Compiladores - MIEIC

Assignment Report

The present report aims to describe the design and implementation of the project assignment for the course on Compilers taken by the authors at the Faculty of Engineering of the University of Porto. The initial assignment specifications were as follows:

This project intends to research techniques to analyze sequences of instructions executed by a microprocessor (e.g., ARM) and generate a dataflow graph exposing data-streaming computations and parallelism. The techniques used shall include memory disambiguation, extraction of address generation units and extraction of repeating patterns. The dataflow graph can then be used in the context of hardware accelerators.

This project will start by existent traces of instructions for a set of benchmarks (they will be provided by the staff). For validation of the project, the dataflow graphs generated by the tool to be developed need to be translated to a model that can be executed (e.g., a C/C++ or Java program).

The work developed focused on the trace files of the 32-bit soft processor, MicroBlaze™ produced by Xilinx®.

The Group

Link to repository - COMP2020-2C,

The authors

- **João Ruano Neto Veiga de Macedo**, up201704464@fe.up.pt
- **Miguel Maio Romariz**, up201708809@fe.up.pt

1. Overview

Throughout the development of this project, we faced many challenges leading us to reframe our objectives and processes to achieve them.

Initially, we conducted research on the instruction set and calling conventions used by the Microblaze processor. For this, we mainly used the MicroBlaze Processor Reference Guide (UG984 (v2018.2) June 21, 2018) made available by Xilinx.

Following this, we began the implementation available on our repository, aiming to translate the trace files to a readable graph. The parsing of the trace information was done using the parser generator JavaCC/JJTree.

We then translated this information into several analysis-friendly, readable formats.

1. A Control Flow Graph
2. A Compound Control and Data flow graph
3. A series of increasingly structured simulated C versions of the isolated function.

This implementation was fully done using Java. Of note, the Java library of graph theory data structures and algorithms JGraphT was used.

2. Parsing of the Trace files

2.1. Instruction set

All MicroBlaze instructions are 32 bits and are defined as either Type A or Type B.

In our implementation of the parser, we divided each type of instructions into sub-types, in order to facilitate the parsing of expressions. Type A instructions are divided into A1, A2 or A3 and type B are divided into B1, B2 or B3. In the documentation numbers are described as immediate values (imm) we will be referring to them as literals.

2.1.1. Type A Instructions

Type A instructions have up to two source register operands and one destination register operand.
A1 instructions use 1 register, A2 instructions use 2 registers and A3 instructions use 3 registers.

Type A1 Instructions:

BR; BRD; BRA; BRAD.

Type A1 example: BR Rd

Type A2 Instructions:

TNEAGETD; TNAPUTD; TNECAGETD; TNCAPUTD; FLT; FINT; FSQRT; DBL; DLONG; DSQRT; SRA; SEXT8; SEXT16; SEXTL32; CLZ; SWAPB; CLZ; SWAPH; WIC; WDC; WDC.FLUSH; WDC.CLEAR; WDC.CLEAR.EA; BRLD; BRALD; BRK; BEQ; BNE; BLT; BLE; BGT; BGE; BEQD; BNED; BLTD; BLED; BGTD; BGED.

Type A2 example: SEXT16 Rd,Ra

Type A3 Instructions:

ADD; RSUB; ADDC; RSUBC; ADDK; RSUBK; CMP; CMPU; ADDKC; RSUBKC; MUL; MULH; MULHU; MULHSU; BSRL; BSRA; BSLL; IDIV; IDIVU; FADD; FRSUB; FMUL; FDIV; FCMP.UN; FCMP.LT; FCMP.EQ; FCMP.LE; FCMP.GT; FCMP.NE; FCMP.GE; DADD; DRSUB * DMUL; DDIV; DCMP.UN; DCMP.LT; DCMP.EQ; DCMP.LE; DCMP.GT; DCMP.NE; DCMP.GE; OR; PCMPBF; AND; XOR; PCMPEQ; ANDN; PCMPNE; LBU; LBUR; LBUEA; LHU; LHUR; LHUEA; LW; LWR; LWX; LWEA; LL; LLR; SB; SBR; SBEA; SH; SHR; SHEA; SW; SWR; SWX; SWEA; SL; SLR.

