The Role of RISC-V

Table of contents

Introduction

This chapter describes, at a high level, the role played by RISC-V and how it fits into the scene. How does a program get compiled and eventually execute on a RISC-V CPU core?

Learning Objectives

By the end of this chapter, you should understand:

– The role of compilers and assemblers;

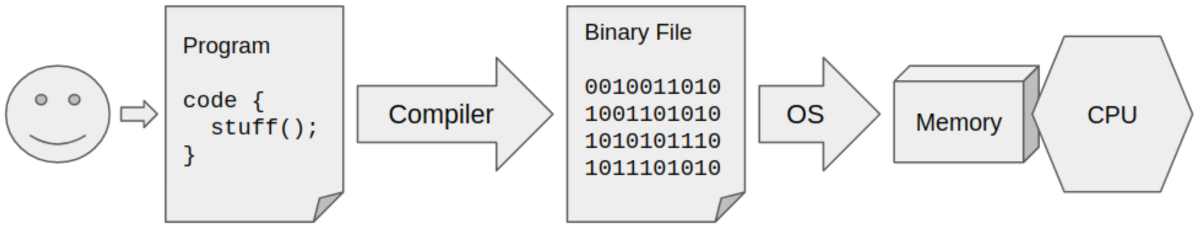
– The role of an instruction set architecture (ISA);

– The general properties of RISC-V versus other ISAs.

Software, Compilers, and CPU

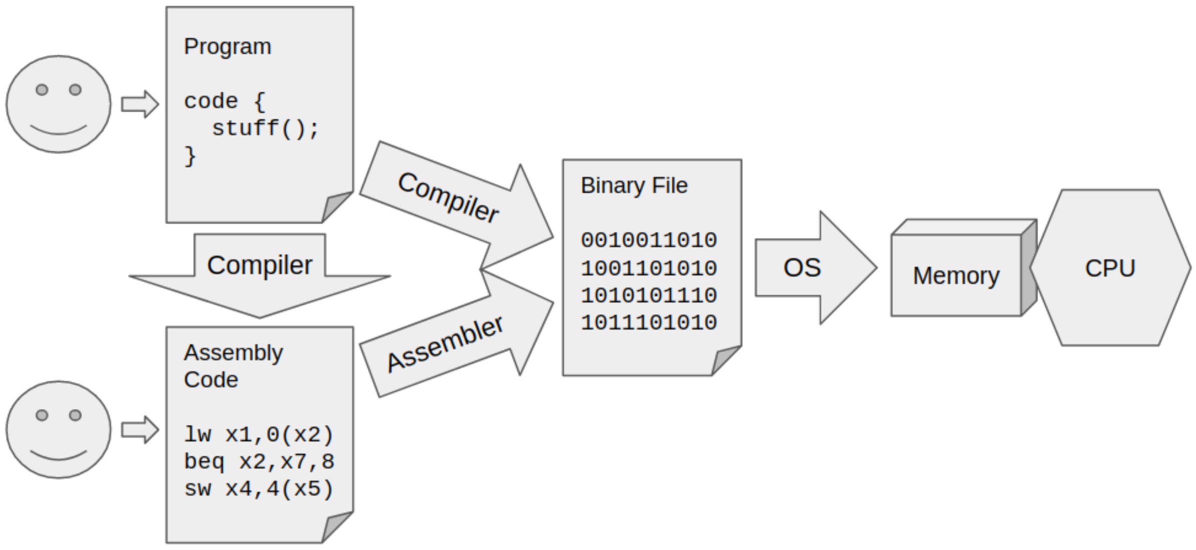
Likely, you have experience writing programs in languages like Python, JavaScript, Java, C++, etc. These languages are portable and can run on just about any CPU hardware. CPU's do not execute these languages directly. They execute raw *machine instructions* that have been encoded into bits as defined by an *instruction set architecture* (ISA). Popular ISAs include x86, ARM, MIPS, RISC-V, etc.

A *compiler* does the job of translating a program's source code into a *binary file* or *executable* containing machine instructions for a particular ISA. An operating system (and perhaps a runtime environment) does the job of loading the binary file into memory for execution by the CPU hardware that understands the given ISA.



