Synthesis of Circuits from Parallel Programs

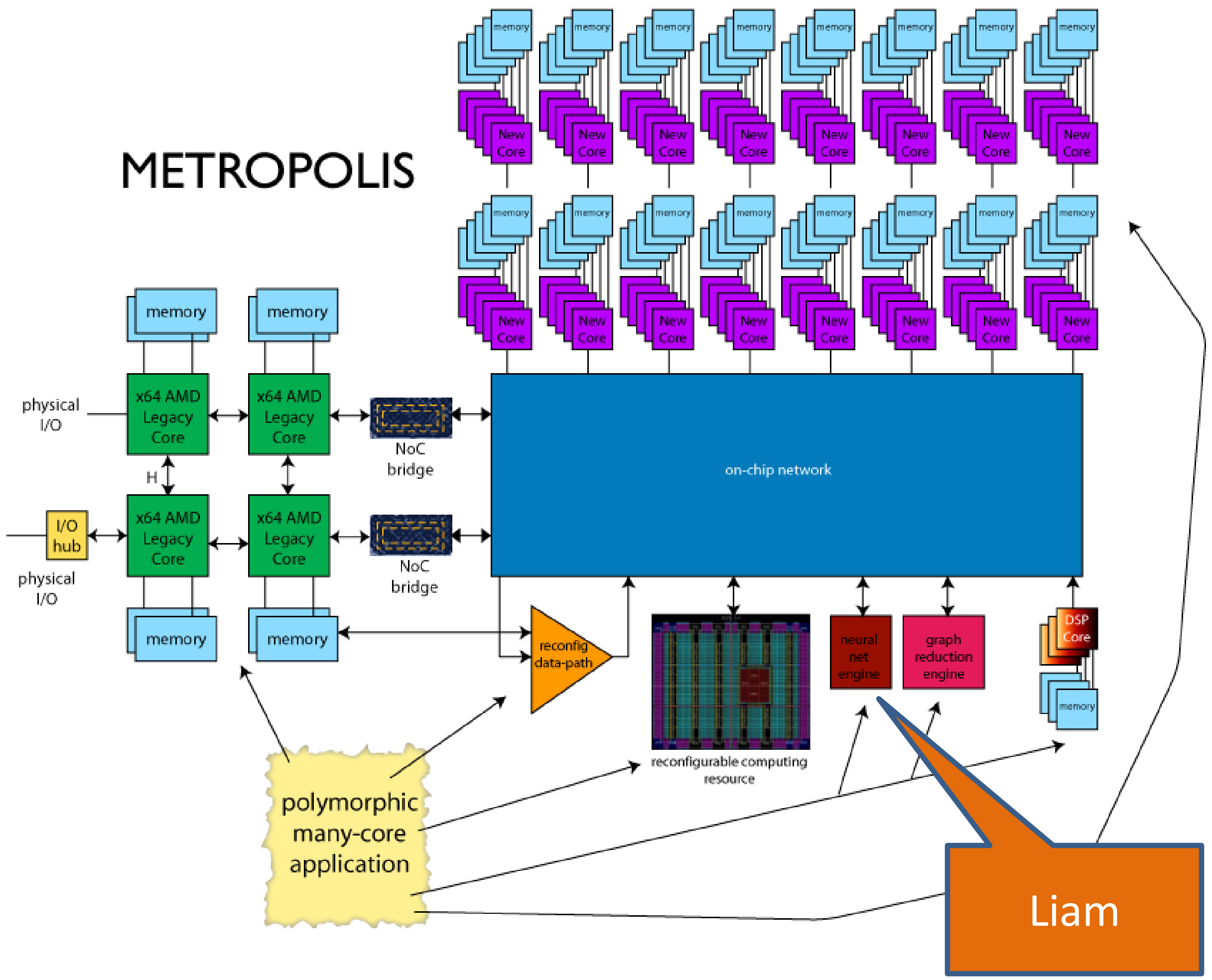
Satnam Singh

Microsoft Research Cambridge, UK

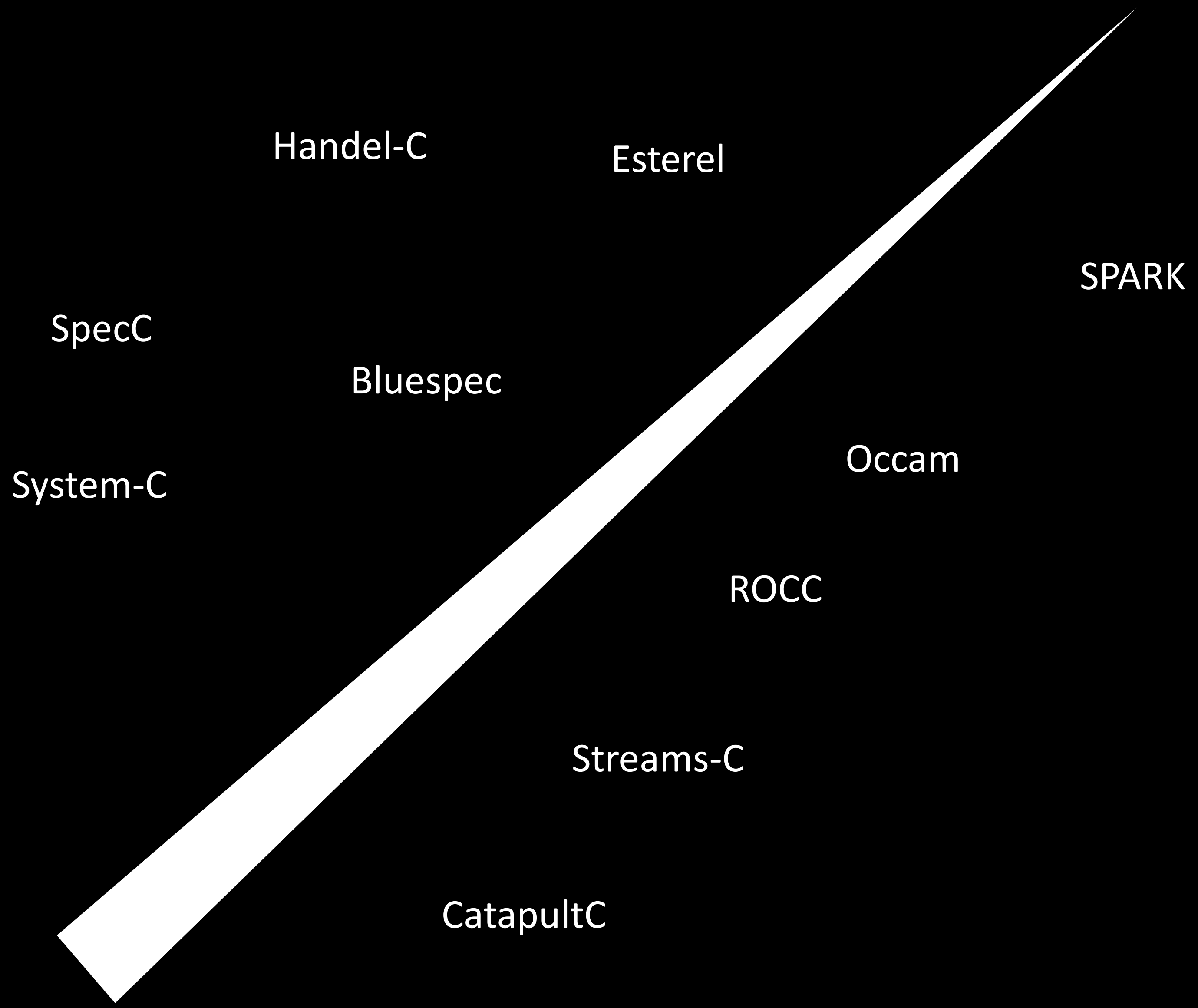
David Greaves

Computer Lab, Cambridge University, UK

The Future is Heterogeneous



ray of light


Handel-C
Esterel
SPARK
SpecC
Bluespec
System-C
Occam
ROCC
Streams-C
CatapultC

Previous Work

- Starts with **sequential** C-style programs.
- Uses various heuristics to **discover** opportunities for parallelism esp. in nested loops.
- Good for certain idioms that can be recognized.
- However, many parallelization opportunities are not discovered because they are not evident in the structure of the program.

