# CS 305: Computer Architecture

**Amit Rajaraman**

Last updated August 12, 2021

## Contents

# §1. Introduction

## 1.1. Overview

What is a computer? It includes anything from a smartphone to an iPad to a data center. Let us delay what "computer architecture" means to the end of the course and try to figure out what it should mean for now. Historically, the heavy lifting is done by the computer architecture. The computing stack is as follows: we have problems, for which we design algorithms, which we implement in programming languages/compilers, which are run using an operating/database/network system, which require computer architecture to run (which is the focus of this course). Most software companies nowadays build their own processors or chips.
Since the dawn of time (1946 for our purposes), any computer has 5 components – the processor which consists of control and datapath, memory, input, and output. Computer architecture changes over time depending on our requirements.
Abstractions are useful if we only care about *what* we want to do. But if we want to know how or why we are doing something, they are all but useless – so we aim to break them in this course.

**MIPS** is a simple yet expressive instruction set architecture (ISA) which we shall use in this course. It is used even today in embedded devices, routers, modems etc. The ISA provides an interface between hardware and software – it veils complexity through a set of simple instructions.
For example, `a=b+c` might be the C code, which gets converted to `add $t1 $t2 $t3` (assembly language) by the compiler, which in turn gets converted to some binary string (machine language) by the assembler.
Operands can be located either in registers or in memory. Registers (32-bit addresses) are closer to the processor so are easier and cheaper to access, while memory (DRAM addresses) is costlier to access. However, registers are limited – we don't have a lot of them.

## 1.2. Introduction to MIPS

In 1949, *EDSAC* had only 18 machine instructions. Today in 2021, x86 has over 1500 such instructions. Why do we need these instructions? Programmers must know what the processor can/cannot do, and the processor knows what it should do. In a world with no such instructions, we would have to communicate tedious binary strings to perform simple tasks.
Let us zoom into the processor for a moment. We have a set of registers (tens of them – 32 or 64 for example) and an Arithmetic/Logic Unit (ALU). The processor communicates with the memory through the *address bus* (the processor sends the address to the memory) and the *data bus* (the memory sends data to the processor). The processor can directly access the data of registers, whereas memory has to be accessed in two steps (using the two buses).
Let us look at the most basic operation of MIPS: `add`. In `add $1, $2, $3`, `add` is the operation, `$1` is the destination, and `$2` and `$3` are the sources. Each of these three refers to a certain register. Most ALU operations in MIPS have two sources and one destination.
If we want to do `a=b+c-d`, we can do

```
add $t0, $s1, $s2
sub $s0, $t0, $s3
```

Here, `$t0` is a temporary register. We also use constants in our program. For this, MIPS have instructions that end with `i`, where `i` means "`i`mmediate". For example, `addi $s0, $s0, 10` adds 10 (the `i` refers to the fact that 10 is a constant). The constants are represented in 16-bit 2s complement form.
We do not need `subi` since we can use an appropriate constant with `addi` instead.
`$0` or `$zero` is a special register that contains zero. In particular, `a=b` becomes `add $s1, $s2, $zero`. This leads to the *pseudo-instruction* `move $s1 $s2`, which is not an actual instruction (meaning that when running the code, it is translated to a simpler instruction) and is merely used for programming convenience.
Let us next look at a couple of `l`ogical operations:

- `sll` shifts to the `l`eft,
- `srl` shifts to the `r`ight, and

- `and`, `or`, `nor`, `andi`, and `ori` mean the usual thing.

Interestingly, there is no `not` instruction, we instead use `nor` with one operand as `0`. When we are dealing with these instructions, one must understand that we are dealing with 32 raw bits, not a 32-bit number.
Now, how would we store a 32-bit constant in a 32-bit register (the instructions so far take 16-bit numbers)? The operation `lui` loads the first 16 bits of a 32-bit number into a register setting the lower bits all 0. We then perform an `or` of this register with the other 16 bits. So, if we wanted `$t0` to store the 32-bit constant `10101010 10101010 11110000 1111000`, then we can do

```
lui $t0, 0xAAAA
ori $t0, $t0, 0xF0F0
```

The above sequence of operations can be shortened to the pseudo-operation `li $t0 0xAAAAF0F0`.
Note that all constants used in the operations are only 16 bits long.

## 1.3. Operations on Memory

Before 1944, memory only stored the data required to perform an operation. So, the stored program as the binary is stored in memory. Von Neumann then came up with the idea to store instructions in memory as well. We refer to a 4-byte binary string as a "word". Memory is then just a collection of words. The *program counter* (PC) or *instruction pointer* (IP) is a special register that stores the address of the instruction. So, if we have a 32-bit processor and the address are of width 32 bits as well, the processor fetches PC, PC+4, PC+8,... sequentially.
When we want some operation done,

1. the processor gets the instruction from memory,
2. the ALU demands the data using the address through the address bus,
3. the memory responds with the data through the data bus, and
4. the ALU performs the operation.

