2.6 Representing instructions in the computer

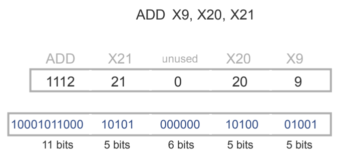
We are now ready to explain the difference between the way humans instruct computers and the way computers see instructions.

Instructions are kept in the computer as a series of high and low electronic signals and may be represented as numbers. In fact, each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the instruction. The 32 registers of LEGv8 are just referred to by their number, from 0 to 31.

The below animation shows the conversion of an assembly instruction into a machine instruction consisting of 0's and 1's. A machine instruction is composed of ***fields***, each field having several bits and representing some part of the instruction.

PARTICIPATION ACTIVITY 2.6.1: Example of translating a LEGv8 assembly instruction into a machine instruction.

Start 2x speed



This layout of the instruction is called the *instruction format*. As you can see from counting the number of bits, this LEGv8 instruction takes exactly 32 bits—a word, or one half of a doubleword. In keeping with our design principle that simplicity favors regularity, all LEGv8 instructions are 32 bits long.

Instruction format: A form of representation of an instruction composed of fields of binary numbers.

To distinguish it from assembly language, we call the numeric version of instructions *machine language* and a sequence of such instructions machine code.

Machine language: Binary representation used for communication within a computer system.

PARTICIPATION ACTIVITY 2.6.2: LEGv8 instruction fields.

Match the assembly instructions item with the instruction field.

Assembly instruction: `ADD X1, X4, X3`

LEGv8 instruction fields:

Field 1	Field 2	Field 3	Field 4	Field 5

unused X4 X3 ADD X1

Field 1

Field 2

Field 3

Field 4

Field 5

Reset

PARTICIPATION ACTIVITY 2.6.3: LEGv8 machine instruction fields and values.

1) How many bits are used to indicate that an instruction performs ADD?

Check Show answer

2) An ADD instruction has fields for three registers. How many bits are used for any one of those registers?

Check Show answer

3) How many total bits is a machine

instruction?

Check Show answer

4) Given: ADD X5, X9, X1
What decimal value is stored in field 2?

Check Show answer

5) Given: ADD X5, X9, X1
What 5-bit value is stored in field 4?

Check Show answer

It would appear that you would now be reading and writing long, tiresome strings of binary numbers. We avoid that tedium by using a higher base than binary that converts easily into binary. Since almost all computer data sizes are multiples of 4, *hexadecimal* (base 16) numbers are popular. As base 16 is a power of 2, we can trivially convert by replacing each group of four binary digits by a single hexadecimal digit, and vice versa. The figure below converts between hexadecimal and binary.

Hexadecimal: Numbers in base 16.

Figure 2.6.1: The hexadecimal-binary conversion table (COD Figure 2.4).

Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

PARTICIPATION ACTIVITY 2.6.4: Hexadecimal.

1) What is 0011 as a hexadecimal digit?

Check Show answer

2) What is 1011 as a hexadecimal digit?

Check Show answer

3) What is 11110000 in 2-digit hexadecimal? Write answer as: a1

Check Show answer

4) What is 2f in 8-bit binary?

Check Show answer

PARTICIPATION ACTIVITY 2.6.5: Binary and hex tool.

Start

dec	bin	hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B

12	1100	C
13	1101	D
14	1110	E
15	1111	F

1 2 3 4

Check Next

Because we frequently deal with different number bases, to avoid confusion, we will subscript decimal numbers with *ten*, binary numbers with *two*, and hexadecimal numbers with *hex*. (If there is no subscript, the default is base 10.) By the way, C and Java use the notation 0x*nnnn* for hexadecimal numbers.

Example 2.6.1: Binary to hexadecimal and back.

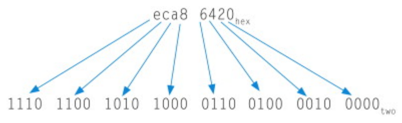
Convert the following 8-digit hexadecimal and 32-bit binary numbers into the other base:

$$\mathrm{eca8\ \ 6420_{hex}}$$

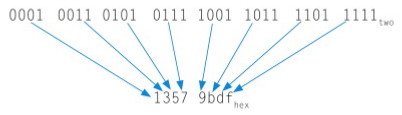
$$\mathrm{0001\ \ 0011\ \ 0101\ \ 0111\ \ 1001\ \ 1011\ \ 1101\ \ 1111_{two}}$$

Answer

Using the previous figure, the answer is just a table lookup one way:



And then the other direction:



LEGv8 fields

LEGv8 fields are given names to make them easier to discuss:

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

Here is the meaning of each name of the fields in LEGv8 instructions:

- *opcode* : Basic operation of the instruction, and this abbreviation is its traditional name.
- *Rm*: The second register source operand.
- *shamt*: Shift amount. (COD Section 2.6 (Logical operations) explains shift instructions and this term; it will not be used until then, and hence the field contains zero in this section.)
- *Rn*: The first register source operand.
- *Rd*: The register destination operand. It gets the result of the operation.

