Technische Universität Chemnitz
Fakultät für Elektrotechnik und Informationstechnik
Professur Schaltkreis- und Systementwurf
Prof. Dr.-Ing. Heinkel

HIGH-LEVEL-SYNTHESIS USING CATAPULTC^{TM}

Manual

Practical course for the lecture ”EDA Tools”
Date: January 19, 2015

Contents

1 Theory and Fundamentals

1.1 Terms and Definitions

The term *High-Level-Synthesis (HLS)*, also known as *architectural synthesis*, *algorithm synthesis* or *behavioral synthesis*, represents one step during the design of digital hardware systems whereas the design is transferred from algorithmic to register transfer level [1, 2]. It is a step of refinement, whereby a description based on behavior of fundamental operations (logical and arithmetical) is mapped into a structural description of data and control path. It can be done by hand or with the help of various tools.

Circuit parts with functional character (operations) are connected on the data path (execution unit) to communication and memory resources on the level of register transfer blocks. The control path (control unit) coordinates these blocks and is displayed on logical level.

1.2 Abstraction Level

We can generally define the synthesis of systems as a realization from behavioral descriptions into structural ones. During the design of electrical systems you can distinguish three perspectives: behavior, structure and geometry. These are presentable in the so called Y-Graph.

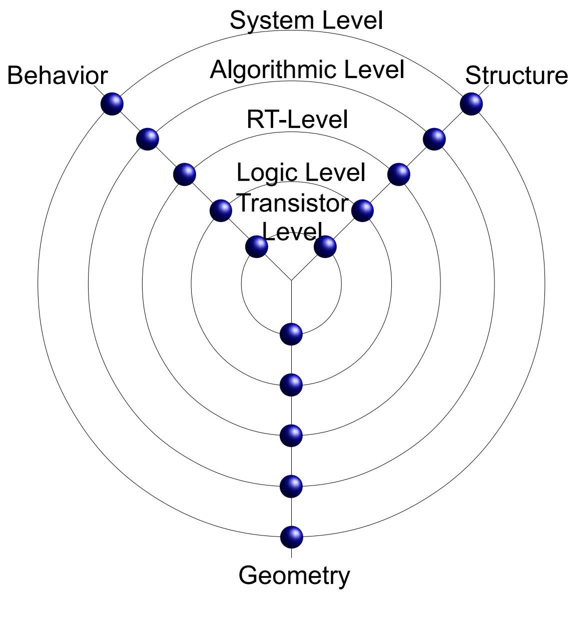


Figure 1: Y-Graph introduced by Gajski et. al. [3]

Additional different abstraction levels are shown as circles to the three perspectives. The bigger the radius the higher the abstraction level. A typical top down design starts on the system level, approaching the circuit level step wise. During the refinement and the reduction of abstraction a switch between branches occurs. Finally the geometry branch on circuit level serves as input for the chip manufacturer in a step that is called *tape out*.

A pure top down procedure is not possible and useful in most cases, because through verification design failures between the levels are revealed, who have to be corrected on higher levels. This process is also known as *yo-yo-Design*.