Type A3 example: SW Rd,Ra,Rb.

Type B Instructions

Type B instructions have one source register and a 16-bit immediate operand (which can be extended to 32 bits by preceding the Type B instruction with an imm instruction). B1 instructions have 1 literal, B2 instructions use 1 literal and 1 register and B3 instructions use 2 registers and 1 literal.

2.1.2. Type B1 Instructions:

IMM; IMML; BRI; MBAR; BRID; BRAI; BRAID.

Type B1 example: BRAI Imm

Type B2 Instructions:

MSRCLR; MSRSET; RTSD; RTID; RTBD; RTED; BRLID; BRALID; BRKI; BEQI; BNEI; BLTI; BLEI; BGTI; BGEI; BEQID; BNEID; BLTID; BLEID; BGTID; BGEID.

Type B2 example: BRALID Rd,Imm

Type B3 Instructions:

ADDI; RSUBI; ADDIC; RSUBIC; ADDIK; RSUBIK; ADDIKC; RSUBIKC; MULI; BSRLI; BSRAI; BSLLI; ORI; ANDI; XORI; ANDNI; LBUI; LHUI; LWI; LLI; SBI; SJI; SWI; SLI.

Type B3 example: LHUI Rd,Ra,Imm

Additionally, there were some other instructions that don't fit any of the described types. Since none of them was used in the programs provided to us we simply didn't take them into account.

2.2 Registers

MicroBlaze has at its disposal 2 types of registers. It has 32-bit general-purpose registers and up to 18 32-bit special-purpose registers.

2.2.1. General Purpose Registers

The 32-bit general-purpose registers are numbered R0 through R31. There are different uses for each register wich are described below:

- R0 - Always has the value of zero. Anything written to it is discarded;
- R1-R13 - 32-bit general-purpose registers;
- R14 - Used to store return addresses for interrupts;
- R15 - Used to store return addresses for user vectors;
- R16 - Used to store return addresses for breaks;
- R17 - General purpose register, or in some cases, used for hardware exceptions;
- R18-R31 - 32-bit general-purpose registers.

2.2.2. Special Purpose Registers

- Program Counter (PC) - Is the 32-bit address of the executing instruction;
- Machine Status Register (MSR) - Contains control and status bits for the processor;
- Exception Address Register (EAR) - Stores the full load/store address that caused an exception;
- Branch Target Register (BTR) - Only exists if the MicroBlaze processor is configured to use exceptions. Stores the branch target address for all delay slot branch instructions;
- Floating-Point Status Register (FSR) - Contains status bits for the floating-point unit;
- Exception Data Register (EDR) - Stores data read on a link that caused a stream exception;

- Stack Low Register (SLR) - Stores the stack low limit use to detect stack overflow;
- Stack High Register (SHR) - Stores the stack high limit use to detect stack underflow;
- Process Identifier Register (PID) - Used to uniquely identify a software process during MMU address translation;
- Zone Protection Register (ZPR) - Used to override MMU memory protection defined in TLB entries;
- Translation Look-Aside Buffer Low Register (TLBLO) - Used to access MMU Unified Translation Look-Aside Buffer (UTLB) entries;
- Translation Look-Aside Buffer High Register (TLBHI) - Used to access MMU Unified Translation Look-Aside Buffer (UTLB) entries;
- Translation Look-Aside Buffer Index Register (TLBX) - Used as an index to the Unified Translation Look-Aside Buffer (UTLB) when accessing the TLBLO and TLBHI registers;
- Translation Look-Aside Buffer Search Index Register (TLBSX) - Used to search for a virtual page number in the UTLB;
- Processor Version Register (PVR) - Is controlled by the C_PVR configuration option on MicroBlaze;

