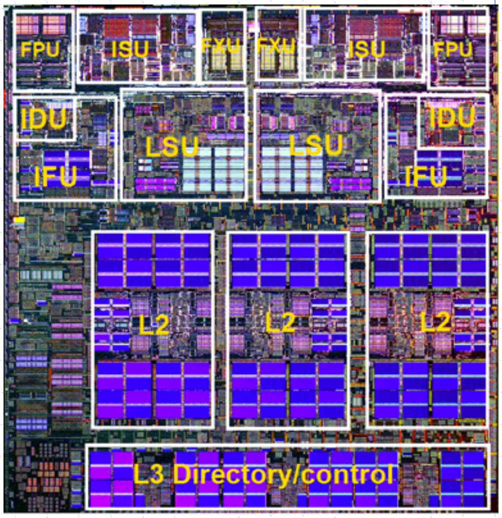
6.827 Multithreaded Parallelism: Languages and Compilers

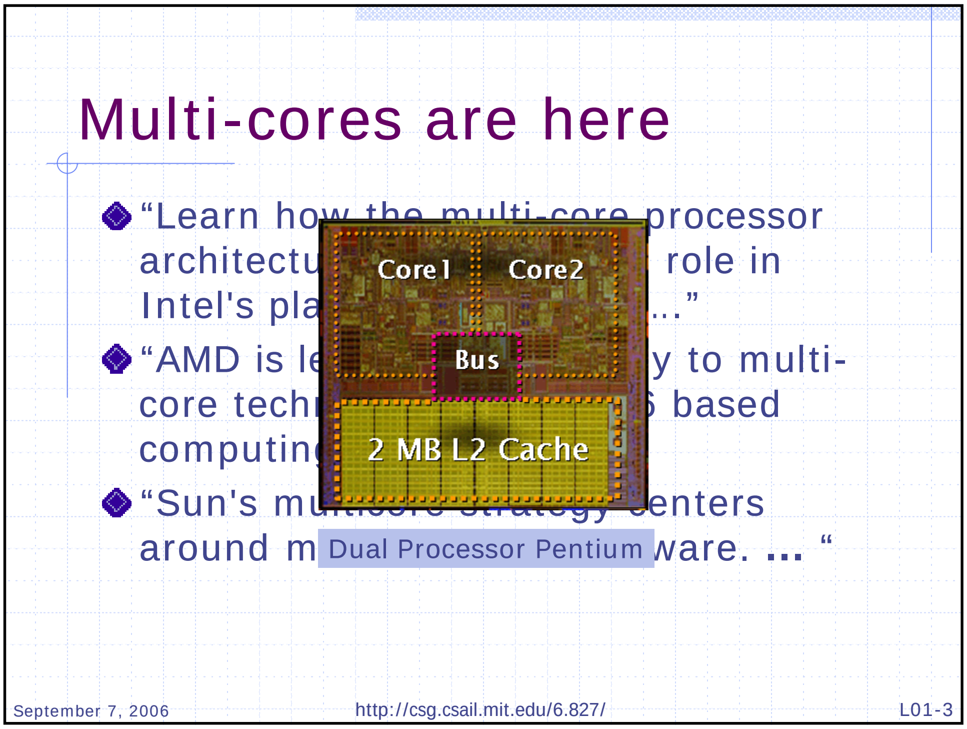
Fall 2006

Lecturer: Arvind
TA: Nirav Dave’
Assistant: Sally Lee

IBM Power 5

- 130nm SOI CMOS with Cu
- 389mm^2
- 2GHz
- 276 million transistors
- Dual processor cores
- 1.92 MB on-chip L2 cache
- 8-way superscalar
- 2-way simultaneous multithreading

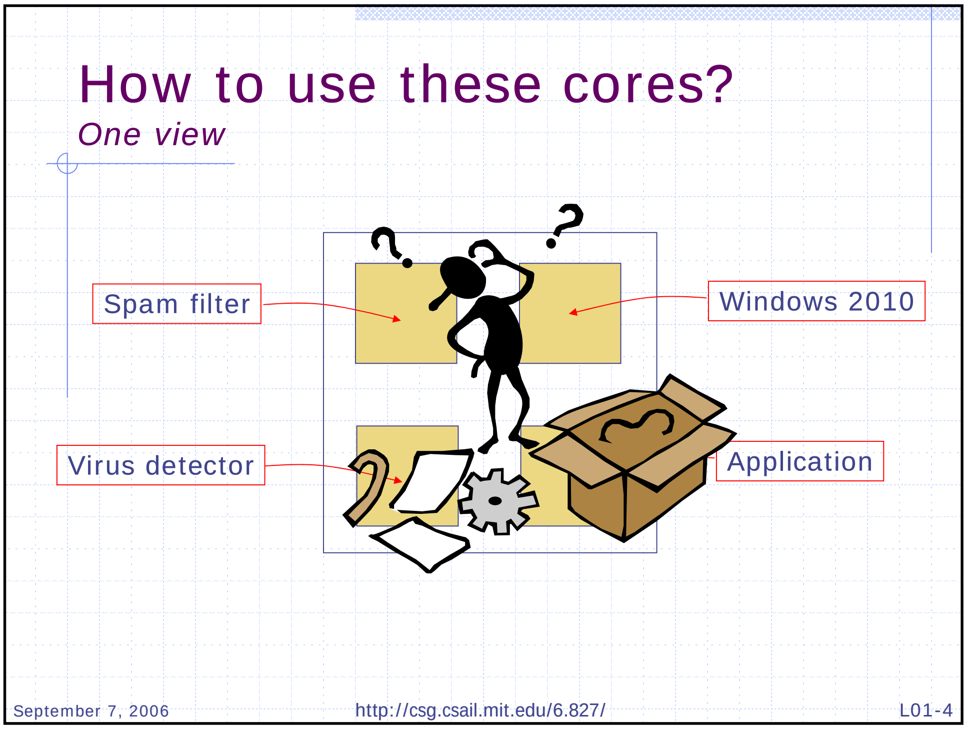




Multi-cores are here

- “Learn how the multi-core processor architectu role in Intel's pla ...”
- “AMD is le y to multi-core techn based computing
- “Sun's mu enters around m ware. ... “

Dual Processor Pentium



How to use these cores?

