Algorithmic C synthesis (High-level synthesis)

Reminder
System level design

The complexity of digital systems grows exponentially because of technological improvements, and user demands. The design entries have been shifted towards higher levels of abstraction. There are automated transformations into lower level models (with manual control and constraints). The human interaction along the design flow decreased significantly.

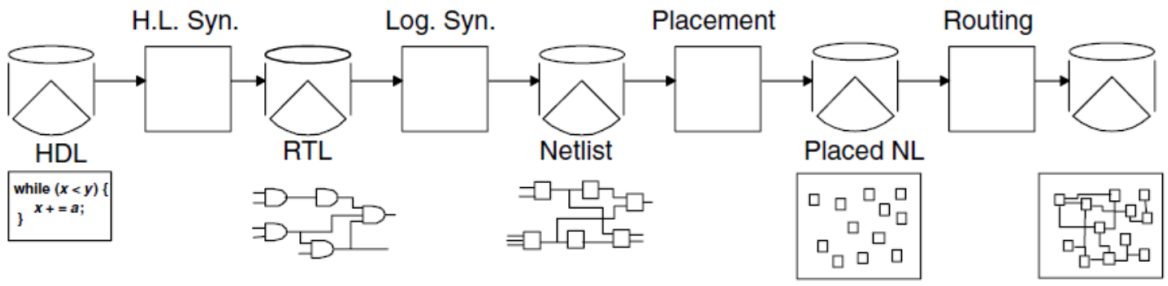

Figure 1. Digital design flow

The main aim is functional partitioning: defining the subtasks of the components, modeling the communication interfaces among design units and the external world. The implementation details are not important this point (the models are independent of the technology).

Tools are general purpose high level programming languages (mainly C, C++).

On algorithmic level the goal is behavioral modeling of the functional blocks. The behavioral models can be simulated. Simulation time is the most important optimization goal, so interpreted high level languages are rarely used, but C & C++ are commonly applied. There are several libraries that make hardware modeling more accurate and efficient (such as SystemC, Algorithmic C etc.). They are able to: modeling of the concept of time, modeling hardware-specific types (bit, logic vector etc.). E.g. System C has these features:

http://accellera.org/downloads/standards/systemc

The RTL model expresses the behavior as consecutive steps of basic data transformations between registers (individually controlled storage elements).

High level synthesis

It is step between the algorithmic level and RTL.

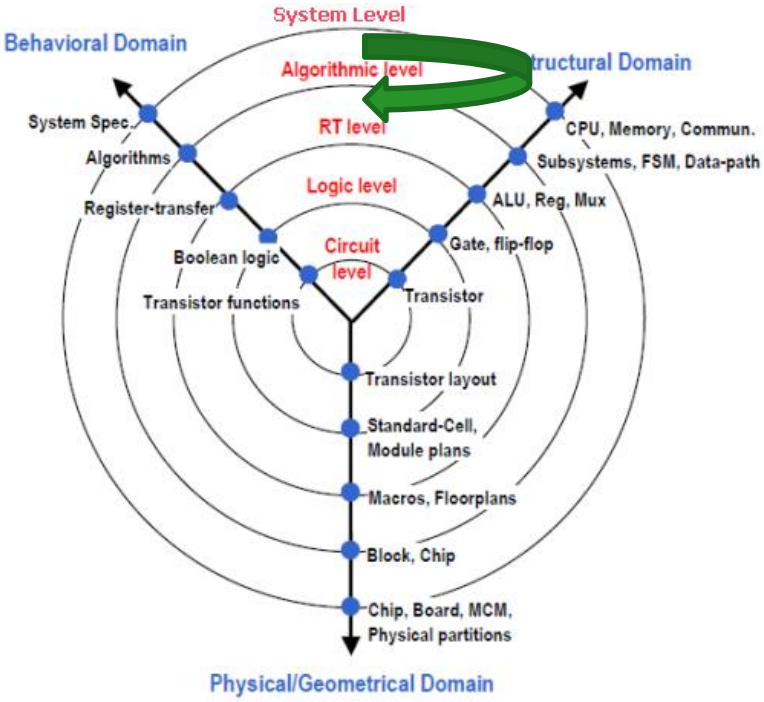

Figure 2. Abstraction levels and high level synthesis

There are some tools like Cadence / C-to-Silicon compiler, Mentor Graphics / Catapult C, Synopsis / Synphony C compiler, etc. The RTL models are generated automatically from the non-timed algorithmic models, and 5-10x productivity is promised by the HLS tool vendors.

