Hardware Construction Languages
Do we need them?

Dr David J Greaves
University of Cambridge
Computer Laboratory

BCS OSSG/OSHUG
Joint Open Source Specialist Group
Open Source Hardware User Group meeting
London June 2015
Hardware Construction Language

A (rich) language for writing programs that print out a circuit diagram.

But does it:

- Ensure syntactically well-formed output?
- Semantically well-formed as well? E.g. no two outputs wired together?
- Include simple optimisations?
  - Discarding disconnected logic
  - Constant propagation and identity folding:
    
    Eg: \( \text{exp} \&\& \text{true} \rightarrow \text{exp} \)
Staged Evaluation

- Part of the program runs at 'compile time' – the *elaborate phase*.
- The elaborated program consists of a hardware circuit that runs at run time: the *execution phase*.

*For example, Verilog and VHDL have generate statements and generate variables which disappear during the first stage.*
Lava HCL

'Lava Hardware Design in Haskell'

Make use of all standard combinators such as Fold, Map and Zip.

Different instantiations of the leaf nodes for
- Simulation
- Synthesis
- Verification

Bjesse, Koen, Sheeran, Singh 1998
Data-Dependent Control Flow?

The 'if' statement is part of any programming language, but how much conditional execution does our language support at run time?

- Lava's elaborate phase is very rich, it certainly contains 'if' statements.
- But all run-time conditional flow was through explicitly printed multiplexors.

Generally we desire greater expressivity than that...
Time/Space Folding

• We would like to use one entry of the design for either:
  – Fast execution using a lot of hardware
  – Slower execution using less hardware

• We should favour languages that are amenable to rapidly changing between these styles,

• while still being 'resource aware' – engineers understand roughly how many gates they are using as they write each line.

Functional programs are generally much easier to manipulate in this way!
SAFL - Statically Allocated Functional Language

Used a variant of ML to describe hardware
- We see powerful combinators for hardware generation
  - The ML 'if' is the run-time 'if' (could not be a DSL)
  - All recursion is tail recursion, hence bounded stack space – finite state.
  - But functional style did not fit comfortably with RAMs

SAFL – Resource Awareness

Baseline rules that control the amount of hardware generated:

1. Leaf operators occurring syntactically are freshly instantiated in the hardware for each syntactic occurrence in the source code.

2. The same goes for function definitions, which means function applications of a named function are serialised with argument and return value multiplexors.

This contrasts with black-box High-Level Synthesis (HLS) where the designer perhaps only broadly constrains how many ALUs and RAMs to use, but the amount of random logic is unpredictable.
Time/Space Folding in SAFL

fun cmult x y =
    let ans_re = x.re*y.re - x.im*y.im
    let ans_im = x.im*y.re + x.re*y.im
    in (ans_re, ans_im) // 4 multipliers, 2 adders.

A function replicator, such as UF, enables control of time/space folding, giving a fresh copy of a function.

let use_time  = g(cmult a b, cmult c d)
    // 4 multipliers, 2 adders + resources for g

let use_space = g(cmult a b, (UF cmult) c d)
    // 8 multipliers, 4 adders + resources for g

Server farms etc are also easy to provide provided everything remains stateless.
Chisel HCL (from UCB)

- Chisel is embedded as a DSL in Scala.
- Scala is a wonderful language
  - A superb mix of functional, imperative and OO
  - Scala has flexible overloading syntax that makes extensions and implicit conversions simple to deploy.
- Chisel provides all the main basic gates and memories, but not much data-dependent control flow (no runtime program counters).
- Scala allows us to build up on top easily.
A varadic Priority Arbiter in Chisel

class genPriEncoder(n_inputs : Int) extends Module
{
  val io  = new Bundle {  }
  val terms = (1 to n_inputs).map
   (n => ("req" + n, "grant" + n))

terms.foldLeft (Bool(false))
{ case (sofar, (in, out)) =>
  val (req, grant) = (Bool(INPUT), Bool(OUTPUT))
  io.elements += ((in, req))
  io.elements += ((out, grant))
  grant := req & !sofar
  val next = new Bool
  next := sofar | req
  next
}
}

