Design Understanding: Identifying Instruction Pipelines in Hardware Designs

Lutz Schammer, Jan Runge, Paula Klimach, Goerschwin Fey
Institute of Embedded Systems, Hamburg University of Technology, 21073 Hamburg, Germany
{lutz.schammer,paula.klimach,goerschwin.fey}@tuhh.de

Abstract—Automated tools help a designer to reduce the time and the effort required to understand details of an unfamiliar design. In this paper we show an approach using static analysis to identify instruction pipelines, which are key structures in processor hardware designs. We present two algorithms which identify pipeline structures. The first algorithm is based on a structural analysis using a graph representation of the design, while the second algorithm uses terms and phrases for pipelines as they are found in literature for a name-matching approach. The two algorithms successfully identified pipelines for e.g. Y86 and edge processor designs.

Index Terms—Design Understanding, Static Analysis, Control-Flow, Verilog

I. INTRODUCTION

Designing hardware often requires understanding unfamiliar designs which costs time and effort. Insufficiently documented designs increase the cost even further. Design understanding is the process of extracting information from an unfamiliar design which helps understanding the behaviour of the design or identifying certain structures in the design. Automated tools help speedup design understanding, by automatically extracting information from a design and generating documentation or semantic annotations as shown in Figure 1. Design understanding is further explored in [1] and related problems are defined. Different approaches using both static analysis as well as dynamic analysis already exist to extract information from hardware designs. One such approach for design understanding is automatic extraction of information about a design, e.g. information about some distinctive structures of the design and in which lines of the source code parts of these structures were found.

Our vision is to provide tools for design understanding that go far beyond such simple extractions but rather provide a comprehensive set of knowledge for a design. One way to approach this are automated semantic annotations about architectural structures. As a first step we consider pipelines in processor designs.

This paper presents two algorithms that support design understanding by identifying instruction pipeline structures in Verilog hardware designs. While both algorithms are based on static analysis of the target designs, the first algorithm uses a structural approach analyzing control-flow graphs to recognize structures resembling instruction pipelines.

For comparing with a simple baseline, the second algorithm uses name-based pattern matching that identifies elements in the graphic representation that indicate the presence of parts of a pipeline structure. Both approaches provide a back annotation for the source code to identify pipelines found by the algorithms. This provides a starting point to the user for further investigation.

Our contributions include two algorithms which identify structures that are candidates for instruction pipelines and return information to the user representing where the structures can be found in the design.

The structure of this paper is as follows. Related work is presented in the next section. Section III presents the basic notations for the algorithms developed in this work. Section IV presents the first algorithm based on the structural approach while Section V presents the name-matching approach. Section VI presents experimental results for both algorithms for pipeline detection in different processor designs. The last section draws a conclusion of the gained results.

II. RELATED WORK

Previous studies describe dynamic and static analysis of hardware designs, which aim to automatically extract information from a design for different purposes, e.g., testing, verification, or design understanding.

In [2], a method for automatic generation of design properties is shown in which dynamic analysis of simulation traces is combined with constraints generated from the Register Transfer Level of the design by static analysis. Similarly the methodology called GoldMine [3], [4], uses the results of static analysis of register-transfer-level designs to guide a data mining process to find assertions for which the design can be tested.

A tool directly connected to Verilog is Pyverilog [5], written in Python and capable of analysing Verilog designs for control-flow and data-flow. The paper shows how a data-flow graph can be created from an abstract-syntact-tree created from a design and how the Control-Flow graph can be created based on the data flow graph. The work in [6] provides a syntax analyser tool, that utilizes Control-Flow and data flow to allow...
B. Control-Flow Analysis

Hardware design. Cells are connected by nets, while the cells are gates or module instantiations of the design. Ports are the input and the output connection of a module, where M, are all contained modules. Each module M of the design is defined as

\[ NL = \{ M_1, M_N \} \]

where M, are all contained modules. Each module M of the netlist consists of ports P, cells C and nets N.

\[ M = \{ P, C, N \} \]

Ports are the input and the output connection of a module, while the cells are gates or module instantiations of the hardware design. Cells are connected by nets.

B. Control-Flow Analysis

a) Control-Flow Graph: A control-flow graph \( G_{cf} \) is defined as

\[ G_{cf} = (V, E) \]

with V and E being the nodes and edges of the graph. The control-flow graph is created from the netlist where the nodes V contain all cells which are relevant for the control-flow of the design.

IV. STRUCTURAL INSTRUCTION PIPELINE DETECTION

A netlist for a hardware design is used to generate the control-flow graph for a design. The graph and netlist are used as the basis for the following algorithms to detect instruction pipelines in a given design.

