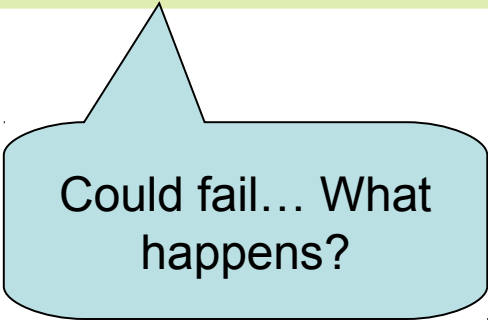


# Laziness and Parallelism

# A Function

```
fun :: Maybe Int -> Int
fun mx | mx == Nothing = 0
      | otherwise      = x + 3
where
  x = fromJust mx
```



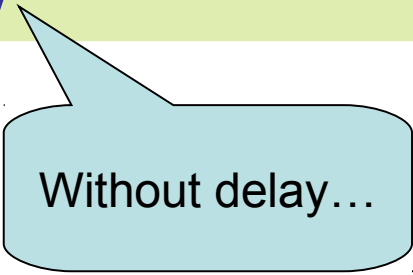
Could fail... What happens?

# Another Function

```
slow :: Integer -> Integer
slow n | n <= 1      = 1
        | otherwise = slow (n-1) + slow (n-2)
```

```
if' :: Bool -> a -> a -> a
if' False x y = x
if' True  x y = y
```

```
Main> if' False 17 (slow 99)
17
```



Without delay...

# Laziness

Haskell is a *lazy* language

- Things are evaluated *at most once*
- Things are only evaluated when they are needed
- Things are never evaluated twice

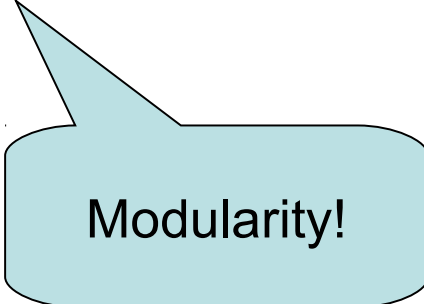
# Understanding Laziness

Use **error** or **undefined** to see whether something is evaluated or not

- if' False 17 undefined
- head [3,undefined,17]
- head (3:4:undefined)
- head [undefined,17,13]
- head undefined

# Lazy Programming Style

- Separate
  - Where the computation of a value is defined
  - Where the computation of a value happens



Modularity!

# Lazy Programming Style

- `head [1..1000000]`
- `zip "abc" [1..9999]`
- `take 10 ['a'..'z']`
- ...

# When is a Value "Needed"?

```
strange :: Bool -> Integer  
strange False = 17  
strange True  = 17
```

```
Main> strange undefined  
Exception: undefined
```

- An argument is evaluated when it is examined by pattern matching (and the result of match is needed)
  - Is the result of `strange` needed?
  - Yes, because `GHCi` wants to print it
- Primitive functions (e.g. `(+)`, `div`, etc.) evaluate their arguments (if their result is needed)



# And?

```
(&&) :: Bool -> Bool -> Bool
True  && True  = True
False && True  = False
True  && False = False
False && False = False
```

Is this a good definition?

No – evaluates more than necessary

# And and Or

```
(&&) :: Bool -> Bool -> Bool  
True  && x = x  
False && x = False
```

```
(||) :: Bool -> Bool -> Bool  
True  || x = True  
False || x = x
```

```
Main> 1+1 == 3 && slow 99 == slow 99  
False
```

```
Main> 2*2 == 4 || undefined  
True
```

# Laziness

Haskell is a *lazy* language

- Things are evaluated *at most once*
- Things are only evaluated when they are needed
- Things are never evaluated twice



“Things”  $\approx$  variables and constants

# At Most Once?

Quiz: How to avoid recomputation?

```
apa :: Integer -> Integer  
apa x = f x + f x
```

f x is  
evaluated  
twice

```
bepa :: Integer -> Integer -> Integer  
bepa x y = f 17 + x + y
```

```
Main> bepa 1 2 + bepa 3 4  
310
```

f 17 is  
evaluated  
twice

# At Most Once!

```
apa :: Integer -> Integer  
apa x = fx + fx  
  where  
    fx = f x
```

# Example: BouncingBalls

Ball represented by  
all the points in its  
life time

Good idea?