H/W components extend Module. They do their I/O via a Bundle. All the standard operators & | ! are overloaded for h/w generation.
Run-time 'if' in Chisel

class Parity extends Module {
  val io = new Bundle {
    val in  = Bool(dir = INPUT)
    val out = Bool(dir = OUTPUT) }
  val s_even :: s_odd :: Nil = Enum(UInt(), 2)
  val state  = Reg(init = s_even)
  when (io.in) {
    when (state === s_even) { state := s_odd  }
    when (state === s_odd)  { state := s_even }
  }
  io.out := (state === s_odd)
}

*The 'when' key word is Chisel's main run-time IF operator, but there are other variants including a switch/case statement. The === operator is used so that Scala's == remains usable.*
Adding TLM to Chisel

We store function entry points in the I/O bundle.

Each function is annotated with its fixed pipeline delay or else can use handshake nets Request/Valid (not shown here).

TLM = Transaction Level Modelling – although here we are not modelling, but doing.

```scala
class Targer_AX extends Module {
  val io = new TLM_bundle_lc {
    // Register TLM callable function with one pipeline delay.
    tlmBind_a1(ax_fun _, 1)
  }
  def ax_fun(x:UInt) = Reg(UInt(32), x + UInt(10))
}

class Targer_DX extends Module {
  val io = new TLM_bundle_lc {
    // TLM callable diadic function with 2 pipeline delays.
    tlmBind_a2(dx_fun _, 2)
  }
  def dx_fun(x:UInt, y:UInt) = Reg(Reg(UInt(32), x + y))
}
```
TLM in Chisel (2)

val unit_a = Module(new Targer_AX())
val unit_b = Module(new Targer_BX())
val unit_d = Module(new Targer_DX())

// Diadic - single use test
//  val answer = unit_d.io.run2(unit_a.io.run1(arg1K), unit_b.io.run1(arg2K))

// Diadic - reuse of same component AX
val answer = unit_d.io.run2(unit_a.io.run1(arg1K), unit_a.io.run1(arg2K))

// Invoke and downconvert to unguarded for rest of design
val answer1 = SAFLImplicitx.ex_drop_chisel_data_from_guarded(answer)
io.z := answer1
io.v := answer.isValid()
Running the TLM Example with SAFL semantics
HardCaml

ML is the perhaps the best-known functional language.

ML + Objects + Better syntax + more advanced types = OCAML

OCAML is the ultimate programming language?

(Well some think so - Mirage operating system is an OCAML linux kernel. I'm beginning to prefer Scala ...)

HardCaml: An open-source domain specific language embedded in OCaml for designing and testing register transfer level hardware designs. --- The HardCaml library provides an API roughly consistent with the structural subset of VHDL and Verilog.

Also: has a snazzy front end embedded in Javascript.
HardCaml Small Example

/* Verilog counter */
module counter
    #(parameter bits = 8)
    (
        input clock, clear, enable,
        output reg [bits-1:0] q
    );

    always @(posedge clock)
        if (clear) q <= 0;
        else if (enable) q <= q + 1;
endmodule

(* HardCaml counter *)
let q = reg_fb r_sync enable bits (fun d -> d :+: 1)
Rule-based hardware generation (Bluespec)

- Recently Bluespec System Verilog has successfully raised the level of abstraction in RTL design:
  - A Bluespec design is expressed as a list of declarative rules that fire atomically and which last less than one clock cycle,
  - Shared variables are mostly replaced with one-place FIFO buffers with automatic handshaking,
  - Rules are allocated a static schedule at compile time and some that can never fire are reported,
  - The wiring pattern of the whole design is elaborated using a powerful embedded functional language (as per Lava).
Bluespec: Tiny Example

module mkTb (Empty);