Microsoft Research

Modelling Circuits in C++ is Nothing New

```
class Counter : public Process {
private:
  // clock is in the base class
  const Signal<std_ulogic> & enable; // input
  Signal<std_ulogic>& iszero; // output
  int count; // state
public:
  Counter(
  // interface specification
  Clock& CLK,
  const Signal<std_ulogic>& EN,
  Signal<std_ulogic>& ZERO
  )
 // initializers - mapping ports
 : Process(CLK), enable(EN), iszero(ZERO)
 { count = 15; } // process initialization
 void entry();
};
```

sequential process declaration for a counter

```
void Counter::entry()
{ if (enable.read() == ‘1’)
  { if (count == 0)
    { write(iszero, ‘1’);
      count = 15;
    }
    else
    { write(iszero, ‘0’);
      count–;
    }
  }
  next();
}
```

body of counter process

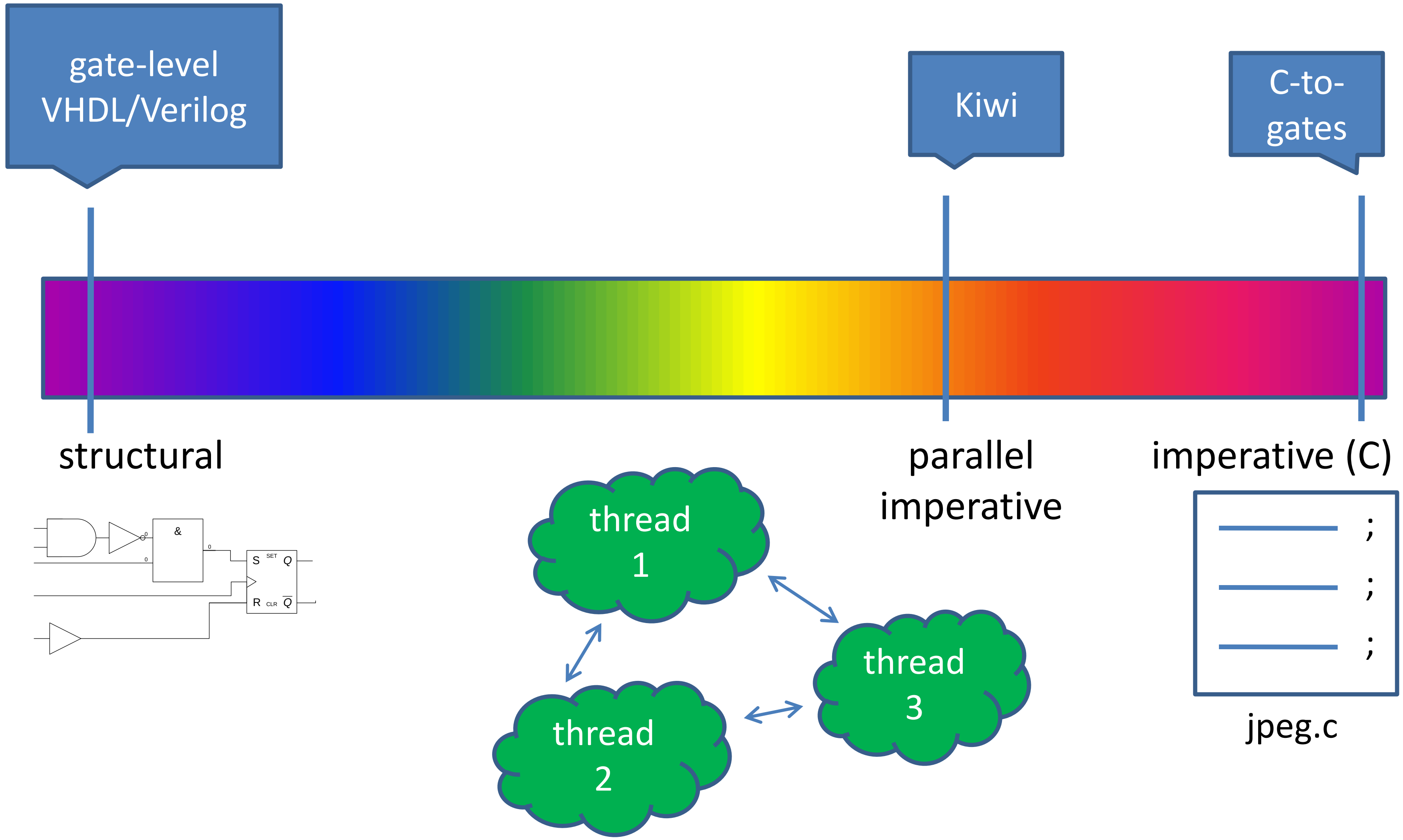
Parallel Programming

- We need to **produce** parallel programs:
  - Multi-core, many-core
  - GPUS
- But do we need to **write** parallel programs?
  - Implicit parallelism?
  - Automatic parallelization of C programs?
  - Write with locks and threads?
  - Write explicit data-parallel programs?
- Separated at birth:
  - Hardware description languages
  - Concurrent and parallel programming languages

Objectives

- A system for **software engineers**.
- **Model** synchronous digital circuits in C# etc.
  - Software models offer greater **productivity** than models in VHDL or Verilog.
- **Transform** circuit models automatically into circuit implementations.
- Exploit existing **concurrent software verification tools**.

Kiwi


gate-level VHDL/Verilog
Kiwi
C-to-gates
structural
parallel imperative
imperative (C)
&
S SET Q
R CLR Q̄
thread 1
thread 2
thread 3
;
;
;
jpeg.c

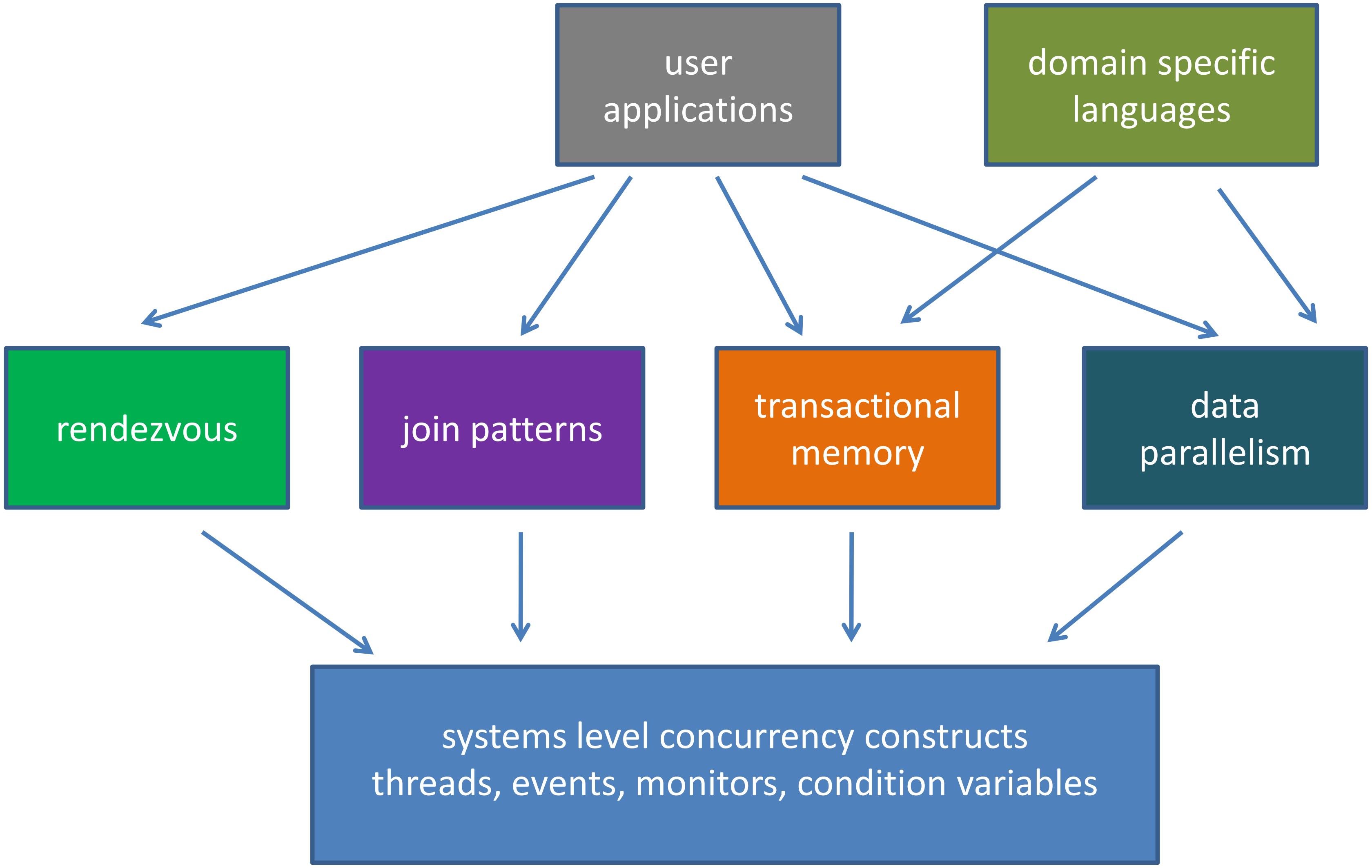
The Accidental Semi-colon

Synthesizing parallel programs
(or borrowing some ideas from hardware design)