A. Instruction Pipelines

Pipelines parallelize certain steps in a process thereby reducing the time required for a job. For processor pipelines this allows work on multiple instructions to be performed in parallel, e.g. while one instruction executes its arithmetic operation, the next instruction executes a memory access operation.

Figure 2 shows an example of processor pipeline construction. Each stage, illustrated in Figure 1, consists of cells and instruction registers. Cells implement the actual function of the stage and results are stored in intermediate registers. Whereas the instruction register stores the stage’s current instruction.

III. BASIC NOTATIONS

A. Netlist

A netlist defines a design by describing the modules of a design. A netlist is defined as

\[ NL = \{ M_1, M_N \} \]

where M, are all contained modules. Each module M of the netlist consists of ports P, cells C and nets N.

\[ M = \{ P, C, N \} \]

Ports are the input and the output connection of a module, while the cells are gates or module instantiations of the hardware design. Cells are connected by nets.

B. Control-Flow Analysis

Fig. 2. Sketch of an example pipeline, each stage consists of a shift register for the instruction and the functionality of the stage.

Many of the prior works have a focus on design verification using the obtained information for debugging or to infer design properties/assertions for testing purposes. Our focus lies in a different area by using the information obtained through static analysis to help the user to better understand the design on a behavioural level as well as on the implementation level.

The pseudo-code is given in Algorithm 1. In the initialization of the algorithm shown in line 2, the control-flow graph is created. In line 3, a subgraph of the control-flow graph is created containing only nodes connected to the clock signal. In line 5, all nodes that have the clock signal as input are traversed since each flip-flop in the pipeline is connected to the clock. After assuring that the currently investigated node represents a flip-flop, the corresponding node in the control-flow graph is determined as shown in line 7. This node is used as a starting point for the pipeline candidate detection. From this node, a subgraph of the control-flow graph is created,
This table shows examples for different patterns that occur in literature describing instruction pipelines.

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Source</th>
</tr>
</thead>
</table>

V. NAME-BASED PIPELINE IDENTIFICATION

The algorithm traverses the nodes of the netlist and compares the names of each node with the pattern-set shown in Table I. Additionally, for every match found an accuracy value is calculated. The described algorithm is sketched in the following, split into two parts. The first part is the pattern matching algorithm itself and the second one is the design traversal.

The pattern matching algorithm, presented in Algorithm 2, has three inputs in line 3, the pattern-set shown in Table I, the sequence inName in that a pattern should be found and an optional feedback value outAcc that allows to return the determined accuracy on demand. Since the pattern-set contains only terms in lower-case, the sequence inName is converted to lower-case in line 2, to ensure a possible match is detected despite differences in capitalisation. The return values and an index to traverse the pattern-set are initialised, as given in lines 3 to 5. Afterwards a loop iterates through all the patterns. In this loop, line 8 checks if inName contains the current pattern. If so, the accuracy of the found match is calculated by dividing the pattern length by the length of inName, shown in line 9. If the newly calculated accuracy is greater than the currently stored one, the stored one is replaced by the new one in line 10 and 11. The index of the pattern is stored in line 12. After the whole pattern-set was traversed, the match with the highest accuracy has been found. If the optional parameter for returning the accuracy is used, the accuracy of the found match is passed to this parameter as shown in lines 17 to 19. Finally, the index of the pattern with the highest accuracy is returned in line 20.

The second algorithm shown in Algorithm 3, the design must be processed. Additionally, to traverse the elements inside the design, the accuracy determined by the matching algorithm is used to filter the results to produce a certain quality. The algorithm takes as its inputs the netlist of the design that should be investigated, the pattern-set and the minimal accuracy that a match must have. All modules in the netlist are traversed. For each module, first the nets and cells are traversed. This is done in an identical way, the following description holds for lines 8 to 15 and lines 16 to 23. First, a variable accuracy is defined in line 4, used as the optional parameter of the findPatternMatch function to return the accuracy of the found pattern. The function findPatternMatch is called in line 5, given the pattern-set and the name of the element that is currently checked. If a match and by that an index is found, the accuracy is compared to the minimal accuracy. Depending whether the accuracy is greater or equal to the minimal value, the currently investigated element is annotated with the found match and that the element might be a pipeline part.

VI. EXPERIMENTS

This section discusses the results for different designs. The framework is executed on a laptop running Windows 10, using...
The three different processor designs that are analysed are based on different Instruction Set Architectures (ISAs). The first processor is a Y86-processor [12] extended with a pipeline, the second is an Edge processor [13] based on the MIPS architecture and the third one is the Ibex processor [14] based on the RISC-V ISA.

Table II shows the runtimes for the structural instruction pipeline detection. The table includes the number of nodes in the CFG, the number of flipflop nodes found and the runtime the algorithm needed for each design.

