Hardware Design with Generalized Arrows

Adam Megacz
megacz@cs.berkeley.edu

03.Oct.2011

This Project

- First nontrivial application of *generalized arrows*.
- Not (even close to) a complete circuit-design solution.

Metaprogramming and the Milner Property

- A metaprogram is a program which produces a program (called the *object program*).
- *Metaprogramming* is the act of writing metaprograms.
- The Milner Property: "well-typed programs don't go wrong."
  - When one is metaprogramming, we want something stronger: *well-typed metaprograms should not be able to produce ill-typed object programs* (therefore the object programs can't go wrong).

Metaprogramming and the Milner Property

- A metaprogram is a program which produces a program (called the *object program*).
- *Metaprogramming* is the act of writing metaprograms.
- The Milner Property: "well-typed programs don't go wrong."
  - When one is metaprogramming, we want something stronger: *well-typed metaprograms should not be able to produce ill-typed object programs* (therefore the object programs can't go wrong).

Metaprogramming and the Milner Property

- A metaprogram is a program which produces a program (called the *object program*).
- *Metaprogramming* is the act of writing metaprograms.
- The Milner Property: "well-typed programs don't go wrong."
  - When one is metaprogramming, we want something stronger: *well-typed metaprograms should not be able to produce ill-typed object programs* (therefore the object programs can't go wrong).

Metaprogramming and the Milner Property

- A metaprogram is a program which produces a program (called the *object program*).
- *Metaprogramming* is the act of writing metaprograms.
- The Milner Property: "well-typed programs don't go wrong."
  - When one is metaprogramming, we want something stronger: *well-typed metaprograms should not be able to produce ill-typed object programs* (therefore the object programs can't go wrong).

What Problem Do Generalized Arrows Solve?

Generalized arrows are a representation for object programs which:

- Has the "Milner property for metaprograms"
- Does not assume that the object language's expressions are a superset of the metalanguage (like Monads and classic Arrows do).
  - A monad's `return` lifts arbitrary Haskell functions into the monad
  - An arrow's `arr` lifts arbitrary Haskell functions into the arrow
- Does not assume that the object language's typing judgments are a superset of the metalanguage's typing judgments (like Monads and classic Arrows do).
  - Monadic metaprogramming cannot handle object languages with:
    - affine types, because of `(return $ \_ -> ())`
    - linear types, because of `(return $ \x -> (x,x))`
    - ordered types, because of `(return $ \(x,y) -> (y,x))`
  - Likewise for arrows.

What Problem Do Generalized Arrows Solve?

Generalized arrows are a representation for object programs which:

- Has the "Milner property for metaprograms"
- Does not assume that the object language's expressions are a superset of the metalanguage (like Monads and classic Arrows do).
  - A monad's `return` lifts arbitrary Haskell functions into the monad
  - An arrow's `arr` lifts arbitrary Haskell functions into the arrow
- Does not assume that the object language's typing judgments are a superset of the metalanguage's typing judgments (like Monads and classic Arrows do).
  - Monadic metaprogramming cannot handle object languages with:
    - affine types, because of `(return $ \_ -> ())`
    - linear types, because of `(return $ \x -> (x,x))`
    - ordered types, because of `(return $ \(x,y) -> (y,x))`
  - Likewise for arrows.

What Problem Do Generalized Arrows Solve?

Generalized arrows are a representation for object programs which:

- Has the "Milner property for metaprograms"
- Does not assume that the object language's expressions are a superset of the metalanguage (like Monads and classic Arrows do).
  - A monad's `return` lifts arbitrary Haskell functions into the monad
  - An arrow's `arr` lifts arbitrary Haskell functions into the arrow
- Does not assume that the object language's typing judgments are a superset of the metalanguage's typing judgments (like Monads and classic Arrows do).
  - Monadic metaprogramming cannot handle object languages with:
    - affine types, because of `(return $ \_ -> ())`
    - linear types, because of `(return $ \x -> (x,x))`
    - ordered types, because of `(return $ \(x,y) -> (y,x))`
  - Likewise for arrows.

What Problem Do Generalized Arrows Solve?

