Bluespec-1: A language for hardware design, simulation and synthesis

Arvind
Laboratory for Computer Science
M.I.T.

January 13, 2003

http://www.csg.lcs.mit.edu/IAPBlue

Current ASIC Design Flow

C/C++ model by hand

Verilog RTL bugs

synthesis netlist place & route

final netlist

Tapeout Physical design

January 13, 2003
http://www.csg.lcs.mit.edu/IAPBlue
Goals of high-level synthesis

- Reduce time to market
  - Same specification for simulation, verification and synthesis
  - Rapid feedback ⇒ architectural exploration
  - Enable hierarchical design methodology
    Without sacrificing performance
    area, speed, implementability, …

- Reduce manpower requirement

- Facilitate maintenance and evolution of IP’s

These goals are increasingly urgent, but have remained elusive

Whither High-level Synthesis?

…Despite concerted efforts for well over a decade the compilers seem to not produce the quality of design expected by the semiconductor industry …
Bluespec: So where is the magic?

- A new semantic model for which a path to generating efficient hardware exists
  - Term Rewriting Systems (TRS)
  - The key ingredient: atomicity of rule-firings
    - James Hoe [MIT ’00] CMU and Arvind [MIT]
- A programming language that embodies ideas from advanced programming languages
  - Object oriented
  - Rich type system
  - Higher-order functions
  - transformable
  - Borrows heavily from Haskell
    - designed by Lennart Augustsson [Sandburst]

Overall implementation: Lennart Augustsson, Mieszko Lis

Outline

- Preliminaries
- A new semantic model for hardware description: TRS
  - Example: The GCD
  - Example: A simple pipelined CPU
Term Rewriting Systems (TRS)

TRS have an old venerable history – an example

Terms

GCD(x, y)

Rewrite rules

GCD(x, y) ⇒ GCD(y, x) if x > y, y ≠ 0 (R₁)

GCD(x, y) ⇒ GCD(x, y-x) if x ≤ y, y ≠ 0 (R₂)

Initial term

GCD(initX, initY)

Execution

GCD(6, 15) ⇒ GCD(3, 6) ⇒ GCD(6, 9) ⇒ GCD(6, 3) ⇒ GCD(3, 3) ⇒ GCD(3, 0)

Hardware description?

TRS as a Description of Hardware

Terms represent the state: registers, FIFOs, memories, ...

Rewrite Rules (condition → action)

represent the behavior in terms of atomic actions on the state
Language support to organize state and rules into *modules*

Modules are like *objects* (private state, interface methods, *rules*). Rules can manipulate state in other modules only via their interfaces.

GCD in Bluespec

```plaintext
mkGCD :: Module GCD
mkGCD =
module
  x :: Reg (Int 32)
  x <- mkReg _
  y :: Reg (Int 32)
  y <- mkReg 0
rules
  when x > y, y /= 0
    ==> action x := y
    y := x
  when x <= y, y /= 0
    ==> action y := y - x
interface
  start ix iy = action x := ix
  y := iy when y == 0
  result = x when y == 0
```

GCD in Bluespec

```plaintext
mkGCD :: Module GCD
mkGCD =
module
  x :: Reg (Int 32)
  x <- mkReg _
  y :: Reg (Int 32)
  y <- mkReg 0
rules
  when x > y, y /= 0
    ==> action x := y
    y := x
  when x <= y, y /= 0
    ==> action y := y - x
interface
  start ix iy = action x := ix
  y := iy when y == 0
  result = x when y == 0
```
External Interface: GCD

```
interface GCD =
  start :: (Int 32) -> (Int 32) -> Action
result  :: Int 32
```

Many different implementations (including in Verilog) can provide the same interface

```
mkGCD :: Module GCD
mkGCD = ...
.
.
.
```