Table III shows how many cell and signal names were searched by the simple algorithm based on pattern matching for each of the three test designs together with the needed runtime.

A. Y86

Table V shows the number of pipeline candidates found by the structural analysis algorithm for the Y86 design for possible pipelines with 2, 5 and 6 stages. The pipeline candidates are further processed to identify the pipeline candidate which most likely corresponds to the pipeline. Of the three used designs the pipeline candidates that were identified for the basic Y86 design are analysed in more detail as an example. From the overall 36 pipeline candidates, 32 have a very similar structure. This structure is visualized in Figure 3, where these 32 pipeline candidates are combined to a graph structure again.

24 of the five-staged and all six-staged pipeline candidates start with the same two flipflops followed by the same multiplexer. This multiplexer is followed by three further multiplexers and a single flipflop which all differ throughout the candidates. The pipeline candidates consist of different numbers of flipflops and multiplexers. Only the number of flipflops is important to determine how many stages a pipeline might have. In Figure 3, this is shown by all the paths leaving the node MUX.0 and ending in MUX.4 and MUX.10. The differing flipflops are shown in one of the two multiplexers directly before the path ends. From the node MUX.4 on, the six-staged pipeline candidates contain the same two multiplexers followed by the same three flipflops shown as the very left path starting with MUX.5 and ending in DFF.10. For the five-staged pipeline candidates the two multiplexers are followed by one of three paths, each containing two multiplexers and two flipflops, which are different depending on the path. The five-staged pipeline candidates are shown in Figure 3 by the three paths next to the path of the six-staged candidates. One can see that the 32 candidates are the combination of each of the eight paths from the node MUX.0 to the nodes MUX.4 or MUX.10 and the four paths in which the graph ends. Using the netlist of this design, one can see that the second flipflop on all the paths is resettable, while the first one is not. The third flipflop on each path is resettable as well, but the fourth again is not. The inconsistency of the flipflops throughout the paths disqualifies all of them to be a pipeline structure since either a complete pipeline is resettable or it is not.

As yet unconsidered are the three two-staged and one five-staged pipeline candidate. From the netlist used for the analysis, it is observed that these two candidates are connecting a 32-bit D-flipflop to a 1-bit D-flipflop. This disregards them as possible pipelines since then only two bits of a 32-bit instruction would be forwarded to the next stage. The remaining two-staged candidate and the five-staged pipeline candidate represent basic shift registers, so both of them might be the instruction registers of a pipeline.

For the Y86 design the algorithm based on the simple pattern matching approach identified the signals IR_IF, IR_DEC, IR_EXE, IR_MEM and IR_WB among others as shown in Table IV. Since the signal names in the netlist are taken from the design, the signals can be searched in the design directly to
DFF_5
DFF_6
MUX_0
MUX_1 MUX_7 MUX_12 MUX_16 MUX_19 MUX_22 MUX_26 MUX_29
MUX_2
MUX_3
DFF_7
MUX_4 MUX_10
MUX_5 ...
DFF_17
MUX_20
MUX_21
DFF_18
MUX_23
MUX_24
DFF_19
DFF_20
DFF_21
MUX_27
MUX_28
DFF_22
MUX_30
MUX_31
DFF_23

Fig. 3. Simplified version of the graph that is obtained when combining all pipeline candidates found for the Y86 design with a common structure.

**TABLE V**

**NUMBERS OF PIPELINE CANDIDATES WITH THE SAME NUMBER OF STAGES FOR THE Y86 DESIGN FOUND BY THE STRUCTURAL APPROACH.**

<table>
<thead>
<tr>
<th>Stages</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline candidates</td>
<td>3</td>
<td>25</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

identify which structures these signals are a part of. Searching the desig indicates that the signals found are parts of a five-staged pipeline. Of the 230 signals identified the user needs to identify which signals are possibly part of the pipeline and can provide further information.

Both algorithms provide sufficient information to identify the pipeline in the Y86 design.

**B. Edge**

Table VI shows the number of pipeline candidates found by the structural analysis algorithm for the Edge design for possible pipelines with 2 to 9 stages. The identified pipeline candidates for the Edge design are processed to identify the most likely candidate to represent a pipeline. As shown in Table VI, more than 700 candidates were found. Again, it is noticeable that especially the large pipeline candidates have large common subgraphs. All candidates with seven or more stages start with one of three paths which end in the same multiplexer. These three paths are checked first instead of checking all 483 pipeline candidates with seven or more stages. All three paths take a value from a register and pass this value to a shift register containing three d-flipflops with multiplexers in between. In all shift registers found, the first d-flipflop is placed in the same module instance. The same holds for the second and the third d-flipflop of each shift register, while the module instances differ. This indicates that the pipeline stages are implemented in separate modules and that the pipeline candidates found represent data paths in the pipeline. To confirm this assumption, the remaining pipeline candidates are searched for other paths through the modules already found. Within the five- and six-staged candidates, more shift registers are found that have their single d-flipflop in the modules that were found before, strengthening the assumption that these modules represent pipeline stages. Comparing the modules found to the given design description and source code shows that the structure found indeed is the pipeline of the design. For example one of the found modules is called IF_ID_REG and is instantiated in line 536 in file Edge_core.v. The associated module is in file if_id_pipereg.v. Since each module is used to store the data for the next stage, the pipeline has five stages for the four modules found.