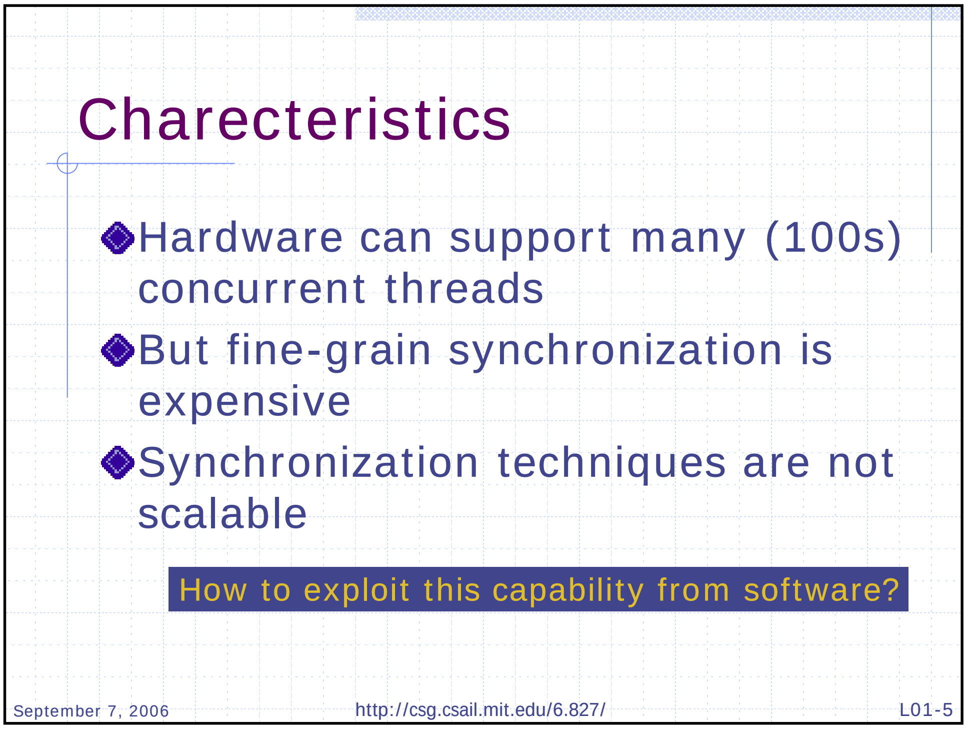
One view

Spam filter

Windows 2010

Virus detector

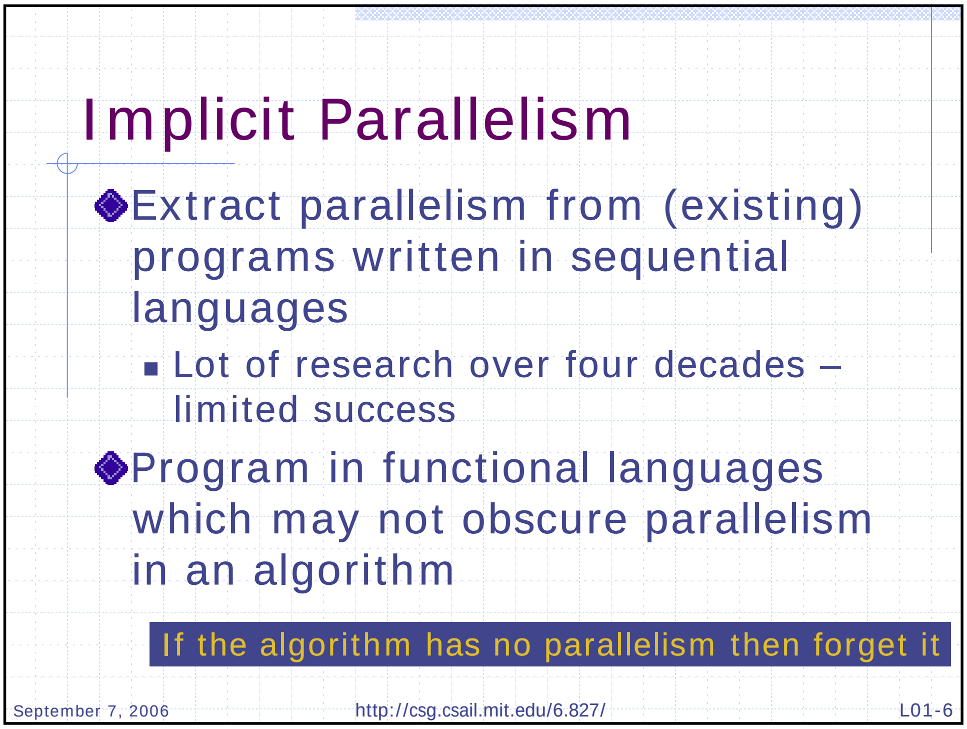
Application



Charecteristics

- Hardware can support many (100s) concurrent threads
- But fine-grain synchronization is expensive
- Synchronization techniques are not scalable

How to exploit this capability from software?

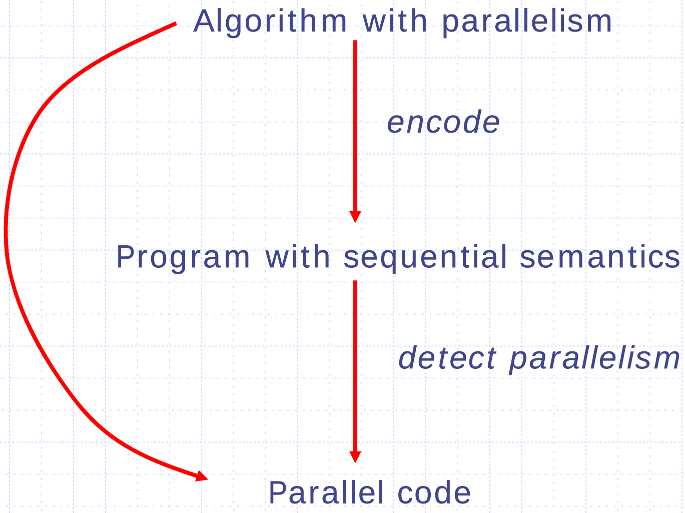


Implicit Parallelism

- Extract parallelism from (existing) programs written in sequential languages
  - Lot of research over four decades – limited success
- Program in functional languages which may not obscure parallelism in an algorithm

If the algorithm has no parallelism then forget it

Why not use sequential languages ?



