Imperial College London

ISA and Compilers: Year 2

Encoding and MIPS architecture

Xin Wang

October 3, 2020

Abstract

In computer science, an instruction set architecture (ISA) is an abstract model of a computer. It defines the supported data types, the registers, the hardware support for managing main memory, and the input/output model of a family of implementations of the ISA. Most importantly, it specifies the behavior of machine code running on implementations of that ISA.

MIPS (Microprocessor without Interlocked Pipelined Stages) is a reduced instruction set computer (RISC) ISA developed by MIPS Technologies. Often studied in university as later architectures are influenced by the ISC architecture used in MIPS.

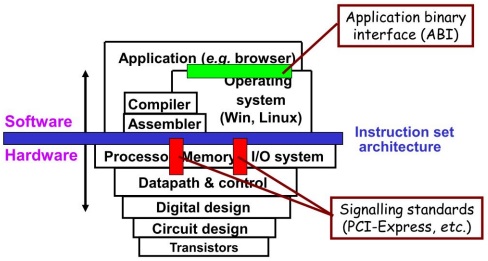
Contents

1 Introduction

To command a computer's hardware, **instructions** need to be issued i.e. words of a computer's language. The vocabulary is called an **instruction set**.

> **Instruction set**: The vocabulary of commands understood by a given architecture.

The Instruction Set Architecture (ISA) is the part of the processor that is visible to the programmer or compiler writer. The ISA serves as the boundary between software and hardware. It translates the high-level languages into machine code that is compatible with the hardware.



In practice, computer hardware languages are quite similar. This is why MIPS architecture is studied as it can easily be applied to the other types of architectures. MIPS are preferred over ARM as there are simply much more academic source material for MIPS than there are for ARM.

The similarity of instruction sets occurs because of two reasons:

- All computers are constructed from hardware technologies based on similar fundamental principles.
- There are few basic operations that all computers must provide e.g. LDA and JMP.

Computer designers have a common goal: Find a language that is:

- Easy to use to implement the hardware.
- Maximising the performance.
- Minimising the cost to implement and the energy used.

By learning how to represent instructions as a compiler does, it includes an important discovery of computing: the **stored-program concept**.

> **Stored-program concept**: The idea that instructions and data of many types can be stored in memory as numbers, leading to the concept of **stored-program computer**.

2 Operations of the computer hardware

Computer instructions are composed of the following: **Opcode** + **Operand**.

Every computer must be able to perform fundamental arithmetic operations thus it is a good starting point to understand a particular architecture.

In regards to basic addition, the MIPS assembly language notation:

```
add a, b, c
```

instructs a computer to do the following:

1. add the two variables `b` and `c`
2. put the sum in `a`

This notation is very rigid in that each MIPS arithmetic instruction performs only one operation and **must always have exactly three variables**. Requiring every instruction to have exactly three operands simplifies design because hardware for a variable number of operands is more complicated than hardware for a fixed number of operands. This illustrates the first of three underlying **principles of hardware design**:

> **Hardware design principle 1**: Simplicity favors regularity.

Example 1: This segment of a C program contains the five variables `a`, `b`, `c`, `d`, and `e`:

```
a = b + c;
d = a - e;
```

Translate from C to MIPS assembly language instructions like a compiler:

```
add a, b, c
sub d, a, e
```

3 Operands of computer hardware

> **Operand**: The part of a computer instruction which specifies what data is to be manipulated or operated on, while at the same time representing the data itself.

Unlike high-level languages, the operands of arithmetic instructions are restricted; the operands must be from a limited number of **registers** i.e. special locations built directly into the hardware.

Registers: A temporary storage area built into the hardware, can be seen as the building blocks of the computer.

The standard size of a register in the MIPS architecture is **32 bits**, often grouped together as a **word**.

Word: The natural unit of access in a computer, usually a group of 32 bits that corresponds to the size of a register in the MIPS architecture.

As there are a limited number of registers, usually 32 on current computers like MIPS, the three operands of MIPS must each be chosen from one of the **32 32-bit registers**.

The reason for the limit of 32 registers of each 32-bit is because a very large number of registers may increase the clock cycle time simply with electronic signals taking longer to travel further. This is the second design principle of hardware technology:

Hardware design principle 2: Smaller is faster.

The MIPS convention to represent a register is to **use two-character names following a dollar sign** e.g. `$s0`, `$s1`, ..., `$s31`.