Computing a ball  
might take long...

```
type Ball = [Point]
```

```
bounce :: Point -> Int -> Ball
```

```
bounce (x,y) v
```

```
  | v == 0 && y >= maxY = replicate 20 (x,y)
```

```
  | y' > maxY           = bounce (x,y) (2-v)
```

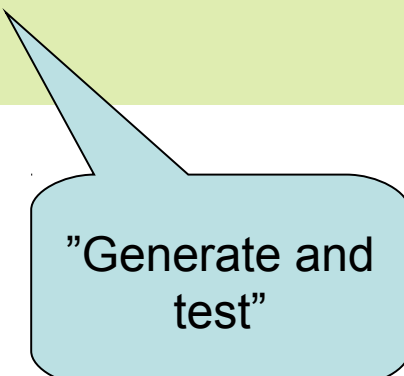
```
  | otherwise           = (x,y) : bounce (x,y') (v+1)
```

```
where
```

```
  y' = y + fromIntegral v
```

# Example: Sudoku

```
solve :: Sudoku -> Maybe Sudoku
solve sud
  | ...
  | otherwise =
    listToMaybe
      [ sol
      | n <- [1..9]
      , ... solve (update p (Just n) sud) ...
      ]
```

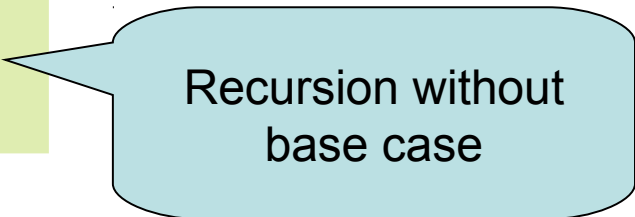


"Generate and  
test"

# Infinite Lists

- Because of laziness, values in Haskell can be *infinite*
- Impossible to compute them completely!
- Instead, only use parts of them

```
ones :: [Integer]
ones = 1 : ones
```



Recursion without  
base case

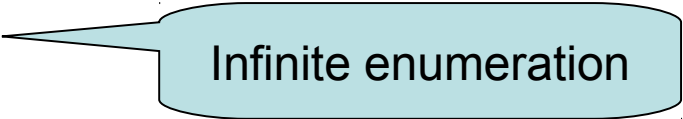
```
Main> take 10 ones
[1,1,1,1,1,1,1,1,1,1]
```



# Examples

## Uses of infinite lists

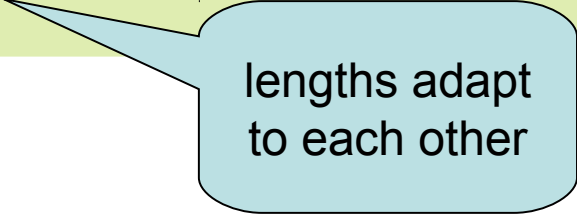
- take n [3..]
- xs `zip` [1..]



Infinite enumeration

# Example: PrintTable

```
printTable :: [String] -> IO ()
printTable xs =
    sequence_ [ putStrLn (show i ++ ": " ++ x)
               | (x,i) <- xs `zip` [1..]
               ]
```



lengths adapt  
to each other

```
Main> printTable ["Häst", "Får", "Snigel"]
1: Häst
2: Får
3: Snigel
```

