




High-level language synthesis overview

Junsong Liao

- 
- Introduction
 - High Level Language
 - Synthesis Methodologies
 - Performance Evaluation
 - Conclusion and future work
 - Reference



Introduction

- Synthesis is the process of converting a digital design written in a hardware description language into a low-level implementation consisting of primitive logic gates.
- Today, synthesis is at RTL level.



Potential benefits for high level synthesis

- Small code size
- Faster Modeling, Simulation and Verification
=>Shorter design time
- High reusability module: change performance, Clock, functionality and communication protocol is easier than RTL module.
- Exploring design space



NEW drive forces

- Time to market is sometime outweigh performance
- System increase in size and complexity
- Availability of high performance FPGA

- Two groups people are pushing the development of HL synthesis:
 - existing HW designer: expertise in HW system
 - SW designer and application developer: little knowledge of HW, but lots of experience in system



Elements of HL synthesis

- High level synthesis include two part:
 - Description Language
 - Synthesis Methodology



HL Description Language Requirement

■ Concurrency

- The explicit description of concurrency is essential to the optimum description of parallel algorithms, deterministic synthesis and verification. It enables designer control over the most critical areas of design.

■ Timing

- Timing is essential for Synthesis to implement clocked processes for hardware and define the clocks, registers and signals.

■ Data Types

- Both abstract data type and low level hw data type are needed

■ Communications

- Method for communications between functional blocks, I/O and on-chip resources (potentially between clock domain)

Migration of languages

- Tradition high level language, such as C/C++, have no :
 - notion of time
 - No event sequencing
 - Concurrency
 - But H/W is inherently concurrent
 - H/W Data Types
 - No 'Z' value for tri-state buses

- C type HW description language added new features:
 - HW timing
 - Concurrency
 - Structure



Migration of languages

- C-based languages, coupled with block-based approaches are the most popular method of describing ESL design and Transaction Level Models (TLM).
- Handel-C and SystemC are two popular C-based languages deployed in ESL design and synthesis.

HardwareC --- *Constraint-based*

- The constraints approach uses labels or tags inside the code to direct the synthesis. It extends C with a notion of concurrent processes, message passing, timing constraints via tagging, resource constraints, explicit instantiation of models, and template models.

```
constraint maxtime from label1 to label3 = 10 cycles;
constraint delay of label2 = 5 cycles;
label1:
A = read(X);
Y = A + 1;
label2:
Z = Y * Y;
label3:
```

Figure 3

Handel-C

*---Explicit control,
rules-based timing and additions-based*

- Handel-C, based on the ANSI-C standard, is a mature, high-level language that adds a minimum of easy-to-understand hardware-oriented constructs for design implementation
- the concurrency is explicit using the par (parallel) statement. Timing is controlled by rules within the language and there are additional types and communications (e.g. the chan (channel) statement).



Handel-C Example Code

```
set clock = external "clk";
void main()
{
    ...
    while(1) par
    {
        ...
        process();
    }
}

void process()
{
    unsigned W A, B, C;

    while(1) par
    {
        ...
        Multiply(A, B, &C);
        ...
    }
}
```

Handel-C Example Code

```
void Multiply(unsigned W A,
              unsigned W B, unsigned W *C)
{
    static unsigned W a[W], b[W], c[W];
    par(
        a[0] = A;
        b[0] = B;
        c[0] = a[0][0] == 0 ? 0 : b[0];
        par (i = 1; i < W; i++)
        {
            a[i] = a[i-1] >> 1;
            b[i] = b[i-1] << 1;
            c[i] = c[i-1] + (a[i][0] == 0 ? 0 :
                            b[i]);
        }
        *C = c[W-1];
    )
}
```

Pipelined

SystemC -----Explicit control and C++ -based

- SystemC is a library of C++ classes, global functions, data types and a simulation kernel that can be used for creating cycle-accurate simulators of hardware architecture.
- The language provides hardware-oriented constructs within the context of C++ as a class library implemented in standard C++.



SystemC levels of abstraction

- There are currently 3 broad categories of classes, which make up levels of abstraction:
 - 1. RTL: The RTL classes make up the bulk of SystemC, and implement a modular structure, process concurrency, and bit-accurate data types. This layer implements all of the features that are normally found in the RTL subset of an HDL (e.g. Verilog).
 - 2. Communication: The communication classes implement data transmission and synchronization protocols which are built on top of the basic process concurrency controls of the RTL layer. These classes are typified by channels which pass data through module ports with a handshake.
 - 3. Verification: Verification classes make up a set of abstractions which are used for creating test benches. A test bench is really just a (more or less) abstract model of the design under test's environment. Verification classes provide features for randomization, data logging, and protocol matching.

SystemC Example Code

```
SC_MODULE(Multiplier)
{
    ...
    SC_CTOR(Multiplier)
    {
        SC_THREAD( process );
        sensitive_pos << clock;
    }
    ...
};

void Multiplier::process()
{
    sc_uint<W> A, B, C;
    ...
    wait();

    while(1)
    {
        ...
        Multiply<W>(A, B, &C);
        ...
        wait();
    }
}

write(Z);
```


SystemC Example

```
template<int W>
void Multiplier::Multiply(sc_uint<W> A, sc_uint<W> B, sc_uint<W> *C)
{
    static sc_uint<W> i, a[W], b[W], c[W];

    a[0] = A;
    b[0] = B;
    c[0] = a[0][0] == 0 ? 0 : b[0];
    for(i = W-1; i >= 1; i--)
    {
        a[i] = a[i-1] >> 1;
        b[i] = b[i-1] << 1;
        c[i] = c[i-1] + (a[i][0]==0 ? 0 :
                        b[i]);
    }
    *C = c[W-1];
}
```

Pipelined

The multiplier template is intended to be used inside a thread, or a clock thread that is sensitive to a clock edge. The approach used in this example is an explicit description of concurrency. There are explicit clocks in the timing described using wait statements, and the SystemC data types `sc_int` and `sc_uint` are used.

What is SystemVerilog?