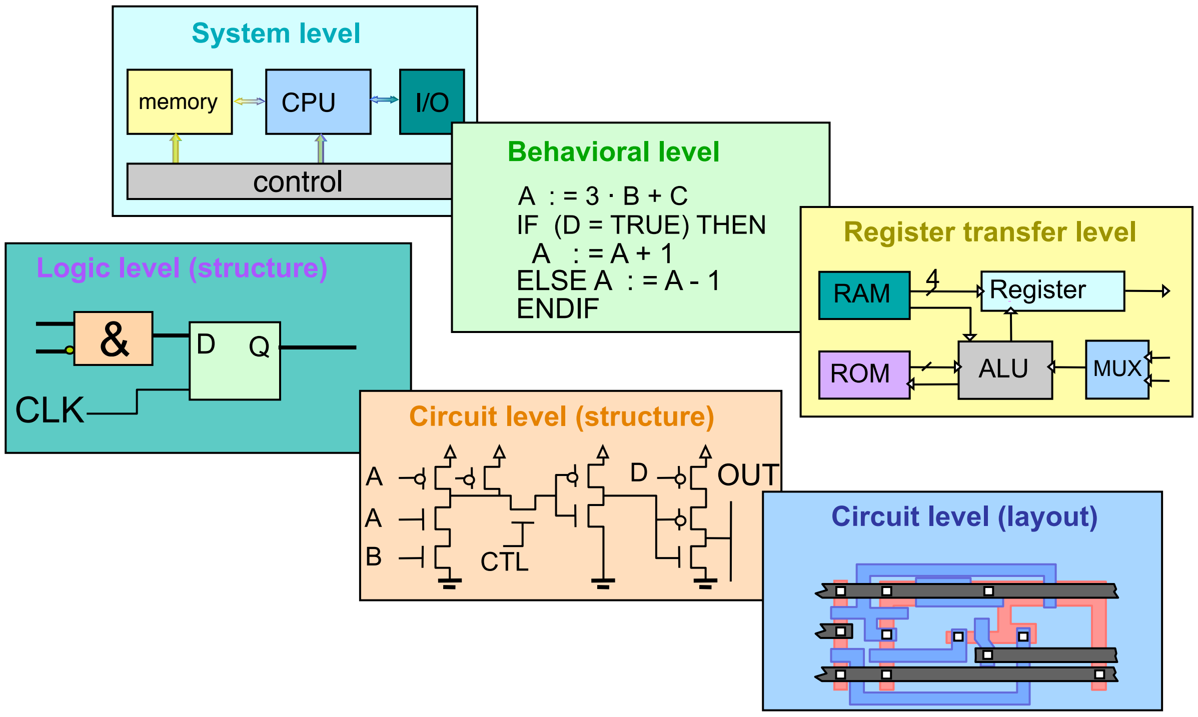


Figure 2: Design Levels

1.3 Synthesis Possibilities and Design Space Exploration

Without design automation, designing of nowadays circuits with hundreds of millions of transistors and millions of logic gates would not be possible. For every new design there is a huge amount of possible hardware architectures on all layers of abstraction, which would lead to a proper implementation of the required functionality. Even for an experienced developer it is not easy to find the appropriate architecture from all choices, which fulfills all constraints by consuming the minimal amount of resources (area, time and power). Synthesis processes help to switch between the various abstraction layers.

Logical synthesis is a procedure, where operations (i.e. shift, add, sub) and registers are mapped onto boolean logic gates. As input a description on the register transfer level (RTL) is needed, supplying a detailed specification of functionality, data movement and timing of signals. Items such as

- logical states,
- the corresponding operations as data path elements (i.e. multiplier and adder),
- an automation (finite state machine) for controlling these values and
- registers/memories that keep the values during multiple clock cycles

must be specified. Thus building a specific architecture needs many human resources and knowledge so that even a little change consumes a lot of additional time. To implement a given implementation in another architecture poses a hard and time consuming task. Therefor, architectural decisions are carried out in advance of the implementation phase where it is impossible to take certain facts, arising during the implementation, into account. The exploration of the entire design space is an impossible task.

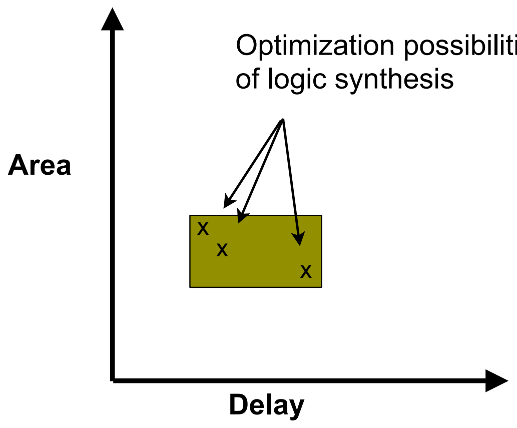


Figure 3: Design space exploration during Logic Synthesis

If we move to a higher abstraction level, like the algorithmic level, some more implementation details are hidden in the description of a design. Only functional specifications that do not include timing, data flow and control signals such as resets are factored in. Considerations according to these aspects are left to the synthesis tool. Design descriptions are formulated in high level languages such as C and/or C++, where only data dependencies between operations and an untimed (usually sequential) control flow within the application are specified. Thus the descriptions are reduced to about a tenth, compared to a RTL-description in HDL. For finding the optimal result, many different architectures, depending on the given constrains, are evaluated during the synthesis. The resulting RTL-code is further on usable as input to the following logical synthesis.

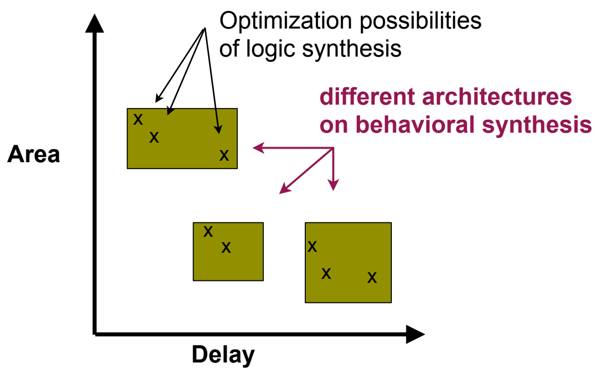


Figure 4: Design space exploration during High Level Synthesis

1.4 High Level Synthesis Process

HLS is divided into a preprocessing, a synthesis and a post-processing or generation task. During the preprocessing the source code is checked for syntactical errors and mapped onto a formal model, followed by an analysis by the synthesis kernel. During synthesis the data dependencies between operations and the control flow (branches, loops etc.) are evaluated, available resources in the circuitry are allocated. After scheduling the operations of the algorithm are finally bound to dedicated resources. During this process the parallelism within the design is extracted and utilized. In the final post-processing task, RT-Code is generated in form of a finite state machine.