# Iterate

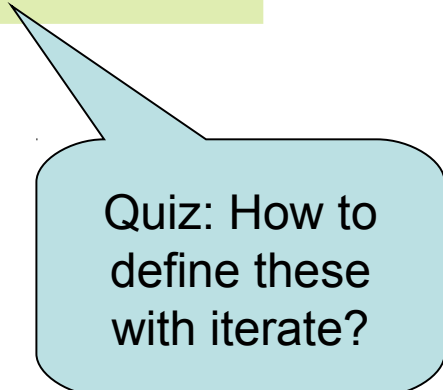
```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
  -- iterate f x = [x, f x, f (f x), f (f (f x)), ...]
```

```
Main> iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512,1024,...]
```

# Other Handy Functions

```
repeat :: a -> [a]
repeat x = x : repeat x

cycle :: [a] -> [a]
cycle xs = xs ++ cycle xs
```



Quiz: How to  
define these  
with iterate?

# Alternative Definitions

```
repeat :: a -> [a]  
repeat x = iterate id x
```

```
cycle :: [a] -> [a]  
cycle xs = concat (repeat xs)
```

# Problem: Replicate

```
replicate :: Int -> a -> [a]  
replicate = ?
```

```
Main> replicate 5 'a'  
"aaaaa"
```

# Problem: Replicate

```
replicate :: Int -> a -> [a]  
replicate n x = take n (repeat x)
```

# Problem: Grouping List Elements

```
group :: Int -> [a] -> [[a]]  
group = ?
```

```
Main> group 3 "apabepacepa!"  
["apa","bep","ace","pa!"]
```



# Problem: Grouping List Elements

```
group :: Int -> [a] -> [[a]]
group n = takeWhile (not . null)
         . map (take n)
         . iterate (drop n)
```

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

# Problem: Prime Numbers

```
primes :: [Integer]
primes = ?
```

```
Main> take 4 primes
[2,3,5,7]
```

# Problem: Prime Numbers

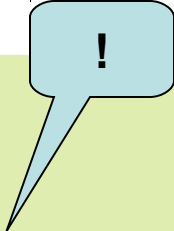
```
primes :: [Integer]
primes = 2 : [ x | x <- [3,5..], isPrime x ]
where
  isPrime x =
    all (not . (`divides` x))
      (takeWhile (\y -> y*y <= x) primes)
```

```
all :: (a -> Bool) -> [a] -> Bool
```

# Infinite animations

Remove friction in Bouncing Balls:

```
bounce :: Point -> Int -> Ball
bounce (x,y) v
  | v == 0 && y >= maxY = replicate 20 (x,y)
  | y' > maxY           = bounce (w,h) (x,y) (0-v)
  | otherwise           = (x,y) : bounce (x,y') (v+1)
where
  y' = y + fromIntegral v
```



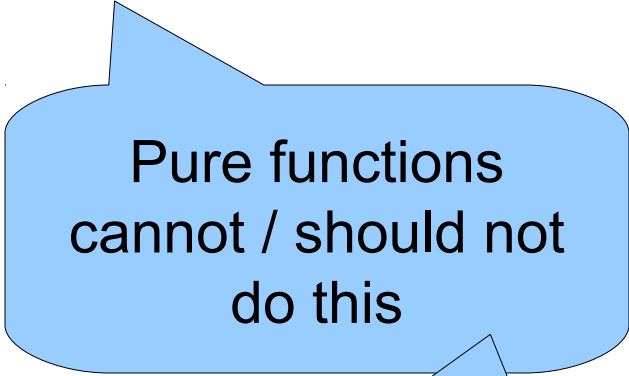
- Ball never stops
- New points produced whenever the animation function needs them