    Reg#(int) x <- mkReg (23);

    rule countup (x < 30);
        int y = x + 1;
        x <= x + 1;
        $display ("x = %0d, y = %0d", x, y);
    endrule

    rule done (x >= 30);
        $finish (0);
    endrule

endmodule: mkTb

But, imperative expression using a conceptual thread is also useful to have, so Bluespec has a behavioural sub-language compiler built in.
Bluespec: Pipe Example

module mkTb (Empty);
    Reg#(int) x <- mkReg ('h10);
    Pipe_ifc pipe <- mkPipe;
rule fill;
    pipe.send (x);
    x <= x + 'h10;
endrule

rule drain;
    let y = pipe.receive();
    $display(" y = %0h", y);
    if (y > 'h80) $finish(0);
endrule
Occam/CSP Hardware Design

Handel-C uses explicit Occam/CSP-like channels ('!' to write, '?' to read):

```// Generator (src)     // Processor                             // Consumer (sink)
while (1)                    while(1)                           while(1)
{                                      {                                      {
  ch1 ! (x);                        ch2 ! (ch1? + 2)               $display(ch2?);
  x += 3;                       }
}                                     }
```

Using channels makes concurrency explicit and allows synthesis to re-time the design.

**Banning shared variables avoids RaW and WaW hazards.**

Handshaking wires within a synthesis unit may disappear during compilation if they would have constant values owing to certain components being always ready.
Do we need HCL's?

- The Chisel and HARDCAML baselines are fairly simple and provide all the 'structural' resources for emitting netlists and cycle-accurate simulation.
- Yet they leverage the full power of their parent language for elaboration.
- They provide interworking with RTL designs in Verilog and VHDL.
- They provide a 'power platform' for supporting your own favourite expression style ...
C-to-Gates: Classical HLS

Take one thread and a body of code:
  generate a custom datapath containing registers, RAMs and ALUs
  and a custom sequencer that implements an efficient, static schedule
  that achieves the same behaviour.

Creates a precise schedule of addresses on register file and RAM ports
and ALU function codes.

Typically unwinds inner loops by some factor.

All current EDA/FPGA vendors now support C++ to gates.

Leading free tool is LegUp from U Toronto.

Profiling or datapath description hints are needed for a sensible datapath
structure since sequencer states are not equiprobable and we do not want
to deploy resource on seldom-used data paths.
C-to-Gates: Classical HLS

For example, best mapping of the record fields x and y to RAMs is different in the two foreach loops:

```c
class IntPair
{
    public bool c;     public int x, y;
}

IntPair [] ipairs = new IntPair [1024];

void customer(bool qcond)
{
    int sum1 = 0, sum2 = 0;
    if (qcond) then foreach (IntPair pp in ipairs)
    {
        sum1 += pp.x + pp.y;
    }
    else foreach (IntPair pp in ipairs)
    {
        sum2 += pp.c ? pp.y: pp.x;
    }
    ...
}
```

The fields x and y could be kept in separate RAMs or a common one. If qcond rarely holds then a common RAM will serve since there is little contention. Whereas if qcond holds most of the time then keeping x and y in separate RAMs will boost performance.
Conclusions

- Functional elaboration language gives expressivity and support folding.
- **RAMs are very important in reality**
  - DRAM is not random access!
  - RAMs and registers suffer WaW RaW hazards!
- People vary in the expression form they prefer.
- Bluespec is good if we use FIFO interfaces exclusively! (Hmm message passing again!)
- Future styles will perhaps be more explicit on state edges: Read, Write, Assoc-Update, ....
Thankyou for you attention.

Open Source Links

- Chisel: https://chisel.eecs.berkeley.edu
- Toy Bluespec: www.cl.cam.ac.uk/~djg11/wwwhpr/toy-bluespec-compiler.html
- TLM for Chisel: http://technotes-djg.blogspot.co.uk/
- Kiwi HLS from C#: www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/kiwic.html
What is emitted by elaborate?

- Gates, wires, flip-flops and RAMs
- Atomic rules (Bluespec)