Algorithm with parallelism

encode

Program with sequential semantics

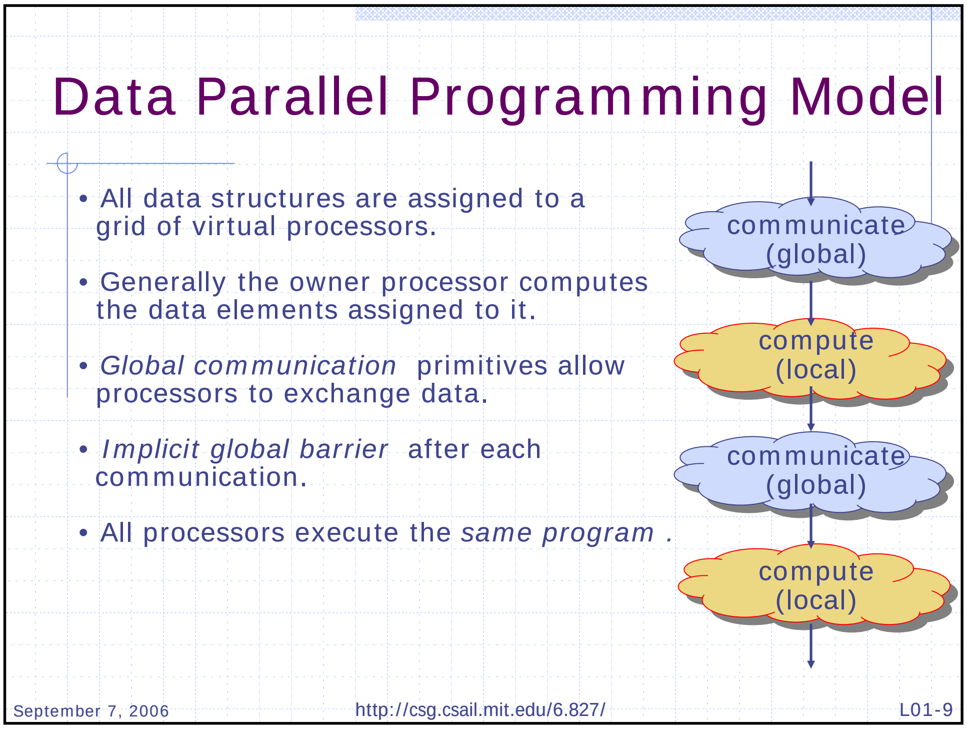
detect parallelism

Parallel code

If parallelism can’t be detected automatically …

Design/use new explicitly parallel programming models …

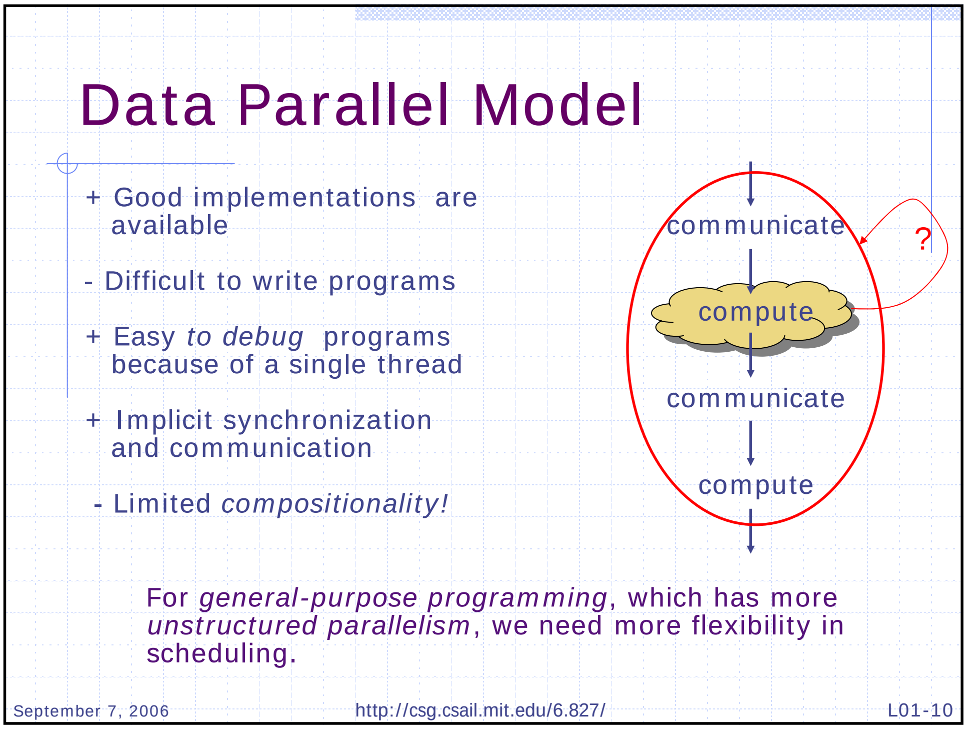
- High-level
  - Data parallel: *Fortran 90, HPF, …*
  - Multithreaded: *Cid, Cilk,…, Java*
    Id, pH, Sisal, …
- Low-level
  - Message passing: *PVM, MPI, …*
  - Threads & synchronization:
    Forks & Joins, Locks, Futures, …



Data Parallel Programming Model

- All data structures are assigned to a grid of virtual processors.
- Generally the owner processor computes the data elements assigned to it.
- *Global communication* primitives allow processors to exchange data.
- *Implicit global barrier* after each communication.
- All processors execute the *same program* .

communicate (global)
compute (local)
communicate (global)
compute (local)



Data Parallel Model

+ Good implementations are available

- Difficult to write programs

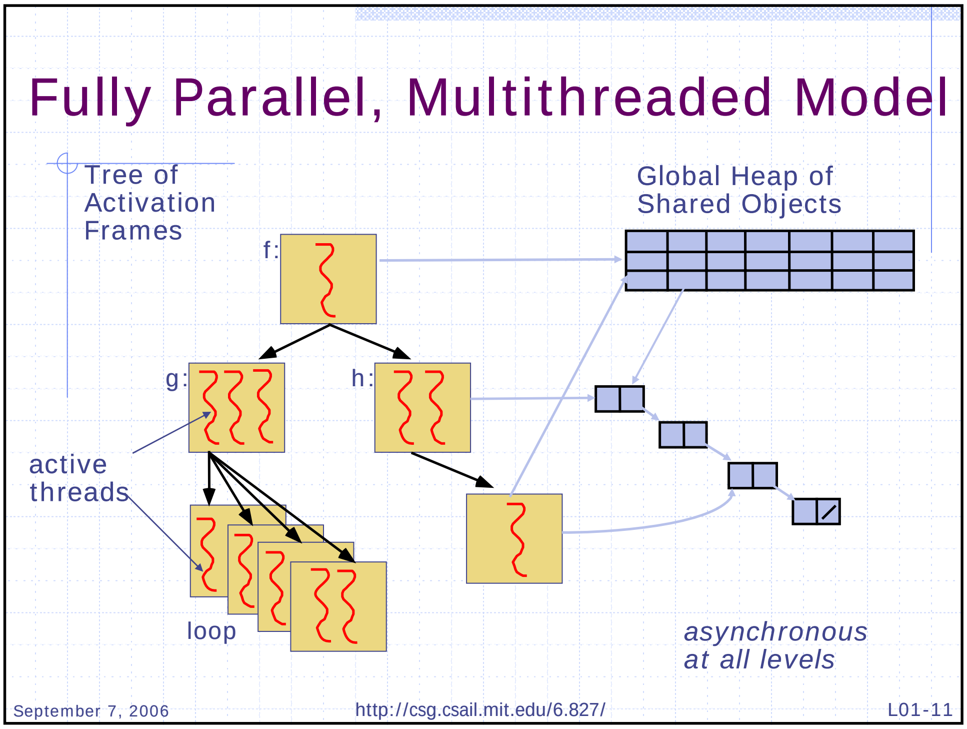
+ Easy *to debug* programs because of a single thread

+ Implicit synchronization and communication

- Limited *compositionality!*

communicate
compute
communicate
compute
?

For *general-purpose programming*, which has more *unstructured parallelism*, we need more flexibility in scheduling.



Fully Parallel, Multithreaded Model

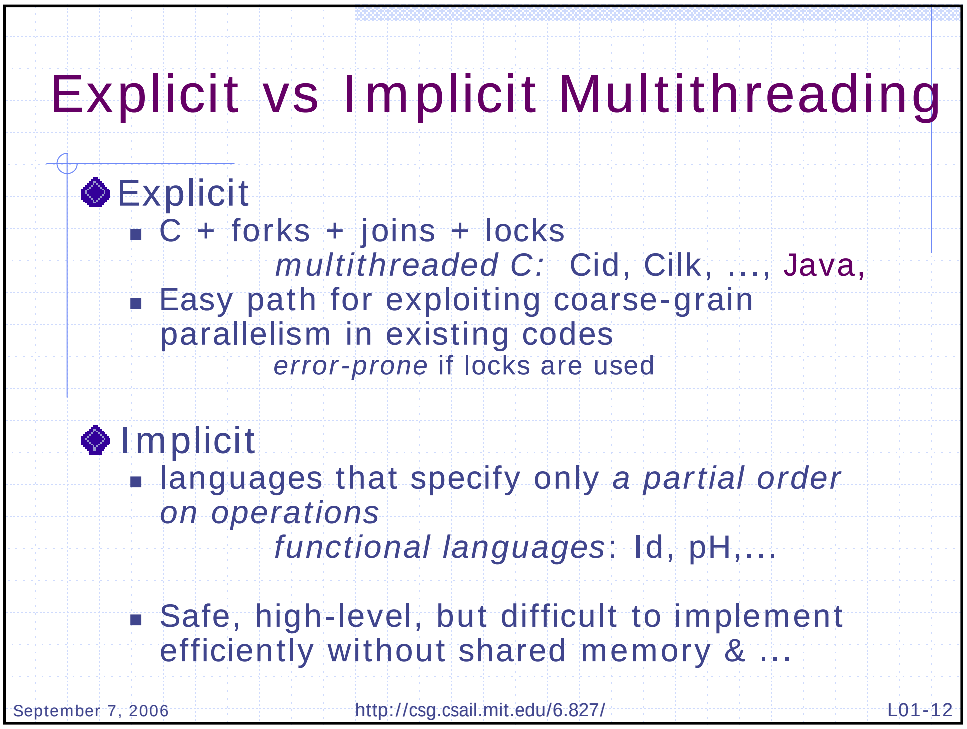
Tree of Activation Frames

Global Heap of Shared Objects

active threads

loop

asynchronous at all levels



Explicit vs Implicit Multithreading

- Explicit
  - C + forks + joins + locks

    multithreaded C: Cid, Cilk, …, Java,
  - Easy path for exploiting coarse-grain parallelism in existing codes

    error-prone if locks are used

- Implicit
  - languages that specify only *a partial order on operations*

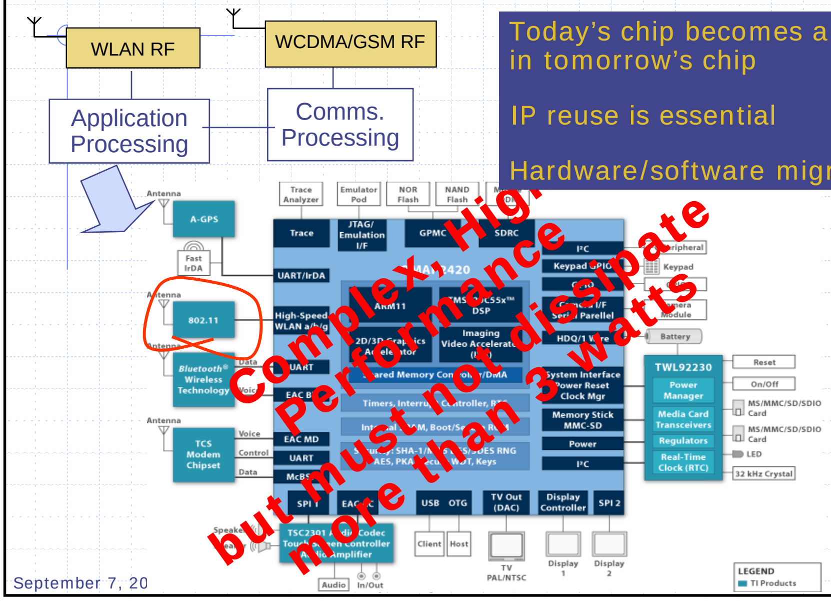
    functional languages: Id, pH,…
  - Safe, high-level, but difficult to implement efficiently without shared memory & …