# Laziness: Summing Up

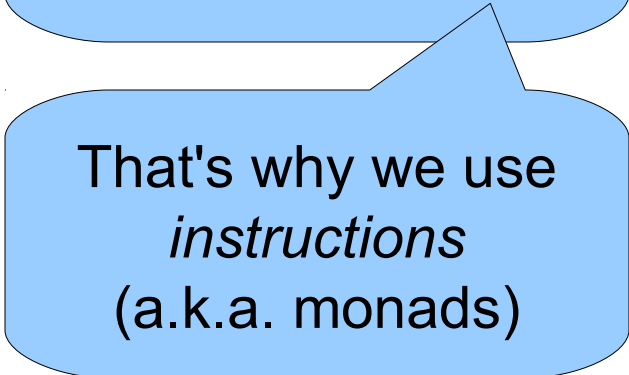
- Laziness
  - Evaluated at most once
  - Programming style
- Do not have to use it
  - But powerful tool!
- Can make programs more “modular”
  - E.g. separate bounce function from drawing in Bouncing Balls

# Side-Effects

- Writing to a file
- Reading from a file
- Creating a window
- Waiting for the user to click a button
- ...
- Changing the value of a variable



Pure functions  
cannot / should not  
do this



That's why we use  
*instructions*  
(a.k.a. monads)

# Some Haskell History

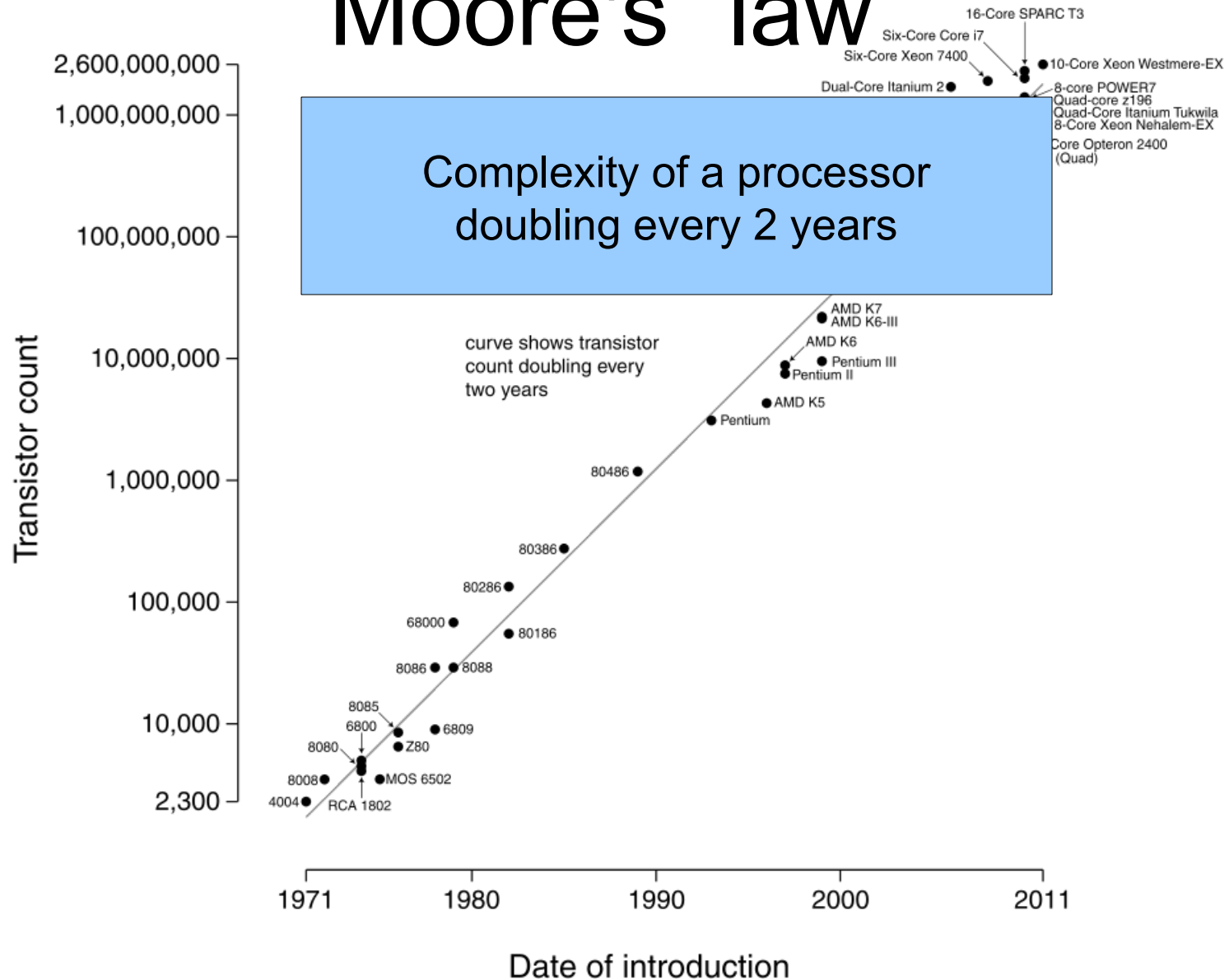
- A primary design goal of Haskell was to be a *lazy* functional programming language
- Lazy programs:
  - Values computed on-demand
  - Compiler chooses the order
- Uncontrolled ordering does not mix with side effects!
  - ... so Haskell had to be a pure language
- See: [A History of Haskell](#)  
(P Hudak, J Hughes, SP Jones, P Wadler – 2007)

