Improving Design Understanding of Processors leveraging Datapath Clustering

Katharina Ruep Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
{katharina.ruep, daniel.grosse}@jku.at

***Abstract*—In this paper, we present a novel approach for design understanding of processors. Our approach uses clustering techniques to identify datapath similarities based on control signal vectors. The resulting dendrogram captures the closeness of instructions wrt. their datapath and control in visual form. We demonstrate how our approach helps in design understanding of a RISC-V processor without reading the HDL code.**

I. Introduction

Becoming familiar with a processor design is a non-trivial task. While the programmer only looks from the perspective of the *Instruction Set Architecture* (ISA), the hardware designer/verification engineer has to understand the microachitecture and the HDL implementation. This can be done by simulating the processor and viewing the resulting waveforms, or using automated tools for *Design Understanding*. The challenges for such tools arise at different levels of the design hierarchy targeting specific problems [1]. Examples include feature localization in RTL descriptions [2], assertion mining at RTL [3], identification of instruction pipelines using static analysis on the netlist [4], or template-based understanding of circuit components [5]. However, all these approaches do not provide means for an abstract view on a processor. In general, a processor is divided into datapath and control where the latter tells the former what needs to be done. For gaining insight into the microarchitecure of a processor it is extremely helpful to understand which datapath is activated by the control unit for a given instruction, which instructions share datapath elements, and where the main differences between two instructions are. This is not only the case when one is unfamiliar with the processor design, but also when optimizations or processor extensions, such as the integration of a custom instruction, are scheduled.

In this paper, we present a novel automated approach for design understanding of processors. First, our approach performs coverage-guided fuzzing to generate a representative set of instructions for the processor at hand. These instructions are then simulated to extract the vector of activated control signals for each instruction. Next, we apply clustering techniques on the vectors to identify datapath similarities. The result is presented in form of a dendrogram, a hierarchical binary cluster tree which visualizes the closeness of the instructions: the closer instructions are to each other, the more datapath elements are shared (and activated by the control). In the experiments, we demonstrate the benefits of our approach for design understanding of a RISC-V processor.

Algorithm 1 Processor Design Understanding Algorithm

```
 1: $TestCases \leftarrow$ generate representative test cases with CGF
 2: $Traces \leftarrow$ get_traces($TestCases$)                  ▷ list of VCD-traces
 3: $Instructions, ControlVectors \leftarrow$ extract_from_traces($Traces$)
 4: $Clusters \leftarrow \varnothing$                            ▷ empty list of clusters
 5: for each $(cv, instr) \in ControlVectors, Instructions$ do
 6:     if $cv \notin Clusters$ then                             ▷ new cluster
 7:         $Clusters$.append($cv$)
 8:     end if
 9:     $Clusters[cv]$.append($Set(cv, instr)$)                 ▷ add to existing cluster
10: end for
11: $Clusters \leftarrow$ unify_clusters_per_instr($Clusters$)
12: $ClusterReps \leftarrow$ extract_cv_per_cluster($Clusters$) ▷ map of each cluster with
    a representative control vector
13: for each $cr \in ClusterReps$ do
14:     $cr.cv \leftarrow$ normalize($cr.cv$, $MaxBitWidthPerControlSig$)
15:     $cr.cv \leftarrow$ order_hierarchical($cr.cv$, $HierarchyPerControlSig$)
16: end for
17: $ClusterRelations \leftarrow$ linkage($ClusterReps$, 'average', 'euclidean_dist')
18: create_dendrogram($ClusterRelations$)
```

II. Design Understanding of Processors

Algorithm 1 shows the pseudo-code of our approach which consists of the following steps:

Input Generation (Line 1-2): Representative test cases for the processor are generated via *Coverage-Guided Fuzzing* (CGF). We utilize the fuzzer AFL++ [6] on top of the yosys-backend CXXRTL [7], and create traces for each test case.

