Computer Systems

Martin de Spirlet

Week	Unit	Title
1	1	Data Representation and Manipulation
2	2	Introduction to Computer Architecture
	3	Instructions, Assembly Language and Machine Code
3	4	Compilation, Interpretation and the JVM

1 Data Representation and Manipulation

1.1 Binary

Computers consist of a collection of switches called transistors, which can either be on or off. This leads to binary representation, where off is 0 and on is 1. All data and instructions are stored in binary. Binary digits are known as 'bits'. It is possible to build computers that use other number systems, but it is less expensive and more reliable to use basic components with only two states.

1.2 Characters

Characters are represented as binary values using text encoding schemes such as ASCII and Unicode. Modern schemes support internationalisation and tend to use 8 bit, 16 bit or 32 bit to encode characters. Strings are sequences of characters.

1.3 Number Bases

1.3.1 Decimal to Binary, Octal and Hexadecimal

$$234_{10}: \quad \left.\begin{array}{r|l} 2\lfloor\underline{234} & 0 \\ 2\lfloor\underline{117} & 1 \\ 2\lfloor\underline{58} & 0 \\ 2\lfloor\underline{29} & 1 \\ 2\lfloor\underline{14} & 0 \\ 2\lfloor\underline{7} & 1 \\ 2\lfloor\underline{3} & 1 \\ 2\lfloor\underline{1} & 1 \end{array}\right\} = 11101010_2$$

$$234_{10}: \quad \left.\begin{array}{r|l} 8\lfloor\underline{234} & 2 \\ 8\lfloor\underline{29} & 5 \\ 8\lfloor\underline{3} & 3 \end{array}\right\} = 352_8$$

$$234_{10}: \quad \left.\begin{array}{r|l} 16\lfloor\underline{234} & 10 \\ 16\lfloor\underline{14} & 14 \end{array}\right\} = EA_{16}$$

1.3.2 Binary, Octal and Hexadecimal to Decimal

$$101011_2: \quad \begin{aligned} 1 \times 2^0 &= 1 \\ 1 \times 2^1 &= 2 \\ 0 \times 2^2 &= 0 \\ 1 \times 2^3 &= 8 \\ 0 \times 2^4 &= 0 \\ 1 \times 2^5 &= \underline{32} \\ &= 43_{10} \end{aligned}$$

$$724_8: \quad \begin{aligned} 4 \times 8^0 &= 4 \\ 2 \times 8^1 &= 16 \\ 7 \times 8^2 &= \underline{448} \\ &= 468_{10} \end{aligned}$$

$$ABC_{16}: \quad \begin{aligned} C \times 16^0 &= 12 \\ B \times 16^1 &= 176 \\ A \times 16^2 &= \underline{2560} \\ &= 2748_{10} \end{aligned}$$

1.3.3 Binary to Octal and Octal to Binary

$$1011010111_2 = 1327_8 : \quad \begin{array}{cccc} 1 & 011 & 010 & 111 \\ 1 & 3 & 2 & 7 \end{array}$$

$$705_8 = 111000101_2 : \quad \begin{array}{ccc} 7 & 0 & 5 \\ 111 & 000 & 101 \end{array}$$

1.3.4 Binary to Hexadecimal and Hexadecimal to Binary

$$1010111011_2 = 2BB_{16} : \quad \begin{array}{ccc} 10 & 1011 & 1011 \\ 2 & B & B \end{array}$$

$$10AF_{16} = 1000010101111_2 : \quad \begin{array}{cccc} 1 & 0 & A & F \\ 0001 & 0000 & 1010 & 1111 \end{array}$$

1.3.5 Octal to Hexadecimal and Hexadecimal to Octal

Conversions between octal and hexadecimal can be performed by first converting to binary.

1.4 Integers in Binary

Counting in powers of 2.

Prefix	Power of 2	Value
kibi	2^{10}	1024
mebi	2^{20}	1048576
gibi	2^{30}	1073741824
tebi	2^{40}	1099511628000

1.4.1 Overflow

In a computer, an integer is represented by a fixed number of bits. The maximum value that can be stored in an unsigned integer of n bits is $2^n - 1$. It is possible that the addition of two n bit numbers yields a result that requires $n + 1$ bits.