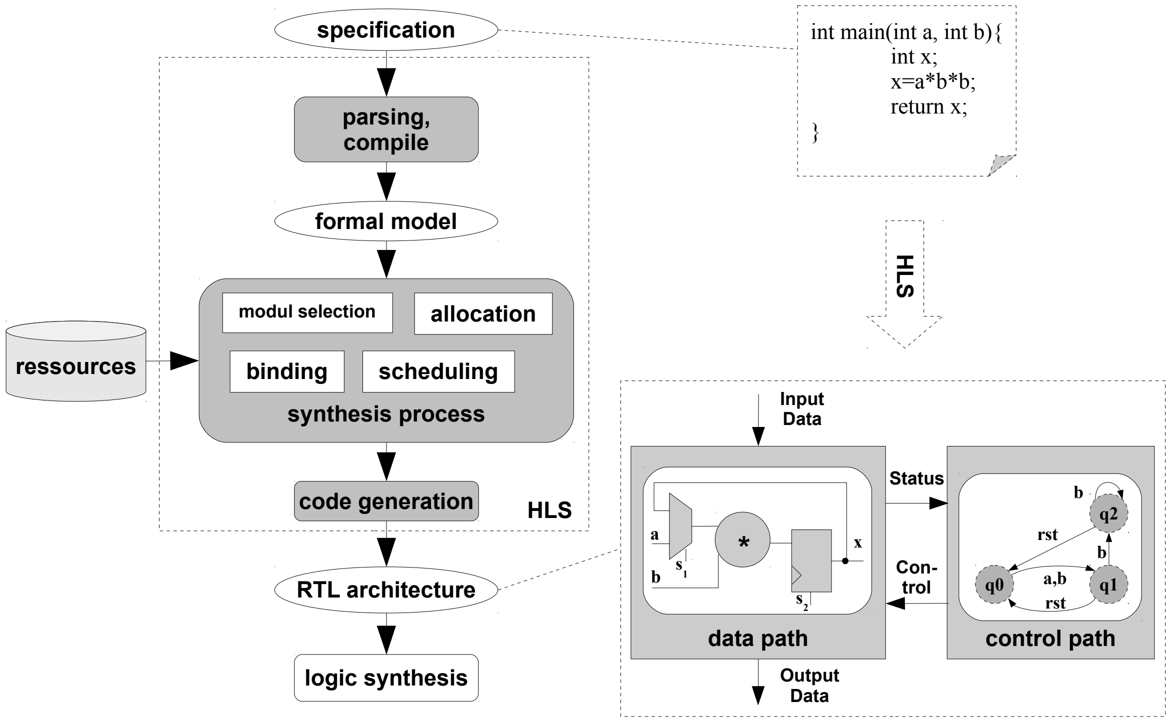


Figure 5: Synthesis Process

1.5 Optimization possibilities during the synthesis process

As mentioned in previous sections with the HLS process, various optimizer decide about the final architecture, affecting a circuit regarding size and time. Some optimizer require guidance and limitation by the user/developer. In this lab we focus on optimizations related to loops in form of *instruction parallelization* and *pipelining*,

1.5.1 Instruction Parallelization

During sequential execution of the operations, the calculation time rises continuously with every single calculation when only a single processing unit is available for all operations. If multiple processing units of the same type can be utilized, some calculations can happen concurrently. Given independence of the data, parallelization can reduce the calculation in a large extent. A drawback consists in the usage of more resources, because every processing unit can only execute a single operation at a time. In High-Level-Synthesis primarily loops are parallelized in this respect. One way is to unroll the loop body, allowing to calculate multiple loop bodies at the same time.

Take a look at the example below where no data dependencies exist between the arrays, enabling the computation of all operations in parallel. In Listing 1 displays the iteration over an array of data. The straight unoptimized synthesis of this loop would lead to an implementation utilizing an adder and a multiplier. The result will be present after a compute time of 10 cycles. In the second Listing 2 we unroll the loop completely and utilize 10 adder and 10 multiplier. In this case, the result will be available after a single cycle. As we can see, there is a trade-off between resource utilization and performance. HLS synthesis tools also allow partial unrolling of loops in order to evaluate trade-offs for meeting given constraints.

Listing 1: original loop

```
for ( i = 0; i < 10; i++){
   a[i] = a[i]*b[i] + c[i];
}
```

Listing 2: unrolled loop

```
a[0] = a[0]*b[0] + c[0];
a[1] = a[1]*b[1] + c[1];
a[2] = a[2]*b[2] + c[2];
a[3] = a[3]*b[3] + c[3];
a[4] = a[4]*b[4] + c[4];
a[5] = a[5]*b[5] + c[5];
a[6] = a[6]*b[6] + c[6];
a[7] = a[7]*b[7] + c[7];
a[8] = a[8]*b[8] + c[8];
a[9] = a[9]*b[9] + c[9];
```

1.5.2 Pipelining

Another form of parallel processing of commands consists in the usage of a pipeline, which can be seen as some kind of assembly line. The calculation is separated into sequentially processed sub operations, wherein consecutive iterations of a loop are processed concurrently shifted by a single step. After the calculation of a sub task is done the corresponding processing unit goes idle. Theoretically it is possible to calculate, after an initial delay, one solution of a loop body with every cycle.