Generalized arrows are a representation for object programs which:

- Has the "Milner property for metaprograms"
- Does not assume that the object language's expressions are a superset of the metalanguage (like Monads and classic Arrows do).
  - A monad's `return` lifts arbitrary Haskell functions into the monad
  - An arrow's `arr` lifts arbitrary Haskell functions into the arrow
- Does not assume that the object language's typing judgments are a superset of the metalanguage's typing judgments (like Monads and classic Arrows do).
  - Monadic metaprogramming cannot handle object languages with:
    - affine types, because of `(return $ \_ -> ())`
    - linear types, because of `(return $ \x -> (x,x))`
    - ordered types, because of `(return $ \(x,y) -> (y,x))`
  - Likewise for arrows.

What Problem Do Generalized Arrows Solve?

Generalized arrows are a representation for object programs which:

- Has the "Milner property for metaprograms"
- Does not assume that the object language's expressions are a superset of the metalanguage (like Monads and classic Arrows do).
  - A monad's `return` lifts arbitrary Haskell functions into the monad
  - An arrow's `arr` lifts arbitrary Haskell functions into the arrow
- Does not assume that the object language's typing judgments are a superset of the metalanguage's typing judgments (like Monads and classic Arrows do).
  - Monadic metaprogramming cannot handle object languages with:
    - affine types, because of `(return $ \_ -> ())`
    - linear types, because of `(return $ \x -> (x,x))`
    - ordered types, because of `(return $ \(x,y) -> (y,x))`
  - Likewise for arrows.

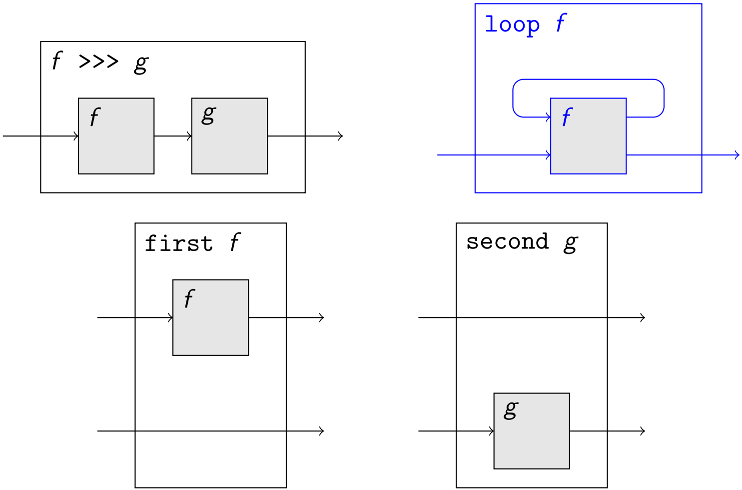
What Problem Do Generalized Arrows Solve?

Generalized arrows are a representation for object programs which:

- Has the "Milner property for metaprograms"
- Does not assume that the object language's expressions are a superset of the metalanguage (like Monads and classic Arrows do).
  - A monad's `return` lifts arbitrary Haskell functions into the monad
  - An arrow's `arr` lifts arbitrary Haskell functions into the arrow
- Does not assume that the object language's typing judgments are a superset of the metalanguage's typing judgments (like Monads and classic Arrows do).
  - Monadic metaprogramming cannot handle object languages with:
    - affine types, because of `(return $ \_ -> ())`
    - linear types, because of `(return $ \x -> (x,x))`
    - ordered types, because of `(return $ \(x,y) -> (y,x))`
  - Likewise for arrows.