In Java, the 'overflow' bits are lost with no error. The remaining bits give the wrong answer. It is important to make sure the data type used is big enough for the values it will represent.

1.4.2 Two's Complement

In 8 bit arithmetic, $255 + 1$ appears to be 0 due to overflow. In this case, 255 is behaving like -1. This leads to the 'two's complement' representation of negative integers in binary.

The two's complement representation of $-x$ is equivalent to the unsigned binary representation of $2^n - x$. Another method to find the representation is to flip all the bits of the binary representation of x and add 1 to the least significant bit.

Using two's complement, a signed binary integer of n bits can hold any value from -2^{n-1} to $2^{n-1}-1$, inclusive. The most significant bit represents -2^{n-1}. If the number of bits in a signed number is increased, the new bits are given the same value as the most significant bit. This is known as sign extension.

In Java, all integers are signed.

1.5 Real Numbers in Binary

1.5.1 Fixed Point Decimal to Binary

The integral part is converted as usual. The fractional part is converted by doubling as follows.

$$
\begin{aligned}
0.537_{10} = 0.100010_2 : & \\
0.537 \times 2 &= \underline{1}.074 \\
0.074 \times 2 &= \underline{0}.148 \\
0.148 \times 2 &= \underline{0}.296 \\
0.296 \times 2 &= \underline{0}.592 \\
0.592 \times 2 &= \underline{1}.184 \\
0.184 \times 2 &= \underline{0}.368
\end{aligned}
$$

1.5.2 Fixed Point Binary to Decimal

$$
\begin{aligned}
1101.0101_2 : & \\
1 \times 2^{-4} &= \ 0.0625 \\
0 \times 2^{-3} &= \ 0 \\
1 \times 2^{-2} &= \ 0.25 \\
0 \times 2^{-1} &= \ 0 \\
1 \times 2^{0} &= \ 1 \\
0 \times 2^{1} &= \ 0 \\
1 \times 2^{2} &= \ 4 \\
1 \times 2^{3} &= \underline{\ 8 \qquad} \\
&= 13.3125_{10}
\end{aligned}
$$

1.5.3 Floating Point Numbers

Fixed point is convenient and intuitive, but has two major problems.

1. Numerical precision — only values that are multiples of the smallest used power of two can be represented.
2. Numerical range — fractional precision comes at the expense of numerical range.

Floating point representation in binary is similar to scientific notation in decimal. In binary, real numbers can be represented in the form $\pm m \times 2^e$. Floating point numbers consist of a sign bit ($\pm$, 0 for positive or 1 for negative), a mantissa (m, the significant bits) and an exponent (e, two's complement signed binary).

S	Offset exponent, e'	Normalised mantissa, m'

The stored representation of a floating point number.

Since the mantissa is normalised (unless the value is small enough to represent without an exponent), the leading bit is almost always 1. It is therefore omitted from the stored representation. The exponent has a bias (offset) of $2^{n-1} - 1$ added to it for engineering purposes.

Floating point data types in Java.

Type	Bits				Bytes	Exponent bias
	Sign	Mantissa	Exponent	Total		
`float`	1	23	8	32	4	127
`double`	1	52	11	64	8	1023

With the 52 bit mantissa of a `double` (and the additional hidden bit), $2^{53} \approx 8 \times 10^{15}$. Hence, the `double` data type can be used to represent 15 significant decimal digits.

In Java, special values are returned for floating point overflow and other unusual circumstances. For example, if the result is too large for the `double` data type, `Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY` are returned. If the result is indistinguishable from zero but known to be negative, `-0.0` is returned. If the result is not a real number, `Double.NaN` is returned.

1.5.4 Numerical Precision

Floating point arithmetic loses accuracy in its less significant digits. This is an issue even in values of type `double`.

It is a bad idea to use floating point representation for money — i.e. with integral pounds and fractional pence — as this will result in rounding errors. Taking £0.10 as an example, $0.10_{10} = 0.0\overline{0011}_2$.

Since currency is inherently integral, integer types with suitable range, such as `int`, `long` or `java.math.BigInteger`, should be used to represent multiples of the smallest denomination.

2 Introduction to Computer Architecture

2.1 Anatomy of a Computer System

A computer is a complex system (machine) that can be instructed to carry out sequences of arithmetic or logical operations automatically via computer programming.

