FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Optimization of ASM code for DLX using LLVM system
Student:	Bc. Michal Bureš
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2019/20

Instructions

DLX is a RISC processor load/store architecture. It is mostly used in universities as a model to study pipelining technique. LLVM is a modular compilerframework.
Tasks:
1) Use clang (frontend for C/C++) to generate C code representation in LLVM IR.
2) Implement a new backend for LLVM that transfers LLVM intermediate representation (LLVM IR) to DLX-Assembly code.
3) Analyze which optimization would be suitable for DLX.
4) Implement those optimizations on LLVM IR.
5) Evaluate the efficiency of the implemented optimizations.
6) Test the backend and optimizations with C code samples.

References

[1] Clang Compiler User's Manual: https://clang.llvm.org/docs/UsersManual.html
[2] LLVM Design & Overview: http://llvm.org/docs/
[3] Writing an LLVM Backend: https://llvm.org/docs/WritingAnLLVMBackend.html
[4] LLVM's Analysis and Transform Passes: https://llvm.org/docs/Passes.html
[5] John L. Hennessy and David A. Patterson: Computer Architecture: A Quantitative Approach, Second Edition ISBN: 978-7111074397
[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman: Compilers Principles Techniques and Tools ISBN: 0-321-48681-1
[7] Peter M. Kogge: Architecture of Pipelined Computers First Printing Edition ISBN: 978-0070352377
[8] Patty Sailer, David R. Kaeli, Philip M. Sailer The DLX Instruction Set Architecture Handbook ISBN: 978-1558603714

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague June 19, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Optimization of ASM code for DLX using LLVM system

Bc. Michal Bureš

Department of Theoretical Computer Science
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.

January 8, 2019

Acknowledgements

First of all, I would like to thank my supervisor doc. Ing. Ivan Šimeček, Ph.D. for all his advice. I would also like to thank all the people who helped me and supported me during the writing of this thesis with a special thank you to my family.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on January 8, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Bureš, Michal. *Optimization of ASM code for DLX using LLVM system.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato práce popisuje proces vytvoření nového backendu pro architekturu DLX pomocí LLVM kompilátoru. Prochází všemi nezbytnými součástmi tvorby nového backendu pro kompilátory, jako například výběr instrukcí nebo přiřazení registrů a popisuje je v rámci LLVM. Analyzuje, jak optimalizace fungují v systému LLVM a implementuje několik optimalizací vhodných pro tuto architekturu, například plánování instrukcí. Výsledkem této práce je nový LLVM backend s optimalizacemi pro architekturu DLX, který může být použit pro kompilaci určitých vyšších programovacích jazyků do DLX assembly kódu.

Klíčová slova DLX, kompilátor, backend, LLVM, optimalizace

Abstract

This thesis describes the process of creating a new LLVM compiler system backend for the DLX architecture. It goes through all the necessary parts of creating a new compiler backend such as instruction selection or register allocation and describes them in terms of LLVM. It looks into how optimizations work in the LLVM system and implement several optimizations suitable for the DLX architecture such as instruction scheduling. The result of this thesis is a new working LLVM backend for the DLX architecture with several optimizations in place. This backend can be used to compile several high-level languages to the DLX assembly code.

Keywords DLX, compiler, backend, LLVM, optimizations

Contents

List of Figures

List of Tables

Introduction

DLX is a RISC processor load/store architecture which is mostly used in universities to study the instruction pipelining technique in computer processors. There is no physical DLX processor for the DLX architecture as the architecture was mainly invented to study computer processor design. For this reason, the DLX architecture is very simple and uses a very simple instruction set to allow the understanding of how computer processors work and it is the reason why many universities around the world have been using this architecture to study a computer processor design, mainly focused on the pipelining technique. The pipelining technique is a technique where multiple instructions are overlapped in execution to speed up the program.

To understand how the pipelining technique works and how a computer processors work in general, it might be useful to see how some high-level programming languages (for example, the C programming language) which better describe the functionality of a program and are much more readable, get transferred to a simple low-level language like the DLX assembly code. This DLX assembly code can be put into a DLX simulator which can show how instructions are executed through the instruction pipeline and further improve the understanding of computer processors and of the pipelining technique. This means that some compiler is needed to translate the high-level language code to the DLX assembly code. This can be achieved by using the LLVM compiler system. The LLVM compiler system is a modern modular system built around the LLVM IR (LLVM intermediate language representation) and is lately gaining popularity as a compiler for C/C++ languages (for example, LLVM is used to compile Chrome on Windows). The LLVM design is divided into three main parts to support many source languages and many target architectures. The first part is a frontend which translates the input source code into the LLVM IR (Clang is a frontend for C/C++ languages). The second part is a backend which translates the LLVM IR into the machine code. To achieve the goal of compiling high-level languages to the DLX assembly code, a new backend for the DLX architecture needs to be created. The

process of creating a new LLVM backend can get quite complicated as the LLVM documentation is not always reliable and existing backends are for much more complicated architectures. Those backends are usually harder to understand when somebody is trying to implement a new LLVM backend but sometimes, they are the only place where to find out how the code generation process works in the LLVM system. So it might be quite useful to create a new LLVM backend for such a simple architecture which is what this thesis tries to accomplish. The goal of this thesis is to compile C programming language codes to the DLX assembly code.

The last part of LLVM is the LLVM optimizer which optimizes the LLVM IR. This optimizer is both source code and target code independent and thus could be used to optimize the code when compiling C programming language codes to DLX assembly codes. Apart from the LLVM optimizer, there are other places where the code can get optimized in the LLVM system. This thesis's second goal is to analyze how optimizations work and where the optimizations can occur in the LLVM system and to implement some of those optimizations for the DLX architecture.

The DLX backend, along with optimizations should then be tested and evaluated using C programming languages codes to check the correctness of the backend and to show that implemented optimizations can speed up the DLX assembly code.

Chapter 1

DLX Architecture

DLX (pronounced "Deluxe") is a RISC processor architecture designed by John L. Hennessy and David A. Patterson in an early 90's who introduced DLX as a simple architecture to study computer design. DLX is a simple load-store architecture which emphasizes design for pipelining efficiency as well as efficiency as a compiler target and uses a fixed instruction set encoding. The name DLX was derived from an average expressed in Roman numerals (AMD 29K, DECstation 3100, HP 850, IBM 801, Intel i860, MIPS M/120A, MIPS M/1000, Motorola 88K, RISC I, SGI 4D/60, Sparcstation-1, Sun-4/110, Sun-4/260)/13 = 560 = DLX[1].

1.1 DLX Registers

DLX architecture has 32 32-bit **General-purpose registers (GPRs)** named R0, R1, ..., R31 and 32 32-bit **Single-precision floating-point registers (FPRs)** named F0, F1, ..., F31 which can be used to store single-precision floating-point values or they can be used as even-odd pairs holding double-precision floating-point values. Thus, the 64-bit **Double-precision floating-point registers (DFPRs)** are named F0, F2, F4, ..., F30. Register R0 is always zero which can be useful for accessing memory or loading constants to registers[1].

There are also 3 miscellaneous registers: **Program Counter (PC)** which contains the address of an instruction currently being retrieved from memory for execution. The PC register can be altered by branches and jumps; **Interrupt Address Register (IAR)** that maintains the 32-bit return address of the interrupted program when a TRAP instruction is encountered; **Floating-Point Status Register (FPSR)** that provides for conditional branching based on the result of floating-point compare instructions. This register is used by floating-point branch instructions[2]. The table 1.1 shows all registers available in the DLX architecture and their sizes.

Table 1.1: DLX Registers[1]

Register type	Size	Note
General-purpose (R0, R1, ..., R31)	32-bit	
Floating-point (F0, F1, ..., F31)	32-bit (64-bit)	can be used to store double-precision values (F0, F2, F4, ..., F30)
Program Counter (PC)	32-bit	
Interrupt Address Register (IAR)	32-bit	
Floating-Point Status Register (FPSR)	1-bit	

1.2 DLX Data Types

The DLX architecture supports three data types for integers and two for floating-point numbers. The integer data types are an 8-bit byte, 16-bit half word, and 32-bit word. For floating-point numbers, a 32-bit value is used for single-precision floating-point numbers and a 64-bit value is used for double-precision floating-point numbers[1].

DLX instructions work either with 32-bit integers or 32-bit (64-bit) floating-point numbers. Bytes and half words are loaded into registers with either zero or the sign bit replicated to fill the 32 bits of the register. Once loaded, they are operated on as if they were 32-bit[1].

Table 1.2: DLX Data Types[1]

Data Type
8-bit (byte) integer
16-bit (half word) integer
32-bit (word) integer
32-bit single-precision floating-point number
64-bit double-precision floating-point number

1.3 DLX Addressing Modes

The DLX architecture supports two addressing modes: displacement and immediate. Both of them are using 16-bit fields to specify either the displacement offset or the immediate constant. The register deferral addressing can be accomplished simply by placing zero in the 16-bit displacement field and absolute addressing with a 16-bit field is accomplished by using the register R0 as the base register. That means that there are four ways how to address memory although only two are supported in the architecture[1].

The DLX memory is addressable by bytes with 32-bit addresses and is stored as Big Endian which indicates the most significant byte of the data is

placed at the byte with the lowest address[3]. As it is a load-store architecture, every memory cell can be loaded and stored using either GPRs, FPRs or DFPRs. Loading or storing the register R0 has no effect. To support all data types, memory access involving the GPRs can be to a byte, to a half word or to a word. All memory accesses must be aligned; access to an object of size S bytes at byte address A is aligned if A mod S = 0[4]. The table 1.3 shows all addressing modes mentioned above and their usage[1].

Table 1.3: DLX Adressing Modes[1]

Adressing mode	**Example instruction**	**Meaning**
Displacement	LW R1, 30(R2)	R1 ← Mem[30+R2]
Immediate	ADDI R1, R2, #3	R1 ← R2 + 3
Register deferral	LW R1, 0(R15)	R1 ← Mem[R15]
Absolute addressing	LW R1, 400(R0)	R1 ← Mem[400]

1.4 DLX Instruction Set

Every instruction in the DLX architecture is 32-bit wide with a 6-bit primary opcode and can be encoded using one of three formats: immediate (I-type) format, register-register (R-type) format or jump (J-type) format. These formats are very simple while providing 16-bit fields for displacement addressing and immediate constants and 16-bit fields (26-bit for J-type format) for branch and jump addresses. The figure 1.1 shows the exact layout for all those formats[1].

DLX instructions can be divided into four classes depending on their purpose: arithmetic/logical (ALU) instructions, floating-point instructions, data transfer instructions and control instructions. To describe single instructions in different classes, some symbols need be defined first[1]:

- $\leftarrow_8$ - a subscript to the symbol ← indicates how many bits are being transferred (in this case 8 bits are transferred).
- $R1_{8..16}$ - a subscript to a register is used to indicate selection of only certain bits. Bits are labeled from the most-significant bit started at 0. The value can be a single value $R4_0$ or a subrange $R4_{24..31}$, which yield the least significant byte of R4.
- Mem stands for main memory, is indexed by a byte address and may transfer any number of bytes.
- A superscript is used to replicate a field (0^{24} replicates zero to 24 zeros).
- The symbol ## concatenates two fields.

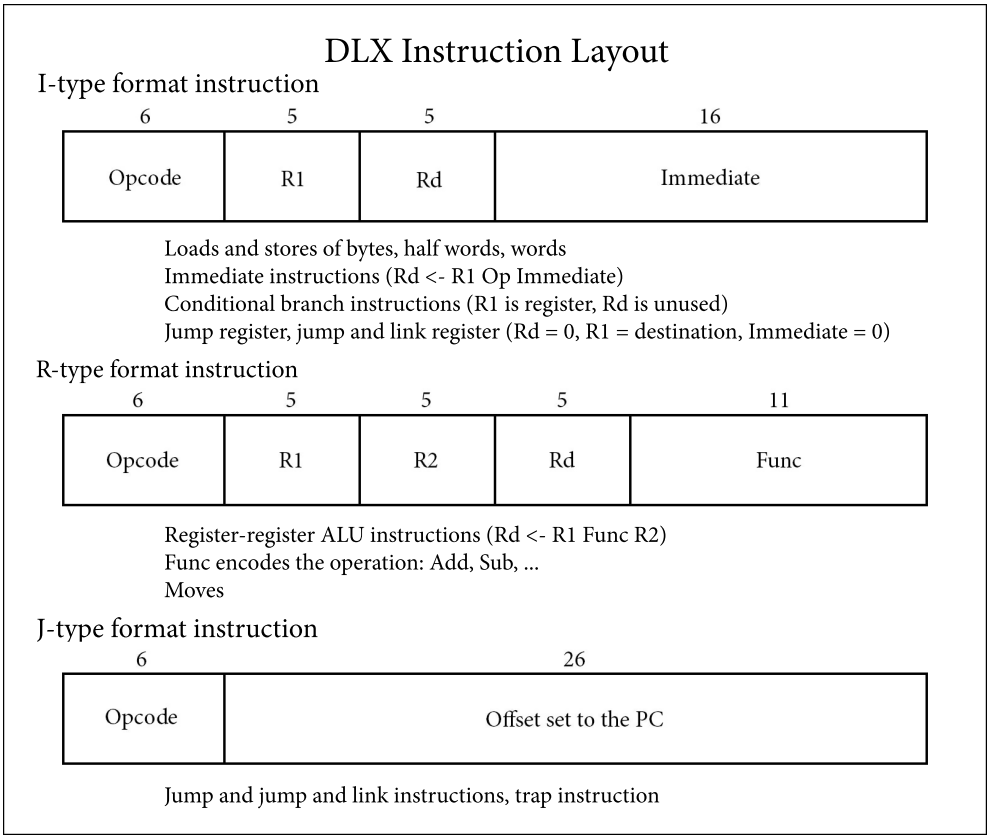

Figure 1.1: DLX Instruction Layout[1]

ALU instructions represent simple operations such as add, subtract, multiply, divide, AND, OR, XOR and various shifts: left logical, right logical and right arithmetic. Both register-register and register-immediate forms are provided. The LHI (load high immediate) instruction loads the top half of a register and sets the lower half to zero, for example, this allows a full 32-bit address to be built. There are also compare instructions that compare either two registers or a register and a 16-bit immediate constant ($=, \neq, <, \leq, >, \geq$). In case a condition is true, the instruction places one into the designated register to represent true otherwise, it places zero to represent false. The result of compare instructions can be tested with two branch instructions: branch equal zero (BEQZ) and branch not equal zero (BNEZ). The branch target address is specified with a 16-bit offset that is set to the program counter if the branch is taken. The table 1.4 shows some examples of ALU instructions[1].

Floating-point instructions operate only on floating-point registers and come both in a single-precision variant and a double-precision variant. A suffix D is used to indicate double-precision floating-point registers are used and

Table 1.4: Examples of DLX ALU Instructions[1]

Instruction	Description	Meaning
ADD R1, R2, R3	Add	R1 $\leftarrow_{32}$ R2 + R3
ADDI R1, R2, #3	Add immediate	R1 $\leftarrow_{32}$ R2 + 3
LHI R1, #42	Load high immediate	R1 $\leftarrow_{32}$ 42 ## 0^{16}
SLLI R1, R2, #5	Shift left logical	R1 $\leftarrow_{32}$ R2 « 5
SLT R1, R2, R3	Set less then	if (R2 < R3) R1 $\leftarrow_1$ 1 else R1 $\leftarrow_1$ 0

a suffix F is used for single-precision floating-point registers. Floating-point instructions consist of add, subtract, multiply and divide (e.g., ADDD, ADDF, SUBD, SUBF, MULTD, MULTF, DIVD, DIVF). There are also instructions to compare floating-point numbers such as LTD (lower than double), GEF (greater equal float) and so on. Those instructions set the FPSR register that can be tested with a pair of branch instructions: branch floating-point true (BFPT) and branch floating-point false (BFPF)[1]. Floating-point numbers use a format defined in *The IEEE Standard for Floating-Point Arithmetic* (IEEE 754)[5].

It is also possible to copy between floating-point registers using MOVF and MOVD instructions and convert registers using CVTx2y instructions, where x, y can stand for: I (integer), F (single-precision floating-point register) or D (double-precision floating-point register). Those convert instructions only take floating-point registers as operands so to actually convert an integer to a floating-point value, the integer register must be first moved to a floating-point register using the MOVI2FP instruction[1]. The figure 1.5 shows some examples of floating-point instructions.

Table 1.5: Examples of DLX Floating-point Instructions[1]

Instruction	Description	Meaning
ADDD F2, F4, F6	Add two double-precision floating-point numbers	F2 $\leftarrow_{64}$ F4 + F6
DIVF F1, F3, F2	Divide two single-precision floating-point numbers	F1 $\leftarrow_{32}$ F3 / F2
CVTF2I F1, F0	Convert a single-precision floating-point number to an integer	F1 $\leftarrow$ F0
EQF F0, F5	Compare two single-precision floating-point numbers and accordingly set the FPSR register	if (F0 = F5) FPSR $\leftarrow_1$ 1 else FPSR $\leftarrow_1$ 0

The DLX architecture provides load and store instructions for all data types mentioned in the section 1.2. Loading a byte or a half word from memory can

be done either as signed or unsigned. In the case of the signed loading, the signed bit will be replicated to the corresponding bits, in case of the unsigned loading the zero bit will fill the corresponding bits[1]. Some examples of data transfer instructions are shown below in the table 1.6.

Table 1.6: Examples of DLX Data Transfers Instructions[1]

Instruction	Description	Meaning
LW R1, 30(R2)	Load world	R1 $\leftarrow_{32}$ Mem[R2 + 30]
LB R1, 40(R3)	Load byte	R1 $\leftarrow_{32}$ $(\text{Mem[R3 + 40]}_0)^{24}$ ## Mem[R3 + 40]
LBU R1, 40(R3)	Load byte unsigned	R1 $\leftarrow_{32}$ 0^{24} ## Mem[R3 + 40]
LF F0, 60(R3)	Load float	F0 $\leftarrow_{32}$ Mem[R3 + 60]
LD F0, 60(R4)	Load double	F0##F1 $\leftarrow_{64}$ Mem[R4 + 60]
SW 200(R4), R3	Store world	Mem[R4 + 200] $\leftarrow_{32}$ R3
SD 40(R3), F0	Store double	Mem[R3 + 40] $\leftarrow_{32}$ F0 ; Mem[R3 + 44] $\leftarrow_{32}$ F1
SB 44(R1), R5	Store byte	Mem[R1 + 44] $\leftarrow_{8}$ $\text{R5}_{24...31}$

Control flow is handled through a set of jumps and a set of branches. The four jump instructions are differentiated by the two ways to specify the destination address and by whether or not a link is made. Two of those jumps use a 26-bit offset to set the program counter. The other two use a register that contains the destination address. There are two flavors of jumps: plain jump and jump and link (used for procedure calls). The latter places the return address - the value PC + 4 in the register R31[1]. The table 1.7 gives some examples of control instructions[1]. The whole instruction set is specified on the figure 1.8.

Table 1.7: Examples of DLX Control Instructions[1]

Instruction	Description	Meaning
J name	Jump	PC $\leftarrow$ name; name $\geq$((PC+4)-2^{25}) && name $<$ ((PC+4)$+2^{25}$)
JAL name	Jump and link	PC $\leftarrow$ name; R31 $\leftarrow$ PC + 4; name $\geq$ ((PC+4)-2^{25}) && name $<$ ((PC+4)$+2^{25}$)
JR R3	Jump register	PC $\leftarrow$ R3
BEQZ R6, name	Branch equal zero	if(R6 = 0) PC $\leftarrow$ name; name $\geq$ ((PC+4)-2^{15}) && name $<$ ((PC+4)$+2^{15}$)
BFPT name	Branch floating-point true	if(FPSR = 1) PC $\leftarrow$ name; name $\geq$ ((PC+4)-2^{15}) && name $<$ ((PC+4)$+2^{15}$)

Table 1.8: DLX Instruction Set[1]

Instruction type/opcode	**Instruction meaning**
Arithmetic/logical	**Operations on integer or logical data in GPRs**
ADD, ADDI, ADDU, ADDUI	Add, add immediate (all immediates are 16 bits); signed and unsigned
SUB, SUBI, SUBU, SUBUI	Subtract, subtract immediate; signed and unsigned
MULT, MULTU, DIV, DIVU	Multiply and divide; signed and unsigned; all operations take and yield 32-bit values
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LHI	Load high immediate; loads upper half of registers with immediate
SLL, SRL, SRA, SLLI, SRLI, SRAI	Shifts; both immediate (S__I) and variable form (S__); shifts are shift left logical, right logical and right arithmetic
S__, S__I	Set conditional: "__" may be LT, GT, LE, GE, EQ, NE
Floating-point	**FP operations on DP and SP numbers**
ADDD, ADDF	Add DP, SP floating-point numbers
SUBD, SUBF	Subtract DP, SP floating-point numbers
MULTD, MULTF	Multiple DP, SP floating-point numbers
DIVD, DIVF	Divide DP, SP floating-point numbers
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F, CVTI2D	Convert instructions; CVTx2y converts from type x to type y. Both operands are FPRs or DFPRs
__D, __F	DP and SP compares; "__" may be LT, GT, LE, GE, EQ, NE; sets bit in the FP status register
Data transfers	**Move data between registers and memory, or between registers**
LB, LBU, SB	Load byte, load byte unsigned, store byte
LH, LHU, SH	Load half word, load half word unsigned, store half word
LW, SW	Load word, store word
LF, LD, SF, SD	Load float, load double, store float, store double
MOVF, MOVD	Copy one FP register or DP pair to another register or pair
MOVFP2I, MOVI2FP	Move 32 bits from/to FP registers to/from integer registers
Control	**Conditional branches and jumps; PC-relative or through registers**
BEQZ, BNEZ	Branch GPRs equal/not equal to zero; 16-bit address
BFPT, BFPF	Test comparison bit in the FP status register and branch; 16-bit address
J, JR	Jumps; 26-bit address or target in register (JR)
JAL, JALR	Jump and link; save PC + 4 in the register R31, target is an address (JAL), or a register(JALR)
TRAP	Transfer to operating system at a vectored address

1.5 DLX Memory Layout

DLX is a very simple architecture and uses a very simple memory layout which divides the memory into three segments: stack segment, data segment, and text segment. Figure 1.2 shows the exact memory layout used by the DLX architecture[1].

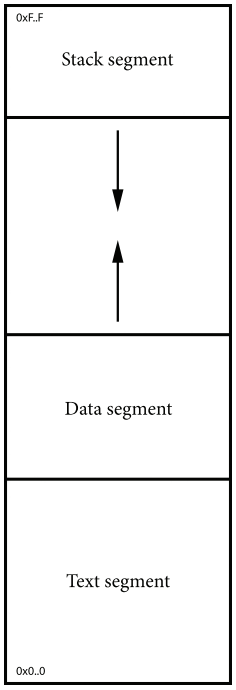

Figure 1.2: DLX Memory Layout

- **Text segment** - This is where the machine language of user code is stored. Text segment has fixed length.
- **Data segment** - This holds data that the program operates on. Data can be divided into two types: static and dynamic. Static data does not change its size during execution and their size is known during compilation. Dynamic data are allocated and deallocated as the program executes and their size is not known during compilation. Data segment grows from lower addresses to higher addresses[6].
- **Stack segment** - Stack segment is used to store stack frames which contain local variables, return addresses and arguments of procedure calls. Stack segment grows from higher addresses to lower addresses. It is necessary to keep track of the size of the stack segment, for that, the register R30 is used (**stack pointer**).

1.6 DLX Calling Convention

A calling convention is an agreement about how procedures (functions) are called and how control is returned to the caller. A procedure is a subroutine, consisting of several lines of code that performs a specific task based on arguments that it has been provided with and might or might not return a result. Every procedure also needs some local memory associated with it to hold incoming arguments, local variables, return address and any other data that the procedure needs to properly run. This part of memory is called a **stack frame**[7]. Each time a procedure is called, a new stack frame is created (called function prologue) and at the end of a procedure, the stack frame must be destroyed (called function epilogue). Stack frames are mainly needed for nested calls as all the data needed to run a procedure, such as a return address or a frame pointer, are saved in a stack frame and will not get overwritten by a nested call. The form of a stack frame is important as it defines how the callee and the caller pass its data. The figure 1.3 shows the exact layout of a stack frame in DLX[8].

A **Frame pointer** will point to the memory location of the first argument which is handy because the stack pointer might change during the execution of a procedure but the frame pointer cannot. This means that during the execution of a procedure, arguments and local variables can be addressed by a fixed offset from the frame pointer. The frame pointer is stored in the register R29[8].

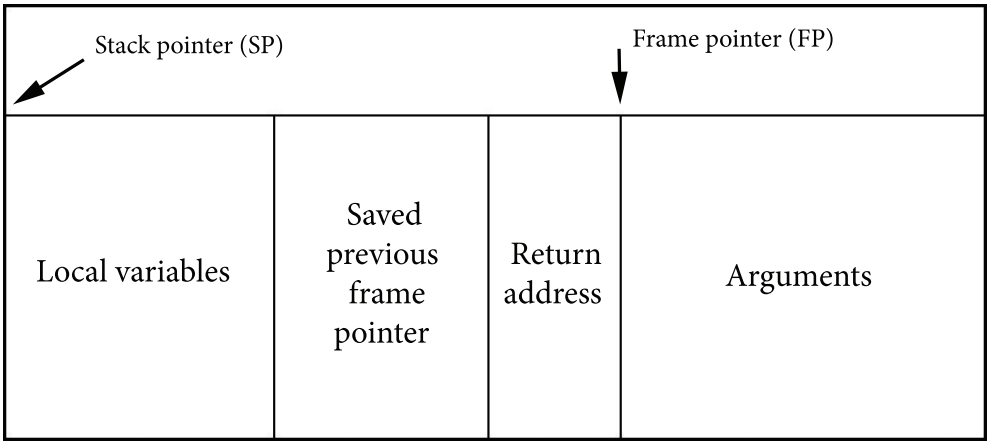

Figure 1.3: DLX Stack Frame Layout

Both the caller and the callee must abide by the defined calling convention in order to correctly pass data and control flow. When calling a procedure, the caller must first place arguments that the callee requires on the stack and to call the procedure it must use the JAL (JALR) instruction which places

the value of PC + 4 to the register R31. The callee must then perform the following tasks to properly run and to return control flow back to the caller[8]:

1. Initialize a new stack frame.
2. Store the return address from the register R31 to the stack frame; DLX architecture does not provide direct access to the PC register so it must be the callee who stores the return address after the jump and link instruction placed it in the register R31.
3. Save the previous frame pointer to the stack frame.
4. Set the new frame pointer to point at the first argument.
5. Allocate the space for local variables.
6. Execute its code using arguments referenced by the frame pointer.
7. If the procedure returns a value, place it in a defined register; for integer values, registers R1 to R4 are used, allowing to return values up to 128 bits. The register F0 is used to return floating-point values.
8. Load the return address from the stack frame and restore the previous frame pointer.
9. Return the control flow to the caller by jumping to the return address.

1.7 DLX Directives

Assembler directives are defined as a dot followed by a name and are used to direct the assembler on how to exactly translate a program. A directive is not translated into any specific machine language instructions[8]. DLX uses following directives[2]:

- **.align** n - Cause the next data/code loaded to be aligned at the next higher address with the lower n bits zeroed.
- **.ascii** ”string1”, ”...” - Store the strings listed on the line in memory as a list of characters. The strings are not terminated by a 0 byte.
- **.asciiz** ”string1”, ”...” - Similar to .ascii except each string is terminated by a 0 byte.
- **.byte** byte1, byte2 - Store the bytes listed on line sequentially in memory.
- **.data** [address] - Cause the following code to be stored in .data segment. It allows to set the exact memory location.