- ◆ SystemVerilog is an extension of the IEEE 1364 Verilog-2001 standard
 - ◆ Adds C and C++ language constructs to Verilog
 - ◆ Adds interfaces to Verilog
 - ◆ Adds verification and testbench constructs to Verilog
 - ◆ Adds assertions to Verilog
 - ◆ Adds many other features to Verilog
- ◆ SystemVerilog is the next generation of the Verilog standard!
 - ◆ Gives Verilog a **much higher level of modeling abstraction**
 - ◆ Gives Verilog **powerful design verification capabilities**
 - ◆ Expected to be adopted by the IEEE in the next revision to the 1364 Verilog standard

SystemVerilog is an Evolution of the Verilog Standard



BERLAND
HDL
Engineers
Wizards

SystemVerilog

verification
modeling

assertions	mailboxes
test program blocks	semaphores
clocking domains	constrained random values
process control	direct C function calls
interfaces	dynamic processes
nested hierarchy	2-state modeling
unrestricted ports	packed arrays
automatic port connect	array assignments
enhanced literals	enhanced event control
time values and units	unique/priority case/if
specialized procedures	root name space access

from C / C++

classes	dynamic arrays	
inheritance	associative arrays	
strings	references	
int	globals	break
shortint	enum	continue
longint	typedef	return
byte	structures	do-while
shortreal	unions	++ -- += -= *= /=
void	casting	>> << >>= <<=
alias	const	&= = ^= %=

Verilog-2001

ANSI C style ports	standard file I/O
generate	\$value\$plusargs
localparam	`ifndef `elsif `line
constant functions	@*

(* attributes *)
configurations
memory part selects
variable part select

multi dimensional arrays
signed types
automatic
** (power operator)

Verilog-1995

modules	\$finish \$fopen \$fclose
parameters	\$display \$write
function/tasks	\$monitor
always @	`define `ifdef `else
assign	`include `timescale

initial
disable
events
wait #@
fork-join
wire reg
integer real
time
packed arrays
2D memory

begin-end
while
for forever
if-else
repeat
+ = * /
%
>> <<

SystemC vs. SystemVerilog

- Both languages support concepts such as signals, events, interfaces, and object orientation, yet each language has its distinct application focus
- SystemC
 - extends the C++ scope towards hardware
 - effective for writing abstract TL models for architectural exploration or performance modeling
 - Good for HW SW co-design, creating virtual prototypes for early software development.
- SystemVerilog
 - extends the Verilog scope to object orientation and testbenches
 - effective for designing advanced testbenches, for both RTL and TL models
 - suitable for describing the final RTL design, Good for synthesis
 - extensive tool support available



News from Industry

- SystemC is extended to analog design. an analog/mixed-signal working group has been proposed in Open SystemC Initiative (OSCI).
- SystemC transaction-level modeling (TLM) and SystemC Verification standards are going to be brought to the IEEE this year
- A proposed standard SystemVerilog synthesis subset is getting good reviews from most synthesis providers, which might overcome the portability problem when people work with multiple vendors' tools.



Problem definition of HL synthesis

- Input: High Level Language program
- Output: RTL DHL file or Gate level net-list
 - Outputting RTL file is more appealing because there are proven technologies to transfer RTL to GDS-II (Graphic Data System); and a lot of available tools to do the job.
 - Outputting RTL also enable reuse of existing IP



Behavioral vs. RTL synthesis

■ Register-transfer synthesis

- cannot figure out how to share resource, nor can it change the schedule of operations
- No schedule, No allocation

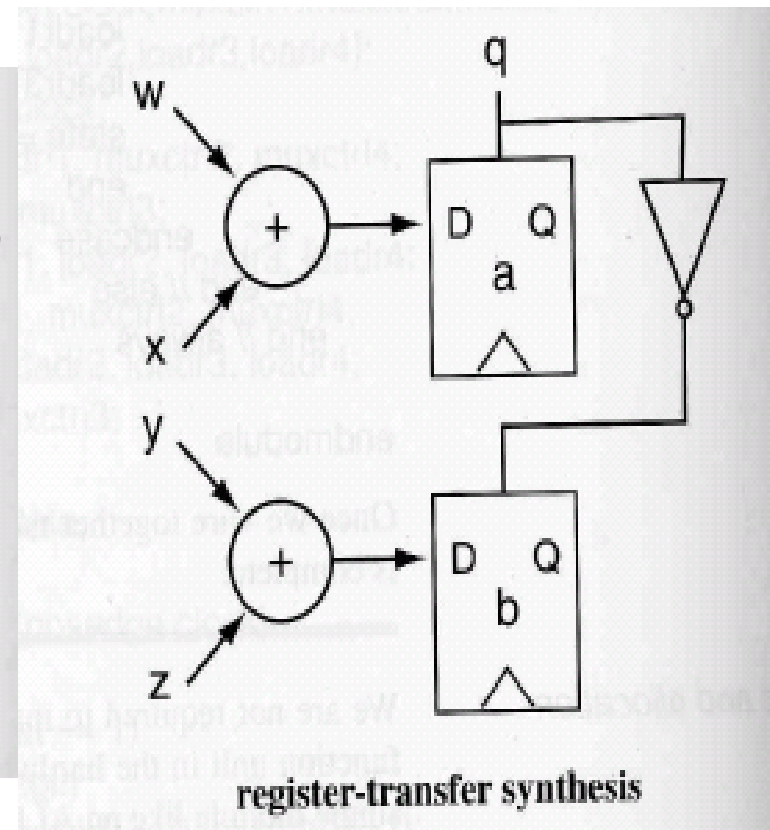
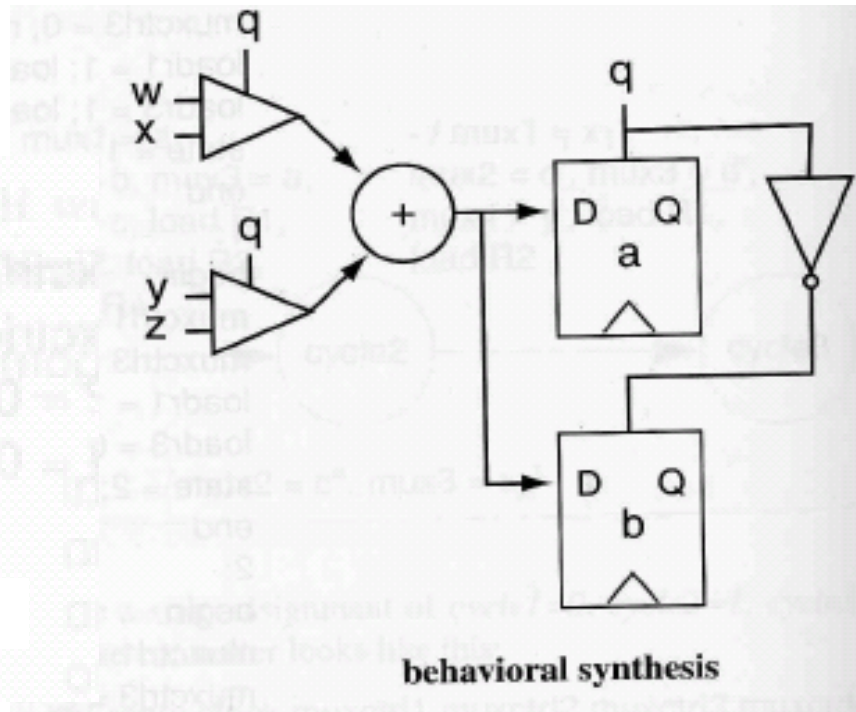
■ Behavioral synthesis

- Schedule & allocation
- Behavioral code does not specify where registers should be in the circuit or the details of the cycle timing; such decisions are left to the synthesis tool or the designer