Example 1: It is the compiler's job to associate program variables with registers. Given the assignment statement:

```
f = (g + h) - (i + j);
```

The variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4` respectively. What is the compiled MIPS code?

Note that temporary registers `$t0` and `$t1` are used.

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1
```

3.1 Memory operands

Programming languages uses simple variables that contain single data elements to build more complex data structures e.g. arrays. These complex data structures can contain many more data elements than registers can support.

The processor can keep only a small amount of data in registers but the computer memory can contain much more data elements. Hence, complex data structures are kept in memory. MIPS includes **data transfer instructions** that transfer data between memory and registers by supplying the memory address to be

accessed. Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0.

A MIPS data transfer instruction only reads one operand or writes one operand, without operating on it. The data transfer instruction is called **load**. MIPS uses `lw` for *loading word* and format of the load instruction is:

1. Name of the operation.
2. The register to be loaded.
3. A constant and register which the sum forms the memory address.

Because the compiler allocates data structures like arrays and structures to locations in memory, the compiler can then place the proper starting address into the data transfer instructions.

Example 1: Assuming that `A` is an array of 100 words and that the compiler has associated the variables `g` and `h` with the registers `$s1` and `$s2` as before. Also assume that the starting address or base address of the array is in `$s3`. Compile this C assignment statement:

```
g = h + A[8];
```

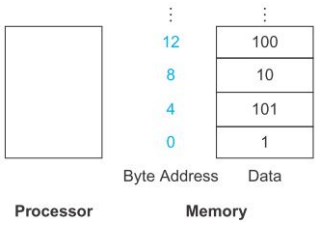
1. Transfer A[8] to a register. The address of this array element is the sum of the base of the array A, found in register `$s3`, plus the number to select element 8.

```
lw $t0, 8($s3) # $t0 gets A[8]
```

2. Add content of `$s2` to `$t0` and put the sum in the register corresponding to `$s1`.

```
add $s1,$s2,$t0 # g = h + A[8]
```

Due to the fundamental nature of programs, 8-bit bytes are useful and virtually all architectures address individual bytes. Therefore, the address of a word would match one of the address the 4 bytes within the word and addresses of sequential words would differ by 4. In MIPS, words must start at addresses



that are multiples of 4. This common architecture restriction is known as an **alignment restriction**.

Alignment restriction: A requirement that data be aligned in memory on natural boundaries.

The instruction complementary to load is traditionally called **store**. It copies data from a register to memory and the format of a store is similar to that of a load:

1. The name of the operation: `sw`
2. The register to be stored
3. Offset to select the array element and the base register

Registers take less time to access and have higher throughput than memory, making data in registers both faster to access and simpler to use. Accessing registers also uses less energy than accessing memory. It is these reasons that to achieve highest performance and conserve energy, an instruction set architecture must have a sufficient number of registers, and compilers must use registers efficiently.

3.2 Constant and immediate operands

Many times a program will use a constant in an operation, many MIPS arithmetic instructions have a constant as an operand. By including constants inside arithmetic instructions, operations are much faster and use less energy than constants loaded from memory.

MIPS uses the add immediate `addi` instruction:

```
addi $s3,$s3,4 # $s3 = $s3 + 4
```

Since MIPS supports negative constants, there is no need for subtract immediate in MIPS.

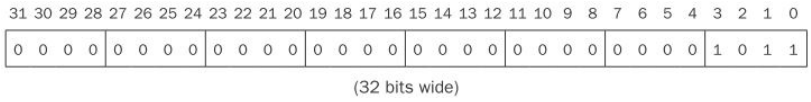
The **constant zero** has another role which is to simplify the instruction set by offering useful variations. MIPS dedicates a register `$zero` to be hard-wired to the value zero. Using frequency to justify the inclusions of constants is another example of the great idea of making the **common case fast**.

4 Signed and unsigned numbers

Numbers are kept in computer hardware as base 2 numbers.

Least significant bit: The rightmost bit in a MIPS word.

Most significant bit: The left most bit in a MIPS word.



The MIPS word is 32 bits long, so can represent 2^{32} different 32-bit patterns.

Hardware can be designed to add, subtract, multiply, and divide these binary bit patterns. If the number that is the proper result of such operations cannot be represented by these rightmost hardware bits, **overflow** has occurred.

Computer programs calculate both positive and negative numbers, so a system to distinguishes the positive from the negative is needed:

1. **Sign and magnitude**: Simplest system but adders for sign and magnitude will need an extra step to set the sign because it is not known in advance what the proper sign will be. Also, a separate sign bit means that sign and magnitude has both a positive and a negative zero, which can lead to problems.

2. **Two's complement**: Leading 0s mean positive, and leading 1s mean negative. It has the advantage that all negative numbers have a 1 in the most significant bit i.e. the sign bit. Hardware needs to test only the sign bit to see if a number is positive or negative.

Refer to Year 1 resources for revision on two's complement.

5 Representing Instructions in the Computer

Instructions are kept in the computer as binary and **can be represented as numbers**. Each individual piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the entire instruction.

There is a convention to mapping register names into a numbered format since registers are referred to in instructions and instructions are numbers as mentioned earlier. This is covered in this chapter. The following shows the real

MIPS language version of the instruction represented symbolically, as a decimal and binary.