- **.double** number1, ... - Store the numbers listed on the line sequentially in memory as double-precision floating point numbers.
- **.float** number1, ... - Store the numbers listed on the line sequentially in memory as single-precision floating point numbers.
- **.global** label - Make the label available for reference.
- **.space** size - Move the current storage pointer forward size bytes (to leave some space in memory)
- **.text** [address] - Cause the following code to be stored in the text(code) area.
- **.word** word1, word2 - Store the word listed on the line sequentially in memory

1.8 DLX Simulators

Since there is no real physical DLX (at least not a fully functioning one) processor, a simulator is needed for running and testing the DLX assembly code. The following DLX simulators were tested to decide which simulator is the most suitable:

- **DLXOS**[9]
  - Small operating system running on DLX simulator (DLXSim).
  - Implements DLX instruction set exactly as defined.
  - Very outdated - 20 years old, does no compile on Solaris or Debian.
  - Not much of a documentation.
  - Uses traps to support functions open(), close(), read(), write() and printf().
- **WinDLX**[10]
  - WinDLX is a 16-bit Windows-based simulator for DLX.
  - Implements DLX instruction set exactly as defined.
  - It needs to be installed on Windows 98 or less - works fine on Windows 95 (OSR2).
  - Uses traps to support functions open(), close(), read(), write() and printf().
  - Low memory limit which can only be set between 0x200 and 0x1000000; maximum 16 MB.

- **DLX Simulator**[11]
  - Created by David Viner as a final project at the University of East Anglia in Norwich.
  - Does not always follow the DLX architecture.
- **OpenDLX**[12]
  - A DLX/MIPS processor simulator written in Java.
  - Does not fully support DLX instruction set.
  - Does not have floating-point registers or floating-point instructions.
- **DLXsim**[13]
  - Does not compile on Solaris or Debian.
  - No source code documentation.
- **DLXwsim**[14]
  - On-line compiler at the Department of Computer Science at the University of Salzburg.
  - Only source codes available, no documentation.
  - Appears to be an edited version of the older DLXsim[13] working on current Unix systems.
  - Does not allow setting up the pipeline configuration - could be set in the code but that might cause some unexpected behavior.

After testing these DLX simulators, the **WinDLX** simulator seems to be the best choice for running and testing DLX assembly codes. For once it completely respects the DLX architecture as defined in J. Hennessy and D. Patterson's book[1], allows setting up the pipeline configuration, has the best visualization of the pipeline and is stable. However, there are two setbacks of using WinDLX. Since it has to be run on Windows 98 or less, writing automated tests might be quite difficult and some parts of testing might have to be done manually. Also, the memory limit is quite low but 16 MB should be enough to run basic programs. The figure 1.4 shows the WinDLX GUI.

1.9 WinDLX Traps

WinDLX has 5 traps to build an interface between DLX programs and I/O. Zero is an invalid argument for a trap instruction, used to terminate program[15].

- **Trap #0: Invalid**

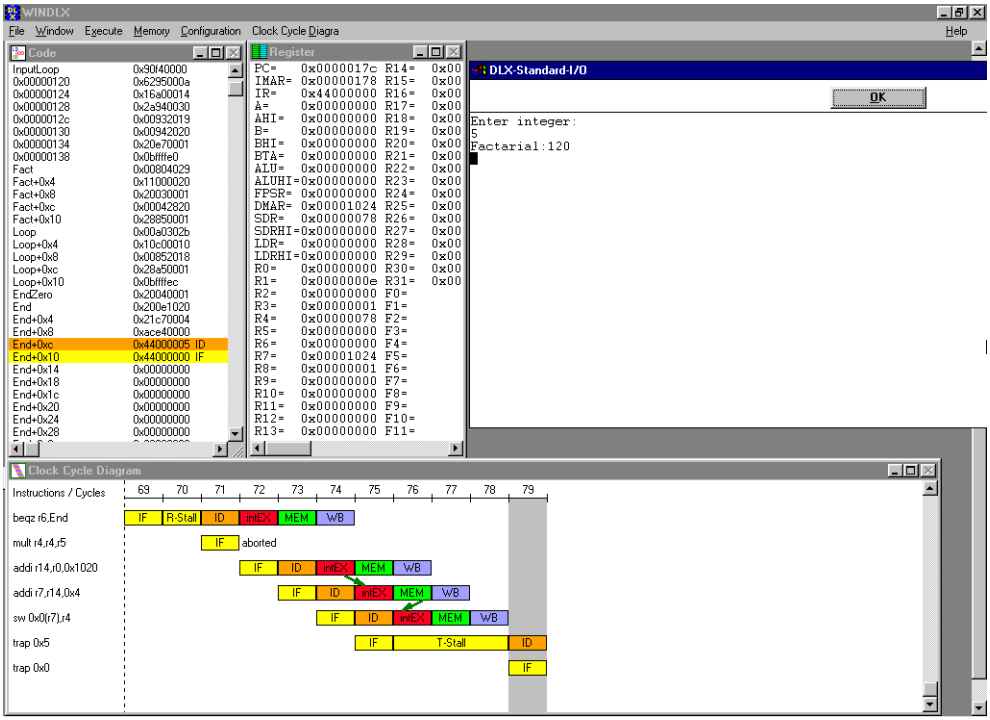

Figure 1.4: WinDLX GUI[10]

- **Trap #1: Open File**
- **Trap #2: Close File**
- **Trap #3: Read Block from File**
- **Trap #4: Write Block to File**
- **Trap #5: Formatted Output to Standard-output**

For all defined traps:

- They match UNIX/DOX system calls resp. C-library functions open(), close(), read(), write() and printf()
- The file descriptors 0, 1 and 2 are reserved for the stdin, stdout and stderr.
- The address of required arguments for system calls must be loaded in the register R14. The result is returned into the register R1.
- All arguments must be 32-bit long (except for DFPRs). Strings are referenced with their begin address.

1.10 DLX Assembly Code Example

The example below shows a simple computation of factorial using the DLX assembly code and the usage of WinDLX traps. The example reads one integer number from stdin using trap 3 (read() call), converts the input integer string to an actual integer value using the inputLoop subroutine, calculates the factorial and writes the result to stdout using trap 5 (printf() call). This example contains no error handling for bad inputs.

```
    .data
    .align 2
FormatStrOutput:    .asciiz     "Factorial:%d\n"

    .align 2
FormatStrInput:     .asciiz     "Enter integer\n"

    .align 2
ParStrOutput:                   ; arguments for the printf() call
    .word  FormatStrOutput      ; printing the result
    .space 4

    .align 2
ParStrInput:                    ; arguments for the printf() call
    .word  FormatStrInput       ; asking for the input

    .align 2
Buffer:                         ; buffer for the input
    .space 8

    .align 2
ParRead:                        ; arguments for the read() call
    .word 0
    .word Buffer
    .word 8

    .text
    .global main
main:
    lhi r14, ParStrInput >> 16    ; address of the ParStrInput
    addui r14, r14, ParStrInput & 0x0000ffff

    trap 5                      ; printf() call

    lhi r14, ParRead >> 16        ; address of the ParRead
    addui r14, r14, ParRead & 0x0000ffff

    trap 3                      ; read() call

    lhi r7, Buffer >> 16          ; address of the Buffer
    addui r7,r7, Buffer & 0x0000ffff

; code continues on the next page
```

```
    addi r19, r0, 10              ; number base = 10
    addi r4, r0, 0

inputLoop:                        ; reads byte after byte
    lbu r20, 0(r7)                ; from input and converts
    seqi r21, r20, 10             ; the string to an
    bnez r21, fact                ; integer value
    subi r20, r20, 48
    multu r4, r4, r19
    add r4, r4, r20
    addi r7, r7, 1
    j inputLoop

fact:                             ; factorial computation
    sne r8, r4, r0
    beqz r8, endZero
    add r5, r0, r4
    subi r5, r4, 1

loop:
    sgt r6, r5, r0
    beqz r6, end
    multu r4, r4, r5
    subi r5, r5, 1
    j loop

endZero:
    addi r4, r0, 1                ; 0! = 1

end:
    lhi r14, ParStrOutput >> 16   ; address of the ParStrOutput
    addui r14, r14, ParStrOutput & 0x0000ffff

    sw 4(r14), r4                 ; store the result

    trap 5                        ; printf() call

    trap 0                        ; terminate program
```

Chapter 2

Pipelining Technique

Pipelining is a technique where multiple instructions are overlapped in execution. A Pipeline can be thought of as an assembly line, for example in an automobile assembly line, there are many steps contributing to the construction of a car, each step operating in parallel on a different car. The same process is used in a computer pipeline where every step in the pipeline completes part of an instruction. Each of these steps is called a *pipe stage*. The stages are connected one to next to form a pipe - an instruction enters at one end, progress through the stages, and exit at the other end, just as cars would in an assembly line.

In an automobile assembly line *throughput* is defined as the number of cars produced per hour and is determined by how often a completed car exits the assembly line. Likewise, the throughput of an instruction pipeline is determined by how often an instruction exits the pipeline. Because pipe stages are connected together, they all must be ready to proceed at the same time, just as it is required in an automobile assembly line. The time required between moving an instruction one step down the pipeline is a machine cycle. Because all stages proceed at the same time, the length of a machine cycle is determined by the time required for the slowest pipe stage, just as in an automobile assembly line. In a computer, this machine cycle is usually one clock cycle. If stages are perfectly balanced, then the time per instruction on the pipelined machine in ideal conditions is equal to[1]:

$$\frac{Time\ per\ instruction\ on\ unpipelined\ machine}{Numbers\ of\ pipe\ stages}$$

Under those conditions, the speedup from pipelining equals the number of pipe stages, just as an automobile assembly line with n stages can ideally produce cars n times as fast. However, pipe stages usually cannot be perfectly balanced and furthermore pipelining does involve some overhead. Thus, the time per instruction on the pipelined machine will not have its minimum possible value, yet it can be close[1].

2.1 DLX Pipeline

The figure 2.1 shows how the pipeline works in the DLX architecture. It starts with the **Instruction fetch (IF)** stage which loads an instruction from memory and begins its execution. The following **Instruction Decode (ID)** stage decodes the loaded instruction and its operands. After decoding the instruction, the execution is performed in the **Execution (EX)** stage. The next **Memory access (MEM)** stage performs access to memory if the instruction requires it and the last stage called **Write Back (WB)** is used for storing the result of that instruction to a register if the instruction needs it. Theoretically, if an instruction is started every clock cycle, the performance will be five times a machine that is not pipelined[1].

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Figure 2.1: DLX Pipeline[1]

The figure 2.2 shows a simplified hardware scheme that performs the described DLX pipeline. The path on that scheme flows from left to right with all stages being active at the same time. A register is added between each stage to convey value and control information from one stage to another. The DLX pipeline can also be thought of as a series of datapaths shifted in time as shown on the figure 2.3. To summarize the DLX pipeline, the DLX pipe stages must perform the following tasks[1]:

Instruction fetch (IF)

- Fetch instruction from memory.
- Increment the PC by 4 to address the next sequential instruction.

Instruction decode (ID)

- Decode the instruction and its operands.
- Fetch values of registers.
- Test to zero for branching instructions.

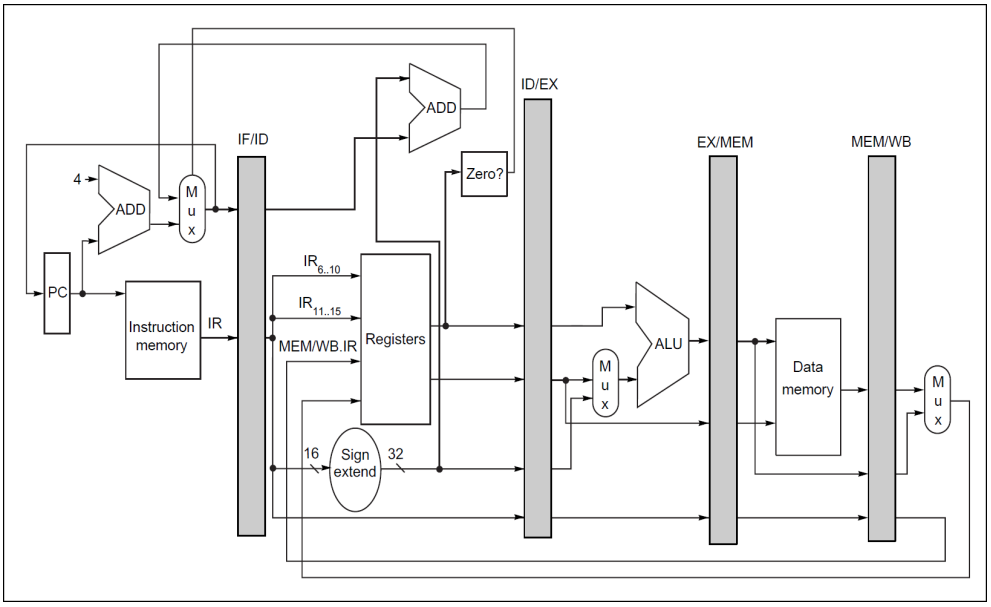

Figure 2.2: DLX Pipeline Hardware Scheme[1]

- Calculate the branch address if the instruction is a branch.
- Replace the program counter if the instruction branches.
- Extend the sign of the lower 16 bits of immediate value and pass it for use in the next stage.

Execution (EX)

- Perform ALU operation.
- The ALU operation is specified by the Opcode value or by the Func value in case of the R-type format instruction.

Memory access (MEM)

- Access memory if needed.

Write-back (WB)

- Write the result to the designated register.
- The result might be a result of an ALU operation or a load from memory.

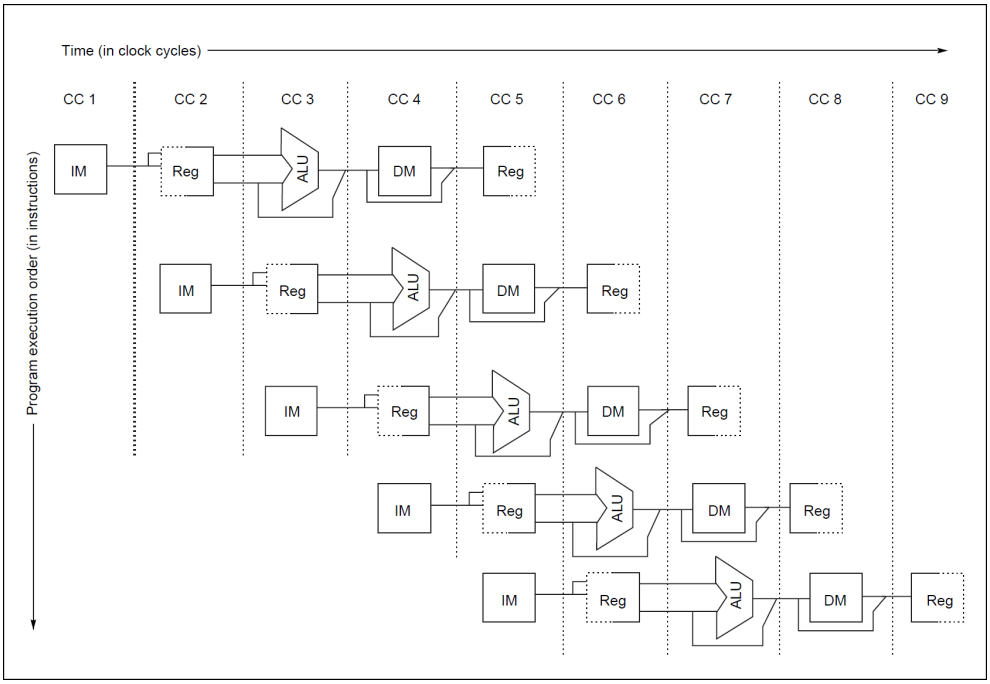

Figure 2.3: DLX Pipeline Datapath[1]

2.2 Pipeline Hazards

There are situations, called hazards that prevent the next instruction in the instruction stream from execution during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three types of hazards[1]:

1. **Structural hazards** arise from resources conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.

2. **Data hazards** arise when an instruction depends on the result of previous instructions in a way that is exposed by the overlapping of instructions in the pipeline.

3. **Control hazards** arise from pipelining of branches and other instructions that change the PC.

Hazards in pipelines can make it necessary to stall the pipeline at the instruction which causes the hazard. When an instruction is stalled, all instructions issued later than the stalled instruction are also stalled. Instructions issued earlier than the stalled instruction must continue since otherwise, the hazard

will never clear. As a result, no new instructions are fetched during the stall which reduces the overall performance of the pipeline[1].

2.2.1 Structural Hazards

If some combination of instructions cannot be accommodated because of resource conflict, the machine is said to have a structural hazard. The most common structural hazard arises when some functional unit is not fully pipelined. Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle. Another common way that structural hazards appear is when resources have not been duplicated enough to allow all combinations of instructions in the pipeline to execute[1].

For example, a machine might have a shared single-memory pipeline for data and instructions. As a result, when an instruction contains a data-memory reference, it will conflict with loading the next instruction from memory. To resolve this, the pipeline must be stalled for one clock cycle when the data memory access occurs as the figure 2.4 shows[1].

	Clock cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

Figure 2.4: A pipeline stalled for a structural hazard - a load with one memory port[1]

2.2.2 Data Hazards

A major effect of pipelining is that it changes the relative timing of instructions by overlapping their execution which can introduce data hazards. Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order by sequential execution. Consider this example:

ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11

All the instructions after the ADD instruction use the result of that instruction. As shown on the figure 2.5, the ADD instruction writes its result to the register R1 in its WB stage, but the SUB instruction needs the result value during its ID stage. This problem is called a data hazard and without taking some precautions to prevent that, the SUB instruction would read the wrong value and try to use it. The AND and OR instructions are also affected as the write of the result to the register R1 does not complete until the end of clock cycle five. The first instruction that would work correctly is the XOR instruction. The next section discusses a technique to reduce stalls for data hazards[1].

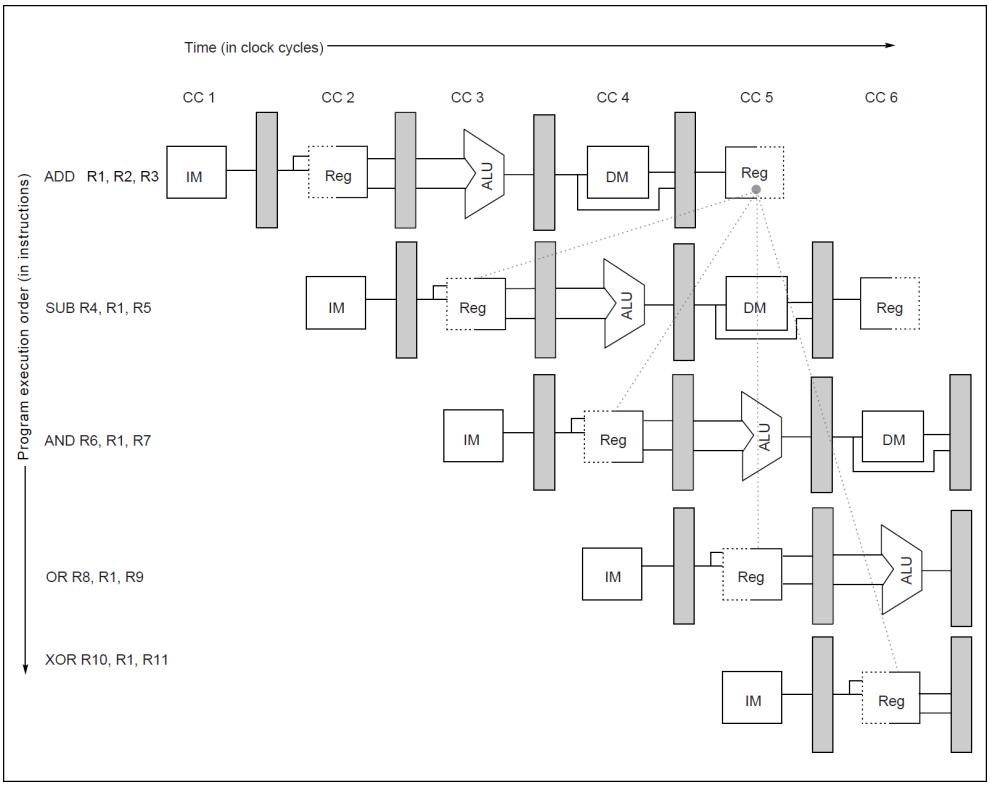

Figure 2.5: DLX Pipeline with data hazards[1]

2.2.3 Forwarding

The problem with data hazards can be solved using a simple hardware technique called *forwarding* (also called bypassing or short-circuiting). Considering the previous example, the key insight in forwarding is that the result is not

really needed by the SUB instruction until after the ADD instruction actually produces it. If the result can be moved from where the ADD instruction produces it to where the SUB instruction needs it, then the need for stall can be avoided. Generally, forwarding can be generalized to include passing a result directly to any functional unit that requires it. A result is forwarded from the output of one unit to the input of another[1].

The figure 2.6 shows the previous example with the bypasses in place to solve data hazards using the forwarding technique. The SUB instruction and the AND instruction get its input from registers after the ALU and the MEM stage. The OR instruction receives its result by forwarding inside the WB stage, which is easily accomplished by reading registers in the second half of the stage and writing them in the first half as the dashed lines indicate. With these bypasses, the previous example can now run without any need for stalls[1].

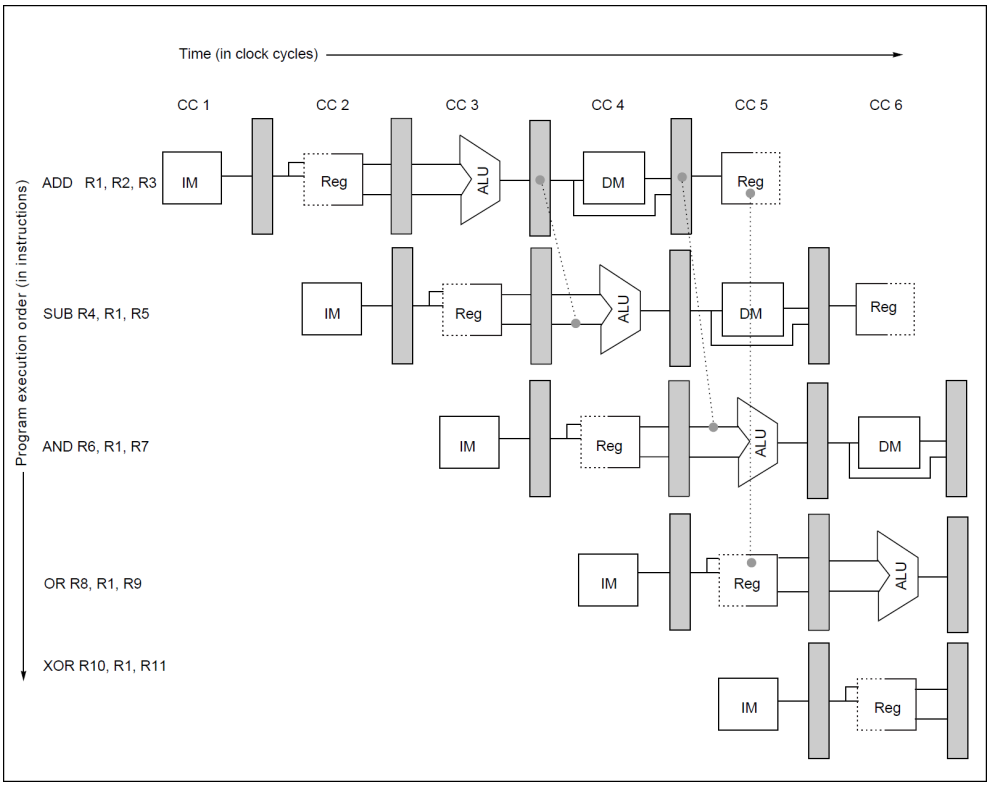

Figure 2.6: DLX Pipeline with solved data hazards using forwarding[1]

Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instruction. Consider two instructions: X and Y, with X occurring before Y. The possible data hazards are[1]:

- **Read after write (RAW)** - Y tries to read a source before X writes it, so Y gets an old value. This is the most common hazard.
- **Write after write (WAW)** - Y tries to write an operand before it is written by X. The writes end up being performed in the wrong order, leaving the value written by X rather than the value by Y in the destination.
- **Write after read (WAR)** - Y tries to write a destination before it is read by X, so X incorrectly gets a new value.

There are also some cases where stalls are inevitable and cannot be handled by forwarding. Consider the following sequence of instructions:[1]:

LW R1, 0(R2)
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9

The LW instruction does not have the loaded data until at the end of the MEM stage, while the SUB instruction needs to have the data from the previous instruction by the beginning of its EX stage. The data hazard from using the result of a load instruction cannot be eliminated with simple hardware because such a bypassing path would have to operate backward in time. Instead, new hardware, called a pipeline interlock, needs to be added to preserve the correct execution order. In general, a pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared[1].

2.2.4 Control Hazards

Control hazards are caused by instructions that change the program counter such as branches or jumps. The decision whether the branch is taken or not taken is usually done later in the pipeline, so for a certain number of cycles it is unknown which instruction should be loaded next into the pipeline and thus the pipeline must be stalled.

The DLX pipeline makes the decision whether the branch is taken or not taken in the ID stage and uses a *never-taken* scheme to minimize stalls. This scheme treats every branch as not taken and allows the pipeline to continue fetching instructions as if the branch was not taken and is careful not change the machine state until the branch outcome is definitely known. The benefit of using the never-taken scheme comes when the branch is not taken as the pipeline can continue without any stalls. However, if the branch is taken, the loaded instruction is aborted as seen on the figure 2.7 and the pipeline continues at the target address, which basically means that the pipeline was stalled for one cycle. On the other hand, the reverse *always-taken* scheme

would not work in the DLX architecture as the decision whether the branch is taken or not taken and the computation of the target address happens in the same stage[1].

Branch instruction	IF	ID	EX	MEM	WB					
Branch instruction + 1		IF								
Branch successor			IF	ID	EX	MEM	WB			
Branch successor + 1				IF	ID	EX	MEM	WB		
Branch successor + 2					IF	ID	EX	MEM	WB	
Branch successor + 3						IF	ID	EX	MEM	WB
Branch successor + 4							IF	ID	EX	MEM
Branch successor + 5								IF	ID	EX

Figure 2.7: DLX Pipeline using a never-taken scheme when a branch is taken[1]

2.3 Floating-point Operations

It would be impractical to require that all DLX floating-point operations (also integer multiply and integer divide) complete in one clock cycle. Doing so would mean accepting a slow clock or using enormous amounts of logic in floating-point functional units or both. Instead, the pipeline allows for a longer latencies for certain operations in the EX stage. The EX stage may be repeated as many times as needed to complete the operation as shown on the figure 2.8. The DLX architecture also uses four separate functional units to perform different operations[1]:

- **Integer unit** - The main integer unit that handles loads and stores, integer ALU operations and branches.
- **FP/integer multiply unit** - Floating-point and integer multiply unit.
- **FP Adder unit** - Floating-point adder unit that handles floating-point adding and subtracting.
- **FP/integer divider unit** - Floating-point and integer divider unit.

Each of these functional units can be described using two parameters[1]:

- **Number of stages** - Specifies how many stages are in a functional unit, in other words, how many instructions can concurrently use this unit in a pipeline.
- **Delay** - Specifies how many clock cycles are needed to finish the operation in this unit.

