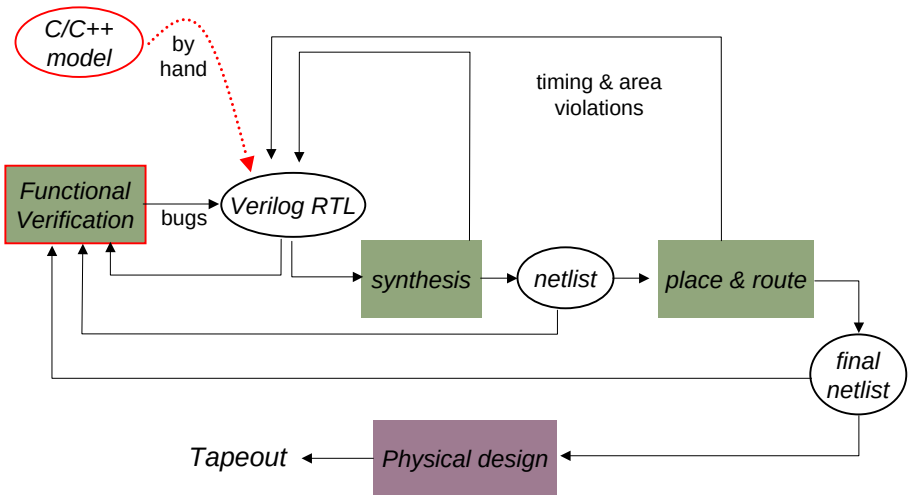
Bluespec-1: A language for hardware design, simulation and synthesis

Arvind
Laboratory for Computer Science
M.I.T.

November 13, 2002

http://www.csg.lcs.mit.edu/6.827

Current ASIC Design Flow



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 impementation: Lennart Augustsson, Mieszko Lis

Outline

- Preliminaries √
- A new semantic model for hardware description: TRS
- An example: A simple pipelined CPU
- Bluespec compilation

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 (R_1)

GCD(x, y) ⇒ GCD(x, y-x) if x≤ y, y≠0 (R_2)

Initial term

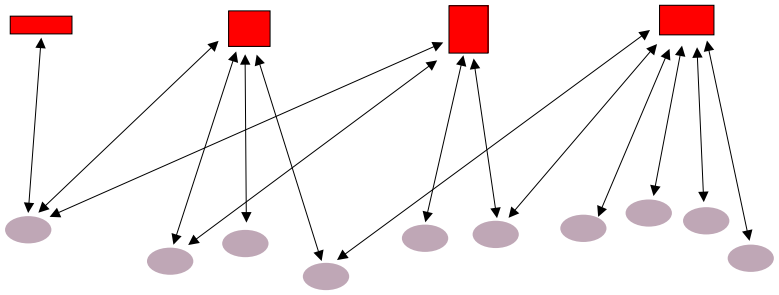
GCD(initX,initY)

Execution

GCD(6, 15) ⇒

TRS as a Description of Hardware

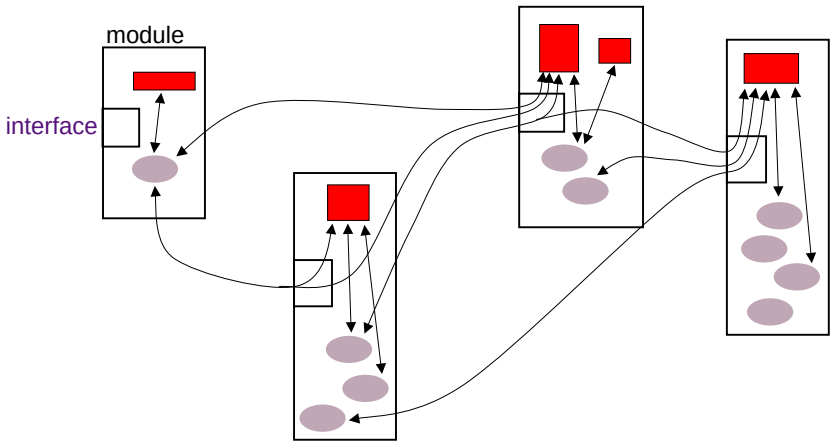
Terms represent the *state*: registers, FIFOs, memories, ...