Example

A HDL Code

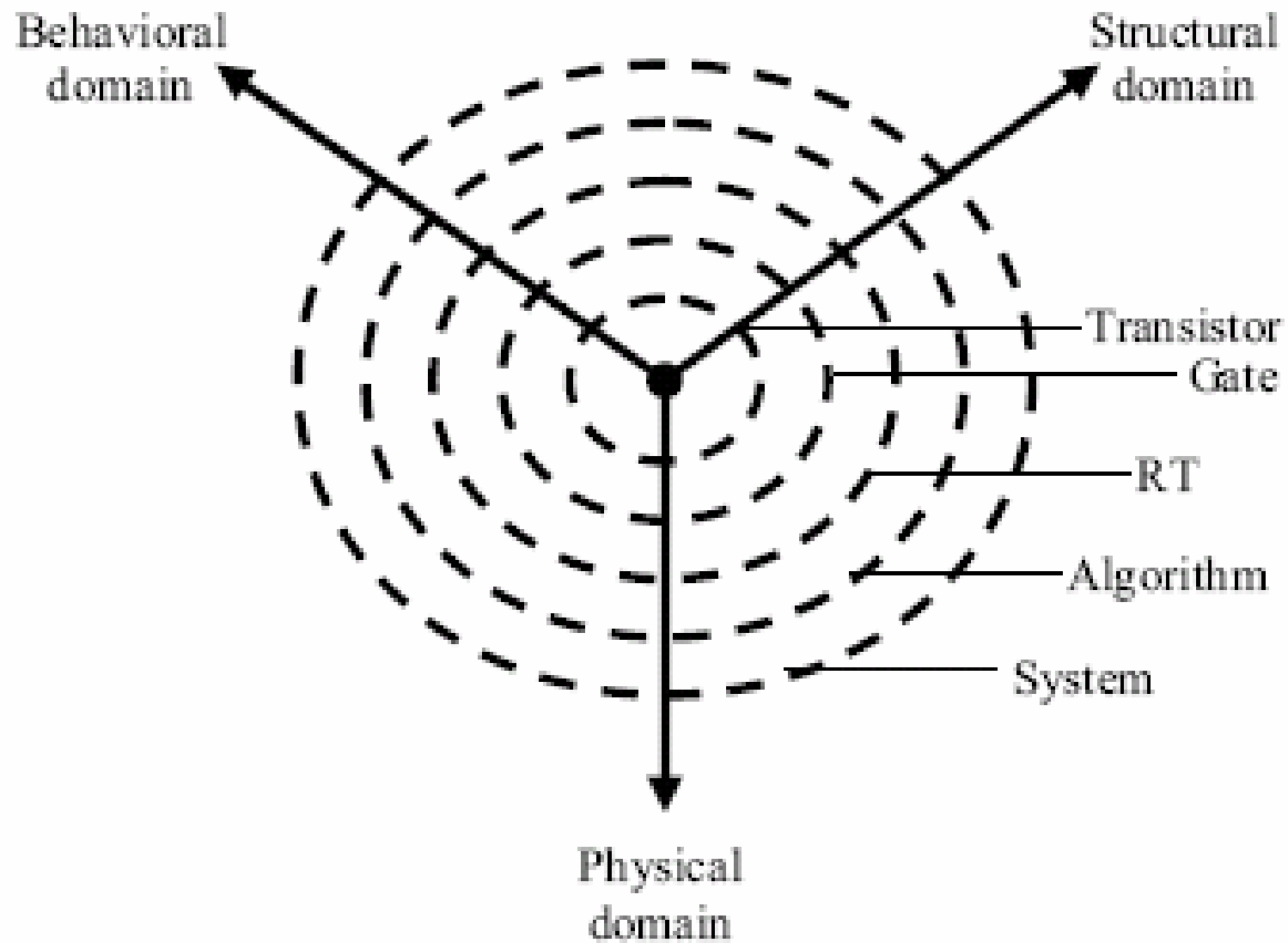
```
if (q)
  a = w + x;
else
  b = y + z;
end
```



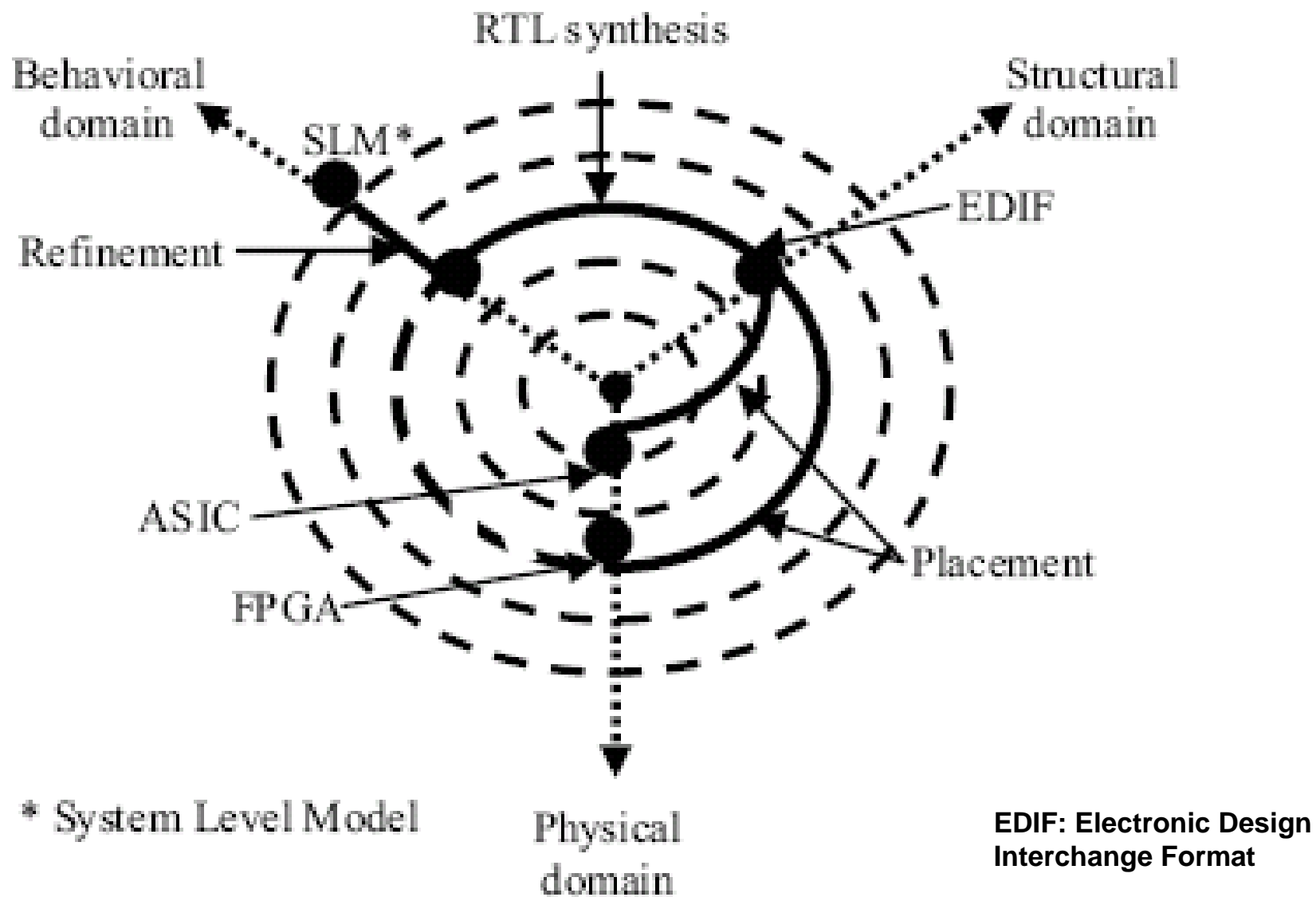
A revised HDL RTL code to share adder

```
If (q)
    Temp1=w;
    Temp2=y;
Else
    Temp1=x;
    Temp2=z;
End
Temp3=temp1+temp2;
If (q)
    a=temp3;
Else
    b= temp3;
End
```