Opcode: The field that denotes the operation and format of an instruction.

A problem occurs when an instruction needs longer fields than those shown above. For example, the load register instruction must specify two registers and a constant. If the address were to use one of the 5-bit fields in the format above, the largest constant within the load register instruction would be limited to only 2^5-1 or 31. This constant is used to select elements from arrays or data structures, and it often needs to be much larger than 31. This 5-bit field is too small to be useful.

Hence, we have a conflict between the desire to keep all instructions the same length and the desire to have a single instruction format. This conflict leads us to the final hardware design principle:

Design Principle 3: Good design demands good compromises.

The compromise chosen by the LEGv8 designers is to keep all instructions the same length, thereby requiring distinct of instruction formats for different kinds of instructions. For example, the format above is called *R-type* (for register) or *R-format*. A second type of instruction format is *D-type* or *D-format* and is used by the data transfer instructions (loads and stores). The fields of D-format are

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

The 9-bit address means a load register instruction can load any doubleword within a region of $\pm2^8$ or 256 bytes ($\pm2^5$ or 32 doublewords) of the address in the base register Rn. We see that more than 32 registers would be difficult in this format, as the Rn and Rt fields would each need another bit, making it harder to fit everything in one word. (The last field of D-type is called Rt instead of Rd because for store instructions, the field indicates a data source and not a data destination.)

Let's look at the load register instruction:

```
LDUR   X9, [X22,#64]   // Temporary reg X9 gets A[8]
```

Here, 22 (for `X22`) is placed in the Rn field, 64 is placed in the address field, and 9 (for `X9`) is placed in the Rt field. Note that in a load register instruction, the Rt field specifies the *destination register*, which receives the result of the load. A ***destination register*** is a register that receives the result of an operation.

We also need a format for the immediate instructions `ADDI`, `SUBI`, and immediate instructions that we will introduce later. While we could have used the D-format instruction since it has a 9-bit field holding a constant, the ARMv8 architects decided it would be useful to have a larger immediate field for these instructions, even shaving a bit from the opcode field to make a 12-bit immediate. The fields of *immediate* or *I-type* format are

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

Although multiple formats complicate the hardware, we can reduce the complexity by keeping the formats similar. For example, the last two fields of all three formats are the identical size and almost the same names, and the opcode field is the same size of in two of the three formats.

In case you were wondering, the formats are distinguished by the values in the first field: each format is assigned a distinct set of values in the first field (opcode) so that the hardware knows how to treat the rest of the instruction. The following figure shows the numbers used in each field for the LEGv8 instructions covered so far.

PARTICIPATION ACTIVITY 2.6.6: LEGv8 R-type, I-type, and D-type instruction encoding (COD Figure 2.5).

Start 2x speed

Instruction formats

			Rm	shamt			(R-type)
			immediate				(I-type)
Instruction	Format	opcode	address	op2	Rn	Rd/Rt	(D-type)
`ADD` (add)	R	1112	reg	0	reg	reg	
`SUB` (subtract)	R	1624	reg	0	reg	reg	
`ADDI` (add immediate)	I	580	constant		reg	reg	
`SUBI` (sub immediate)	I	836	constant		reg	reg	
`LDUR` (load register)	D	1986	address	0	reg	reg	
`STUR` (store register)	D	1984	address	0	reg	reg	

Sample instructions

	opcode	Rm	shamt	Rn	Rd
`ADD X1, X2, X3`	1112	3	0	2	1
`SUB X1, X2, X3`	1624	3	0	2	1

	opcode	immediate	Rn	Rd
`ADDI X1, X2, #100`	580	100	2	1
`SUBI X1, X2, #100`	836	100	2	1

	opcode	address	op2	Rn	Rt
`LDUR X1, [X2, #100]`	1986	100	0	2	1
`STUR X1, [X2, #100]`	1984	100	0	2	1

PARTICIPATION ACTIVITY 2.6.7: R-type, I-type, and D-type instructions.