3. Graph Generation

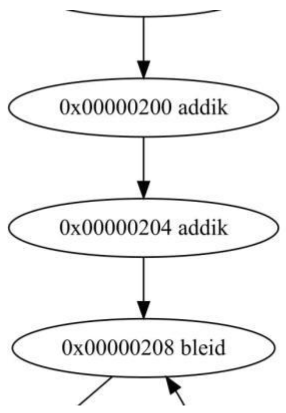
3.1 Generation of Control-flow graph

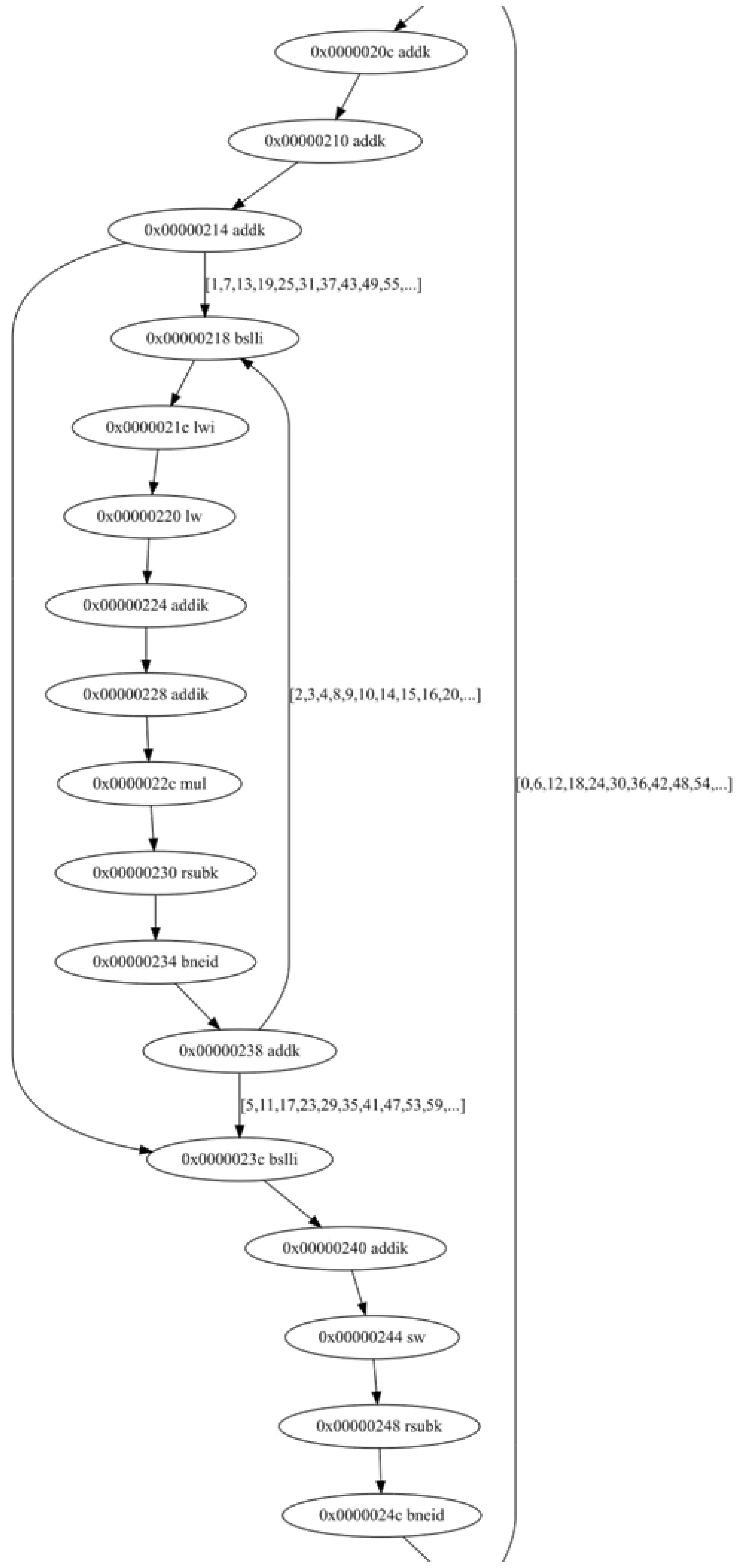
After completing the initial parsing phase using JavaCC/JJTree, we proceeded to incorporate in our project the library JGraphT, in order to facilitate the creation of complex graphs and following exporting in the DOT format.