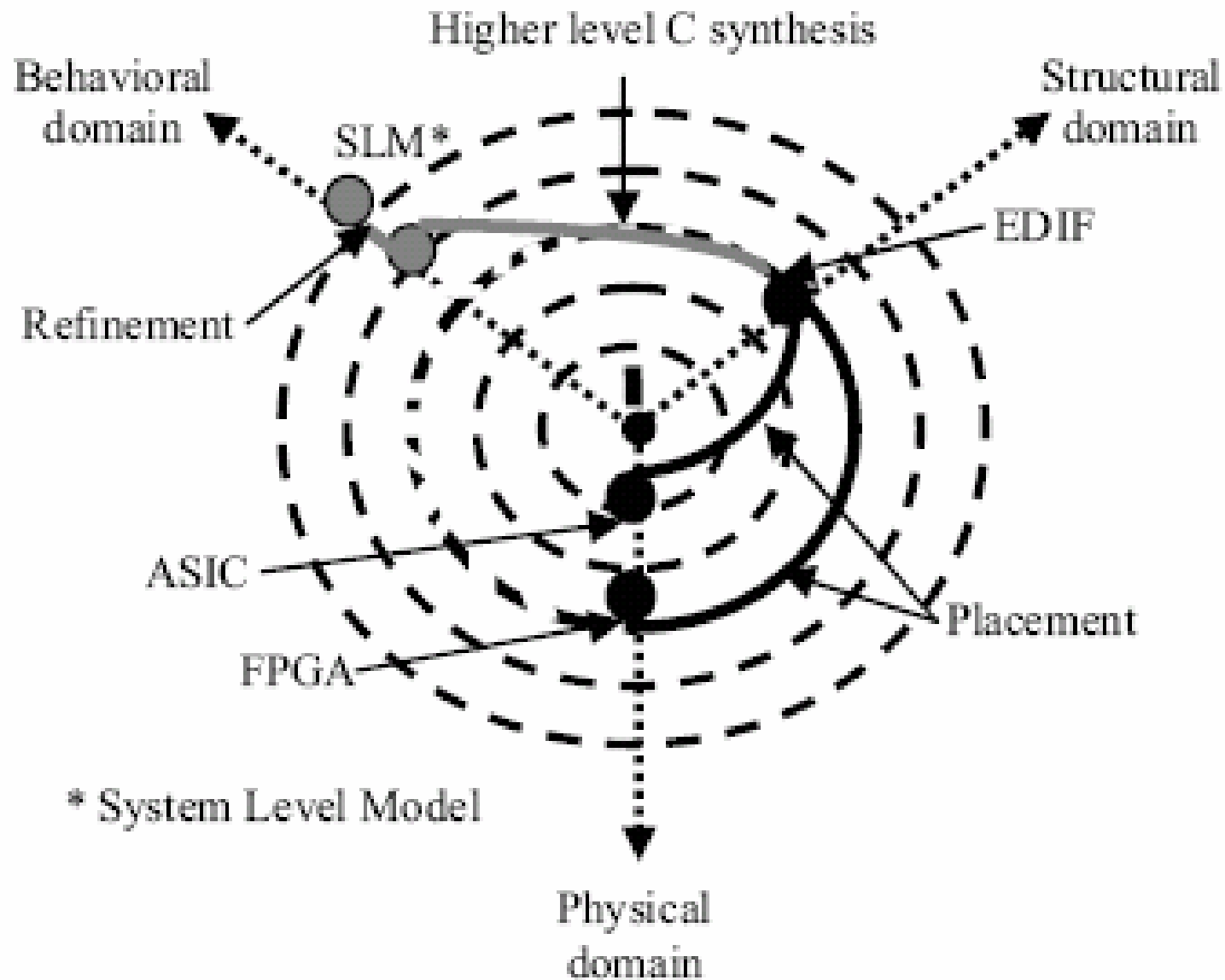
Gajski and Kuhn's Y-Chart



RTL synthesis



Behavior Synthesis





Challenges in synthesis

- Discover parallelism from sequential description
 - Naturally, HL language are sequential rather than parallel. The fully automated synthesis of optimum parallel hardware architectures from sequential source descriptions, without designer control or guidance, remains a basic challenge. The explicit description of concurrency is essential to the optimum description of parallel algorithms, deterministic synthesis and verification. It enables designer control over the most critical areas of design.
- Explore design space
 - Little information from constrains to guide synthesis
 - schedule and allocation interact with each other



Stages of the behavioral synthesis

- Various behavioral synthesis tools perform these activities in different orders using different algorithms. Some of these activities or perform them iteratively to converge on the desired solution
- ***Lexical processing***
- Lexical processing parses the high-level language source code and transforms it into an internal representation. Lexical processing for behavioral synthesis is similar to that used in conventional high-level language compilation.

Stages of the behavioral synthesis

- ***Algorithm optimization***
- Optimizations that can be performed on the algorithm itself include common subexpression elimination and constant folding.

- ***Control/Dataflow analysis***
- The result of this process is usually a Control/Dataflow Graph (CDFG). This determines which values are needed prior to computation of other values. No concept of time exists in the CDFG.



Stages of the behavioral synthesis

- ***Library processing***

- Library processing reads the available libraries and determines the functional, timing, and area characteristics of the available parts.

- ***Resource allocation***

- Resource allocation establishes a set of functional units that will be adequate to implement the design. In many behavioral synthesis systems, an initial resource allocation is performed and subsequently modified during scheduling and/or binding.

