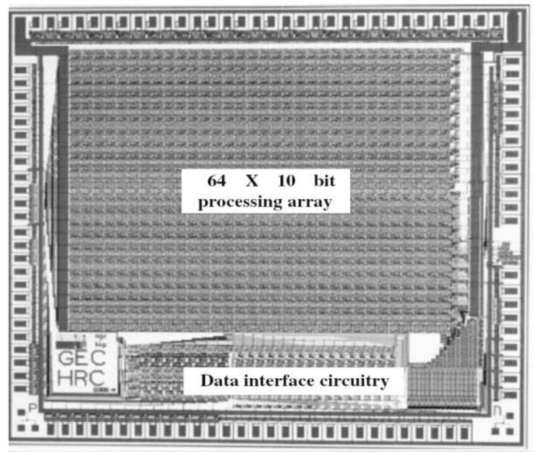
Design description and optimisation

- FPGA technology offers customisation opportunities
  - some data may remain constant: e.g. algebraic simplification
  - adopt different data structures: e.g. number representation
  - transform: e.g. enhance parallelism, pipelining, serialisation
- re-use possibilities
  - description: repeating unit, parametrisation
  - transforms: patterns, laws, proofs
  - optimisation: improve efficiency, parallelism, regularity
- industrial languages e.g.VHDL or Verilog
  - complex, over 50% of time on design validation
- Ruby: declarative block description language
  - customise design by parametrisation and transformation (compile time)
  - simple and concise, verification of designs and transformations

What has Ruby been used for?

- signal processing architectures, especially regular ones
  - linear and non-linear filters, butterfly circuits for FFT, DCT
- non-numercial architectures
  - sorters, DNA sequence matchers, priority queues, LRU designs
- arithmetic building blocks
  - add, multiply, divide, square root
  - retarget for various number representations



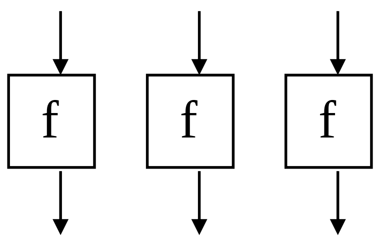
- data parallel and data serial designs
  - derive implementations with different trade-offs
- design containing hardware and software components
  - algebraic laws useful for partitioning into hardware and software
- Intel supports related work for next-generation hardware
- novel designs which are non-obvious but correct...

Relating text description and picture

- Ruby is similar to Haskell or Miranda:
  - polymorphic types e.g. id $x = x$
  - higher-order functions e.g. map

- Haskell - map f : applies f to all elements of a list

$xs = [x_0,\ x_1,\ x_2]$

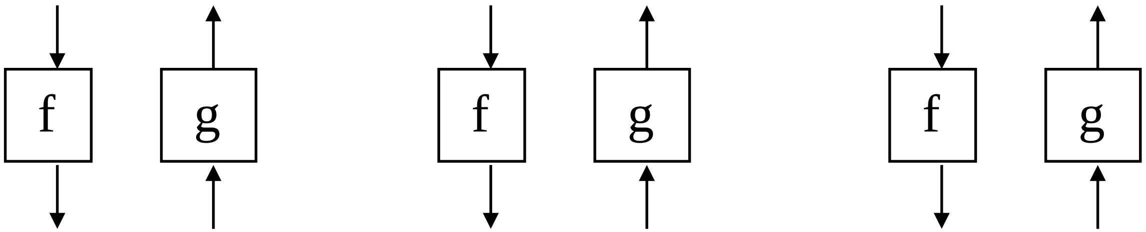


$ys = [y_0,\ y_1,\ y_2] \quad = \quad \text{map } f \text{ xs} = [f\ x_0,\ f\ x_1,\ f\ x_2]$

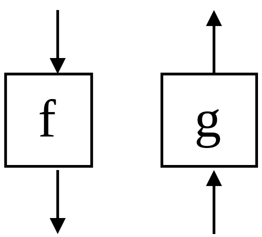
- e.g. f = inc where inc $x = x + 1$
  - Haskell : map inc
  - Ruby : map_n inc (n=3 for the above picture)
  - Rebecca : map n inc (current Ruby tools, including compiler)

How about more complex data flow?