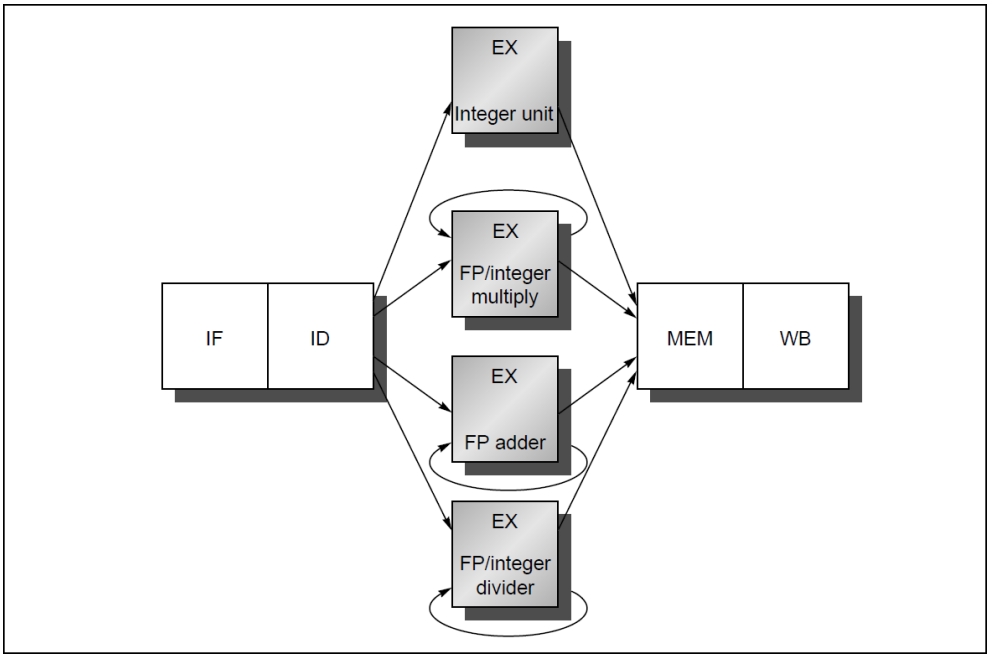

Figure 2.8: DLX Pipeline Functional Units[1]

The figure 2.10 shows the DLX pipeline with one stage for the integer unit, seven stages for the FP/integer multiply unit, four stages for the FP adder unit and twenty-four stages for the FP/integer divider unit. Delays are the same as the number of stages for all units. The use of different functional units introduces hazards [1]. For example, the following code causes a hazard when using the multiply unit with two stages and with the delay of four cycles:

MULTD F0, F2, F4
MULTD F6, F8, F10
MULTD F12, F14, F16

As the figure 2.9 shows, the first two MULTD instructions work as expected but the third one needs to be stalled until the first MULTD instruction finishes its EX stage. This is an example of a structural hazard where the hardware cannot accommodate this combination of instructions because the multiply unit has only two stages.

MULTD	IF	ID	M1	M2	M3	M4	MEM	WB			
MULTD		IF	ID	M1	M2	M3	M4	MEM	WB		
MULTD			IF	ID	stall	stall	M1	M2	M3	M4	MEM

Figure 2.9: DLX Pipeline stalled for a structural hazard in the EX stage

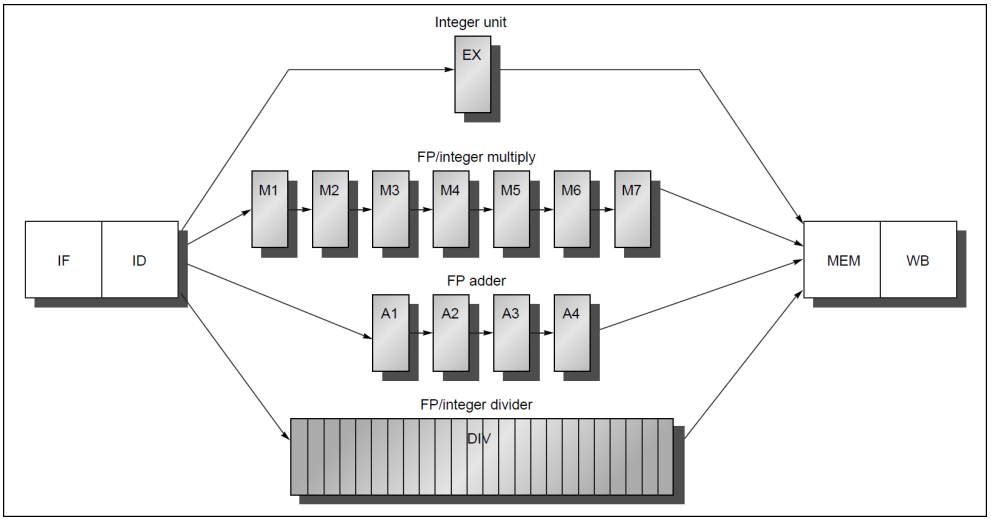

Figure 2.10: DLX Pipeline with different number of stages for different functional units[1]

The next example shows that a structural hazard can also occur outside the EX stage. Considering the following code when using the multiply unit with two stages and with the delay of two cycles and the integer unit with one stage and the delay of one cycle:

MULTD F0, F2, F4
ADD R1, R2, R3

As seen on the figure 2.11, the ADD instruction must be stalled for one cycle because it arrives at the MEM stage at the same time as the previous instruction MULTD. This hazard arose from having units with varying running times.

MULTD	IF	ID	M1	M2	MEM	WB	
ADD		IF	ID	EX	stall	MEM	WB

Figure 2.11: DLX Pipeline stalled for a structural hazard outside of the EX stage[1]

Using units with varying running time also brings some new problems concerning data hazards. Since instructions no longer reach WB stage in order, it can cause WAW hazards[1]. Consider the following code when using the multiplication unit with four stages and with delay of four cycles and the FP adder unit with two stages and the delay of two cycles:

MULTD F0, F2, F4
ADDD F0, F6, F8

The ADDD instruction wants to write the result to the register F0 before the MULTD instruction. The ADDD instruction must be stalled in the ID stage until the MULTD instruction finishes its EX stage because otherwise, the F0 register would end up with a value from the MULTD instruction which would not be correct as the register F0 should contain a result from the ADDD instruction as shown on the figure 2.12

MULTD	IF	ID	M1	M2	M3	M4	MEM	WB		
ADDD		IF	ID	stall	stall	stall	A1	A2	MEM	WB

Figure 2.12: DLX Pipeline with a WAW hazard[1]

2.4 Pipeline Configuration in WinDLX

WinDLX allows setting up the number of stages and delays for all functional units except for the integer unit which always has one stage and the delay of one cycle. Only the terminology is a bit different. WinDLX calls the number of stages as the *Number of units in each class* and the functional units are named[15]:

- **Addition unit (FP adder unit)**
- **Multiplication unit (FP/integer multiply unit)**
- **Division unit (FP/integer divider unit)**

These functional units can be set with the limits defined in the following 2.1 table:

Table 2.1: WinDLX Pipeline Configuration

Configuration	Value
Number of units in each class	$1 \leq M \leq 8$
Delay (Clock cycles)	$1 \leq M \leq 50$

Chapter 3

LLVM System

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. LLVM began as a research project at the University of Illinois, with the goal of providing a modern compiler capable of compilation of arbitrary programming language with a focus on modularity. Since then LLVM has grown to be an umbrella project consisting of a number of a compiler related subprojects[16].

The LLVM compiler design can be divided into three fundamental parts as shown on the figure 3.1 and described below. Each part consists of several phases that little by little transform the input source code to the final machine code[17]:

- **Fronted** that is responsible for converting the input source code to the LLVM intermediate code (LLVM IR) and usually contains those phases[18]:
  1. Lexical analysis
  2. Syntax analysis (Parser)
  3. Semantic analysis
  4. LLVM intermediate code generation
- **Optimizer** which runs machine independent optimizations on the LLVM intermediate code.
- **Backed** that transforms the LLVM intermediate code to the final machine code and performs the following tasks[18]:
  1. Instruction selection
  2. Instruction scheduling
  3. Register allocation
  4. Late machine dependent optimizations

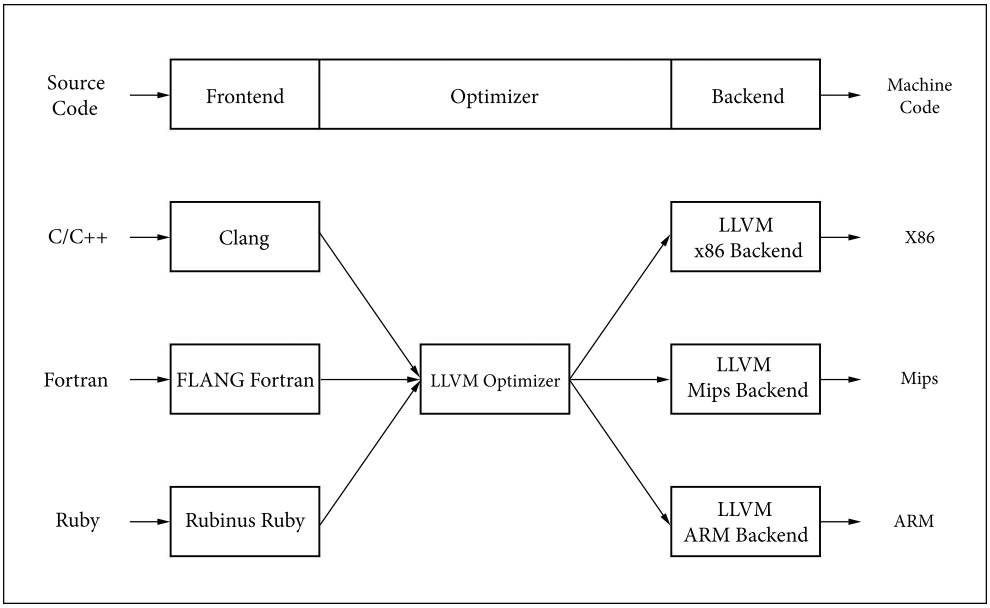

Figure 3.1: LLVM Compiler Design[17]

With this design, adding support to a new source language only requires implementing a new LLVM frontend, but the LLVM optimizer with all existing optimizations and all LLVM backends can be reused. Also, only a backend part needs to be created to support a new machine target. If those parts were not separated it would take N*M compilers to support N source languages and M machine targets as opposed to N frontends plus M backends when using the LLVM compiler structure[17].

LLVM itself comes in three pieces. The first is the *LLVM suite*. This contains all of the tools, libraries, and header files needed to use the LLVM system. The second piece is *Clang*, an LLVM fronted for C, C++, Objective-C and Objective-C++ and the third piece is called *LLVM Test suite* and contains programs and tools to test LLVM's functionality and performance[19].

3.1 LLVM IR

The LLVM IR is an intermediate code representation that is used in the LLVM system. It is a linear code that somewhat reminds an assembly code where some things look like a high-level language such as functions or strong typing when other looks more like a low-level language such as branching or basic-blocks[20]. It has an infinite number of registers, uses an SSA (Single Static Assignment) form and provides type safety, low-level operations, flexibility and capability of representing all high languages[21]. The LLVM IR is generated

by a frontend and is used throughout the entire optimizer part of the LLVM. After the optimizations are done, the LLVM IR code is sent to a backend where it is transformed into the target machine code.

The LLVM intermediate code can be represented in three different ways: as in-memory compiler IR, as on-disk bitcode representation or as a human readable assembly language representation[21].

3.1.1 SSA

Single static assignment is a form of a code representation where all of the following conditions are true[22].

SSA conditions:

- Each variable is defined exactly once.
- Each variable's definition dominates all of its uses.
- Each variable is defined before it is used.

Converting a code into an SSA form is primarily a simple matter of replacing the target of each assignment with a new variable, and replacing each use of a variable with the version of the variable reaching that point. For example, consider the following code[23]:

$$\begin{array}{l} \mathbf{X = 1} \\ \mathbf{X = 5} \\ \mathbf{Z = X} \end{array} \quad \rightarrow \quad \begin{array}{l} \mathbf{X_1 = 1} \\ \mathbf{X_2 = 5} \\ \mathbf{Z_1 = X_2} \end{array}$$

The only thing that changed is that each variable got a version based on the order of its usage. This is fairly simple as the example was just a sequential code. But usually, variables can have multiple values depending on the previous control flow. As the example below shows, the variable Y can either have a value of X1 or X2 depending on the condition of the if statement. In that case, some sort of control is needed to determine where the control flow came from and what the value of Y will be[23].

$$\begin{array}{l} \textbf{if(...)} \\ \quad \mathbf{X = 2} \\ \textbf{else} \\ \quad \mathbf{X = 3} \\ \mathbf{Y = X} \end{array} \quad \rightarrow \quad \begin{array}{l} \textbf{if(...)} \\ \quad \mathbf{X_1 = 2} \\ \textbf{else} \\ \quad \mathbf{X_2 = 3} \\ Y_1 = \textbf{?????} \end{array}$$

To resolve this, a special statement is inserted, called a ϕ (PHI) function. A ϕ function generates a new value by selecting the proper value based on where

the control flow arrived from. The following code shows the transformation to the code in an SSA form using the ϕ function. A ϕ function translates into no instructions in the machine code because the control flow is known during the execution[23].

if(...)		**if(...)**
X = 2		$\mathbf{X_1 = 2}$
else	$\rightarrow$	**else**
X = 3		$\mathbf{X_2 = 3}$
Y = X		$X_3 = \mathbf{phi}(X_1, X_2)$
		$Y_1 = X_3$

The biggest advantage of an SSA form comes from simplifying certain optimizations. The ones which are most enhanced by the SSA form, mainly from its property that each variable is assigned only once are, for example, constant propagation, dead code elimination, dead instruction elimination or strength reduction[23].

3.1.2 Basic Block

A basic block is a linear sequence of code which has exactly one entry point (the first instruction executed) and exactly one exit point (the last instruction executed - called the terminator instruction). This ensures that if a basic block is executed, it will be executed from the first instruction to the last instruction without jumping or branching anywhere in the middle of the basic block. Basic blocks may have many predecessors and many successors which represents the control flow of the code. The control flow can be captured in a directed **control flow graph (CFG)** where nodes represent basic blocks and edges represent control flow paths. The control flow graph specifies all possible execution paths[24]. Example of such graph can be seen on the figure 3.2 which shows a CFG constructed for a simple C programming language function foo that counts from a to b as shown below.

```
int foo(int a, int b)
{
    while(a<=b)
    {
        a++;
    }
    return a;
}
```

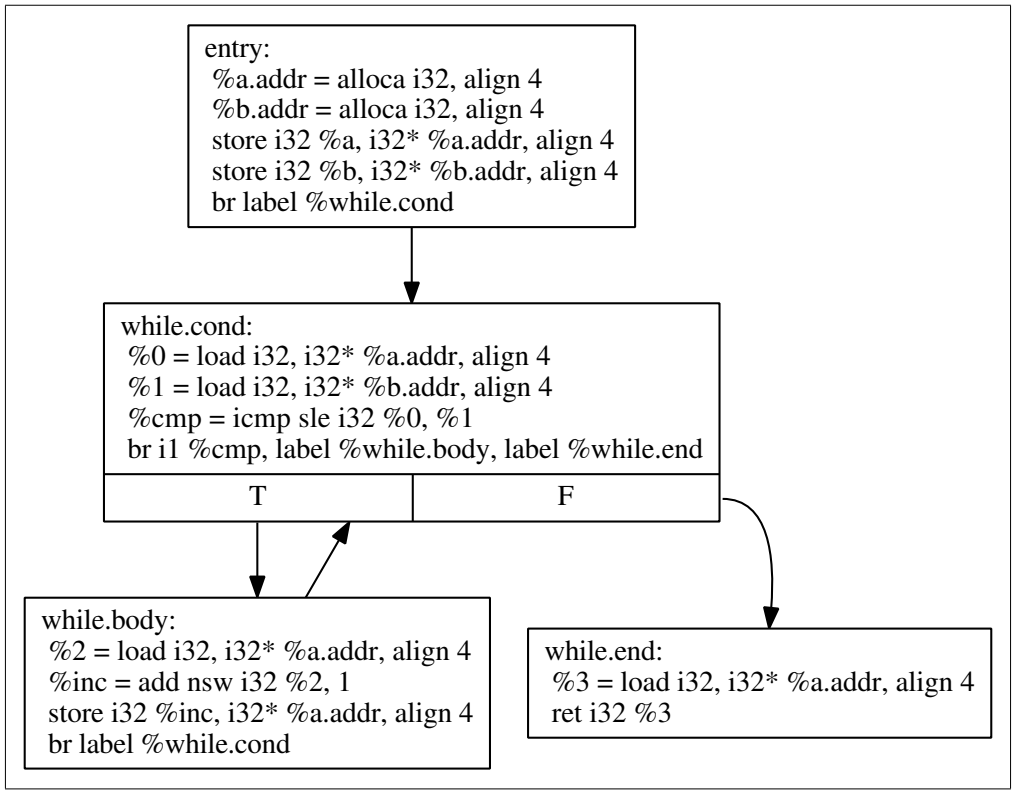

Figure 3.2: Control Flow Graph constructed for the foo function

3.2 LLVM IR Language

This section describes the LLVM IR language constructs such as identifiers, modules, functions, types, and instructions. Due to limited space, the constructs are simplified only to necessary parts. The full description of the LLVM IR language can be obtained in the *LLVM Language Reference Manual* document[21].

3.2.1 Identifiers

LLVM identifiers come in two basic types: global and local. Global identifiers begin with a prefix @ and local identifiers begin with a prefix %. Additionally, there are three different formats for identifiers, for different purposes[21]:

- **Named values** are represented as strings with their prefixes (%foo, @bar).
- **Unnamed values** are represented as unsigned numeric values with their prefixes (%12, @6).

- **Constants** - LLVM has several different types of constants: boolean constants (true, false), integer constants, floating-point constants and a null pointer constant (null).

3.2.2 Modules

LLVM programs are composed of modules, each of which is a translating unit of the input program. Each model consists of functions, global variables and symbol table entries[21].

3.2.3 Functions

A function contains a list of basic blocks, forming a CFG. The first basic block in a function is specific in two ways: it is immediately executed on entrance to the function, and it is not allowed to have any predecessor basic blocks.

LLVM function definition consists of the *define* keyword and parameters such as linkage type, visibility style, calling convention type and argument list. The syntax with the example is shown below[21].

```
define [linkage] [visibility] [cconv] <ResultType> @<FunctionName>
                    ([argument list]) {... }

; Example:
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
    %tmp = mul i32 %x, %y
    %tmp2 = add i32 %tmp, %z
    ret i32 %tmp2
}
```

3.2.4 Type System

The LLVM IR is a very strongly-typed language and pretty much everything has some sort of type. The types consist of void type, numeric types, function types, pointer types and aggregate types such as array types and structure types[21].

Void Type

The void type does not represent any value and has no size[21].

Integer Type

The integer type is a simple type that specifies an arbitrary bit width for the integer type desired. Any bit from 1 to $2^{23} - 1$ can be specified. The LLVM does not make a distinction between signed and unsigned types but uses different instructions for signed and unsigned values when needed[21].

Syntax:

```
iN  ; The N specifies the bit width for the integer type

; Examples:
i1  ; A single bit integer
i32 ; A 32-bit integer
```

Floating-point Type

LLVM uses several types for floating-point numbers[21]:

- **half** - 16-bit floating-point value
- **float** - 32-bit floating point value
- **double** - 64-bit floating-point value
- **fp128** - 128-bit floating-point value

Pointer Type

The pointer type is used to specify memory locations, pointers are commonly used to reference objects in memory[21].

Syntax:

```
<type> *

; Examples:
i32 *           ; A pointer to a 32-bit integer
[4 x i23]*      ; A pointer to an array of four i32 values
i32 (i32*) *    ; A pointer to a function that takes an i32*
                ; pointer and returns an i32 value
```

Array Type

The array type is a very simple aggregate type that arranges elements of the same type sequentially in memory. The array type requires a size (number of elements) and an underlying data type. The number of elements is a constant integer value and the element type may be any type with a size[21].

Syntax:

```
[<#elements> x <type>]

; Examples:
[40 x i32]          ; An array of 40 32-bit integers
[12 x [10 x float]] ; A two-dimensional 12x10 array of
                    ; 32-bit floating-point values
```

Structure Type

The structure type is used to represent a collection of data members together in memory. The elements of the structure may be any type with a size. Structures in memory are accessed using *load* a *store* instructions by getting a pointer to a field with the *getelementptr* instruction[21].

Syntax:

```
%t = type { <type list> }

; Examples:
{i32, i32, i32}  ; A structure with three i32 values
{float, i32*}    ; A structure with a float and a pointer to an
                 ; i32 value
{i8, i32}        ; A structure with an i8 value and an i32 value
```

Function Type

The function type can be thought of as a function signature. It consists of the return type and a list of formal parameter types[21].

Syntax:

```
<return type> ( <parameter list> )

; Examples:
i32 (i32, i32)    ; A function that takes two i32 values and
                  ; returns a single i32 value
i32 (i8*)         ; A function that takes an i8* pointer and
                  ; returns an i32 value
{i32, i32} (i32) ; A function that takes an i32 value and returns
                  ; a structure with two i32 values
```

3.3 LLVM IR Instructions

This section describes the LLVM IR instruction set which consists of several different classes of instructions: terminator instructions, binary instructions, bitwise binary instructions, memory instructions, and some other instructions[21].

3.3.1 Terminator Instructions

As mentioned previously, every basic block in a program ends with a terminator instruction that indicates which block should be executed after the current block is finished. These terminator instruction yield a void value as they produce control flow, not values[21].

ret instruction

The *ret* instruction is used to return the control flow from a function back to the caller. It may or may not return a value. If it does return a value, it accepts a single argument, the return value[21].

Syntax:

```
ret <type> <val>     ; Return a value from a non-void function
ret void             ; Return a void

; Examples:
ret i32 5       ; Return an i32 value 5
ret void        ; Return void
ret {i32, i8} {i32 4, i8 2}  ; Return a struct with an
                             ; i32 value 4 and an i8 value 2
```

br instruction

The *br* instruction is used to cause the control flow to transfer to a different basic block in the current function. There are two forms of this instruction, corresponding to a conditional branch and an unconditional branch. The conditional form evaluates an i1 argument and if the value is true, control flows to the iftrue label argument and if the value is false, control flows to the iffalse label argument. The unconditional form simply transfers the control flow to the destination label.

Syntax:

```
br i1 <cond>, label <iftrue>, label <iffalse> ; Conditional branch
br label <destination>    ; Unconditional branch

; Example:
test:
    %cond = icmp eq i32 %a %b

    ; Branch on label IfEqual if the condition is true, otherwise
    ; branch on label IfUnEqual
    br i1 @cond, label %IfEqual, label %IfUnEqual

IfEqual:
    ret i1 1

IfUnEqual
    ret i1 0
```

switch instruction

The *switch* instruction is used to transfer the control flow to one of several different places. It is a generalization of the *br* instruction, allowing a branch

to one of many possible basic blocks. It specifies a table of values and basic blocks. When the switch instruction is executed, this table is searched for the given value. If the value is found, the control flow is transferred to the corresponding basic block. Otherwise, the control flows to the default basic block[21].

Syntax:

```
switch <type> <val>, label <default>
                              [<type> <val>, label <dest> ...]
; Example:
; The control flow braches to either onzero, or to onone, or to
; otherwise based on the value of the val variable
switch i32 %val, label %otherwise [ i32 0, label %onzero
                                    i32 1, label %onone ]
```

3.3.2 Binary instructions

Binary instructions are used to do the most computing in a program. They require two arguments of the same type and produce a single value as a result of the operation. The operation opcode can be: add, fadd, sub, fsub, mul, fmul, udiv, sdiv, fdiv, urem (a reminder of an unsigned division), srev (a reminder of a signed division), frev (a reminder of a floating-point division). The f prefix signifies a floating-point operation[21].

Syntax:

```
<result> = <opcode> <type> <val1> <val2>

; Examples:
%res = mul i32 %x, 2
%res = fadd float %x, 4.0
```

3.3.3 Bitwise binary instructions

Bitwise binary instructions are used to do various bit-twiddling in a program and they are usually very efficient. They require two arguments and return a single value as a result of the bitwise operation. The operation opcode can be: shl (shift left), lshr (logical shift right), ashr (arithmetic shift right), and, or, xor[21].

Syntax:

```
<result> = <opcode> <type> <val1> <val2>

; Examples:
%res = shl i32 4, 2
%res = xor i32 4, %x
```

3.3.4 Memory instructions

This section describes how to read, write and allocate memory in LLVM.

alloca instruction

The *alloca* instruction allocates memory on the stack frame of the currently executing function. The instruction allocates $sizeof(type) * \#elements$ bytes of memory and returns a pointer of an appropriate type[21].

Syntax:

```
<result> = alloca <type> [, <type> <#elements>]

; Examples:
%ptr = alloca i32             ; Allocate one i32 value
%ptr = alloca i32, i32 4      ; Allocate four i32 values
```

load instruction

The *load* instruction is used to read from memory. The argument pointer to the *load* instruction specifies the memory address from which to load[21].

Syntax:

```
<result> = load <type>, <type>* <pointer>

; Example:
%val = load i32, i32* %ptr
```

store instruction

The *store* instruction is used to write to memory. There are two arguments to the *store* instruction: the value to store and the address of memory at which to store. The *store* instruction returns a void value[21].

Syntax:

```
store <type> <val>, <type>* <pointer>

; Example:
store i32 3, i32* %ptr
```

getelementptr instruction

The *getelementptr* instruction is used to get the address of an element of an aggregate object such as an array or a structure. It performs address calculation only and does not access memory[21]. The first argument is a type which is used as the basis for the calculations, the second argument is a pointer which

is the base address. The remaining arguments are indices that indicate which of the elements of an aggregate object are indexed. In the example below, the first indice indexes through the pointer and the second indice indexes the fields of the t_str structure[25].

Syntax:

```
<result> = getelementptr <type>, <type>* <pointer>{,<type> <idx>}*

; Example:
%struct.t_str = type { i32, i32 }

; Compute the address of the second element of the t_str struct
%x = getelementptr %struct.t_str, %struct.t_str* %0, i32 0, i32 1

; Store the i32 value 8 to the second element of the t_str struct
store i32 8, i32* %x
```

3.3.5 Conversion instructions

The following instructions perform various conversion between types. All of these instructions take a single value argument and a type to which convert the value argument[21].

Syntax:

```
<result> = conv_inst <type> <val> to <type>
```

Where the conv_inst may be one of the following:

- **trunc** - Truncates integer value to a smaller type.
- **zext** - Zero extends integer value to a larger type.
- **sext** - Sign extends integer value to a larger type.
- **fptrunc** - Truncates a floating-point value to a smaller type.
- **fpext** - Extends a floating-point value to a larger type.
- **fptoui** - Converts a floating-point value to its unsigned integer equivalent of the desired type.
- **fptosi** - Converts a floating-point value to its signed integer equivalent of the desired type.

Examples:

```
%res = trunc i32 123 to i1          ; Return i1:true
%res = zext i1 true to i32          ; Return i32:1
%res = fptosi double -123.0 to i32  ; Return i32:-123
```

3.3.6 Other instructions

This section describes some more instructions in the LLVM instruction set that defy better classification[21].

icmp and fcmp instruction

LLVM has two compare instructions, one for integer values (*icmp*) and other for floating-point values (*fcmp*). Both of these instructions return a boolean value based on a comparison of its arguments which must be of the same type. These instructions also take a condition code argument to indicate which comparison condition should be used[21].

Syntax:

```
<result> = icmp <cond> <type> <val1>, <val2>
<result> = fcmp <cond> <type> <val1>, <val2>

; Examples:
<result> = icmp eq i32 4, 5          ; i32:4 == i32:5
<result> = icmp ne float* %x, %y     ; float*:%x != float*:%y
<result> = icmp ult i16  4, 5        ; i16:4 < i16:5
<result> = fcmp one float 4.0, 5.0   ; float:4.0 != float:5.0
<result> = fcmp ueq double 1.0, 2.0  ; double:1.0 == double:2.0
```

select instruction

The *select* instruction is used to choose one value based on a condition without IR-level branching. This instruction takes an i1 value indicating the condition and two value arguments of the same type. If the condition evaluates to one, the instruction returns the first value argument. Otherwise, it returns the second value argument.