The example below shows a pipeline where a subtractor, a multiplier and an adder execute four consecutive iterations of a loop of the equation $(a - b) \times c + e$.

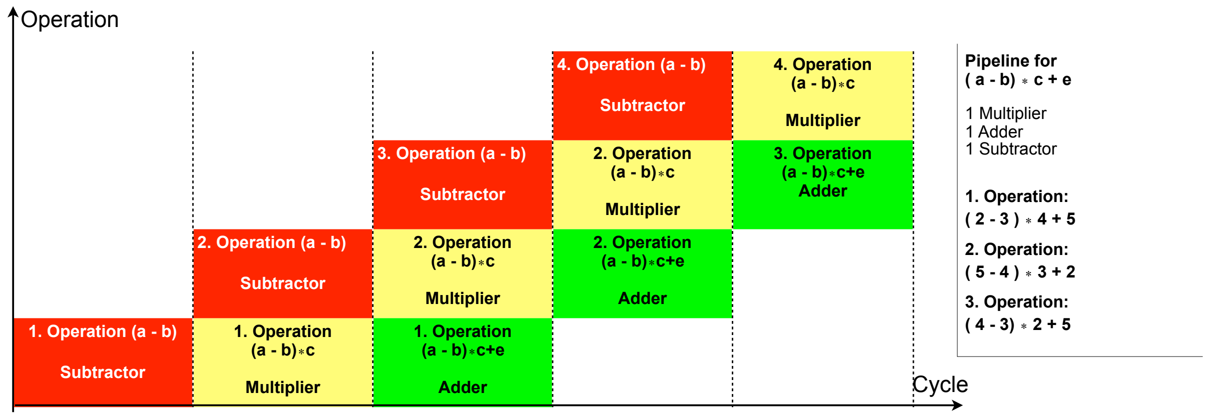

Operation
Cycle
1. Operation (a - b) Subtractor
2. Operation (a - b) Subtractor
3. Operation (a - b) Subtractor
4. Operation (a - b) Subtractor
1. Operation (a - b)*c Multiplier
2. Operation (a - b)*c Multiplier
3. Operation (a - b)*c Multiplier
4. Operation (a - b)*c Multiplier
1. Operation (a - b)*c+e Adder
2. Operation (a - b)*c+e Adder
3. Operation (a - b)*c+e Adder
Pipeline for (a - b) * c + e
1 Multiplier
1 Adder
1 Subtractor
1. Operation: (2 - 3) * 4 + 5
2. Operation: (5 - 4) * 3 + 2
3. Operation: (4 - 3) * 2 + 5

Figure 6: Pipelined operations

The speed profit of the pipeline is accomplished with the help additional resources: for every stage an additional operation unit including registers is needed. The registers are mandatory, as the intermediate results have to be stored. In general a pipeline is more resource-conserving compared to loop unrolling.

2 Catapult C Synthesis by Mentor

Catapult C Synthesis (short: CatapultC) is a High-Level-Synthesis tool, which enables the implementation from ANSI C/C++ and SystemC into RTL netlists for FPGA or ASIC platform. Here CatapultC generates netlists, simulation scripts, schematics and reports. CatapultC is just one of many HLS tools on the market. Other programs are for example AutoPilot by AutoESL, Synphony HLS by Synopsys or C-to-Silicon by Cadence.

2.1 Coding guidelines for C/C++ with Catapult C Synthesis

Catapult C Synthesis supports C/C++ but some language constructs have specific meaning, when synthesized into hardware. Some restriction have to be taken into account, because it is not possible to transfer a few C/C++ constructs into hardware. CatapultC also introduces new datatypes which are bit accurate.

2.1.1 Structures, Datatypes and Variables

With a background in C/C++, the developer should already have experience with the different structures, datatypes, (static) variables and pointer arithmetic. CatapultC supports every datatype present in C/C++ as well as structures, with exception of union constructs. Here CatapultC splits the elements of the data structure in a single elementary datatype to make them applicable. Some restrictions apply on floating point datatypes. In hardware they have a length of 32 bit, so the accuracy is different to *float* type in software. Also pointer can only be utilized under special circumstances.

CatapultC Datatypes CatapultC supports the conventional datatypes from C, as shown in the following chart. The representation of these datatypes in hardware is also shown in VHDL language.