We can also think of a step 0 where a request is sent based on the PC itself.
Why do we use memory and not registers? Registers are fast because they are limited – the more registers we have, the higher the access time becomes. We shall look at this in more detail later.

How do we access data from memory?
The two primary instructions for this are `lw` (`load word`) and `sw` (`store word`). The former loads data from the memory to the register, while the latter writes data from the registers to the memory.
For example, `lw $t0, 1($a0)` performs `$t0 = Memory[$a0+1]`. The processor sends a "load request" to the address `$a0+1` and the memory responds with the data at that address. Similarly, `sw $t0, 1($a0)` does `Memory[$a0+1] = $t0`.
When we do an `lw` operation, there are two fetches from memory – one for the instruction itself and another for the data. When we do an `add` operation on the other hand, there is only one fetch since there is no memory access.
Let us now move on to decision-making instructions.

- `beq` (`branch equals to`): If we do `beq $t0, $t1, L1`, the PC goes to "label" `L1` (`goto L1`) if `$t0` and `$t1` are equal.
- `bne` (`branch not equals to`): If we do `bne $t0, $t1, L1`, the PC goes to label `L1` if `$t0` and `$t1` are *not* equal. We shall define what a label is later.
- `slt` (`set on less than`): If `t1` and `t2` contain $a$ and $b$, then `slt $t3, $t1, $t2` does the following: if $a < b$, it sets the content of `t3` to $1$ and otherwise, it sets the content to $0$. We also have `slti`, where one of the operands is a constant.

Next, we look at unconditional jumps (as opposed to the conditional jumps we just looked at):

- `j` (jump): It allows us to jump to a label. This instruction loads an immediate into the PC – it can be specified by either an offset or the label (the assembler converts this label to an offset).

  Now, suppose are running some C code wherein we call a function, and return after the successful running of the function. How do we return to the appropriate place after the code finishes running?

- `jal` and `jr` (jump and link and jump register): `jal L1` jumps to the label `L1` which has to be instruction to be executed next and saves the address of the next instruction (PC + 8)[1] in `$ra`. `ra` is a special register that stores the return address. `jr $ra` then returns to the required spot after execution of the intermediate function.

It is possible that some code in the main function accesses, say, register `$R2` and something in a called function accesses register `$R2` as well. How do we coordinate this? What data is really accessed when we access a certain register (does the function then attempt to access main's data?) What is the protocol using which the caller and callee interact? MIPS allows *four* arguments to be passed from the caller to the callee while using `jal`, which uses registers `$a0` through `$a3`. A callee returns upto two values to the caller, using registers `$v0` and `$v1`.
How does this work out when we have nested functions? The data transferred by the second function to the third in the `$a0` to `$a3` registers will overwrite the data transferred by the first to the second! Similarly, the value of `$ra` changes as well, so how do we go back to the original function and fix these issues? We might think of allotting some registers to be used by the caller and some to be used by the callee. Unfortunately, the callee does not know the registers used by callers (which may be many in number), and the caller does not know the callee's plan either. We can't just allot new registers because we do not know how many nested functions are there, and we only have a limited number of registers: In MIPS, there are 32 registers:

| Registers | Number |
|---|---|
| `$Zero` | 1 |
| Return value registers (`$v0`, `$v1`) | 2 |
| Argument registers (`$a0` to `$a3`) | 4 |
| Return address (`$ra`) | 1 |
| Saved registers (`$s0` to `$s7`) | 8 |
| Temporary registers (`$t0` to `$t9`) | 10 |
| Global pointer (`$gp`) | 1 |
| Stack pointer (`$sp`) | 1 |
| Frame pointer (`$fp` or `$t10`) | 1 |
| OS kernel activities (`$k0` and `$k1`) | 2 |
| Assembler (`$at`) | 1 |

The above described problem is known as **register spilling**, we are running out of registers to use.
So where else can we store data? The only other place to store data at all is the memory, so that is what we shall choose!
The stack pointer (`$sp`) points to the stack, which is part of the DRAM. The stack is private to each function call to ensure that other functions do not overwrite its data. `$sp` points to the address where the stack *ends*. The stack grows *downwards*. When we push something to the stack, we decrement the `$sp` by 4 and store the register content in the appropriate place in memory.

`$t0` through `$t9` (registers R8 to R15, R24, and R25) are caller-saved registers. Their values are not preserved across function calls and are said to be *call-clobbered*.
`$s0` through `$s9` (registers R16 to R23) on the other hand are callee-saved registers. Their values are maintained across function calls and said to be *call-preserved*.
Are `$sp` and `$ra` caller- or callee-saved registers?

