

Chapter 13

A simplified DLX

In this chapter we describe a simple microprocessor called the simplified DLX.

13.1 Why use abstractions?

The term architecture according to the Collins Dictionary means *the art of planning, designing, and constructing buildings*. Computer architecture refers to computers rather than buildings. Computers are rather complicated. To simplify things, people focus on certain aspects of computers and ignore other aspects. For example, the hardware designer ignores questions such as: which programs will be executed by the computer? The programmer often does not even know exactly which type of computer will be executing the program. It is the task of the architect to be aware of different aspects so that the designed system meets certain price and performance goals.

To facilitate focusing on certain aspects, abstractions are used. Several abstractions are used in computer systems. For example, the C programmer uses the abstraction of a computer that runs C programs, owns a private memory, and has access to various peripheral devices (such as a printer, a monitor, a keyboard, etc.). Supporting this abstraction requires software tools (compiler, linker, loader), an operating system (to coordinate between several programs running in the same time), and hardware (that actually executes programs, but not in C).

In this chapter, our starting point is actually a midpoint. We specify a microprocessor from the programmer's point of view. However, this is not a C programmer or a programmer that is programming in a high level language. Instead, this is a programmer programming in machine language. Since it is not common anymore for people to program in machine language, the machine language programmer is actually a program!

Programs in machine language are output by a program called an assembler. The input of an assembler is a program in assembly language. Most assembly programs are also written by programs called compilers. Compilers are input a program in a high level and output assembly programs.

This chain of translations starting from a C program and ending with a machine language program has several advantages:

1. The microprocessor executes programs written in a very simple language (machine language). This facilitates the design of the microprocessor.
2. The C programmer need not think about the actual platform that executes the program. Hence the same program can be compiled and assembled so that it can be executed on different architectures.
3. Every stage of the translation works in a certain abstraction. The amount of detail increases as one descends to lower level abstractions. In each translation step, decisions can be made that are optimal with respect to the current abstraction.

One can see that all these advantages have to do with good engineering practice. Namely, a task is partitioned in smaller subtasks that are simpler and easier. Clear and precise boundaries between the subtasks guarantee correctness when the subtasks are “glued” together.

Question 13.1 *Explain why it is not common anymore for people to program in assembly or machine code. Consider issues such as: cost of programming in a high level language compared to assembly or machine code, ease of debugging programs, protections provided by high level programming, and length and efficiency of final machine code program.*

13.2 Instruction set architecture

The term instruction set architecture refers to the specification of the computer from the point of view of the machine language programmer. This abstraction has the following components:

- The objects that are manipulated. The objects are either words stored in registers or in memory.
- The instructions (or commands) that tell the computer what to do to the objects.

13.2.1 Architectural Registers and Memory

Both the registers and the memory store words. A word is a 32-bit string. The memory is often called also the main memory.

The memory is used to store both the program itself (i.e., instructions) and the data (i.e., constant and variables used by the program). We regard the memory as an array $M[0 : 2^{32} - 1]$ of words. Each element $M[i]$ in the array holds one word. The memory is organized like a Random Access Memory (RAM). This means that the processor can access the memory in one of two ways:

- Read or load $M[i]$. Request to copy the contents of $M[i]$ to a register called MDR .
- Write or store in $M[i]$. Request to store the contents of a register called MDR in $M[i]$.

The architectural registers of the simplified DLX are all 32 bits wide and listed below.

- 32 General Purpose Registers (GPRs): R0 to R31. Loosely speaking, the general purpose registers are the objects that the program directly manipulates. For example, a high level instructions $x := y + z$ is implemented by storing the sum of two registers in a third register; say $R1 \leftarrow R2 + R3$.
- Program Counter (PC). The PC stores the address (i.e., location in memory) of the instruction that is currently being executed.
- Instruction Register (IR). The IR stores the current instruction.
- Special Registers: MAR, MDR. These registers are used for specifying the interface between the microprocessor and the memory.