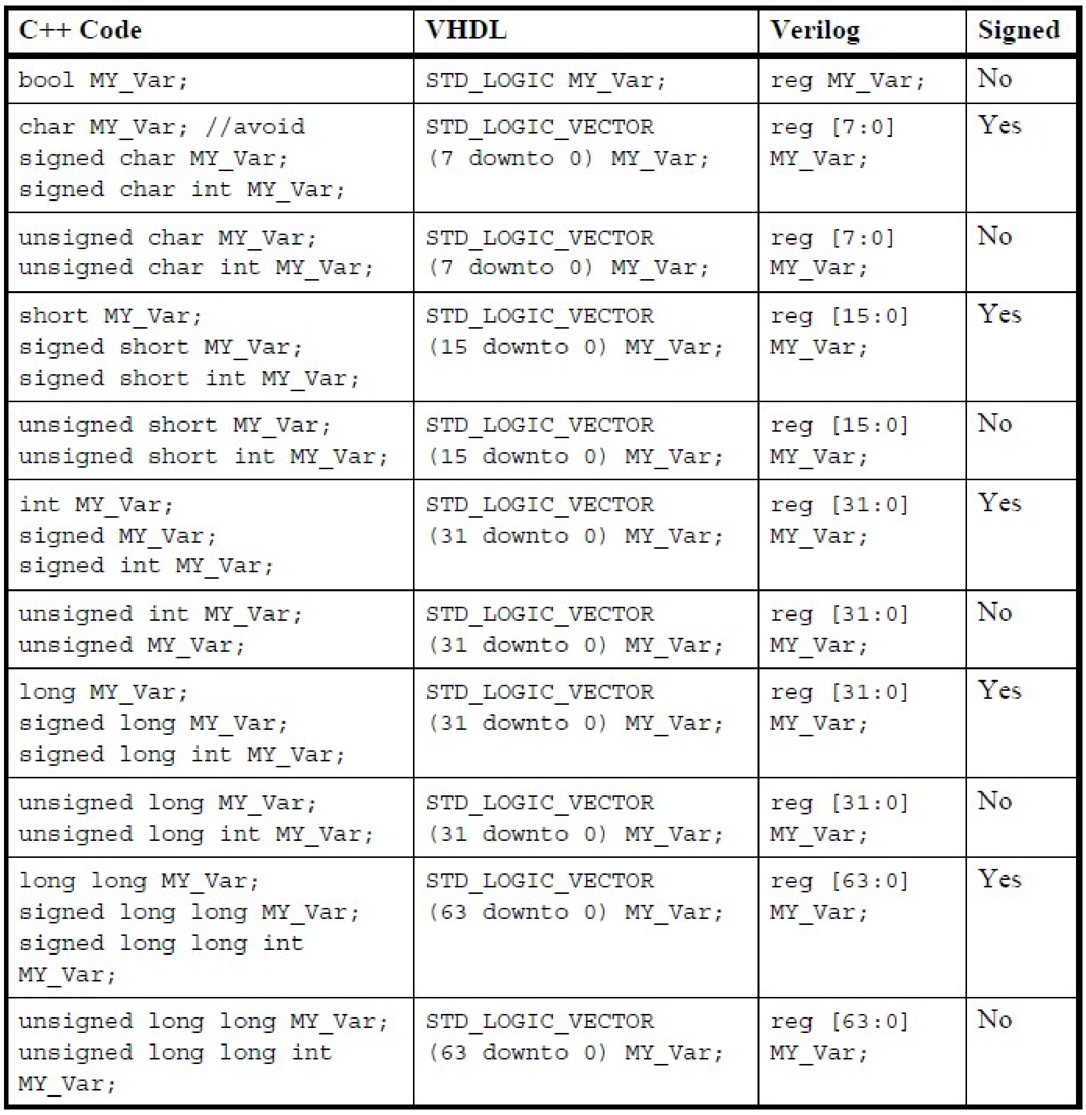
Besides these, CatapultC introduces more datatypes to achieve a bit wise precision for the hardware. These are shown in the following table.

Datatype	Description	value range
ac_int<W,false>	unsigned integer	0 to 2W - 1
ac_int<W,true>	signed integer	- 2W-1 bis 2W-1 - 1
ac_fixed<W,I,false>	unsiged fixed-point	0 bis (1 - 2-W)*2I
ac_fixed<W,I,true>	signed fixed-point	(-0.5)*2I bis (0.5 - 2-W)*2I

The W in brackets stands for the bit width, true or false stand for signed or unsigned datatypes. The fixed-point datatypes have an additional value I, indicating the number of integer digits before the dot.

Variables and Pointer While programming C-code with CatapultC you have to take care **not** to use unstatic global variables as they are ignored by the compiler. That is important, because the parser of CatapultC accepts global variables. There is merely a note for their usage. Global static variables and global pointers are valid in CatapultC; references to function are not supported. Pointer are a field of its own as they are only allowed to use when:

- the length of the content is fixed,
- they contain a transfer parameter of a function.



C++ Code	VHDL	Verilog	Signed
`bool MY_Var;`	`STD_LOGIC MY_Var;`	`reg MY_Var;`	No

`char MY_Var; //avoid`
`signed char MY_Var;`
`signed char int MY_Var;`
	`STD_LOGIC_VECTOR`
	`(7 downto 0) MY_Var;`
		`reg [7:0]`
		`MY_Var;`
			Yes

`unsigned char MY_Var;`
`unsigned char int MY_Var;`
	`STD_LOGIC_VECTOR`
	`(7 downto 0) MY_Var;`
		`reg [7:0]`
		`MY_Var;`
			No

`short MY_Var;`
`signed short MY_Var;`
`signed short int MY_Var;`
	`STD_LOGIC_VECTOR`
	`(15 downto 0) MY_Var;`
		`reg [15:0]`
		`MY_Var;`
			Yes

