Chapter 13

The ISA of a simplified DLX

In this chapter we describe a specification of a simple microprocessor called the simplified DLX. The specification is called an instruction set architecture (ISA). The ISA consists of objects and a machine language which is simply a list of instructions. The ISA describes the set of objects (memory and registers) and how they are manipulated by the machine language instructions.

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; even a very simple microprocessor is built from tens of thousands of gates and an operating system spans thousands of lines of code. To simplify things, people focus at a given time 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, on the other hand, often does not even know exactly which type of computer will be executing the program she is writing. 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 (e.g., editor, compiler, linker, loader, debugger). The user, who runs various applications, uses the abstraction of a computer that is capable of running several applications concurrently, supports a file system, and responds to mouse movements and typing on the keyboard. Supporting the user's abstraction requires an operating system (to coordinate between several programs running in the same time and manage the file system), and hardware (that executes programs, but not in C). The hardware designer, is given a specification, called the Instruction Set Architecture (in short, ISA). Her goal is to design a circuit that implements this specification while minimizing cost and delay.

The architect is supposed to be aware of these different viewpoints. The architect's main goal is to suggest an ISA. On one hand, this ISA should provide support for the users of the ISA (these are the programmer, the end user, and even the operating system). On the other, the ISA should be simple enough so that the hardware designer can come up with an implementation that is not too expensive or slow.

What exactly is the ISA? The ISA is a specification of the 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. Hence a C program undergoes the following sequence of translations: 1. The compiler translates it to an assembly program. 2. The assembler translates it to a machine language program.

This two-stage sequence 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 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 borderlines 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

We now describe the ISA of the simplified DLX. 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 words (i.e. binary strings) 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. In the DLX ISA, a word is a 32-bit string. The memory is often called also the main memory.

The 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 is 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 (Memory Data Register).
- Write or store in $M[i]$. Request to store the contents of a register called MDR in $M[i]$.

Note that writing to the memory require two “operands”. Namely, we need to specify which word we would like to store and we need to specify where we wish to store it. As mentioned above, a special register, called the MDR, stores the word that we wish to write to the memory. The index or address i in which we would like to store the contents of the MDR is output by a register called the MAR (Memory Address Register).

Hence the (partial) semantics of a write operation are:

$$M[\langle MAR \rangle] \leftarrow MDR.$$

Note the angular brackets around the MAR; they signify that we interpret the binary string stored in the MAR as a binary number.

Similarly, the (partial) semantics of a read operation are:

$$MDR \leftarrow M[\langle MAR \rangle].$$

The reason that we refer to this description as a partial semantics is that an actual write operation involves loading the MAR and MDR. (In a read operation we need to load the MAR and copy to MDR to a final destination.) However, from the point of view of the memory the above semantics is correct when data is written or read.

This relatively neat description is incorrect when we consider the task of reading an instruction from the memory. As we will see later, the address of an instruction is stored in a register called PC and $M[\text{PC}]$ is stored in a register called IR.

Registers. The registers serve as the working space of the microprocessor. They have three main purposes: (1) to control the microprocessor (e.g., the PC and IR), (2) to serve as the scratch pad for data (e.g., the GPRs), or (3) an interface with the main memory (e.g., MAR and MDR). The architectural registers of the simplified DLX are all 32 bits wide and listed below.

- 32 General Purpose Registers (GPRs) called R0 to R31. Loosely speaking, the general purpose registers are the objects that the program directly manipulates.
- Program Counter (PC). The PC stores the address (i.e., index in memory) of the instruction that is currently being executed.
- Instruction Register (IR). The IR stores the current instruction (i.e., IR $= M[\mathrm{PC}]$).
- Special Registers: MAR, MDR. As mentioned above, these registers are used for specifying the interface between the microprocessor and the memory when data is written and read.

Example 13.1 *Consider a high level instructions $x := y + z$. Such an instruction is implemented by the following sequence of instructions. Suppose that x is stored in $M[1]$, y is stored in $M[2]$, and z is stored in $M[3]$. We first need to copy x and y to the GPRs. Namely, we first need to perform two read operations that copy $M[1]$ to* R1 *and $M[2]$ to* R2. *We then perform the actual addition:* $\mathrm{R3} \leftarrow \mathrm{R1} + \mathrm{R2}$. *Finally, we copy* R3 *using a write operation to the memory location $M[3]$.*

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?*

Question 13.3 *We said that the same memory is used to store operating system programs and data as well as the user's program and data. How can we make sure that the user program does not write to areas in the memory that "belong" to the operating system?*

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 (in binary representation) the indexes of general purpose registers. The immediate field encodes (in two's complement representation) a constant. The function field (in an R-type instruction format) is used to encode the instruction.