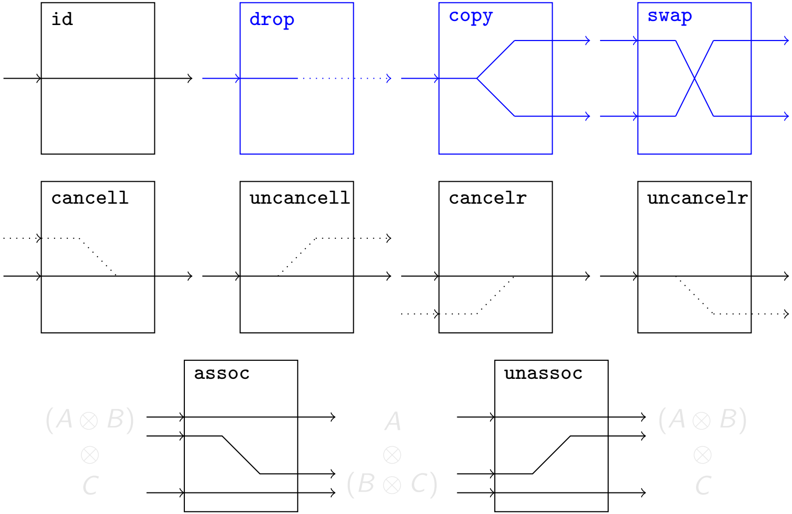
What are Generalized Arrows? (1/2)

Four *operations* on elements (`loop` is defined in a subclass, `GArrowLoop`):



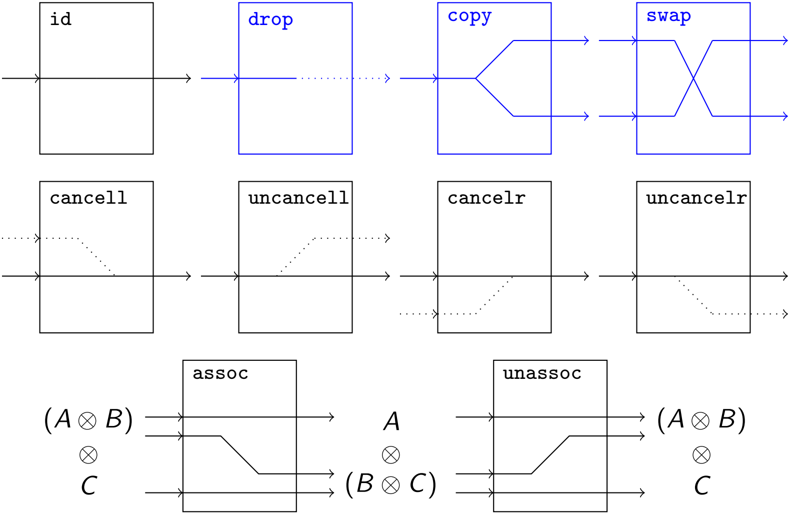
What are Generalized Arrows? (2/2)

... and ten primitive elements (`drop`, `copy`, and `swap` are defined in subclasses):



What are Generalized Arrows? (2/2)

... and ten primitive elements (drop, copy, and swap are defined in subclasses):



Generalized Arrows

```
class Category g => GArrow g (**) u where
--id            :: g x x
--(>>>)         :: g x y -> g y z -> g x z
  ga_first      :: g x y -> g (x ** z) (y ** z)
  ga_second     :: g x y -> g (z ** x) (z ** y)

  ga_cancell    :: g (u**x)         x
  ga_cancelr    :: g    (x**u)      x
  ga_uncancell  :: g     x        (u**x)
  ga_uncancelr  :: g     x           (x**u)

  ga_assoc      :: g ((x** y)**z ) ( x**(y **z))
  ga_unassoc    :: g ( x**(y **z)) ((x** y)**z )

  ga_copy       :: g x (x**x)
  ga_drop       :: g x u
  ga_swap       :: g (x**y) (y**x)

  ga_loop       :: g (x**z) (y**z) -> g x y
```

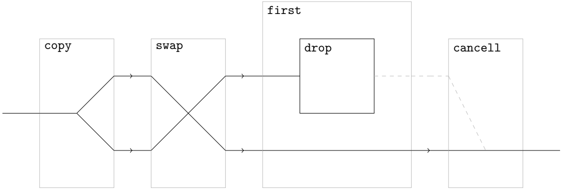
Every Arrow is a GArrow

```
instance Arrow a => GArrow a (,) () where
  ga_first      =  first
  ga_second     =  second
  ga_cancell    =  arr (\((),x) -> x)
  ga_cancelr    =  arr (\(x,()) -> x)
  ga_uncancell =  arr (\x -> ((),x))
  ga_uncancelr =  arr (\x -> (x,()))
  ga_assoc      =  arr (\((x,y),z) -> (x,(y,z)))
  ga_unassoc    =  arr (\(x,(y,z)) -> ((x,y),z))

instance Arrow a => GArrowDrop a (,) () where
  ga_drop       =  arr (\x -> ())

instance Arrow a => GArrowCopy a (,) () where
  ga_copy       =  arr (\x -> (x,x))

instance Arrow a => GArrowSwap a (,) () where
  ga_swap       =  arr (\(x,y) -> (y,x))

instance ArrowLoop a => GArrowLoop a (,) () where
  ga_loop       =  loop
```