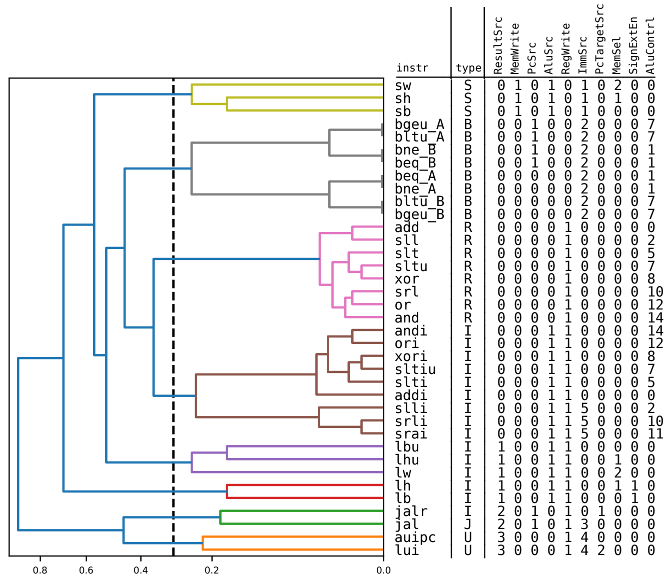
Information Extraction (Line 3): From each trace the instruction and all control signals (based on the interface between datapath and control unit) are extracted using WAL [8].

Datapath Clustering (Line 4-11): The initial clusters are created based on the control vectors, with each unique set of control signals creating a cluster. If multiple instructions end up in one cluster, these clusters are split, so that each cluster contains only elements with the same control vector and instruction.

Hierarchical Clustering (Line 12-17): Hierarchical clustering is performed to relate the clusters to each other. For this, representative sets of instructions and control vectors are extracted for each cluster. If several clusters represent the same instruction, a suffix (e.g.:`_A`,`_B`) is added to distinguish them. Each control signal then is normalized, i.e. it is divided by the largest representable value based on its bit width. In addition, the individual signals can be weighted in relation to each other according to their order in the design.

The agglomerative hierarchical clustering itself is done with SciPy [9] and based on distance calculation with euclidean metric for each control signal and average method to get a single distance over all signals between two clusters.

Visualization: (Line 18): To obtain a visual representation of the hierarchical clustering, a dendrogram is created.



instr	type	ResultSrc	MemWrite	PcSrc	AluSrc	RegWrite	ImmSrc	PcTargetSrc	MemSel	SignExtEn	AluContrl
sw	S	0	1	0	1	0	1	0	2	0	0
sh	S	0	1	0	1	0	1	0	1	0	0
sb	S	0	1	0	1	0	1	0	0	0	0
bgeu_A	B	0	0	1	0	0	2	0	0	0	7
bltu_A	B	0	0	1	0	0	2	0	0	0	7
bne_B	B	0	0	1	0	0	2	0	0	0	1
beq_B	B	0	0	1	0	0	2	0	0	0	1
beq_A	B	0	0	0	0	0	2	0	0	0	1
bne_A	B	0	0	0	0	0	2	0	0	0	1
bltu_B	B	0	0	0	0	0	2	0	0	0	7
bgeu_B	B	0	0	0	0	0	2	0	0	0	7
add	R	0	0	0	0	1	0	0	0	0	0
sll	R	0	0	0	0	1	0	0	0	0	2
slt	R	0	0	0	0	1	0	0	0	0	5
sltu	R	0	0	0	0	1	0	0	0	0	7
xor	R	0	0	0	0	1	0	0	0	0	8
srl	R	0	0	0	0	1	0	0	0	0	10
or	R	0	0	0	0	1	0	0	0	0	12
and	R	0	0	0	0	1	0	0	0	0	14
andi	I	0	0	0	1	1	0	0	0	0	14
ori	I	0	0	0	1	1	0	0	0	0	12
xori	I	0	0	0	1	1	0	0	0	0	8
sltiu	I	0	0	0	1	1	0	0	0	0	7
slti	I	0	0	0	1	1	0	0	0	0	5
addi	I	0	0	0	1	1	0	0	0	0	0
slli	I	0	0	0	1	1	5	0	0	0	2
srli	I	0	0	0	1	1	5	0	0	0	10
srai	I	0	0	0	1	1	5	0	0	0	11
lbu	I	1	0	0	1	1	0	0	0	0	0
lhu	I	1	0	0	1	1	0	0	1	0	0
lw	I	1	0	0	1	1	0	0	2	0	0
lh	I	1	0	0	1	1	0	0	1	1	0
lb	I	1	0	0	1	1	0	0	0	1	0
jalr	I	2	0	1	0	1	0	1	0	0	0
jal	J	2	0	1	0	1	3	0	0	0	0
auipc	U	3	0	0	0	1	4	0	0	0	0
lui	U	3	0	0	0	1	4	2	0	0	0

