Applications (2)

Peter Marwedel
Informatik 12, TU Dortmund

Gliederung

Zeitplan

- Einführung
- SystemC
  - Vorlesungen und Programmierung

3,5 Wochen

- FPGAs
  - Vorlesungen
  - VHDL-basierte Konfiguration von FPGAs mit dem XUP VII Pro Entwicklungssystem

3,5 Wochen

- Algorithmen
  - Mikroarchitektur-Synthese
  - Automatensynthese
  - Logiksynthese
  - Layoutsynthese

6 Wochen

Typical applications

- Diploma theses @ Dortmund: real-time computations
- Graphics accelerators
- Encryption/decryption
- Bio-sequence database scanning
- Network applications (e.g. network intrusion detection)
- Parallel pattern recognition in physics
- Emulation (of new hardware processors)

Survey: Use of FPGAs in Bioinformatics

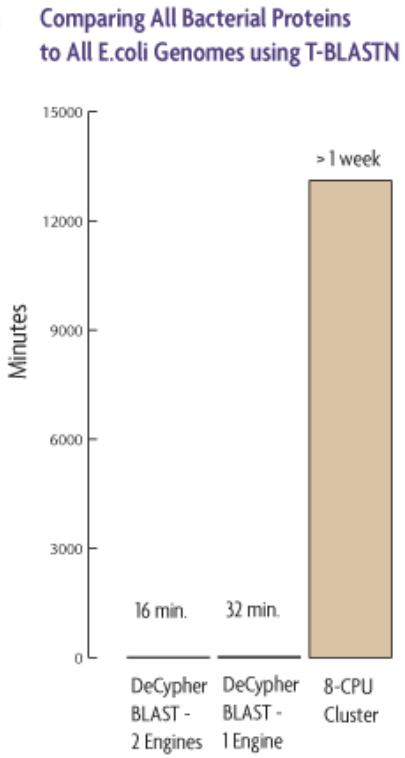
- Bioinformatics applications:
  - little I/O
  - dynamic programming
  - big lookup tables

- Ideal for FPGA

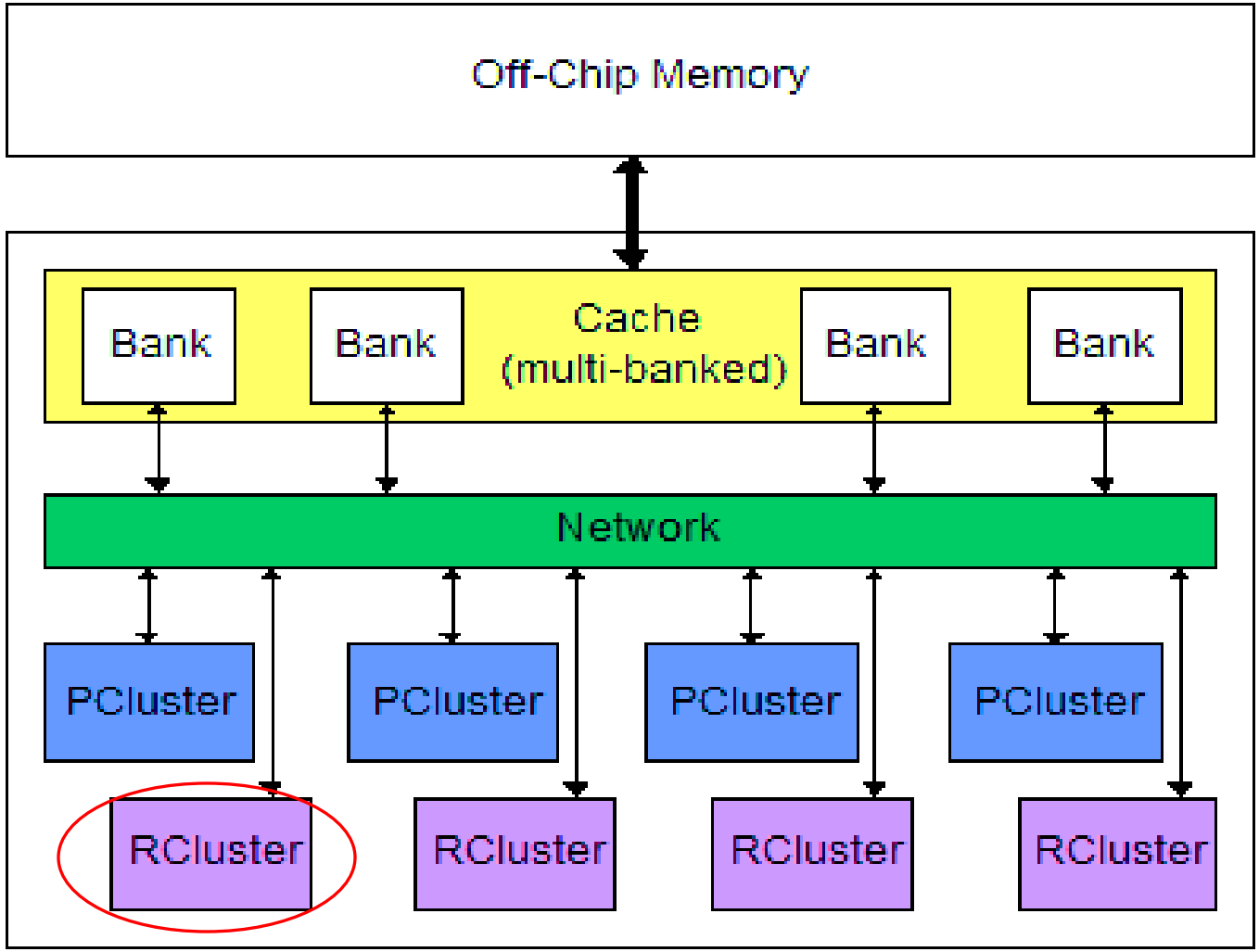
- Currently one company (TimeLogic) sells FPGA cards for BLAST and Smith-Waterman