Stages of the behavioral synthesis

■ ***Scheduling***

- Scheduling introduces parallelism and the concept of time. It transforms the algorithm into an FSM representation. Using the data dependencies of the algorithm and the latencies of the functional units in the library, the operations of the algorithm are assigned to specific clock cycles. There are often many possible schedules. Directives that constrain the result with respect to latency, pipelining, and resource utilization will affect the schedule that is chosen.

■ ***Functional unit binding***

- Binding assigns the operations of the algorithm to specific instances of functional units from the library.

Stages of the behavioral synthesis

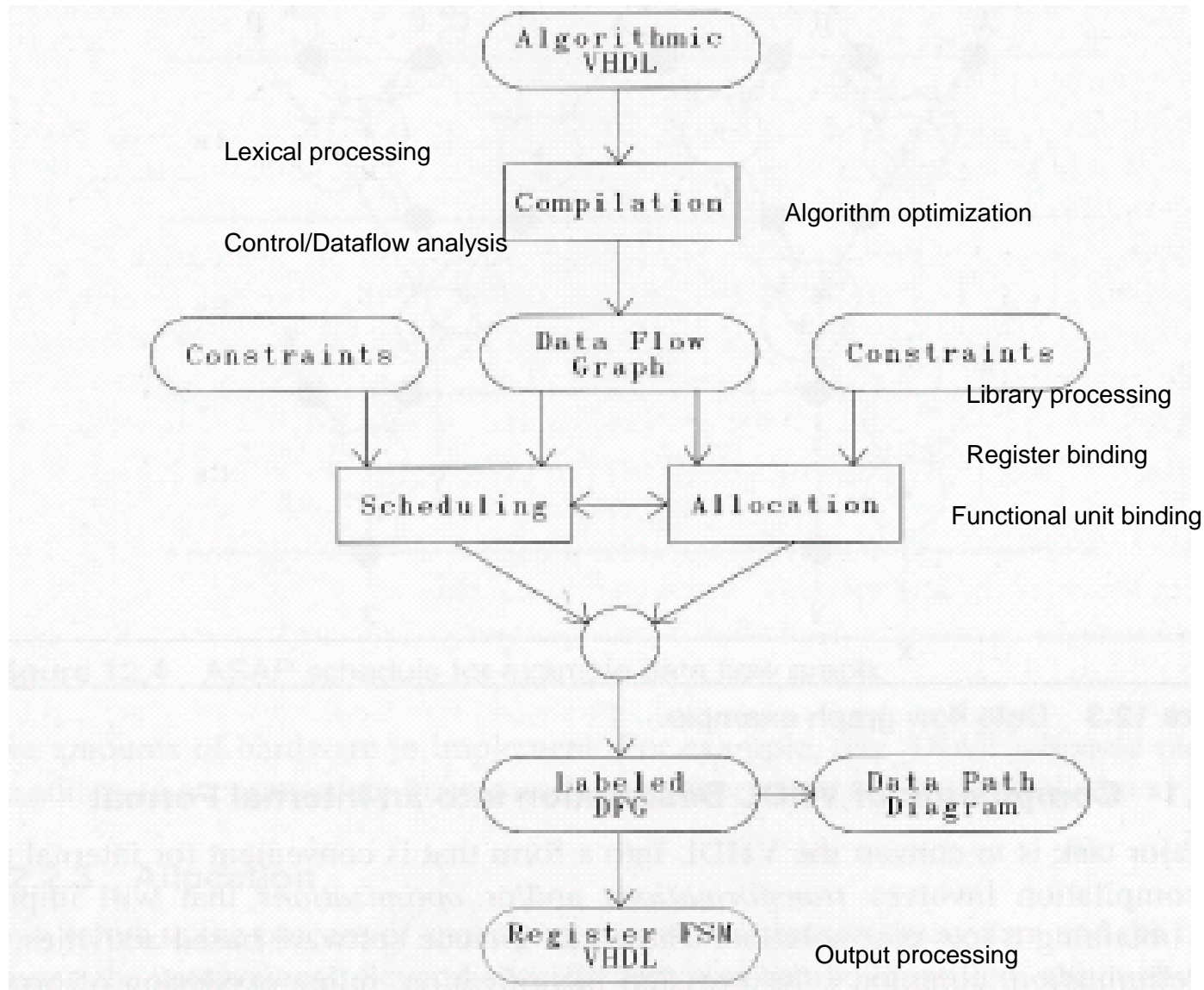
■ *Register binding*

- The register binding process allocates registers as needed and assigns each value to a physical register. Analysis of the lifetime of each data value can identify opportunities to use the same physical register to store different values at different times.

■ *Output processing*

- The datapath and finite state machine resulting from all of the previous steps are written out as RTL source code in the target language. This code can be structured in a number of ways to optimize the downstream logic synthesis process or to enhance the readability of the code.

Stages of the behavioral synthesis





CSP and TRS

- Fundamentally, synthesis High Level language requires technology that can translate sequential software semantics into parallel, state machine-based hardware
- This translation uses languages such as communicating sequential processes (CSP) and methodologies such as Term Rewriting Systems (TRS)



CSP: Communicating Sequential Processes

- In computer science, Communicating Sequential Processes (CSP) is a formal language for describing patterns of interaction in concurrent systems. It is a member of the family of mathematical theories of concurrency known as process algebras, or process calculi.
- CSP allows the description of systems in terms of component processes that operate independently, and interact with each other solely through message-passing communication. The relationships between different processes, and the way each process communicates with its environment, are described using various process algebraic operators. Using this algebraic approach, quite complex process descriptions can be easily constructed from a few primitive elements.



TRS: Term Rewriting Systems

- Term Rewriting Systems (TRS) are a well-understood formalism from computer science to describing concurrent systems that embody asynchronous and nondeterministic behavior in their specifications.
- A TRS consists of
 - "terms" which describe hardware states
 - "rules" which describe behavior. A "rule" captures both a state-change (an "action") and the conditions under which it can occur.

TRS description

- A TRS consists of a set of terms and a set of rewriting rules. The general structure of rewriting rules is:

- $pat_{lhs} \text{ if } p \rightarrow exp_{rhs}$

A rule can be used to rewrite a term s if the rule's left-hand-side pattern pat_{lhs} matches s and the predicate p evaluates to true. When a rule is applied, the resulting term is determined by evaluating the right-hand-side expression exp_{rhs} in the bindings generated during pattern matching.