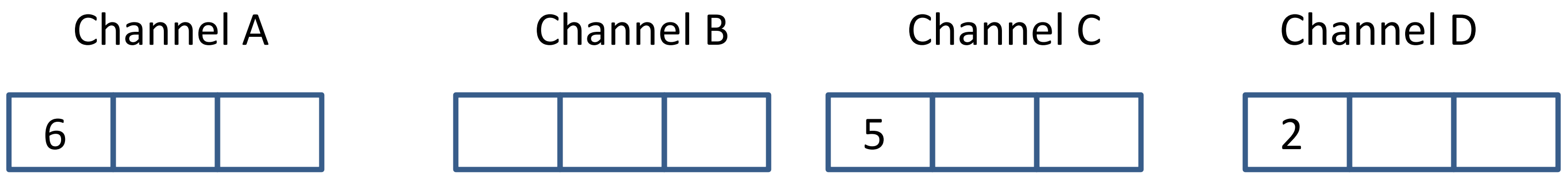
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

6.189

January 24, 2007


user applications
domain specific languages
rendezvous
join patterns
transactional memory
data parallelism
systems level concurrency constructs
threads, events, monitors, condition variables

Join Patterns



join pattern

~~print x+y~~

handler

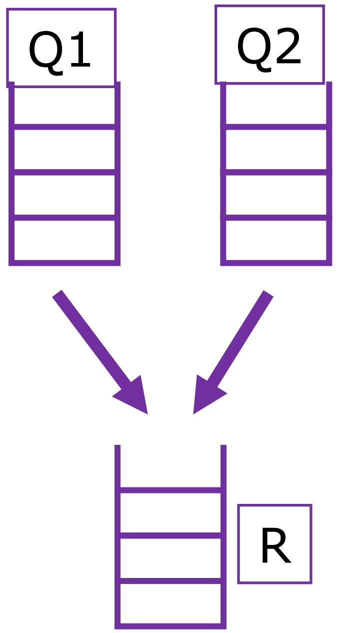
A(x) & D(y) & C(z) -> print x-y+z

A(x) & C(y) -> print x-y

x=6, y=2, output = 4

Transactional Memory

```
void GetEither() {
        atomic {
                do { i = Q1.Get(); }
                orelse { i = Q2.Get(); }

                R.Put( i );
}       }
```



do {...this...} orelse {...that...} tries to run “this”

- If “this” retries, it runs “that” instead
- If both retry, the do-block retries. GetEither() will thereby wait for there to be an item in *either* queue

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()) ;
  }
}
```

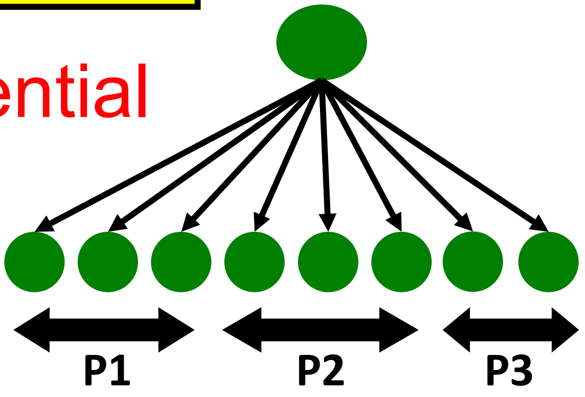
Flat data parallel

e.g. Fortran(s), *C
MPI, map/reduce

- The brand leader: widely used, well understood, well supported

```
foreach i in 1..N {
    ...do something to A[i]...
}
```

- BUT: "**something**" is sequential
- Single point of concurrency
- Easy to implement: use "chunking"
- Good cost model



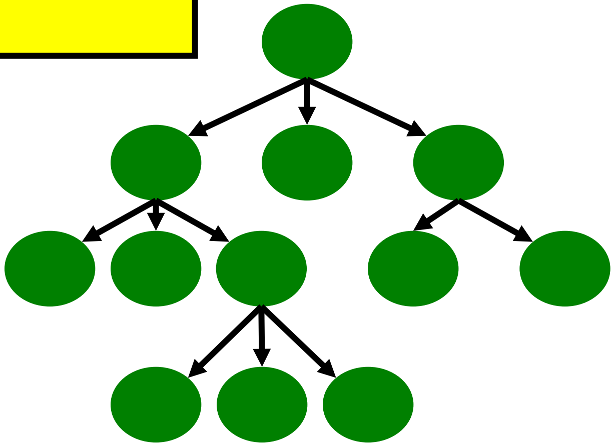
1,000,000's of (small) work items

Nested data parallel

- Main idea: **allow “`something`” to be parallel**

```
foreach i in 1..N {
    ...do something to A[i]...
}
```

- Now the parallelism structure is recursive, and un-balanced
- Still good cost model
- Hard to implement!



Still 1,000,000’s of (small) work items

Array comprehensions

[:Float:] is the type of parallel arrays of Float