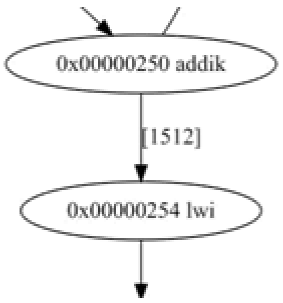
We then focused on managing the sheer amount of instructions in the trace file by creating a control flow graph that would only contain one instance of each instruction, while keeping all the information regarding the repeating patterns by condensing them into labels associated to the graph edges.

When a vertex has more than one outgoing edge, the labels indicate the path that was followed by the execution of the program generating the trace. If a reader imagines the instantiation of an integer variable that we will call "Path", initiated at 0, the path followed would be any non -labels edges first, and if not possible, solely the edge containing the label equal to Path. After traversing this edge, the variable "Path" is incremented and the same conditions apply.

This graph allowed us to easily visualize the loops in the program and paved the way to the extraction of C code, by structuring the information in a more optimal way.




0x0000020c addk
0x00000210 addk
0x00000214 addk
[1,7,13,19,25,31,37,43,49,55,...]
0x00000218 bslli
0x0000021c lwi
0x00000220 lw
0x00000224 addik
0x00000228 addik
[2,3,4,8,9,10,14,15,16,20,...]
0x0000022c mul
[0,6,12,18,24,30,36,42,48,54,...]
0x00000230 rsubk
0x00000234 bneid
0x00000238 addk
[5,11,17,23,29,35,41,47,53,59,...]
0x0000023c bslli
0x00000240 addik
0x00000244 sw
0x00000248 rsubk
0x0000024c bneid



3.2 Generation of the Compound Control and Data Flow Graph

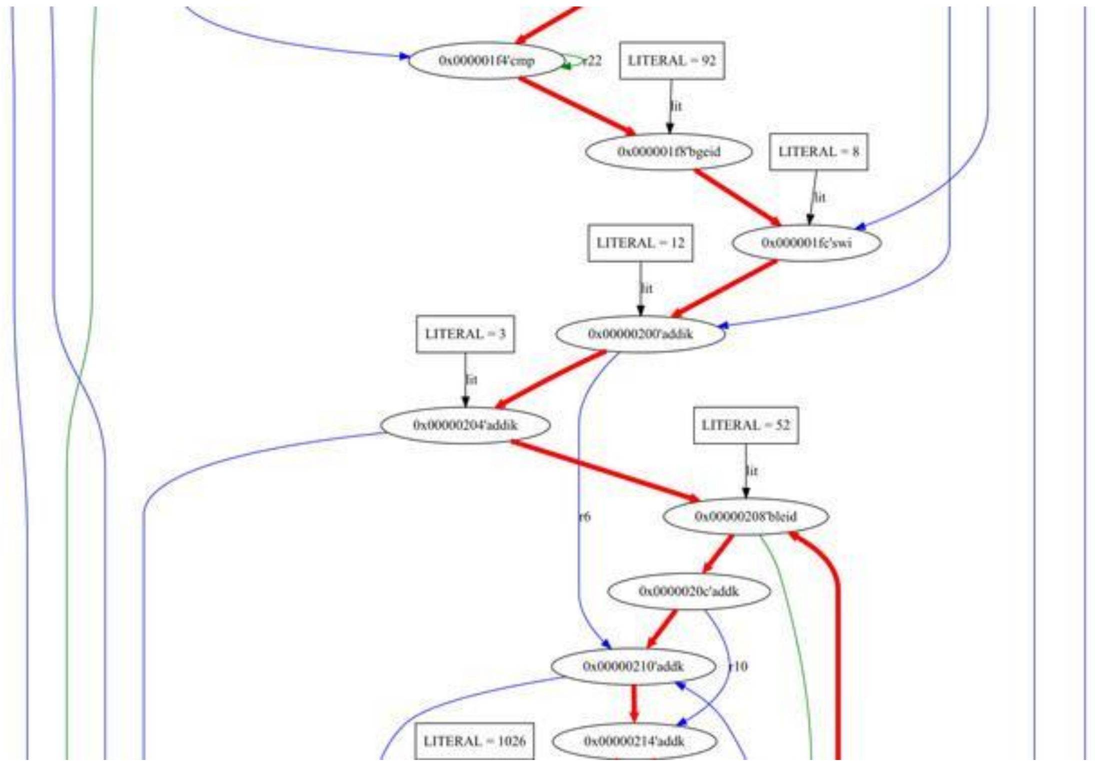
The first graph that was created gave us a lot of infomation about the structure of the executed program, but it certainly did not contain any regarding the flow of data throughout it. As such, we set off to create a Data Flow Graph.