Only reason for parallel programming used to be performance

- This made programming very difficult
  - Had to know a lot about the machine
  - Codes were not portable – endless performance tuning on each machine
  - Parallel libraries were not composable
  - Difficult to deal with heap structures and memory hierarchy
  - Synchronization costs were too high to exploit fine-grain parallelism

Has the situation changed?

Current Cellphone Architecture



Today's chip becomes a block in tomorrow's chip

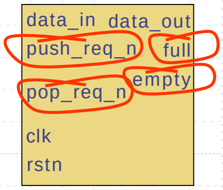
IP reuse is essential

Hardware/software migration

Complex, High Performance but must not dissipate more than 3 watts

IP re-use sounds great until you try it…

Example: Commercially available FIFO IP block



An error occurs if a push is attempted while the FIF l.

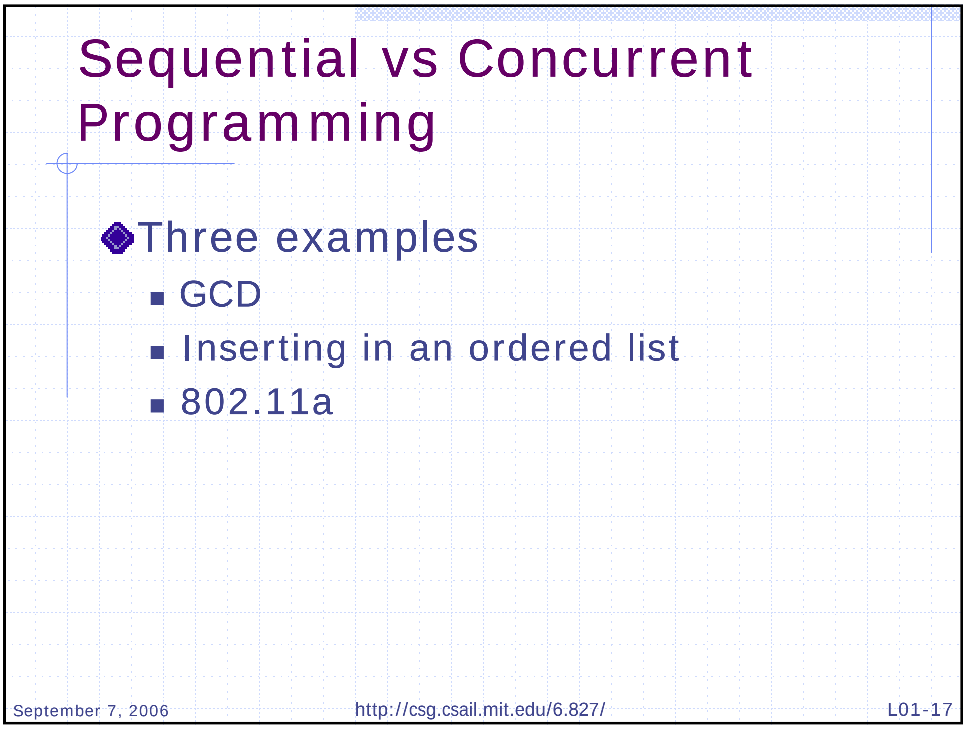
Thus, there is no conflict in a simult when the FIFO is full. A simultaneous push and pop FIFO is empty, since there is no pop data to prefetch. How red in the FIFO.

A pop o pop_req_n is asserted (LOW), as long as the FIFO is not en eq_n causes the internal read pointer to be incremented on the nex clk. Thus, the RAM read data must be captured on the clk following the a on of pop_req_n.

No machine verification of such informal constraints is feasible

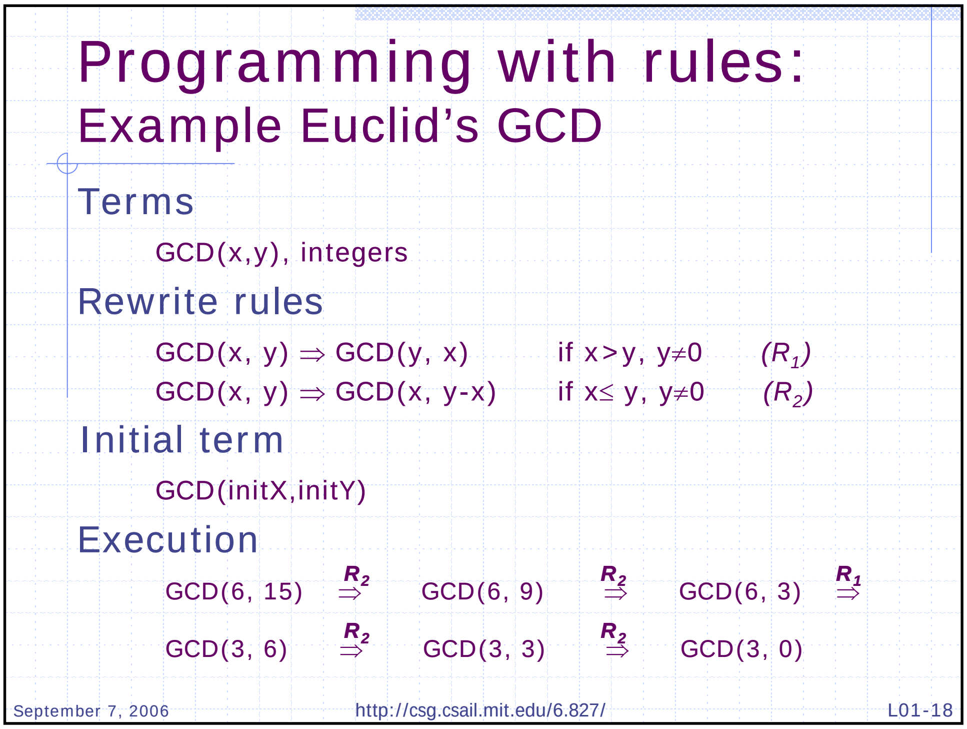
These constraints are spread over many pages of the documentation…

What is needed is a way to assemble (parallel) systems from well designed components