1) What type of instruction is ADD?
   - ○ R-type
   - ○ I-type

2) What type of instruction is ADDI (add immediate)?
   - ○ R-type
   - ○ I-type

3) What type of instruction is STUR (store register)?
   - ○ D-type
   - ○ I-type

4) For ADD, ADDI, and LDUR instructions, Rn represents a register.
   - ○ True
   - ○ False

5) Because I-type and D-type instructions involve a constant or address, I-type and D-type instruction use more bits.
   - ○ True
   - ○ False

Example 2.6.2: Translating LEGv8 assembly language into machine language.

We can now take an example all the way from what the programmer writes to what the computer executes. If `X10` has the base of the array `A` and `X21` corresponds to `h`, the assignment statement

```
A[30] = h + A[30] + 1;
```

is compiled into

```
LDUR  X9, [X10,#240]  // Temporary reg X9 gets A[30]
ADD   X9, X21, X9     // Temporary reg X9 gets h+A[30]
ADDI  X9, X9, #1      // Temporary reg X9 gets h+A[30]+1
STUR  X9, [X10,#240]  // Stores h+A[30]+1 back into A[30]
```

What is the LEGv8 machine language code for these four instructions?

Answer

For convenience, let's first represent the machine language instructions using decimal numbers. From the previous figure, we can determine the four machine language instructions:

opcode	Rm/address	shamt/op2	Rn	Rd/Rt
1986	240	0	10	9
1112	9	0	21	9
580	1		9	9
1984	240	0	10	9

The `LDUR` instruction is identified by 1986 (see COD Figure 2.5) in the first field (opcode). The base register 10 is specified in the fourth field (Rn), and the destination register 9 is specified in the last field (Rt). The offset to select `A[30]` (240 = 30 × 8) is found in the second field (address).

The `ADD` instruction that follows is specified with 1112 in the first field (opcode). The three register operands (9, 21, and 9) are found in the second, fourth, and fifth fields, with 0 in the third field (shamt).

The following `ADDI` instruction is specified with 580 in the first field (opcode), the immediate value 1 in the second, and the register operands (9 in both cases) in the last two fields.

The `STUR` instruction is identified with 1984 in the first field. The rest of this final instruction is identical to the `LDUR` instruction.

Since $240_{ten} = 0\ 1111\ 0000_{two}$, the binary equivalent to the decimal form is:

1111100001̲0	011110000	00	01010	01001
10001011000	01001	000000	10101	01001
1001000100	000000000001		01001	01001
1111100000̲0	011110000	00	01010	01001

Note the similarity of the binary representations of the first and last instructions. The only difference is in the tenth bit from the left, which is highlighted here.

Elaboration

*ARMv8 assembly language programmers aren't forced to use **`ADDI`** when working with constants. The programmer simply writes **`ADD`**, and the assembler generates the proper opcode and the proper instruction format depending on whether the operands are all registers (R-format) or if one is a constant (I-format). We use the explicit names in LEGv8 for the different opcodes and formats as we think it is less confusing when introducing assembly language versus machine language.*

Elaboration

*Note that unlike MIPS, the LEGv8 immediate field in I-format is zero-extended. Thus, LEGv8 includes both **`ADDI`** and **`SUBI`** instructions, while MIPS has just **`ADDI`** and both positive and negative immediates.*

PARTICIPATION ACTIVITY 2.6.8: LEGv8 machine language example.

Given these LEGv8 machine instructions, indicate what each value represents.

10 (in row 1)   240 (in row 1)   1986 (in row 1)   1 (in row 3)   21 (in row 2)

Opcode for a load register instruction.

X10, containing a base address.

An offset

X21

Immediate value

Reset

PARTICIPATION ACTIVITY 2.6.9: Translating LEGv8 instructions to machine language.

Translate `ADDI X7, X4, 5` to the corresponding LEGv8 machine language code. The fields of an I-format instruction are provided below:

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

1) What 10-bit value is stored in the opcode field?

Check   Show answer

2) What 5-bit value is stored in Rn field?

Check Show answer

3) What 5-bit value is stored in Rd field?

Check Show answer

4) What 12-bit value is stored in the immediate field?

Check Show answer

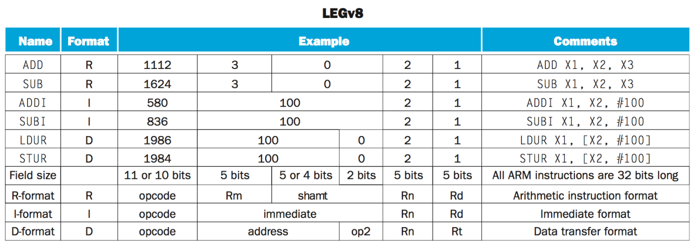
Hardware/Software Interface

The desire to keep all instructions the same size conflicts with the desire to have as many registers as possible. Any increase in the number of registers uses up at least one more bit in every register field of the instruction format. Given these constraints and the design principle that smaller is faster, most instruction sets today have 16 or 32 general-purpose registers.