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

```
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
```

State

Internal behavior

External interface

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 = ...
  .
  .
  .
mkGCD1 :: Module GCD
mkGCD1 = ...
```


Basic Building Blocks: Registers

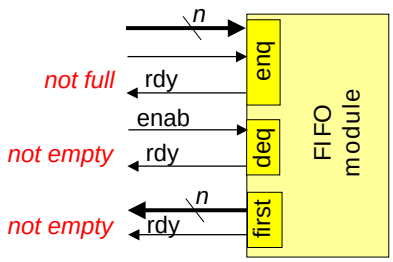
- Bluespec has no built-in primitive modules
  - there is, however, a systematic way of providing a Bluespec view of Verilog (or C) blocks

```
interface Reg a =
    get :: a              -- reads the value of a register
    set :: a -> Action    -- sets the value of a register

Special syntax:
        – x       means x.get
        – x := e means x.set 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

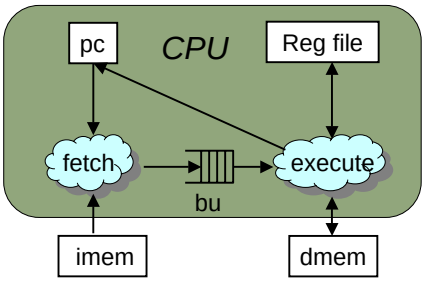
```
interface  Array index a =
    uda     :: index -> a -> Action  -- store an item
    (!)     :: index -> a            -- retrieve an item


    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 √
- An example: A simple pipelined CPU
- Bluespec compilation

CPU with 2-stage Pipeline

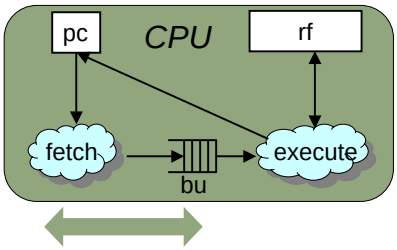



```
mkCPU :: Imem -> Dmem -> Module CPUinterface
mkCPU imem dmem =
  module
      pc :: Reg Iaddress <- mkReg 0
      rf  :: Array RName (Bit 32) <- mkArray
      bu :: FIFO Instr <- mkFIFO
      rules ...
      interface ...
```


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
           | Jz      Cond Addr
           | Load    Dest Addr
           | Store   Val  Addr
```

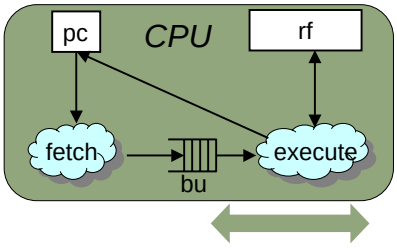

Processor - Fetch Rules




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

Note that this rule pays no special attention to branch instructions

Processor - Execute Rules

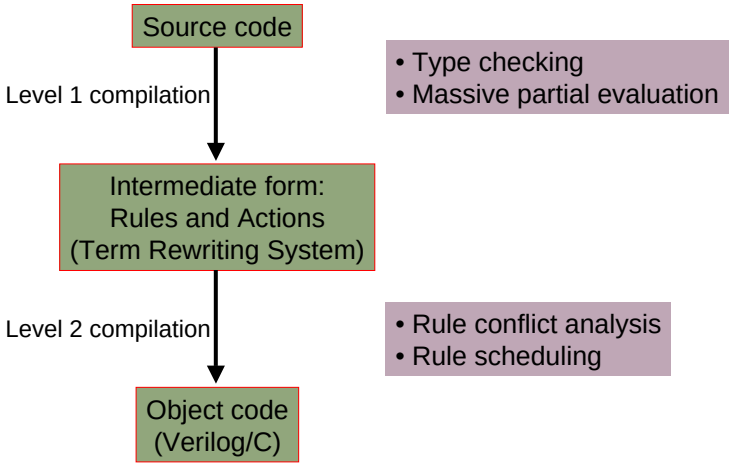



```
"Add":
  when (Add rd rs rt) <- bu.first
        ==> action      rf!rd := rf!rs + rf!rt
                        bu.deq
"Bz Not Taken":
  when (Bz rc ra) <- bu.first, rf!rc /= 0
        ==> action      bu.deq
"Bz Taken":
  when (Bz rc ra) <- bu.first, rf!rc == 0
        ==> action      pc := rf!ra
                        bu.clear
```