- Cost: ~$20,000 for cards

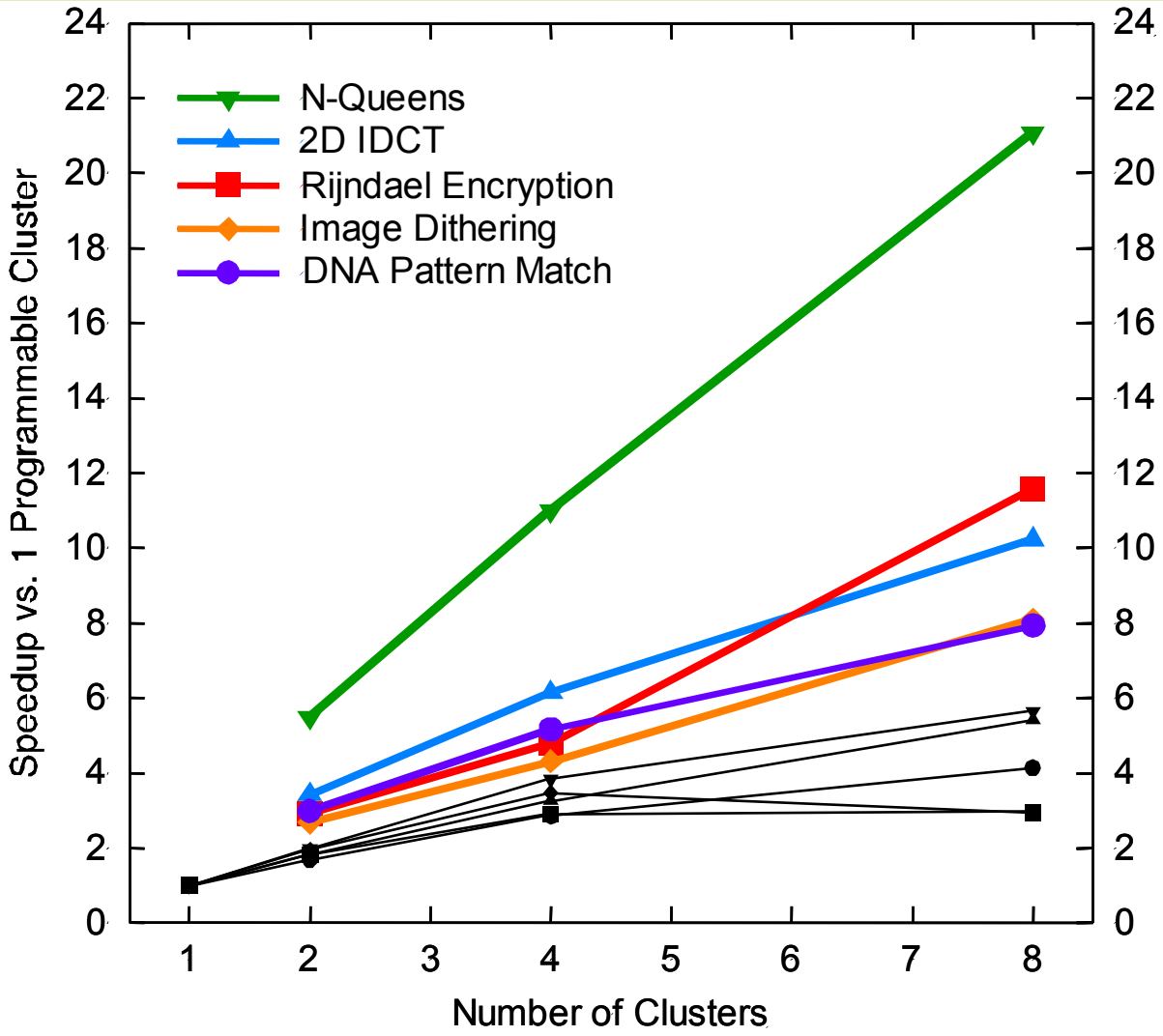
- Used routinely in biotech / pharma companies