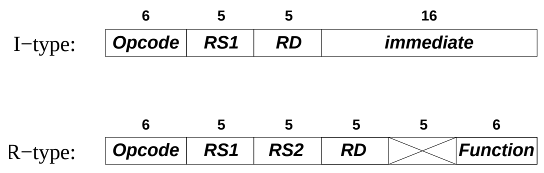

Figure 13.1: Instruction formats of the simplified DLX. (Bits are ordered in descending order; namely, the leftmost bit is in position [31] and the rightmost bit is in position [0].)

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 microprocessor also performs the operation:

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

Informally, Equation 13.1 simply means add one to the binary number represented by the PC. To be precise, the sum is computed modulo 2^{32}, namely, if the sum equals 2^{32}, then replace the sum by zero. Note that (unsigned) binary representation is used for storing the address of the current instruction in the PC.

Load/Store Instructions (I-type). Load and store instructions deal with copying words between the memory and the GPRs. An informal and abbreviated interpretation of the load and store instruction is given in the table below.

Load/Store	Semantics
`lw    RD RS1 imm`	RD := M[sext(imm+RS1]
`sw    RD RS1 imm`	M[sext(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 $\mathrm{R}j$ denote the word stored in the GPR whose index is j. Let $\langle \mathrm{R}j \rangle$ denote the binary number represented by $\mathrm{R}j$. Recall that $[imm]$ denotes the two's complement number represented by the 16-bit field imm. We denote the effective address by* ea. *Then,*

$$\mathrm{ea} \overset{\triangle}{=} \mathrm{mod}(\langle \mathrm{R}j \rangle + [imm], 2^{32}).$$

The following questions help clarify the cumbersome definition of the effective address.

Question 13.4 *If we ignore the issue of overflow, then the effective address is simply $\langle \mathrm{R}j \rangle + [imm]$. Why is this better than say using the definition of $\langle \mathrm{R}j \rangle + \langle imm \rangle$ as the effective address?*

In the next question we show the modulo operation implies that there is no difference between addition in two's complement and in binary representation, provided that sign extension takes place.

Question 13.5 *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,*

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

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

Question 13.6 *Consider the computation of the effective address. Suppose that we wish to detect the event that the computation overflows. Formally,*

$$\langle \mathrm{R}j \rangle + [imm] \geq 2^{32} \quad \textit{or} \quad \langle \mathrm{R}j \rangle + [imm] < 0.$$

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 $\mathrm{R}i$ denote the word stored in the GPR whose index is i. A load instruction has the following meaning:*

$$\mathrm{R}i \leftarrow M[\mathrm{ea}].$$

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

A store instruction has the following meaning:

$$M[\mathrm{ea}] \leftarrow \mathrm{R}i.$$

This means that the word stored in $\mathrm{R}i$ is copied to $M[\mathrm{ea}]$.

Notation. Following the notation used for load and store instructions we use the following notation:

- Ri denotes the word stored in the GPR whose index is $\langle RD \rangle$.
- Rj_1 denotes the word stored in the GPR whose index is $\langle RS1 \rangle$.
- Rj_2 denotes the word stored in the GPR whose index is $\langle RS2 \rangle$.

Obviously, $\langle \mathrm{R}j_1 \rangle$ denotes the binary number represented by the word Rj_1. Similarly, $[\mathrm{R}j_2]$ denotes the two's complement number represented by the work Rj_2.

Add Instruction (I-type). There are two add instructions in the ISA. We describe below the add instruction that belongs to the I-type format. In the table below an informal description is provided.

Instruction	Semantics
`addi  RD RS1 imm`	RD := RS1 + sext(imm)

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

$$RD \leftarrow bin(\mathrm{mod}([\mathrm{R}j_1] + [imm], 2^{32})). \tag{13.2}$$

Equation 13.2 is too terse; we clarify it now. The goal is to add two numbers. The first addend is the two's complement number represented by the word stored in the register whose index is $\langle RS1 \rangle$. The second addend is 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? Namely, why not simply use the definition $[RD] = [\mathrm{R}j_1] + [imm]$? The problem with this "simple" specification is what to do if the result overflows.

The following question shows that if no overflow occurs then Equation 13.2 is identical to "ordinary" two's complement addition.

Question 13.7 *Let $\vec{A}$ and $\vec{C}$ denote 32-bit binary strings. Let $\vec{B}$ denote a binary string of any length. Think of A as the 32 bit two's complement representation of $[X] + [Y]$ if no overflow occurs. Think of B as the representation of $[X] + [Y]$ in two's complement (using as many bits as required). Suppose that $\left[\vec{A}\right] = \left[\vec{B}\right]$ and that $\langle \vec{C} \rangle = \mathrm{mod}(\left[\vec{B}\right], 2^{32})$. Prove that $\vec{A} = \vec{C}$.*

When we deal with interrupts, we will also define two additional "side-effects" of addition instructions, namely, the setting of the overflow and negative flags.

Shift Instructions (R-type). The shift instructions perform a logical shift by one position either to the left or to the right. The input is word Rj_1 and the shifted word is stored in Ri.