# Parallelism

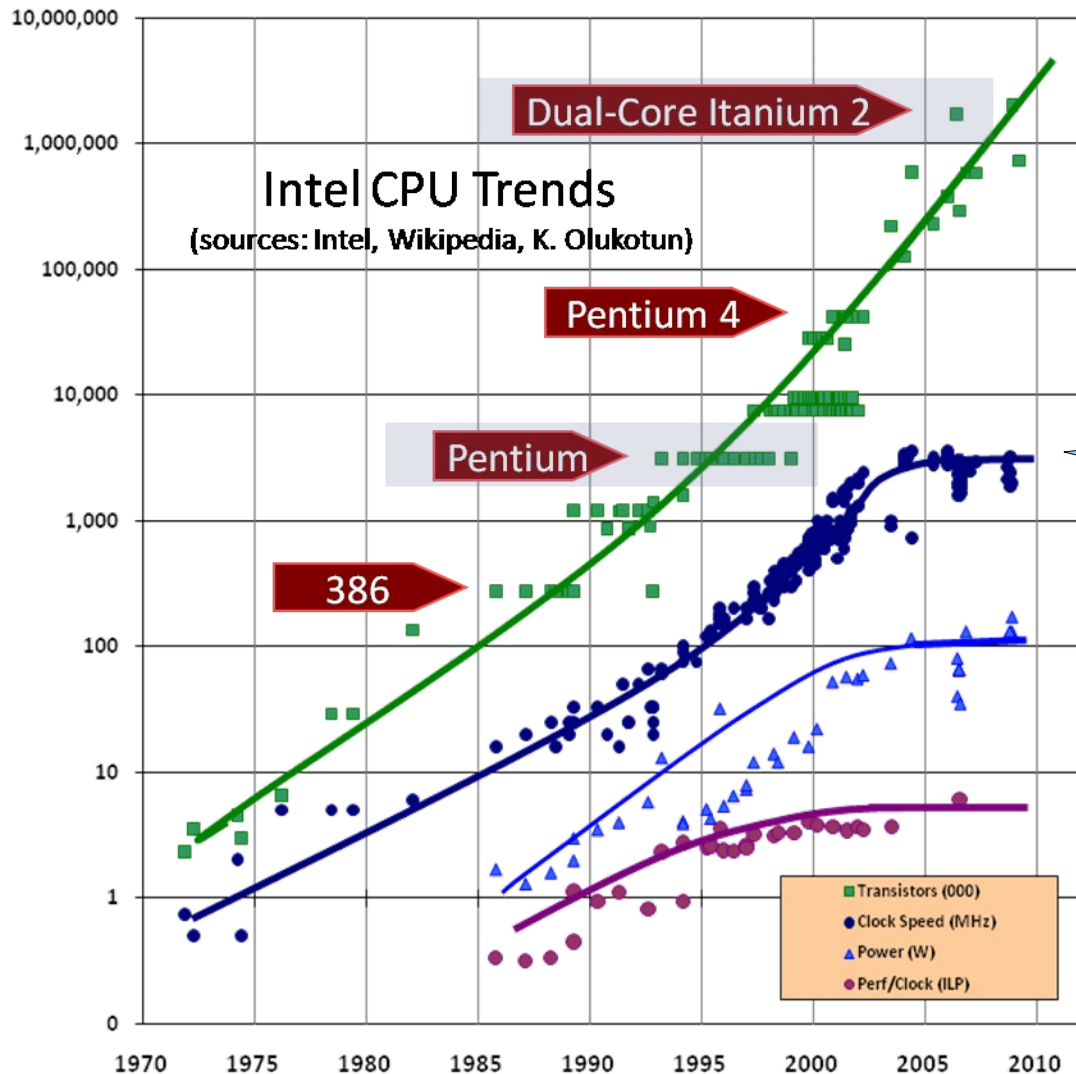


## Microprocessor Transistor Counts 1971-2011 & Moore's Law

# Moore's "law"



# More Moore

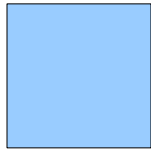


Clock speed no longer grows exponentially

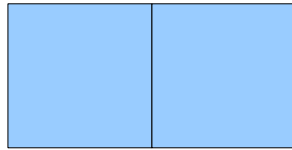
Sequential code does not get faster

Solution: multicore!