Mid-Grain: Amalgam

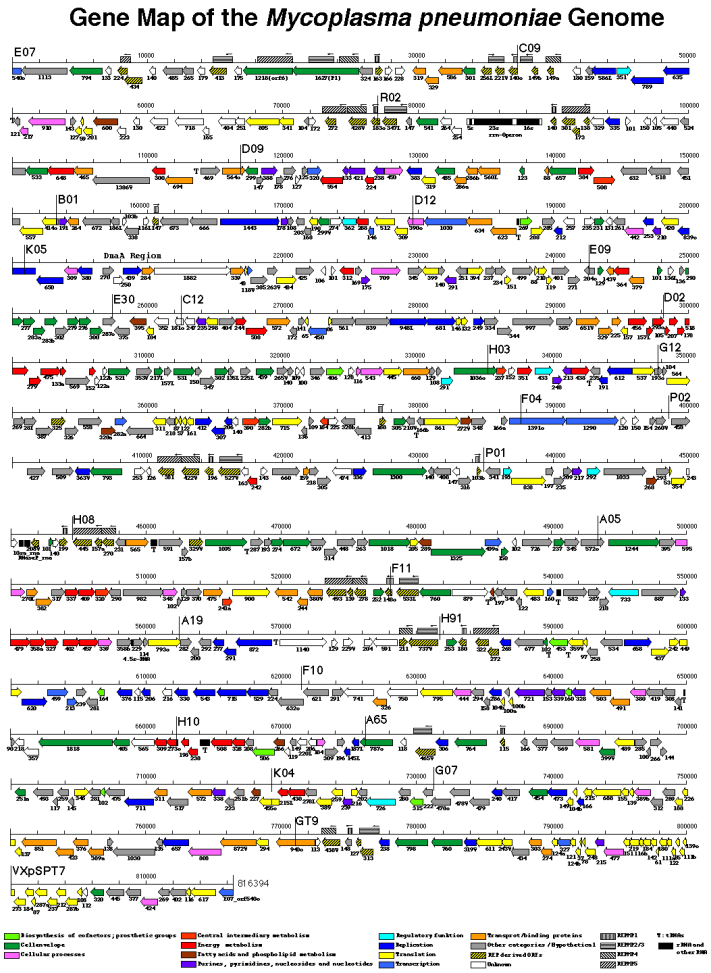


Amalgam Performance



Bioinformatics 1: Sequence Alignment

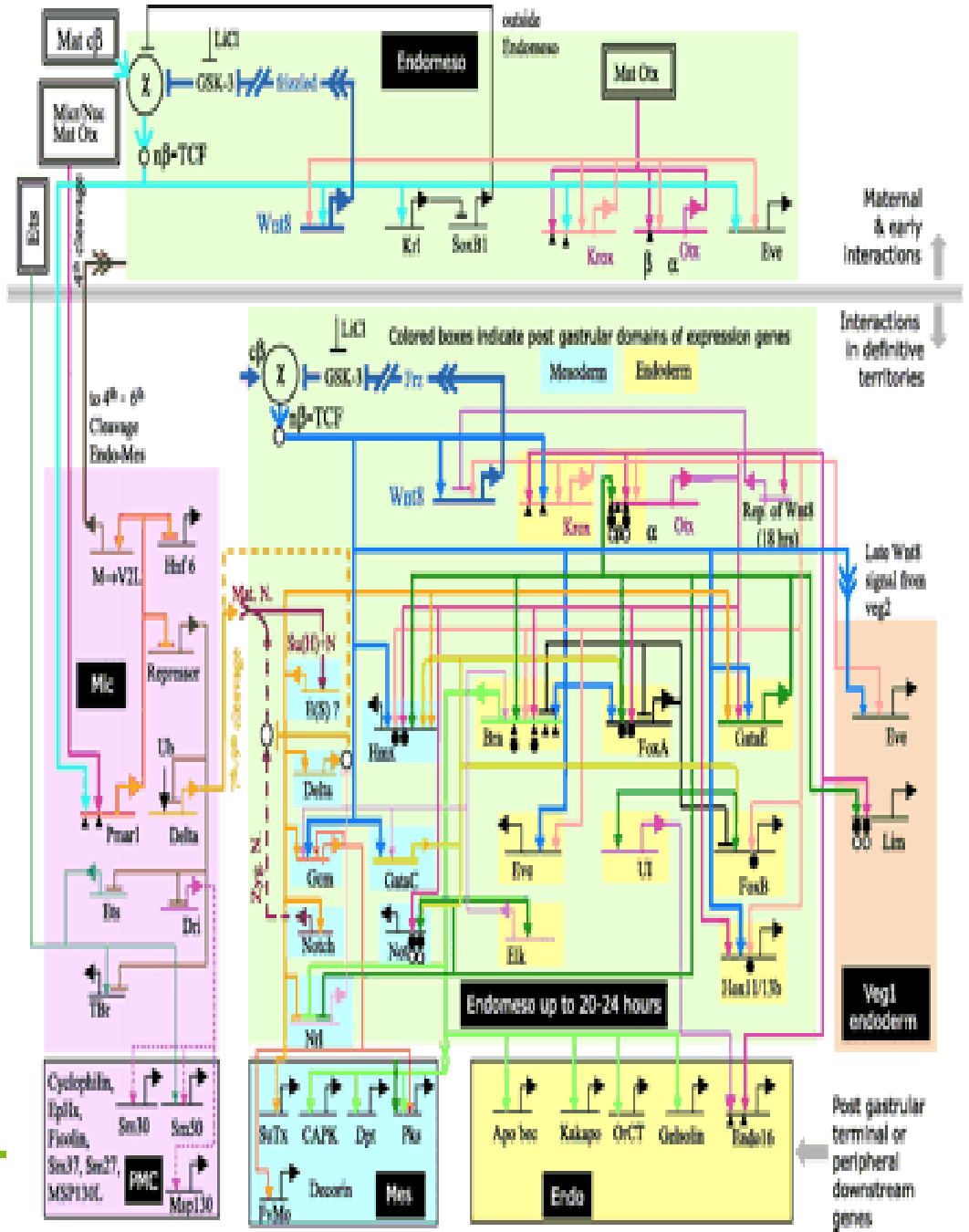
- The genome, and the proteins, are very complex and difficult to decipher:
- A genome is a string of DNA bases, A, C, G or T; from ½ million (mycoplasma) to 10 billon (lilly).
- The genome is interpreted by working out which regions of the DNA code for protein, and what the likely function of these proteins is.
- We currently do this by matching sequences to those of DNA or proteins used in previous experiments.
- *A priori* prediction of structure/function still a distant dream.



Simplified diagram of the smallest genome known, *Mycoplasma pneumoniae*

Bioinformatics 2: Reconstruction of regulatory networks

- An emerging field
- Attempts to model the complex interactions between DNA and the protein it encodes, that ultimately control the processes of life.
- Often uses analogies to circuitry. However biological data is always incomplete, sometimes very sparse.
- SANs, PetriNets, locally Mobius utilized.