- designs (blocks) containing counter-flowing data



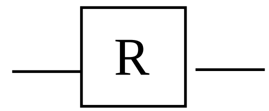
- Haskell: ?
  Ruby: map_3 [f, g]



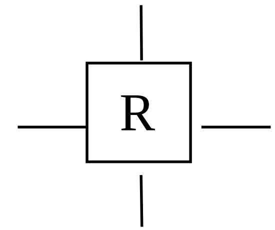
[f, g] (parallel composition)

Representing designs in Ruby

- component classification:

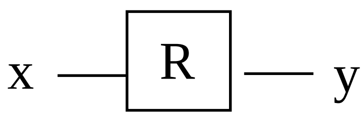


chain component



grid component

- design described by a binary relation:



$x \mathrm{R}\, y$

domain range

(focus on interfaces: how the block can be connected)

- example: x inc2 y $\Leftrightarrow$ $x + 2 = y$ or x inc2 $(x + 2)$
  Haskell: inc2 x = $x + 2$ = inc (inc x), inc $x = x + 1$
  Rebecca: ``inc2 = VAR x . x $rel (`inc`(`inc` x)).``

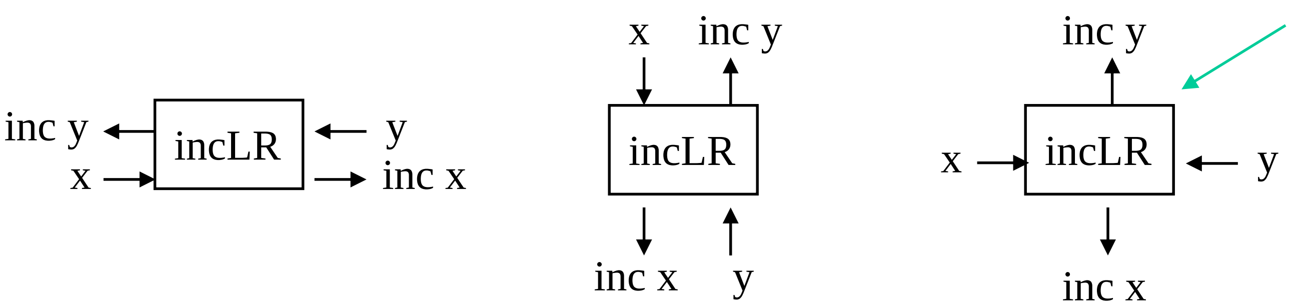
Ruby tools

do not forget the dot!

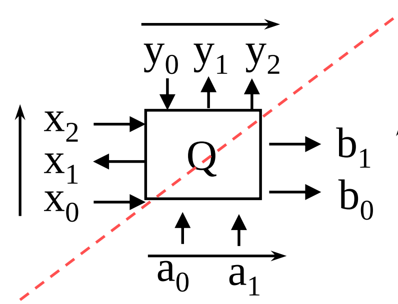
- if R is a function, then $x \mathrm{R}\, y$ can be written as $y = \mathrm{R}\, x$

Composite data

- no distinction between list/vector and tuple/record
  - <1, 2, 3, 4, 5> : $list_5$ uint — unsigned integer
  - <T, <F, T>> : <bit, <bit, bit>> — not always obvious from pictures
- inputs can be in range, outputs can be in domain
  - e.g. <*x*, inc *y*> incLR <inc *x*, *y*>

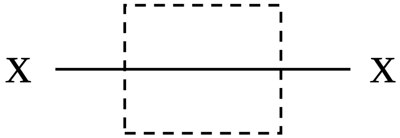


grid component:
relates pair to pair

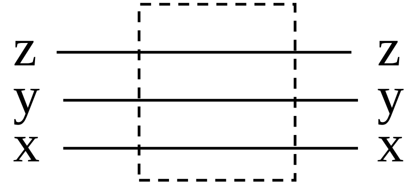


- convention: count from bottom-up or left-right, so $\langle\langle x_0,x_1,x_2\rangle, \langle y_0,y_1,y_2\rangle\rangle$ Q $\langle\langle a_0,a_1\rangle, \langle b_0,b_1\rangle\rangle$
- Rebecca: ``incLR = VAR x y.<x,`inc`y> $rel <`inc`x, y>.``

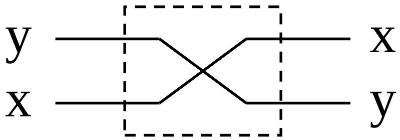
Wire blocks: polymorphic – one for many

- 

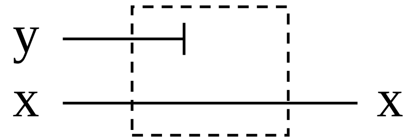
  x id x

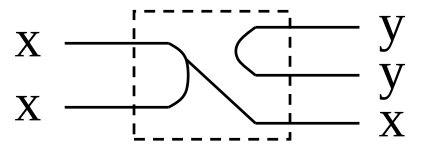
  <x, <y, z>> id <x, <y, z>>

- 

  <x,y> swap <y,x>

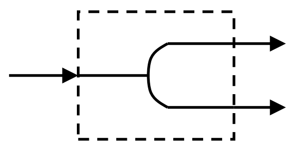
  

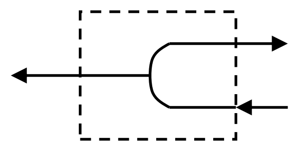
  <x,y> π_1 x
  `pi1`

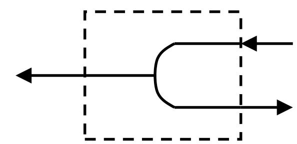
  

  <x,x> mywire <x,y,y>

- connected wires: one input, multiple outputs
  – e.g. fork

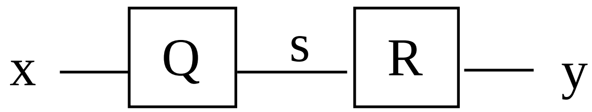






- Rebecca:
```
fork = VAR x . x $rel <x, x>.
```
  or
```
fork = x $wire <x, x>.
```