Question 13.2 *Parts of the main memory in many computers are read-only memory and even nonvolatile. Read-only means that the contents cannot be changed. Nonvolatile means that the contents are kept even when power is turned off. Can you explain why?*

13.2.2 Instruction Set

The machine language of a processor is often called an instruction set. In general, a machine language has very few rules and a very simple syntax. In the case of the simplified DLX, every sequence of instructions constitutes a legal program (is this the case in C or in Java?). This explains why the machine language is referred to simply as a set of instructions.

Instruction formats. Every instruction in the instruction set of the simplified DLX is represented by a single word. There are two instruction formats: I-type and R-type. The partitioning of each format into fields is depicted in Figure 13.1. The opcode field encodes the instruction (e.g., load, store, add, jump). The $RS1$, $RS2$, RD fields encode indexes of general purpose registers. The immediate field encodes a constant. The function field (in an R-type instruction format) is used to encode the instruction.

List of instructions. We list below the instruction set of the simplified DLX. In this list, imm denotes the immediate field in an I-Type instruction and $sext(imm)$ denotes a two's complement sign extension of imm to 32 bits. The semantics of each instruction are informally abbreviated and are formally explained after each group of instructions.

Note that every instruction (except for jump instructions and halt), has the side effect of increasing the PC. Namely, apart from doing whatever the instructions says, the

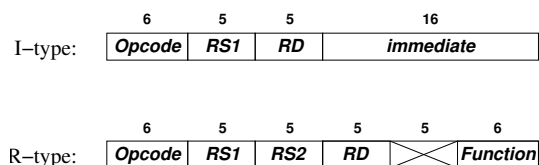


Figure 13.1: Instruction formats of the simplified DLX.

microprocessor also performs the operation:

$$\langle PC \rangle \leftarrow \text{mod}(\langle PC \rangle + 1, 2^{32}). \quad (13.1)$$

Equation 13.1 means the following: (i) Add one to the binary number represented by the PC. (ii) The sum is computed modulo 2^{32} , namely, if the sum equals 2^{32} , then replace the sum by zero. (iii) Store the sum in binary representation in the PC.

Load/Store Instructions (I-type). Load and store instructions deal with copying words between the memory and the GPRs.

Load/Store				Semantics
lw	RD	RS1	imm	$RD := M(\text{sext}(\text{imm}) + RS1)$
sw	RD	RS1	imm	$M(\text{sext}(\text{imm}) + RS1) := RD$

The precise semantics of load and store instructions are rather complicated. We first define the effective address; informally, the effective address is the index of the memory word that is accessed in a load or store instruction.

Definition 13.1 *The effective address in a load or store instruction is defined as follows. Let $j = \langle RS1 \rangle$, namely the binary number represented by the 5-bit field RS1 in the instruction. Let R_j denote the word stored in the GPR whose index is j . Let $\langle R_j \rangle$ denote the binary number represented by R_j . Recall that $[imm]$ denotes the two's complement number represented by the 16-bit field imm . We denote the effective address by ea . Then,*

$$ea \triangleq \text{mod}(\langle R_j \rangle + [imm], 2^{32}).$$

Question 13.3 *Explain the disadvantage of using $\langle R_j \rangle + [imm]$ as the effective address.*

Question 13.4 *Let $X[31 : 0]$ and $Y[31 : 0]$ be two binary strings. Prove that addition modulo 2^{32} is not sensitive to binary or two's complement representation. Namely,*

$$\text{mod}(\langle \vec{X} \rangle + \langle \vec{Y} \rangle, 2^{32}) = \text{mod}(\left[\vec{X} \right] + \langle \vec{Y} \rangle, 2^{32}) = \text{mod}(\left[\vec{X} \right] + \left[\vec{Y} \right], 2^{32}).$$

Prove that $ea = \text{mod}([imm] + [R_j], 2^{32}) = \text{mod}(\langle \text{sext}(imm) \rangle + \langle R_j \rangle, 2^{32})$.

Question 13.5 *Consider the computation of the effective address. Suppose that we wish to detect the event that the computation overflows. Formally,*

$$\langle R_j \rangle + [imm] \geq 2^{32} \quad \text{or} \quad \langle R_j \rangle + [imm] < 0.$$

Suggest how to compute the effective address and how to detect overflow.

The semantics of load and store instruction are as follows.