… but `GArrow` does not let *arbitrary Haskell functions "leak" in* since there is no `arr`.

Example

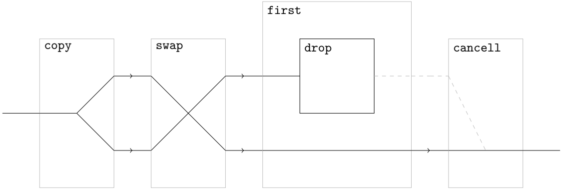
```
sample1 =
  ga_copy            >>>
  ga_swap            >>>
  ga_first ga_drop >>>
  ga_cancell
```



The text format above is nice for *processing* `GArrow` expressions. In fact, all of the diagrams in the paper and these slides were produced by the `GArrowTikZ` instance, which emits TikZ code for these diagrams. Unfortunately it is a pain for users to write `GArrow` expressions this way.

Example

```
sample1 =
  ga_copy            >>>
  ga_swap            >>>
  ga_first ga_drop >>>
  ga_cancell
```



The text format above is nice for *processing* `GArrow` expressions. In fact, all of the diagrams in the paper and these slides were produced by the `GArrowTikZ` instance, which emits TikZ code for these diagrams. Unfortunately it is a pain for users to write `GArrow` expressions this way.

Solution: Two-Level Expressions and Types

e_0 ::= ... | `<[`e_1`]>` (level-0 expressions)

e_1 ::= ... | `~~`e_0 (level-1 expressions)

τ_0 ::= ... | `<[`τ_1`]>@`α (level-0 types)

τ_1 ::= ... (level-1 types)

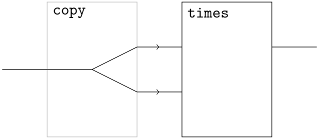
Flattening is a translation from two-level expressions to one-level expressions *by induction on the typing derivation*.

- Translation by induction on the expression's typing derivation.
- $[\![\Gamma \vdash^{\alpha} \tau]\!] =$ `GArrow g => g` $[\![\Gamma]\!]$ $[\![\tau]\!]$
- Structural rules {weakening, exchange, contraction} become primitive elements {`ga_drop`, `ga_swap`, `ga_copy`}.
- Cut/Let becomes `(>>>)`
- LetRec becomes `loop`
- Var becomes `id`

Gory details in [Meg11] (preprint on arXiv).

Flattening Examples

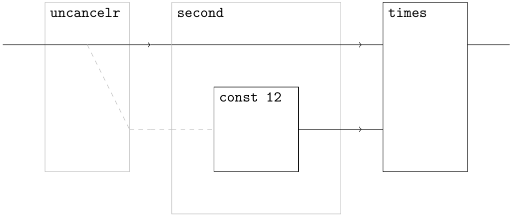
```
demo1 :: <[ (a,a)~~> b ]>@z ->
         <[    a ~~> b ]>@z

demo1 times =
  <[ \y -> ~~times y y ]>
```



This diagram, and all the rest on future slides, were produced by running the flattener and instantiating the resulting term with `GArrowTikZ`

Two-Level Syntax

```
demo2 :: (Int -> <[ ()~~>a ]>@z) ->
           <[ (b,a)~~> c ]>@z ->
           <[     b ~~> c ]>@z

demo2 const times =
   <[ \y -> ~~times y ~~(const 12) ]>
```



Shallow/Deep/Multi-Level Embeddings

- In a *shallow embedding*, the Haskell program *is the circuit*.
- In a *deep embedding*, the Haskell program explains how to *build the circuit*.
- In a *multi-level embedding*, the level-1 terms *are the circuit* and the level-0 terms *build the circuit*.

