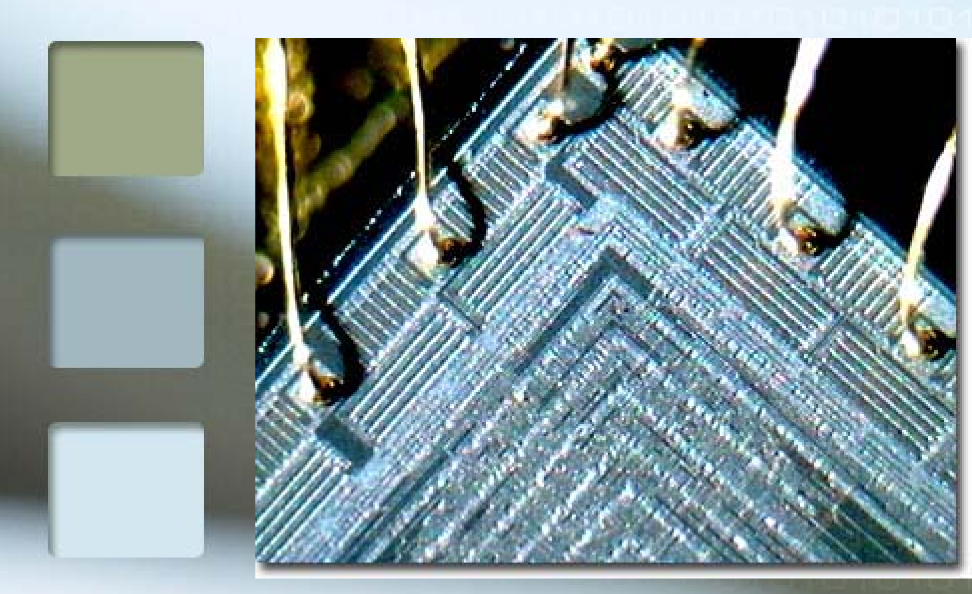
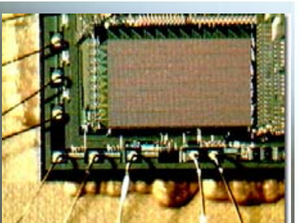


Concurrent Programming with Join Patterns via STMs

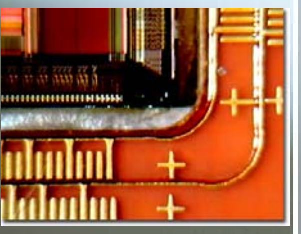
Satnam Singh, Microsoft

Overview



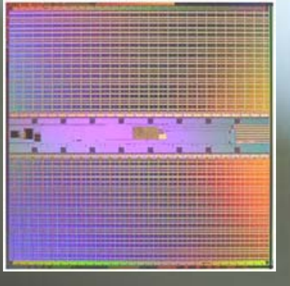
- Nothing clever.
- Bait and switch operation.
- Join patterns in Comega
- Joins encoded with STMs:
  - Synchronous and asynchronous joins.
  - Choice.
  - Dynamic Joins.

Comega asynchronous methods



```
using System ;

public class MainProgram
{ public class ArraySummer
  { public async sumArray (int[] intArray)
    { int sum = 0 ;
      foreach (int value in intArray)
        sum += value ;
      Console.WriteLine ("Sum  = " + sum) ;
    }
  }

  static void Main()
  { Summer = new ArraySummer () ;
    Summer.sumArray (new int[] {1, 0, 6, 6, 1, 9, 6, 6}) ;
    Summer.sumArray (new int[] {3, 1, 4, 1, 5, 9, 2, 6}) ;
    Console.WriteLine ("Main method done.") ;
  }
}
```