Syntax:

```
<result> = select i1 <cond>, <type> <val1>, <type> <val2>

; Example:
%res = select i1 true, i8 17, i8 42   ; Return a i8 value 17
%res = select i1 %val, float %x, float %y
```

phi instruction

The *phi* instruction is used to implement the ϕ function. The first argument specifies types of incoming values. After this, the instruction takes a list of predecessors of the current basic block and their associated values. The *phi* instruction must be the first instruction in a basic block[21].

Syntax:

```
<result> = phi <type> [<val1>, <label1>], [<val2>, <label2>], ...

; Example:
; Return a true value if the control flow came from the basic
; block entry1 or return the variable x if the control flow
; came from the basic block entry2
%retval = phi i1 [ true, %entry1 ], [ %x, %entry2 ]
```

call instruction

The *call* instruction represents a simple function call. This instruction is used to cause the control flow to transfer to a specified function with its arguments bound to specific values. Upon *ret* instruction in the called function, control flow continues with the instruction after the function call[21].

Syntax:

```
<retval> = call <type> <function> (<function args>)

; Example:
%retval = call i32 @test(i32 %x)
```

3.4 LLVM Tools

LLVM has numerous tools to deal with various parts of the compilation process:

- **LLVM static compiler** - The *llc* command compiles LLVM source inputs (LLVM IR) into the assembly language for a specified target[26].
- **LLVM optimizer** - The *opt* command is the LLVM optimizer and analyzer. It takes the LLVM source files (LLVM IR) as input, runs the specified optimizations or analyses on it, and then outputs the optimized files or the analysis results[26].

3.5 Clang

Clang provides a language frontend for the LLVM system and tooling infrastructure for languages in the C language family (C, C++, Objective-C/C++, OpenCL, CUDA, and RenderScript). Both a GCC-compatible compiler driver (clang) and an MSVC-compatible compiler driver (clang-cl.exe) are provided[27]. The main features of clang are[28]:

- **Fast compile time and low memory use** - The major focus of clang is to be fast, light, scalable and to save memory.

- **Expressive diagnostics** - Clang aims to be extremely user-friendly by making error and warning messages as useful and human-readable as possible.
- **GCC compability** - GCC is currently the defacto standard open source compiler today, and it routinely compiles a huge volume of code. GCC supports a huge number of extensions and features (many of which are undocumented) and a lot of code and header files depend on these features in order to build. All the gcc extensions are put in clang because many users just want their code to compile and they don't care to argue about whether it is pedantically by the standard or not.

3.5.1 Clang Example

The following code shows the LLVM IR generated by clang for the foo function from the section 3.1.2 which counts from a to b. Basic blocks are represented as labels. Clang allocates variables on the stack and they are later transformed into SSA variables and phi functions in the Promote Memory to Register (mem2reg) pass during optimizations[29].

```
define dso_local i32 @_Z3fooii(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  br label %while.cond

while.cond:                                       ; preds = %while.body, %entry
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %cmp = icmp slt i32 %0, %1
  br i1 %cmp, label %while.body, label %while.end

while.body:                                       ; preds = %while.cond
  %2 = load i32, i32* %a.addr, align 4
  %inc = add nsw i32 %2, 1
  store i32 %inc, i32* %a.addr, align 4
  br label %while.cond

while.end:                                        ; preds = %while.cond
  %3 = load i32, i32* %a.addr, align 4
  ret i32 %3
}
```

Chapter 4

LLVM Backend for DLX

This chapter describes the process of creating an LLVM compiler backend that converts the LLVM Intermediate Representation (LLVM IR) to assembly code for the described DLX architecture.

4.1 Analysis of Existing Backends

Before creating the backend, these following backends were analyzed to get a sense of how LLVM backends work and what approach should be taken to create a new LLVM backend. Also, the LLVM documentation does not always describe LLVM backend constructs in full detail and existing backends provide a lot of useful information on the process of creating a new LLVM backend. These backends were chosen based on the fact that they are RISC architectures as is the DLX architecture and also that they are quite simple which allows the understanding of the structure of an LLVM backend. Alongside the three main backends below, ARM, x86 and Cpu0 backends were also studied for some information.

Lanai Backend

The Lanai architecture is the simplest RISC architecture for which an LLVM backend is implemented. Its instruction set is quite similar to the DLX instruction set although it contains some advanced instructions like select. It is an architecture that only uses integer registers and integer instructions[30]. This backend can serve as a good stepping stone to create a new RISC LLVM backend.

Mips Backend

The Mips architecture is a bit more complicated architecture that is to some extent a superset to the DLX architecture which means it has some features

that are needed in the DLX backend[31].

Sparc Backend

Sparc is a pretty typical RISC architecture that has almost everything that the DLX architecture uses and some more but not too much like the Mips architecture which is quite complicated to be a starting point for creating a new LLVM backend[32]. This means that the Sparc backend contains almost all the necessary parts for creating a new DLX backend. The Sparc backend source codes are quite well documented and the source codes comments contain very useful information about LLVM backend constructs. The Sparc backend is also used as an example in parts of the LLVM text *Writing an LLVM Backend*[33].

4.2 Target Description

The LLVM target description files (the .td files) provide a description of the target machine. These files are designed to capture the properties of the target architecture (such as instructions and registers it has), and do not incorporate any particular pieces of code generation algorithms. These files use a TableGen syntax and the tool to translate these files is also called TableGen[34].

4.2.1 TableGen

TableGen's purpose is to help a human develop and maintain records of domain-specific information. It is specifically designed to allow writing flexible descriptions and for common features of these records to be factored out. This reduces the amount of duplication in the description, reduces the chance of error, and makes it easier to structure domain-specific information. TableGen files are interpreted by the TableGen tool which generates .inc files and those files can be included in C++ source files the same way as C++ header files[35].

4.2.2 DLX Target Description

The DLX backend uses TableGen files to describe registers, instruction set and calling conventions of the architecture. DLX uses the following files to describe the architecture (every class and every TableGen file that belongs to the DLX target has a ZR prefix which is an alias used to create a new DLX backend):

- **ZR.td** - Describes information about the architecture such as the ProcessorModel or the InstPrinter. It serves as a base class for the TableGen tool when generating DLX target machine files.

- **ZRRegisterInfo.td** - Describes registers and their properties.
- **ZRInstrInfo.td** - Describes the whole instruction set, operands and other classes that are mainly used in the instruction selection stage.
- **ZRCallingConv.td** - Describes calling conventions.

These files are then processed by the TableGen tool which generates the *ZRGenInstrInfo.inc*, *ZRGenAsmWriter.inc*, *ZRGenCallingConv.inc*, *ZRGenRegisterInfo.inc*, *ZRGenDAGISel.inc* and *ZRGenSubtargetInfo.inc* files which are then included in several DLX backed files to help the translation to DLX assembly codes.

4.3 Target Machine

The LLVMTargetMachine class is designed to be a base class for all backends implemented in the LLVM system. To actually create a concrete LLVM backend this class must be specialized by a concrete backend class that implements various virtual methods. This class provides virtual methods that are used to access the backend specific implementations of backend descriptions via get*Info() methods. The DLX backend uses the following: getInstrInfo(), getFrameLowering(), getRegisterInfo(), getTargetLowering() and getSelectionDAGInfo()[33].

4.3.1 DataLayout

DataLayout specifies information about how the target lays out memory for structures, the alignment requirements for various data types, the size of pointers in the target, and whether the target is little-endian or big-endian[21]. The layout specification consists of a list of specifications separated by the minus sign character ('-'). The DLX backend uses the following list of specifications.

```
"E"         // Big endian
"-p:32:32" // 32-bit pointers, 32 bit aligned
"-i32:32"  // 32-bit integers, 32 bit aligned
"-f32:32"  // 32 bit floating-point numbers, 32 bit aligned
"-f64:64"  // 64-bit floating-point numbers, 64 bit aligned
"-a:0:32"  // 32 bit alignment of objects of an aggregate type
"-n32"     // 32 bit native integer width
"-S32"     // 32 bit natural stack alignment
```

4.4 Defining Registers

DLX registers are defined in the *ZRRegisterInfo.td* TableGen file. This file starts with the definition of base register classes for representing an integer

register and a floating-point register as shown below. Each register has a seven bit number for identification and the specified string n becomes the name of a register.

```
// Base class for integer registers
class RI<bits<7> num, string n, list<Register> subregs = [],
         list<string> altNames = []> : Register<n, altNames> {
  field bits<7> Num; let Num = num;
  let Namespace = "ZR"; let SubRegs = subregs;
}
class RF ... // Base class for floating-point registers
```

As shown below, every register in DLX is defined using one the base register class. There is a definition of all integer registers, floating-point registers (both single-precision and double-precision) and the FPSR register.

```
// Integer registers
foreach i = 0-31 in {
  def R#i : RI<i, "r"#i>, DwarfRegNum<[i]>; }

// Single-precision floating-point register F0
def F0 : RF<32, "F0">, DwarfRegNum<[32]>; def F1 ...

// Double-precision floating-point register D0 that has two
// subregisters F0 and F1
def D0 : RF<64, "F0", [F0, F1]>, DwarfRegNum<[64]>; def D1 ...

// Floating-point status register
def FPSR : RI<0, "FPSR">, DwarfRegNum<[80]>;
```

In the same TableGen file, the RegisterClass is used to define an object that represents a group of related registers and also defines the default allocation order. In the DLX backend, four RegisterClass objects are defined: GPRegs, FPRegs, DFPRegs, and StatusRegs (contains only the FPSR register) as can be seen below.

```
def GPRegs : RegisterClass<"ZR", [i32], 32,
  (add R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13,
    R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25,
      R26, R27, R28, R29, R30, R31)>;

def FPRegs : RegisterClass<"ZR", [f32], 32,
  (sequence "F%u", 0, 31)>;

def DFPRegs : RegisterClass<"ZR", [f64], 64,
  (add D0, D1, D2, D3, D4, D5, D6, D7, D8,
    D9, D10, D11, D12, D13, D14, D15)>;

def StatusRegs : RegisterClass<"ZR", [i32], 32, (add FPSR)> {
  let CopyCost = -1; // Don't allow copying of FPSR
  let isAllocatable = 0;  // Don't allow allocating of FPSR
}
```

4.5 Defining Instructions

The DLX instruction set is defined in two TableGen files: *ZRInstrFormat.td* and *ZRInstrInfo.td*. The *ZRInstrFormat.td* file defines the base class for all instructions and defines format base classes which follow formats of instructions defined in the 1.4 section. Also, the Pseudo base class is defined for pseudo instructions that are later used mainly in the instruction selection stage.

```
class ZRInst<dag outs, dag ins, string asmstr, list<dag> pattern,
        string opstr = ""> : Instruction {

  field bits<32> Inst;
  bits<6> Opcode = 0;
  let Inst{31-26} = Opcode;
  string BaseOpcode = opstr;
  let OutOperandList = outs; // A DAG with the MI def operand list
  let InOperandList = ins;   // A DAG with the MI use operand list
  // The .s format to print the instruction with
  let AsmString = asmstr;
  let Pattern = pattern;     // A DAG pattern for this instruction
  let Namespace = "ZR";
}
```

The ZRInst is the base class for all instructions used in the DLX target. It creates a 32-bit field to store the binary representation in which the first six bits are used for an instruction Opcode. Every instruction has a prepared structure to store its binary representation even though the DLX target sets all bits to zero as there is no need for an object file writer. The OutOperandList, InOperandlist and the pattern fields are used in the instruction selection stage. The format base classes and the pseudo base class are then defined using this ZRInst base class as shown below[33]:

```
// I-type format base class
class FI<bits<6> op, dag outs, dag ins, string asmstr, list<dag>
        pattern> : ZRInst<outs, ins, asmstr, pattern> {

  //Binary representation of instructions
  bits<5> rt; bits<5> rs; bits<16> imm16; let Inst{25-21}= rs;
  let Inst{20-16}= rt; let Inst{15-0}= imm16; let Opcode = op;
}

class FR : ZRInst { ... } // R-type format
class FJ : ZRInst { ... } // J-type format

// Base class for pseudo instructions
class Pseudo<dag outs, dag ins, string asmstr, list<dag> pattern>
    : ZRInst<outs, ins, asmstr, pattern> {
  let isCodeGenOnly = 1; let isPseudo = 1;
}
```

Before defining DLX instructions, operands of those instructions must be de-

fined first. Operands along with instructions are defined in the *ZRInstrInfo.td* file. As shown below, the DLX target defines several types of operands for different instructions. It defines call target operand, branches and jumps target operands, immediate operands (both signed and unsigned), an immediate operand for shifts, an immediate operand for LHI instruction and a condition code operand which is used for floating-point conditions. LO16 and HI16 operands are used to get the high and low 16 bits of immediates. The ComplexPattern ADDri is a special operand used for memory accesses that need custom handling during the instruction selection stage[33].

```
def CallTarget: Operand<i32> { } // Call operand
def BrTarget16: Operand<OtherVT> { } // Branch 16-bit operand
def BrTarget26: Operand<OtherVT> { } // Branch 26-bit operand
def simm16: Operand<i32> { let PrintMethod= "printSImm16Operand";}
def zimm16: Operand<i32> { let PrintMethod= "printZImm16Operand";}
def CCOp: Operand<i32> { let PrintMethod = "printCCOperand"; }

def LO16: SDNodeXForm<imm, [{return CurDAG->getTargetConstant(
  (uint64_t)N->getZExtValue() & 0xffff, SDLoc(N), MVT::i32); }]>;

def HI16: SDNodeXForm<imm, [{return CurDAG->getTargetConstant(
  (uint64_t)N->getZExtValue() >> 16, SDLoc(N), MVT::i32); }]>;

def immShift: Operand<i32>, PatLeaf<(imm), [{ int Imm =
  N->getSExtValue(); return Imm >= 0 && Imm <= 31;}]>;

def SETHIimm: PatLeaf<(imm), [{
  return isShiftedUInt<16, 16>(N->getZExtValue()); }], HI16>;

def MEMri: Operand<i32> { let PrintMethod = "printMemRIOperand";
  let MIOperandInfo = (ops GPRegs, i32imm); }

def ADDRri: ComplexPattern<i32,2,"selectAddrRi",[frameindex],[]>;
```

The example below shows the definition of two DLX instructions. As every instruction in the DLX target, both of those instructions are defined using one of the format base class, namely the I-type format base class. The ADDI instruction uses a signed 16-bit immediate operand and one register operand as inputs and one register operand as output. The BNEZ instruction uses one register operand and a BrTarget16 operand as inputs and has no output operands.

```
def ADDI : FI < 0b000000, (outs GPRegs:$dst),
  (ins GPRegs:$src1, simm16:$c), "addi $dst, $src1, $c",
  [(set i32:$dst, (add i32:$src1, immSExt16:$c))]>;

let isBranch = 1, isTerminator = 1 in {
def BNEZ : FI<0b000000, (outs),
  (ins GPRegs:$src1, BrTarget16:$addr), "bnez $src1, $addr",
  [(brcond GPRegs:$src1, bb:$addr)]>;
}
```

4.6 Defining Calling Conventions

DLX calling conventions are defined in the *ZRCallingConv.td* TableGen file and they specify[33]:

- Where parameters and return values are placed.
- Which registers may be used for parameters or return values.
- The order of parameter allocation.

```
// DLX calling convention
def CC_ZR : CallingConv<[
  CCIfType<[i8, i16], CCPromoteToType<i32>>,
  CCIfType<[i32, f32], CCAssignToStack<4, 4>>,
  CCIfType<[f64], CCAssignToStack<8,4>>
]>;

// DLX return value calling convention
def RetCC_ZR : CallingConv<[
  CCIfType<[i32], CCAssignToReg<[R1, R9, R10, R11]>>,
  CCIfType<[f32], CCAssignToReg<[F0]>>,
  CCIfType<[f64], CCAssignToReg<[D0]>>
]>;
```

As seen above, the DLX target defines two types of calling conventions: one for calling a function and the second one for returning a value from a function. The CC_ZR calling convention places all values on the stack with the specified alignment and promotes all i8 and i16 values to i32 values. The RetCC_ZR calling convention specifies in which registers, values should be returned. It specifies four registers for integer values to allow returning values up to 128 bits. Float values and double values are returned in the F0 (D0) register.

4.7 Target Code Generation

The target code generation in LLVM is done using numerous stages where every stage performs a different task which gradually transforms the input LLVM IR into the final machine code (the DLX assembly code in this case). It is divided into the following stages (also demonstrated on the figure 4.1)[34]:

- **Build initial SelectionDAG** - Performs a simple translation from the input LLVM IR to an illegal SelectionDAG.
- **Legalize SelectionDAG Types** - Transforms a SelectionDAG to eliminate any types that are unsupported by the target.

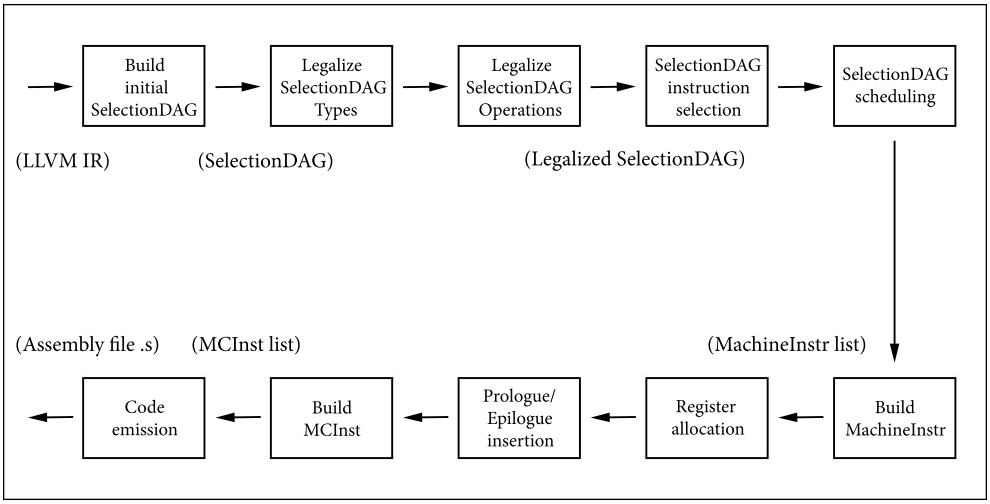

Figure 4.1: Code Generation Stages

- **Legalize SelectionDAG Operations** - Transforms a SelectionDAG to eliminate any operations that are unsupported by the target.
- **SelectionDAG instruction selection** - Matches the DAG operations to target instructions and translates the input SelectionDAG into another SelectionDAG of target instructions.
- **SelectionDAG scheduling** - Assigns a linear order to instructions from the input SelectionDAG.
- **Build MachineInstr** - Converts the scheduled target instruction SelectionDAG into the list of MachineInstrs.
- **Register allocation** - Performs mapping virtual registers to physical registers.
- **Prologue/Epilogue code insertion** - Inserts the prologue and epilogue of functions.
- **Build MCInst** - Translates the MachineInstr list to the list of MCInsts.
- **Code emission** - Emits the machine assembly code (object file).

Each stage usually contains a several passes. The LLVM system performs around 70 passes to translate the LLVM IR into the final machine code. Some of these passes work independently but others require some additional custom code to handle the DLX target specifics. Additionally, the LLVM system performs some optimizations in certain stages and passes. The next chapter 5 gives more details about optimizations in the LLVM system[34].

4.8 Build Initial SelectionDAG

The first stage creates the initial SelectionDAG structure. The SelectionDAG is a directed acyclic graph whose nodes are instances of the SDNode class. The SDNode consists of operation code (Opcode) that indicates what operation the node performs and the operands to the operation which are edges to the nodes defining the used value. Each value produced by an SDNode has an associated MVT (Machine Value Type) indicating what the type of the value is. The edges may represent two kinds of values: those that represent data flow and those that represent control flow dependencies called chain edges. These chain edges provide ordering between nodes and by convention they are always the first operand. The glue operand glues nodes together to avoid separation in SelectionDAG stages[34]. Usually, the LLVM SelectionDAG SDNodes are not enough to represent the target machine constructs entirely and some custom SDNodes must be defined as is the case of the DLX target. DLX defines several custom SDNodes to represent target dependent constructs in the ZRISD namespace in the *ZRISelLowering.h* file and their description is in the 4.10 section.

This stage performs a simple translation from the input LLVM IR to an illegal SelectionDAG and is mostly hard-coded, an LLVM IR ”add” turns to an SDNode add, while instructions like ”getelementptr” are expanded into the appropriate arithmetics. This stage also requires target specific hooks to handle calling conventions, in particular, it needs custom code to handle calls, returns, and formal arguments. For these, the TargetLowering interface is used[34]. The DLX target utilize the TargetLowering interface in the *ZRISelLowering.h* file where three of the TargetLowering methods are overridden to properly handle DLX calling conventions:

- **LowerFormalArguments()** - This method lowers the incoming (formal) arguments and either assigns them to registers or creates for them a space on stack according to the used calling convention. It also handles variable-length argument lists[36].
- **lowerReturn()** - This method lowers the return based on the specified calling convention. It creates a custom ZRISD::RetFlag SDNode glued to a resulting value to represent the return.
- **lowerCall()** - This method lowers calls into specified SelectionDAG SDNodes. It creates a custom ZRISD::Call SDNode to represent the call and also creates a ZRISD::CALLSEQ_START SDNode and a ZRISD::CALLSEQ_START SDNode to mark the beginning and the end of the function call.

The following LLVM IR code is used to illustrate how the target code generation process works. It is one simple function that takes one integer and

one floating-point number and adds them together along with a floating-point constant 5.00. The integer argument is converted to a floating-point value first. Due to limited space, the LLVM IR code example must be this small because the resulting SelectionDAGs are quite large.

```
define float @addFPandINT(float, i32) {
  %3 = sitofp i32 %1 to float
  %4 = fadd float %3, %0
  %5 = fadd float %4, 5.000000e+00
  ret float %5
}
```

The figure 4.2 shows the initial SelectionDAG built for the LLVM IR example above. The SelectionDAG has some nodes with the same Opcodes (LLVM is not too consistent - the Opcodes are usually slightly different) as the input LLVM IR (fadd, sint_to_fp) but has some others nodes which came from lowering the calling convention. Firstly, the LowerFormalArguments() method placed the function arguments on the stack and also generated corresponding loads. Secondly, the lowerReturn() method created a custom ZRISD::RET_FLAG SDNode glued together with the resulting value which is placed in the register F0. This exactly follows the DLX calling convention.

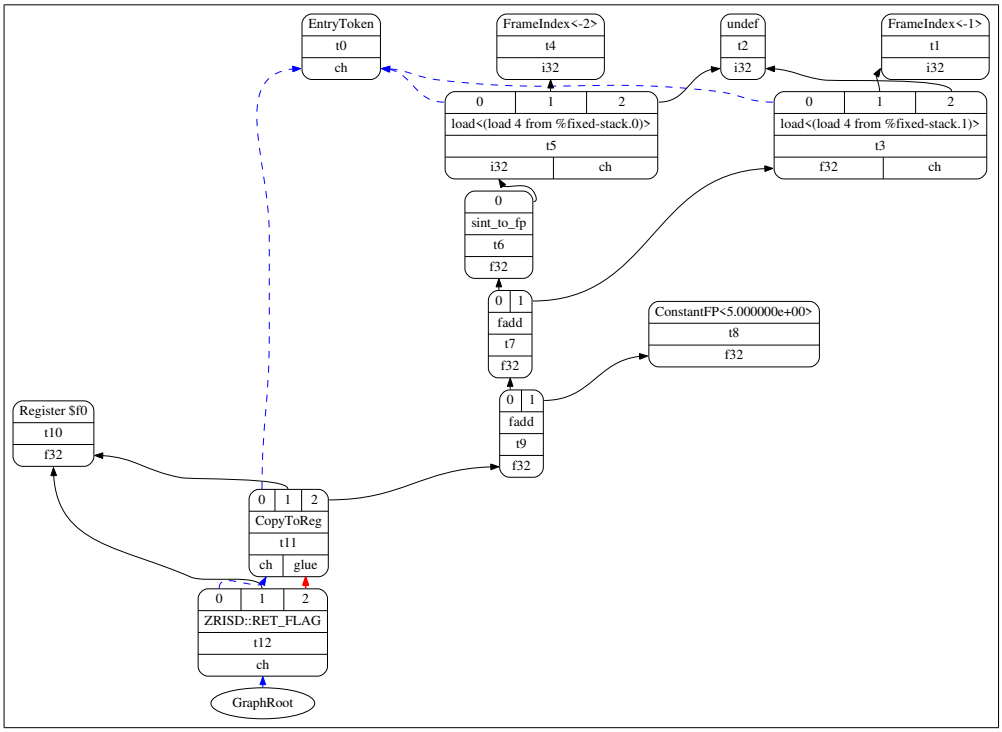

Figure 4.2: SelectionDAG of the addFPandINT function after construction

4.9 Legalize SelectionDAG Types

This legalize stage is in the charge of converting a SelectionDAG to only use the types that are natively supported by the target. A target implementation tells the LLVM system which types are supported and which register classes to use for them by calling the addRegisterClass() method[34].

DLX target supports three types of registers and calls the addRegisterClass() method for each of them in ZRTargetLowering class constructor in the *ZRISelLowering.cpp* file which tells the LLVM legalizer to fit all values in one of these registers. The usage of the addRegisterClass() in the DLX target is shown below. There are two main ways of converting values of unsupported types to values of supported types: converting small types to larger types called promoting (for example i1 or i8 value will become i32 value and will be stored in an integer register), and breaking up large types into smaller ones called expanding (for example i64 and i128 values will be expended to 2 and 4 i32 values and they will be stored in multiple integer registers)[34].

```
addRegisterClass(MVT::i32, &ZR::GPRegsRegClass);
addRegisterClass(MVT::f32, &ZR::FPRegsRegClass);
addRegisterClass(MVT::f64, &ZR::DFPRegsRegClass);
```

4.10 Legalize SelectionDAG Operations

This stage transforms a SelectionDAG to eliminate any operations that are unsupported by the target. A target must tell the LLVM legalizer which operations are not supported and what to do with them using the setOperationAction() method. The setOperationAction() method takes three arguments: SDNode Opcode that represents the operation, a type that may refer to either the type of a result or that of an operand and an action which may be one the following[33]:

- **Expand** - The operation is expanded into multiple operations by the LLVM system, for example, the SREM operation is expanded into the SDIV operation and the appropriate arithmetics to get the reminder. The DLX target expends BR_CC, BRCOND, BR_JT, SELECT, STACKSAVE, STACKRESTORE, FNEG operations and many others that can be found along with the types for which they are expanded in the *ZRISelLowering.cpp* file. This has its limitations and cannot handle all cases.

- **Legal** - The legal action simply indicates that an operation is supported by the target. Since the legal action is the default action for all operations, it is rarely used. Basically, the only usage of this action is to override the action when using subtargets. The DLX target does not use this action.

- **Custom** - For some operations expansion may be insufficient and some custom code must be implemented to properly handle the operation.