Shallow/Deep/Multi-Level Embeddings

- In a *shallow embedding*, the Haskell program *is the circuit*.
- In a *deep embedding*, the Haskell program explains how to *build the circuit*.
- In a *multi-level embedding*, the level-1 terms *are the circuit* and the level-0 terms *build the circuit*.

Multi-Level Embeddings let the Types Say More

A shallow embedding's type system does not distinguish between:

- A circuit with input type `A` and output type `B`.
- A program which takes a circuit of output type `A` and uses it to build a circuit of output type `B`.

A *multi-level* embedding makes this distinction in its types:

```
circuit            :: <[ A~~>B ]>
circuitTransformer :: <[ x~~>A ]> -> <[ x~~>B ]>
```

Why distinguish these? One answer: to distinguish *feedback* from *unrolling*.

- Monadic deep embeddings do this by distinguishing between `mfix` from (recursive) `let`.

Multi-Level Embeddings let the Types Say More

A shallow embedding's type system does not distinguish between:

- A circuit with input type `A` and output type `B`.
- A program which takes a circuit of output type `A` and uses it to build a circuit of output type `B`.

A *multi-level* embedding makes this distinction in its types:

```
circuit              :: <[ A~~>B ]>
circuitTransformer :: <[ x~~>A ]> -> <[ x~~>B ]>
```

Why distinguish these? One answer: to distinguish *feedback* from *unrolling*.

- Monadic deep embeddings do this by distinguishing between `mfix` from (recursive) `let`.

Multi-Level Embeddings let the Types Say More

A shallow embedding's type system does not distinguish between:

- A circuit with input type `A` and output type `B`.
- A program which takes a circuit of output type `A` and uses it to build a circuit of output type `B`.

A *multi-level* embedding makes this distinction in its types:

```
circuit            :: <[ A~~>B ]>
circuitTransformer :: <[ x~~>A ]> -> <[ x~~>B ]>
```

Why distinguish these? One answer: to distinguish *feedback* from *unrolling*.

- Monadic deep embeddings do this by distinguishing between `mfix` from (recursive) `let`.

Multi-Level Embeddings let the Types Say More

A shallow embedding's type system does not distinguish between:

- A circuit with input type `A` and output type `B`.
- A program which takes a circuit of output type `A` and uses it to build a circuit of output type `B`.

A *multi-level* embedding makes this distinction in its types:

```
circuit            :: <[ A~~>B ]>
circuitTransformer :: <[ x~~>A ]> -> <[ x~~>B ]>
```

Why distinguish these? One answer: to distinguish *feedback* from *unrolling*.

- Monadic deep embeddings do this by distinguishing between `mfix` from (recursive) `let`.

Multi-Level Embeddings let the Types Say More

A shallow embedding's type system does not distinguish between:

- A circuit with input type `A` and output type `B`.
- A program which takes a circuit of output type `A` and uses it to build a circuit of output type `B`.

A *multi-level* embedding makes this distinction in its types:

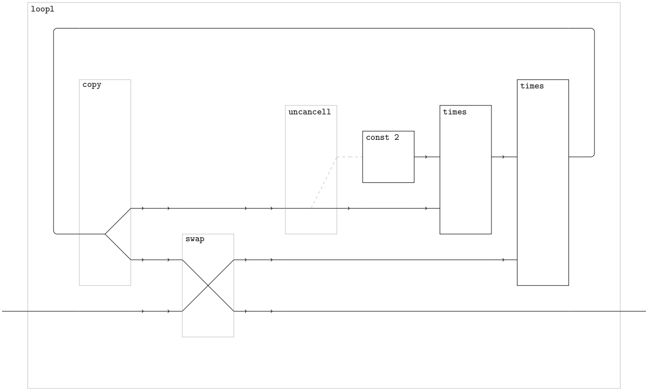
```
circuit            :: <[ A~~>B ]>
circuitTransformer :: <[ x~~>A ]> -> <[ x~~>B ]>
```

Why distinguish these? One answer: to distinguish *feedback* from *unrolling*.

