work to do

- Build a real GUI-based *transformer*
- Give alternative back-ends, particularly asynchronous and GALS.