```
vecMul :: [:Float:] -> [:Float:] -> Float
vecMul v1 v2 = sumP [: f1*f2 | f1 <- v1 | f2 <- v2 :]
```

sumP :: [:Float:] -> Float

An array comprehension: "the array of all f1*f2 where f1 is drawn from v1 and f2 from v2"

Operations over parallel array are computed in parallel; that is the only way the programmer says "do parallel stuff"

NB: no locks!

Venn Diagram

event-based simulation

Kahn networks

multi-clock

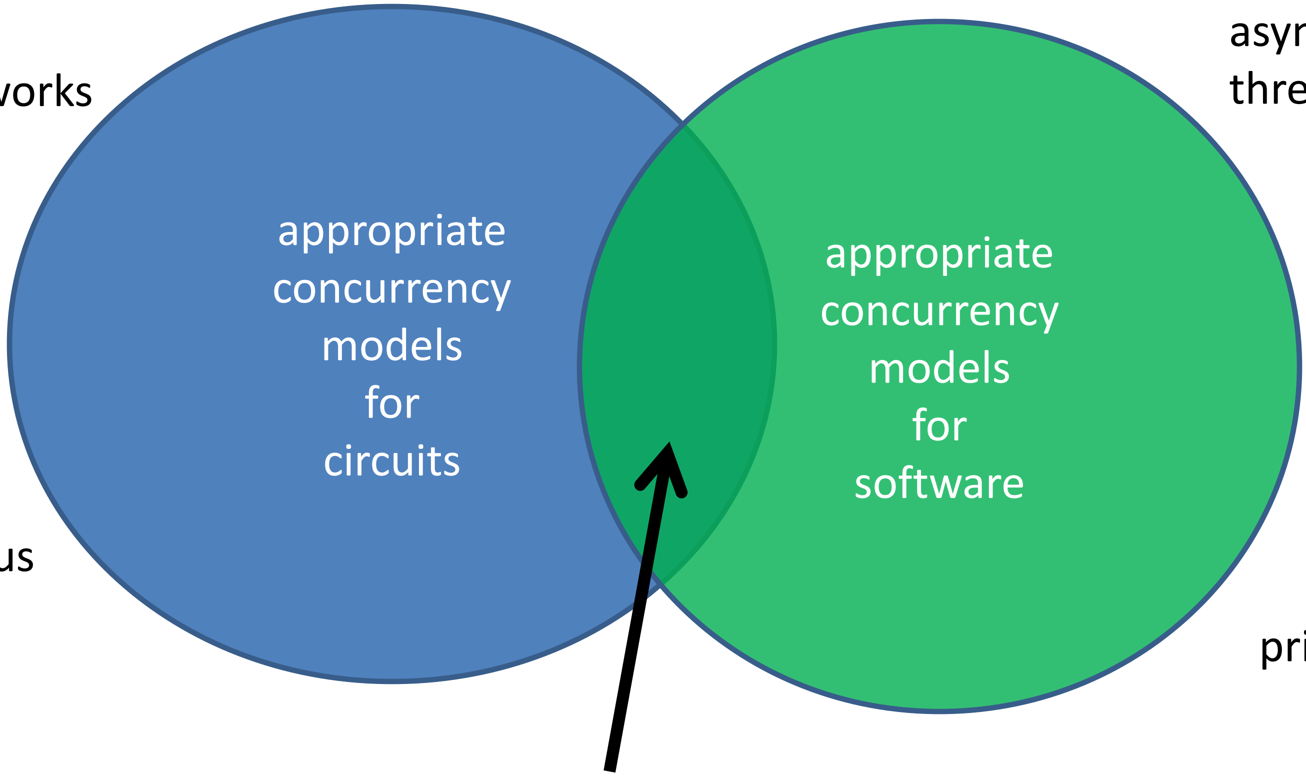
synchronous
data-flow

asynchronous
threads

monitors

events

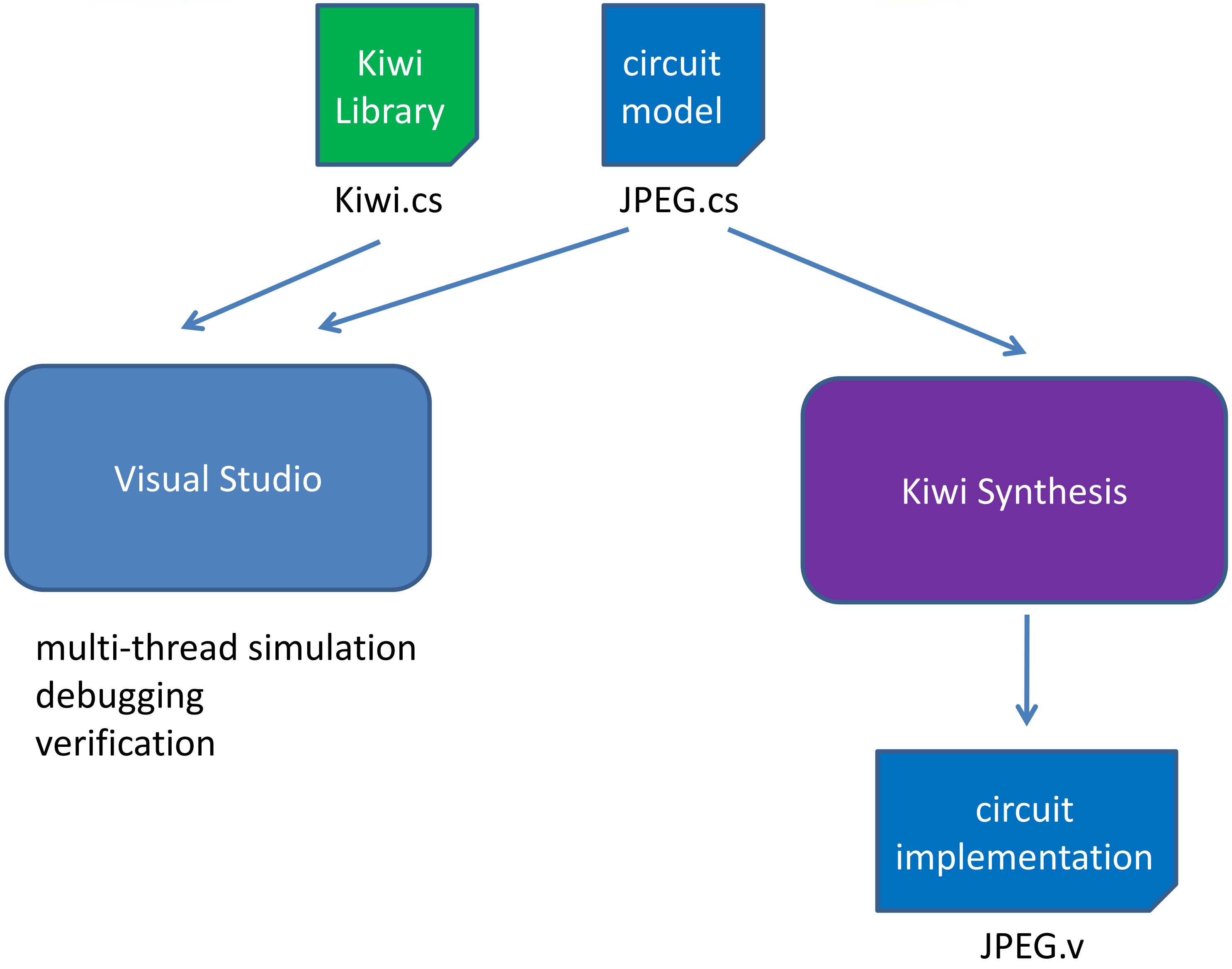
priorities


appropriate concurrency models for circuits
appropriate concurrency models for software

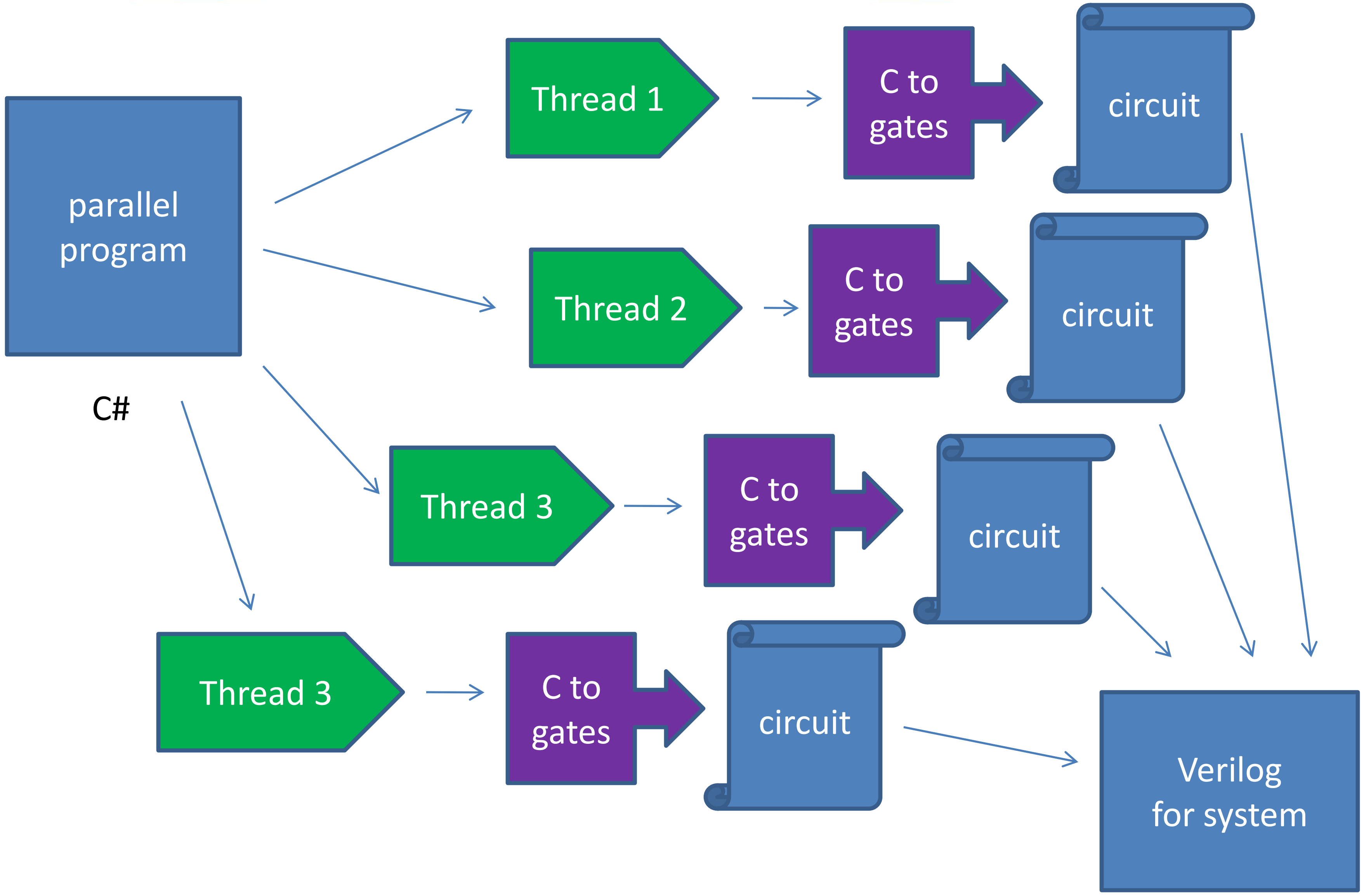
Are there enough concurrency abstractions
that make sense in hardware **and** software?

Our Idea

- Write **parallel** programs in C# (F# etc.)
- Use the **parallel decription** to specify top-level **circuit architecture**.
- Analyze existing **concurrency idioms** to see what can be efficiently translated to circuits.
- Capture useful design idioms and represent them in a **concurrency library** for circuit description: Kiwi.



multi-thread simulation
debugging
verification


parallel program
C#
Thread 1
Thread 2
Thread 3
Thread 3
C to gates
C to gates
C to gates
C to gates
circuit
circuit
circuit
circuit
Verilog for system

System.Threading

- We have decided to target hardware synthesis for a sub-set of the concurrency features in the .NET library System.Threading
  - – Events (clocks)
  - – Monitors (synchronization)
  - – Thread creation etc. (circuit structure)

Kiwi Concurrency Library

- A conventional concurrency library Kiwi is exposed to the user which has two implementations:
  - A software implementation which is defined purely in terms of the support .NET concurrency mechanisms (events, monitors, threads).
  - A corresponding hardware semantics which is used to drive the .NET IL to Verilog flow to generate circuits.
- A Kiwi program should always be a sensible concurrent program but it may also be a sensible parallel circuit.

Higher Level Concurrency Constructs

- By providing hardware semantics for the system level concurrency abstractions we hope to then automatically deal with other higher level concurrency constructs:
  - Join patterns (C-Omega, CCR, .NET Joins Library)
  - Rendezvous
  - Data parallel operations

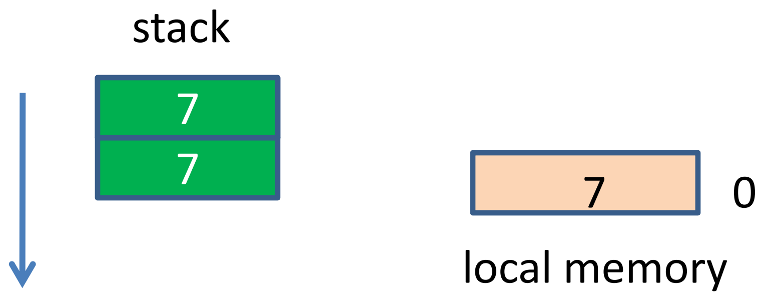
Our Implementation

- Use regular Visual Studio technology to generate a .NET IL assembly language file.
- Our system then processes this file to produce a circuit:
  - The .NET stack is analyzed and removed
  - The control structure of the code is analyzed and broken into basic blocks which are then composed.
  - The concurrency constructs used in the program are used to control the concurrency / clocking of the generated circuit.