- Monadic deep embeddings do this by distinguishing between `mfix` from (recursive) `let`.

Feedback

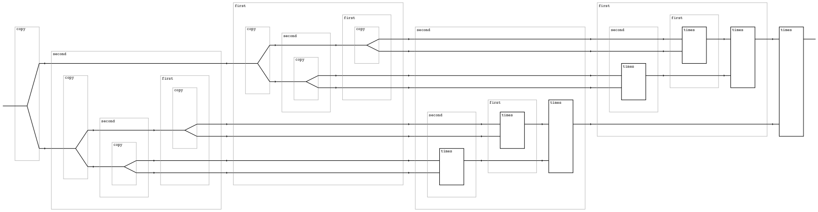
```
demo const times =
   <[ \x ->
      let out   = ~~times (~~times ~~(const 2) out) x
       in out
    ]>
```



Slogan: "Recursion (letrec) *inside* the brackets means feedback."

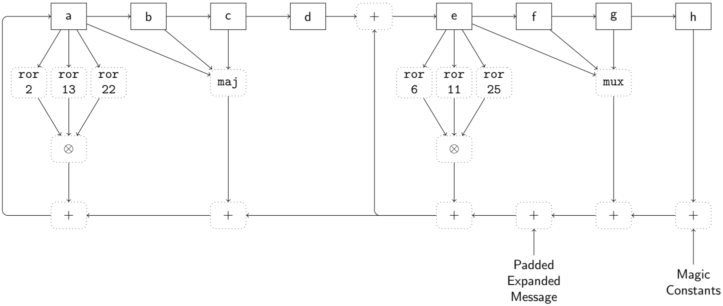
Unrolling

```
demo3 times = pow 8
 where
  pow 0 x = const 0
  pow 1 x = x
  pow n x = <[ ~~times
                   ~~(pow (n/2) x)
                   ~~(pow (if n `mod` 2 == 0
                           then n/2
                           else n/2+1) x) ]>
```



Slogan: “Recursion *outside* the brackets means repetitive structure.”

The SHA-256 Algorithm



The SHA-256 Algorithm. Each solid rectangle is a 32-bit state variable; the path into each rectangle computes its value in the next round based on the values of the state variables in the previous round. The standard specifies initialization values for the state variables prior to the first message block.

Necessary Hardware Primitives

```
class BitSerialHardwarePrimitives g where
  type Wire

  high     :: <[                  ()    ~~> Wire ]>@g
  low      :: <[                  ()    ~~> Wire ]>@g

  not      :: <[             Wire,()    ~~> Wire ]>@g
  xor      :: <[       Wire,(Wire,())   ~~> Wire ]>@g
  or       :: <[       Wire,(Wire,())   ~~> Wire ]>@g
  and      :: <[       Wire,(Wire,())   ~~> Wire ]>@g
  mux2     :: <[ Wire,(Wire,(Wire,()))  ~~> Wire ]>@g
  maj3     :: <[ Wire,(Wire,(Wire,()))  ~~> Wire ]>@g
  reg      :: <[             Wire,()    ~~> Wire ]>@g

  repeat   ::        [Bool] -> <[    () ~~> Wire ]>@g
  fifo     ::        Int -> <[  Wire,() ~~> Wire ]>@g

  probe    ::        Int -> <[ Wire,()  ~~> Wire ]>@g
  oracle   ::        Int -> <[       () ~~> Wire ]>@g
```

A Bit-Serial Adder

```
xor3 = <[ \x y z -> xor (xor x y) z ]>

adder =
  <[ \in1 in2 ->
     let firstBit  = ~~(repeat [ i/=0 | i<-[0..31] ])
         carry_out = reg (mux2 firstBit
                               zero
                               carry_in)
         carry_in  = maj3 carry_out in1 in2
     in  xor3 carry_out in1 in2
   ]>
```