Sequential vs Concurrent Programming

- Three examples
  - GCD
  - Inserting in an ordered list
  - 802.11a



Programming with rules: Example Euclid's GCD

Terms

GCD(x,y), integers

Rewrite rules

GCD(x, y) ⇒ GCD(y, x) if x>y, y≠0 *(R_1)*

GCD(x, y) ⇒ GCD(x, y-x) if x≤ y, y≠0 *(R_2)*

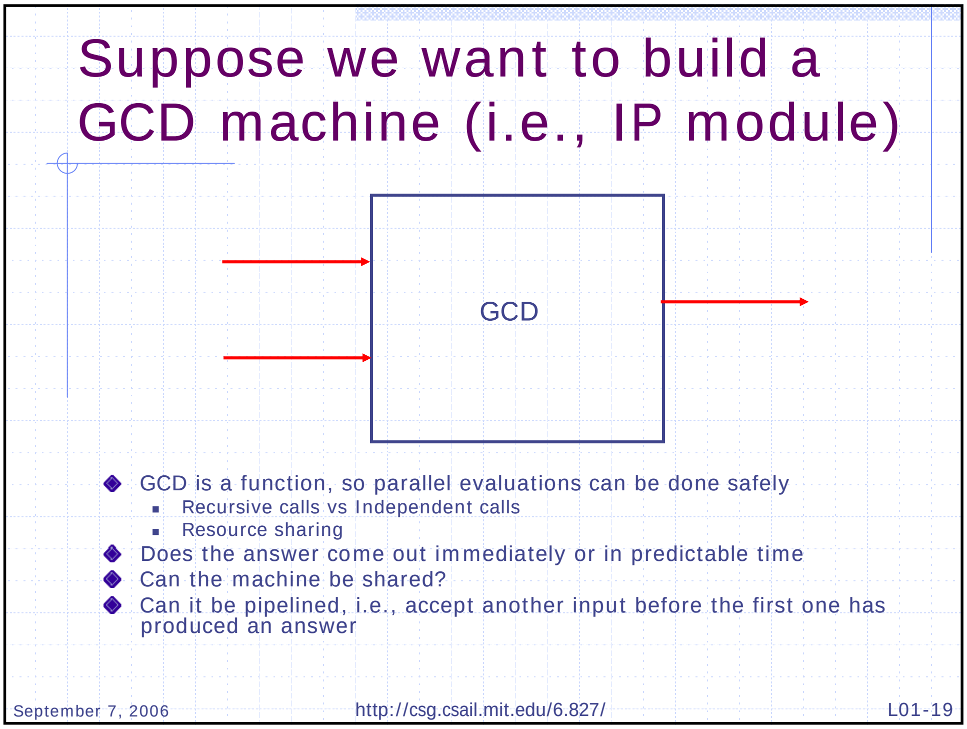
Initial term

GCD(initX,initY)

Execution

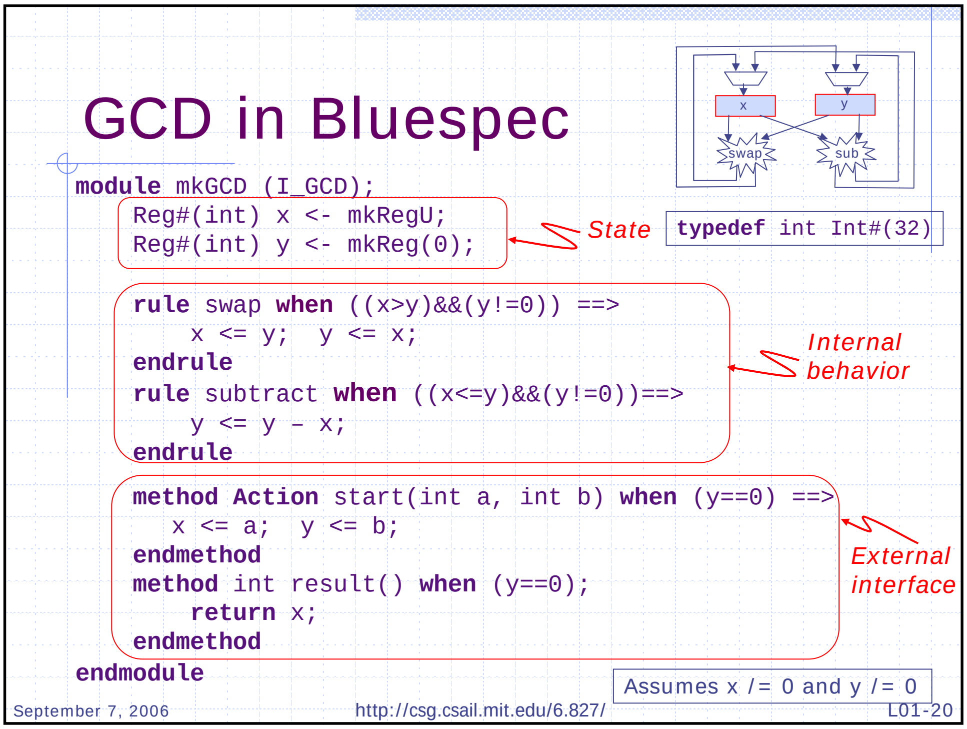
$$\text{GCD}(6, 15) \overset{R_2}{\Rightarrow} \text{GCD}(6, 9) \overset{R_2}{\Rightarrow} \text{GCD}(6, 3) \overset{R_1}{\Rightarrow}$$

$$\text{GCD}(3, 6) \overset{R_2}{\Rightarrow} \text{GCD}(3, 3) \overset{R_2}{\Rightarrow} \text{GCD}(3, 0)$$



Suppose we want to build a GCD machine (i.e., IP module)

- GCD is a function, so parallel evaluations can be done safely
  - Recursive calls vs Independent calls
  - Resource sharing
- Does the answer come out immediately or in predictable time
- Can the machine be shared?
- Can it be pipelined, i.e., accept another input before the first one has produced an answer



GCD in Bluespec

```
module mkGCD (I_GCD);
   Reg#(int) x <- mkRegU;
   Reg#(int) y <- mkReg(0);

   rule swap when ((x>y)&&(y!=0)) ==>
       x <= y;  y <= x;
   endrule
   rule subtract when ((x<=y)&&(y!=0))==>
       y <= y – x;
   endrule

   method Action start(int a, int b) when (y==0) ==>
     x <= a;  y <= b;
   endmethod
   method int result() when (y==0);
       return x;
   endmethod
endmodule
```