Series composition

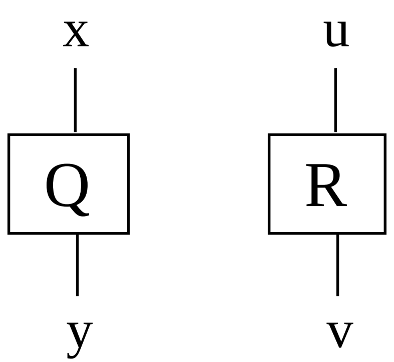
- 

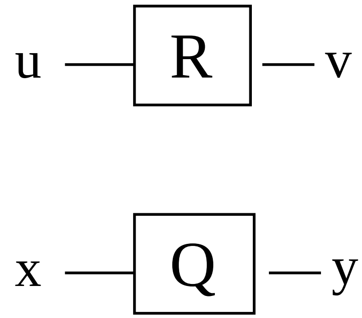
  $$x\,(Q\,;R)\,y \;\Leftrightarrow\; \exists\, s\,.\;\; x\,Q\,s \,\wedge\, s\,R\,y$$

  there exists hidden, local variable

- example:
  - x inc $(x + 1)$ then x (inc ; inc) $(x + 2)$
  - x double $(x + x)$ then x (double ; inc) $(2x + 1)$
  - x sq x^2 then x (inc ; sq) $(x + 1)^2$
- also known as a *higher-order function* or a *combinator*: combines 2 components into a composite design
- "built-in" operator in Ruby compiler
- key: understand the type of local variable s

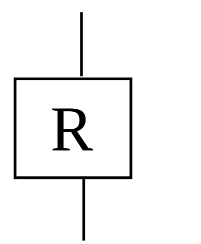
Parallel composition

- 



$$\langle x, u\rangle \, [Q, R] \, \langle y, v\rangle \quad \Leftrightarrow \quad x \; Q \; y \;\wedge\; u \; R \; v$$

- law: $[\,(P\,;\,Q\,),\,(R\,;\,S)\,] \quad = \quad [P, R]\,;\,[Q, S]$
- abbreviations:
  - **`fst R = [R, id].`**
  - **`snd R = [id, R].`**



fst R

Rebecca: tools for Ruby

- for Department of Computing Linux PC
- in your work directory, execute the command

  source /homes/wl/public/setup.ruby

- this sets up the alias for
  - Ruby compiler rc
  - Ruby evaluator re
- this also copies two Ruby files to your work directory
  - prelude.rby: library definitions
  - egs.rby: many simple examples

rbs format

- **`rc`** ***`file.rby`***: compile ***`file.rby`*** into ***`current.rbs`***
- **`re`** **"*`sim-data`*"**: simulate ***`current.rbs`***, input ***`sim-data`***

Flattened netlist description: rbs format

- a block is described by:
  name, domain (input wire), range (output wire)
- wire name: a dot followed by an integer, e.g. **`.3`**
- direction/wire name of overall block i/o wires
- word-level blocks e.g. **`add`**, bit-level blocks e.g. **`and`**
- an example block containing 3 components:

```
Name   Domain       Range
-----------------------------
mult   <•1,•1>       •2
add    <•1,•1>       •3
inc     •2            •4
-----------------------------
Directions - <in,out> ~ out
Wiring     - <•1,•4>  ~ •3
Inputs     - •1
```

Using Rebecca compiler and evaluator

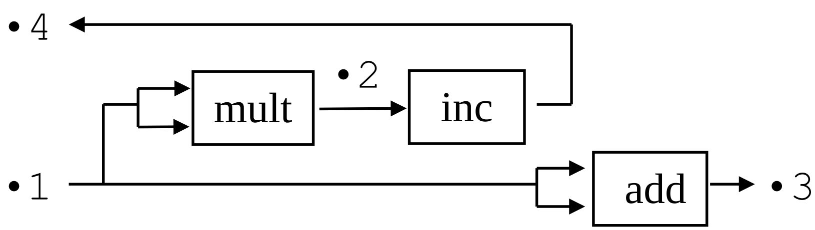
input output

- prepare a file, say **`test.rby`** for $\langle x, x^2{+}1\rangle \; R \; (x + x)$

comment
include prelude
end of definition
expression to be compiled

```
# A test file
INCLUDE "prelude.rby".
both f = fork ; f.
current = VAR in . <in,`both mult ; inc` in>
              $rel (`both add` in).
```