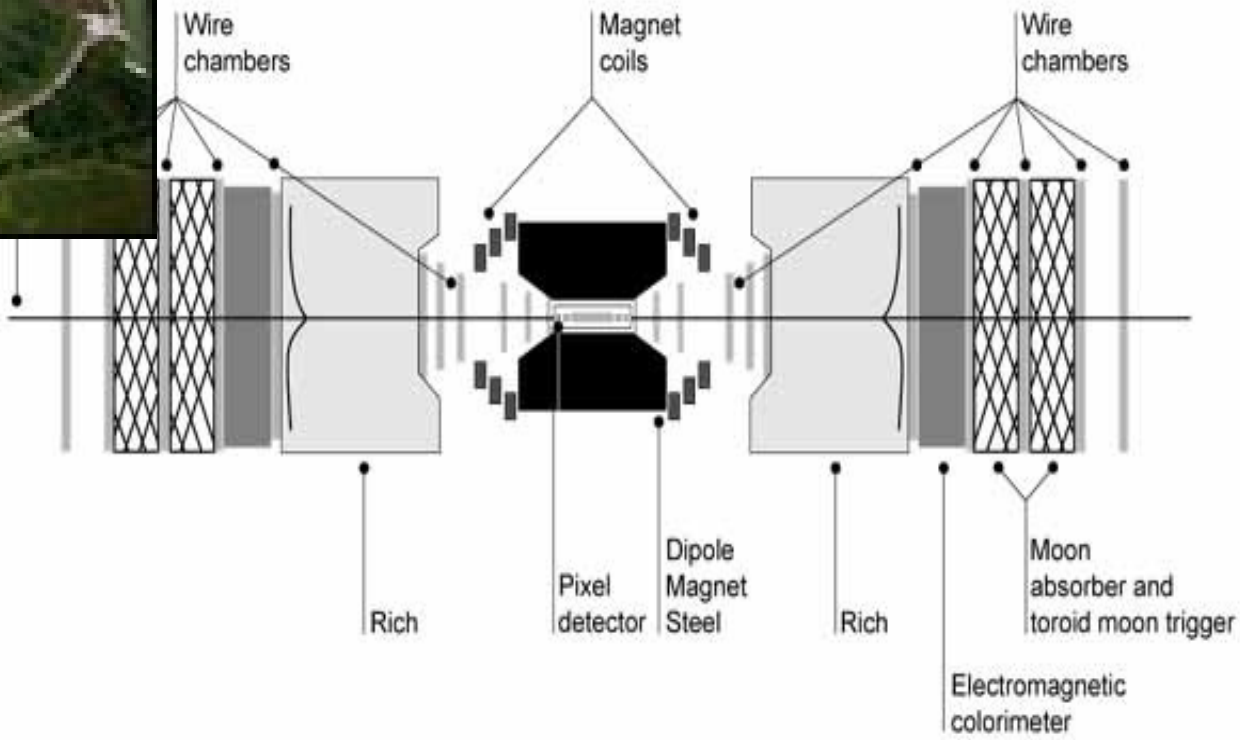
Typical applications

- Diploma theses @ Dortmund: real-time computations
- Graphics accelerators
- Encryption/decryption
- Bio-sequence database scanning
- Network applications (e.g. network intrusion detection)
- Parallel pattern recognition in physics
- Emulation (of new hardware processors)

High Energy Physics

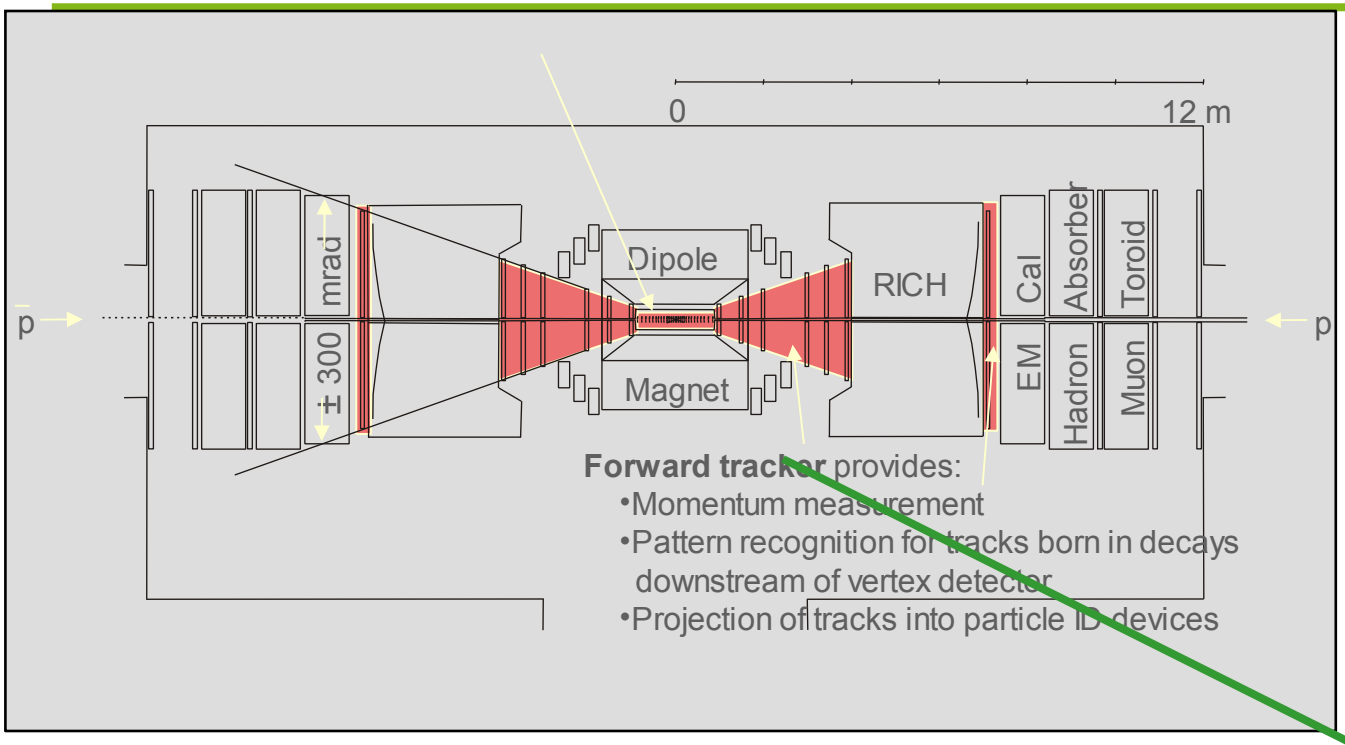
FermiLab Accelerator

BTeV Experiment

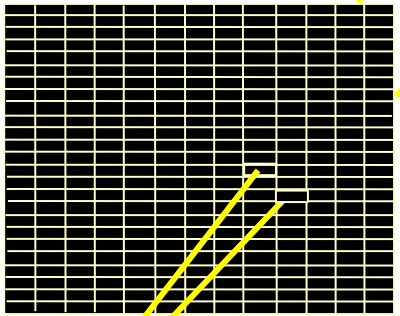


Source:
http://false2002.vanderbilt.edu/talks/Bapty.ppt

Particle Measurement



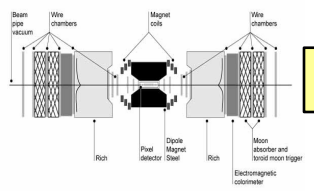
Detector Grids



Problem:

- Massive amounts of data (Terabytes/Sec)
- Determine the set of particle trajectories
- Decide if it is interesting, keep or toss
- Hardware => 2500 DSP's + 2500 PC's
- Never Fail (ok to degrade)

Trigger System
(20,000 ft. view)



Memory Queue, ~ms

Store

Pre-Process (FPGA)

1st Level (DSP)

2nd Level (PC)

~2000 Nodes

~2000 Nodes

Typical applications

- Fast Object recognition
  - Medical, security, …
  - Military
- Graphics accelerators
- Encryption/decryption
- Bio-sequence database scanning
- Network applications (e.g. network intrusion detection)
- Parallel pattern recognition in physics
- Emulation (of new hardware processors)