Picture 1. Software development and execution flow

The binary file is easily interpreted by hardware, but not so easily by a human. The ISA defines a human-readable form of every instruction, as well as the mapping of those human-readable *assembly instructions* into bits. In addition to producing binary files, compilers can generate *assembly code*. An *assembler* can compile the assembly code into a binary file. In addition to providing visibility to compiler output, assembly programs can also be written by hand. This is useful for hardware tests and other situations where direct low-level control is needed. You will use assembly-level test programs in this course to debug your RISC-V design.



Picture 2. Assembly code development and execution flows

RISC-V Overview

In this course, you will build a simple CPU that supports the RISC-V ISA. RISC-V has very rapidly gained popularity due to its open nature--its explicit lack of patent protection and its community focus. Following the lead of RISC-V, MIPS and PowerPC have subsequently gone open as well.

RISC-V is also popular for its simplicity and extensibility, which makes it a great choice for this course. "RISC", in fact, stands for "reduced instruction set computing" and contrasts with "complex instruction set computing" (CISC). RISC-V (pronounced "risk five") is the fifth in a series of RISC ISAs from UC Berkeley. You will implement the core instructions of the base RISC-V instruction set (RV32I), which contains just 47 instructions. Of these, you will implement 31 (Of the remaining 16, 10 have to do with the surrounding system, and 6 provide support for storing and loading small values to and from memory).

Like other RISC (and even CISC) ISAs, RISC-V is a *load-store architecture*. It contains a register file capable of storing up to 32 values (well, actually 31). Most instructions read from and write back to the register file. Load and store instructions transfer values between memory and the register file.

RISC-V instructions may provide the following fields:

– **opcode:** Provides a general classification of the instruction and determines which of the remaining fields are needed, and how they are laid out, or encoded, in the remaining instruction bits.

– **function field** (funct3/funct7): Specifies the exact function performed by the instruction, if not fully specified by the opcode.

– **rs1/rs2:** The indices (0-31) identifying the register(s) in the register file containing the source operand values on which the instruction operates.

– **rd:** The index (0-31) of the register into which the instruction’s result is written.

– **Immediate:** A value contained within the instruction bits themselves. This value may provide an offset for indexing into memory or a value upon which to operate (in place of the register value indexed by rs2).

All instructions are 32 bits. The R-type encoding provides a general layout of the instruction fields used by all instruction types. R-type instructions have no immediate value. Other instruction types use a subset of the R-type fields and provide an immediate value in the remaining bits.

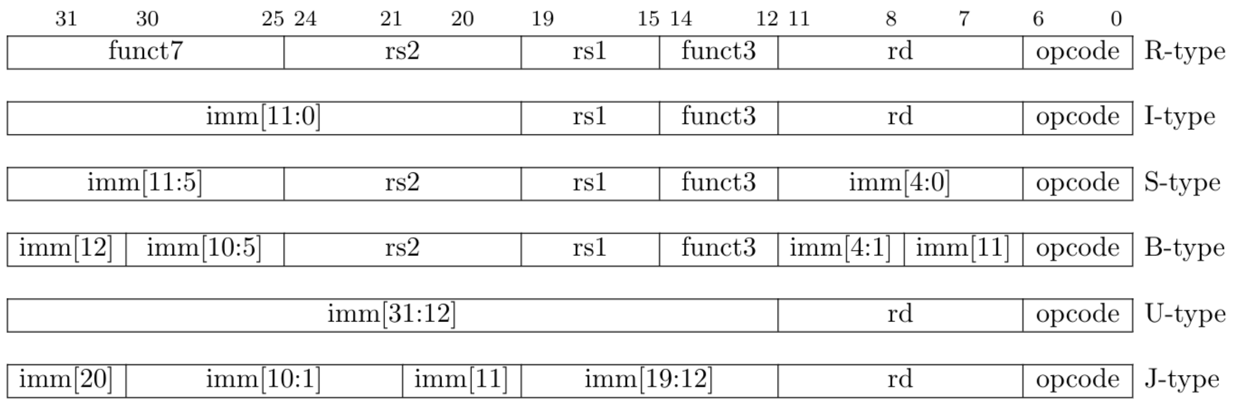

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

Figure 2.3: RISC-V base instruction formats showing immediate variants.

Picture 3. RISC-V base instruction formats (from the RISC-V specifications [1])

You’ll learn further details of the ISA as you build your CPU.

[1] https://riscv.org/technical/specifications/