The DLX target provides custom handling for the following operations and types. For each operation registered with the custom action, a case statement is added in the LowerOperation() method to indicate what function to call to handle the operation. The DLX target names these functions as LowerXXX() methods where XXX stands for the SDNode Opcode, for example when legalizing the BR_CC operation, the LowerBR_CC() method gets called.

```
setOperationAction(ISD::BR_CC, MVT::f32, Custom);
setOperationAction(ISD::BR_CC, MVT::f64, Custom);
setOperationAction(ISD::SELECT_CC, MVT::i32, Custom);
setOperationAction(ISD::SELECT_CC, MVT::f32, Custom);
setOperationAction(ISD::SELECT_CC, MVT::f64, Custom);
setOperationAction(ISD::VASTART, MVT::Other, Custom);

setOperationAction(ISD::GlobalAddress, MVT::i32, Custom);
setOperationAction(ISD::BlockAddress, MVT::i32, Custom);
setOperationAction(ISD::JumpTable, MVT::i32, Custom);
setOperationAction(ISD::ConstantPool, MVT::i32, Custom);

setOperationAction(ISD::DYNAMIC_STACKALLOC, MVT::i32, Custom);

for (MVT VT : MVT::integer_valuetypes()) {
  setOperationAction(ISD::SINT_TO_FP, VT, Custom);
  setOperationAction(ISD::FP_TO_SINT, VT, Custom);
}
```

All of these SDNodes with custom handling are transformed to one or multiple custom SDNodes (except for VASTART SDNode which is transferred into a simple store SDNode). The DLX target uses the following custom SDNodes:

- **ZRISD::CALL** - A necessary custom SDNode to represent call. It is used in the lowerCall() method when building a SelectionDAG.
- **ZRISD::HI** - Used to represent the high 16 bits of addresses.
- **ZRISD::LO** - Same as ZRISD::HI but represents the lower 16 bits.
- **ZRISD::ADJDYNALLOC** - Represents a dynamic size allocation on stack.
- **ZRISD::RET_FLAG** - Another necessary custom SDNode to represent return. It is used in the lowerReturn() method when building a SelectionDAG.
- **ZRISD::FBR_CC** - Represents a floating-point branch and it is always glued together with a ZRISD::FSET_FLAG SDNode.

- **ZRISD::FSET_FLAG** - Represents a floating-point compare. The FSET_FLAG SDNode has three operands, two representing the values to compare and a condition code operand that represents the condition. The condition codes are defined in the *ZRCondCodes.h* file.
- **ZRISD::MOVE** - This SDNode is used to represent move instructions. It is also a part of conversion between a floating-point value and an integer value.
- **ZRISD::CONVERTI2F** - Represents a conversion from an integer value to a floating-point value.
- **ZRISD::CONVERTF2I** - Represents a conversion from a floating-point value to an integer value.
- **ZRISD::XSELECTX_CC** - This is a collection of SDNodes that represent a select_cc SDNode. Select_cc is not a real instruction in LLVM but comes from expanding a select operation. The expansion of a select operation merges the condition and the operands together into one SDNode. Thus, a select_cc SDNode has five operands: two values to compare, two values to choose from and a condition to compare the values with. There are four types based on the types of values which is necessary because DLX handles differently integer conditions and floating-point conditions:
  - **ZRISD::FSELECTF_CC** - Choose a floating-point value based on a floating-point condition. This SDNode has two floating-point operands to choose the value from and it is glued together with the FSET_FLAG SDNode which represents a floating-point compare.
  - **ZRISD::ISELECTI_CC** - Choose an integer value based on an integer condition. This SDNode has two integer operands to choose the value from and another operand that represents an integer compare (setcc SDNode).
  - **ZRISD::ISELECTF_CC** - Choose an integer value based on a floating-point condition.
  - **ZRISD::FSELECTI_CC** - Choose a floating-point value based on an integer condition.

The example below illustrates a custom code that transforms a floating-point BR_CC SDNode into two custom SDNodes that are glued together: ZRISD::FSET_FLAG and ZRISD::FBR_CC. The ZRISD::FSET_FLAG SDNode takes two values to compare (LHS, RHS) and a condition code (TargetCC). The ZRISD::FBR_CC only takes a basic block (Dest) which is a target address when the branch is taken.

```
SDValue
ZRTargetLowering::LowerBR_CC(SDValue Op, SelectionDAG &DAG) const {
  SDValue Chain = Op.getOperand(0);
  ISD::CondCode CC= cast<CondCodeSDNode>(Op.getOperand(1))->get();
  SDValue LHS = Op.getOperand(2);  SDValue RHS = Op.getOperand(3);
  SDValue Dest = Op.getOperand(4);
  SDLoc DL(Op);

  if(LHS.getValueType().isFloatingPoint() && RHS.getValueType()
        .isFloatingPoint()) {
   LPCC::CondCode ZRCC = FPCondCCodeToFCC(CC);
   SDValue TargetCC = DAG.getConstant(ZRCC, DL, MVT::i32);
   SDValue Flag =
   DAG.getNode(ZRISD::FSET_FLAG,DL,MVT::Glue, LHS, RHS, TargetCC);

   return DAG.getNode(ZRISD::FBR_CC, DL, LHS.getValueType(),
        Chain, Dest, Flag); }
}
```

The figure 4.3 shows the legalized SelectionDAG for the addFPandINT function. The sint_to_fp SDNode was transformed to a custom ZRISD::MOVE SDNode and a custom ZRISD::CONVERTI2F SDNode which closely corresponds to DLX instructions used to convert integer values to floating-point values. The constant pool value 5.00 was transformed to a ZRISD::HI SDNode and a ZRISD::LO SDNode to retrieve the high 16 bits and the low 16 bits of the address and a load SDNode to load the constant.

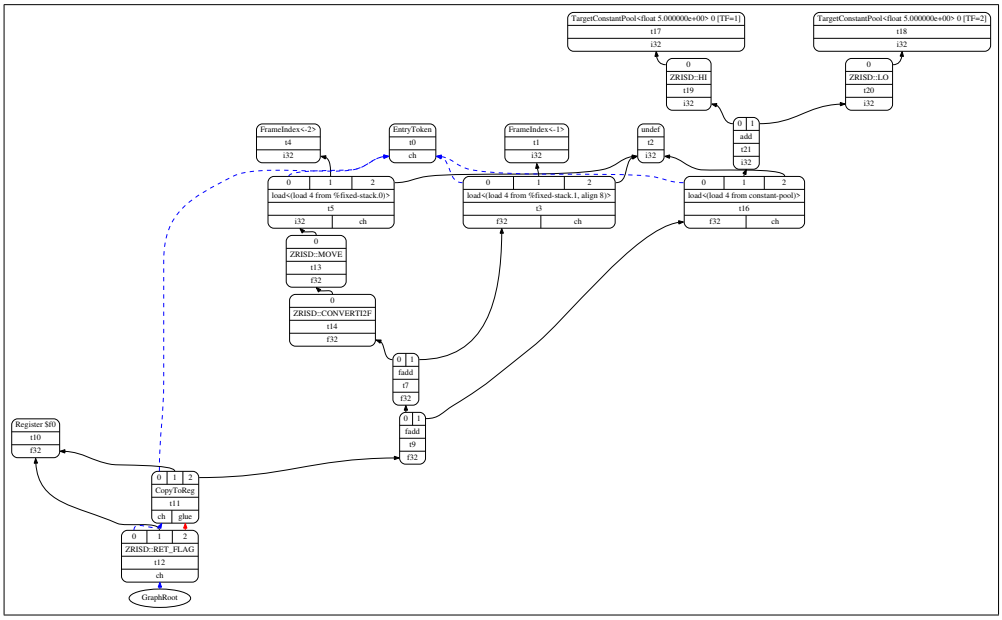

Figure 4.3: SelectionDAG of the addFPandINT function after legalization

4.11 SelectionDAG Instruction Selection

The instruction selection stage takes a legalized SelectionDAG as an input, pattern matches the instructions supported by the target and produces a new SelectionDAG of target instructions. A legalized SelectionDAG can also be matched by pseudo instructions which are transformed into target instructions in ensuing stages because they require some information not known in this stage. Most of the selection process is handled using the TableGen instruction selector which reads target instruction patterns in the *ZRInstInfo.td* TableGen file and automatically builds parts of the pattern matching code for the DLX target. The following code shows some examples of such patterns used in the DLX target (the pattern is specified in square brackets)[34].

```
def ADD : FR<..., [(set i32:$dst, (add i32:$src1, i32:$src2))]>;
def ADDUI: FI<...,[(set i32:$dst,(add i32:$src1, immZExt16:$c))]>;
def ADDF : FR<..., [(set f32:$dst, (fadd f32:$src1, f32:$src2))]>;
def LWri : FI<..., [(set i32:$dst, (load ADDRri:$addr))]> {...}
def LFri : FI<..., [(set f32:$dst, (load ADDRri:$addr))]> {...}
def CVTF2I : FR<..., [(set f32:$rd, (ConvertF2I f32:$rs))]>;
def MOVI2FP : FR<..., [(set f32:$rd, (Move i32:$rs))]>;
```

As shown above, the ADD instruction pattern matches a simple add SDNode with two i32 operands, the ADDUI instruction pattern matches one i32 operand and a 16-bit unsigned immediate operand. The LWri and LFri instruction patterns match loads from memory with corresponding types. The CVTF2I instruction pattern matches a custom ZRISD::CONVERTF2I SDNode and the MOVI2FP instruction pattern matches a ZRISD::MOVE SDNode. As mentioned earlier, a legalized SelectionDAG is not only matched by real target instructions but can also be matched by pseudo instructions. The following code shows some pseudo instructions that are used in the DLX target.

```
let isCall=1, Uses=[R30], Defs=[R31] in {
 def CALLI: Pseudo<(outs),(ins CallTarget:$addr),"",[]>;
 def CALLR: Pseudo<(outs),(ins GPRegs:$Rs1),"",[(Call i32:$Rs1)]>;
}

let isReturn=1, isTerminator=1, isBarrier=1, Uses=[R2] in {
 def RET: Pseudo<(outs), (ins), "jr r2", [(RetFlag)]>;
}

let Defs = [R30], Uses = [R30] in {
 def ADJDYNALLOC: Pseudo<(outs GPRegs:$dst),(ins GPRegs:$src), "",
     [(set GPRegs:$dst, (ZRAdjDynAlloc GPRegs:$src))]>;
}
```

The first two pseudo instructions match function calls, the RET pseudo instruction matches a custom ZRISD::RetFlag SDNode and the ADJDYNALLOC pseudo instruction matches an another custom ZRISD::ADJDYNALLOC SDNode. The DLX target also uses ADJCALLSTACKDOWN and ADJ-

CALLSTACUP pseudo instructions to match the function beginning and the function end and ZRISD::XSELECTX_CC SDNodes are also matched by pseudo instructions with the same names as those SDNodes have.

The previous code also shows that some additional properties must be set to instructions to tell the LLVM system how to handle them. For example, the "isCall" property tells the LLVM system that the instruction is a call and cannot simply be removed during optimizations. The "Uses" property is a list of non-operand registers that the instruction uses and the "Defs" property is a list of non-operand registers which are modified by the instruction. There is a whole bunch of these prosperities than can (or must) be set to an instruction like whether the instruction stores to memory or loads from memory, if it has side effects if it is a branch or a terminator instruction and etc[37].

In addition to instructions patterns, targets can specify arbitrary patterns to the TableGen instruction selector that map to one or more instructions using the Pat class. The following code shows the Pats defined in the DLX target[34].

```
def : Pat<(i32 imm:$imm), (ORI (SETHIi (HI16 imm:$imm)),
                                                (LO16 imm:$imm))>;

def : Pat<(extloadi8  ADDRri:$src), (i32 (LBri ADDRri:$src))>;
def : Pat<(extloadi16 ADDRri:$src), (i32 (LHri ADDRri:$src))>;

def : Pat<(Call tglobaladdr:$dst), (CALLI tglobaladdr:$dst)>;
def : Pat<(Call texternalsym:$dst), (CALLI texternalsym:$dst)>;

def : Pat<(HI tconstpool:$dst), (SETHIi tconstpool:$dst)>;
def : Pat<(LO tconstpool:$dst), (ORI (i32 R0), tconstpool:$dst)>;

def : Pat<(HI tglobaladdr:$dst), (SETHIi tglobaladdr:$dst)>;
def : Pat<(LO tglobaladdr:$dst), (ORI (i32 R0),tglobaladdr:$dst)>;
```

The first Pat class maps an arbitrary i32 value to an ORI instruction (or a 16-bit immediate) and a SETHIi instruction (set a 16-bit immediate, where the immediate is shifted to the left 16 bits). The second Pat maps extload SDNode (extend load where the top bits are undefined) with signed load instructions. The rest of Pat classes handle different types of symbols. Those symbols that are represented as ZRISD:HI and ZRISD:LO SDNodes from the previous stage, get translated to a SETHi and an ORI instruction to load the 32-bit symbol address to a register. The last two Pat classes are also defined for texternalsym, tblockaddress and tjumptable symbols.

While the TableGen instruction selector has many strengths, the DLX target still needs some custom code to properly match all legalized SDNodes with target instructions. For example, the TableGen instruction selector cannot handle ComplexPatterns that support complex addressing modes or handle matching stack frame values[34]. The selection of a ComplexPattern ADDRri is done in the selectAddrRi() method in the *ZRISelDAGtoDAG.cpp* file and

it matches the memory address with a base register and an appropriate offset to get the value from memory. Stack frame values are matched with a new TargetFrameIndex object in the selectFrameIndex() method in the same file.

The figure 4.4 shows the SelectionDAG of target instructions for the addFPandINT function after the instruction selection stage. As can be seen, every legalized SDNode was matched by some target instruction except for the RetFlag SDNode which was matched by the RET pseudo instruction. The address of the constant value 5.00 is loaded using the SETHi instruction and the ORI instruction. Loads turned into LWri and LFri instructions based on the type of the value they load.

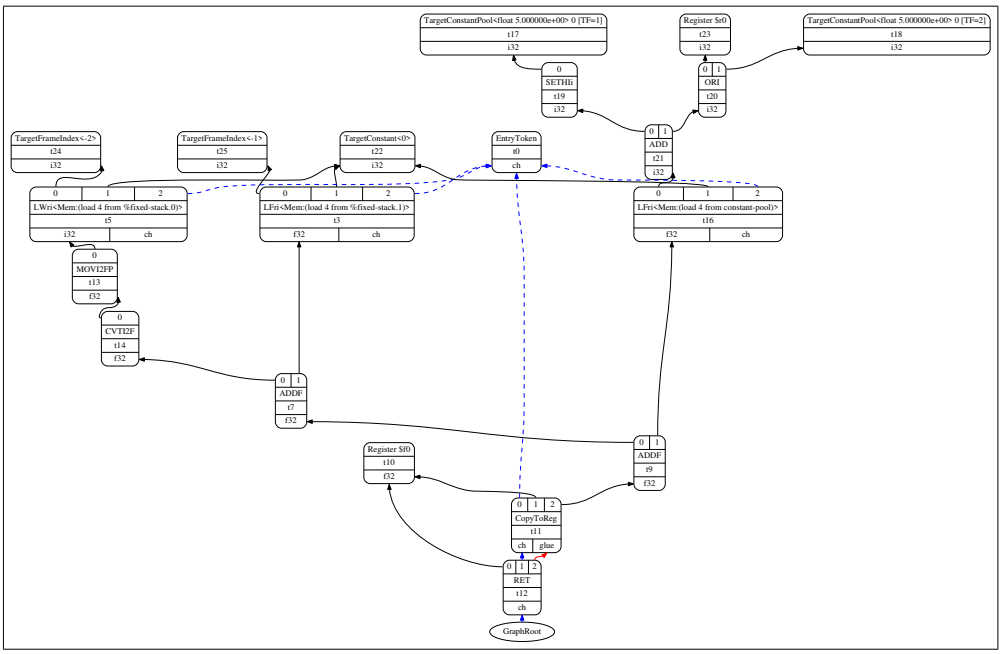

Figure 4.4: SelectionDAG of the addFPandINT function after instruction selection

4.12 SelectionDAG Scheduling

The scheduling stage takes a SelectionDAG of target instructions and assigns them a linear order. The LLVM system takes care of the whole process of assigning an order to instructions based on the information contained in the SelectionDAG. The order of instructions will not be optimal and can further be optimized which is described in the following chapter 5. The order of instructions for the addFPandINT function can be seen on the figure 4.5.

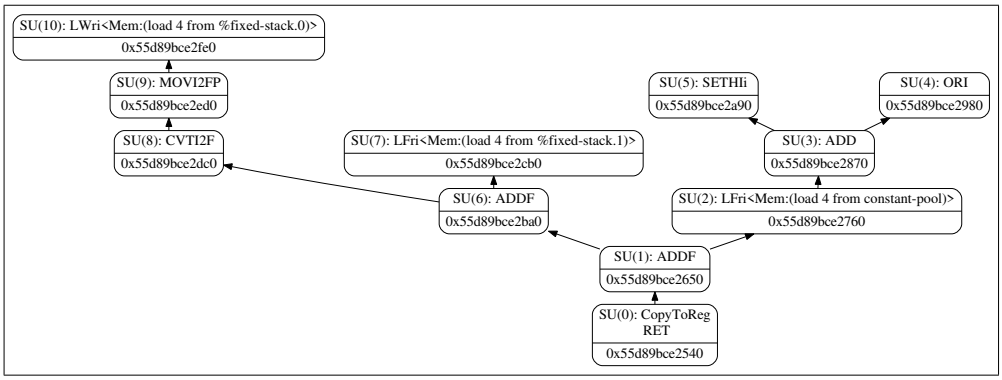

Figure 4.5: SelectionDAG of the addFPandINT function after scheduling

4.13 Build MachineInstr

In this stage, a scheduled SelectionDAG is transformed into a machine specific representation formed out of MachineFunctions, MachineBasicBlocks, and MachineInstrs. This representation is an extremely abstract way of representing machine instructions, in particular, it only keeps track of an opcode number, a set of operands and certain flags to describe the instruction. The opcode number is a simple unsigned integer that only has meaning to a specific target. MachineInstrs are initially created in an SSA form and are maintained in an SSA form until register allocation happens. For the most part, this is trivially simple since the LLVM IR is already in an SSA form and LLVM PHI nodes become machine code PHI nodes, and virtual registers are only allowed to have a single definition. This stage is mostly hardcoded in the LLVM system and relies on the information generated from the *ZRInstInfo.td* TableGen file to generate the MachineInstrs but there are still certain things that need to be handled specifically.

XSELECTX_CC pseudo instructions are lowered into target instructions in this stage. Those pseudo instructions are marked with a "usesCustomInserter" flag as shown below to tell the LLVM system that those instructions need special support when converting to MachineInstrs. The specified MachineInstrs for those instructions are created anyway but they are not inserted into any basic blocks. This causes the EmitInstrWithCustomInserter() method in the *ZRISelLowering.cpp* file to get called to handle the conversion, generally this method can expand the MachineInstrs into a sequence of instructions, potentially creating new basic blocks and control flow[38].

```
def FSELECTF_CC : Pseudo <(outs FPRegs:$dst),
          (ins FPRegs:$src1, FPRegs:$src2),"",
          [(set f32:$dst, (ZRFSelectFCC f32:$src1, f32:$src2))]>
          { let usesCustomInserter = 1; }
```

Those XSELECTX_CC pseudo instructions represent choosing a value based on a condition. Because the DLX target does not have a select instruction they are transformed to target instructions using the following triangle control-flow pattern.

```
    ThisMBB
    |  \
    |   IfFalseMBB
    |  /
    NextMBB
```

This triangle control-flow pattern causes the control flow to change based on the condition from the XSELECTX_CC instruction by inserting a branch instruction into the ThisMMB block. The value is then chosen either in the ThisMBB basic block or in the IfFalseMBB basic block. Target PHI node with ThisMBB and IfFalseMBB as predecessors is inserted into the NextMBB basic block to keep the code in an SSA form.

The following code shows the list of MachineInstrs for the addFPandINT function. The COPY instruction was generated by the LLVM system and is handled in the following stage. The last instruction is the RET pseudo instruction. Besides instructions, it also shows frame objects (objects stored in the function stack frame) which were created in the instruction selection stage and constants used in the function.

```
Frame Objects:
  fi#-2: size=4, align=4, fixed, at location [SP+4]
  fi#-1: size=4, align=4, fixed, at location [SP]

Constant Pool:
  cp#0: 5.000000e+00, align=4

bb.0 (%ir-block.2):
  %0:fpregs = LFri %fixed-stack.1, 0::(load 4 from %fixed-stack.1)
  %1:gpregs = LWri %fixed-stack.0, 0::(load 4 from %fixed-stack.0)
  %2:fpregs = MOVI2FP killed %1:gpregs
  %3:fpregs = CVTI2F killed %2:fpregs
  %4:fpregs = ADDF killed %3:fpregs, killed %0:fpregs
  %5:gpregs = ORI $r0, target-flags(<unknown>) %const.0
  %6:gpregs = SETHIi target-flags(<unknown>) %const.0
  %7:gpregs = ADD killed %6:gpregs, killed %5:gpregs
  %8:fpregs = LFri killed %7:gpregs,0::(load 4 from constant-pool)
  %9:fpregs = ADDF killed %4:fpregs, killed %8:fpregs
  $f0 = COPY %9:fpregs
  Pseudo implicit $r2, implicit $f0
```

4.14 Register Allocation

This stages takes a list of MachineInstrs and assigns physical registers to virtual registers or spill registers to memory if all physical registers are occu-

pied[39]. Firstly, the LLVM system must be told how to copy registers. This is done using the copyPhysReg() method in the *ZRInstrInfo.cpp* file which tells the LLVM system what instructions should be used to copy registers. The ORI instruction is used to copy integer registers and FMOV, DMOV instructions are used to copy floating-point registers.

There are two ways to map virtual registers to physical registers (or to memory slots) in LLVM. The first way is called direct mapping and relies on the VirtRegMap class in order to insert loads and stores sending and getting values to and from memory. The second way is called indirect mapping and it is based on the use of methods of the TargetRegisterClass class and it is what the DLX target uses. The DLX target implements several methods to help the LLVM system to allocate registers such as storeRegToStackSlot(), loadRegFromStackSlot(), expandPostRAPseudo(), isLoadFromStackSlot(), isStoreToStackSlot, getReservedRegs(), eliminateFrameIndex(), requiresRegisterScavenging() and trackLivenessAfterRegAlloc(). These methods specify which registers can be allocated, which are reserved to special use (R0, R2 - return value, R29 - frame pointer, R30 - stack pointer, R31 - JAL instructions) and what to do when registers need to be spilled to memory. The LLVM system then assigns the actual registers using one of the register allocators described in the 5.11 section. The order of register allocation is defined in the *ZRRegisterInfo.td* TableGen file.

The following code shows the list of MachineInstrs for the addFPandINT function after the register allocation stage. It shows that all ten virtual registers were assigned into seven physical registers. The LLVM system tries to map those efficiently as shown on the mapping of the return register F0. It already loads the float argument to that register and then performs both ADDF instructions inside that register to save registers.

```
Frame Objects:
  fi#-2: size=4, align=4, fixed, at location [SP+4]
  fi#-1: size=4, align=4, fixed, at location [SP]

Constant Pool:
  cp#0: 5.000000e+00, align=4

bb.0 (%ir-block.2):
  $f0 = LFri %fixed-stack.1, 0 :: (load 4 from %fixed-stack.1)
  $r1 = LWri %fixed-stack.0, 0 :: (load 4 from %fixed-stack.0)
  $r3 = ORI $r0, target-flags(<unknown>) %const.0
  $r4 = SETHIi target-flags(<unknown>) %const.0
  $r3 = ADD killed $r4, killed $r3
  $f1 = LFri killed $r3, 0 :: (load 4 from constant-pool)
  $f2 = MOVI2FP killed $r1
  $f2 = CVTI2F killed $f2
  $f0 = ADDF killed $f2, killed $f0
  $f0 = ADDF killed $f0, killed $f1
  Pseudo implicit $r2, implicit $f0
```

4.15 Prologue/Epilog Code Insertion

This stage inserts the prologue and the epilogue code to all functions and happens in the *ZRFrameLowering.cpp* file. It begins with the calculation of the stack frame size which is done by the determineFrameLayout() method. Before the actual Prologue/Epilogue insertion, the DLX target takes care of some pseudo instructions, namely ADJDYNALLOC, ADJCALLSTACK-DOWN, and ADJCALLSTACKUP. The ADJDYNALLOC pseudo instruction is replaced by the ADDI instruction with maximum calculated call stack frame size in the replaceAdjDynAllocPseudo() method. The ADJCALLSTACKUP pseudo instruction and the ADJCALLSTACKDOWN pseudo instruction are simply removed because their usage was to indicate a place for other target code generation process. The prologue code insertion is done by the emitPrologue() method and consists of the following instructions:

```
BuildMI(MBB, MBBI, DL, LII.get(ZR::NOP));
BuildMI(MBB, MBBI, DL, LII.get(ZR::NOP));

BuildMI(MBB, MBBI, DL, LII.get(ZR::STri))
  .addReg(ZR::R30).addImm(0).addReg(ZR::R31);

BuildMI(MBB, MBBI, DL, LII.get(ZR::STri))
  .addReg(ZR::R30).addImm(-4).addReg(ZR::R29);

BuildMI(MBB, MBBI, DL, LII.get(ZR::SUBI), ZR::R30)
  .addReg(ZR::R30).addImm(4);

BuildMI(MBB, MBBI, DL, LII.get(ZR::ADDI), ZR::R29)
  .addReg(ZR::R30).addImm(8);

if (StackSize != 0) {
  BuildMI(MBB, MBBI, DL, LII.get(ZR::SUBI), ZR::R30)
    .addReg(ZR::R30).addImm(StackSize)
}
```

The first two NOP instructions are used to stall the pipeline until the JAL (JALR) writes the return address into the register R31. Next instructions sequentially push a return value on the stack, push an old frame pointer on the stack, adjust a stack pointer, generate a new frame pointer and allocate a new space on the stack if needed. The epilogue code insertion is done by the emitEpilogue() method and consists of the following instructions.

```
BuildMI(MBB, MBBI, DL, LII.get(ZR::ADDI), ZR::R30)
  .addReg(ZR::R29).addImm(0);

BuildMI(MBB, MBBI, DL, LII.get(ZR::LWri), ZR::R2)
  .addReg(ZR::R29).addImm(-4);

BuildMI(MBB, MBBI, DL, LII.get(ZR::LWri), ZR::R29)
  .addReg(ZR::R29).addImm(-8);
```

The epilogue firstly restores the stack pointer using the callee's frame pointer value, then loads the return address from the stack frame to the register R2 and at the end, it restores the old frame pointer.

The next code shows the addFPandINT function with inserted function prologue (flagged as frame-setup) and function epilogue (flagged as frame-destroy).

```
Frame Objects:
  fi#-4: size=4, align=4, fixed, at location [SP-8]
  fi#-3: size=4, align=4, fixed, at location [SP-4]
  fi#-2: size=4, align=4, fixed, at location [SP+4]
  fi#-1: size=4, align=4, fixed, at location [SP]

Constant Pool:
  cp#0: 5.000000e+00, align=4

bb.0 (%ir-block.2):
  frame-setup NOP
  frame-setup NOP
  frame-setup STri $r30, 0, $r31
  frame-setup STri $r30, -4, $r29
  $r30 = frame-setup SUBI $r30, 4
  $r29 = frame-setup ADDI $r30, 8
  $r30 = frame-setup SUBI $r30, 8
  $f0 = LFri $r29, 0 :: (load 4 from %fixed-stack.3)
  $r1 = LWri $r29, 4 :: (load 4 from %fixed-stack.2)
  $r3 = ORI $r0, target-flags(<unknown>) %const.0
  $r4 = SETHIi target-flags(<unknown>) %const.0
  $r3 = ADD killed $r4, killed $r3
  $f1 = LFri killed $r3, 0 :: (load 4 from constant-pool)
  $f2 = MOVI2FP killed $r1
  $f2 = CVTI2F killed $f2
  $f0 = ADDF killed $f2, killed $f0
  $f0 = ADDF killed $f0, killed $f1
  $r30 = frame-destroy ADDI $r29, 0
  $r2 = frame-destroy LWri $r29, -4
  $r29 = frame-destroy LWri $r29, -8
  Pseudo implicit $r2, implicit $f0
```

4.16 Build MCInst

The MC layer is used to represent code at the raw machine code level, devoid of any high-level information like constant pools or global variables. At this level, LLVM handles things like label names, machine instructions, and sections. The code in this layer is used for writing assembly files or object files. This layer consists of: **MCSymbols** which represents labels, **MCInsts** which represents instructions and **MCSections** that represents sections such as data or text sections. DLX target declares a custom .data section. The

other section .text is handled automatically by the LLVM system. The .data section is defined in *ZRTargetObjectFile.cpp* file as follows.

```
DataSection = getContext().getELFSection(
  ".data", ELF::SHT_PROGBITS, ELF::SHF_WRITE | ELF::SHF_ALLOC
 )
```

The code in this file also tells the LLVM system that all sections that it produces for the default ELF format (.bss, .rodata, etc.)[40] should be placed in the .data section as DLX does not have those sections.