Source:
http://bwrc.eecs.berkeley.edu/Presentations/Retreats/Winter_Retreat_2004/Monday%20PM/Wawrzynek-HERC_BEE2v1.ppt

High-End Reconfigurable Computing

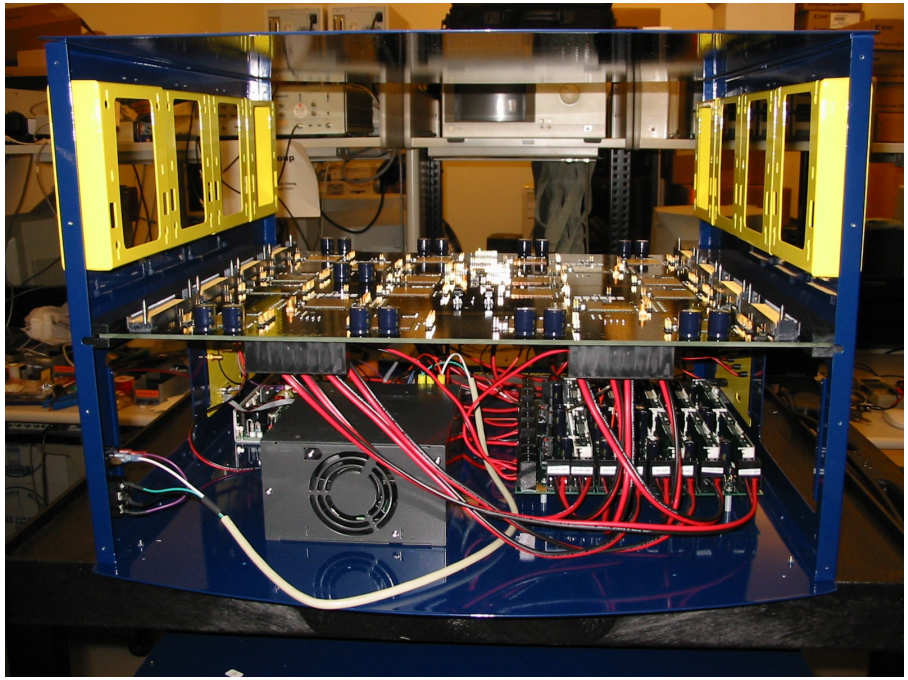
Berkeley Wireless Research Center

January 2004

John Wawrzynek, Robert W. Brodersen, Chen Chang, Vason Srini, Brian Richards

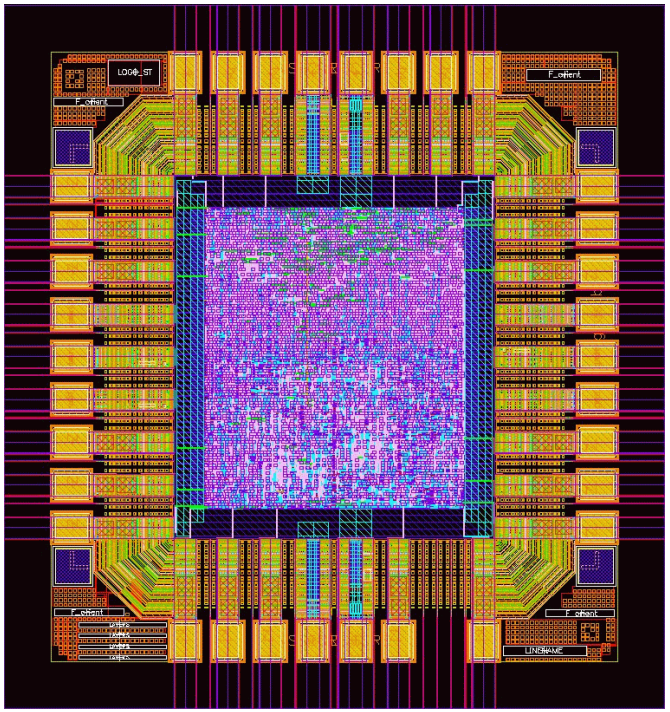
Berkeley Emulation Engine

- FPGA-based system for real-time hardware emulation:
  - Emulation speeds ***up to 60 MHz***
  - Emulation capacity of ***10 Million*** ASIC gate-equivalents, corresponding to ***600 Gops*** (16-bit adds) *(although not a logic gate emulator.)*
  - ***2400*** external parallel I/Os providing ***192 Gbps*** raw bandwidth.



Status

- Four BEE processing units built
- Three in continuous “production” use
- Supported universities
  - CMU, USC, Tampere, UMass, Stanford
- Successful tapeout of:
  - 3.2M transistor pico-radio chip
  - 1.8M transistor LDPC decoder chip
- System emulated:
  - QPSK radio transceiver
  - BCJR decoder
  - MPEG IDCT
- On-going projects
  - UWB mix-signal SOC
  - MPEG transcoder
  - Pico radio multi-node system
  - Infineon SIMD processor for SDR

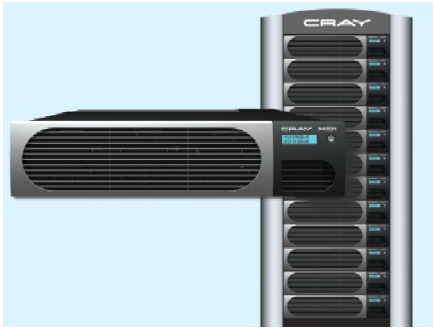


Lessons from BEE

- Simulink based tool-flow very effective FPGA programming model in DSP domain.
- **Many system emulation tasks are significant computations in their own right – high-performance emulation hardware makes for high-performance general computing.**
- ***Is this the right way to build supercomputers?***
- BEE could be scaled up with latest FPGAs and by using multiple boards ⇒ TeraBEE (B2).

Is Reconfigurable Going Mainstream?

- Cray XD1:
  - 12 Opterons per chassis
  - 6 Xilinx Virtex Pro 50 FPGAs
  - Communication network/processor



- SGI Altix 350:
  - 1-32 Itanium2s
  - 1 Xilinx XCV6000 per processor pair
  - Fast interconnect technology

- SRC:
  - Pentium 4 Xeons
  - MAPs: multiple XCV6000 FPGAs
  - X-bar

- TimeLogic, Starbridge, others…
- NCSA is exploring this technology.

Vendors other than Xilinx?