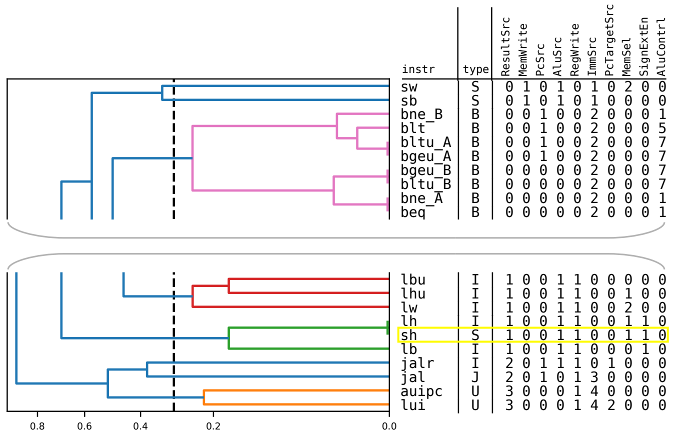
Fig. 1: Clusters with instructions, types and control vectors.

III. Experiments

We evaluated our design understanding approach on a RISC-V processor which has been implemented in VHDL and supports RV32I [10]. For this processor, coverage-guided fuzzing generated in total 7,215 unique testcases in 1 hour on an Intel Core i7-10700 with 64 GB of main memory. The subsequent steps of our approach took 83 seconds.

The dendrogram finally generated by our approach is depicted in Fig. 1. Each leaf of the tree in Fig. 1 is annotated as follows: `instruction | RISC-V type | control signal vector`. The vertical dashed line shows the current coloring threshold and gives the data set a meaningful clustering. On the right side of this line, each cluster has a different color. If we now move this line to the left or to the right, we essentially choose the granularity level of how we look at the datapath similarities of the processor. The further to the right a split occurs, the fewer differences are found in the underlying control vectors of the instructions, which means more datapath elements are shared. This is most evident with *branch*-instruction pairs. As a concrete example, consider the pair `beq_B` (branch equal) and `bne_B` (branch not equal) and ignore the suffix '`_B`' for a moment. Both instructions share the same control vector and are therefore grouped together. On the other hand, our approach also grouped together `beq_A` and `bne_A` as they also share the same control vector, but a different one as the pair before. The reason for this grouping is that in case of a conditional instruction, the branch is either taken or not, depending to the evaluation of the branch condition. Consequently, different datapaths have to be activated which leads to varying control vectors and finally to two different datapaths. Our approach eases understanding by identifying both of them (and therfore appends suffix `_A` and suffix `_B`, respectively).