Definition 13.2 Let $i = \langle RD \rangle$, namely the binary number represented by the 5-bit field RD in the instruction. Let R_i denote the word stored in the GPR whose index is i . A load instruction has the following meaning:

$$R_i \leftarrow M[\text{ea}].$$

This means that the word stored in $M[\text{ea}]$ is copied to register R_i .

A store instruction has the following meaning:

$$M[\text{ea}] \leftarrow R_i.$$

This means that the word stored in R_i is copied to $M[\text{ea}]$.

Notation. From this point on, we will abuse notation to abbreviate. We will use $RS1$ to denote the word stored in the register whose index is $\langle RS1 \rangle$. This word was denoted by R_j in the definition of the effective address. The same abbreviation is used for $RS2$ and RD . Similarly, $\langle RS1 \rangle$ will denote the binary number represented by $RS1$. This number was denoted by $\langle R_j \rangle$ in the definition of the effective address. Finally, $[RS1]$ will denote the two's complement number represented by $RS2$.

Immediate Instructions (I-type). The only immediate instruction we have is an addition instruction.

Instruction	Semantics
addi RD RS1 imm	$RD := RS1 + \text{sext}(\text{imm})$

The semantics of an add-immediate instruction are as follows.

$$RD \leftarrow \text{bin}(\text{mod}([RS1] + [\text{imm}], 2^{32})). \quad (13.2)$$

Equation 13.2 is too terse; we clarify it now. The goal is to add: (i) the two's complement number represented by the word stored in the register whose index is $\langle RS1 \rangle$ and (ii) the two's complement number represented by the string stored in the immediate field of the instruction. The addition is modulo 2^{32} . The binary representation of the sum is stored in the register whose index is $\langle RD \rangle$.

This definition is a bit confusing. One might ask why not encode the sum as a two's complement number? The following question deals with this issue.

Question 13.6 Suppose that $[\vec{A}] = [\vec{B}]$ and that $\langle \vec{C} \rangle = \text{mod}([\vec{B}], 2^{32})$. Prove that $\vec{A} = \vec{C}$.

Shift Instructions (I-type). The shift instructions perform a logical shift by one position either to the left or to the right.

Instruction	Semantics
slli RD RS1	$RD := RS1 \ll 1$
srlr RD RS1	$RD := RS1 \gg 1$

ALU Instructions (R-type). ALU instructions add, subtract, or perform logical bitwise operations.

Instruction	Semantics
add RD RS1 RS2	$RD := RS1 + RS2$
sub RD RS1 RS2	$RD := RS1 - RS2$
and RD RS1 RS2	$RD := RS1 \wedge RS2$
or RD RS1 RS2	$RD := RS1 \vee RS2$
xor RD RS1 RS2	$RD := RS1 \oplus RS2$

Test Instructions (I-type). The test instructions compare the two's complement number stored in RS1 with the two's complement number stored in the immediate field of the instruction. The result of the comparison is stored in RD.

Instruction	Semantics
<i>srel</i> i RD RS1 imm	$RD := 1$, if condition is satisfied, $RD := 0$ otherwise
if <i>rel</i> =lt	test if $RS1 < sext(imm)$
if <i>rel</i> =eq	test if $RS1 = sext(imm)$
if <i>rel</i> =gt	test if $RS1 > sext(imm)$
if <i>rel</i> =le	test if $RS1 \leq sext(imm)$
if <i>rel</i> =ge	test if $RS1 \geq sext(imm)$
if <i>rel</i> =ne	test if $RS1 \neq sext(imm)$

Branch/Jump Instructions (I-type). Branch and jump instructions control the PC according to a condition.

Instruction	Semantics
beqz RS1 imm	$PC = PC + 1 + sext(imm)$, if $RS1 = 0$ $PC = PC + 1$, if $RS1 \neq 0$
bnez RS1 imm	$PC = PC + 1$, if $RS1 = 0$ $PC = PC + 1 + sext(imm)$, if $RS1 \neq 0$
jr RS1	$PC = RS1$
jalr RS1	$R31 = PC+1$; $PC = RS1$

Miscellaneous Instructions (I-type). Here we list instructions that are “special”.

Instruction	Semantics
special-nop	causes transition to Init/Fetch states
halt	causes transition to HALT state