```
public static int max2(int a, int b)
{ int result;
  if (a > b)
     result = a;
  else
     result = b;
  return result;
}
```

```
.method public hidebysig static
int32
         max2(int32 a,
              int32 b) cil managed
{
  // Code size       12 (0xc)
  .maxstack  2
  .locals init ([0] int32 result)
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  ble.s      IL_0008

  IL_0004:  ldarg.0
  IL_0005:  stloc.0
  IL_0006:  br.s       IL_000a

  IL_0008:  ldarg.1
  IL_0009:  stloc.0
  IL_000a:  ldloc.0
  IL_000b:  ret
}
```

```
max2(3, 7)
```



```
public static int SumArray()
{
  int[] a = new int[] { 7, 3, 5, 2, 1 };
  int sum = 0;
  foreach (int n in a)
    sum += n;
  return sum;
}
```

dynamic memory allocation

garbage collection

```
IL_0000:  ldc.i4.5
IL_0001:  newarr     [mscorlib]System.Int32
...
IL_000c:  call       void
[mscorlib]System.Runtime.CompilerServices.RuntimeHelpers::
InitializeArray(class [mscorlib]System.Array,

valuetype [mscorlib]System.RuntimeFieldHandle)
      IL_0011:  stloc.0
```

native OO support

Stack-based to Register-based

```
ldc.i4.42
ldloc.5
mul
dup
add
```

→

```
loadreg r1 #42
loadreg r2 &5
mult r1, r2, r3
movereg r3 r2
add r2, r3, r1
```

Worked Example

Two-phase handshake on parallel port

```
using System;
using KiwiSystem;
public class parallel_port
{  [Kiwi.OutputWordPort("dout")]
   static byte dout;
   [Kiwi.OutputBitPort("strobe")]
   static bool strobe;
   [Kiwi.InputBitPort("ack")]
   static bool ack;

   public static void putchar(byte c)
   { while (ack == strobe)
       Kiwi.Pause();
       dout = c;
       Kiwi.Pause();
       strobe = !strobe;
    }
}
```

implicit synchronization with a clock

Top Level Driver

```
class TopLevelPortDriver
{
  public static void parallel_print(string s)
  {
    for (int i = 0; i<s.Length; i++)
        parallel_port.putchar((byte)s[i]);
  }

  public static void Main()
  { parallel_print("Hello World\n"); }
}
```

Internal Virtual Machine

- We use an internal virtual machine:
  - .NET IL parsed into intermediate machine
  - Intermediate machine supports imperative code sections
  - Code sections can be in series or parallel (SER/PAR blocks)
  - IL elaboration subsumes a number of variables including object pointers

IL Elaboration

- The IL elaborator takes the parse tree and list of root method names identified by the user.
- A symbol table is built up (heap) containing variables with different kinds of status:
  - subsumed: value tracked entirely at compile time
  - elaborated: value appears in output of machine
  - undecided: no decision has been forced yet
- Stack eliminated using additional heap (spill) variables at IL transfer of control (jump or branch).

IL Elaboration

- Two passes:
  1. Determine quantity and type of values on the stack
     - Multiple branches to the same destination must share the same stack format
  2. Emit HPR code from IL method body
- Elaboration involves direction translation of control structures.
- Symbolic manipulation of other structures
  - Assignment for stind, stsfld, stfld
  - Side effecting function call when code pops and discards something from stack
- A newobj and newarr instruction causes allocation of a symbolic constant: variables over such constants are subsumed.

Internal Virtual Machine Format

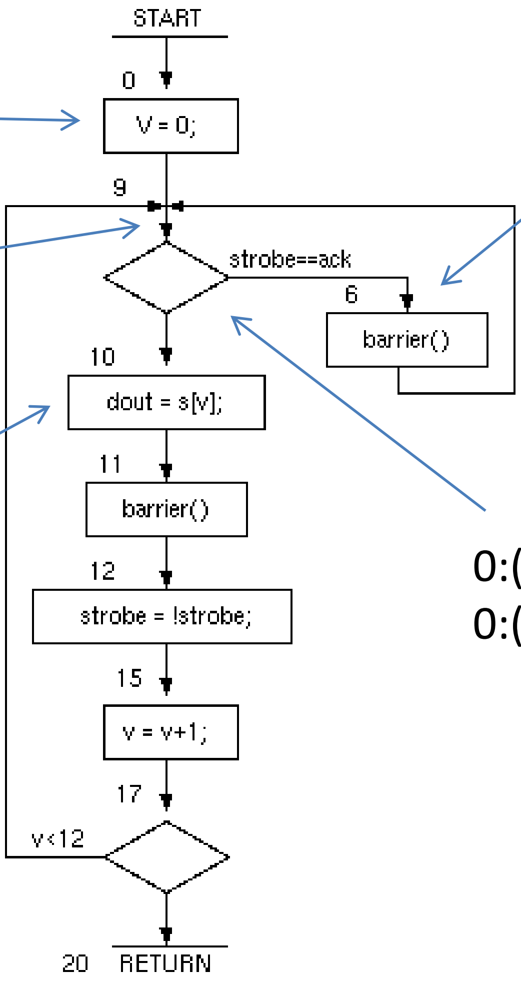
sensitivity=NONE Listing: id=Main
0:test9_parallel_print_V_0 := 0;
1:Xgoto(test9/parallel_print/IL_0018, 16);
2:test9/parallel_print/IL_0007:
3:Xgoto(cilreturn115, 4); 4:cilreturn115:
5:Xgoto(parallel_port/putchar/IL_000a, 8);
6:parallel_port/putchar/IL_0005:
7:*APPLY:hpr_barrier();
8:parallel_port/putchar/IL_000a:
9:beq(!!(parallel_port_ack^parallel_port_strobe),parallel_port/putchar/IL_0005, 6)
10:parallel_port_dout := "Hello World\n"[test9_parallel_print_V_0]&mask(7..0);
11:*APPLY:hpr_barrier();
12:parallel_port_strobe := !parallel_port_strobe; 13:Xgoto(cilreturn116, 14); 14:cilreturn116:
15:test9_parallel_print_V_0 := test9_parallel_print_V_0+1;
16:test9/parallel_print/IL_0018: 17:beq(10<=test9_parallel_print_V_0,test9/parallel_print/IL_0007, 2) 18:Xgoto(cilreturn117, 19);
19:cilreturn117: 20:return 0;

- Stack eliminated
- Subroutine calls flattened
- Main loop directly manipulates ports

Representation

- Finite-state machine edges have one two forms:
  - (g, v, e)
    - Assign e to v when g holds
  - (g, f, [args])
    - Call built-in function f with args when g holds
- Pending activation queue
  - (p==v, g, S)
    - When program counter is v and g holds perform variable updates in S

Strobe Example



0:(pc==0, true, [])

0:(pc==9, true, [0/V])

0:(pc==6, strobe==ack, [0/V])

6:(pc==9, true, [])

0:(pc==6, strobe==ack, [0/V])
0:(pc==10, strobe!=ack, [0/V])

0:(pc==10, strobe!=ack, [0/V])
0:(pc==11, strobe!=ack, [0/V, s[0]/dout])
11:(pc==12, true, [])
11:(pc==15, true, [!strobe/strobe])
11:(pc==17, true, [V+1/V, !strobe/strobe])

Conversion to a Finite State Machine

- A virtual machine to virtual machine transformation.
- A user provided unwind budget specifying how many basic blocks to consider in any loop unwind operation.
- When loops are nested or there is a fork in control flow the budget is appropriately divided.

Generated Verilog