A computing device must be able to

- load a program (input interface),
- process instructions in the correct order (track progress, storage and decoding of instructions),
- access data according to its instructions (local storage),
- perform computations (calculation 'engine'),
- make decisions according to its computations (control mechanism), and
- send results to an external device (output interface).

Thus, all computers, regardless of their implementing technology, have five basic subsystems.

1. Memory
2. Control unit
3. Arithmetic logic unit (ALU)
4. Input unit
5. Output unit

2.1.1 Memory

Memory locations have finite capacity. Data may not fit in a memory location. Blocks of four or eight bytes are used so often as a unit that they are known as memory 'words'.

Computer memory is known as random access memory (RAM). This simply means that the computer can refer to memory locations in any order.

2.1.2 Control unit

The control unit of a computer is where the fetch/execute cycle occurs. It fetches a machine instruction from memory and performs other operations of the fetch/execute cycle accordingly.

2.1.3 Arithmetic Logic Unit (ALU)

The arithmetic logic unit carries out each machine instruction with a separate circuit. It contains various circuits for arithmetic and logic, such as addition, subtraction, multiplication, comparison and logic gates.

2.1.4 Input and Output Units

These components are the wires and circuits through which information travels into and out of a computer. Without input or output, a computer is useless.

Peripherals connect to the computer through input/output (I/O) ports. They are not considered parts of the computer.

2.2 The Fetch-Execute Cycle

A program is loaded into memory and the address of the first instruction is placed in the program counter (PC).

The fetch-execute cycle proceeds as follows.

1. Instruction fetch (IF)
2. Instruction decode (ID)
3. Data fetch (DF) / operand fetch (OF)
4. Instruction execution (EX)
5. Result return (RR) / store (ST)

2.2.1 Instruction Fetch (IF)

The instruction at the memory address given by the PC is copied to the instruction register of the control unit. The PC is incremented to point at the next instruction to be fetched.

2.2.2 Instruction Decode (ID)

The ALU is prepared for the operation specified by the instruction. The decoder finds the addresses of the instruction operands. These addresses are passed to the ALU circuit that fetches the operands from memory in the next stage. The decoder also finds the destination address for the result. This is passed to the RR circuit.

2.2.3 Data fetch (DF)

The operands are copied from the specified memory addresses into the ALU circuits. The data remains in memory and is not destroyed.

2.2.4 Instruction Execute (EX)

The ALU performs the operation on its operands. The result is held in its circuitry.

2.2.5 Result Return (RR)

The result of the EX stage is stored at the specified destination memory address. The cycle begins again.

2.3 Machine Instructions

A computer "knows" very few instructions. The decoder hardware in the control unit recognises, and the ALU performs, of the order of 100 instructions. Everything that a computer does must be reduced to some combination of these primitive instructions. Computers can carry out millions of these instructions per second.

3 Instructions, Assembly Language and Machine Code

3.1 The von Neumann Architecture

3.1.1 Central Processing Unit (CPU)

The central processing unit (CPU) consists of a control unit, ALU and registers. These registers include

- the program counter (PC) (holds the address of the next instruction),
- the instruction register (holds the instruction currently being decoded or executed),
- the address register (holds a memory address from which data will be fetched, or to which data will be returned), and
- the accumulator (holds temporarily the results of ALU computations).

3.1.2 System Bus

The system bus includes

- a control bus (carries commands from the CPU and returns status signals from devices),
- an address bus (carries information about the device with which the CPU is communicating, namely physical memory addresses), and
- a data bus (carries the data being processed).

3.1.3 Memory

In the von Neumann architecture, the memory contains both program instructions and data.

3.1.4 Clock

A clock cycle is a signal that oscillates between high and low. It is used to coordinate actions. The rate of the fetch-execute cycle is determined by the clock. Instructions begin execution on the rising edge of the signal. The time between rising edges is known as the 'clock period'. The time between the rising and falling edges of the signal is known as the 'clock width'. The number of clock cycles per second is used to determine the speed of a CPU.

Modern computers attempt to start an instruction on each clock tick. Since each stage of the fetch-execute cycle is handled by separate circuitry, the fetch unit is freed to start the next instruction before the previous instruction is complete. This process is known as 'pipelining'. When the pipeline is filled, five instructions are in process at a time, each at a different stage of the fetch-execute cycle Additionally, one instruction is finished on each clock tick, making it appear as though the computer is running one instruction per tick.