```
add $t0,$s1,$s2
```

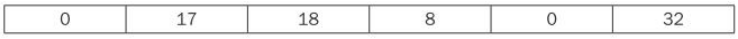


Figure 1: Decimal format of the instruction

Figure 2: Binary format of the instruction

Each of these segments of an instruction is called a **field**.

- First (0) and last (32) field: Indicate to MIPS computer that this instruction performs addition.
- Second (17) field: Number of the register that is the **first source operand** of the addition operation i.e. $17 = \$s1$
- Third (18) field: The other source operand for the addition i.e. $18 = \$s2$
- Fourth 8 field: The number of the register that is to receive the sum i.e. $8 = \$t0$
- Fifth 0 field: Usually deals with shift instructions, unused in this instruction so set to 0.

Notice that all MIPS instruction takes exactly 32 bits, this is in keeping with the design principle that simplicity favors regularity. This layout of the instruction is called the **instruction format**.

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

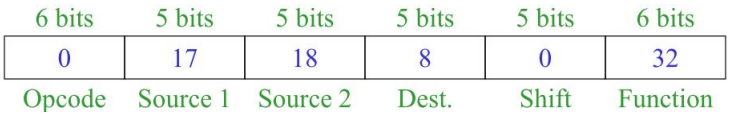
To distinguish it from assembly language, the numeric version of instructions is called **machine language** and a sequence of such instructions **machine code**.

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

Due to long binary numbers, using a higher base that can convert easily into binary, is much more efficient. Since all computer data sizes are multiples of 4, hexadecimal numbers are used.

5.1 MIPS fields

MIPS fields are given names:



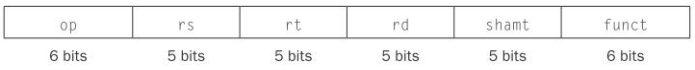
A problem occurs when an instruction needs a longer field than those shown above. For example, with the load word instruction must specify two registers and a constant. If the address were to use one of the 5-bit fields in the format above, the constant used to select elements from arrays or data structures would be limited to only 2^5 or 32. This 5-bit field is too small to be useful. This leads to the fi nal hardware design principle:

> **Hardware design principle 3**: Good design demands good compromises.

The compromise chosen by the MIPS designers is to keep all instructions the same length and requiring **different kinds of instruction formats** for different kinds of instructions.

Due to this compromise, MIPS has three types of instruction formats:

1. **R-type** (Register):

   Used by arithmetic instructions e.g. `add` and `sub`.

2. **I-type** (Immediate):

   Used by data transfer instructions e.g. `addi`, `lw` and `rw`. The 16-bit address means a load word instruction can load any word within a region of $\pm 2^{15}$ or $32,768$ bytes of the address. Similarly, add immediate is limited to constants no larger than $\pm 2^{15}$.

3. **J-type** (Jump):

   The MIPS jump instructions have the simplest addressing. The J-type, which consists of 6 bits for the operation field and the rest of the bits for the address field.

Although multiple formats complicate the hardware, the complexity can be reduced by keeping some parts of the formats similar.

6 Logical operators

Although the first computers operated on full words, it was eventually clear that it was useful to operate on **fields of bits within a word** or even on **individual bits**. These instructions are called logical operations.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

Figure 3: Note: MIPS implements NOT using a NOR with one operand being zero.

The first class of such operations is called **shifts** - shift left logical `sll` and shift right logical `srl`. These instructions move all the bits in a word to the left or right and filling the emptied bits with 0s. Shift left logical provides the equivalent of multiplying by 2 per shift.

For example,