Another important part of the MC layer is the **MCStreamer API** which is best thought of as an assembler API. It is an abstract API which is implemented in different ways (e.g. to output a .s file, output a .o file, etc). There are two major implementations: one for writing out a .s file (MCAsmStreamer), and one for writing out a .o file (MCObjectStreamer). The DLX target implements both of these streamers even thought the ZRMCObjectStreamer is never used and the ZRMCAsmStreamer relies mostly on the default implementation of the LLVM system. DLX MCStreamers are defined in the *ZRTargetStreamer.cpp* file in the MCTargetDesc folder. This folder contains files with properties of the target assembly code: the *ZRMCExpr.cpp* file handles assembly expressions such as high and low parts of the addresses of labels; the *ZRMCBaseInfo.cpp* file defines certain flags used by the DLX target and the *ZRMCTargetDesc.cpp* file contains registration of necessary target description components for the DLX target as shown below.

```
extern "C" void LLVMInitializeZRTargetMC() {
// Register the MC asm info.
RegisterMCAsmInfo<ZRMCAsmInfo> X(getTheZRTarget())

// Register the MC instruction info.
TargetRegistry::RegisterMCInstrInfo(getTheZRTarget(), ... )

// Register the MC register info.
TargetRegistry::RegisterMCRegInfo(getTheZRTarget(), ... )

// Register the object target streamer.
TargetRegistry::RegisterObjectTargetStreamer(getTheZRTarget() ,...)

// Register the asm streamer.
TargetRegistry::RegisterAsmTargetStreamer(getTheZRTarget(), ... )

// Register the MC subtarget info.
TargetRegistry::RegisterMCSubtargetInfo(getTheZRTarget(), ... )

// Register the MCInstPrinter.
TargetRegistry::RegisterMCInstPrinter(getTheZRTarget(), ... )
```

The last *ZRMCAsmInfo.cpp* file contains assembler specific information about the DLX target. The DLX target uses the following assembler settings.

```
PrivateGlobalPrefix = "";
PrivateLabelPrefix = "";
LinkerPrivateGlobalPrefix = "";
ExceptionsType = ExceptionHandling::None;
Data32bitsDirective = "\t.word\t";
HasMachoZeroFillDirective = false;
ZeroDirective = "\t.space\t";
HasIdentDirective = false;
UseIntegratedAssembler = true;
HasDotTypeDotSizeDirective = false;
HasSingleParameterDotFile = false;
CommentString = ";";
GlobalDirective = "\t.global\t";
SupportsDebugInformation = false;
UsesNonexecutableStackSection = false;

bool ZRMCAsmInfo::shouldOmitSectionDirective(StringRef
    SectionName) const {
    return true;
}
```

As shown above, the DLX target does not use any prefixes for any type of labels, does not use any labels for debugging, has no .ident and .zerofill directive. It sets the 32-bit directive to .word and the global directive to .global. It also overrides the method shouldOmitSectionDirective() to omit .section directive before all sections.

The actual lowering from MachineInstrs to MCInsts is then done in the *ZRMCInstLower.cpp* file using the Lower() method. This method takes MachineInstr instructions one by one and creates for them a new MCInst instruction. The conversion is pretty straightforward as only the Opcode is copied and operands are lowered using the LowerOperand() method which either creates a register, an immediate or a symbol.

The MC layer is highly coupled with the target assembly (target object file) and therefore is highly coupled with the next code emission stage. Basically, the MC layer's main purpose is to be an underlying layer for emitting either a .s file (.o file).

4.17 Code Emission

The last stage is responsible for writing the target assembly code (.s file). There are two main classes which are responsible for emitting the DLX assembly code: the ZRAsmPrinter class and the ZRInstPrinter class. Firstly, the ZRAsmPrinter utilizes the underlying MC layer and invokes the EmitInstruction() method which causes the ZRInstPrinter class to print the instruction. This method also transforms the CALLI (CALLR) pseudo instruction into two instructions: "SUBI R30, R30, 4" instruction to make room for a return address and "JAL (JALR) CallTarget" instruction to perform the call.

The ZRInstPrinter class is responsible for the actual .s file being emitted. It uses the TableGen ZRGenAsmWriter.inc file which contains printing methods generated from the instruction print patterns as shown below.

```
def ADD : FR < ..., "add $dst, $src1, $src2", ... >;
def LFri : FI < ..., "lf $dst, $addr", ... > {...}
def CVTF2I : FR< ..., "cvtf2i $rd, $rs", ... >;
def MOVI2FP : FR< ..., "movi2fp $rd, $rs", ... >;
```

TableGen generates two methods based on these instruction print patterns: getRegisterName() method which returns the assembler name for the specified register and the printInstruction() method which prints the instruction based on the instruction pattern. The DLX target still needs to implement routines to print custom operands such as: printZImm16Operand(), printSImm16Operand(), printCCOperand() and printMemRIOperand(). The following code shows the final assembly code for the addFPandINT function which can be run on a DLX simulator. The floating-point constant is labeled as CPI0_0 and is stored using the .word directive which is more general than using the .float directive. The "CPI0_0 » 16" is used to get the high 16 bits of CPI0_0 and the "CPI0_0 & 0x0000ffff" is used to get the low 16 bits.

```
    .text
    .data
    .align  2
CPI0_0:
    .word   1084227584

    .text
    .global addFPandINT
addFPandINT:
    nop
    nop
    sw 0(r30), r31
    sw -4(r30), r29
    subi r30, r30, 0x4
    addi r29, r30, 0x8
    subi r30, r30, 0x8
    lf f0, 0(r29)
    lw r3, 4(r29)
    ori r4, r0, (CPI0_0 & 0x0000ffff)
    lhi r5, (CPI0_0 >> 16)
    add r4, r5, r4
    lf f1, 0(r4)
    movi2fp f2, r3
    cvti2f f2, f2
    addf f0, f2, f0
    addf f0, f0, f1
    addi r30, r29, 0x0
    lw r2, -4(r29)
    lw r29, -8(r29)
    jr r2
```

4.18 Target Registration

In order to use the new DLX backend in LLVM tools like llc or clang, it needs to be registered, so LLVM tools can look up and use the target at runtime. Firstly, the DLX source codes must be inserted in the lib/Target/ folder where LLVM expects to find the DLX backend. Secondly, all targets must declare a global Target object which is used to represent the target during registration. This is done using the RegisterTargetMachine template as shown below.

```
extern "C" void LLVMInitializeZRTarget() {
  RegisterTargetMachine<ZRTargetMachine>registered_target(
      getTheZRTarget());
}
```

Additionally, these following files must be edited to fully register a new backend:

- **CMakeLists.txt** - The DLX target must be added to the list of compiled targets.
- **llvm/lib/Target/LLVMBuild.txt** - The ZR folder must be added to the list of folders that are visited when LLVM looks for source code files.
- **llvm/include/llvm/ADT/Triple.h** - A new target triple must be created in ArchType enum (a triple is a string that describes the target).
- **llvm/lib/Support/Triple.cpp** - The DLX target must add functionality to the following methods:
  - getArchTypeName() - Return string "zr" for the Triple::ZR.
  - getArchTypePrefix() - Also return string "zr" for the Triple::ZR.
  - getArchTypeForLLVMName() - Return Triple::ZR for given string "zr".
  - parseArch() - Does exactly the same as the previous method.
  - getDefaultFormat() - Return a format of an object file. The DLX target currently returns an ELF format because there is no support for generating object files in the DLX target and some format must be specified.
  - getArchPointerBitWidth() - Return integer 32.
  - get32BitArchVariant() - Say that the DLX target is already 32-bit.
  - get64BitArchVariant() - Say there is no 64-bit variant of the DLX target.
  - getLittleEndianArchVariant() - Say there is no little-endian variant of the DLX target.

- **llvm/include/llvm/BinaryFormat/ELF.h** - A new value in the enum describing machine architectures must be created. This value needs to have a unique number. The DLX target currently uses a 250 integer. Because the DLX target will not be part of an official LLVM distribution, this number might need to be changed if other backend starts to use it[41].

This list only covers a target that generates .s files. If a new target needs to support generating object files, there are several other files that need to be modified.

4.18.1 Clang registration

The DLX target can also be registered into Clang. Clang is a very powerful tool which can adjust the output LLVM IR based on some information about the target. To register a DLX target two new files were created, namely the *ZR.h* file and the *ZR.cpp* file in the clang/lib/Basic/Targets folder. Those files expose information about the target such as CPUKind or DataLayout (this DataLayout must be kept in sync with the one used in the DLX backend). There are also some other files which need to be modified (most of these files just require copy and paste from some existing target as most of the code is just letting clang know that there is a new target):

- **clang/lib/Basic/CMakeLists.txt** - The DLX target must be added to the list of compiled targets.
- **clang/lib/CodeGen/TargetInfo.cpp** - A new class describing the ABI implementation of DLX target must be inserted. This class inherits DefaultABIInfo class and relies on the default implementation.
- **clang/lib/Driver/Driver.cpp** - Allows DLX target to be used in the Clang Driver which is a tool encapsulate logic for constructing compilation processes from a set of gcc-driver-like command line arguments[42].
- **clang/lib/Driver/ToolChains/Clang{.cpp, .h}** - Adds new arguments for the Clang Driver. The DLX target does not add any. This also needs a new class defined in the in the *ZR.h* file in the clang/lib/-Driver/ToolChains folder.
- **clang/lib/Driver/ToolChains/CommonArgs.cpp** - Pass the TargetCPU value to the Clang Driver.

4.19 Notes

This section describes four modifications that were made in the source codes of LLVM. This was necessary because the DLX architecture is very simple

but LLVM expects something little more advanced that contains common features. According to the LLVM review[43], LLVM uses a hardcoded .p2align directive to indicate the alignment in sections. This had to be changed in the *lib/MC/MCAsmStreamer.cpp* file because DLX uses a simple .align directive. The second modification was to tell LLVM not to print .comm (.lcomm) directive (directive used for common symbols) which was done in the same file by commenting out the printing of this directive. The third modification, also in the same file, was made because LLVM assumes that every target has directives for 8-bit, 16-bit, 32-bit and 64-bit values but DLX only has an 8-bit directive .byte and a 32-bit directive .world. Those directives can be renamed but not removed. Luckily it is quite easy to fix it as only two lines in a simple switch need to be commented out as the code below shows.

```
const char *Directive = nullptr;
switch (Size) {
  default: break;
  case 1: Directive = MAI->getData8bitsDirective();  break;
  //case 2: Directive = MAI->getData16bitsDirective(); break;
  case 4: Directive = MAI->getData32bitsDirective(); break;
  //case 8: Directive = MAI->getData64bitsDirective(); break;
}
if (!Directive) {
  // LLVM comment
  /* We couldn't handle the requested integer size so we fallback
  by breaking the request down into several, smaller, integers.
  Since sizes greater or equal to "Size" are invalid, we use
  the greatest power of 2 that is less than "Size" as our largest
  piece of granularity. */ }
```

This will allow splitting up 16-bit directives into two 8-bit directives and anything above 32-bit can be split into multiple 32-bit directives.

The last fix was made in the clang frontend because it uses fancy names to name values. It names everything with its name and a prefix and connects them with a dot which results in names like "main.array". The LLVM target does not have a simple way of changing those name and they end up in the final assembly code. For testing purposes, the dot was removed because the WinDLX simulator does not handle dots in label names. The following code shows the change made in the *tools/clang/lib/CodeGen/CGDecl.cpp* file.

```
if (const auto *FD = dyn_cast<FunctionDecl>(DC))
  ContextName = CGM.getMangledName(FD);
else if (const auto *BD = dyn_cast<BlockDecl>(DC))
  ContextName = CGM.getBlockMangledName(GlobalDecl(), BD);
else if (const auto *OMD = dyn_cast<ObjCMethodDecl>(DC))
  ContextName = OMD->getSelector().getAsString();<

ContextName += "" + D.getNameAsString(); // "" instead of "."
return ContextName;
```

Chapter 5

Optimizations

The term optimizations in compilers refer to the attempts that a compiler makes to produce a code that is more efficient than the obvious code. Optimization is thus a misnomer since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task[18].

Compiler optimizations must meet the following design objectives. The most important objective in writing a compiler optimization is that it is correct because if the optimized code is not correct, it does not matter how fast the code is. The second objective is that the compiler optimization must be effective in improving the performance of many input programs. Normally, performance means the speed of the program execution but sometimes it might the size of the generated code or any other measurable criteria. Besides performance, usability aspects such as error reporting and debugging are also important and the compilation time also must be kept reasonable[18]. Compiler optimizations can be divided into the following categories[44]:

- **High-level optimizations** - Operate at a level close to that of source code and they are often source language dependent.
- **Intermediate-level optimizations** - Majority of compiler optimizations falls here. They are typically language independent and performed on a intermediate code representation.
- **Low-level optimizations** - Usually specific to each architecture and performed at a code close to a target code.
- **Link-time optimizations** - Type of optimizations performed by the compiler (linker) at the link time[45].

Optimizations can also be divided by the scope of a optimization: **local** - concerning only one basic block, **global** - across basic blocks in a function, based

on an analysis of a whole function and **inter-procedural** - across functions, based on an analysis of a whole module[46].

5.1 Analysis of LLVM Optimizations

The LLVM system provides a whole selection of optimizations. Due to its modularity, it performs several optimizations on every code representation from the source code to the target code. LLVM optimizations can be divided into these four categories[47]:

- **LLVM frontend optimizations** - Various frontends can perform language dependent optimizations. For example, Clang performs Return value optimization (RVO), Named RVO (NRVO) or Copy-Elision optimization[48].
- **LLVM IR optimizations** - These optimizations perform optimizations on the LLVM IR and they are both source language and target code independent[29]. They are closely described in the following 5.1.1 section.
- **LLVM backend optimizations** - An LLVM backend performs several optimizations close to a target code. Those optimizations can be divided into two groups. The first group performs optimizations on SelectionDAGs called DAG Combiner optimizations and they are described in the 5.1.2 section. The second group performs optimizations on lists of MachineInsrts and contains optimizations such as Optimize machine instruction PHIs, Remove dead machine instructions, Machine Common Subexpression Elimination or Machine Copy Propagation Pass[34].
- **Target specific optimizations** - An LLVM backend can perform target-specific optimizations.

5.1.1 LLVM IR Optimizations

This category of optimizations is performed by the LLVM Optimizer which reads the LLVM IR in, chews on it a bit, then emits another LLVM IR, which hopefully will execute faster. In LLVM, the optimizer is organized as a pipeline of distinct optimization passes each of which is run on the input and has a chance to do something with the LLVM IR[49]. Those passes can either collect some information for other passes or transform the program. LLVM optimization passes can be divided into three categories: analysis passes, transform passes and utility passes. Analysis passes compute information that other passes can use or for debugging or program visualization purposes. This category contains passes like Basic CallGraph Construction (-basiccg) pass or Dependence Analysis (-da) pass. Transform passes can use (or invalidate) the analysis passes. Transform passes all mutate the program in some way and

contain passes like Aggressive Dead Code Elimination (-adce) pass, Combine redundant instructions (-instcombine) pass or Unroll and Jam loops (-loop-unroll-and-jam) pass. Utility passes provide some utility but don't otherwise fit categorization. For example passes to verify modules (-verify) or assign names to anonymous instructions (-instnamer) are neither analysis nor transform passes[29]. A lot of these passes are also run multiple times. Those LLVM IR optimizations are used by the opt tool which uses the following options to determine which passes it should run (it is also possible to run single passes by this tool)[50][51]:

- **O0 level** - This level means "no optimization" and it compiles the fastest and generates the most debuggable code (11 passes).
- **O1 level** - This level is the basic level for optimization as it tries to optimize the code without expanding code size (245 passes).
- **O2 level** - This level is a moderate level of optimization which enables most optimizations. This level tries not to explode too much in code size nor consume all resources while compiling (264 passes).
- **O3 level** - This level is like the O2 level, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster) (267 passes)
- **Os level** - Os level is like O2 with some optimizations dropped to reduce the code size (251 passes).
- **Oz level** - Like Os but reduces code size further. Does not even try to run things that could potentially increase code size (247 passes).

5.1.2 LLVM Backend Optimizations - the DAG Combiner

The DAG Combiner optimization pass is run two times for the target code generation process on the SelectionDAG representation, immediately after a SelectionDAG is built and the second pass is run after the legalization stage[34].

The first run of this pass allows the initial code to be cleaned up (e.g. performing optimizations that depend on knowing that the operators have restricted type inputs). In other words, the initial SelectionDAG is built to support an arbitrary target without knowing anything about the actual target and this pass allows to eliminate things which are not relevant to the actual target. The second run of this pass cleans up the messy code generated by the legalization stage, which allows the legalization stage to be very simple as it can focus on making code legal instead of focusing on generating good and legal code. Both of these runs of this pass take target specific information into account. One important class of optimizations performed by the DAG Combiner is optimizing inserted sign and zero extension instructions. Consider

the following LLVM IR example which is a simple function that takes two bytes and adds them together[34].

```
define i8 @addBytes(i8, i8) {
    %3 = add i8 %0, %1
    ret i8 %3
}
```

The figure 5.1 shows the initial SelectionDAG built for the addBytes function. It contains an SDNode for a truncate operation after loading the arguments and truncates the loaded values to bytes. Those bytes are then added together and an SDNode any_extend is inserted to extend the result byte to a 32-bit integer value to fit into an integer register. Clearly, this is not needed in the DLX architecture as it operates on bytes as if they were 32-bit values.

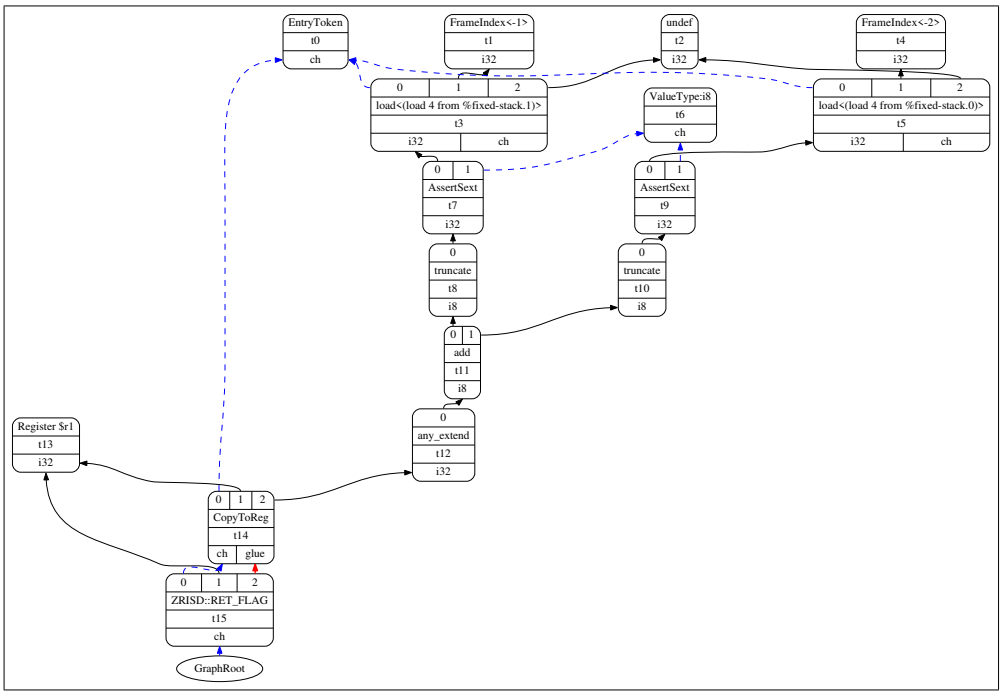

Figure 5.1: Initial SelectionDAG for the addBytes function

The figure 5.2 then shows the SelectionDAG after the first DAG combine optimization pass which takes target specific information into account. Because it now knows how the DLX target handles byte values, the DAG combine pass removed the truncate SDNode and also removed the any_extend SDNode as they are no longer needed. AssertSext SDNodes record if a register contains a value that has already been zero or sign extended from a narrower type.

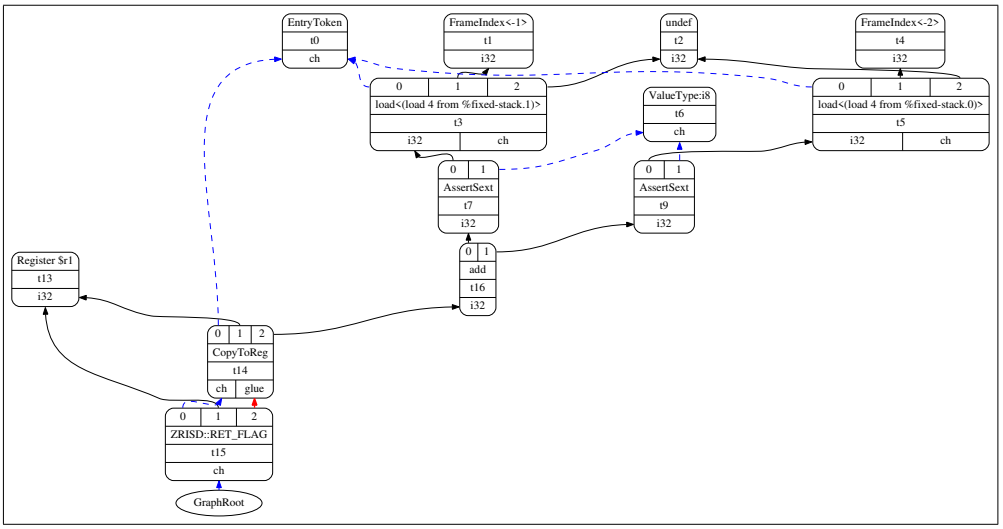

Figure 5.2: SelectionDAG for the addBytes function after the first DAG combine optimization pass

5.1.3 DLX Optimizations

This section describes which optimizations would be suitable for the DLX architecture when using the LLVM system. When compiling some source language to the DLX assembly code using the LLVM system, the code will get optimized in several places along the way. Firstly, the source language will get optimized in the frontend for that source language. That being said, there is no point in writing optimizations for DLX in frontends.

The second place where the code will be optimized is in the LLVM Optimizer if the optimizer is used. This is the place where all heavy optimizations are done using numerous passes on the LLVM IR. This set of passes provides a good chunk of optimizations for DLX as they these optimizations are source code and target code independent. A lot of these passes who handle standard optimizations on intermediate codes are already written by people who focus on optimizations and it is not very likely that those passes could be made better for the DLX target. This means that all of these passes will contribute to making the final DLX target assembly code faster.

The code will also be optimized in the LLVM backend for DLX and will have all the advantages from LLVM backend optimizations which are based on target specific information.

On the other hand, DLX can use numerous target specific optimizations. Two major ones are instruction selection and mainly instruction scheduling which can probably make most of the benefits. Then there is a lot of smaller optimizations like removing unnecessary labels, branch folding, register rema-

terialization, passing variables in registers or adding callee-saved registers and the function prologue could also be optimized to take fewer instructions.

5.2 Instruction Selection

The LLVM instruction selection process is very powerful but can be further optimized using specific patterns to tell LLVM how to select certain patterns as sometimes, there are several ways how to handle specific patterns and the LLVM instruction selector cannot know which of these patterns is better for the target. For example, the DLX target uses these following patterns to optimize the selection of immediate values[52].

```
def : Pat<(i32 immZExt16:$imm), (ADDUI (i32 R0), imm:$imm)>;
def : Pat<(i32 immSExt16:$imm), (ADDI (i32 R0), imm:$imm)>;
```

Those two patters above handle 16-bit immediate values as they match them with ADDI (ADDUI for unsigned) instructions. Consider the following function which only returns a value 8.

```
define i32 @retIntConst() {
    ret i32 8
}
```

Without these patterns, LLVM would always use two instructions to load the constant into a register, one for loading the top half of a register and a second one for loading the bottom half of a register which generally supports loading of an arbitrary constant to a register. This is shown on the following code which shows the relevant part of the example without these patterns used.

```
...
lhi r1, 0x0
ori r1, r1, 0x8
...
```

As mentioned, these patterns fit the immediate value into the ADDUI instruction if the value fits (the value 8 fits in a 16-bit value) as shown below. The order of these patterns is also important as LLVM will match the first one (ADDUI is matched in this case as the value 8 will fit in the ADDUI immediate).

```
...
addui r1, r0, 0x8
...
```

The second set of optimizing patterns used in the DLX target focuses on building target addresses. Those patterns match an add (or) SDNode with one register as the operand and a ZRISD::LO SDNode as the second operand

with the ADDUI instruction. The patterns are shown below and they are also defined for tglobaladdr, texternalsym, tblockaddress and tjumptable symbols.

```
def : Pat<(add GPRegs:$r, (LO tconstpool:$in)),
                          (ADDUI $r, tconstpool:$in)>;

def : Pat<(or GPRegs:$r, (LO tconstpool:$in)),
                          (ADDUI $r, tconstpool:$in)>;
```

To illustrate how these patterns help optimize the target code, consider the following LLVM IR function which returns a floating-point constant. The constant will be stored in the .date section with a label.

```
define float @mainFloatConst() {
    ret float 0x4013851EC0000000
}
```

The following code shows a snippet of the assembly code for the mainFloatConst function which builds the address of the constant in memory. The address is built using two instructions, the ORI instruction is used to get the low part of the address and the LHI instruction is used to get the high part of the address. Those parts are then added together to get the full address of that constant.

```
CPI0_0:
  .word 1083975926

...
ori r1, r0, (CPI0_0 & 0x0000ffff)
lhi r3, (CPI0_0 >> 16)
add r1, r3, r1
...
```

It is obvious that the ORI instruction and the ADD instruction can be merged together which is exactly what those patterns do as shown on the following code, saving one instruction and one register.

```
CPI0_0:
  .word 1083975926

...
lhi r3, (CPI0_0 >> 16)
addui r3, r3, (CPI0_0 & 0x0000ffff)
...
```

5.3 Utilizing Register R0

This optimization utilizes the target specific register R0. The register R0 is always zero which can be used to optimize certain constructs. This optimization happens in the selection instruction stage where the zero value is

matched using the following code. It materializes zero constants as copies from the register R0 which allows the coalescer to propagate these into other instructions.

```
ConstantSDNode *ConstNode = cast<ConstantSDNode>(Node);

if (ConstNode->isNullValue()) {
    SDValue New = CurDAG->getCopyFromReg(CurDAG->getEntryNode(),
        SDLoc(Node), ZR::R0, MVT::i32);
    return ReplaceNode(Node, New.getNode());
}
```

To demonstrate this optimization, consider the following C language code that sets a global variable to a zero value.

```
int a = 15;

int main() {
  a = 0;
  return 0;
}
```

Without this optimization, a zero value must first be first loaded to a register using the LHI instruction as shown below and then the store instruction can use this register to set the global variable to a zero value.

```
a:
  .word 15

...
lhi r1, 0x0
ori r3, r0, (a & 0x0000ffff)
lhi r4, (a >> 16)
or r3, r4, r3
sw 0(r3), r1
...
```

The following code shows the DLX assembly code with the utilization of the R0 register as the store can now directly use the register R0 as zero instead of loading zero to another register. This optimized code also has one more advantage and that it saves one register (this code would also be optimized by instruction patterns from the previous section).

```
a:
  .word 15

...
ori r3, r0, (a & 0x0000ffff)
lhi r4, (a >> 16)
or r3, r4, r3
sw 0(r3), r0
...
```

5.4 Instruction Scheduling

Scheduling is a process of reordering instructions to take the best advantage of the pipeline to improve performance. This optimization is among the most important for most programs. The scheduling mechanism tries to find the optimal ordering of instructions which has a minimal number of hazards and thus, has a minimal number of stalls[53]. The problem of scheduling instructions is generally an NP-complete task[54], meaning heuristics are needed to find a better ordering in a reasonable time such as register pressure, clustering, critical resources or list scheduling which are same examples of heuristics used by the LLVM system[55].