BASIC SYNTHESIS STRATEGY

- Compiler maps a TRS to a synchronous FSM by
 - Mapping TRS terms to storage elements (e.g., registers, register files and other abstract datatypes)
 - Mapping TRS rules to combinational logic that generates next state values and enable signals for storage elements

Circuit synthesis

In a functional interpretation, a rule of the form

$$\begin{array}{l} pat_{lhs} \\ \text{if } exp_p \text{ where } pat_1 = exp_{lhs,1}, \dots, pat_n = exp_{lhs,n} \\ \rightarrow exp_{rhs} \\ \text{where } var_1 = exp_{rhs,1}, \dots, var_m = exp_{rhs,m} \\ || \\ || \\ || \\ || \\ ==\Rightarrow \end{array}$$
$$\begin{array}{l} rule = \lambda s. \text{ case } s \text{ of} \\ pat_{lhs} \Rightarrow \\ \text{case } exp_{lhs,1} \text{ of} \\ pat_1 \Rightarrow \\ \dots \\ \text{case } exp_{lhs,n} \text{ of} \\ pat_n \Rightarrow \\ \text{if } exp_p \text{ then} \\ \text{let} \\ \text{var}_1 = exp_{rhs,1}, \dots, var_m = exp_{rhs,m} \\ \text{in} \\ exp_{rhs} \\ \text{else} \\ s \\ _ \Rightarrow s \\ \dots \\ _ \Rightarrow s \\ _ \Rightarrow s \end{array}$$

Circuit synthesis

This function can be broken down into its two components: Π and δ .

The Π function determines a rule's applicability to a term and has the type, *Typeof* (*paths*) *Boolean*. The δ function, on the other hand, determines the new term in case evaluates to *true*.

For hardware synthesis, we break down R into actions on individual storage elements. For each storage element *e* affected by a rule R, δ gives its next state value. Π is the enable signal of all the affected registers.

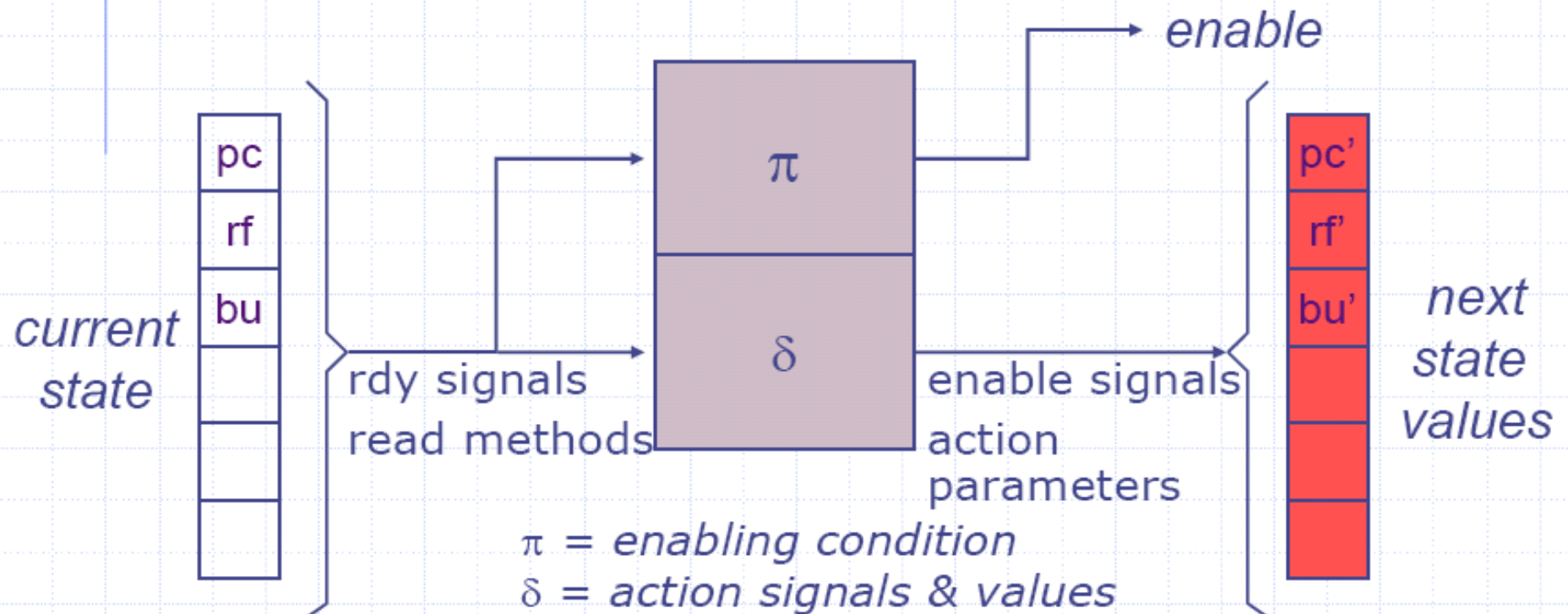
```
 $\pi = \lambda s. \text{case } s \text{ of}$   
   $pat_{lhs} \Rightarrow$   
     $\text{case } exp_{lhs,1} \text{ of}$   
       $pat_1 \Rightarrow$   
        ...  
         $\text{case } exp_{lhs,n} \text{ of}$   
           $pat_n \Rightarrow exp_p$   
           $\_ \Rightarrow false$   
        ...  
       $\_ \Rightarrow false$   
     $\_ \Rightarrow false$ 
```

```
 $\delta = \lambda s. \text{let}$   
   $pat_{lhs} = s$   
   $pat_1 = exp_{lhs,1}, \dots, pat_n = exp_{lhs,n}$   
   $var_1 = exp_{rhs,1}, \dots, var_m = exp_{rhs,m}$   
in  
   $exp_{rhs}$ 
```

Compiling a Rule

"Bz Taken":

```
when (Bz rc ra) <- bu.first, rf.sub rc == 0  
==> action pc := rf.sub ra  
bu.clear
```