Comega chords

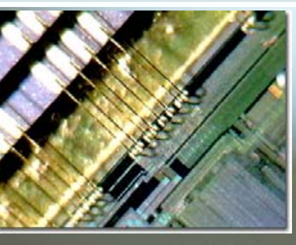


```
using System ;

public class MainProgram
{ public class Buffer
  { public async Put (int value) ;
    public int Get () & Put(int value)
    { return value ; }
  }

  static void Main()
  { buf = new Buffer () ;
    buf.Put (42) ;
    buf.Put (66) ;
    Console.WriteLine (buf.Get() + " " +
 buf.Get()) ;
  }
}
```

“STM”s in Haskell

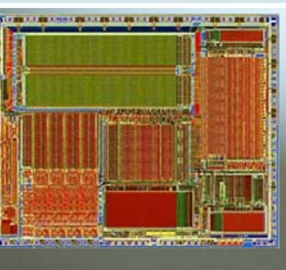
```
data STM a
instance Monad STM
-- Monads support "do" notation and sequencing
-- Exceptions
throw :: Exception -> STM a
catch :: STM a -> (Exception->STM a) -> STM a
-- Running STM computations
atomically :: STM a -> IO a
retry :: STM a
orElse :: STM a -> STM a -> STM a
-- Transactional variables
data TVar a
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Join2



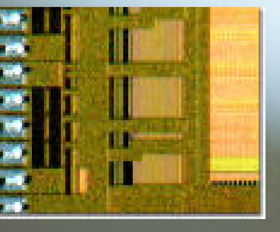
```
module Main
where

import Control.Concurrent
import Control.Concurrent.STM

join2 :: TChan a -> TChan b -> IO (a, b)
join2 chanA chanB
  = atomically (do a <- readTChan chanA
                   b <- readTChan chanB
                   return (a, b)
               )

taskA :: TChan Int -> TChan Int -> IO ()
taskA chan1 chan2
  = do (v1, v2) <- join2 chan1 chan2
       putStrLn ("taskA got: " ++ show (v1, v2))

main
  = do chanA <- atomically newTChan
       chanB <- atomically newTChan
       atomically (writeTChan chanA 42)
       atomically (writeTChan chanB 75)
       taskA chanA chanB
```

One-Shot Synchronous Join



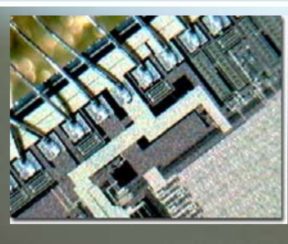
```
(&) :: TChan a -> TChan b -> STM (a, b)
(&) chan1 chan2
  = do a <- readTChan chan1
       b <- readTChan chan2
       return (a, b)

(>>>) :: STM a -> (a -> IO b) -> IO b
(>>>) joinPattern handler
   = do results <- atomically joinPattern
        handler results

example chan1 chan2
  = chan1 & chan2 >>>
    \ (a, b) -> putStrLn (show (a, b))
```

Puzzle



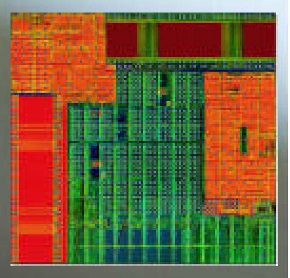
```
main :: IO ()
main
  = do chan1 <- atomically $ newTChan
       atomically $ writeTChan chan1 42
       atomically $ writeTChan chan1 74
       chan1 & chan1 >>>
         \ (a, b) -> putStrLn (show (a,b))
```

Repeating Asynchronous Join



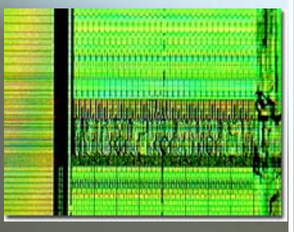
```
(>!>) :: STM a -> (a -> IO ()) -> IO ()
(>!>) joins cont
   = do forkIO (asyncJoinLoop joins cont)
        return () -- discard thread ID

asyncJoinLoop :: (STM a) -> (a -> IO ()) -> IO ()
asyncJoinLoop joinPattern handler
  = do joinPattern >>> forkIO . handler
       asyncJoinLoop joinPattern handler

example chan1 chan2
  = chan1 & chan2 >!>
    \ (a, b) -> putStrLn (show ((a, b)))
```

Exploiting Overloading