```
module PARP(clk, reset, parallel_port_ack,
parallel_port_dout, parallel_port_strobe);
input clk;
input reset;
input parallel_port_ack;
output [7:0] parallel_port_dout;
reg [7:0] parallel_port_dout;
output parallel_port_strobe;
reg parallel_port_strobe;
reg [1:0] pcnet119p;
parameter str99 = "Hello World\n"; integer
test9_parallel_print_V_0;

always @(posedge clk)
  begin case (pcnet119p)
    0: begin
         if (reset) pcnet119p <= 0;
         if (parallel_port_ack==parallel_port_strobe &&
!reset) pcnet119p <= 2; if
```

Example: I2C Bus Controller

- I2C is a commonly used serial protocol.
- Circuit developed to initialize a DVI video chip on a FPGA board.
- First version written by hand in VHDL with nested case statements (horrible).
- Second version written in C# and translated into Verilog using our system (much nicer!).

I2C Bus Control in VHDL

```
elsif control clock'event and control clock='1' then
  case state is
    when initial   => case phase is
                        when 0 => scl <= '1' ; sda out <= '1' ;
                        when 1 => null ;
                        when 2 => sda out <= '0' ; -- Start condition
                        when 3 => scl <= '0' ;
                                  index := 6 ;
                                  state := deviceID state ;
                      end case ;
    when deviceID state => case phase is
                              when 0 => sda out <= deviceID (index) ; -- Get data ready on SDA
                              when 1 => scl <= '1' ; -- Start SCL high pulse
                              when 2 => scl <= '0' ; -- End SCL high pulse
                              when 3 => sda out <= '0' ; -- Clear SDA data
                                        if index = 0 then
                                          state := setrw read ;
                                        else
                                          index := index - 1 ;
                                        end if ;
                            end case ;
```

Ports and Clocks

```
public static class I2C
    {   [OutputBitPort("scl")]
         static bool scl;

         [InputBitPort("sda_in")]
         static bool sda_in;

         [OutputBitPort("sda_out")]
         static bool sda_out;

         [OutputBitPort("rw")]
         static bool rw;
```

circuit ports identified by custom attribute

System Composition

- We need a way to separately develop components and then compose them together.
- Don’t invent new language constructs: reuse existing concurrency machinery.
- Adopt channels for the composition of components.
- Model channels with regular concurrency constructs (monitors).

Channels and Condition Variables

```
public class channel<T>
   {    T datum;
        bool empty = true;

        public void write(T v)
        {
            lock(this)
            {
                while (!empty)
                    Monitor.Wait(this) ;
                datum = v ;
                empty = false ;
                Monitor.PulseAll(this);
            }
        }
```

Channels: Reading with Monitor

```
public T read()
      {  T r ;
         lock (this)
         {
            while (empty)
               Monitor.Wait(this);
            empty = true;
            r = datum;
            Monitor.PulseAll(this);
         }
         return r;
      }
```

Producer/Consumer Example

```
class ConsumerClass
{  channel<int> x;
   public ConsumerClass(channel<int> c)
   {  x = c; }

   public void process()
   { while (true)
      {  int r = x.read();
         Console.Write("{0} ", r);
      }
   }
}
```

```
class TimesTable
{
   static int limit = 5;

   public static void Main()
   {  int i, j;
      channel<int> mych = new channel<int>()
      ConsumerClass consumer = new ConsumerClass(mych);

       Thread thread1 =
          new Thread(new ThreadStart(consumer.process));
       thread1.Start();

       Console.WriteLine("Times Table Up To " + limit);
       for (i = 1; i <= limit; i++)
       {
          for (j = 1; j <= limit; j++) mych.write(i * j);
          Console.WriteLine("");
       }
   }
}
```

Generated Verilog

```
reg hpr_testandset_res205;
  reg hpr_testandset_res206;
  reg hpr_testandset_res209;
  reg hpr_testandset_res210;
   always @(posedge clk) begin if (!nel_1____Orangelib_channel_1_empty && pcnet212p==0 || pcnet212p==1 && !nel_1____Orangelib_channel_1_empty)
           $write("%d "
  , nel_1____Orangelib_channel_1_datum);

  if (pcnet212p==1) hpr_testandset_res210 <= pcnet212p==1 ? 0: 1'bx;
  if (!nel_1____Orangelib_channel_1_empty && pcnet212p==0 || pcnet212p==1 && !nel_1____Orangelib_channel_1_empty) process_V_0
   <= !nel_1____Orangelib_channel_1_empty && pcnet212p==0 || pcnet212p==1 && !nel_1____Orangelib_channel_1_empty ?
           nel_1____Orangelib_channel_1_datum: 1'bx;
  if (!nel_1____Orangelib_channel_1_empty && pcnet212p==0 || pcnet212p==1 && !nel_1____Orangelib_channel_1_empty) Orangelib_channel_1_read_V_0
   <= !nel_1____Orangelib_channel_1_empty && pcnet212p==0 || pcnet212p==1 && !nel_1____Orangelib_channel_1_empty ?
           nel_1____Orangelib_channel_1_datum: 1'bx;
  if (!nel_1____Orangelib_channel_1_empty && pcnet212p==0 || pcnet212p==1 && !nel_1____Orangelib_channel_1_empty)
           nel_1____Orangelib_channel_1_empty
   <= !nel_1____Orangelib_channel_1_empty && pcnet212p==0 || pcnet212p==1 && !nel_1____Orangelib_channel_1_empty ? 1: 1'bx;
  if (pcnet212p==0 || pcnet212p==1) nel_1_mutex <= pcnet212p==0 || pcnet212p==1 ? 0: 1'bx;
  if (pcnet212p==0 || pcnet212p==1 && !nel_1____Orangelib_channel_1_empty) hpr_testandset_res209
   <= pcnet212p==0 || pcnet212p==1 && !nel_1____Orangelib_channel_1_empty ? 0: 1'bx;
  pcnet212p
   <= reset ? 0
      : pcnet212p==1 && !nel_1____Orangelib_channel_1_empty ? 1
        : pcnet212p==1 && nel_1____Orangelib_channel_1_empty ? 1
          : pcnet212p==0 && !nel_1____Orangelib_channel_1_empty ? 1: pcnet212p==0 && nel_1____Orangelib_channel_1_empty ? 1: pcnet212p;
  if (4<Main_V_1 && nel_1____Orangelib_channel_1_empty && pcnet208p==1) $display("");

  if (pcnet208p==0) $display("%s%d", "Times Table Up To ", 5);
```

The problem with **int**

[Kiwi.HwWidth(5)] [Kiwi.OutputPort(””)] static byte out

Temporal Assertions