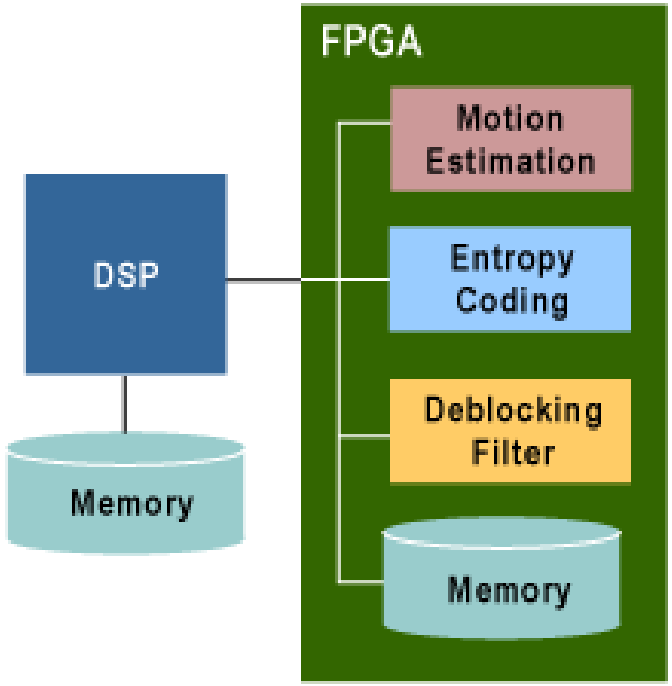
PLD=FPGA+CPLD Sales 2002

Rank	Company	Sales (M$)	Share [%]	FPGA-Share 2004* [%]
1	Xilinx	1,125	49	~55
2	Altera	712	31	~30
3	Lattice	235	10	
4	Actel	135	6	
5	Cypress	??	2	
6	Quicklogic	33	1	
7	Atmel	25		

Source: *: Altera (http://biz.yahoo.com/e/050311/altr10-k.html), other numbers from IC Insights

Altera support tools

Anwendung: Video-Dekodierung



http://www.altera.com/technology/dsp/devices/fpga/dsp-fpga_coprocessor.html

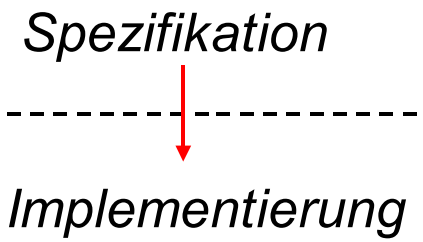
Stile der FPGA-Synthese

Peter Marwedel
TU Dortmund, Informatik 12
Germany

Synthese

Def.: **Synthese** ist das Zusammensetzen von Komponenten oder Teilen einer niedrigen (Modell-) Ebene zu einem Ganzen mit dem Ziel, ein auf einer höheren Ebene beschriebenes Verhalten zu erzielen.

[Advanced Learners Dictionary of Contemporary English].



„*correctness by construction*“

Sofern Synthesemethoden garantiert korrekte Entwürfe erstellen, brauchen diese nicht mehr per Simulation überprüft zu werden.
Die Korrektheit der Methoden ist allerdings bisher nur in wenigen Fällen nachgewiesen worden. Simulationen sind daher weiterhin sinnvoll, aber weniger wichtig.

Ideal: für alle Entwurfsaufgaben gute Syntheseverfahren.

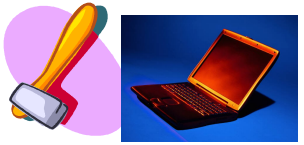
Ideal und Wirklichkeit

Für alle Entwurfsaufgaben gute Synthese: kaum erreichbar.

Nach H. de Man liegt ist dies prinzipiell begründet:

- EDA-Hersteller mit der Pflege vorhandener Software beschäftigt.
- Anwender wollen aufgrund der für Wartung aufzuwendenden Kosten keine Werkzeuge selbst entwickeln und pflegen.
- Forschungsinstitute müssen Probleme erfahren, Lösungen erarbeiten & mit EDA-Firmen in Produkte umsetzen. Dissertationen: ca. 5 Jahre ☞ vom Auftauchen des Problems bis zur breiten Markteinführung knapp 10 Jahre.

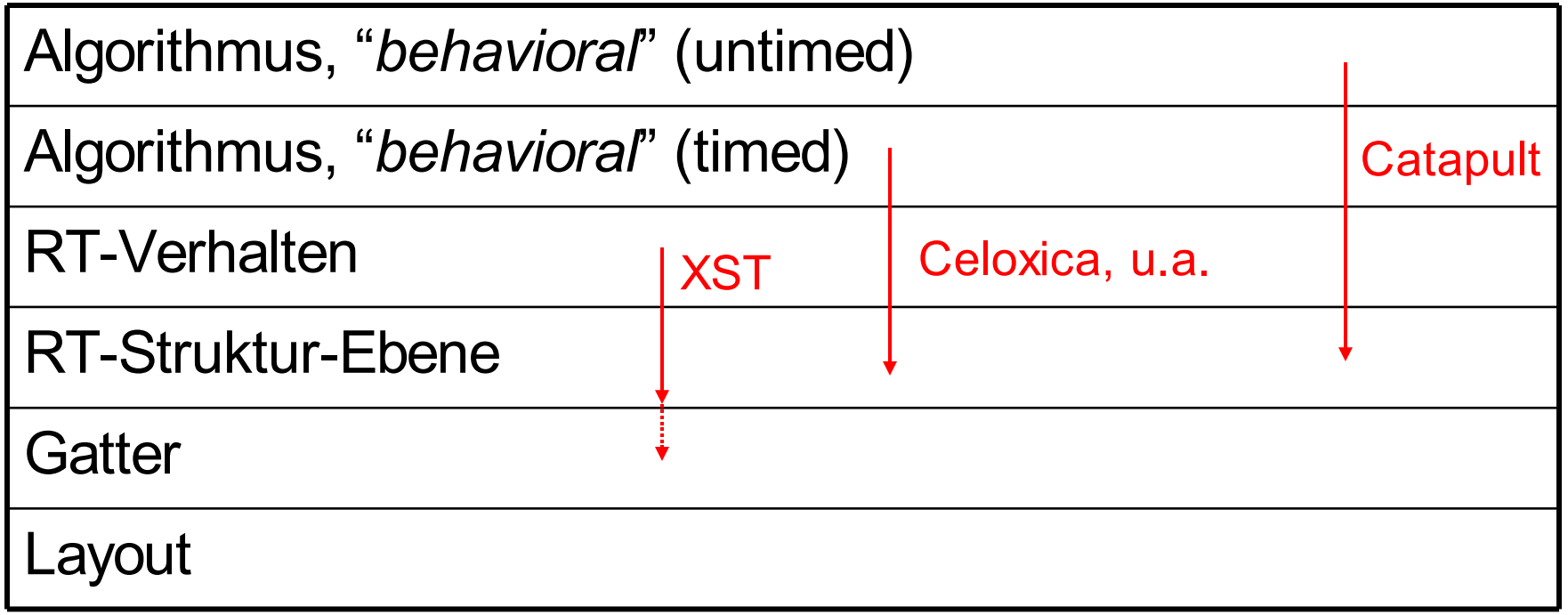
de Man: *tomorrow's tools solve yesterday's problems.*