`unsigned short MY_Var;`
`unsigned short int MY_Var;`
	`STD_LOGIC_VECTOR`
	`(15 downto 0) MY_Var;`
		`reg [15:0]`
		`MY_Var;`
			No

`int MY_Var;`
`signed MY_Var;`
`signed int MY_Var;`
	`STD_LOGIC_VECTOR`
	`(31 downto 0) MY_Var;`
		`reg [31:0]`
		`MY_Var;`
			Yes

`unsigned int MY_Var;`
`unsigned MY_Var;`
	`STD_LOGIC_VECTOR`
	`(31 downto 0) MY_Var;`
		`reg [31:0]`
		`MY_Var;`
			No

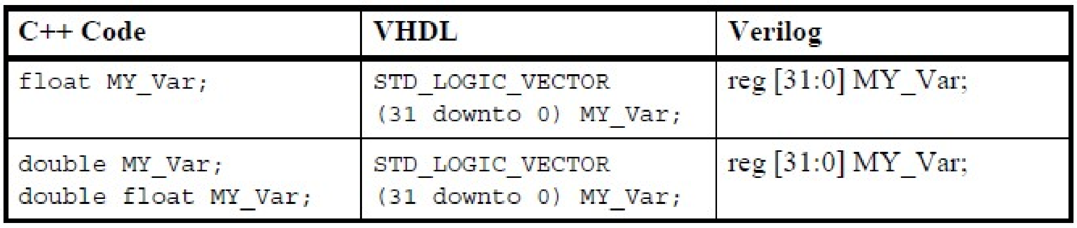
`long MY_Var;`
`signed long MY_Var;`
`signed long int MY_Var;`
	`STD_LOGIC_VECTOR`
	`(31 downto 0) MY_Var;`
		`reg [31:0]`
		`MY_Var;`
			Yes

`unsigned long MY_Var;`
`unsigned long int MY_Var;`
	`STD_LOGIC_VECTOR`
	`(31 downto 0) MY_Var;`
		`reg [31:0]`
		`MY_Var;`
			No

`long long MY_Var;`
`signed long long MY_Var;`
`signed long long int`
`MY_Var;`
	`STD_LOGIC_VECTOR`
	`(63 downto 0) MY_Var;`
		`reg [63:0]`
		`MY_Var;`
			Yes

`unsigned long long MY_Var;`
`unsigned long long int`
`MY_Var;`
	`STD_LOGIC_VECTOR`
	`(63 downto 0) MY_Var;`
		`reg [63:0]`
		`MY_Var;`
			No

Figure 7: CatapultC Datatypes 1



C++ Code	VHDL	Verilog

`float MY_Var;`	`STD_LOGIC_VECTOR`
	`(31 downto 0) MY_Var;`
		reg [31:0] MY_Var;

`double MY_Var;`
`double float MY_Var;`
	`STD_LOGIC_VECTOR`
	`(31 downto 0) MY_Var;`
		reg [31:0] MY_Var;

Figure 8: CatapultC Datatypes 2

2.1.2 Methods, Ports and Interfaces

Like in C++, the functionality of a circuit is realized by methods or functions. In hardware they represent a component or a subsystem with ports, which are provided along with defined interfaces to communicate with other components or subsystems. Transfer parameter and return values result in *Inputs* and *Outputs* of the hardware component. Parameters can be variables, arrays, pointer and references. Take a look at the following example to get a more in depth explanation:

```
#pragma hls_design top
int my_function( int a, int b )
{
   return a + b;
}
```

In the function *my_function* the return value of the integer type forms the output. The transfer parameters *a* and *b* represent the inputs to the component. During the transfer, the variables are first stored within a register before the generated hardware handles them. However, arrays, pointer or references used in the function change the meaning of the transfer parameter (!), as they can be input, output or bidirectional input/output. Arrays will be synthesized into a memory fraction with bidirectional access. In order to allow the synthesis tool the allocation of this memory, the array size must be known at compile time. Pointers are treated according to their usage within the function. Writing a pointer indicates an output, reading a functional input. If the pointer is read and written at the same time, it acts as input/output.

For each component a Main-Function has to be defined and tagged as the "Top-Level" of the design. In this part of the circuit all other subparts of this component will be embedded.

2.1.3 Loops

Every loop type from C/C++ is supported by CatapultC. If possible, loops should alway have a defined bound (fixed maximal number of iterations). For example:

```
const int MAX = 10;
for (i=0; i<MAX; i++) {
   if (i==n) break; // variable break point
}
```

The implementation of loops in hardware can be optimized by unrolling, instruction parallelization or pipelining. For the pipelining, all loops have to be nested into each other. In succession arranged loops cannot result in a pipeline, but there are optimizers during HLS that try to merge loops within a design description. Best practice is to combine loops manually!

2.2 Coding restrictions in C/C++ for HLS

At present CatapultC and HLS in general do not support:

- Dynamic memory allocation (e.g. dynamic lists),
- Recursive function calls,
- Global variables,
- Union constructs.