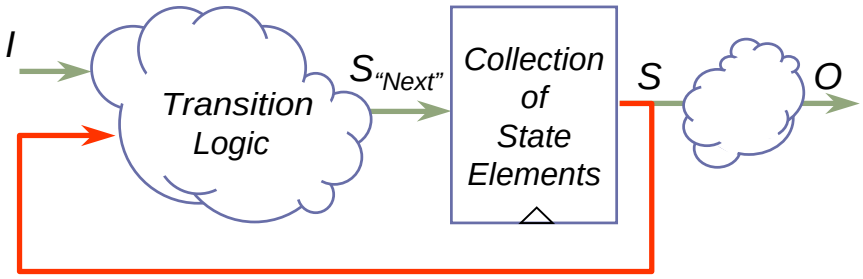

Outline

- Preliminaries √
- A new semantic model for hardware description: TRS √
- An example: A simple pipelined CPU √
- Bluespec compilation

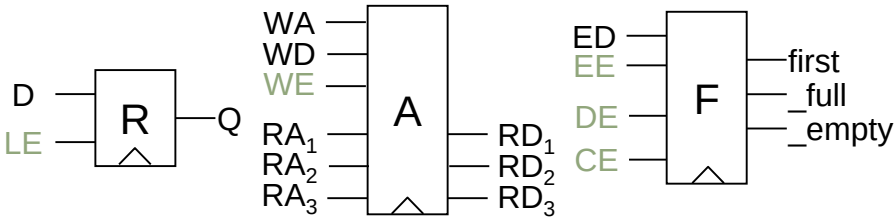
Bluespec: A two-level language



From TRS to Synchronous CFSM



Synchronous State Elements



Bit[N]
Tag[N]

Array

Fifo

TRS Execution Semantics

Given a set of rules and an initial term s

While (some rules are applicable to s)
- choose an applicable rule
  (non-deterministic)
- apply the rule atomically to s

The trick to generating good hardware is to schedule as many rules in parallel as possible without violating the sequential semantics given above

Rule: As a State Transformer

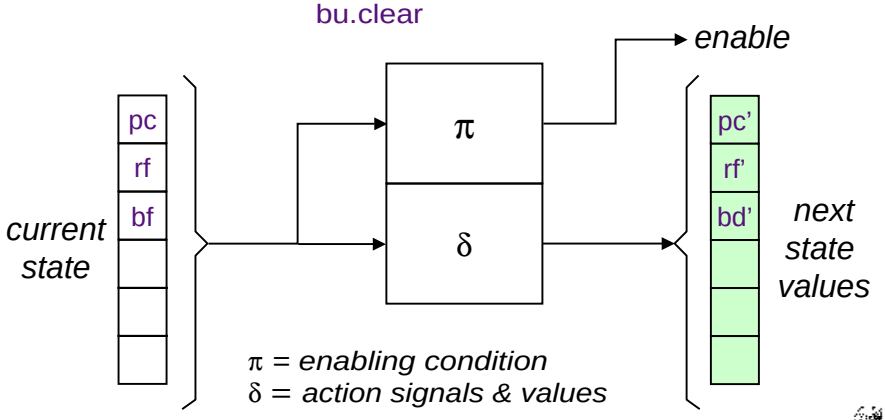
- A rule may be decomposed into two parts π(s) and δ(s) such that