Moreover, it can be seen in Fig. 1 that the clustering determined by our design understanding approach matches to



instr	type	ResultSrc	MemWrite	PcSrc	AluSrc	RegWrite	ImmSrc	PcTargetSrc	MemSel	SignExtEn	AluContrl
sw	S	0	1	0	1	0	1	0	2	0	0
sb	S	0	1	0	1	0	1	0	0	0	0
bne_B	B	0	0	1	0	0	2	0	0	0	1
blt	B	0	0	1	0	0	2	0	0	0	5
bltu_A	B	0	0	1	0	0	2	0	0	0	7
bgeu_A	B	0	0	1	0	0	2	0	0	0	7
bgeu_B	B	0	0	0	0	0	2	0	0	0	7
bltu_B	B	0	0	0	0	0	2	0	0	0	7
bne_A	B	0	0	0	0	0	2	0	0	0	1
beq	B	0	0	0	0	0	2	0	0	0	1
lbu	I	1	0	0	1	1	0	0	0	0	0
lhu	I	1	0	0	1	1	0	0	1	0	0
lw	I	1	0	0	1	1	0	0	2	0	0
lh	I	1	0	0	1	1	0	0	1	1	0
sh	S	1	0	0	1	1	0	0	1	1	0
lb	I	1	0	0	1	1	0	0	0	1	0
jalr	I	2	0	1	1	1	0	1	0	0	0
jal	J	2	0	1	0	1	3	0	0	0	0
auipc	U	3	0	0	0	1	4	0	0	0	0
lui	U	3	0	0	0	1	4	2	0	0	0

Fig. 2: Dendrogram with bug in `sh` instruction.

the RISC-V type specification[1] without using this information up-front/in the clustering algorithm. At this point, we want to discuss three insights: (1) For the S-type instructions `sw`, `sh`, and `sb` at the top of the Fig. 1 it can be seen that the underlying control vectors differ only at *MemSel* and therefore they are close to each other. (2) Less obvious is the grouping of the load counterpart (`lw`, `lh`, `lb`), as `lw` is closer to the S-type instructions than to the `lh`/`lb` pair. However, looking at their control vectors shows that the latter are the only instructions that set *SignExtEn* (to enable the sign-extension unit) and thus have a larger distance to `lw`. (3) The matching of the dendrogram/clustering to the RISC-V type specification can be interpreted as a lightweight validation of the implementation of the processor. If the clustering does not match, it potentially reveals a bug in the implementation. The dendrogram in Fig. 2 shows such a case. As can be seen, the S-type instruction `sh` (store half) has been grouped together with `lh` (load half), which obviously does not make sense. The reason was an incorrect setting of control signals due to a copy&paste error.

In summary, the generated dendrogram supports the user in design understanding of processors. The user gets an abstract view on the microarchitecture of the processor and can explore important behavior without reading HDL code.

References

[1] S. Ray, I. G. Harris, G. Fey, and M. Soeken, "Multilevel design understanding: from specification to logic," in *ICCAD*, 2016.
[2] J. Malburg, A. Finder, and G. Fey, "A simulation-based approach for automated feature localization," *TCAD*, vol. 33, no. 12, pp. 1886–1899, 2014.
[3] S. Vasudevan *et al.*, "Goldmine: Automatic assertion generation using data mining and static analysis," in *DATE*, 2010, pp. 626–629.
[4] L. Schammer, J. Runge, P. Klimach, and G. Fey, "Design understanding: Identifying instruction pipelines in hardware designs," in *MOCAST*, 2022.
[5] A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanović, and S. Malik, "Template-based circuit understanding," in *FMCAD*, 2014, pp. 83–90.
[6] "AFL++ - American Fuzzy Lop++," https://github.com/AFLplusplus/AFLplusplus.
[7] "yosys - Yosys Open SYnthesis Suite," https://github.com/YosysHQ/yosys.
[8] L. Klemmer and D. Große, "WAL: a novel waveform analysis language for advanced design understanding and debugging," in *ASP-DAC*, 2022, pp. 358–364.
[9] P. Virtanen *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.
[10] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.

[1]S: stores, B: conditional branches, R: register-register, I: short immediates and loads, J: unconditional jumps, U: long immediates