Digital Logic Design: a rigorous approach ©

Chapter 21: The ISA of a Simplified DLX

Guy Even Moti Medina

School of Electrical Engineering Tel-Aviv Univ.

June 7, 2020

Book Homepage:
http://www.eng.tau.ac.il/~guy/Even-Medina

Why use abstractions?

- According to the Collins Dictionary “architecture” means the art of planning, designing, and constructing buildings.
- Computer architecture refers to computers instead of buildings.
- Computers are complicated.
- Very simple microprocessor is built from tens of thousands of gates and an operating system spans thousands of lines of instructions.

Why use abstractions? (cont)

- To simplify things, people focus at a given time on certain aspects of computers and ignore other aspects.
  - 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.
- The architect is supposed to be aware of different aspects so that the designed system meets the required price and performance goals.

Abstractions in Computer Systems

- Several abstractions are used in computer systems.
- We focus on three abstractions used by different “players”:
    - The C programmer (who writes programs)
    - The user (who executes programs)
    - The hardware designer.

Our Players

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).

Our Players

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).

Our Players

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 and the ISA

- 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.

ISA?

- 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!

The Pair: Compiler & Assembler

- 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 language 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.

The Pair: Compiler & Assembler

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.
3. Only one compiler is required. For each platform, there is an assembler that translates the assembly programs to the machine language of the platform.
4. 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 suited to the current abstraction.

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.

The Main Memory

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

The Memory - MAR and MDR

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:

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

Memory Access

For example, in a read operation we need to

1. compute the address and store it in the *MAR*,
2. copy the contents of the accessed word in the memory to the *MDR*, and
3. copy the contents of the *MDR* to a general purpose register.

However, from the point of view of the memory, the interaction with the microprocessor is via the *MAR* and *MDR*.
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 $M[\langle \mathrm{PC} \rangle]$ 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).

Registers

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

- 32 General Purpose Registers (GPRs) indexed from 0 to 31. We refer to these registers as $\mathrm{R0}$ to $\mathrm{R31}$. Loosely speaking, the general purpose registers are the objects that the program directly manipulates. Register $\mathrm{R0}$ is an exception, as its contents always equals 0^{32} and cannot be modified.
- 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., $\mathrm{IR} = M[\langle \mathrm{PC} \rangle]$).
- Special Registers: MAR, MDR. As mentioned above, these registers serve as the interface between the microprocessor and the memory when data is written and read.

GPR - example

- Instructions are separated to memory accesses and "computations".
- The arguments and result of computations are stored in GPRs.

Example

Consider a high level instructions $z := x + y$. 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 $\mathrm{R1}$ and $M[2]$ to $\mathrm{R2}$. We then perform the actual addition: $\mathrm{R3} \leftarrow \mathrm{R1} + \mathrm{R2}$. Finally, we copy $\mathrm{R3}$ using a write operation to the memory location $M[3]$.

Instruction Formats

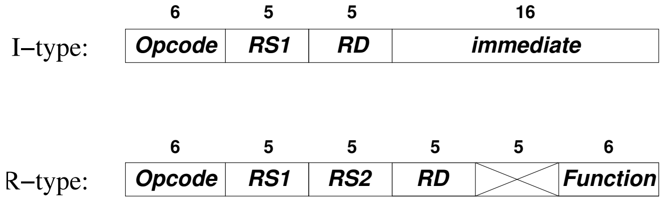

Figure: Instruction formats of the simplified DLX. (Bits are ordered in descending order; namely, the leftmost bit is in position [31] and the rightmost bit is in position [0].)

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

Effective Address

Definition

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 register of 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,

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

Overflow?

We point out that the event that $\langle \mathrm{R}j\rangle + [imm] \notin \{0, \ldots, 2^{32} - 1\}$ is (most likely) an indication of a programming error. In certain architectures, such an event creates a segmentation fault. In the simplified DLX, we do not consider this event to be an error, and the modulo operation is a side effect of using a simple adder for computing the effective address

The semantics of load and store

Definition

Let $i = \langle RD \rangle$, namely i is the number represented in binary representation by the 5-bit field RD in the instruction. Let $\mathrm{R}i$ denote the word stored in the ith register in the GPR.

1. A load instruction has the following meaning:

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

   This means that the word stored in $M[ea]$ is copied to register $\mathrm{R}i$. Of course, $M[ea]$ retains its value.

2. A store instruction has the following meaning:

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

   This means that the word stored in $\mathrm{R}i$ is copied to $M[ea]$. Of course, $\mathrm{R}i$ retains its value.

Notation

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

- $\mathrm{R}i$ denotes the word stored in the register of the GPR whose index is $\langle RD \rangle$.
- $\mathrm{R}j_1$ denotes the word stored in the register of the GPR whose index is $\langle RS1 \rangle$.
- $\mathrm{R}j_2$ denotes the word stored in the register of the GPR whose index is $\langle RS2 \rangle$.