Termingerechter Entwurf ohne Synthese nicht möglich.

Formen der Synthese

Unterschiedliche Ziele und Quellen;
mögliche Darstellung:



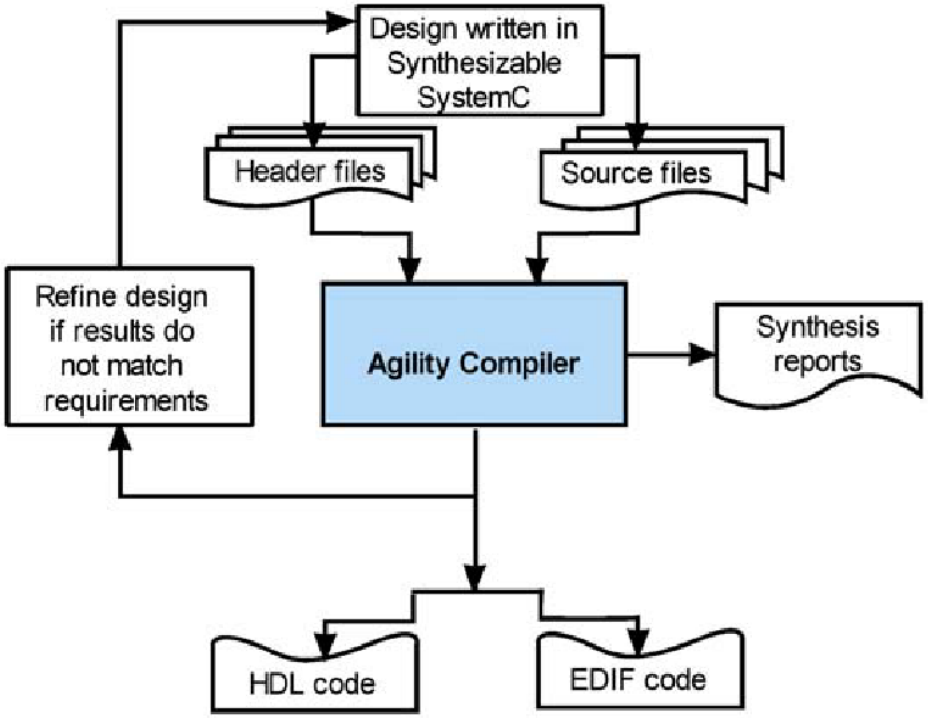
Nachteile einer Spezifikation auf RT-Ebene

Nachteile einer Spezifikation wie bei XST

- Man gibt die Schaltungsblöcke bereits vor.
- Man muss sich überlegen, was in welchem Taktzyklus stattfindet.
- Die Vorteile von VHDL, Algorithmen beschreiben zu können, werden kaum genutzt.
- Liegt die ursprüngliche Spezifikation on C oder SystemC vor, so muss die VHDL-Beschreibung manuell neu erstellt werden.
  - Es können sich viele Fehler einschleichen.
  - Der Aufwand ist enorm.

Synthese von „*timed behavioral descriptions*“

- …
- Co-Centric
- Celoxica: *DK design suite* (Handel-C), Agility (SystemC)



SystemC Synthese Ablauf

1. Produce a simulation model of your design in SystemC/C++.
2. Produce a testbench (also in SystemC/C++).
3. Simulate and test the model.
4. Refine the simulation design into synthesizable code.
   Carry this out manually. Good practice to split your design into header (.h) files and the corresponding implementations in .cpp files for easier recompilation.
5. Compile your design into RTL SystemC using Agility.
6. Test the synthesizable model.
7. Compile your design into HDL (EDIF, Verilog or VHDL) using Agility. Different synthesis flags and optimizations available.
8. Examine the synthesis reports.
9. If constraints not met: refine your design and repeat.
10. Once requirements are met, run Place and Route tools.

[Celoxica: Agility Compiler manual, v 1.1, 2006]

Agility code requirements

The source code and header files include:

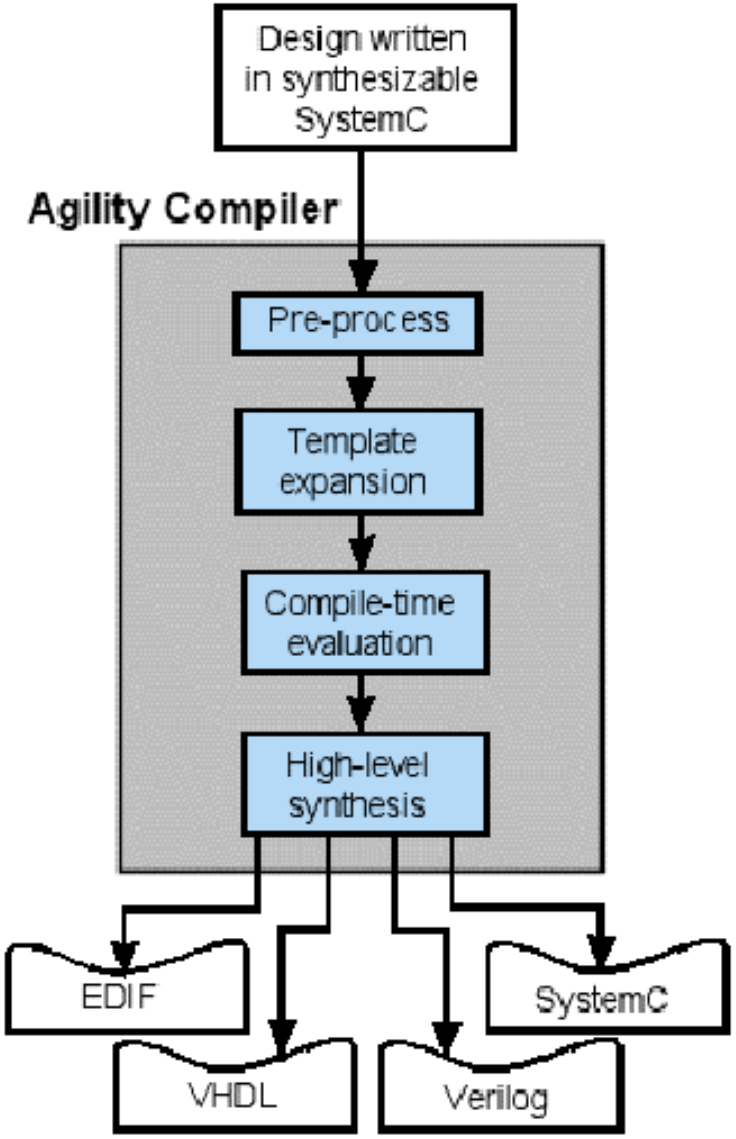
- header files appropriate for simulation or synthesis (as specified in a preprocessor directive)
- an **ag_main** function, used for synthesis, comprising an instantiation of a top-level module and any user constraints
- an **sc_main** function, called for simulation, and comprising an instantiation of a top-level module & user constraints
- a top-level module, instantiating modules to be synthesized
- modules, threads, methods describing the logic of your design
- Signals, FIFOs, channels describing communication

[© Celoxica]

Supported types

Type	*Available for compile-time evaluation*	*Available for synthesis*
`char`	ok	ok
`short int`		
`int`		
`long int`		
`long long int`		
`float`	ok	no
`double`		
`long double`		
`bool`	ok	ok
Pointers	ok	If compile-time resolvable
References	ok	If compile-time resolvable
Unions	no	no
Bit fields	no	no
Arrays	ok	Depends on contents
Structs	ok	Depends on contents
sc_ufixed	no	no
sc_int, sc_uint sc_bigint, sc_biguint, sc_fixed	ok	ok

Steps used in the Agility compiler



[© Celoxica]

Compile-time evaluation

Agility unrolls loops, resolves pointers, inlines functions, evaluates values of variables known at compile-time.

- ***Loop unrolling***
  Loop unroller unrolls loops that must be unrolled and skips over those that do not. Any loops without **wai**t() statements unrolled to produce combinational logic.

[© Celoxica]

Loop unrolling limitations

The loop unroller cannot unroll loops containing:

- **wait** statements
- loop conditions including unknown variables.

Loops containing **wait**() statements are skipped and the values of variables in their bodies are tagged as *unknown*.

If the condition of a loop that must be unrolled contains a variable with an *unknown* value, it causes an error.

Example

```
int k = 0;
for (int i=0; i<10; i++) { k = k + 5; wait(); }
for (int j=0;j<k; j++)    { ... code without waits }
```

1st loop is skipped due to wait statement ☞ k = *unknown*.
2nd loop cannot be unrolled.

[© Celoxica]

Compile-time recursive definitions

Synthesis of recursive definitions whose arguments can be determined at compile time.
Be sure recursive definitions terminate!

Example: Fibonacci function

```
int Fibonacci( int n )
{ if (n == 0) // check for termination
  return 0;
  if (n == 1) // check for termination
  return 1;
  if (n > 1) return ( Fibonacci(n-1) + Fibonacci(n-2) );
 }
```

Call to Fibonacci(6) returns 8.

[© Celoxica]

Function inlining

Agility inlines all function definitions.

Example:

```
int multiply_and_accumulate(int x, int y, int z)
{
return (x * y + z);
}
```

would be replaced by function body (suitably instantiated).
For 2 calls to the function within the same clock cycle,
☞ 2 multipliers, 2 adders would be introduced.

Effect on the overall circuit size depends on the sharing that Agility may impose.

[© Celoxica]

Pointer resolution

Agility can synthesize pointers that are resolved at compile-time i.e. pointers that are not dynamically changed during the execution of the program.

Example:

```
SC_MODULE(MyModule)
{ DaughterModule * DaughterA;
...
SC_CTOR(MyModule)
{ DaughterA = new DaughterModule();
...
}}
```

[© Celoxica]

High-level synthesis: Register inference

Registers are inferred from the use of variables.

Example:

```
int i = 1, j = 2, k;
for (;;)
  {k = i + j; j++; i = k; // Clock cycle 1
  wait();
  i++;                    // Clock cycle 2
  wait();
  }
```

- k only set and read on the same clock cycle and does not persist to another clock cycle: k is a wire.
- i is set and read in every clock cycle: i is a register.
- j is read in 1st clock cycle and this value is read in the next execution of the loop: j is a register.

[© Celoxica]

Fine-grained logic sharing

Logic can be shared within a thread if ***resource constraints exist***. Constraints can be defined for:

- multipliers, adders, subtractors, relational operators.

Agility can share operators when:

- Operations in different branches of conditional block
- Operations in different clock cycles

Example:

```
a = b*c;
wait();
d = e*f;
wait();
if (Condition) a = b*c; else d = e*f;
```

single multiplier

[© Celoxica]

Special Case

It is not always possible to share entire expressions to produce the minimum hardware.

Example:

```
a = (b * c) + d; // multiply followed by add
wait();
e = f * (g + h); // add followed by multiply
```

The order of evaluation of the operations is different.

If a user placed a constraint of 1 multiplier and 1 adder on this code, Agility could not satisfy these constraints (it does not add the necessary multiplexers and control state logic).

Instead, Agility would automatically add an extra resource for an operation.

[© Celoxica]

Summary

Applications (2)

- Bio-sequence database scanning
- Parallel pattern recognition in physics
- Emulation (of new hardware processors)

General view on synthesis

- Definition of synthesis
- Different source and target levels of abstractions
- Synthesis from SystemC or C slowly becoming available
- Restrictions for Celoxica-Agility:
  - Using *timed behaviors …*