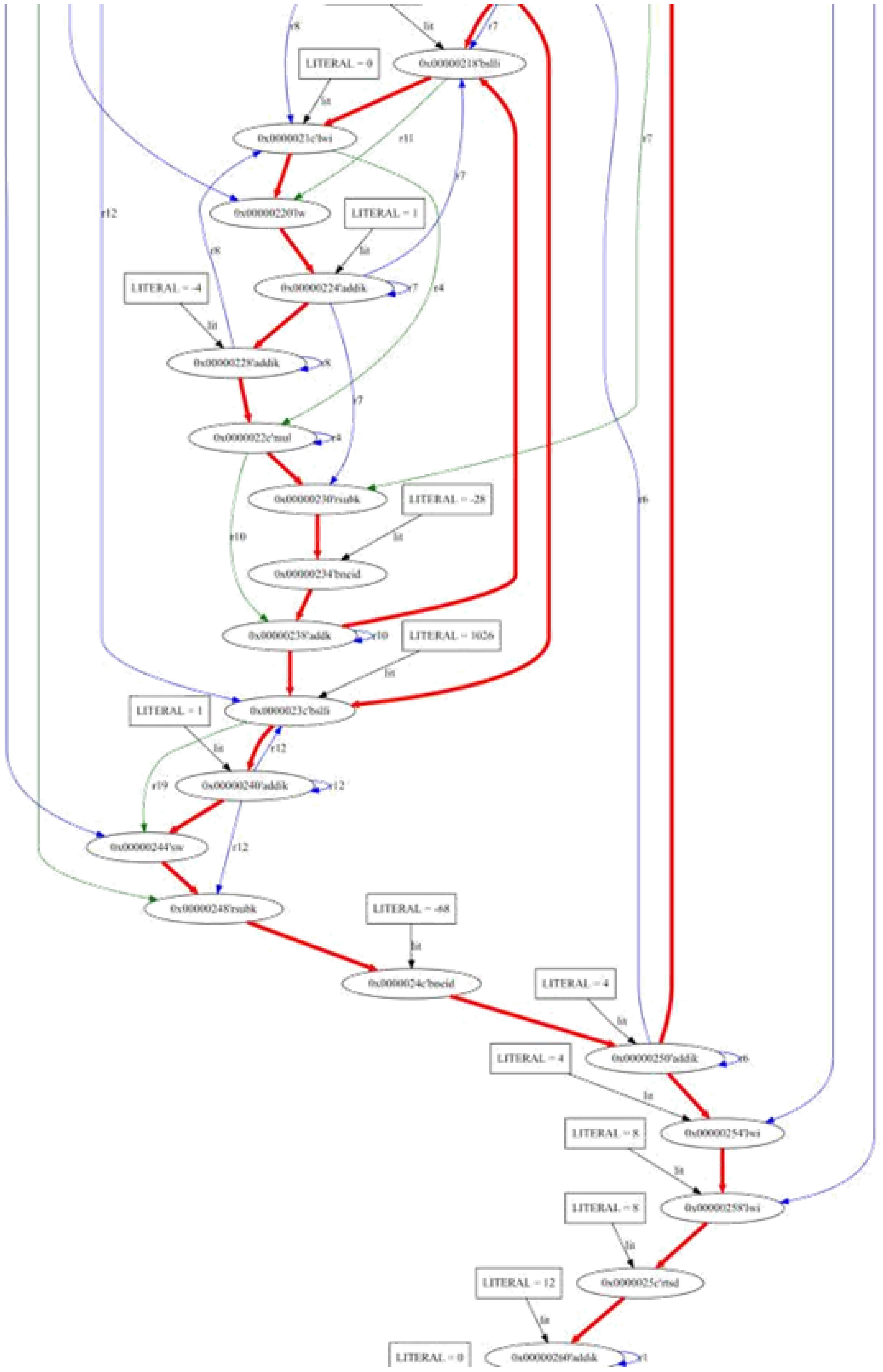
In our first iteration, we saved all instances of each executed instruction, leading to a graph that was not only unreadable but also of such size that it couldn't be properly displayed by the tool we used for the visualization of these graphs, GraphViz.