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.

$$\mathrm{R}i \leftarrow \mathit{bin}(\mathrm{mod}([\mathrm{R}j_1] + [\mathit{imm}]\,, 2^{32})). \tag{1}$$

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 $\mathrm{R}j_1$ and the shifted word is stored in $\mathrm{R}i$.

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

ALU Instructions (R-type).

Instruction	Semantics
add RD RS1 RS2	RD := RS1 + RS2
sub RD RS1 RS2	RD := RS1 − RS2
and RD RS1 RS2	RD := $\textsc{and}$(RS1, RS2)
or RD RS1 RS2	RD := $\textsc{or}$(RS1, RS2)
xor RD RS1 RS2	RD := $\textsc{xor}$(RS1, RS2)

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

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

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

Test Instructions (I-type).

The test instructions compare the two's complement numbers $[\mathrm{R}j_1]$ and $[imm]$. The result of the comparison is stored in $\mathrm{R}i$. For example, consider the `slti` instruction. The semantics of the `slti` instruction are:

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

Test Instructions (I-type).

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 $\leq$ sext(imm)
if *rel* $=$`ge`	test if RS1 $\geq$ sext(imm)
if *rel* $=$`ne`	test if RS1 $\neq$ sext(imm)

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 to be executed is the instruction stored in $M[\mathrm{R}j_1]$.

Branch/Jump Instructions (I-type).

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

$$\begin{aligned}\mathrm{R31} &\leftarrow \mathit{bin}(\mathrm{mod}(\langle PC\rangle + 1, 2^{32}))\\ \mathrm{PC} &\leftarrow \mathrm{R}j_1.\end{aligned}$$

Branch/Jump Instructions (I-type).

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.

Branch/Jump Instructions (I-type).

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).

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.

I-type instructions

IR[31 : 26]	Mnemonic	Semantics
Data Transfer		
100 011	lw	$RD = M[sext(imm)+RS1]$
101 011	sw	$M[sext(imm)+RS1] = RD$
Arithmetic, Logical Operation		
001 011	addi	$RD = RS1 + sext(imm)$
Test Set Operation		
011 rel	s rel i	$RD = (RS1 \text{ rel } sext(imm))$
011 001	sgti	$RD = (RS1 > sext(imm))$
011 010	seqi	$RD = (RS1 = sext(imm))$
011 011	sgei	$RD = (RS1 \geq sext(imm))$
011 100	slti	$RD = (RS1 < sext(imm))$
011 101	snei	$RD = (RS1 \neq sext(imm))$
011 110	slei	$RD = (RS1 \leq sext(imm))$
Control Operation		
000 100	beqz	$PC = PC + 1 + (RS1 = 0 \ ? \ sext(imm) :$
000 101	bnez	$PC = PC + 1 + (RS1 \neq 0 \ ? \ sext(imm) :$
010 110	jr	$PC = RS1$

R-type instructions

IR[5 : 0]	Mnemonic	Semantics
Shift Operation		
000 000	sll	$\text{RD} = \text{RS1} \ll 1$
000 010	srl	$\text{RD} = \text{RS1} \gg 1$
Arithmetic, Logical Operation		
100 011	add	$\text{RD} = \text{RS1} + \text{RS2}$
100 010	sub	$\text{RD} = \text{RS1} - \text{RS2}$
100 110	and	$\text{RD} = \text{RS1} \wedge \text{RS2}$
100 101	or	$\text{RD} = \text{RS1} \vee \text{RS2}$
100 100	xor	$\text{RD} = \text{RS1} \oplus \text{RS2}$

Table: R-type Instructions (in R-type instructions $\text{IR}[31:26] = 0^6$)

Example 1 of Program Segments

Convert the C code segment below to a simplified DLX's machine code.

```
    if (i==j)
       goto L1;
    f=g+h;
L1: f=f-i;
```

Register Assignment

Variable	Register
f	R1
g	R2
h	R3
i	R4
j	R5

Table: Register assignment for Example 1

Code conversion

C code	DLX's machine code
if (i==j)	xor R6 R4 R5
goto L1;	beqz R6 1
f=g+h;	add R1 R2 R3
L1: f=f-i;	sub R1 R1 R4

Example 2

Convert the C code segment below to a simplified DLX's machine code.

```
LOOP: g=g+A[i];
      i=i+j;
      if (i!=h) goto LOOP;
```

Register Assignment

Variable	Register
g	R1
h	R2
i	R3
j	R4
A	R5
A+i	R6
A[i]	R7
i!=h	R8

Code Conversion

C code	DLX's machine code
LOOP: g=g+A[i];	add R6 R5 R3
	lw R7 R6 0
	add R1 R1 R7
i=i+j;	add R3 R3 R4
if (i!=h) goto LOOP;	xor R8 R3 R2
	bnez R8 -6

Summary

- In this chapter we described the ISA of the simplified DLX.
- Even though the ISA is rather simple, C instructions and programs can be translated to the DLX machine language.
- Missing in this description is supporting systems calls, distinguishing between protected mode and user mode, etc.