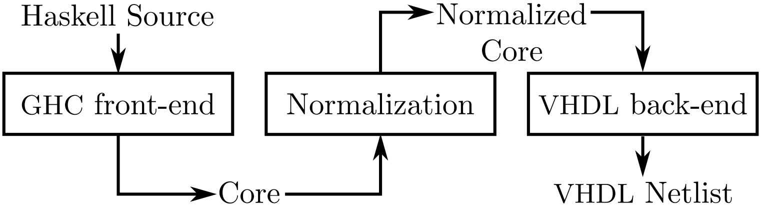
Haskell-ish Hardware Descriptions

Matthijs Kooijman, Christiaan Baaij, Jan Kuper

Dutch Haskell user group day, 2010

Compiler



Haskell Source

GHC front-end

Core

Normalization

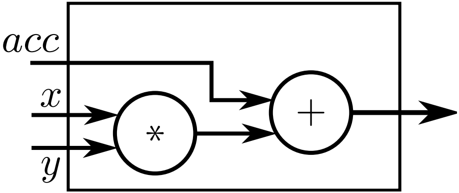
Normalized Core

VHDL back-end

VHDL Netlist

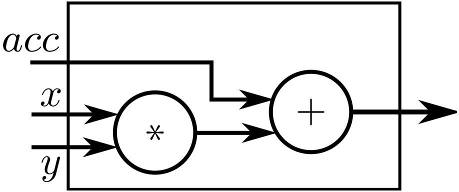
Multiply-accumulate

```
mac :: Num a => a → a → a → a

mac x y acc = acc + x * y
```



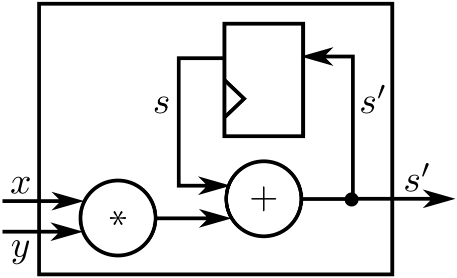
Multiply-accumulate

```
type Word = SizedWord D16
mac :: Word → Word → Word → Word

mac x y acc = acc + x * y
```

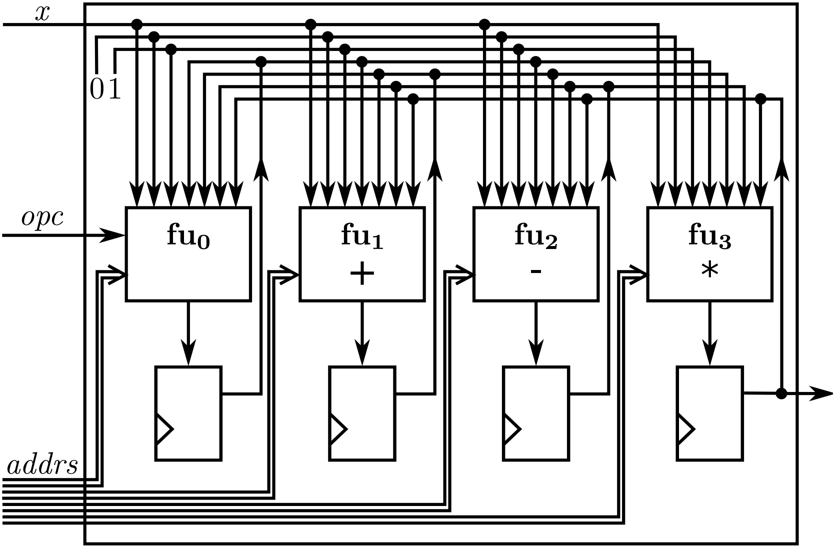


Stateful multiply-accumulate

```
newtype State a = State a

smac :: State Word → Word → Word → (State Word, Word)
smac (State s) x y = (State s', s')
  where s' = s + x * y
```