2.3 Workflow in CatapultC

The synthesis of RTL-code with CatapultC is structured in 5 major steps:

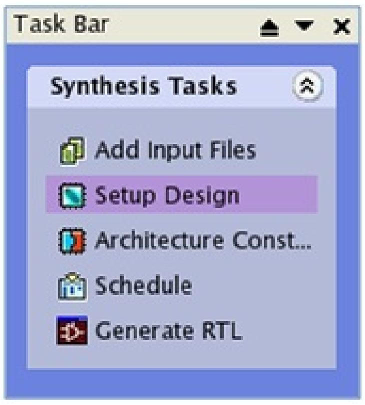


Figure 9: Taskbar

- Specify input files,
- Configure Design,
- Determine boundaries and constraints,
- Implementation of the design,
- Architecture scheduling and eventually
- Generate RTL-code.

3 Lab Instructions

3.1 Preface

CatapultC is a High-level-synthesis tool, which generates structural RTL-netlists from behavioral ANSI C/C++ code [4]. Additionally it provides a simulation, a testing as well as a verification environment.

The following project will give you a small insight in coding behavioral C/C++ code for hardware synthesis as well as an understanding of the workflow for High-Level-Synthesis on the example of CatapultC.

3.2 Start

The very first step is to prepare the environment for the CatapultC tool and transfer the lab-template to your local account. Perform the following steps:

- Right-click on the desktop to open a terminal
- Type in these commands:

```
$ cd ~/edat
$ mkdir pract_hls
$ cp -a /home/4all/tmp/edat_pract/hls/ ~/edat/pract_hls
$ cd ~/edat/pract_hls/hls
```

You will find the 2 folders *project1* and *project2*, which contain templates for the following tasks, within *../pract_hls/hls/*.

3.3 Optimization of a FOR-loop-Implementation

Given is the following C-function:

```
void calculate(int x[], int y[], int z[])
{
        for (int i=0; i<15; i++)
        {
                z[i] = x[i] * i + y[i];
        }
}
```

The first task is to get an impression about the influence of Loop-Unrolling and Pipeling to the resource usage after synthesis.

- Type in these commands:

```
$ cd project1/
$ . setenv.sh
```

- Start CatapultC and the given project with (This may take some time!):

```
$ catapult -file HLS-project1.ccs
```

If you want to take a look at the C-function click on calculate.cpp on the left-hand side of the CatapultC main window. Furthermore you will find there *Synthesis Tasks*.

- **Click on Architecture Constraints** and partial unroll the for-loop three times.
- **Apply the changes and click on Schedule.**

The architecture will be scheduled and the solution will be drawn in a Gantt chart. You can see the temporal sequence of the operations in form of a bar. Its length corresponds to the time period of an operation.

- **Click on Generate RTL.**

The design will be generated now. You can find the output files, like reports about the build process, RTL-code files or different schematics, in the output folder.

- **How many resources (adder, multiplier) are used for the calculation and what is the number of throughput cycles?**

Hint: Look at the Schedule Tab and study the RTL report under *Output Files/Reports/RTL* and *Output Files/Reports/Cycle*.

Now, **synthesize the for-loop as pipeline** by changing the *Architecture Constraints* and repeating the steps from above.

- What are you expecting about the resource usage and the throughput time?
- **Determine the number of resources and the throughput cycles.**

When you finished this task, **close CatapultC** (you may save the project). Change to the folder *../pract_hls/project2/* using the terminal:

```
$ cd ..
$ cd project2/
```

3.4 Implementation of a CRC

Your task is to implement a CRC-CCITT in C++ in the provided template. The implementation should calculate the CRC of a given junk of data. For this the following function prototype should be used:

```
void crc_ccitt(
 data_t        datastream[MAX_DATABITS],       // junk of data
 param_t       databits,                       // size of the junk data
 result_t      &res                            // reference to the result
 );
```

Just modify the file **crc_ccitt.cpp** that already contains this function prototype. Additional information is given in the function header. You can find a pseudo code which simply describes the structure of the CRC-code that must be implemented. Take notice of the header file **crc_ccitt.h** where all types and macros are defined. Contrary to the other crc-implementations of the previous labs the data is not handed over bit by bit, it is provided via memory as a whole ”junk”.

- Start CatapultC in shell mode:

```
$ catapult -shell
```

- Load the project template and set all necessary constraints of our design, the technology and the circuitry by calling the script *setup.tcl*. Type into the CatapultC shell:

```
source setup.tcl
```

In this way the HLS selects a library of suitable components (e.g. adder, multiplier or RAM, ROM), which are now available for synthesis. The library also contains information about the timing of these components for estimating the temporal behavior and exploration of the design space by the tool. The clockspeed is set to 100 MHz and the handshake process of the design is configured using a Start Flag and a Done Signal. At last the synthesis tool uses calculate.cpp as the top-level of the design.