3.2 The Harvard Architecture

The main difference between the von Neumann and Harvard architectures is their storage of instructions and data in memory.

A von Neumann (or 'stored-program') machine stores its instructions and data in a shared memory. This means that an instruction fetch and a data operation cannot occur at the same time because they share a common bus. This is known as the 'von Neumann bottleneck' and can impinge on the performance of the system. However, the architecture allows for self-modifying code that can tune its performance at runtime.

A Harvard machine stores its instructions and data in separate memories. This makes the machine more complex, but allows instruction fetch and data operation to occur at the same time.

3.3 MIPS R4000

MIPS R4000 uses the modified Harvard architecture. Instructions and data are stored in the same memory (as in von Neumann), but they have separate interfaces (as in Harvard).

MIPS R4000 was one of the first 64 bit microprocessors. It features integrated caches (primary on-chip and secondary off-chip), an integrated floating point unit and a deep pipeline. It uses MIPS III — a compact instruction set with a regular format — which makes it easy to convert from high-level code to machine code.

3.3.1 Registers

The CPU contains 32 registers denoted `$0`–`$31` (or `r0`–`r31`). The registers can be either 32 bit or 64 bit wide, depending on the mode of operation. `$0` (`r0`) always stores zero. `$31` (`r31`) is the link register used by jump and link instructions. It should not be used by other instructions.

The ALU takes input from up to two registers and sends output to registers only.

3.3.2 Deep Pipeline

The typical MIPS pipeline consists of the following five stages.

1. Instruction fetch (IF)
2. Instruction decode (ID) and register fetch (RF)
3. Execute (EX)
4. Memory access (MEM)
5. Write back (WB)

MIPS R4000 is super-pipelined; its pipeline contains eight stages rather than the regular five. The additional stages exist due to the extension of the IF and MEM stages to account for cache overheads. When the pipeline is full, eight instructions are in process at a time — i.e. the pipeline is eight-deep.

3.4 Assembly Language

Binary is the only form in which a computer can be given instructions. Computers can be programmed to translate instructions expressed in other forms into binary code. This is known as 'assembly'.

Assembly language is a human-readable alternative to machine language. A computer can scan assembly code and convert words to binary. The binary is assembled into instructions.

High-level languages are compiled to assembly language, which is then assembled into binary.

3.5 Instruction Sets

Every machine has a unique instruction set architecture (ISA). This is the set of primitive instructions that the CPU can execute. MIPS R4000 uses the MIPS III ISA. Each MIPS III instruction has a 32 bit representation. It can be represented as human-readable assembly code, or binary machine code. There are three types of MIPS III instruction.

I-type requires one or two registers and an operand.
R-type requires three registers.
J-type requires an operand.

4 Compilation, Interpretation and an Overview of the Java Virtual Machine (JVM)

4.1 Levels of Programming Languages

High-level programming languages, such as Java, C, C++ and C# are closer to the English language and, therefore, easier for humans to read and write. Code written in these languages is more concerned with problem-solving than implementation. Programmers are less likely to make errors when programming in these languages and do not need to worry about memory management and other complexities.

Low-level languages, such as machine code, use instructions stored in memory (opcodes) and refer to specific locations in memory. Code written in these languages reflects the processes being used rather than the problem being solved. Machine language is difficult for humans to read and write. Since it runs directly on hardware, it is also hardware-specific.

Assembly language is slightly higher than machine code. It uses mnemonics so that it can be more easily read or written by humans. This is used to translate high-level languages to machine code.

4.2 Language Processing Systems

To run on a computer, a program must be supplied in machine code. Programs written in high-level languages are converted to binary through a series of phases.

1. Preprocessor (prepares high-level code for the compiler)
2. Compiler/interpreter (produces assembly/intermediate code)
3. Assembler (produces relocatable machine code)
4. Linker/loader (produces object code)

The preprocessor organises and prepares the source code for the compiler. This includes importing packages, macro processing and file inclusion.

The compiler reads the entire source code in one go and creates tokens. It checks the meaning of these tokens and generates intermediate assembly code. Alternatively, the interpreter reads the source statement by statement, converting each statement to intermediate code that is executed immediately.