Simple CPU



Fixed function function units

```
fu :: (... u ~ n :-: D1 ...) => (a → a → t)
                 → Vector n a
                 → (Index u, Index u)
                 → t

fu op inputs (a1, a2) = op (inputs!a1) (inputs!a2)
```

```
fu1 = fu (+)
fu2 = fu (-)
fu3 = fu (*)
```

Multi-purpose function unit

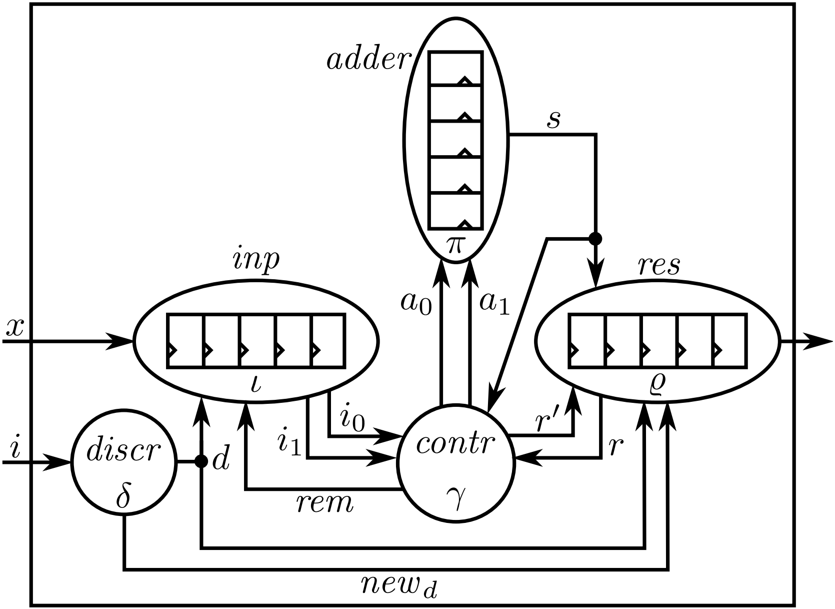
```
data Opcode = Shift | Xor | Equal

multiop :: Opcode → Word → Word → Word
multiop Shift   = shift
multiop Xor     = xor
multiop Equal   = λa b → if a ≡ b then 1 else 0
```

```
fu0 c = fu (multiop c)
```

The complete CPU

```
type CpuState = State (Vector D4 Word)

cpu :: CpuState
  → (Word, Opcode, Vector D4 (Index D6, Index D6))
  → (CpuState, Word)
cpu  (State s) (x, opc, addrs) = (State s’, out)
  where
    inputs = x +> (0 +> (1 +> s))
    s’ = (fu0 opc inputs (addrs!(0 :: Index D3))) +> (
         (fu1     inputs (addrs!(1 :: Index D3))) +> (
         (fu2     inputs (addrs!(2 :: Index D3))) +> (
         (fu3     inputs (addrs!(3 :: Index D3))) +> (
         (empty)))))
    out = last s
```

Floating point reduction circuit



Controller function

```
controller (inp1, inp2, pT, from_res_mem) =
    (arg1, arg2, shift, to_res_mem)
  where
    (arg1, arg2, shift, to_res_mem)
      | valid pT ∧ valid from_res_mem
        = (pT , from_res_mem , 0, False)
      | valid pT ∧ valid inp1 ∧ discr pT ≡ discr inp1
        = (pT , inp1 , 1, False)
      | valid inp1 ∧ valid inp2 ∧ discr inp1 ≡ discr inp2
        = (inp1 , inp2 , 2, valid pT)
      | valid inp1
        = (inp1 , (True, (0, discr inp1)) , 1, valid pT)
      | otherwise
        = (notValid, notValid , 0, valid pT)
```

Future work

- More systematic normalization
- Recursion / normal lists
- Nested state abstraction
- Multiple clock domains / asynchronicity
- Graphical output

Thanks!

http://wwwhome.cs.utwente.nl/ baaijcpr/ClaSH/Index.html

or just

http://google.com/search?q=**C**λ**aSH**&btnI=I'm Feeling Lucky