In our second iteration, we only displayed one instance of each instruction, leading to a graph with an obvious lack of structure, as the many disconnected nodes and the unrepresented repeating patterns hindered the ability of the user to analyse the progress of the execution of the instructions and the flow of the data itself.

Finally, we were able to create a graph that gave a structured representation of the program's execution and the flow of data throughout it by combining the edges of both the second iteration of the Data Flow Graph and the aforementioned Control Flow Graph, in what we ended up calling a Compound Control and Data Flow Graph. By merging the Control-flow graph with the Data-flow graph we gained valuable information about the flow of information within the program and allowed us to see clearly the path that the program took through all steps.

For reference, the round nodes represent the individual instructions and the four-sided polygon nodes represent the Literal values used by those instructions. The Red edges represent the Control Flow Graph edges, the blue edges represent the flow of data to the first input register used by the posterior node and the green edge represent the second input register.





4. Code Generation

4.1. Building Code Blocks

At this point we figured that the best step to transition into the generation of C code was by, firstly analyzing the traces and extracting all of the instructions that are used in the programs provided to us, and secondly referring to the MicroBlaze Processor Reference Guide and study each of the instructions in order to discover what they do individually. Finally, for each instruction, the corresponding C code is written.

Adding to this, we soon realised that the generation of a working C function did not consist solely on the aggregation of individually translated instructions, as things such as delayed instructions, memory simulation and interdependent instructions (of note, IMM), stood as obstacles to be transposed.

4.2. Generation of C code

At an early stage of production, we translated every line of the trace to the C equivalent, while handling the aforementioned executions, from the traversing of the Control Flow Graph.

An obvious flaw of this approach was the sheer length and lack of readability of the generated code because as with the first iteration of the Data Flow Graph every instance each instruction was written.

From the condensed information in the control-flow graph regarding program counter changes (jumps), we added goto instructions in the corresponding and labels at each instruction to simulate this control flow. At this point, we were faced with the reality of delayed instructions and we promptly switched the order of the delayed instructions and the ones that followed them.

Further improvements on readability came as we deleted all the unconditional jumps (as we are using the trace as our source these jumps were effectively goto instructions to the subsequent line) and deleted all unused labels.

The last step to structure the generated code was by substituting the goto statements, by implementing the outlined in section 2 of this paper.

4.3. Memory simulation

Simulation of memory was one of our most prominent challenges. We ended up abstracting this by creating a map through the library uthash. Additionally, we added the C functions “store” and “load” in the generated C files.

4.4. Using the Generated C code

To use the generated C code several steps are needed. Firstly, when generating the code the user will be prompted to input the number of variables in the function to be simulated, and a “Function isolation” integer. This input should represent the cardinality of the BRLID instruction that initiates the function in the trace.

A header file is then generated, from which the user should particularly make use of two functions. The function “generated”, in substitution of the original function. and the function “ storeInSimulatedMemory” with which the user can load arrays into the function’s simulated memory. Additionally, the user should cast to integer the pointers to arrays passed to the “generated” function.