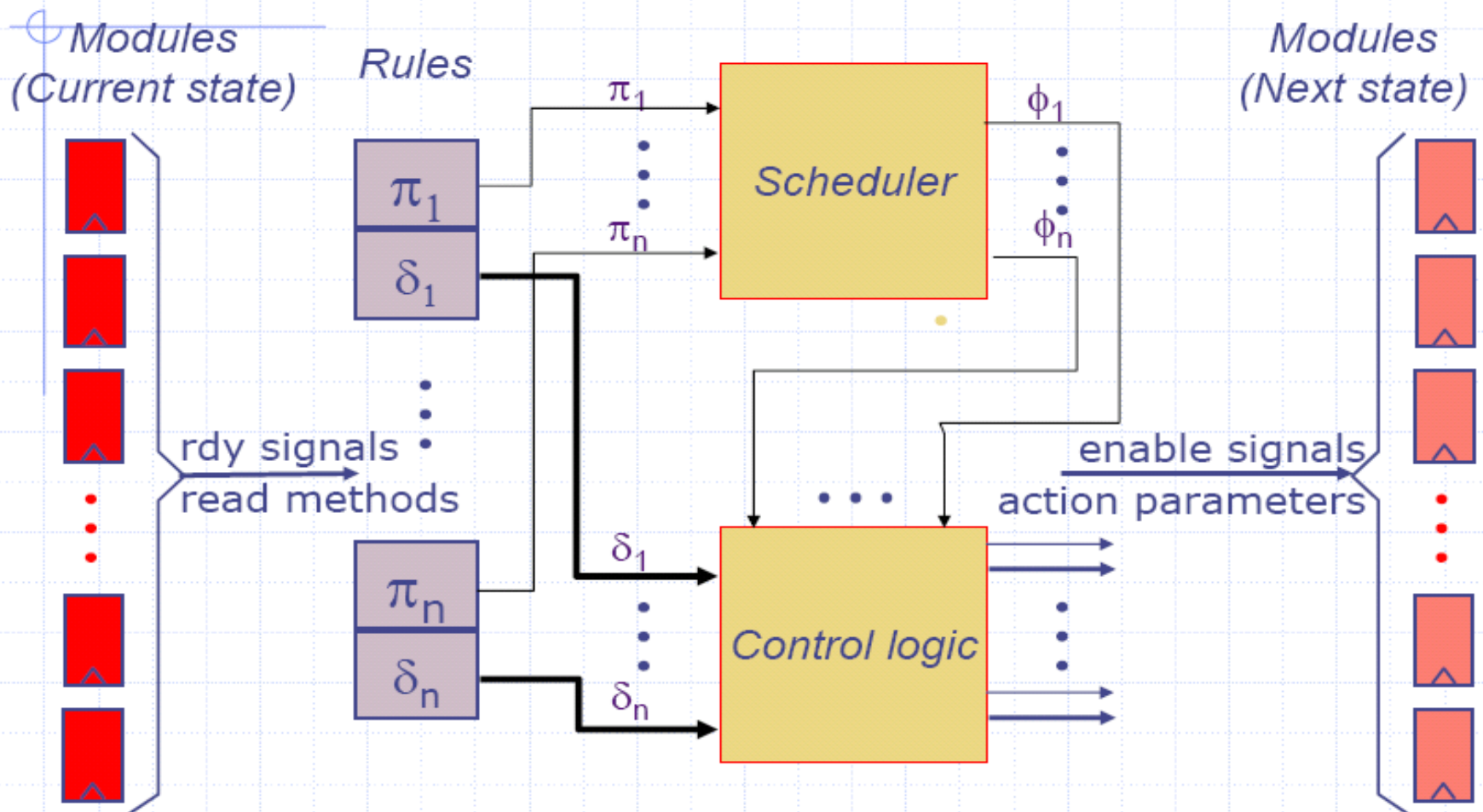
Schedule in TRS

- a scheduler that is currently implemented in TRAC that makes use of conflict-free (CF) relationships.
- an exact test for CF relationship between two arbitrary rule instances is expensive. Instead, TRAC performs several conservative tests to find as many CF relationships as possible.
 - First, two rule instances that read and write non-overlapping parts of the systems are CF.
 - If two rule instances do not rewrite the same registers,
 - if none of the registers affected by the δ of one is used by the Π and δ of the other, and vice versa, then the two rules are CF
 - If pairs of Π 's to conservatively determine when a pair can never be satisfied simultaneously and thus are CF

Schedule in TRS

- After TRAC has establish CF relationships between as many rule instances as possible, a graph of rule instances can be constructed by adding an edge between each non-CF pairs.
- Scheduling groups is formed by partitioning the graph into connected components. Different groups never interfere and can be scheduled independently.
- For each group, a round-robin priority encoder can be used to map Π to Φ for arbitration.

Scheduling and control logic



TRS Execution Semantics

Given a set of rules and an initial term s

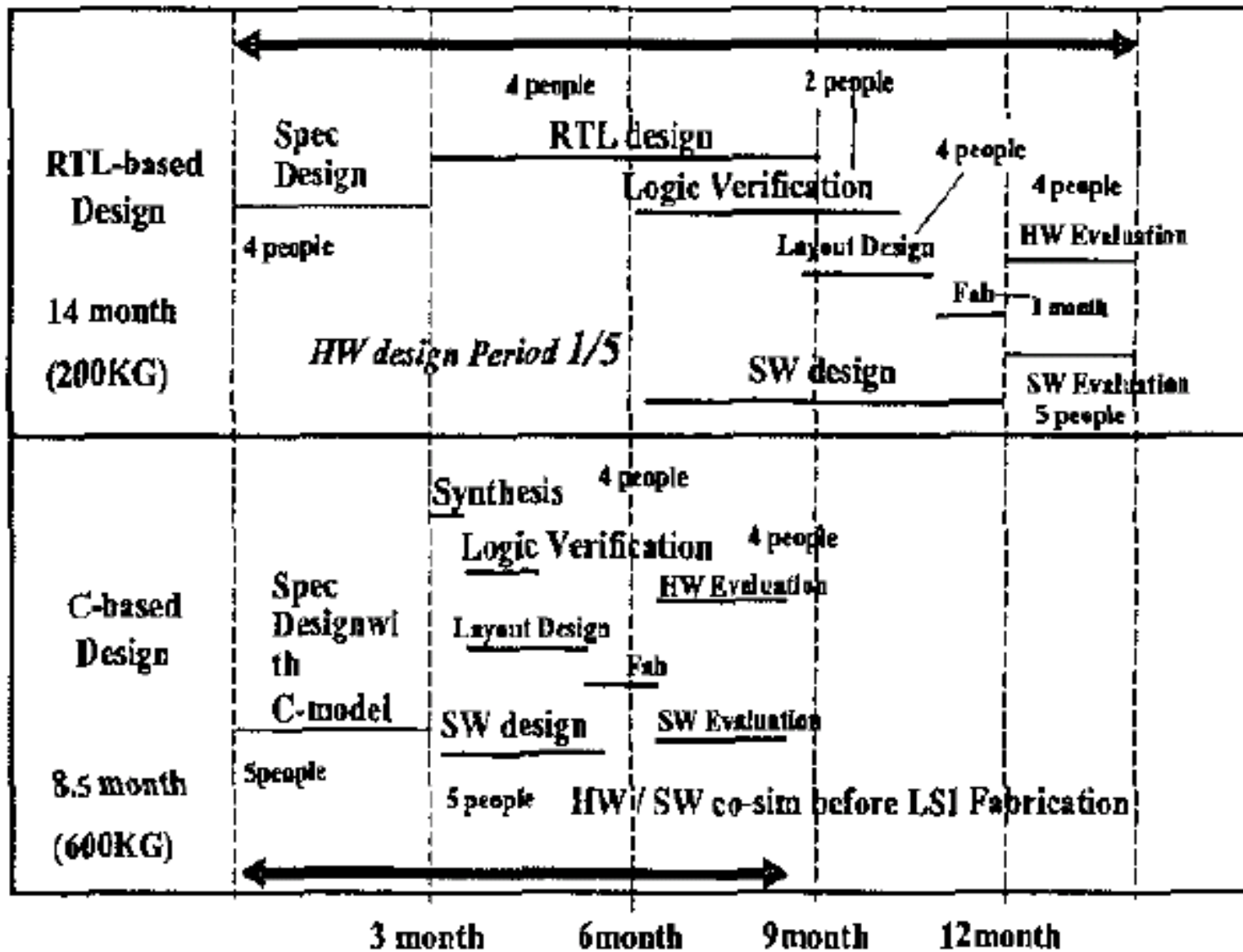
While (some rules are applicable to s)

- ◆ choose an applicable rule
(non-deterministic)
- ◆ apply the rule atomically to s

Synthesis problem: Generate a hardware scheduler that allows execution of as many enabled rules as possible at each clock without violating the semantics and generate all the associated control logic.