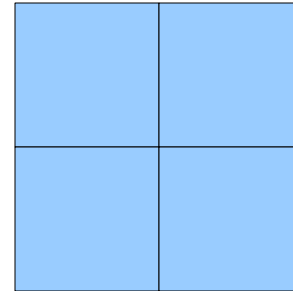
# Processors Today and Tomorrow



single  
core

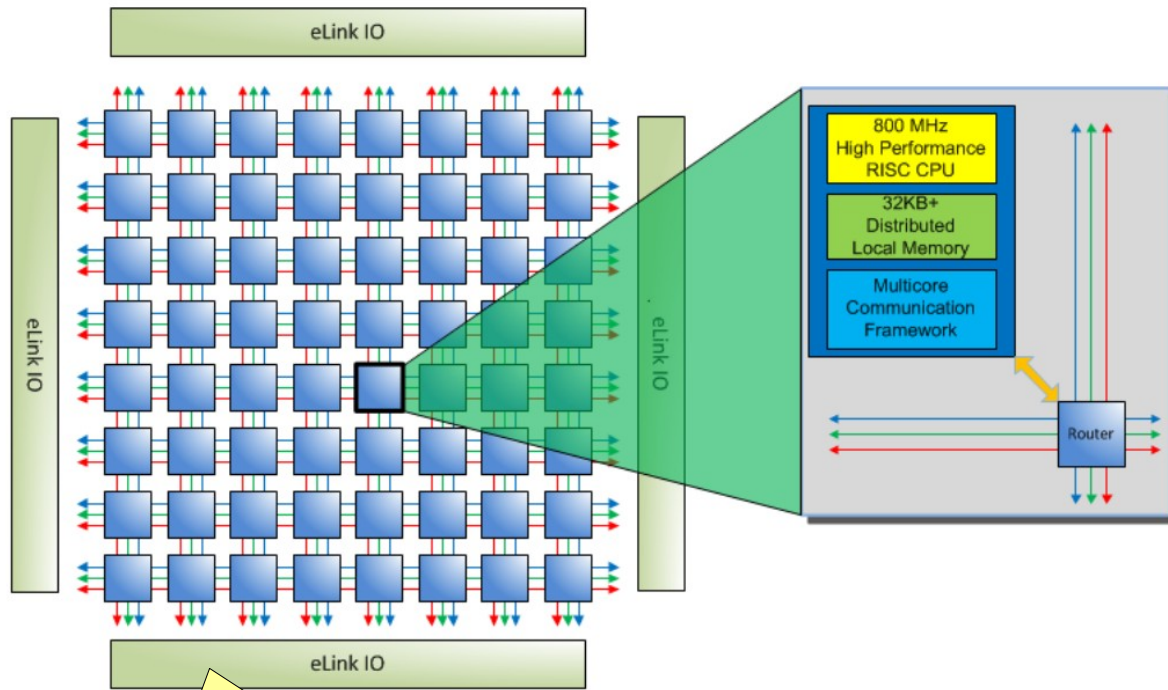


dual core



quadcore

# Processors Today and Tomorrow



Adapteva 64 cores  
(Architecture supports  
up to 4096 cores)

How to program  
these?

# Parallelism

- Previously, computation went one step at a time
- Now, we can (and have to) do *many things at the same time, “in parallel”*
- Side effects and parallelism do not mix well: *race conditions*
  - Think: Many people cooking in the same kitchen

# Basic parallelism in Haskell

**import Control.Parallel**

`pseq :: a -> b -> b`

`pseq x y`:  
“first evaluate `x`, then  
produce `y` as a result”

`par :: a -> b -> b`

`par x y`:  
“produce `y` as a result,  
but also evaluate `x`  
*in parallel*”

Safe, because `x` has  
no side effects

# Parallelism in Haskell

```
parList :: [a] -> b -> b
parList []      y = y
parList (x:xs) y = x `par` (xs `parList` y)
```

```
pmap :: (a -> b) -> [a] -> [b]
pmap f xs = ys `parList` ys
  where
    ys = map f xs
```

(understand the result:  
remove all the pars)

# Parallelism in Haskell (2)

```
data Expr = Num Int
          | Add Expr Expr
```

```
peval :: Expr -> Int
peval (Num n)      = n
peval (Add a b) = x `par` y `par` x+y
  where
    x = peval a
    y = peval b
```