- **`rc test.rby`** produces **`current.rbs`**:




```
Name        Domain      Range
------------------------------
mult        <•1,•1>      •2
add         <•1,•1>      •3
inc           •2         •4
------------------------------
Directions  - <in,out> ~ out
Wirings     - <•1,•4>  ~ •3
Inputs      - •1
```

wire name

- **`re "2; 4.6; x"`**

cycle domain range

0 - <2, 5> ~ 4

1 - <4.6, 22.16> ~ 9.2

2 - <x, (inc(x*x))> ~ (x+x)

(shows one input per cycle)

Observations

- re is versatile
  - supports numerical and symbolic simulation
  - has Boolean/integer/real primitives
  - has non-implementable primitives, e.g. bit2unit, AD
    see simulator primitives link in course homepage
- rbs simple, but tedious to produce by hand
  - flattened, no sub-blocks, no parametric description, no loops
  - no facilities for design re-use
- Ruby: high-level language features for blocks
  - simple mapping: between description and picture
  - block description: parametrisable and composable
  - functions: customise common patterns of composition
  - algebraic laws: formalise, codify and generalise design intuition
  - use: rapid development, generate flexible building blocks

Compare rbs and Rebecca Ruby

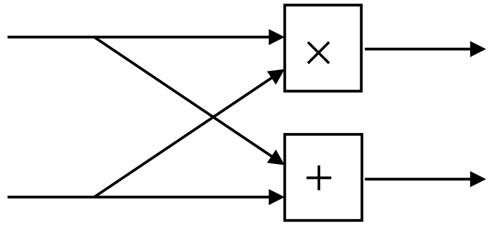
	rbs	Ruby
• wire name format	restricted	flexible
• wire variable	monomorphic	polymorphic
• block description	flattened	hierarchical
• block interface	separate i/o	mixed i/o
• block parameter	not allowed	size, block
• connection	identical wire between ports	$rel, combinators
• recursion	not supported	supported
• connection pattern	not supported	recursion

Notes

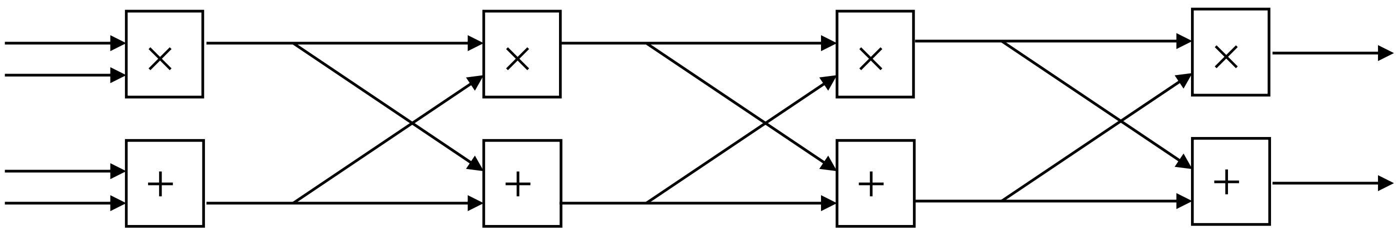
- many examples can be found in course homepage, e.g.
  - https://www.doc.ic.ac.uk/~wl/teachlocal/cuscomp/rebecca/egs.rby
- make sure you get the brackets (the types) right
  - e.g. the domain/range for [[P,Q,[R,S]],T]: <<p,q,<r,s>>,t>
- if **R** is a function, then **x R y** can be written as **y = R x**
- in Rebecca, use `` `R` x `` only when **R** is a function
  - e.g. `` inc3' = VAR x . (`inc; inc; inc` x) $rel x ``
    (there is a more concise way of defining inc3')
- **(((x))) = x** but **(x, x)** is illegal; use **<x, x>**
- LET cannot be used in **$rel** or **$wire** definitions
  - also cannot have local functions using **LET**:
    g = LET both f = fork;f IN … is illegal

Unassessed Coursework 1

1. Describe the following in Ruby (use/not use $rel):

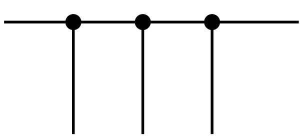


2. Describe and simulate the following design:

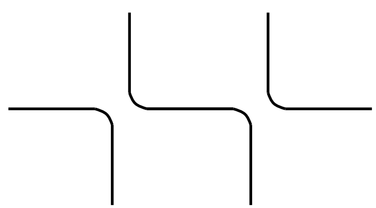


3. Describe and simulate the following designs:

(a)



(b)



(c)