A simple shift register - Example

Let's design the high level description of a serial-parallel converter! The width of the shift register is 16-bit, and if the enable input is high it shifts the data connected to „serial_in" at every clock signal. It has a synchronous reset which is low active.

The RTL model of a shift register

```
module shiftreg(input clk, input rst, input enable, input serial_in,
                output reg[15:0] data);

always @(posedge clk)
    if (~rst)
        data <= 16'd0;
    else if (enable)
        data <= { data[14:0], serial_in };
endmodule
```

The same function described in C:

```
unsigned short shiftreg( bool enable, bool serial_in, unsigned short data)
{
        return enable ? ( (data<<1) | serial_in) : data;
}
```

We can notice the followings:

- The function is a valid C++ code, there is no extension, pragma, or other tools
- The „bit” data type is represented by the bool data type of the C++
- The operation is described by a single function.
- The inputs are *passed by values.*
- The output is return value (*pointer* ill. *reference*)

Now we have to create a System C wrapper class which defines the ports of the hardware (so the inputs and the outputs). You can see such a wrapper class below.

```
#include <systemc.h>
#include "shiftreg.h"
#ifdef _DEBUG
#include <iomanip>
using namespace std;
#endif

SC_MODULE(shiftreg_sc)
{
        // input ports
        sc_in<bool>             reset;
        sc_in<bool>             enable;
        sc_in<bool>             clk;
        sc_in<bool>             serial_in;
        // output port
        sc_out<sc_uint<16> >    shftreg;
        // internal register
        unsigned short reg;
        // main method.
        void    shiftreg_cthread()
        {
                while (1) {
                        if (reset.read() == 0) {
                                reg = 0;        // low active reset
                        } else {
                                // call of the C code
                                reg=shiftreg(enable.read(), serial_in.read(), reg );
                        }
                        // write the output
                        shftreg.write(reg);
#ifdef _DEBUG
                        // test code
                        cout << setw(10) << sc_time_stamp();
                        cout << setw(2) << enable;
                        cout << setw(2) << reset;
                        cout << setw(2) << serial_in;
                        cout << setw(8) << hex << reg << endl;
#endif
                        wait();
                }
        }
        // constructor
        SC_CTOR(shiftreg_sc)
        {
                // opreation description module
                SC_CTHREAD( shiftreg_cthread, clk.pos());
                // sensitive for positive edge of clock signal
        }
};

#ifdef __CTOS__
SC_MODULE_EXPORT(shiftreg_sc);
#endif
```

The sc_in<> and sc_out<> templates define the ports of the hardware, the operation is described by a clocked thread. This is an event driven kernel of the SystemC. The thread runs when an

event occurs e.g. the change in clock signal. This thread is an infinite loop which ends with a wait(), so it waits until the next clock event.

Our SystemC module can be tested by the testbench implemented in *main.cpp*.

It is very important to understand that the simulator is the compiled executable file itself!

Step 1 - C to Verilog RTL

Open the **src subdirectory**, build the code using **make** command. Load the converted waveform file to the gtkwave, and verify the operation.

```
make
./shiftreg
gtkwave wave.vcd &
```