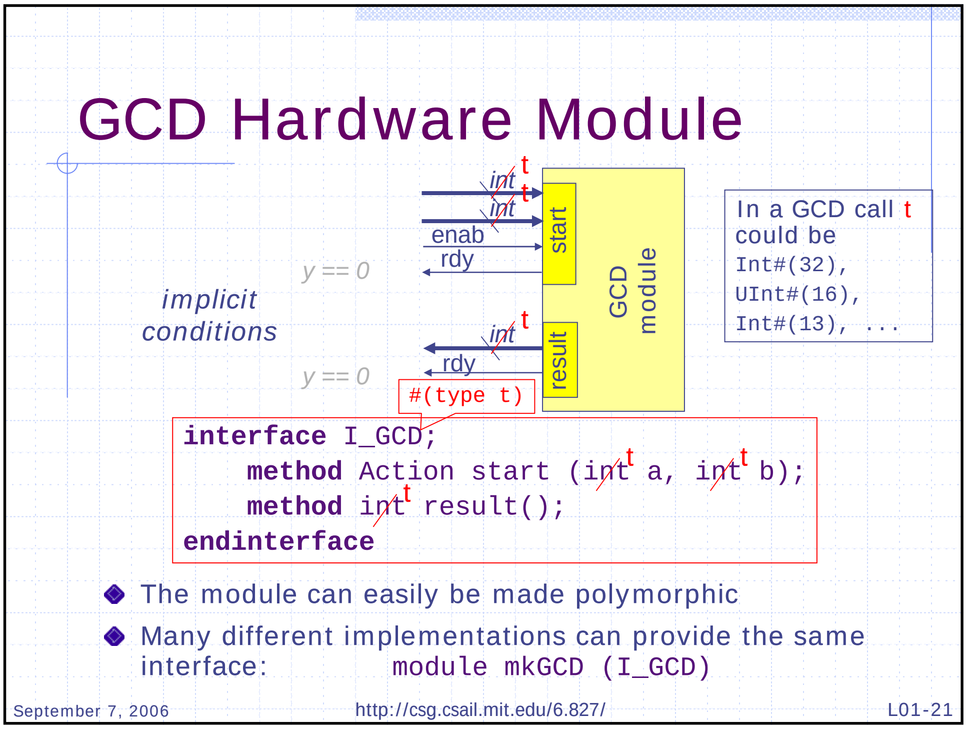
State

`typedef int Int#(32)`

Internal behavior

External interface

Assumes x /= 0 and y /= 0



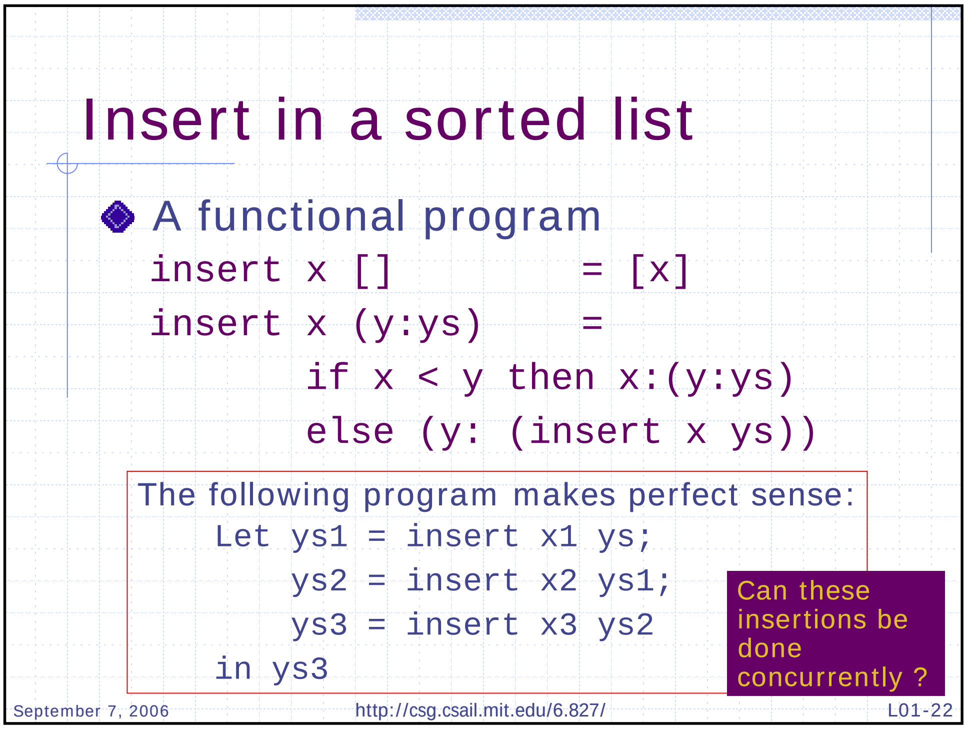
GCD Hardware Module

In a GCD call t could be Int#(32), UInt#(16), Int#(13), ...

#(type t)

interface I_GCD;
    method Action start (~~int~~t a, ~~int~~t b);
    method ~~int~~t result();
endinterface

- The module can easily be made polymorphic
- Many different implementations can provide the same interface: module mkGCD (I_GCD)



Insert in a sorted list

- A functional program

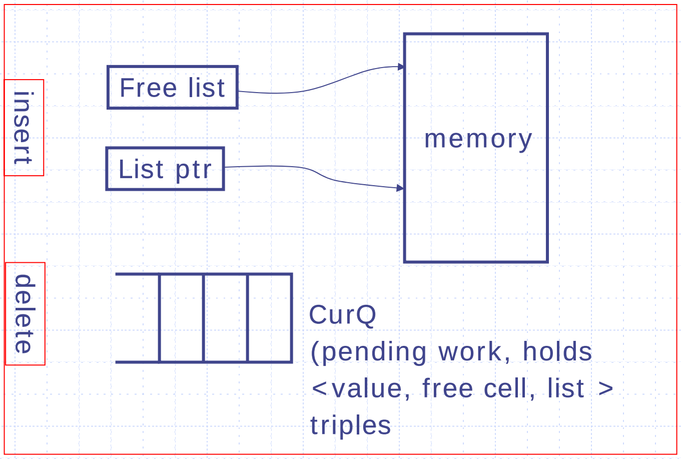
```
insert x []         = [x]
insert x (y:ys)     =
      if x < y then x:(y:ys)
      else (y: (insert x ys))
```

The following program makes perfect sense:

```
   Let ys1 = insert x1 ys;
       ys2 = insert x2 ys1;
       ys3 = insert x3 ys2
   in ys3
```

Can these insertions be done concurrently ?

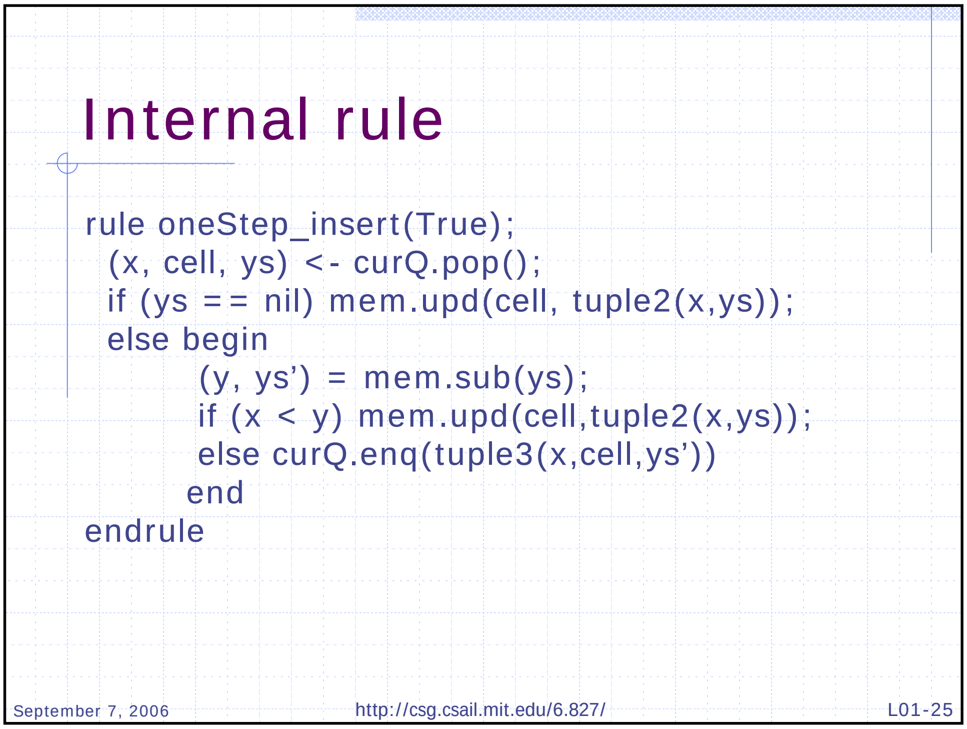
Pipelined insertions



CurQ
(pending work, holds <value, free cell, list > triples

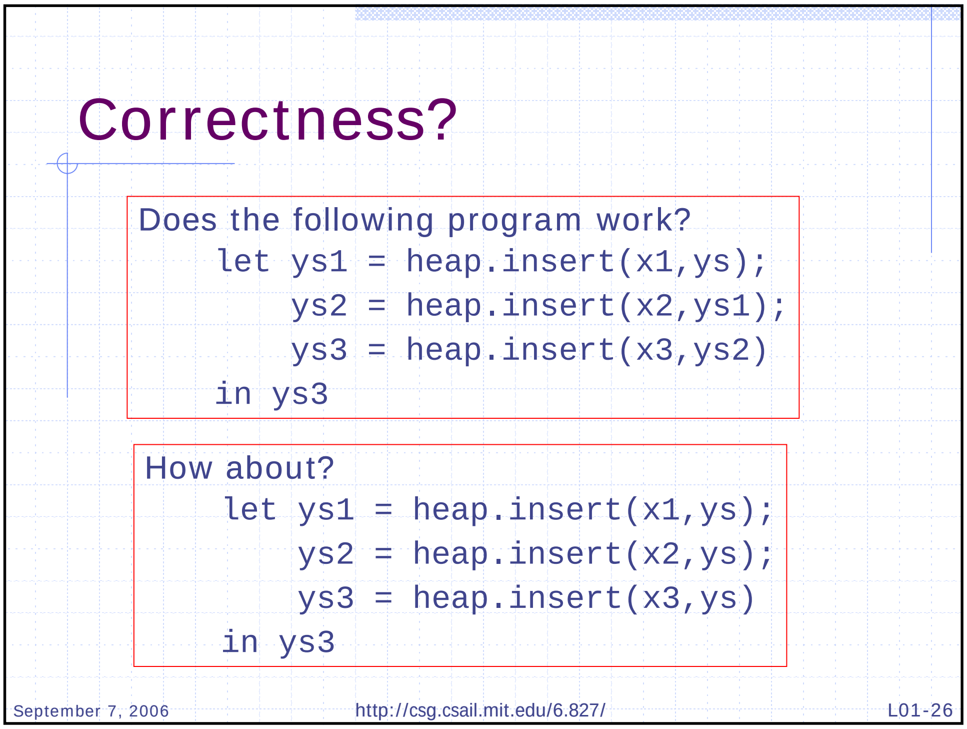
Insert method

```
method Actionvalue  insert(x, ys);
  let cell <- freelist.pop();
  if (ys == nil) then
     begin  mem.upd(cell, tuple2(x, ys)); return(cell); end
  else
     begin (y, ys') = mem.sub(ys);
              if (x < y) begin  mem.upd(cell, tuple2(x, ys));
                                return(cell); end
             else // not smallest so enqueue for a recursive call
                  begin curQ.enq(tuple3(x, cell, ys'));
                  return(ys); end
     end
endmethod
```



