Digital Design and Synthesis of Embedded Systems

Summer Term 2025

Exercise 6 : Deadline 29.06.2025 - 23:59 o'clock

The solutions are submitted via ILIAS as a packed archive (ZIP/TAR.GZ). This ZIP contains the written answers to questions, the source code (not as a listing in the PDF), the simulation results as VCD files, and the Makefile. Each of your programming tasks **must** be executable with your Makefile or with a standard python interpreter. If your python solution uses non-standard packages you have to provide a requiretmens.txt file. If you do not follow this format, your handing may be graded with 0 points!

Exercise 1 Questions about the lecture [4 Points]

(a) You've seen that the allocation and scheduling step of the HLS are closely coupled. We've shown you an algorithm that can be used to estimate the amount of time steps given an allocation. However, it is also possible to use a slightly modified ASAP algorithm for this task. Your task is to re-imagine the ASAP algorithm such that it return the minumum amount of time-steps required, given an allocation. You can handin your solution in pseudocode. Next to the Graph $G(V, E)$, your algorithm receives the following inputs:

   (a) A list of ressources: ressource := $\{r_0, r_1, \dots\}$.
   (b) A function that returns the maximum amount of instances for a ressource: $i(r)$:= max number of instances.
   (c) A function that check if an operation $v \in V$ can be schedule on a ressource: $e(v, r) = \texttt{true}$ or `false`.

Exercise 2 Scheduling Algorithmen [8 Points]

In this task you will implement scheduling algorithms on your own. For this, we have provided you with a tiny framework which you can find it in the file `graph.py`. The framework is based on the widely used framework networkx. You will use this framework to implement the scheduling algorithms $ASAP$ and $ALAP(\lambda)$. At the beginning of the file you will find an enumeration of the supported operation types. The naming is based on the naming of the AST module of Python. The class `DSESGraph` functions as wrapper class around some networkx functionalities and includes the following information:

1. A hash map `node_attributes` that containts the IDs of added nodes to the graph and their attribues.
2. An instance of a *directed graph* with type `networkx.DiGraph` is stored in variable `_GRAPH`;
3. Helper functions to add nodes or query properties. (Check the code commends for more information).

We've provided you with an example in the main function that shows you how to create graphs within the framework. *Note:* This is only an example. You must define other graphs for your test cases.

(a) Familiarize yourself with the basic framework. Explain which node attributes are available. Describe how to add new nodes to the graph and how to create new connections (edges).

(b) Create a function *(*asap), which does not receive or return any values. Make sure to call the function `reset_graph()` in the first line of your function. Then implement the ASAP scheduling algorithm. Your algorithm should change the `scheduled_time` values of the nodes.

(c) Create a function *(*alap), which is passed a maximum execution time as an integer and does not return a value. Make sure to call the function `reset_graph()` in the first line of your function. Then implement the ALAP scheduling algorithm. Your algorithm should change the `scheduled_time` values of the nodes. The ALAP algorithm should stop with a `RuntimeError` if a schedule with the provided maximum execution time is not possible.

(d) Create **3** different graphs with at least 9 nodes and test your ASAP and ALAP implementation with them. The execution times of your nodes are freely selectable. With the function `schedule_to_dot(`*`filename`*`)` you can create a PDF with the current schedule. If the result looks suspicious, an error may have snuck into your implementation of the scheduling algorithms. The plotting function accesses the `scheduled_time` variables of the nodes for the correct display.

Note: To be able to change fields of a `namedtuple`, you must use the `_replace` function of the tuple. You can find the documentation here: `https://docs.python.org/3/library/collections.html`. You do not need to add a additional start and end node to your graph. FYI: The output can look wonky when you use any Apple products. The reasons for this is a different implementation of the dot-graph engine which we use to plot the graphs.

Exercise 3 High-level synthesis [8 Points]

In this task, you will implement the encryption/decryption design from exercise sheet 2 with an HLS. Write your CPP code in a file with the name `cbc.cpp`. You must include the header file ap_int.h, which allows you to define variables with a fixed bit size. Use a `ap_int<bitwidth>` as the base type and specialize the template with the required bit size. For this exercise, you will create two different implementations of the Cypher Block Chaining (CBC). As in exercise sheet 2, your design should differentiate between decryption and encryption using a `encrypt_decrypt` flag. It must be possible to switch during operation. To simulate your design you have to check the generated sourced by the HLS. Figure 1 shows you a reminder how the CBC should work. *Note:* Start with a relatively low clock freq. target of approx. 50ns.

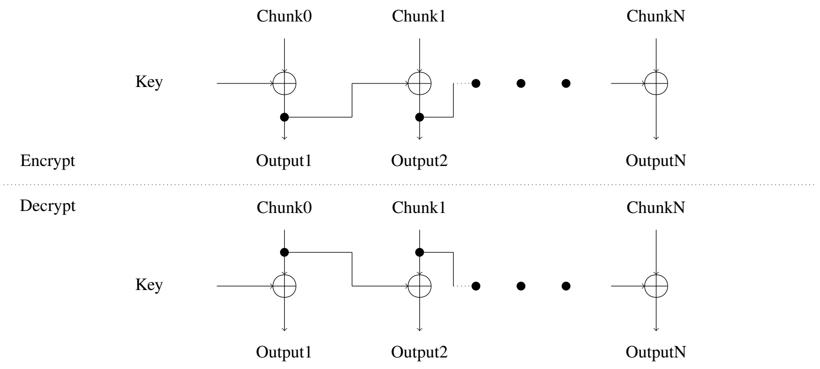

Figure 1: Cipher block chaining from exercise sheet 2

(a) As in exercise sheet 2, a message is made up of n blocks consisting of m bytes. Implement two variables using `#define`-statements that represent n and m.

(b) Create a function which is passed a block, returns a key of the same length and an encrypted block (input xor key).

(c) Create a function that receives a message and a key and returns an encrypted message. Please use the data type `bool` for the flag `encrypt_decrypt`. Your implementation should include a for loop that iterates over the individual blocks of the message. **Attention:** How you write code can have drastic effects on the quality of result for an HLS. In order for this exercise to work as intended, you have to write you code following the layout shown below:

```
    // Note: You have to choose the template parameters for the
    //       ap_int<...> accordingly. Also, don't forget to add
    //       the missing function arguments.
    ap_int<...> n_block_cbc(bool encrypt_decrypt, ...) {
     // Local variable declaration, no function calls
      ...;
     // Begin of compution
     for (...) {
        // Add your logic/function calls/etc here
     }
     // Return the result
     return ...;
    }

```

Write the following code above the line in which you call the function from the last subtask:

```
#pragma HLS inline
```

You can find here how this pragma influences the result.

(d) Perform the HLS and implement a SV testbench for your design. Write at least 4 test cases.

(e) Write the following code directly after the loop header of the for loop:

```
#pragma HLS unroll
```

You can find here how this pragma influences the result. Do you see any difference when you change the clock period constraint inside your Makefile? Please explain what you see.

(f) Perform the HLS and implement a SV testbench for your design. Write at least 4 test cases.

(g) Describe how your design changes after you have used the `unroll`-pragma.