There are two types of processors: in-order and out-of-order processor. An in-order processor means that the processor will execute instructions in-order as scheduled by the compiler. Out-of-order processors can dynamically rearrange instructions in execution. DLX is an in-order processor and must rely on static scheduling by the compiler[56].

The scheduling in LLVM is divided into two parts: Pre-RA scheduling which happens before register allocation (after the instruction selection stage) and Post-RA scheduling after register allocation. There are many approaches how to handle scheduling in LLVM: ScheduleDAGRRLists, Itineraries, and the newest Machine Scheduler which is used by the DLX target[55][57].

5.4.1 Machine Scheduler

The Machine Scheduler is mainly based on TableGen files which model the target processors and need to do the following steps to describe a machine model describing the target pipeline[57].

- Define description of the pipeline and resources.
- Define own operand categories. The Machine Scheduler works in operands: input operands (what instructions need) and output operands (what instructions produce). Output operands are then associated with processor resources.
- Associate operand categories with actual instructions.

5.4.2 DLX Processors

Firstly, the DLX target must define processors it supports using the ProcessorModel class and assign them scheduling models of the pipeline. The DLX target at this moment supports these four following processors.

```
def : ProcessorModel<"generic", NoSchedModel, []>;
def : ProcessorModel<"generic-v1", ZR1SchedModel, []>;
def : ProcessorModel<"generic-v2", ZR2SchedModel, []>;
def : ProcessorModel<"generic-v3", ZR3SchedModel, []>;
```

The first one is the generic processor with no instruction scheduling as indicated by the NoSchedModel property. The generic-v1 is a processor which has all units fully pipelined. Generic-v2 and generic-v3 are processors without having all units fully pipelined. The exact number of stages for all units and their delays is described in the table 5.1.

Table 5.1: DLX Processors

Processor	Unit	Number of stages	Delay
generic-v1	ALU Unit	1	1
	FP Add Unit	2	2
	FP Multiply Unit	4	4
	FP Divide Unit	8	8
generic-v2	ALU Unit	1	1
	FP Add Unit	1	4
	FP Multiply Unit	1	4
	FP Divide Unit	1	8
generic-v3	ALU Unit	1	1
	FP Add Unit	2	2
	FP Multiply Unit	2	4
	FP Divide Unit	2	8

5.4.3 DLX Target Scheduling Model

This section describes the scheduling model for the generic-v1 processor. As mentioned before this processor has all units fully pipelined and is the easiest to describe but the changes to allow not fully pipelined units are very simple and are described in the next 5.4.4 section. This scheduling model is defined in the *ZRGenericV1.td* TableGen file.

The basic class which defines the scheduling model is shown below, it defines the top level features of the pipeline[58](the other two processors also uses this setting for their scheduling models).

```
def ZR1SchedModel : SchedMachineModel {
  let MicroOpBufferSize = 0;  // ZR1 is an in-order processor
  let IssueWidth = 1;
  let LoadLatency = 1;
  let CompleteModel = 0;
  let PostRAScheduler = 1;
}
```

The MicroOpBufferSize property set to zero indicates an in-order processor. The issueWidth property says only one instruction can be issued in a single clock cycle as DLX does not support multiple instruction issue. Load latency is set to one as DLX simulators can handle loads from memory in one clock cycle

(in real physical processors this number is much higher). It is not a complete model as pseudo instructions do not have any scheduling information. It allows scheduling after register allocation. After the description of the top level features of the pipeline, processor functional units must be defined using the ProcResource class[58]. The following code shows units used in the DLX target. They are all set to one as all units are fully pipelined.

```
let SchedModel = ZR1SchedModel in {
  def ZR1UnitALU    : ProcResource<1> { let BufferSize = 0; }
  def ZR1UnitMEM    : ProcResource<1> { let BufferSize = 0; }
  def ZR1UnitFPALU  : ProcResource<1> { let BufferSize = 0; }
  def ZR1UnitFPMUL  : ProcResource<1> { let BufferSize = 0; }
  def ZR1UnitFPDIV  : ProcResource<1> { let BufferSize = 0; }
}
```

The following code shows the operands defined for the generic-v1 processor. As DLX processors are in-order and each instruction always goes through the same pipeline, the output operands (SchedWriteRes) can represent a whole pipeline or in other words, the EX stage is the only interesting stage as it differs for certain instructions. It sets delays for all units using latencies. The store instruction uses the ZR1WriteMEM output operand which takes two cycles, meaning that there needs to be at least one instruction issued after the load before the loaded register can be used (load produces values in the WB stage). With forwarding in mind, the only input operand ZR1Read_EX (SchedReadAdvance) can be set to zero as the input operand starts at the EX stage.

```
let SchedModel = ZR1SchedModel in {
 def ZR1WriteMEM   : SchedWriteRes<[ZR1UnitMEM]>{let Latency= 2;}
 def ZR1WriteALU   : SchedWriteRes<[ZR1UnitALU]>{let Latency= 1;}
 def ZR1WriteFPALU: SchedWriteRes<[ZR1UnitFPALU]>{let Latency= 2;}
 def ZR1WriteFPMUL: SchedWriteRes<[ZR1UnitFPMUL]>{let Latency= 4;}
 def ZR1WriteFPDIV: SchedWriteRes<[ZR1UnitFPDIV]>{let Latency= 8;}

 def ZR1Read_EX    : SchedReadAdvance<0>;
}
```

After the definition of operands, they are associated with the corresponding instructions using the InstRW which associates the output operands and input operands with the instructions as shown on the code below. Other instructions use the units according to the 2.3 section.

```
let SchedModel = ZR1SchedModel in {
  def : InstRW<[ ZR1WriteALU, ZR1Read_EX, ZR1Read_EX],
           (instrs ADD, SUB, XOR, OR, AND, SLL, SRA, SRL)>;
  ...
 }
```

For example, the ADD instruction must go through the ALU EX stage of the pipeline a uses two ZR1Read_EX operands which read registers. To illus-

trate that this model of the generic-v1 pipeline works, consider the following example.

```
define i32 @plus(i32, i32, i32, i32) {
  %5 = mul i32 %0, %0
  %6 = mul i32 %5, %5
  %7 = add i32 %2, %2
  %8 = add i32 %1, %1
  %9 = add i32 %3, %3
  %10 = add i32 %8, %7
  %11 = add i32 %10, %9
  %12 = add i32 %11, %6
  ret i32 %12 }
```

This is a function that takes four integer arguments and does some simple operations using only ADD and MUL instructions. The important part of this code are the first two MUL instructions where the second MUL instruction depends on the first one and would need to be stalled until the first one writes its result. This is a result of the MUL instruction using the multiplication unit which takes 4 cycles to complete. Without the custom scheduler, the DLX assembly code will be in the same order as the input LLVM IR as shown below. Function arguments are stored in registers R17, R18, R19 and R20.

```
...
mult r3, r17, r17
mult r3, r3, r3
add r4, r19, r19
add r5, r18, r18
add r6, r20, r20
add r4, r5, r4
add r4, r4, r6
add r1, r4, r3
...
```

It is easy to check that this code could be rearranged to minimize stalls as the first five instructions after the second MULT instruction do not depend on that MULT instruction a could be moved between the first two MULT instructions as shown on the following code which uses a custom scheduler.

```
...
mult r3, r17, r17
add r4, r19, r19
add r5, r18, r18
add r4, r5, r4
mult r3, r3, r3
add r6, r20, r20
add r4, r4, r6
add r1, r4, r3
...
```

Three ADD instructions were moved between the two MULT instructions to do some work until the first MULT instruction finishes its work. As shown,

the instruction scheduling optimization can save many stalls in programs and thus improve their performance.

5.4.4 DLX Target Scheduling Model - not fully pipelined units

This section describes the changes made to the previous scheduling model to allow not fully pipelined units. The DLX provides two processors with not fully pipeline units as described previously. The following code shows the description of the scheduling model for the generic-v3 processor.

```
let SchedModel = ZR3SchedModel in {
  def ZR3UnitALU    : ProcResource<1> { let BufferSize = 0; }
  def ZR3UnitMEM    : ProcResource<1> { let BufferSize = 0; }
  def ZR3UnitFPALU  : ProcResource<2> { let BufferSize = 0; }
  def ZR3UnitFPMUL  : ProcResource<2> { let BufferSize = 0; }
  def ZR3UnitFPDIV  : ProcResource<2> { let BufferSize = 0; }

  def ZR3WriteMEM: SchedWriteRes<[ZR3UnitMEM]> {let Latency = 2;}
  def ZR3WriteALU: SchedWriteRes<[ZR3UnitALU]> {let Latency = 1;}
  def ZR3WriteFPALU: SchedWriteRes<[ZR3UnitFPALU]>
            { let Latency = 2; let ResourceCycles = [2]; }
  def ZR3WriteFPMUL: SchedWriteRes<[ZR3UnitFPMUL]>
            { let Latency = 4; let ResourceCycles = [4]; }
  def ZR3WriteFPDIV: SchedWriteRes<[ZR3UnitFPDIV]>
            { let Latency = 8; let ResourceCycles = [8]; }
}
```

The ProcResource is now set to the number of stages in a functional unit and the actual pipeline stages use a ResourceCycles information which says that the pipeline stage is occupied for that number of cycles. To illustrate how this works, consider the following LLVM IR example which is a function with four arguments and some ADD and MUL instructions. The important part are again the MUL instructions which in this case do not depend on each other.

```
define i32 @plus(i32, i32, i32, i32) {
  %5 = mul i32 %0, %0
  %6 = mul i32 %2, %2
  %7 = mul i32 %3, %3
  %8 = add i32 %0, %0
  %9 = add i32 %1, %1
  %10 = add i32 %2, %2
  %11 = add i32 %9, %10
  %12 = add i32 %11, %8
  %13 = add i32 %12, %7
  %14 = add i32 %13, %6
  %15 = add i32 %14, %5
  ret i32 %15 }
```

The following code shows the code with no scheduling model which naturally results with the instructions in the same order as the input LLVM IR.

```
...
mult r3, r17, r17
mult r4, r19, r19
mult r5, r20, r20
add r6, r17, r17
add r7, r18, r18
add r8, r19, r19
add r7, r7, r8
add r6, r7, r6
add r5, r6, r5
add r4, r5, r4
add r1, r4, r3
...
```

With the generic-v3 pipeline setting, the third MULT instruction must be stalled because there is no other multiplication unit to handle it but the scheduler can see that some ADD instructions can be reordered between the second and the third MULT instruction to reduce stalls as shown below.

```
...
mult r5, r20, r20          ; scheduled for the generic-v3 processor
mult r4, r19, r19
add r7, r18, r18
add r8, r19, r19
mult r3, r17, r17
add r6, r17, r17
add r7, r7, r8
add r6, r7, r6
add r5, r6, r5
add r4, r5, r4
add r1, r4, r3
...
```

The code below shows the same code scheduled for the generic-v2 processor which only has one multiplication unit. In this case, the scheduler rearranges the ADD instructions not just between the second and the third MULT instruction but also between the first and the second MULT instruction.

```
...
mult r5, r20, r20          ; scheduled for the generic-v2 processor
add r7, r18, r18
add r8, r19, r19
mult r4, r19, r19
add r6, r17, r17
add r7, r7, r8
mult r3, r17, r17
add r6, r7, r6
add r5, r6, r5
add r4, r5, r4
add r1, r4, r3
...
```

5.4.5 Adding a new processor to the DLX target

Adding a new schedule model to support a new processor with a different number of stages and different delays is a simple matter of copying an existing DLX schedule model and changing the values of stages and delays. Adding a new processor only means adding a processor the list of processors and assigning it a scheduling model (it also needs to be added to the list of processors in clang).

5.5 Removing Unnecessary Labels

This optimization does not speed up the computation but can decrease the size of the output assembly code with removing unnecessary labels. It happens in the isBlockOnlyReachableByFallthrough() method in the *ZRAsmPrinter.cpp* file which checks if the basic block is a fallthrough by checking its predecessors and their terminator instructions. To demonstrate how this works, consider the following DLX assembly code with a simple function that only contains a single select instruction.

```
  .text
main:
  ...                       ; function prologue
  lw r3, 4(r29)
  lw r1, 0(r29)
  slt r4, r1, r3
  bnez r4, BB0_2
BB0_1:                      ; this basic block is fallthrough
  ori r1, r3, 0x0
BB0_2:
  ...                       ; function epilogue
  jr r2
```

It is evident that the BB0_1 label is useless as it never gets jumped on and the only way how to reach it is by fallthrough from the previous block if the BNEZ branch instruction is not taken, also the basic block is not marked as global. This means that it can be simply removed as the following code shows.

```
  .text
main:
  ...                       ; function prologue
  lw r3, 4(r29)
  lw r1, 0(r29)
  slt r4, r1, r3
  bnez r4, BB0_2
  ori r1, r3, 0x0
BB0_2:
  ...                       ; function epilogue
  jr r2
```

5.6 Branch Folding

Performance can be improved by eliminating instructions that are never reached or are useless in the sense that the code would do the same without those instructions. The AnalyzeBranch() method in the *ZRInstrInfo.cpp* file is implemented to examine branch instructions at the of basic blocks and to remove unnecessary instructions if needed. It looks at the end of a machine basic blocks for opportunities for improvement. In the simplest case, if a block ends without a branch instruction, then it falls through to the successor block. If a block ends with a single unconditional branch instruction, it is removed if it is a fallthrough. If a block ends with two or more unconditional branch instructions, then every branch after the first one is never reached and is simply removed. If a block ends with both a conditional branch and an ensuing unconditional branch, then the ensuing unconditional branch is removed if it is a fallthrough. If a block ends with indirect branch instruction followed by an unconditional branch instruction that the unconditional branch instruction is also removed. The implementation of branch folding in the LLVM system also requires implementation of the insertBranch() method to insert branch instructions and the implementation of the removeBranch() method to remove branch instructions[33].

Consider the following DLX assembly code generated from a function with a floating-point select instruction. The first block ends with a floating-point conditional branch instruction followed by an unconditional branch instruction.

```
  ...
  gef f2, f1
  bfpt BB0_2
  j BB0_1                    ; this instruction is useless
BB0_1:
  addf f0, f0, f1
BB0_2:
  addi r30, r29, 0x0
...
```

The unconditional branch instruction is not necessary as it is only a fallthrough and can be simply removed. The label was also removed by the previous removing unnecessary labels optimization. This assumes that there was no other jump to the BB0_1 basic block.

```
  ...
  gef f2, f1
  bfpt BB0_2
  addf f0, f0, f1
BB0_2:
  addi r30, r29, 0x0
  ...
```

5.7 Register Rematerialization

Certain values in functions can be recomputed at any point, as the required source operands will always be available for the computation. Such values are called never killed values. During global register allocation pass, if such never killed values cannot be kept in registers and need to be spilled, the register allocator should recognize when it is cheaper to recompute the value i.e. to rematerialize it, rather than to store and reload it from stack[59]. The LLVM can be told which instructions are allowed to rematerialize values by using the isReMaterializable property. The DLX target uses this property for following instructions: LWri, LBri, LHri, LBUri, LHUri, LFri and LDri. To demonstrate how this rematerialization can optimize the DLX assembly code, consider the following LLVM IR example with two functions. The main function takes one argument, call the plus function with that one argument and to the result, it adds the same argument. It is not important what the plus function does.

```
define i32 @main(i32) {
    %2 = call i32 @plus(i32 %0)
    %3 = add i32 %0, %2
    ret i32 %3
}

define i32 @plus(i32) {
    ...
}
```

The following code shows a snippet of the relevant piece of the DLX assembly code generated from the LLVM IR above to show how register rematerialization works. The argument is first loaded to the register R1, is saved to the stack to be preserved during the call of the function plus and after the call it must be loaded back to be added to the result of the plus call.

```
main:
    ...
    lw r1, 0(r29)
    sw -12(r29), r1          ; save R1 to the stack frame
    sw 0(r30), r1            ; argument to the function call plus
    subi r30, r30, 0x4
    jal plus
    lw r3, -12(r29)          ; load R1 back from the stack frame
    add r1, r3, r1
    ...
```

It is easy to notice, that the argument is still in the same place in the stack frame (as an argument of the function main) after the function call and there is no need to save it and restore it to/from the stack frame and can be easily be loaded from the stack frame again (as an argument of the function main) when it is needed to be added to the result of the function call plus as shown on the code below. This saves one store instruction (it only saves the store

instruction as the load instruction still needs to happen just from a different place).

```
main:
    ...
    lw r1, 0(r29)
    sw 0(r30), r1
    subi r30, r30, 0x4
    jal plus
    lw r1, 0(r29)          ; rematerialization of the argument
    add r1, r3, r1
    ...
```

5.8 Passing Values in Registers

Passing function arguments in registers can be a very useful optimization as it can substantially save loads and stores from/to memory. The DLX target reserves four integer registers to pass arguments: R17, R18, R19, and R20[60]. Every other argument is passed on the stack the same way as before. Those registers must be added to the definition of the DLX calling convention, shown below, before assigning values on the stack in the *ZRCallingConv.td* TableGen file as the assigning order is important.

```
// ZR Calling convention.
def CC_ZR : CallingConv<[
 ...
 CCIfNotVarArg<CCIfType<[i32],CCAssignToReg<[R17,R18,R19,R20]>>>,
 CCIfType<[i32, f32], CCAssignToStack<4, 4>>,
 ...
]>;
```

Passing arguments in registers must also be added to the implementation when building the initial SelectionDAG, e.g in the LowerFormalArguments() method and the LowerReturn() method. To illustrate passing arguments in registers, consider the following LLVM IR example where the main function calls the plus function with one integer argument.

```
define i32 @main(i32) {
    %2 = call i32 @plus(i32 %0)
    ret i32 %2
}

define i32 @plus(i32) {
    %2 = add i32 %0, %0
    ret i32 %2
}
```

The following code shows the plus function before the passing values in registers optimization where arguments are placed on the stack.

```
plus:
  ...
  lw r1, 0(r29)
  add r1, r1, r1     ; return value of the function plus
  ...
```

The following code shows the optimized version and it assumes that the main function placed the argument in the register R17 as the calling convention states. This code saves one instruction as the plus function does not have to perform a load instruction to get the argument (this assumes the main function already had the value in the R17 register, for example, did some computation and the result was put directly in the R17 register)

```
plus:
  ...
  add r1, r17, r17   ; return value of the function plus
  ...
```

5.9 Calee-saved Registers

Another register oriented optimization is to use callee-saved registers. Callee-saved registers are registers where the caller does not expect the callee to change them and it is the responsibility of the callee to save/restore those registers if it wants to use them. Callee-saved registers are used to hold long-lived values that should be preserved across calls[61].

The DLX target defines callee-saved registers in the *ZRCallingConv.td* TableGen file where it creates a list which is then used by the getCalleeSavedRegs() method in the *ZRRegisterInfo.cpp* file. The definition of callee-saved registers in the DLX target is below.

```
def CSR : CalleeSavedRegs<(add R19,R20,R21,R22,R23,R24,R25,R26)>;
```

To illustrate this optimization, consider the following LLVM IR example. It is a function that takes two arguments, adds them together and then multiplies them by results of three foo functions. It is not important what those foo functions do.

```
define i32 @main(i32, i32) {
  %3 = add i32 %0, %1
  %4 = call i32 @foo(i32 %0, i32 %1)
  %5 = mul i32 %3, %4
  %6 = call i32 @foo2(i32 %0, i32 %1)
  %7 = mul i32 %5, %6
  %8 = call i32 @foo3(i32 %0, i32 %1)
  %9 = mul i32 %7, %8
  ret i32 %7
}
```

The following code shows the unoptimized version of the main function above in the DLX assembly code. Before each call, the commutated value must be stored to the stack frame because otherwise it might get overridden by the call of foo functions.

```
...
lw r3, 4(r29)          ; get arguments
lw r1, 0(r29)
add r4, r1, r3
sw -20(r29), r4        ; store the value to the stack frame

sw 0(r30), r1          ; call foo
addui r1, r30, 0x4
sw 0(r1), r3
subi r30, r30, 0x4
jal   foo

lw r3, -20(r29)
mult r1, r3, r1
sw -20(r29), r1        ; store the value to the stack frame

lw r1, 0(r29)          ; call foo2
sw 0(r30), r1
addui r1, r30, 0x4
lw r3, 4(r29)
sw 0(r1), r3
subi r30, r30, 0x4
jal   foo2

lw r3, -20(r29)
mult r1, r3, r1
sw -20(r29), r1        ; store the value to the stack frame

lw r1, 0(r29)          ; call foo3
sw 0(r30), r1
addui r1, r30, 0x4
lw r3, 4(r29)
sw 0(r1), r3
subi r30, r30, 0x4
jal   foo3

lw r3, -20(r29)
mult r1, r3, r1
...
```

The next code shows the optimized version with callee-saved registers. The main function starts with saving callee-saved registers to the stack frame which is important because the main function is the caller in this case but it can also be the callee and thus cannot change the value of those registers. The main function can make use of callee-saved registers as it does not have to save them to the stack frame before every function call and can rely on the callee not to change them. This can save many loads and writes to/from memory.

It also loads arguments to callee-saved registers which means that arguments do not have to be read again and again as it happens in the version without callee-saved registers and that also removes a significant number of memory instructions.

```
...
sw -12(r29), r17        ; save callee-saved registers
sw -16(r29), r18
sw -20(r29), r19

lw r17, 4(r29)          ; get arguments
lw r18, 0(r29)

add r19, r18, r17

sw 0(r30), r18          ; call foo
addui r3, r30, 0x4
sw 0(r3), r17
subi r30, r30, 0x4
jal   foo

mult r19, r19, r1

sw 0(r30), r18          ; call foo2
addui r3, r30, 0x4
sw 0(r3), r17
subi r30, r30, 0x4
jal   foo2

mult r19, r19, r1

sw 0(r30), r18          ; call foo3
addui r3, r30, 0x4
sw 0(r3), r17
subi r30, r30, 0x4
jal   foo3

mult r1, r19, r1

lw r19, -20(r29)        ; load callee-saved registers
lw r18, -16(r29)
lw r17, -12(r29)
...
```

This optimized code shows that seven instructions were saved by adding callee-saved registers and the number would grow with more foo functions. This assumes that foo functions took the same number of instructions as they did before this optimization.

5.10 Function Prologue

In the process of creating the DLX backend, the prologue of function was implemented exactly as the DLX calling convention states with no regards to optimizations. The code below shows a function prologue before this optimization.

```
  nop
  nop
  sw 0(r30), r31
  sw -4(r30), r29
  subi r30, r30, 0x4
  addi r29, r30, 0x8
  subi r30, r30, 0x8
  lw r1, 0(r29)
  ...
```

As seen above, those instructions can be reordered to eliminate nop instructions. The new function prologue, which saves two instructions, is shown below.

```
  sw -4(r30), r29
  subi r30, r30, 0x4
  sw 4(r30), r31
  addi r29, r30, 0x8
  subi r30, r30, 0x8
  lw r1, 0(r29)
  ...
```

5.11 Register Allocation

The register allocation can be optimized by rearranging the order of register allocation in the ZRRegisterInfo.td file as shown below.

```
// Register class for integer registers
def GPRegs : RegisterClass<"ZR", [i32], 32,
  (add R3, R4, R5, R6, R7, R8, R12, R13, R14, R15, R16,
    R17, R18, R19, R20 // register used for passing values
    R21, R22, R23, R24, R25, R26, R27, R28 // callee-saved reg.
    R1, R9, R10, R11,   // return values
    R29,  // frame pointer
    R30,  // stack pointer
    R0    // constant 0
    R2    // used for storing return address
)>;
```

This order helps the register allocator to start with general registers and to only use the special ones if all general registers are occupied, possibly saving some spilling to memory. The LLVM system also comes with several register

allocators which can be chosen when compiling code to the DLX assembly, possibly producing a faster code[34]:

- **pbqp** - A Partitioned Boolean Quadratic Programming (PBQP) based register allocator. This allocator works by constructing a PBQP problem representing the register allocation problem under consideration, solving this using a PBQP solver, and mapping the solution back to a register assignment.
- **greedy** - The default allocator. This is a highly tuned implementation of the Basic allocator that incorporates global live range splitting. This allocator works hard to minimize the cost of spill code.
- **fast** - This register allocator is the default for debug builds. It allocates registers on a basic block level, attempting to keep values in registers and reusing registers as appropriate.
- **basic** - This is an incremental approach to register allocation. Live ranges are assigned to registers one at a time in an order that is driven by heuristics. Since code can be rewritten on-the-fly during allocation, this framework allows interesting allocators to be developed as extensions. It is not itself a production register allocator but is a potentially useful stand-alone mode for triaging bugs and as a performance baseline.

Chapter 6

Testing

This chapter describes the process of testing the implemented DLX backend for the LLVM system and testing optimizations that were implemented for it. Testing was performed on the WinDLX Simulator. It also contains the evaluation of optimizations.

6.1 Testing DLX Backend

The first part of testing the LLVM backend consists of 50 C++ files that generate LLVM IR using the C++ LLVM API[62]. Those tests generate 50 test files with LLVM IR used as an input to the llc tool and they consist of basic ALU operations both on integer types and floating types, branches, passing values with all types to functions, returning values of all types from functions, definition and usage of variables, converting between types, calling functions, nested calls, spilling to memory or select instructions. These tests are usually quite simple just to test that single functionality in the DLX backend and they cover pretty much the custom implementation of the DLX backend and they test how the LLVM IR is translated to the DLX assembly code.

The second part of testing consists of 39 C programming language code files that are fed to clang to generate the LLVM IR. The first 27 tests start with testing the basic functionality but in a little more complicated way. They continue by testing the constructs of the C language such as loops, while loops, dynamic size arrays, pointers or structs. At the end they test mathematical functions like power, gcd, factorial using both iteration and recursion, sqrt, listing first n prime numbers, computing the Euler totient function which returns a number of prime numbers lower than the input or dijkstra algorithm for finding the shortest path in a weighted graph. The last 12 tests contain a lot of computation on two-dimensional arrays and they are mainly used later for evaluating optimizations, especially scheduling. The job of these tests is to show that the LLVM system with the DLX backend can compile C codes to the DLX assembly code. This would probably deserve a little more testing

but without the implementation of standard C functions (malloc, free) and with the need of manually running the tests, this is very limited.

To actually run these tests, some environment must be set up first to initialize the stack pointer, initial stack frame, frame pointer and to store the proper arguments to memory in the right places. The following code shows the setup for a main function with two integer arguments.

```
  .data
  .align 2
Name:
  .asciiz "test"

  .align 2
Arg1:
  .asciiz "15"

  .align 2
Arg2:
  .asciiz "20"

  .align 2
Array:
  .space 12

  .text
start:
  lhi r2, (Name >> 16)
  addui r2, r2, (Name & 0x0000ffff)
  lhi r3, (Arg1 >> 16)
  addui r3, r3, (Arg1 & 0x0000ffff)
  lhi r4, (Arg2 >> 16)
  addui r4, r4, (Arg2 & 0x0000ffff)
  lhi r5, (Array >> 16)
  addui r5, r5, (Array & 0x0000ffff)

  sw 0(r5), r2            ; store arguments to memory
  sw 4(r5), r3
  sw 8(r5), r4
  addi r30, r30, 0x7ffc   ; stack pointer
  addi r29, r29, 0x7ffc   ; frame pointer

  addi r18, r18, 3        ; prepare arugments for the function call
  add r19, r19, r5
  sw 0(r30), r19
  sw -4(r30), r18
  subi r30, r30, 8

  jal main                ; call main function

end:
  trap 0                  ; terminate program
```

This environment places the stack to the highest position (for testing, the

WinDLX memory was set to 0x8000). It places the arguments to memory in a C programming language manner (argc argument, argv argument consisting of the name of the program and other arguments as strings) and then jumps to the main function. After returning from the main function, it only executes a trap 0 instruction which indicates the end of the program.