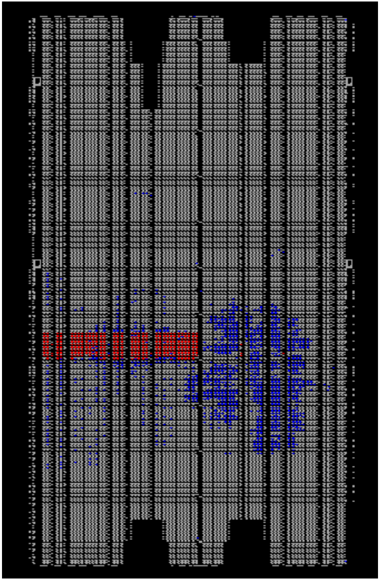
A Bit-Serial Right Rotator

```
rotRight n =
  <[ \input ->
     let sel   = ~~(repeat [ i >= 32-n | i<-[0..31] ])
         fifo1 = ~~(fifo (32-n)) input
         fifo2 = ~~(fifo  32   ) fifo1
     in  mux2 sel fifo1 fifo2
   ]>
```

One Round of SHA-256

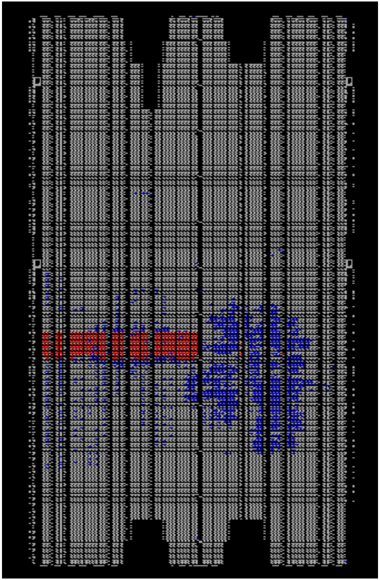
```
sha256round =
  <[ \load input k_plus_w ->
     let a    = ~~(fifo 32) (mux2 load a_in input)
         b    = ~~(fifo 32) a
         c    = ~~(fifo 32) b
         d    = ~~(fifo 32) c
         e    = ~~(fifo 32) (mux2 load e_in d)
         f    = ~~(fifo 32) e
         g    = ~~(fifo 32) f
         h    = ~~(fifo 32) g
         s0   = xor3 (~~(rotRight  2) a_in)
                     (~~(rotRight 13) a_in)
                     (~~(rotRight 22) a_in)
         s1   = xor3 (~~(rotRight  6) e_in)
                     (~~(rotRight 11) e_in)
                     (~~(rotRight 25) e_in)
         a_in = adder t1 t2
         e_in = adder t1 d
         t1   = adder
                    (adder h s1)
                    (adder (mux2 e g f) k_plus_w)
         t2   = adder s0 (maj3 a b c)
     in h ]>
```

32 Copies of the Circuit



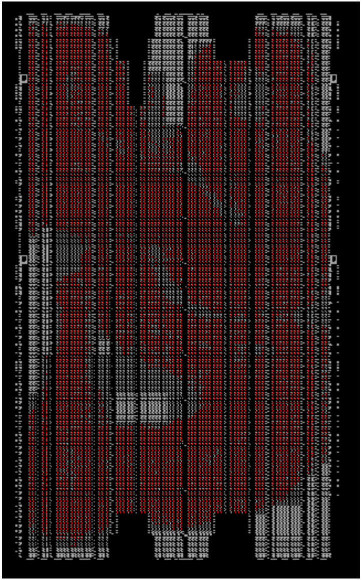
The region in red holds 32 copies of the SHA-256 circuit. Slices shown in blue are the “overhead” shared among all copies (address generators, etc).

Performance



- Bit-serial adder's carry chain is a "wire through time" rather than a "wire through space."
- With extra pipeline registers, 350mhz is possible on a Spartan-6 with no manual placement constraints.
- The leading open-source *bit-parallel* SHA-256 core needs manual placement constraints to reach 180Mhz.

Performance: Disappointing



Unfortunately, the bit-parallel design (shown here) still gives more throughput than a device full of bit-serial hashers.

Silver lining: the bit-parallel design is unable to use more than half the device. Many independent copies of the bit-serial design can be used to "fill in the gaps" left behind, making use of otherwise-idle area.

Questions?

http://www.cs.berkeley.edu/~megacz/garrows/

Extra Slides Follow