The following figure summarizes the portions of LEGv8 machine language described in this section. As we shall see in COD Chapter 4 (The Processor), the similarity of the binary representations of related instructions simplifies hardware design. These similarities are another example of regularity in the LEGv8 architecture.

Figure 2.6.2: LEGv8 architecture revealed thus far (COD Figure 2.6).

The three LEGv8 instruction formats so far are R, I and D. The last 10 bits contain a *Rn* field, giving one of the sources; and the *Rd* or *Rt* field, which specifies the destination register, except for store register, where it specifies the value to be stored. R-format divides the rest into an 11-bit opcode; a 5-bit *Rm* field, specifying the other source operand; and a 6-bit *shamt* field, which COD Section 2.6 explains. I-format combines 12 bits into a single *immediate* field, which requires shrinking the opcode field to 10 bits. The D-format uses a full 11-bit opcode like the R-format, plus a 9-bit *address* field, and a 2-bit *op2* field. The op2 field is logically an extension of the opcode field.



LEGv8

Name	Format	Example						Comments
ADD	R	1112	3	0		2	1	ADD X1, X2, X3
SUB	R	1624	3	0		2	1	SUB X1, X2, X3
ADDI	I	580	100			2	1	ADDI X1, X2, #100
SUBI	I	836	100			2	1	SUBI X1, X2, #100
LDUR	D	1986	100		0	2	1	LDUR X1, [X2, #100]
STUR	D	1984	100		0	2	1	STUR X1, [X2, #100]
Field size		11 or 10 bits	5 bits	5 or 4 bits	2 bits	5 bits	5 bits	All ARM instructions are 32 bits long
R-format	R	opcode	Rm	shamt		Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format

PARTICIPATION ACTIVITY 2.6.10: LEGv8 machine language.

1) Opcode 1986 indicates a _____ instruction.

- ○ load register
- ○ store register

2) Opcode _____ indicates a store register instruction.

- ○ 1980
- ○ 1984

3) Opcode 1624 indicates a(n) _____ instruction.

- ○ ADD
- ○ SUB

4) ADDI's opcode is _____.

- ○ 580
- ○ 100

The Big Picture

Today's computers are built on two key principles:

1. Instructions are represented as numbers.

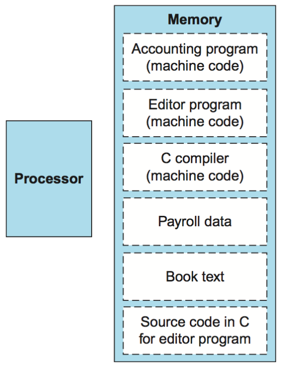
2. Programs are stored in memory to be read or written, just like data.

These principles lead to the *stored-program concept*; its invention let the computing genie out of its bottle. The following figure shows the power of the concept; specifically, memory can contain the source code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generated the machine code.

One consequence of instructions as numbers is that programs are often shipped as files of binary numbers. The commercial implication is that computers can inherit ready-made software provided they are compatible with an existing instruction set. Such "binary compatibility" often leads industry to align around a small number of instruction set architectures.

Figure 2.6.3: The stored-program concept (COD Figure 2.7).

Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the computer can understand.


Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program

PARTICIPATION ACTIVITY 2.6.11: Stored-program concept.

1) The stored-program concept means:

○ Programs are stored in memory along with data.

○ A computer supports a store instruction.

○ Programs are stored on external disks.

PARTICIPATION ACTIVITY 2.6.12: Check yourself: LEGv8 instructions.

1) Which LEGv8 instruction does the following represent?

opcode	Rm	shamt	Rn	Rd
1624	9	0	10	11

○ `SUB X9, X10, X11`

○ `ADD X11, X9, X10`

○ `SUB X11, X10, X9`

○ `SUB X11, X9, X10`

Elaboration

You might be asking yourself why the LEGv8 opcode field is so big given the modest number of instructions in COD Figure 2.1? The main reason is that the full ARMv8 instruction set is very large; depending how you count, it is on the order of 1000 instructions. We'll survey the full ARMv8 instruction set in some of the last sections of this chapter and COD Chapters 3 (Arithmetic for computers) and 5 (Large and fast...).

Provide feedback on this section