Internal rule

```
rule oneStep_insert(True);
  (x, cell, ys) <- curQ.pop();
  if (ys == nil) mem.upd(cell, tuple2(x,ys));
  else begin
          (y, ys’) = mem.sub(ys);
          if (x < y) mem.upd(cell,tuple2(x,ys));
          else curQ.enq(tuple3(x,cell,ys’))
         end
endrule
```



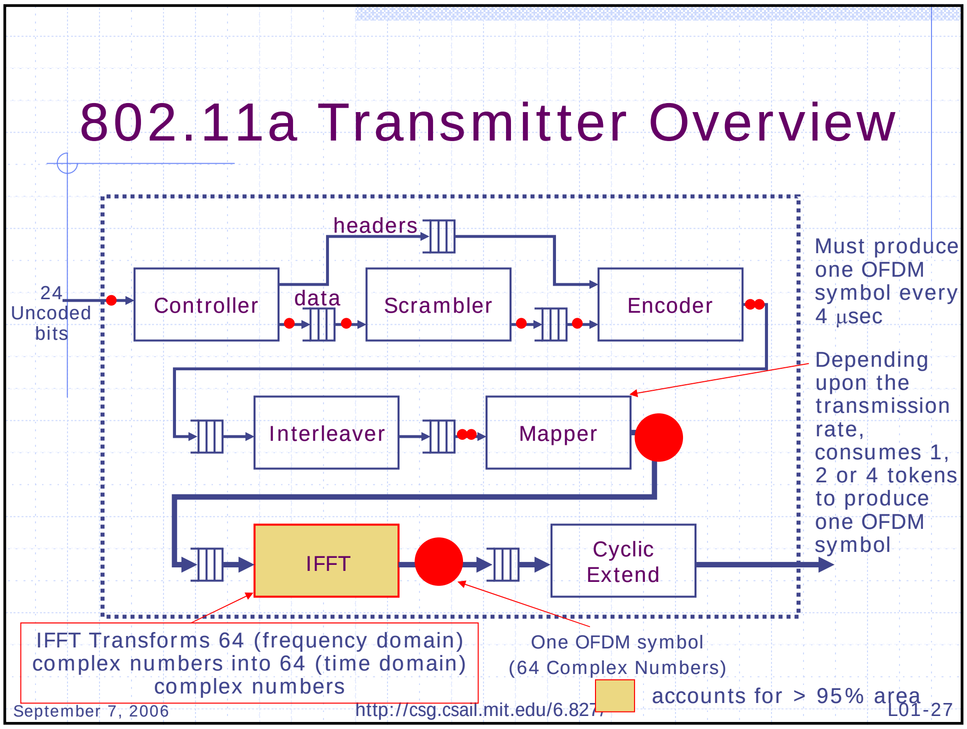
Correctness?

Does the following program work?

```
       let ys1 = heap.insert(x1,ys);
           ys2 = heap.insert(x2,ys1);
           ys3 = heap.insert(x3,ys2)
       in ys3
```

How about?

```
       let ys1 = heap.insert(x1,ys);
           ys2 = heap.insert(x2,ys);
           ys3 = heap.insert(x3,ys)
       in ys3
```



802.11a Transmitter Overview

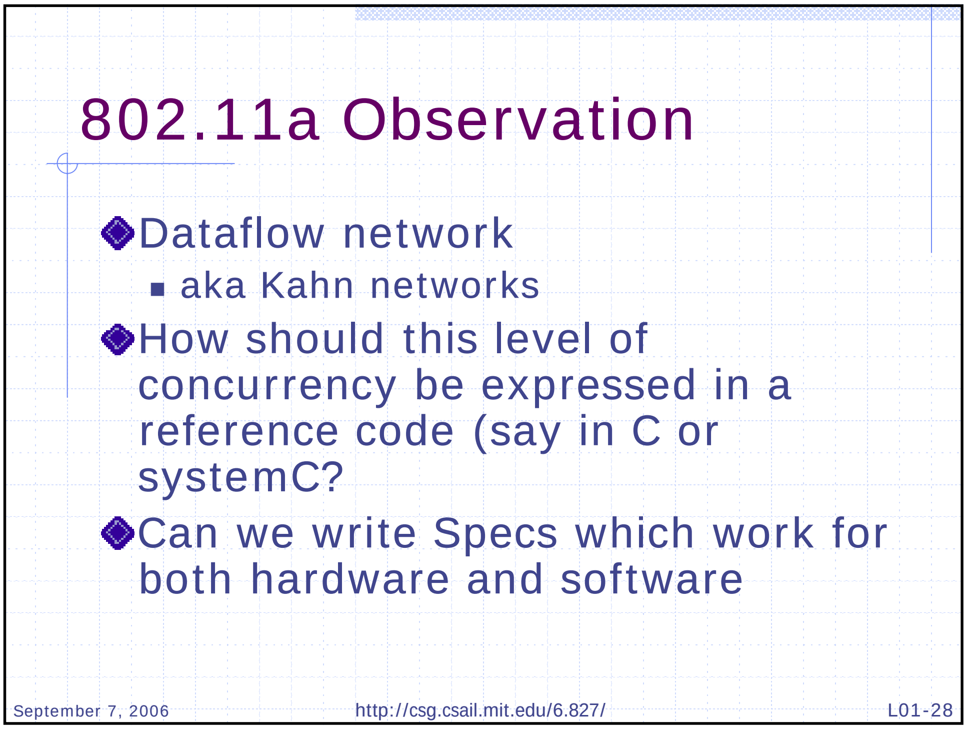
Must produce one OFDM symbol every 4 μsec

Depending upon the transmission rate, consumes 1, 2 or 4 tokens to produce one OFDM symbol

IFFT Transforms 64 (frequency domain) complex numbers into 64 (time domain) complex numbers

One OFDM symbol
(64 Complex Numbers)

▨ accounts for > 95% area



802.11a Observation

- Dataflow network
  - aka Kahn networks
- How should this level of concurrency be expressed in a reference code (say in C or systemC?
- Can we write Specs which work for both hardware and software