The assembler calculates relative addresses for jumps and produces relocatable machine code.

The linker combines assembled parts into a whole. Alternatively, the loader places the code directly into memory.

4.3 High-Level Program Execution

A program written in a high-level language can be run in two different ways.

1. Compiled into a program in the native machine language of the target machine
2. Directly interpreted and executed via simulation on the target machine

4.4 Compilation

A compiler converts source code into assembly, object or machine code that does the same thing as the original. Object code is usually relocatable, so it can later be linked or loaded.

This is done just once for each program. Hardware features can be exploited to optimise object code so that it will run more quickly.

However, this is more difficult than interpretation, as the compiler must understand the entire program in order to convert it. Compilers are also hardware-dependent, so cannot run on different platforms.

A compiler runs on the same platform as the target machine. A cross-compiler can produce code that will run only on a separate platform.

A compiler can be divided into a front-end and a back-end. Both perform their operations in a sequence of phases that each generate a data structure to be used by the following phase.

Front-end:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generation

Back-end:

5. Optimisation
6. Code generation

4.4.1 Lexical Analysis

During lexical analysis, the compiler breaks the source code into meaningful words known as 'lexemes' and generates a sequence of tokens from the lexemes. A lexeme is a word recognised by the compiler according to the language specification. This includes keywords, identifiers, operators etc. A token is an object describing a lexeme. Along with the value of the lexeme, it includes information about the type of the lexeme (keyword, identifier, operator etc.).

4.4.2 Syntax Analysis (Parsing)

During syntax analysis, the compiler interprets the structure of the source code. This is known as 'parsing' and involves grouping tokens into higher-level constructs. This phase produces an abstract syntax tree (AST).

This is also the phase where syntax errors are detected. If no error is found, the compiler continues to the semantic analysis phase.

4.4.3 Semantic Analysis

During semantic analysis, the compiler interprets the meaning of the AST. The compiler checks that the program is consistent with the rules of the source language.

The compiler can deal with ambiguity in a number of ways.

- Type inference (the compiler annotates nodes in the AST with inferred type information),
- Type checking (the compiler checks that all values assigned to variables are of the correct type).
- Symbol management (the compiler uses a symbol table to determine whether variables have been declared or whether multiple variables with the same name exist in the same scope).

Semantic analysis produces an annotated AST.

4.4.4 Intermediate Code Generation

During intermediate code generation, the compiler produces code in an intermediate language between that of the source code and machine language.

4.4.5 Optimisation

During optimisation, the compiler simplifies or removes unnecessary code. This allows the program to run more quickly or use fewer resources.

4.4.6 Code Generation

During code generation, the compiler maps the optimised intermediate code to the target machine language.

4.5 Interpretation

An interpreter is a program that follows the source code and performs appropriate actions accordingly. A CPU can be viewed as a hardware implementation of an interpreter for machine code.

Interpreters begin execution immediately and facilitate interactive debugging and testing. A user can read and modify the values of variables and invoke procedures from the command line. Interpreted languages are not hardware-dependent. However, interpreted programs have slower execution than compiled programs.

Compilers are compute-intensive and require more preparation time. Additionally, compiled programs are hardware-dependent. However, they can run very quickly.

4.6 Combined Compilation and Interpretation

Combined compilation and interpretation is a method by which programs are compiled to an intermediate language that can be interpreted efficiently. Execution of programs produced by this method is slower than pure compilation, but quicker than pure interpretation.

Compilation for any platform requires only a single compiler that is independent of the target CPU. Interpretation of the intermediate language is delegated to the target CPU.

Java was conceived as a language that uses combined compilation and interpretation. The `javac` compiler converts `.java` source code to `.class` bytecode, which is interpreted by the Java Runtime Environment (JRE) on the Java Virtual Machine (JVM).

4.7 Compilation and Execution on Virtual Machines

A virtual machine executes an instruction stream using software rather than hardware. Virtual machines emulate hardware using software and are, therefore, hardware-independent. Virtual machines are used in languages such as Java, Pascal and C#.

Java compilers generate bytecode that is interpreted by the JVM. This requires a JVM to exist on the target machine. The JVM may translate portions of bytecode to machine code at runtime via just-in-time (JIT) compilation if it finds those portions are used frequently.