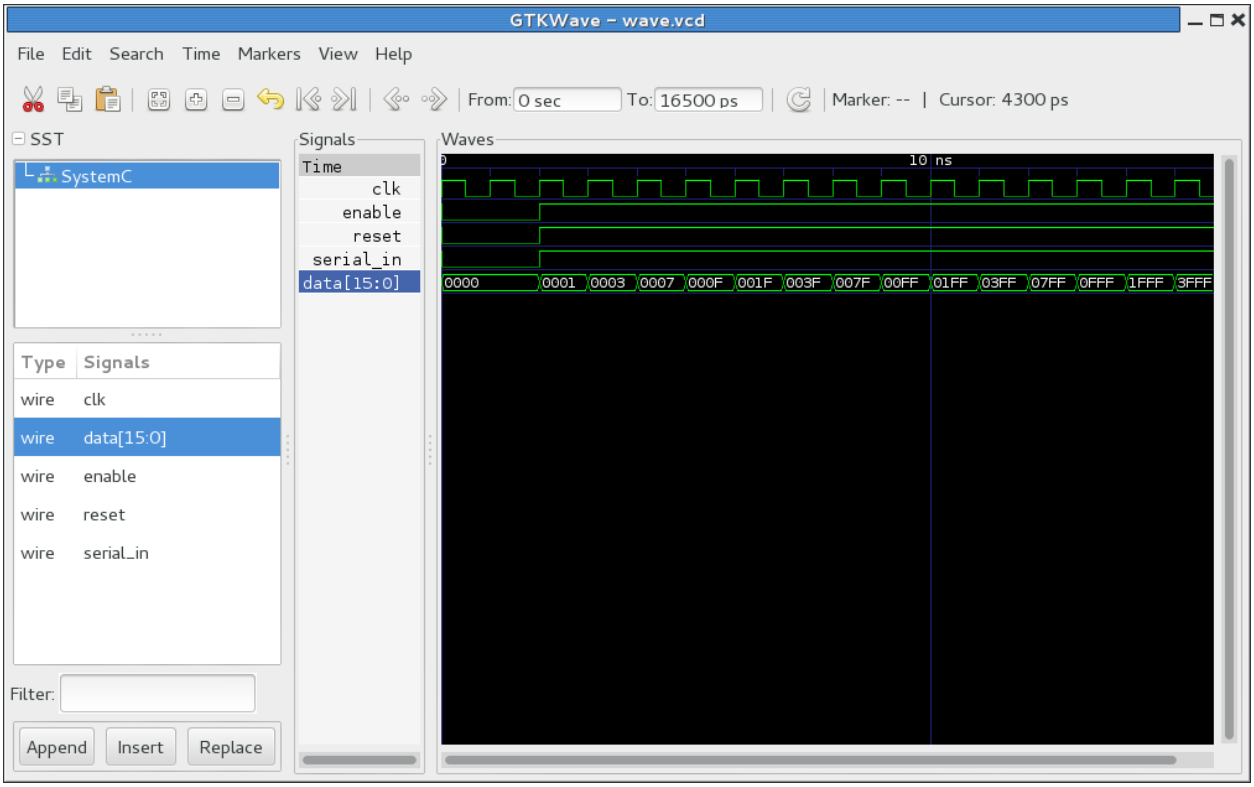

Figure 3. Waveforms in GTKWave

If everything is fine, we can create the RTL level hardver description written in Verilog using a high level syntheser. Please go back one level using **cd..**, and start the C-to-Silicon program from terminal:

```
/soft/cadence/run018.csh ctosgui &
```

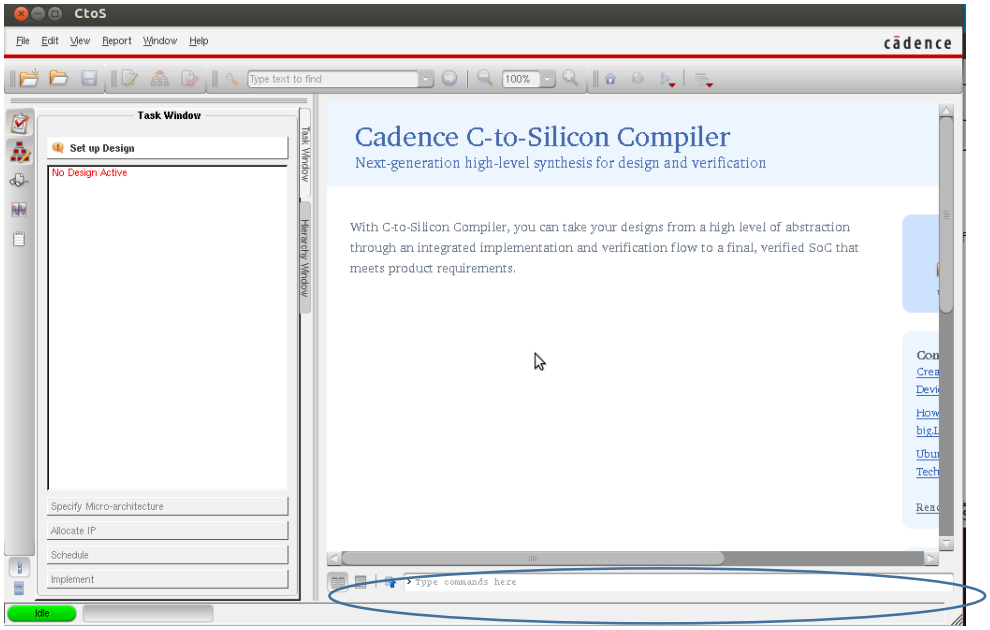

Figure 4. C-to-Silicon Compiler

Please insert this line into the command input line (Figure 4.):

```
source ctos.tcl
```

It starts a srcipt which will synthesize the C code, and display it as well. In „Report” menu you can see the resource usage and timing data.

Step 2 - Verilog RTL to gate level hardware description

The next step is the logic synthesis: conversion between register transfer level and structural level. We will use the *Encounter RTL compiler* tool of Cadence. The *ctos.tcl* script has generated a *rc.tcl* script. Please open it in a text editor (e.g. in **gedit**), and delete the last line (exit).