```
class Joinable t1 t2 where
  (&) :: t1 a -> t2 b -> STM (a, b)

instance Joinable TChan TChan where
  (&) = join2

instance Joinable TChan STM where
  (&) = join2b

instance Joinable STM TChan where
  (&) a b = do (x,y) <- join2b b a
               return (y, x)

chan1 & chan2 & chan3 >>>
\ ((a, b), c) -> putStrLn (show (a,b,c))
```

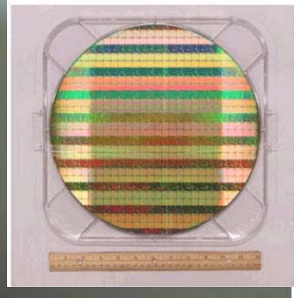
Biased Synchronous Choice



```
(|+|) :: (STM a, a -> IO c) ->
         (STM b, b -> IO c) ->
         IO c
(|+|) (joina, action1) (joinb, action2)
  = do io <- atomically
               (do a <- joina
                   return (action1 a)
               `orElse`
                do b <- joinb
                   return (action2 b))
       io
```

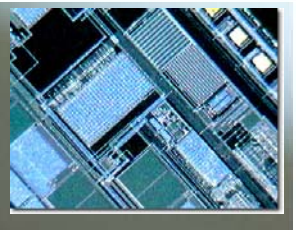
```
  (chan1 & chan2 & chan3,
     \ ((a,b),c) -> putStrLn (show (a,b,c)))
|+|
  (chan1 & chan2,
     \ (a,b) -> putStrLn (show (a,b)))
```

Dynamic Joins



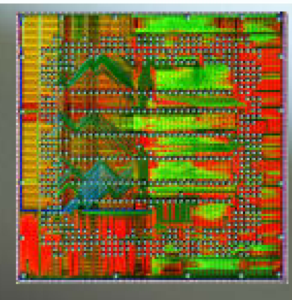
```
example numSensors numSensors chan1 chan2 chan3
  = if numSensors = 2 then
       chan1 & chan2 >!> \ (a, b) ->
       putStrLn (show ((a, b)))
    else
       chan1 & chan2 & chan3 >!> \ (a, (b, c))
         -> putStrLn (show ((a, b, c)))
```

Conditional Joins



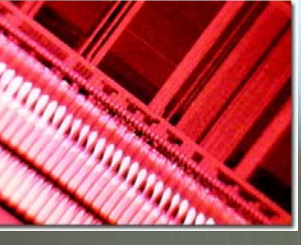
```
(??) :: TChan a -> (a -> Bool) -> STM a
(??) chan predicate
  = do value <- readTChan chan
       if predicate value then
          return value
         else
          retry