$$s_{next} = \textit{if } \pi(s) \textit{ then } \delta(s) \textit{ else } s$$

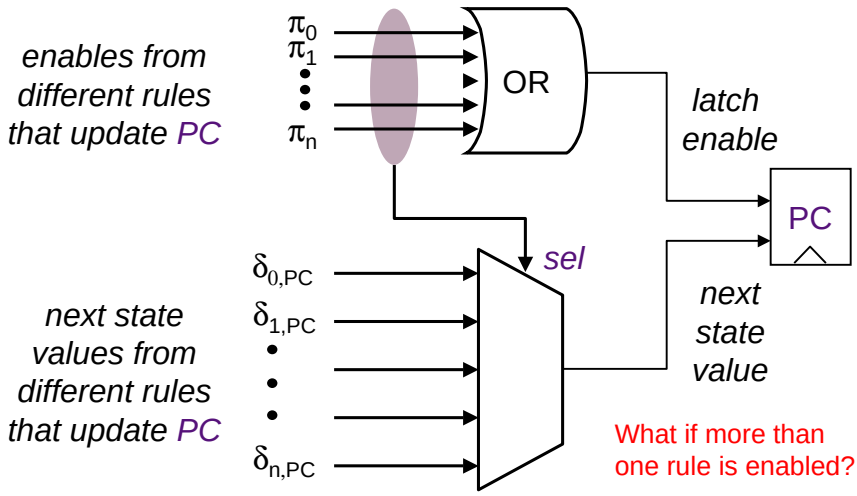
δ(s) is expressed as (atomic) actions on the state elements. These actions can be enabled only if π(s) is true

Compiling a Rule

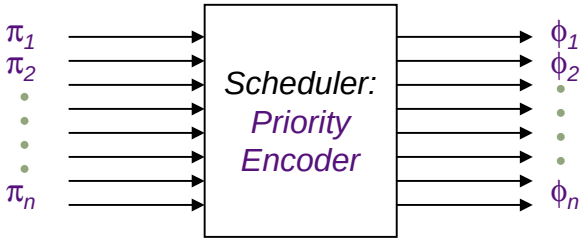
```
"Bz Taken":
  when (Bz rc ra) <- bu.first, rf!rc == 0
    ==> action pc := rf!ra
                bu.clear
```



Combining State Updates



Single-rewrite-per-cycle Scheduler



1. $\phi_i \Rightarrow \pi_i$

2. $\pi_1 \vee \pi_2 \vee .... \vee \pi_n \Rightarrow \phi_1 \vee \phi_2 \vee .... \vee \phi_n$

3. *One rewrite at a time*
   i.e. at most one ϕ_i is true

Executing Multiple Rules Per Cycle

```
"Fetch":
  when True
    ==> action  pc := pc+1
                bu.enq (imem.read pc)
"Add":
  when (Add rd rs rt) <- bu.first
    ==> action  rf!rd := rf!rs + rf!rt
                bu.deq
```

Can these rules be executed simultaneously?

Conflict-Free Rules

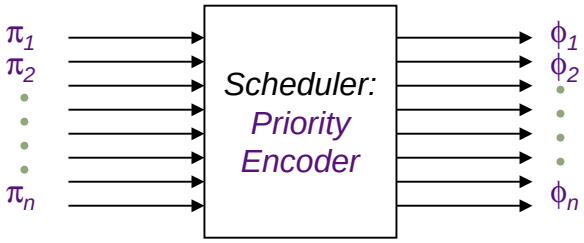
Rule$_a$ and Rule$_b$ are conflict-free if

$\forall s \,.\, \pi_a(s) \wedge \pi_b(s) \Rightarrow$

1. $\pi_a(\delta_b(s)) \wedge \pi_b(\delta_a(s))$
2. $\delta_a(\delta_b(s)) == \delta_b(\delta_a(s))$
3. $\delta_a(\delta_b(s)) == \delta_a(s) \oplus \delta_b(s)$

Theorem: Conflict-free rules can be executed concurrently without violating TRS's sequential semantics

Multiple-rewrite-per-cycle Scheduler



1. $\phi_i \Rightarrow \pi_i$

2. $\pi_1 \vee \pi_2 \vee .... \vee \pi_n \Rightarrow \phi_1 \vee \phi_2 \vee .... \vee \phi_n$

3. $\phi_i \wedge \phi_j \Rightarrow$ *Rule$_i$ and Rule$_j$ are "conflict-free"*

Multiple Rewrites Per Cycle

```
"Fetch":
  when True
        ==> action    pc := pc+1
                      bu.enq (imem.read pc)
"Bz Taken":
  when (pc' , Bz rc ra) <- bu.first, rf!rc == 0
        ==> action    pc := rf!ra
                      bu.clear
```

Can these rules be executed simultaneously?