Please start the RTL compiler in the terminal:

```
/soft/cadence/run018.csh rc -gui -files rc.tcl
```

The script has just generated the structural model. In the **Report** menu you can see the number of the required gates, and the timing data.

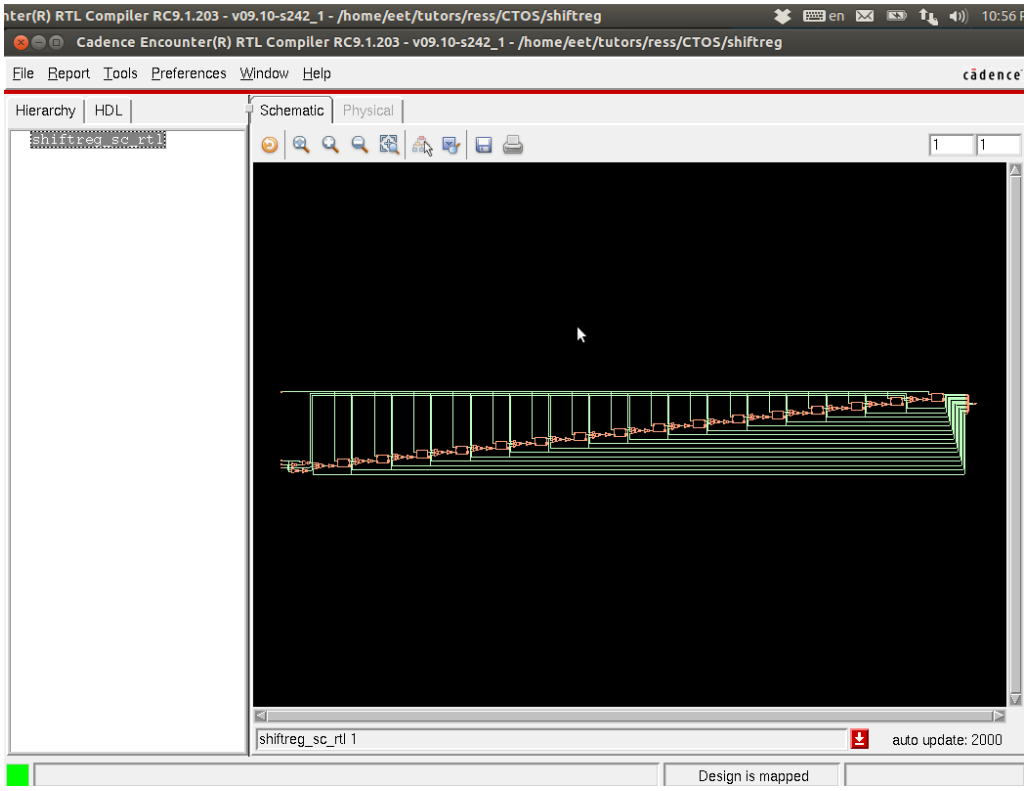

Figure 5. The schematic view generated by the RTL Compiler

Report Mapped Gates

Generated by: Encounter(R) RTL Compiler RC11.21 - v11.20-s012_1 (Jun 12 2012)
Generated on: Nov 13 2013 10:50:10
Module: shiftreg_sc_rtl
Technology library: h18_CORELIB_TYP revision 2.1
Operating conditions: typical (balanced_tree)
Wireload mode: enclosed

Gate	Instances	Area	Library
AND2X3	2	28.22	h18_CORELIB_TYP
AO22X3	16	361.27	h18_CORELIB_TYP
DFX2	16	858.01	h18_CORELIB_TYP
INVXL	1	5.64	h18_CORELIB_TYP
TOTAL	35	1253.14	

Close Help

Figure 6. Statistics

There are some warnings, beacuse the input rise-time and fall-time, and the output load is undefined.

We can stop here, because the next step is just the place & route in ASIC design. Now we can verify the operation of the circuit.

The RTL compiler generates the launch terminal. Please type „**reset**”, and press enter.

Exercises

Hints:

- please DO NOT CHANGE the name of the functions (shiftreg and shiftreg_cthread). But you can change the parameters, data, etc.)
- If you use a state variable, please use it in a class, and pass it as a reference (static cannot be synthesizable)
- Please use standard C++ only (e.g. 0b111 cannot be used).
- Please do not forget to modify the testbench in *main.cpp*.

Exercises

Task 1) „KITT light bar (red scanner lights) ☺” After RESET signal, the value of the register is 0x01.

Task 2) Triple KITT – like Task 1, but the moving pattern is 111.

Task 3) Circular shift register. Moving the MSB value to the first position (LSB), while shifting all other entries to the next position.