```
sll  $t2,$s0,4  # reg $t2 = reg $s0 << 4 bits
```

where register `$s0` contained:

```
0000 0000 0000 0000 0000 0000 0000 1001 = 9
```

and the instruction to shift left by 4 was executed, the new value would be:

```
0000 0000 0000 0000 0000 0000 1001 0000 = 144
```

Shift instructions are **R-type** instructions and uses the **shamt** field of the R-format. Hence, the machine language version of the instruction above is:

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

The second class are called **bit-by-bit operators** - AND and OR. AND is used to isolate fields since it leaves a 1 in the result only if both bits of the operands are 1. Such a bit pattern in conjunction with AND is called a **mask**, since the mask "conceals" some bits.

For example,

```
and $t0,$t1,$t2 # reg $t0 = reg $t1 & reg $t2
```

where register `$t2` contains:

```
0000 0000 0000 0000 0000 1101 1100 0000
```

and register `$t1` contains:

```
0000 0000 0000 0000 0011 1100 0000 0000
```

the value of register `$t0` would be:

```
0000 0000 0000 0000 0000 1100 0000 0000
```

The final logical operation is a **contrarian** - NOT takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa. In keeping with the three-operand format, the designers of MIPS decided to use the instruction NOR instead of NOT. If one operand is zero, then it is equivalent to NOT.

If the register `$t1` is unchanged from the preceding example and register `$t3` has the value 0, the result of the MIPS instruction:

```
nor $t0,$t1,$t3 # reg $t0 = NOT(reg $t1 | reg $t3)
```

the value in register `$t0`:

```
1111 1111 1111 1111 1100 0011 1111 1111
```

6.1 Instructions for Making Decisions

The ability to make decisions is the distinguishing feature of a computer from a calculator. Based on the input data and the values created during computation, different instructions execute.

MIPS assembly language includes two decision-making instructions, similar to an *if* statement with a *go to*.

1. **Branch if equal**: `beq register1, register2, L1`

   Go to the statement labeled `L1` if the value in `register1` **equals** the value in `register2`.

2. **Branch if not equal**: `bne register1, register2, L1`

   Go to the statement labeled `L1` if the value in `register1` **does not equal** the value in `register2`.

6.2 Loops

Decisions are important both for choosing between two alternatives found in *if* statements and for iterating a computation found in loops.

There are three distinct types of loops in high level programming: **do/while**, **while** and **for**. They are all functionally identical, any **for** loop can be turned into a **while** loop with a bare minimum of effort.

Given a traditional loop in C:

```
while (save[i] == k)
    i += 1;
```

where `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array save is in `$s6`.

What is the MIPS assembly code corresponding to this C segment?

1. Load `save[i]` into a temporary register `$t1`. Before `i` is added to the base of array to form the address to load, multiply the index `i` by 4 due to the byte addressing problem where each address has 4 bits.

2. To get the address of `save[i]`, add `$t1` and the base of `save` in `$s6`:

3. Use that address to load `save[i]` into a temporary register:

4. The next instruction performs the loop test, exiting if `save[i]` $\neq$ `k`:

5. The next instruction adds 1 to `i`:

6. The end of the loop branches back to the while test at the top of the loop. Add the Exit label after the loop.

```
Loop: sll $t1,$s3,2 # Temp reg $t1 = i * 4
    add $t1,$t1,$s6 # $t1 = address of save[i]
    lw $t0,0($t1) # Temp reg $t0 = save[i]
    bne $t0,$s5, Exit # Exit if save[i] != k
    addi $s3,$s3,1 # i = i + 1
    j     Loop # go to Loop
Exit:
```

Sometimes it is useful to see if a variable is less than another variable. MIPS assembly language has an instruction called `slt` (*set on less than*) that compares two registers and sets a third register to:

- 1: if the first is less than the second
- 0: otherwise

```
slt $t0, $s3, $s4   # $t0 = 1 if $s3 < $s4
```

There is an immediate version of the set on less than instruction. To test if register `$s2` is less than the constant 10:

```
slti $t0,$s2,10      # $t0 = 1 if $s2 < 10
```

7 The processor