Instruction	Semantics
`sll RD RS1`	RD := RS1 << 1
`srl RD RS1`	RD := RS1 >> 1

ALU Instructions (R-type). The R-type arithmetic and logical unit (ALU) instructions are: add, subtract, and logical bitwise operations (e.g., OR, AND, XOR). An informal description of these instruction appears in the following table.

Instruction	Semantics
`add RD RS1 RS2`	RD := RS1 + RS2
`sub RD RS1 RS2`	RD := RS1 − RS2
`and RD RS1 RS2`	RD := AND(RS1, RS2)
`or RD RS1 RS2`	RD := OR(RS1, RS2)
`xor RD RS1 RS2`	RD := XOR(RS1, RS2)

Formally, the semantics of the add and subtract instructions are:

$$RD \leftarrow bin(\text{mod}([\text{R}j_1] + [\text{R}j_2]\,, 2^{32}))$$
$$RD \leftarrow bin(\text{mod}([\text{R}j_1] - [\text{R}j_2]\,, 2^{32})).$$

The semantics of the bitwise logical instructions are simple. For example, in an AND instruction $RD[i] = \text{AND}(\text{R}j_1[i], \text{R}j_2[i])$.

Test Instructions (I-type). The test instructions compare the two's complement numbers $[\text{R}j_1]$ and $[imm]$. The result of the comparison is stored in RD.

For example, consider the `slti` instruction. The semantics of the `slti` instruction are:

$$RD = \begin{cases} 1 & \text{if } [\text{R}j_1] < [imm] \\ 0 & \text{otherwise.} \end{cases}$$

There are six different test instructions: `slti, seqi, sgti, slei, sgei, snei`. We summarize there functionality below.

Instruction	Semantics
`s`*rel*`i RD RS1 imm`	RD := 1, if condition is satisfied,
	RD := 0 otherwise
if *rel* =`lt`	test if RS1 < sext(imm)
if *rel* =`eq`	test if RS1 = sext(imm)
if *rel* =`gt`	test if RS1 > sext(imm)
if *rel* =`le`	test if RS1 ≤ sext(imm)
if *rel* =`ge`	test if RS1 ≥ sext(imm)
if *rel* =`ne`	test if RS1 ≠ sext(imm)

Question 13.8 *How is the condition* $[\mathrm{R}j_1] < [imm]$ *computed? Let us return to the negative flag of the signed adder/subtractor. Is it crucial that the negative flag indicates correctly whether the sum/difference is negative even in case of an overflow?*

Branch/Jump Instructions (I-type). Branch and jump instructions modify the value stored in the the PC. Recall that during the execution of every instruction the PC is incremented. In a branch or jump instruction an additional change is made to the PC.

The simplest instruction in this set is the "jump register" (`jr`) instruction. It simply changes the PC so that $\mathrm{PC} \leftarrow \mathrm{R}j_1$. Hence the next instruction is the instruction stored in $M[\mathrm{R}j_1]$.

A somewhat more evolved instruction is the "jump and link register" (`jalr`) instruction. This instruction saves the incremented PC in R31. The idea is that this instruction is used for calling a procedure and the return address is stored in R31. Formally, the semantics of `jalr` are:

$$\begin{aligned} \mathrm{R31} &\leftarrow \mathrm{PC} + 1 \\ \mathrm{PC} &\leftarrow \mathrm{R}j_1. \end{aligned}$$

We also have two branch instructions: "branch if zero" (`beqz`) and "branch if not zero" (`bnez`). In a `beqz` instruction, if $\mathrm{R}j_1 = 0^{32}$ then a branch takes place and the address of the next instruction is $\mathrm{PC} + 1 + [imm]$. If $\mathrm{R}j_1 \neq 0^{32}$, then the branch is not taken, and the address of the next instruction is $\mathrm{PC} + 1$. In a `bnez` instruction, the conditions are reversed.
We summarize these four instructions in the following table.

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

Question 13.9 *Why is the address of the next instruction defined as* $\mathrm{PC} + 1 + [imm]$ *instead of* $\mathrm{PC} + [imm]$ *when a branch is taken?*

Miscellaneous Instructions (I-type). There are a few special instructions in the I-type format. The first special instruction is a the "no operation" (`special-nop`) instruction. This instruction has a null effect, and the only thing that happens during its execution is that the PC is incremented.

The second special instruction is the "halt" (`halt`) instruction. This instruction causes the microprocessor to "freeze" and stop the execution of the program.

Question 13.10 *Try to explain when the no-operation and halt instruction are used.*

13.3 Summary

In this chapter we described the ISA of the simplified DLX. Even though the ISA is rather simple, most C instructions and programs can be translated to the DLX machine language. There are a few exceptions such as supporting systems calls, distinguishing between protected mode and user mode, etc. This important issues will be addressed when we discuss interrupts.