```
[Kiwi.AssertCTL("AG", "pred1 failed")]
public bool pred1()
{ return (... ); }
```

Current Limitations

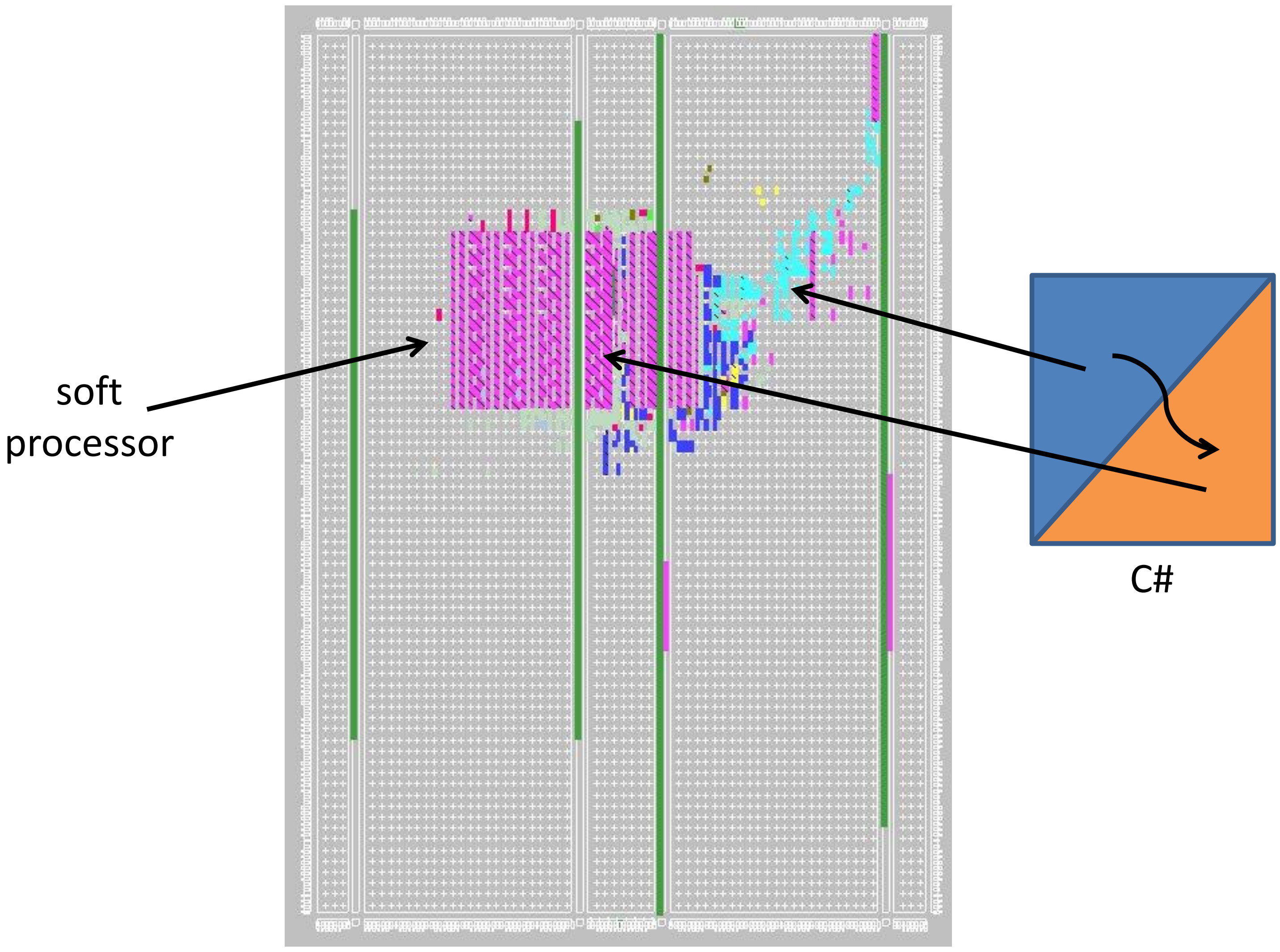
- Only integer arithmetic and string handling.
- Floating point could be added easily.
- Generation of statically allocated code:
  - Arrays must be dimensioned at compile time
  - Number of objects on the heap is determined at compile time
  - Recursive function calling must bottom out at compile time (so depth can not be run-time dependent)

Impedance Match with Synthesis Tools

- FPGA design tools come with efficient synthesis tools that translates behavioural Verilog/VHDL descriptions to decent hardware.
- Generating a totally synthesized netlist (AND gates, OR gates, flip-flops) does not exploit this power.
- At what level of abstraction should the Verilog/VHDL output of a .NET IL synthesizer be produced?
- We probably over-synthesize.

Next Steps

- Consider a series of concurrency constructs and their meaning in hardware:
  - Transactional memory
  - Rendezvous.
  - Join patterns / chords
  - Data Parallel Descriptions
- Solve impedance mismatch with back-end tools to improve performance.


soft processor
C#

New Relevant Developments

- Separation Logic
  - What part of a program uses what part of memory when
  - A formal basis for partitioning C programs into parallel chunks
- Region Types
  - Language level support for disciplined sharing of information between concurrent processes
- Termination Proofs
- These technologies can make a radical contribution to automatic C to gates technology.

The Future is Asynchronous

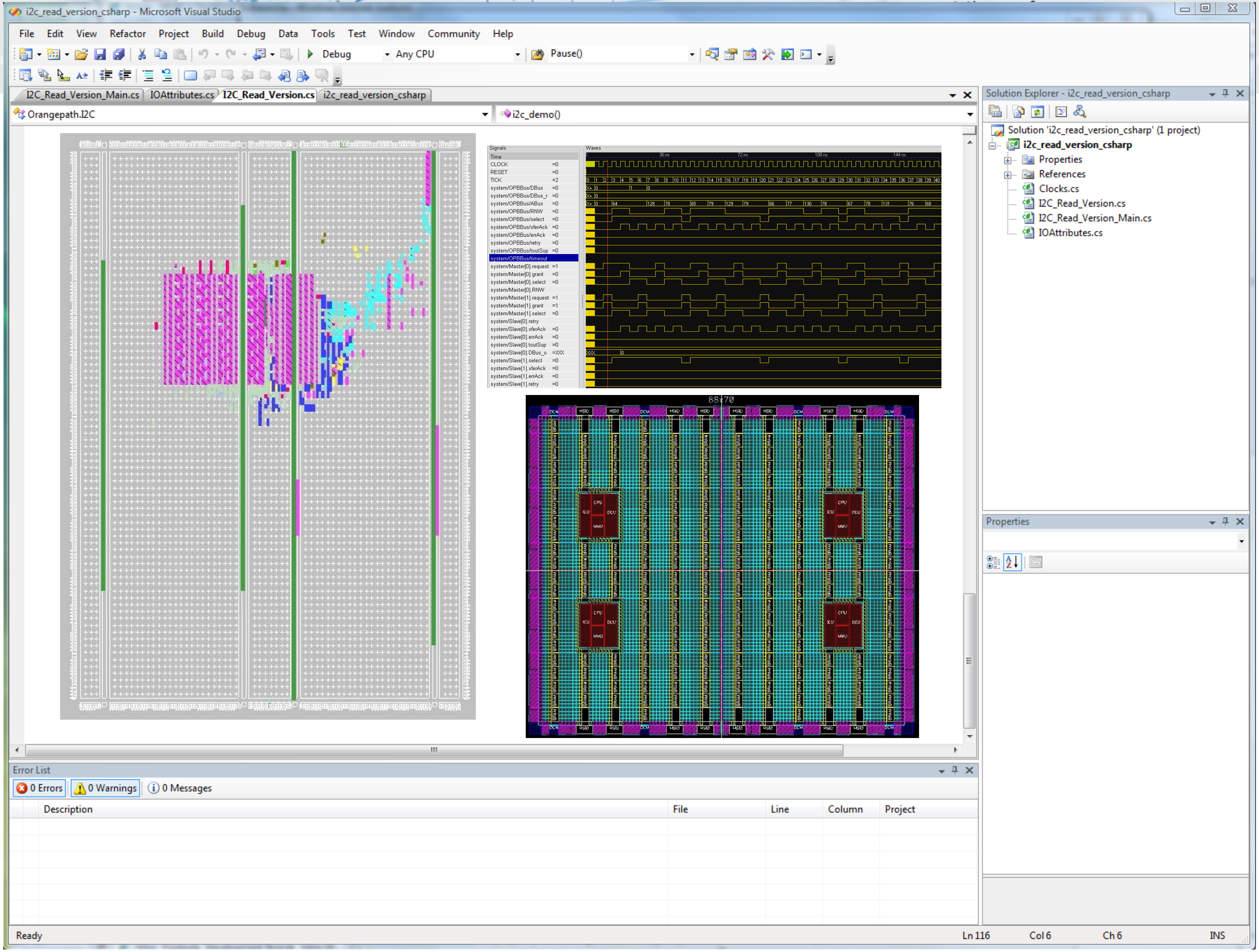
- There is no clock in my parallel program...
- Why is there a clk net in my Verilog netlist?

Summary

- Model circuits as **parallel programs**.
- Transform parallel circuit models into digital circuit implementations.
- Exploit shared memory and passage passing idioms for **co-design**.
- We don't need to invent a new language:
  - Exploit rich **existing knowledge of concurrent programming**.
- Initial small step towards programming models and techniques for **manycore systems**.
- More information about Kiwi synthesis at http://research.microsoft.com/~satnams

Conclusions

- Design techniques based on **conventional HDLs will not work**.
- Parallel hardware technology can be exploited by developers by exploiting **concurrent** and **parallel** programming models.
- **Formal models** of computation and composition essential.
- **Verification** and design intertwined from the start.



i2c_read_version_csharp - Microsoft Visual Studio

File Edit View Refactor Project Build Debug Data Tools Test Window Community Help

Debug | Any CPU | Pause()

I2C_Read_Version_Main.cs | IOAttributes.cs | I2C_Read_Version.cs | i2c_read_version_csharp

Orangepath.I2C | i2c_demo()