Dream

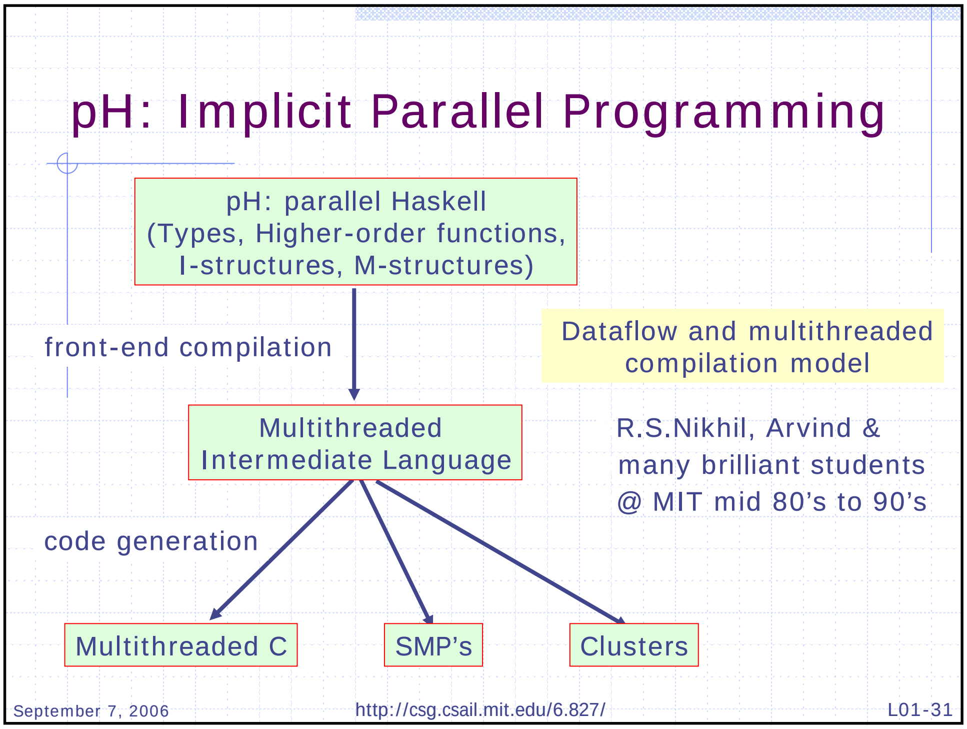
A time when Freshmen will be taught sequential programming as a special case of parallel programming

This subject is about

- The foundations of functional languages:
  - the λ calculus, types, monads, confluence, operational semantics, TRS…
- General purpose implicit parallel programming in Haskell & pH
- Parallel programming based on atomic actions or transactions in Bluespec
- Dataflow model of computation

and understanding connections …

Bluespec and pH borrow heavily from functional languages but have completely different execution models



pH: Implicit Parallel Programming

pH: parallel Haskell
(Types, Higher-order functions,
I-structures, M-structures)

front-end compilation

Multithreaded
Intermediate Language

code generation

Multithreaded C

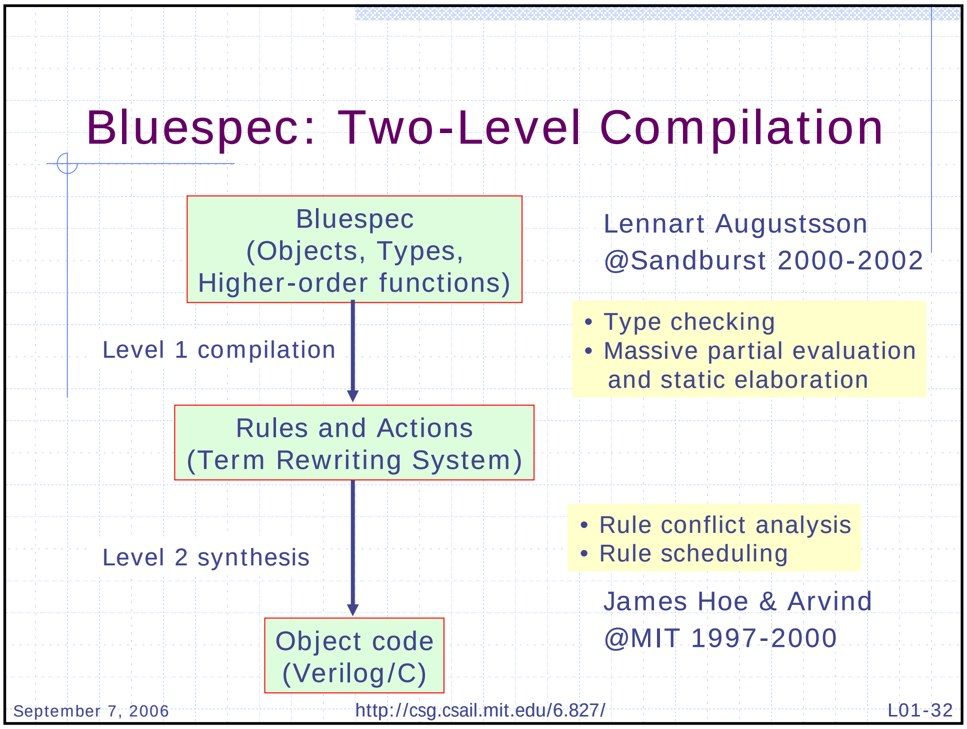
SMP's

Clusters

Dataflow and multithreaded
compilation model

R.S.Nikhil, Arvind &
many brilliant students
@ MIT mid 80's to 90's

September 7, 2006 http://csg.csail.mit.edu/6.827/ L01-31



Bluespec: Two-Level Compilation

Bluespec
(Objects, Types,
Higher-order functions)

Level 1 compilation

Rules and Actions
(Term Rewriting System)

Level 2 synthesis

Object code
(Verilog/C)

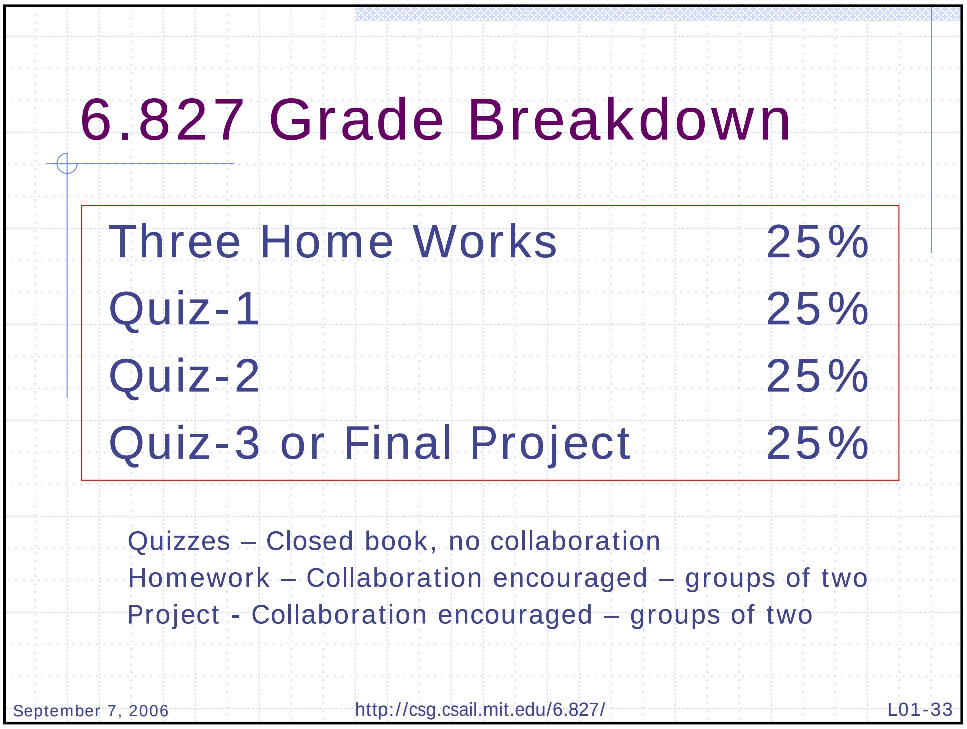
Lennart Augustsson
@Sandburst 2000-2002

- Type checking
- Massive partial evaluation and static elaboration

- Rule conflict analysis
- Rule scheduling

James Hoe & Arvind
@MIT 1997-2000

September 7, 2006 http://csg.csail.mit.edu/6.827/ L01-32



6.827 Grade Breakdown

Three Home Works	25%
Quiz-1	25%
Quiz-2	25%
Quiz-3 or Final Project	25%

Quizzes – Closed book, no collaboration
Homework – Collaboration encouraged – groups of two
Project - Collaboration encouraged – groups of two