- For editing **crc_ccitt.cpp** and looking at the header file, just open them with e.g. *gedit* using the file browser (righ-click on crc_ccitt.cpp → open with → gedit).
- You can find the files in your project folder in the subdirectory **../pract_hls/project2/Input_Files**.
- **Start implementing the CRC!**

Once you saved your implementation, CatapultC will recognize changes in **crc_ccitt.cpp** and branch a new solution. The output should show something like:

```
solution.v1{2}># Input file has changed
go new
solution file set {$PROJECT_HOME/Input_Files/crc_ccitt.cpp} -updated
# Info: Branching solution 'solution.v2' at state 'new' (PRJ-2)
```

Acknowledge this by hitting **Enter** on your keyboard.

Now is a good time to save your current project!

- Type **project save** into the CatapultC shell.

This should be done everytime you change your **crc_ccitt.cpp**.

- For compiling your design type **go compile**.
- The name of the solution may change from *solution.v2* to *crc_ccitt.v1* or similar to that. Do not worry about that.

Once the compilation worked without an error, a makefile will be generated which checks the Design with the included testbench **crc_ccitt_tb.cpp**. There exists a small script which runs the makefile.

- Typing **source check.tcl** will call that script.

In the *CatapultC Shell* window you will see an information if your code is correct or has to be changed.
If your code is correct you will see:

```
Correct result! Great!
```

If your implementation fails somehow, it will show:

```
ERROR! Wrong result!
```

Repeat the steps above until it works. **You will not pass the lab otherwise!!!**

If the result was correct **save the project** and **exit the CatapultC shell mode**:

```
project save
# Saving project file '../pract_hls/project2/HLS-project2.ccs'.
application exit
```

3.4.1 Architecture Scheduling

Now it is time to get some impression about the timing of your design. Therefor we use the graphical environment of CatapultC again.

- Start the CatapultC GUI with opening your project by:

  ```
  $ catapult -file HLS-project2.ccs
  ```

At first we leave all settings on default.

- **Click on Schedule** in the *Synthesis Tasks* window.

3.4.2 Generate RTL-Code

- **Click on Generate RTL.**
- **Study the RTL report** under *Output Files/Reports/RTL.*
- Look for **Total Area, Throughput Cycles, Latency Time and Throughput Time** and compare it with the values you can find under *Table* in the right window.

3.4.3 Optimization

There are optimizations available to achieve a better result. The Following constraints have to be met:

- *Latency Time* $<$ 20.00 ns
- *Throughput Time* $<$ 20.00 ns
- *Total Area* $<$ 300.00

Go back to the *Architecture Constraints* in the *Taskbar* and change the specifications. For the main loop in your code you will find different optimization possibilities and settings. **Try them!** CatapultC creates another version of the design and saves its changes there. In the *Table* under *Schedule* you can see the properties of the design versions and their values in comparison, such as cycles, total area, and latency time.

The lab will be passed once the design constraints above are met.

Good Luck!

3.4.4 Simulation

The following chapter is voluntary and not necessary to pass this practical course! It will give you some more insights into your design and the design flow using HLS tools. It is worth the time!

Now we want to simulate the design with the simulation tool Modelsim.

During the RTL-code generation process special makefiles were created, which now can be found under *Verification/Modelsim*. You can compile and execute them with a double-click.

- **Compile and Execute the makefile "Cycle VHDL output 'cycle.vs' vs Untimed C++"**.

After that the Modelsim program window opens. Start the Simulation by clicking in the menu bar

- **Simulate → Run → Run -all**

Alternatively you can type into the *Transcript* window following command:

```
$ run -all
```

You can follow the simulation in the wave form viewer and if it is successful you will see a message in the transcript window:

```
Simulation passed
```

If the display detail is too small, you can detach the window with the 2^{nd} button up right, maximize it and finally get a better overview with zoom buttons.

Push the *'Insert Cursor'*-button and position both cursors so that the whole *MAIN_LOOP* is embedded. The buttons *Find/Next/Previous Falling/Rising Edge* could be useful. If the optimization targets are met the embedded range should have a duration of 640,000 ps (corresponding of a 64 bit datastream and a throughput-time of 10 ns).

References

[1] Herrmann G. and D. Müller. *ASIC - Entwurf und Test*. Carl-Hanser-Verlag, 2004.

[2] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[3] D.D. Gajski and R.H. Kuhn. New VLSI Tools. *Computer*, 16(12):11–14, dec 1983.

[4] Michael Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corp., May 2010. http://www.eet.bme.hu/˜timar/data/hls_bluebook_uv.pdf.

”EDA Tools” Lab: High-Level-Synthesis
Version: January 19, 2015

Name: Signature
Group: Supervisor:

Taskpaper

Please solve the following tasks and bring this paper to your practical course. Not solving these tasks causes your exclusion of this lab!

Task 1 A design is transformed between which levels of the digital design by High-Level-Synthesis?

Task 2 What are the fundamental values that a synthesis process has to trade of between?

Task 3
Is this code fragment example synthesizable? Justify your answer!.

```
int function( int const n )
{ printf('Hello␣World'); return n == 0 ? 1 : n; }
```

Task 4
Draw a graph of a pipelined execution of the given loop.

```
#define B 19
#define C 23
void function(int x[], int y[])
{
for (int i=0; i<4; i++)
{ y[i]=x[i]*x[i] + B*x[i] + C; }
}
```