```
Signals
Time
CLOCK                        =0
RESET                        =0
TICK                         =2
system/OPBBus/DBus           =0
system/OPBBus/DBus_r         =0
system/OPBBus/ABus           =0
system/OPBBus/RNW            =0
system/OPBBus/select         =0
system/OPBBus/xferAck        =0
system/OPBBus/errAck         =0
system/OPBBus/retry          =0
system/OPBBus/toutSup        =0
system/OPBBus/timeout
system/Master[0].request     =1
system/Master[0].grant       =0
system/Master[0].select      =0
system/Master[0].RNW
system/Master[1].request     =1
system/Master[1].grant       =1
system/Master[1].select      =0
system/Slave[0].retry
system/Slave[0].xferAck      =0
system/Slave[0].errAck       =0
system/Slave[0].toutSup      =0
system/Slave[0].DBus_o       =XXX
system/Slave[1].select       =0
system/Slave[1].xferAck      =0
system/Slave[1].errAck       =0
system/Slave[1].retry        =0
```

Solution Explorer - i2c_read_version_csharp

- Solution 'i2c_read_version_csharp' (1 project)
  - i2c_read_version_csharp
    - Properties
    - References
    - Clocks.cs
    - I2C_Read_Version.cs
    - I2C_Read_Version_Main.cs
    - IOAttributes.cs

Properties

Error List

0 Errors | 0 Warnings | 0 Messages

Description	File	Line	Column	Project

Ready   Ln 116   Col 6   Ch 6   INS

Co-Design

- FPGAs can now interface directly to Intel's new front-side bus.
- Memory can be **shared** with the processor(s).
- Hardware processes can **communicate** and **synchronize** with software via shared memory.
- A Kiwi-style approach makes it feasible to provide a unified **co-design** environment.
- Imagine the **applications**:
  - Accelerating web search functions.
  - Accelerating image processing.
  - Accelerating SAT solvers and model checkers.

Key Points

- This is **early stage** work on compiling parallel C# and F# programs into parallel hardware.
- Important because future processors will be **heterogeneous** and we need to find ways to model and program multi-core CPUs, GPUs, FPGAs etc.
- Previous work has had some success with compiling **sequential** programs into hardware.
- Our hypothesis: it's much better to try and produce **parallel** hardware from **parallel** programs.
- Our approach involves compiling **.NET concurrency constructs** into gates.

Benefits of .NET

- We can exploit existing compilers, tools, debuggers for our hardware designs.
- We use custom attributes to mark up input ports, output ports, clock signals etc.
- We use existing concurrency constructs and re-map their semantics to appropriate hardware idioms.
- We try to always have a sensible piece of concurrent software that corresponds to each synthesized circuit.

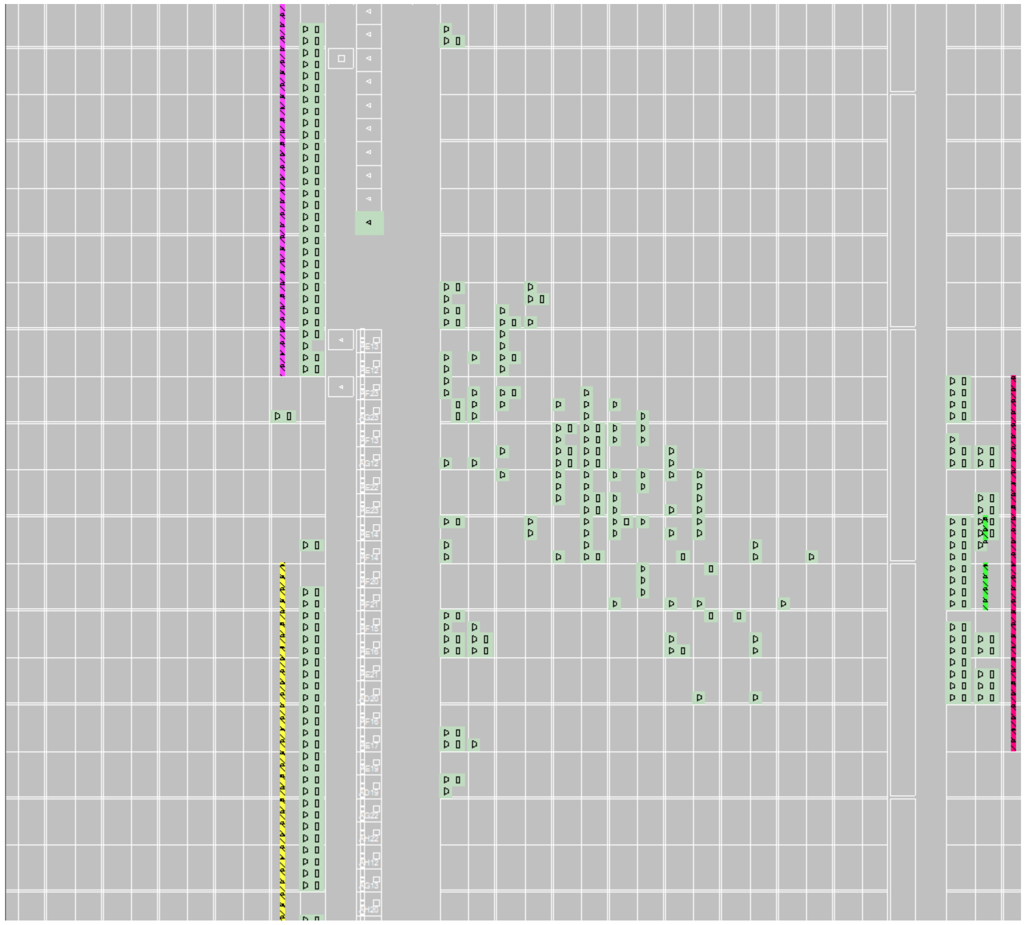
I2C Control

```
private static void SendDeviceID()
        {
            Console.WriteLine("Sending device ID");
            // Send out 7-bit device ID 0x76
            int deviceID = 0x76;

            for (int i = 7; i > 0; i--)
            {
                scl = false; sda_out = (deviceID & 64) != 0; Kiwi.Pause();
                // Set it i-th bit of the device ID
                scl = true; Kiwi.Pause(); // Pulse SCL
                scl = false; deviceID = deviceID << 1; Kiwi.Pause();
            }
        }
```

Generated Verilog

```
module i2c_demo(clk, reset, I2CTest_I2C_scl, I2CTest_I2C_sda);

   input clk;
   input reset;
   reg i2c_demo_CS$4$0000;
   reg I2CTest_I2C_SendDeviceID_CS$4$0000;
   reg I2CTest_I2C_SendDeviceID_second_CS$4$0000;
   reg I2CTest_I2C_ProcessACK_ack1;
   reg I2CTest_I2C_ProcessACK_fourth_ack1;
   reg I2CTest_I2C_ProcessACK_second_ack1;
   reg I2CTest_I2C_ProcessACK_third_ack1;
   integer I2CTest_I2C_SendDeviceID_deviceID;
   integer I2CTest_I2C_SendDeviceID_second_deviceID;
   integer I2CTest_I2C_SendDeviceID_i;
   integer i2c_demo_i;
   integer I2CTest_I2C_SendDeviceID_second_i;
   integer i2c_demo_inBit;
   integer i2c_demo_registerID;
   output I2CTest_I2C_scl;
   output I2CTest_I2C_sda;
```

Generated FPGA Circuit



FSM Synthesis (1)

- Resulting machine simulated with all inputs set to don’t care
  - Discovery of compile time constants
  - Constructor code must not depend on runtime inputs
- No stack or dynamic allocation.

FSM Synthesis (2)

- The next stage produces an array of machines, one per thread with the following kinds of statements:
  - Assign
  - Conditional branch
  - Exit
  - Calls to certain built in functions including:
    - Atomic test and set
    - “printf” for debugging
    - Barrier
    - All the usual arithmetic and logical operations in .NET
    - String handling

FSM Synthesis (3)

- Final output form is stylised such that there is no program counter and every statement operates in parallel.
- This form is readily translated into hardware level netlists and then into VHDL or Verilog for the final synthesis to gates.