(chan1 ?? \x -> x > 3) & chan2 >>>
  \ (a, b) -> putStrLn (show (a, b))
```

Summary and Questions



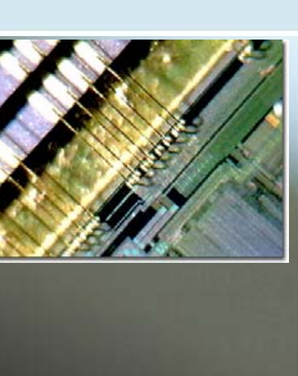
- "Free" joins encoded nicely in terms of STMs.
- Model for understanding join patterns in terms of STMs.
- A good literal implementation (?)
  - Parallel execution?
- Joins as statements instead of declarations.
- Other work: JSR-166 library
- What are joins good for anyway?

Conditional Joins



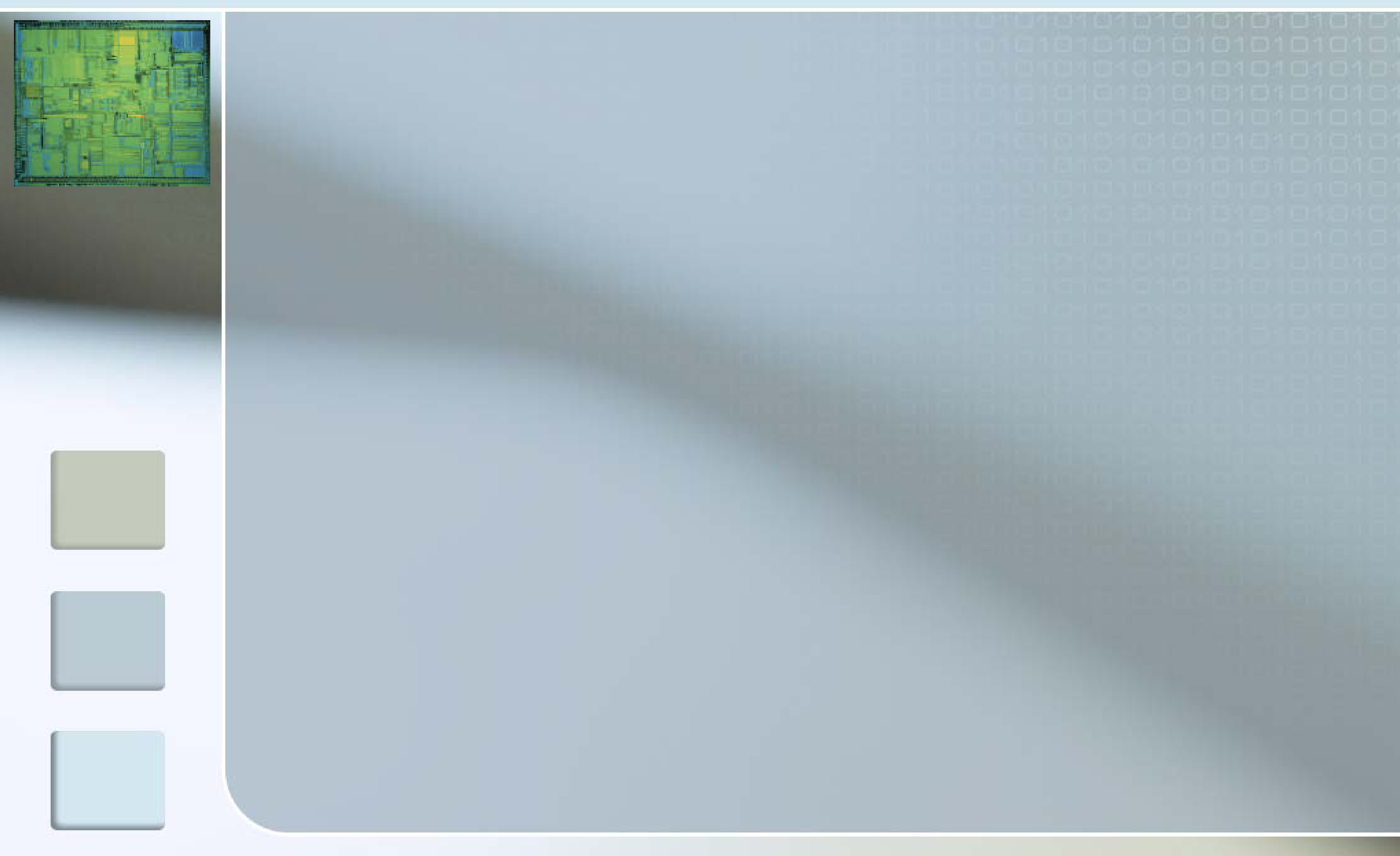
```
(?) :: TChan a -> Bool -> STM a
(?) chan predicate
  = if predicate then
      readTChan chan
    else
      retry

(chan1 ? cond) & chan2 >>>
\ (a, b) -> putStrLn (show (a, b))
```

Conditional Joins



```haskell
(?) :: TChan a -> STM Bool -> STM a
(?) chan predicate
  = do cond <- predicate
       if cond then
          readTChan chan
        else
          retry
```

Backup



Backup