Moreover, the user should open the generated C file and, if he wishes to use the passed arguments correctly, instead of the ones passed in the trace during the execution of the original program, copy the assignment of the passed parameters to the proper arrays to after the assignment made originally.

The generated function, in addition to simulating the original function, prints to the console all the memory access, as well as the value of the registers at the end of the execution.

5. Research on further improvements

5.1 Parallelism

A computer program is a stream of instructions executed by a processor. Without instruction-level parallelism, a processor can only issue less than one instruction per clock cycle (IPC < 1). These instructions can be re-ordered and combined into groups which are then executed in parallel without changing the result of the program. This is known as instruction-level parallelism.

All modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action the processor performs on that instruction in that stage; a processor with an N-stage pipeline can have up to N different instructions at different stages of completion and thus can issue one instruction per clock cycle (IPC = 1).

Most modern processors also have multiple execution units. They usually combine this feature with pipelining and thus can issue more than one instruction per clock cycle (IPC > 1).Instructions can be grouped together only if there is no data dependency between them. Scoreboarding and the Tomasulo algorithm are two of the most common techniques for implementing out-of-order execution and instruction-level parallelism.

5.2 Memory disambiguation

Memory disambiguation is a set of techniques employed by high-performance out-of-order execution microprocessors that execute memory access instructions (loads and stores) out of program order.

When attempting to execute instructions out of order, a microprocessor must respect true dependencies between instructions. And when executing loads and stores out of order can produce incorrect results if a dependent load/store pair was executed out of order.

So in order to resolve ambiguous dependencies and recover when a dependence was violated
Modern microprocessors use the following mechanisms:

- **Avoiding Write-After-Read (WAR) and Write-After-Write (WAW) dependencies** - Values from store instructions are not committed to the memory system when they execute. Instead, the store instructions, including the memory address and store data, are buffered in a store queue until they reach the retirement point. When a store retires, it then writes its value to the memory system.
- **Store to load forwarding** - In addition to buffering stores until retirement, the store queue serves a second purpose: forwarding data from completed but not-yet-retired ("in-flight") stores to later loads. When a load executes, it searches the store queue for in-flight stores to the same address that is logically earlier in program order. If a matching store exists, the load obtains its data value from that store instead of the memory system. If there is no matching store, the load accesses the memory system as usual; any preceding, matching stores must have already retired and committed their values. This technique allows loads to obtain correct data if their producer store has completed but not yet retired.

6. Conclusions

Even though we couldn’t reach all of the objectives that we set for ourselves, due to our team being cut short by two members mid-production, we believe that we achieved good results in our implementation.

Since our generated C code doesn’t translate directly from our Compound Control and Data Flow Graph, we feel that we failed to achieve our initial goal of validating the aforementioned graph through the generated code. But due to our situation, we gathered our thoughts and decided to take the project in a slightly different route creating the analysis tool.

There were some roadblocks during this process, some that we easily passed through like the use of delayed instructions which required some extra steps to be dealt with or the merging of the Control-flow graph and the Data-flow graph. But there were others which we couldn't conquer, like the correct translation of all of the instructions to C code, this was the main difficulty that we encountered and the one which mostly stopped us from reaching greater lengths.

Our work sheds light on the usefulness of translating trace files to C code, as the analysis of the contained information is done more effectively by running this code, adapting parameters and adding logs of the usage of the "store" and "load" functions.

7. References

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug984-vivado-microblaze-ref.pdf

https://troydhanson.github.io/uthash/userguide.html?fbclid=IwAR3oNYyJ4UHW9CtqySjQOU02F0Sz1FwrVyCRTglrXNFvHMKjltYb3QEq5-I

https://dzone.com/articles/goto-elimination-algorithm?fbclid=IwAR3jTJTIedkStvLiJOXPWEh-tgbVimkZiz6XyMkGx5u3Geru-a4y-QgoAbk

http://citeseer.ist.psu.edu/viewdoc/download;jsessionid=1B9F0F1D747EC90DA9ADAD03DEA392AF?doi=10.1.1.42.1485&rep=rep1&type=pdf