The simple algorithm based pattern matching found four cells in the Edge processor design shown in Table VII. Investigating the netlist yields that all four cells are representing module instantiations in the top-module of the design. Further, the inputs and outputs of these module instantiations indicate that an instruction is processed and that some outputs and inputs between these module instantiations are connected and data is propagated from one module to the next. These observations hint towards all four module instantiations being a part of a pipeline. Since the signals that the last stage could use are defined by the MEM_WB_REG instance, a fifth module instantiation with the name rf is found. Inspecting the modules of the design belonging to the cells, shows that the identified cells are part of a pipeline. The matches not considered up to here often describe signals to or from the pipeline stages, but they also contain random hits, especially in the lowest accuracy range.

The pipeline in the Edge processor design is identified by both approaches.

**C. Ibex**

The final design that is investigated is the Ibex processor. Table IX shows the number of pipeline candidates found by the structural analysis algorithm for the Ibex design for possible pipelines with 2 to 5 stages. First, the longest pipeline candidates are considered, containing four or five stages. These pipeline candidates all contain identical subgraphs except one single path. Investigating this single path further shows only
<table>
<thead>
<tr>
<th>Cell</th>
<th>if$_stage_i$</th>
<th>id$_stage_i$</th>
<th>ex$_block_i$</th>
<th>load$_store_unit_i$</th>
<th>wb$_stage_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>if$_stage$</td>
<td>id$_stage$</td>
<td>ex</td>
<td>store</td>
<td>wb$_stage$</td>
</tr>
<tr>
<td>Accuracy</td>
<td>80%</td>
<td>80%</td>
<td>20%</td>
<td>29.4%</td>
<td>80%</td>
</tr>
</tbody>
</table>

### Table IX

Numbers of pipeline candidates with the same number of stages for the Ibex design found by the structural approach.

<table>
<thead>
<tr>
<th>Stages</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline candidates</td>
<td>10</td>
<td>21</td>
<td>32</td>
<td>22</td>
</tr>
</tbody>
</table>

For the Ibex design the simple pattern-matching algorithm identified cells as the most promising elements. The three matches if$\_stage\_i$, id$\_stage\_i$ and wb$\_stage\_i$, as shown in Table VIII, already indicate that the cells with these names represent modules which are pipeline stages. Without further assumptions on the functionalities of the signals, investigating the design shows that many signals that are outputs of the if$\_stage\_i$ instance are inputs to the id$\_stage\_instance$ instance. Similar connections are found between the id$\_stage\_i$ cell and the wb$\_stage\_i$ cell together with connections from the id$\_stage\_i$ cell to the ex$\_block\_i$ cell. Investigating the modules belonging to these cells further, one can see that the input signals to the module of the id$\_stage\_i$ cell are not forwarded directly, but instead they are interpreted and new signals are created, which in turn are given to the next elements. This shows that the pattern matching algorithm is capable of identifying different pipeline structures which are constructed in a different way than the instruction pipelines shown in IV-A. The simple algorithm based on pattern matching successfully identified the pipeline for the Ibex processor design. The structural analysis algorithm identified only parts of the pipeline because the pipeline architecture differs from the architecture assumed by the algorithm.

### VII. Conclusion

This work introduced two algorithms for identifying instruction pipelines in hardware designs in an effort to assist design understanding. The first algorithm searches a graph representation of a design for subgraphs with specific elements. Since shift registers were introduced as a possible implementation for instruction pipelines, the algorithm searches for shift registers, basic as well as extended ones, in the graph. The resulting subgraphs are then given to the designer, who can investigate the design parts to which the subgraphs belong. The algorithm using a structural analysis approach identified the pipelines in two of the designs. The pipeline in the third design was only partially identified because the structural approach is limited by its definition of the pipeline architecture.

For design understanding purposes both approaches are helpful tools. The most helpful approach for design understanding is likely to use a combination of both algorithms to maximize the gained information from an unfamiliar design. These algorithms focus on pipelines as an example but similar approaches are applicable to other components of interest in unfamiliar hardware designs.

### References