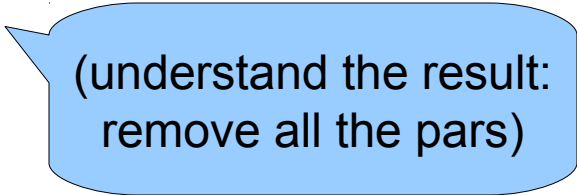


# Parallelism in Haskell (3)

```
testPar = print $ sum $  
    pmap slow (concat $ replicate 5 [25..30])
```

# Pure Functions...

- ...enable easier *understanding*
  - only the arguments affect the result
- ...enable easier *testing*
  - stimulate a function by providing arguments
- ...enable *laziness*
  - powerful programming tool
- ...enable easy *parallelism*
  - no head-aches because of side effects



(understand the result:  
remove all the pars)

# Do's and Don'ts

Repetitive code  
– hard to see  
what it does...

```
lista :: a -> [a]  
lista x = [x,x,x,x,x,x,x,x,x]
```

```
lista :: a -> [a]  
lista x = replicate 9 x
```

# Do's and Don'ts

```
siffra :: Integer -> String
siffra 1 = "1"
siffra 2 = "2"
siffra 3 = "3"
siffra 4 = "4"
siffra 5 = "5"
siffra 7 = "7"
siffra 8 = "8"
siffra 9 = "9"
siffra _ = "###"
```

Repetitive code  
– hard to see  
what it does...

Is this really  
what we want?

```
siffra :: Integer -> String
siffra x | 1 <= x && x <= 9 = show x
         | otherwise       = "###"
```

# Do's and Don'ts

How much time  
does this take?

```
findIndices :: [Integer] -> [Integer]
findIndices xs = [ i | i <- [0..n], (xs !! i) > 0 ]
  where
    n = length xs-1
```

```
findIndices :: [Integer] -> [Integer]
findIndices xs = [ i | (x,i) <- xs `zip` [0..], x > 0 ]
```