Synthesis performance Evaluation

- *"While behavioral synthesis tools offered some great productivity advantages over RTL synthesis, we found that most customers were either not willing to take the QoR [quality of results] hit necessary to use these products, or they were not willing to complicate their verification flow," --- Synopsys's chief executive officer Aart de Geus, 2004*
- In 2004, Synthesis giant Synopsys Inc. gave up on a 10-year effort to sell Behavioral Compiler and made an "end-of-life" announcement for SystemC Compiler, which is based on Behavioral Compiler. Cadence Design Systems Inc., meanwhile, has not actively marketed the behavioral synthesis technology it acquired from Get2Chip Inc



The design of 2.5 generation Mobile chip

RTL_based Design is slow because of

1. RTL_based co-simulation is not feasible because RTL is too slow.
2. SW design has to wait until the hardware finish

Fig.3 Design Period Comparison

TRS PERFORMANCE EVALUATION

- Synthesis of the GCD Circuit: Euclid's Algorithm for finding the greatest common divisor (GCD) of two integers

Version	FF (bit)	Util. (%)	Freq. (MHz)	Elapse (cyc)
Example 1	64	20	44.2	54
Example 2	102	38	31.5	104
Hand RTL	64	16	53.1	54

TRS PERFORMANCE EVALUATION

■ Synthesis of the Unpipelined Microprocessor

Version	FF (bit)	Util. (%)	Freq. (MHz)
Example 3	161	60 %	40.0
Hand RTL	160	50 %	41.0

as the problem size increases, the pay-back of hand optimizations diminishes while the effort required increases dramatically. This is evident in the synthesis of this example. The TRAC generated RTL and a hand-coded Verilog RTL of the unpipelined processor are comparable both in size and speed.



Other HL synthesis method

- block diagram-based algorithm
 - Popular in special application area, such as DSP domain.
 - With a rich set of domain-specific high-level block libraries
- algorithm designers can quickly construct complex DSP and communication systems at various abstraction levels and simulate to verify the desired behavior without tedious coding.



New high level synthesis approach

- based on optimized architecture synthesis
- using configurable IP as building block.
 - combines the notion of design-time configurable IP for processors and accelerators, along with design exploration and configuration tools to configure the IP from C applications.
 - offers the promise of being able to create complete application engines such as an H.264 video codec or a CDMA modem—engines that are completely beyond the scope of traditional behavioral synthesis technology.



optimized architecture synthesis

■ Advantages

- eliminates the problem of scope by the use of design-time configurable IP
- effectively addresses several major design challenges such as verification, system-on-chip (SoC) integration, software integration, and timing and physical closure

■ limitations

- success depends on the available IP Library
- applications using this approach could be limited because of the IP availability



Conclusion

- Behavioral synthesis will be a "catalyst" for ESL design
- Combined with FPGA, high level synthesis enable rapid prototyping, rapid evaluation and more architectural exploration.
- Tools available now stirred up a lot of interest but the performance of these tools has not been proven from industry

Future work

- Future work will focus on how to deliver what HL synthesis promise to do
- Promise 1: reducing the time it takes to go from functional specification to verified RTL while also accounting for timing and physical closure.
 - there's the risk that a designer reduces design capture time, only to extend the time it takes to verify the design and complete timing and physical closure
 - it was difficult to verify the results in any reasonable way, because the process of high-level synthesis changed the timing of the code (with regard to the cycles in which actions occurred) to the extent that it wasn't possible to use the same testbench on the pre and post-synthesis representations of the design. It is true for present situation. To improve the verification capability is essential.
 - When it has errors in design, how to debug and correct them

Future work

- Promise2: explore alternative implementations to make trade-offs in area, performance, and power.
 - It outputs "candidate" RTL implementations along with reports that help designers pick the best one to route through an RTL synthesis tool.
 - How to exert constraints to direct synthesis
 - the user has little control over the quality of the output. the input is hard to relate to the output if should the result need optimizing.
 - How to explore the design space
 - Overcome the local optimization to reach the global optimization



Future work

- Promise 3: enable HL reuse for different area/performance designs.
 - RTL has an implied micro-architecture that makes it impossible to re-use at significantly different points of area or performance
 - How to design IP in a way that its parameters could give the explicit of it area and performance; and this parameters could guide synthesis

News from industry

- Early this year, Xilinx Inc. has teamed up with a number of companies to form a system-level design initiative that it hopes will set standards for high-level, software-driven synthesis of FPGAs.
- “expands the reach of ESL solutions and FPGAs to new applications and to users who have never before implemented designs in programmable logic,” ---- *Wim Roelandts, president and CEO of Xilinx in a statement.*
- “Our focus continues to be on providing C-based tools that do not require low-level FPGA design skills. The Xilinx ESL Initiative helps spread the message that software-to-hardware is a practical and productive method of design,” --- *David Pellerin, chief technology officer, Impulse Accelerated Technologies Inc., in the same statement.*
- The participating companies include: Bluespec Inc.; Celoxica Ltd.; CriticalBlue Ltd.; Impulse Accelerated Technologies Inc.; Mitrionics AB; Nallatech Ltd.; Poseidon Design Systems Inc.; SystemCrafter Ltd.; and Teja Technologies Inc.

Reference

- Michael Meredith, “A look inside behavioral synthesis”, EEdesign.com, 2004-04-08
- Michael C. McFarland, Alice C. Parker, Raul Carnposano, “Tutorial on High-Level Synthesis”, IEEE, 25th ACM/IEEE Design Automation Conference, 1988
- Chen Chang, “Design and Applications of a Reconfigurable Computing System for High Performance Digital Signal Processing”, 2002
- James C. Hoe and Arvind, “Hardware Synthesis from Term Rewriting Systems”, 1999
- Chris Sullivan, Alex Wilson, Stephen Chappell, “Deterministic hardware synthesis for compiling high-level descriptions to heterogeneous reconfigurable architectures”, Proceedings of the 38th Hawaii International Conference on System Sciences – 2005
- Kazutoshi Wakabayashi, “CyberWorkBench: Integrated Design Environment Based on C-based Behavior Synthesis and Verification”, IEEE, 2005