To simplify testing, several functions were created to allow printing values in the WinDLX simulator such as printI (print one integer), printUI (print one unsigned integer), pringF (print float) and printD (print double). Those are regular functions which can be called from a C programming language code (they only need to be declared, for example, to use the printI function: "void printI(int i);"). They all use trap 5 to print the value which uses R14 register and therefore this register must be saved to the stack frame along with any other registers that these functions use, to not change values for the caller. The code below shows the code of the printI function.

```
  .data
  .align 2
FormatStrOutput:    .asciiz       "Output i:%d\n"

  .align 2
ParStrOutput:
  .word  FormatStrOutput
  .space 4

  .text
printI:
  nop
  nop
  sw 0(r30), r31
  sw -4(r30), r29
  sw -8(r30), r14
  sw -12(r30), r3
  subi r30, r30, 0xc
  addi r29, r30, 0x10
  subi r30, r30, 0x10

  lw r3, 0(r29)          ; load the value to print

  lhi r14, ParStrOutput >> 16
  addui r14, r14, ParStrOutput & 0x0000ffff

  sw 4(r14), r3          ; store the value to the proper place

  trap 5                 ; print the integer value

  addi r30, r29, 0x0
  lw r2, -4(r29)
  lw r14, -12(r29)
  lw r3, -16(r29)
  lw r29, -8(r29)
  jr r2
```

To automatize these tests, a script was created which acts as a very simply linker. The job of this script is to generate files which can be run on the WinDLX simulator. It needs to add the proper environment setup code and add printing functions if the test requires it.

6.2 Testing and Evaluating Optimizations

The DLX target with all implemented optimizations was tested using the same tests to ensure that the produced code is still valid and correct. In the process of generating the DLX target code of these tests, the opt tool was used to run O2 level of optimizations to optimize the LLVM IR. These tests were all run for all processors and their scheduling models defined in the DLX target. The environment setup codes had to be changed a little to use the optimized prologue and passing arguments in registers. The printing functions also had to be changed to load arguments from registers and not from the stack frame.

After making sure optimizations did not change the validity or correctness of the code, optimizations were evaluated to show if optimizations can make the DLX assembly code run faster as opposed to the code without optimizations. The evaluation was done using the 39 C programming languages codes only as the previous LLVM IR tests are too simple to offer any place for optimizations. Every test was measured with a number of clock cycles it takes and a number of stalls. As shown on the table 6.1 and 6.2, these tests were run without any optimizations, with optimizations (meaning O2 level of optimization and target specific optimizations) and with optimizations and scheduling at the same time. Columns of these tables are described below:

- **No opti** - no optimizations, generic-v1 setting of the pipeline.
- **Opti** - optimizations without scheduling, generic-v1 setting of the pipeline.
- **V1 sched** - optimizations with scheduling, generic-v1 setting of the pipeline.
- **V2** - optimizations without scheduling, generic-v2 setting of the pipeline.
- **V2 sched** - optimizations with scheduling, generic-v2 setting of the pipeline.
- **V3** - optimizations without scheduling, generic-v3 setting of the pipeline.
- **V3 sched** - optimizations with scheduling, generic-v3 setting of the pipeline.

Table 6.1: Evaluation of the generic-v1 processor

Test Case	**No opti**	**Opti**	**V1**
000_empty_main	35 [5]	28 [5]	28 [5]
001_atoi_function	259 [68]	134 [36]	129 [31]
002_plus_function	289 [73]	157 [41]	152 [36]
003_nested_call	389 [87]	158 [41]	153 [36]
004_integers	478 [104]	231 [56]	226 [51]
005_floats	674 [157]	315 [93]	299 [75]
006_branches	314 [87]	168 [49]	163 [44]
007_switch	204 [49]	120 [30]	115 [25]
008_for	852 [237]	514 [129]	510 [125]
009_while	496 [137]	280 [74]	275 [69]
010_array	568 [147]	51 [10]	51 [10]
011_array_dynamic_size	904 [308]	330 [73]	311 [54]
012_pointers	369 [90]	201 [48]	196 [42]
013_pointers2	320 [78]	157 [41]	152 [36]
014_infinite_loop	N/A	N/A	N/A
015_struct	324 [80]	163 [47]	155 [39]
016_struct_pointers	317 [78]	163 [47]	155 [39]
017_global_variables	150 [28]	112 [22]	112 [22]
018_unsigned	298 [75]	160 [41]	155 [36]
019_gcd	384 [110]	194 [61]	189 [56]
020_power	505 [144]	217 [55]	209 [47]
021_sqrt	633 [204]	259 [98]	248 [85]
022_prime_numbers	17386 [8390]	9229 [5438]	9225 [5434]
023_factorial_iteration	291 [78]	138 [32]	131 [28]
024_factorial_recursive	405 [104]	142 [32]	138 [33]
025_euler_totient	2644 [1042]	1045 [532]	1041 [529]
026_graph_dijkstra	7267 [2109]	N/A	N/A
s000_data	14994 [3981]	1449 [44]	1445 [40]
s001_data	16195 [4381]	1606 [62]	1597 [52]
s002_data	28925 [7764]	2730 [63]	2726 [59]
s003_data	30125 [8164]	2815 [80]	2807 [72]
s004_data	38926 [10564]	4297 [1182]	3647 [512]
s005_data	36236 [11364]	6359 [1471]	6350 [1462]
s006_data	35226 [9784]	7535 [1243]	7131 [839]
s007_data	22366 [7124]	4931 [1663]	3627 [59]
s008_data	23166 [7704]	5251 [1763]	3787 [59]
s009_data	24445 [8604]	6071 [2303]	4207 [99]
s010_data	30845 [12184]	13014 [6143]	10290 [2979]
s011_data	24485 [7944]	6155 [1783]	4851 [119]

Table 6.2: Evaluation of the generic-v2 and generic-v3 processor

Test Case	V2	V2 sched	V3	V3 sched
000_empty_main	28 [5]	28 [5]	28 [5]	28 [51]
001_atoi_function	134 [36]	129 [31]	134 [36]	129 [31]
002_plus_function	157 [41]	152 [36]	157 [41]	152 [36]
003_nested_call	158 [41]	153 [36]	158 [14]	153 [36]
004_integers	231 [56]	226 [51]	231 [56]	226 [51]
005_floats	316 [94]	300 [76]	315 [93]	299 [75]
006_branches	168 [49]	163 [44]	168 [49]	163 [44]
007_switch	120 [30]	115 [25]	120 [30]	115 [25]
008_for	514 [129]	510 [125]	514 [129]	510 [125]
009_while	280 [74]	275 [69]	280 [74]	275 [69]
010_array	51 [10]	51 [10]	51 [10]	51 [10]
011_array_dynamic_size	330 [73]	311 [54]	330 [73]	311 [54]
012_pointers	201 [48]	196 [43]	201 [48]	196 [43]
013_pointers2	157 [41]	152 [36]	157 [41]	152 [36]
014_infinite_loop	N/A	N/A	N/A	N/A
015_struct	163 [47]	155 [39]	163 [47]	155 [39]
016_struct_pointers	163 [47]	155 [39]	163 [47]	155 [39]
017_global_variables	112 [22]	112 [22]	112 [22]	112 [22]
018_unsigned	160 [41]	155 [36]	160 [41]	155 [36]
019_gcd	194 [61]	189 [56]	194 [61]	189 [56]
020_power	219 [58]	210 [49]	217 [55]	209 [47]
021_sqrt	259 [98]	248 [85]	259 [98]	248 [85]
022_prime_numbers	9229 [5438]	9225 [5434]	9229 [5438]	9225 [5434]
023_factorial_iteration	138 [32]	131 [28]	138 [32]	131 [28]
024_factorial_recursive	142 [32]	138 [33]	142 [32]	138 [33]
025_euler_totient	1045 [532]	1041 [529]	1045 [532]	1041 [529]
026_graph_dijkstra	N/A	N/A	N/A	N/A
s000_data	1449 [44]	1445 [40]	1449 [44]	1445 [40]
s001_data	1661 [121]	1645 [105]	1622 [80]	1612 [69]
s002_data	2730 [63]	2726 [59]	2730 [63]	2726 [59]
s003_data	2815 [80]	2807 [72]	2815 [80]	2807 [72]
s004_data	4297 [1182]	3647 [512]	4297 [1182]	3647 [512]
s005_data	6359 [1471]	6350 [1462]	6359 [1471]	6350 [1462]
s006_data	7535 [1243]	7131 [829]	7535 [1243]	7131 [839]
s007_data	5051 [1783]	4167 [859]	4931 [1663]	3847 [519]
s008_data	5451 [2023]	4227 [739]	5251 [1763]	4087 [439]
s009_data	6271 [2563]	5007[1179]	6071 [2303]	4467 [619]
s010_data	13194 [6223]	10470[3219]	13014 [6143]	10310 [3059]
s011_data	6275 [1903]	5131 [519]	6155 [1783]	4891 [159]

The results of the 014 test are not available for obvious reasons as the loop runs forever. The optimized versions for the 026 test are not available as LLVM optimizes the static definition of the two-dimensional array representing the graph with a memset operation which is not implemented for the DLX target.

The first 6.1 table measures how optimizations sped up the assembly code. For this comparison, the generic-v1 processor was used and it shows that the optimized code can run significantly faster in comparison with the unoptimized versions. For example, the computation of the first n prime numbers takes roughly half the clock cycles it takes without optimizations and the computation of the Euler's totient function takes less than the half of clock cycles then the unoptimized variant. The third column then adds scheduling and also shows that scheduling can further optimize the code as every test case was sped up at least a little. The biggest speedup from scheduling happened in last 12 tests because they have the most room for scheduling as they contain many alu operations and were designed that way to test that the implemented schedulers work.

The second table 6.2 is focused on scheduling the generic-v2 and generic-v3 processors. It compares the scheduled code with the unscheduled code for those two processors and shows that the scheduled variants are indeed faster. The overall scheduling results are as expected, with the comparison, the generic-v1 processor produces the fastest code as it is a fully pipelined processor and the generic-v2 is a little slower than the generic-v3 as it has a lower number of stages in functional units. The good thing is that the scheduler works for all processors as the code is always faster with scheduling that the code without scheduling.

Conclusion

This thesis analyzed the DLX architecture and its pipeline and based on that it created a new working backend for the DLX architecture which can be used in the LLVM system. This allows numerous programming languages, meaning languages that have a frontend implemented for the LLVM system, to be compiled into the DLX assembly code. It also gives a guideline on how to create a new LLVM backend for a simple RISC architecture.

It analyzed how optimizations work in the LLVM system and analyzed different places where optimizations can occur. It implemented several target specific optimizations like instruction selection or instruction scheduling and few others to speed up the DLX assembly code. It created three different types of processors with a different number of stages in functional units with different scheduling models to schedule the DLX assembly according to the used processor.

At the end, the backend was successfully tested using LLVM IR inputs to ensure that the backend produces a valid DLX assembly code. The backed was also successfully tested using C code languages inputs to show that the LLVM system now can compile C programming language codes to the DLX assembly code. Optimizations were first tested using the same tests as the backend to ensure that optimizations did not change the meaning of the code. The evaluation proved that the implemented optimizations along with LLVM optimizations can speed up the DLX assembly code.

What to improve

The DLX backend could be extended to compile into object files but that would require a new DLX simulator which would allow such a thing. A new DLX simulator would be handy either way as it could support dots in labels, support .p2align directive, support .comm directive and would implement directives for storing 16-bit and 64-bit values to memory. Those changes would remove the custom changes made in LLVM source codes which would make

for a much cleaner solution. Some sort of implementation of standard C functions would be very helpful, most importantly, functions used to dynamically allocate memory such as malloc and free would really widen the group of C programming language codes to be compiled into the DLX assembly code. It would also be nice if the number of stages in units and delays could be set using arguments to the llc tool. At this moment, the LLVM system allows passing target specific arguments to the llc tool but does not fully support passing those arguments to the TableGen files which are used for scheduling models. Lastly, as there is never too many optimizations in a compiler, additional optimizations could be implemented for the DLX target.

Bibliography

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 2nd ed. Morgan Kaufmann Publishers. ISBN: 978-1558603295.

[2] Kashay Pingali. *The DLX Instruction Set Architecture*. URL: http://www.cs.utexas.edu/~pingali/CS378/2015sp/lectures/DLX.pdf (visited on 12/28/2018).

[3] Central Connecticut State University. *Big Endian and Little Endian*. URL: https://chortle.ccsu.edu/AssemblyTutorial/Chapter-15/ass15_3.html (visited on 12/28/2018).

[4] Gurpur M. Prabhu. *COMPUTER ARCHITECTURE TUTORIAL*. URL: http://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/addressAlign.html (visited on 12/28/2018).

[5] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[6] Central Connecticut State University. *Memory layout*. URL: http://chortle.ccsu.edu/assemblytutorial/Chapter-10/ass10_3.html (visited on 12/28/2018).

[7] Paul Krzyzanowski. *Stack frames*. URL: https://www.cs.rutgers.edu/~pxk/419/notes/frames.html (visited on 12/28/2018).

[8] Gert Lanckriet. *Introduction to Computer Engineering - Instructions: Assembly Language*. URL: eceweb.ucsd.edu/~gert/ece30/CN2.pdf (visited on 12/28/2018).

[9] Ethan L. Miller. *The DLX Operating System (DLXOS)*. URL: https://www2.ucsc.edu/courses/cmps111-elm/dlx/dlxos (visited on 12/28/2018).

[10] Javier Echaiz. *WinDLX Tutorial*. URL: https://cs.uns.edu.ar/~jechaiz/arquitectura/windlx/wdlxtut.pdf (visited on 12/28/2018).

[11] David Viner. *DLX Simulator*. URL: https://www.davidviner.com/dlx.html (visited on 12/28/2018).

[12] S. Metzlaff, A. Vogelgsgang, and N. Krezic-Luger. *openDLX v1.0 - A DLX/MIPS processor simulator*. URL: github.com/smetzlaff/openDLX (visited on 12/28/2018).

[13] Carl Burch. *DLXsim - Simulator and debugger for DLX assembly programs*. URL: http://www.cs.cmu.edu/~cburch/pgss99/work/self/proj2man.pdf (visited on 12/28/2018).

[14] University of Salzburg. *DLX-Simulator*. URL: http://lv.cosy.sbg.ac.at/digitale/dlxwsim/ (visited on 12/28/2018).

[15] Prof. Herbert Grunünbacher. *WinDLX documentation: Traps - the System Interface*. January 1992.

[16] LLVM Project. *The LLVM Compiler Infrastructure*. URL: https://llvm.org (visited on 12/28/2018).

[17] Chris Lattner. *The Architecture of Open Source Applications: LLVM*. URL: http://www.aosabook.org/en/llvm.html (visited on 12/28/2018).

[18] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Addison Wesley. ISBN: 978-0321486813.

[19] LLVM Project. *Getting Started with the LLVM System*. URL: https://llvm.org/docs/GettingStarted.html (visited on 12/28/2018).

[20] LLVM Project. *Mapping High Level Constructs to LLVM IR Documentation*. URL: https://media.readthedocs.org/pdf/mapping-high-level-constructs-to-llvm-ir/latest/mapping-high-level-constructs-to-llvm-ir.pdf (visited on 12/28/2018).

[21] LLVM Project. *LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html (visited on 12/28/2018).

[22] Frances E. Allen. "Contorl Flow Analysis". In: *ACM SIGPLAN Notices - Proceedings of a symposium on Compiler optimization Volume 5 Issue 7* (July 1970).

[23] Ron Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems Volume 13 Issue 4* (Oct. 1991).

[24] Wei Le. *Control Flow Analysis*. URL: http://web.cs.iastate.edu/~weile/cs513x/4.ControlFlowAnalysis.pdf (visited on 12/28/2018).

[25] LLVM Project. *The Often Misunderstood GEP Instruction*. URL: https://llvm.org/docs/GetElementPtr.html (visited on 12/28/2018).

[26] LLVM Project. *LLVM Command Guide*. URL: https://llvm.org/docs/CommandGuide/ (visited on 12/28/2018).

[27] LLVM Project. *Clang: a C language family frontend for LLVM*. URL: http://clang.llvm.org/index.html (visited on 12/28/2018).

[28] LLVM Project. *Clang - Features and Goals*. URL: https://clang.llvm.org/features.html (visited on 12/28/2018).

[29] LLVM Project. *LLVM's Analysis and Transform Passes*. URL: https://llvm.org/docs/Passes.html (visited on 12/28/2018).

[30] Google. *Lanai Instruction Set*. URL: http://g.co/lanai/isa (visited on 12/28/2018).

[31] Charles Price. *MIPS IV Instruction Set*. URL: https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf (visited on 12/28/2018).

[32] Oracle Corporation. *Oracle SPARC Architecture 2015*. URL: https://community.oracle.com/servlet/JiveServlet/downloadBody/1005258-102-3-148654/OracleSparcArchitecture2015.pdf (visited on 12/28/2018).

[33] LLVM Project. *Writing an LLVM Backend*. URL: https://llvm.org/docs/WritingAnLLVMBackend.html (visited on 12/28/2018).

[34] LLVM Project. *The LLVM Target-Independent Code Generator*. URL: https://llvm.org/docs/CodeGenerator.html (visited on 12/28/2018).

[35] LLVM Project. *TableGen*. URL: https://llvm.org/docs/TableGen/index.html (visited on 12/28/2018).

[36] Lancester University. *Variable-length argument lists*. URL: http://www.lancaster.ac.uk/~simpsons/char/lang/valist (visited on 12/28/2018).

[37] LLVM Project. *Target.td - Target Independent TableGen interface*. URL: https://github.com/llvm-mirror/llvm/blob/master/include/llvm/Target/Target.td (visited on 12/28/2018).

[38] LLVM Project. *llvm::TargetLowering Class Reference 8.0.0*. URL: http://llvm.org/doxygen/classllvm_1_1TargetLowering.html (visited on 12/28/2018).

[39] Todd C. Mowry. *Register Allocation & Spilling*. URL: http://www.cs.cmu.edu/afs/cs/academic/class/15745-s13/public/lectures/L15-Register-Allocation.pdf (visited on 12/28/2018).

[40] Tool Interface Standards. *Executable and Linkable Format (ELF), Ver 1.1*. URL: http://www.skyfree.org/linux/references/ELF_Format.pdf (visited on 12/28/2018).

[41] Xinuos Inc. *ELF Header*. URL: http://www.uxsglobal.com/developers/gabi/latest/ch4.eheader.html (visited on 12/28/2018).

[42] The Clang Team. *Driver Design & Internals*. URL: https://clang.llvm.org/docs/DriverInternals.html (visited on 12/28/2018).

[43] Dan Gohman. *[MC] Use .p2align instead of .align*. URL: https://reviews.llvm.org/D16549 (visited on 12/28/2018).

[44] R. Sekar. *CSE 304 Compiler Design - Optimization and Program analysis*. URL: http://seclab.cs.sunysb.edu/sekar/cse304/dataflow.pdf (visited on 12/28/2018).

[45] Johan Engelen. *Link Time Optimization (LTO), C++/D cross-language optimization*. URL: http://johanengelen.github.io/ldc/2016/11/10/Link-Time-Optimization-LDC.html (visited on 12/28/2018).

[46] Jan Janoušek. *Optimalizace: úvod, konverze 3AC a AST, lokální optimalizace v rámci základního bloku*. URL: https://courses.fit.cvut.cz/MI-GEN/media/lectures/08/gen-lecture8.pdf (visited on 12/28/2018).

[47] Annie Cherkaev. *LLVM Optimizations*. URL: https://anniecherkaev.com/2016/11/10/LLVM-optimizations.html (visited on 12/28/2018).

[48] Shahar Mike. *Return Value Optimization*. URL: https://shaharmike.com/cpp/rvo/ (visited on 12/28/2018).

[49] CppDepend. *Quick overview of how Clang works internally*. URL: https://cppdepend.com/blog/?p=321 (visited on 12/28/2018).

[50] FreeBSD. *FreeBSD Manual Pages - Clang(1)*. URL: http://www.freebsd.org/cgi/man.cgi?query=clang++&sektion=1&manpath=FreeBSD+9.0-RELEASE (visited on 12/28/2018).

[51] John Dallman, David Tweed, and Renato Golin Linaro. *Meaning of LLVM optimization levels*. URL: http://clang-developers.42468.n3.nabble.com/Meaning-of-LLVM-optimization-levels-td4032493.html (visited on 12/28/2018).

[52] David R. Koes and Seth C. Goldstein. *Near-Optimal Instruction Selection on DAGs*. URL: https://llvm.org/pubs/2008-CGO-DagISel.pdf (visited on 12/28/2018).

[53] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. ISBN: 1-55860-320-4.

[54] Abid M. Malik et al. *Learning Heuristics for Basic Block Instruction Scheduling*. URL: https://pdfs.semanticscholar.org/4650/5990d3453ebc99ab11e81f2632fc0810b83f.pdf (visited on 12/28/2018).

[55] Dave Estes. *SchedMachineModel: Adding and Optimizing a Subtarget*. URL: https://llvm.org/devmtg/2014-10/Slides/Estes-MISchedulerTutorial.pdf (visited on 12/28/2018).

[56] Susan Eggers. *Introduction to Out-of-Order Execution*. URL: https://courses.cs.washington.edu/courses/csep548/06au/lectures/introOOO.pdf (visited on 12/28/2018).

[57] Javed Absar and Florian Hahn. *Writing Great Machine Schedulers*. URL: http://llvm.org/devmtg/2017-10/#tutorial1 (visited on 12/28/2018).

[58] LLVM Project. *TargetSchedule.td - Target Independent Scheduling*. URL: https://github.com/llvm-mirror/llvm/blob/master/include/llvm/Target/TargetSchedule.td (visited on 12/28/2018).

[59] Mukta Punjani. *Register Rematerialization In GCC*. URL: ftp://gcc.gnu.org/pub/gcc/summit/2004/Register%20Rematerialization.pdf (visited on 12/28/2018).

[60] Virgil Bistriceanu. *Register Usage & Procedures*. URL: http://www.cs.iit.edu/~virgil/cs402/Labs/Lab4.pdf (visited on 12/28/2018).

[61] Michelle Strout. *Compiler Construction - Register Allocation*. URL: http://www.cs.colostate.edu/~mstrout/CS553Fall06/slides/lecture05-regalloc2.pdf (visited on 12/28/2018).

[62] LLVM System. *LLVM Programmer's Manual*. URL: http://llvm.org/docs/ProgrammersManual.html (visited on 12/28/2018).

[63] LLVM System. *LLVM Building LLVM with CMake*. URL: https://llvm.org/docs/CMake.html (visited on 12/28/2018).

Appendix A

Install

This appendix describes how to set up and use the LLVM backend for DLX and how to use LLVM optimizations, different DLX processors and how to run tests.

A.1 Build DLX Backend

In order to build this project, these following prerequisites are needed:

- **CMake** >= 3.4.3 - CMake is an open-source, cross-platform family of tools designed to build, test and package software.
- **Ninja** (tested on version 1.7.2) - Ninja is a small build system with a focus on speed.
- **GNU Make** (tested on version 3) - Makefile/build processor.
- **GCC** >= 4.8.0 - GNU C/C++ compiler.

The source code folder contains two subfolders:

- **llvm** - Contains all the LLVM source codes with the DLX target with all optimizations and with all DLX processors.
- **llvm_un** - Contains all LLVM source codes with the DLX target without any target optimizations and does not use any DLX processors.

LLVM source codes are of this versions:

- **LLVM** - Version 9.0 Revision: 341764
- **Clang** - Version 8.0 Revision 341764

To build the LLVM system with the DLX target, these following steps must be performed:

1. Make a new directory where the llvm will live.

2. Run the two commands printed below to either compile the debug version or the release version. The llvm_source can be either the version with optimizations or the version without optimizations.

Release version:

```
cmake -G "Ninja" -DLLVM_OPTIMIZED_TABLEGEN=1 -DLLVM_ENABLE_RTTI=1
    -DCMAKE_BUILD_TYPE="Release" -DLLVM_TARGETS_TO_BUILD="ZR"
    -DLLVM_ENABLE_ASSERTIONS=1  llvm_source

ninja
```

Debug version:

```
cmake -G "Ninja" -DLLVM_OPTIMIZED_TABLEGEN=1 -DLLVM_ENABLE_RTTI=1
    -DLLVM_TARGETS_TO_BUILD="ZR" -DCMAKE_BUILD_TYPE="Debug"
    --enable-debug-symbols --with-oprofile llvm_source

ninja
```

Options[63]:

- DLLVM_OPTIMIZED_TABLEGEN:BOOL - If enabled and building a debug or asserts build the CMake build system will generate a Release build tree to build a fully optimized tablegen for use during the build. Enabling this option can significantly speed up build times especially when building LLVM in Debug configurations.

- DLLVM_ENABLE_RTTI:BOOL - Build LLVM with run-time information.

- CMAKE_BUILD_TYPE:STRING - Possible values are Release, Debug, RelWithDebInfo and MinSizeRel.

- LLVM_USE_LINKER:STRING - Build LLVM using a specified linker. The gold linker can save substantial amount of memory especially when building debug version.

- DLLVM_TARGETS_TO_BUILD - Specifies which targets should be built.

- DLLVM_ENABLE_ASSERTIONS:BOOL - Enables code assertions. This also allows the release version produce SelectionDAG graphs.

A.2 Use DLX Backend

The DLX backend is integrated into LLVM tools such as clang, opt, or llc tool. The following code gives some examples of how to run the DLX target with these tools. These tools are created inside the bin folder when the LLVM system is built.

```
CLANG_PATH -O2 -S -emit-llvm --target=zr -mcpu=generic-v1 src -o
    out

OPT_PATH -O2 -S -march=zr -mcpu=generic-v1 src -o out

LLC_PATH -march=zr -mcpu=generic-v1 -asm-verbose=false src -o out
```

The first command takes a C programming language code and compiles it into the LLVM IR. The second tool takes the LLVM IR and optimizes it and the llc tool compiles the LLVM IR into the DLX assembly code for the generic-v1 processor. The asm-verbose argument at the llc tool might be necessary as, for example, the WinDLX simulator cannot handle comments after directives.

A.3 Tests

Tests can be generated using two bash scripts described below. These scripts generate the final DLX assembly codes which can be run on a DLX simulator. Both scripts must be run from the test root directory and the paths to the tools must be specified by a full path.

- **runS.sh** - Create DLX assembly codes without any optimizations (this test should use the DLX target in the llvm_un folder). This test takes two arguments: a path to clang (-c) and a path to llc (-l). Creates a new inputS folder that contains the tests.
- **runO.sh** - Create DLX assembly codes with optimizations (this test should use the DLX target in the llvm folder). It uses the -O2 level for optimizations. It uses the same arguments but also needs a path to opt tool (-o) and a type of processor (-p). Creates a new inputO folder that contains the tests.

The examples of using both scripts are shown below.

```
./runO.sh -o opt_path -c clang_path -l llc_path -p processor

./runS.sh -c clang_path -l llc_path
```

Appendix B

Acronyms

ALU Arithmetic Logical Unit

CFG Control Flow Graph

DAG Directed Acyclic Graph

DLX Deluxe

FP Floating-point

FSPR Floating-point Status Register

PC Program Counter

SSA Single Static Assignment

Appendix C

Contents of Enclosed Data Storage

- `readme.txt` ..... brief description of contents of enclosed data storage
- `src`
  - `llvm_src.tar.gz` ..... LLVM source codes with the unoptimized DLX backend and the DLX optimized backend
  - `tests` ..... source codes of tests
  - `thesis` ..... source codes of this thesis in LaTeX format
- `text` ..... thesis
  - `DP_Bures_Michal_2019.pdf` ..... thesis in the PDF format