GCD Hardware Module

```
  implicit conditions
```

```
y == 0
```

```
y == 0
```

```
GCD module
```

```
start
```

```
result
```

```
32
```

```
32
```

```
enab
```

```
rdy
```

```
rdy
```

```
32
```
The Generated TRS: GCD

```
x :: Prelude.VReg <- RegUN 32;
y :: Prelude.VReg <- RegN 32 0;
_d1 :: Bit 32 = x.get;
_d2 :: Bit 32 = y.get;
_d5 :: Bit 1 = _d2 == 0;
_d8 :: Bit 1 = (_d1 <= _d2) && (! (_d2 == 0));
_d7 :: Bit 1 = (! (_d1 <= _d2)) && (! (_d2 == 0));
_d9 :: Bit 32 = _d2 - _d1;

rules
  when _d7   ==> { x.set _d2; y.set _d1; };
  when _d8   ==> { y.set _d9; };
```

The Generated Verilog: GCD

```
module mkGCD(CLK, RST_N,start__1, start__2, E_start_, ...)
  input CLK; ... 
  output start__rdy; ... 
  wire [31 : 0] x$get; ... 
  assign result_ = x$get;
  assign _d5 = y$get == 32'd0;
  ... 
  assign _d3 = x$get ^ 32'h80000000) <= (y$get ^ 32'h80000000); 
  assign C___2 = _d3 && !_d5; 
  ... 
  assign x$set = E_start_ || P___1; 
  assign x$set_1 = P___1 ? y$get : start__1; 
  assign P___2 = _d3 && !_d5; 
  ... 
  assign y$set_1 =
    {32{P___2}} & y$get - x$get | {32{dt1}} | x$get | 
    {32{dt2}} & start__2; 
  RegUN #32 i_x(.CLK(CLK), .RST_N(RST_N), .val(x$set_1), ...) 
  RegN #32 i_y(.CLK(CLK), .RST_N(RST_N), .init(32'd0), ...) 
endmodule
```
Basic Building Blocks: Registers

- Bluespec has no built-in primitive modules
  - there is, however, a systematic way of providing a Bluespec view of Verilog (or C) blocks

```
interface Reg a =
  _read :: a  -- reads the value of a register
  _write :: a -> Action  -- sets the value of a register
```

Special syntax:
- `x` means `x._read`
- `x := e` means `x._write e`

```
mkReg :: a -> Module (Reg a)
The mkReg procedure interfaces to a Verilog implementation of a register
```

---

FIFO

```
interface FIFO a =
  enq :: a -> Action  -- enqueue an item
  deq :: Action  -- remove the oldest entry
  first :: a  -- inspect the oldest item
```

- when appropriate notfull and notempty are implicit conditions on FIFO operations
- mkFIFO interfaces to a Verilog implementation of FIFO

---

$n = \#$ of bits needed to represent the values of type “a”
Array

Arrays are a useful abstraction for modeling register files

interface Array index a =
  upd :: index -> a -> Action -- store an item
  sub :: index -> a -- retrieve an item

mkArray :: Module (Array index a)

- There are many implementations of mkArray depending upon the degree of concurrent accesses

Outline

- Preliminaries
- A new semantic model for hardware description: TRS
  - Example: The GCD
  - Example: A simple pipelined CPU
CPU with 2-stage Pipeline

```
module
  pc :: Reg Iaddress <- mkReg 0
  rf :: Array RName (Bit 32) <- mkArray
  bu :: FIFO Instr <- mkFIFO
  rules ...
  interface ...
```

---

CPU Instructions

```
data RName = R0 | R1 | R2 | ... | R31

  type Src       = RName
  type Dest     = RName
  type Cond     = RName
  type Addr     = RName
  type Val      = RName
  
  data Instr = Add Dest Src Src
              | Bz     Cond Addr
              | Load   Dest Addr
              | Store  Val Addr
```

Processor - Fetch Rules

"Fetch":
when True
  ==> action pc := pc + 1
      bu.enq (imem.get pc)

Note that this rule pays no special attention to branch instructions

Processor - Execute Rules

"Add":
when (Add rd rs rt) < - bu.first
  ==> action
      rf.upd rd (rf.sub rs + rf.sub rt)
      bu.deq

"Bz Not Taken":
when (Bz rc ra) < - bu.first (rf.sub rc) /= 0
  ==> action
      bu.deq

"Bz Taken":
when (Bz rc ra) < - bu.first (rf.sub rc) == 0
  ==> action
      pc := rf.sub ra
      bu.clear

bu.notfull check is implicit!

bu.notempty check is implicit!