To handle register spilling, MIPS stores the data of the registers in the stack and moves on. After the function is finished, we pop the data off the stack and change the register content back to the initial state. How does this work in MIPS? To save data to a register, we do

[1] One might expect to jump to PC+4 instead. We shall look at the reason for the 8 later.

```
addi $sp, $sp, -4
sw R4, $sp
```

and to restore data,

```
lw R4, $sp
addi $sp, $sp, 4
```

So when we call a function, all the relevant data (`$ra` in particular) is stored into the stack so that we know where to return to.
The stack is also used to store local variables and data structures for a function along with the return addresses. To look at these, we have a *frame pointer* (`$fp`). It points to the highest address (on the downward growing stack) in the procedure frame and is used to look at local variables and saved registers. It stays there throughout the procedure while the stack pointer moves around. This is just to make life easier for the compiler/programmer. Technically, we do not require it and can make do with the stack pointer alone (with a suitable offset), but that is very inconvenient.

## 1.4. Instruction Decoding

We have seen how MIPS instructions allow the programmer to interact with the processor. How are these instructions decoded?
Recall that instructions are of size 32 bits (in MIPS). How does the processor infer these bits?
Since MIPS has 32 registers, any register can be specified using 5 bits. A 32 bit R-type instruction is split into 4 parts:

- Bits 0 through 15 specify the immediate.
- Bits 16 through 20 specify a register `rt`.
- Bits 21 through 25 specify another register `rs`.
- Bits 26 through 31 specify an operation `op`, such as say `add`, `mul`, `lw`, `bne`, etc.

A general instruction in MIPS has

- an operation code `op`,
- up to three registers `$rs`, `$rt`, and `$rd`,
- `shamt` (`shift amount`) for logical operations, and
- the functionality field `funct`.

Let us look at the value of each of the above parameters for a couple of instructions.

| Instruction | Format | op | rs | rt | rd | shamt | funct | address |
|---|---|---|---|---|---|---|---|---|
| add | R | 0 | reg | reg | reg | 0 | 32 | n.a. |
| sub | R | 0 | reg | reg | reg | 0 | 34 | n.a. |
| addi | I | 8 | reg | reg | n.a. | n.a. | n.a. | constant |
| lw | I | 35 | reg | reg | n.a. | n.a. | n.a. | address |
| sw | I | 43 | reg | reg | n.a. | n.a. | n.a. | address |

The format R means that it is a register-type instruction, while I means that it is an immediate-type instruction.
We see that the `funct` field is what differs `add` and `sub`.
The lowest 16 bits are not used by I instructions since they are replaced by a constant or address, as specified. The op code informs the processor how to treat the last set of fields, as three or one.

In all, the three types of instructions are:

| Type | 31 26 | 25 21 | 20 16 | 15 11 | 10 06 | 05 00 |
|---|---|---|---|---|---|---|
| R | opcode | `$rs` | `$rt` | `$rd` | `shamt` | `funct` |
| I | opcode | `$rs` | `$rt` | immediate | | |
| J | opcode | address | | | | |

Such a format is good since

- We have fixed length instructions which allows for convenient decoding.
- Fields are at the same or similar location.
- All three formats look similar.

Next, let us look at 5 addressing modes of the MIPS ISA.

- Immediate: We have already seen this above. For example, `addi $t0 $t1 5` is in this mode.
- Register: We have seen this as well. For example, `add $t0 $t1 $t2` is in this mode.
- Base: It has an opcode, `$rs`, `$rt`, and an address.
  Here, the address is given by a register. For example, `lw $t1 4($s2)` or `lw $t1 ($s2)`. We possibly add a constant to the address in a register, and access the memory pointed at by this.
- PC-relative: These are essentially the J-type instructions we looked at earlier. The label gives an offset, which is what we jump to. An example of this is `beq $t0 $t1 end`.
- Pseudodirect: It has an opcode and an address.
  An example of this is `jal`. We concatenate the content of the PC and the address provided to get another address, using which we get the operand.

## 1.5. Why ISAs?

### 1.5.1. Microarchitecture v. ISAs

The ISA is the interface between hardware and software. It enables the programmer to not bother about what is going on under the hood at all.
A *microarchitecure* is a specific implementaiton of an ISA (which the programmer cannot see/access). For example, the `add` instruction would come under the ISA, but the specific implementation of an adder (ripple carry/lookahead) comes under the microarchitecture.
We shall look at microarchitecture in more detail in a later part of the course.
The actual working of the ISA can be thought of using a state machine. As long as the new state that is moved to on receiving an instruction satisfies the ISA specifications, it will work. The information held in the processor at the end of an instruction provides context for the next instruction.
Computer architecture as a whole focuses on both ISA and microarchitecture.
Now, should we keep the ISA closer to the hardware or closer to the high-level software? This leads to two schools of thought – the latter being called CISC (Complex Instruction Set of Computers) and the former being RISC (Reduced Instruction Set of Computers).
If we are closer to high-level language, there is a small semantic gap between the two and we have complex instructions. For example, quicksort may be an instruction. An example of a CISC architecture is something like x86.
If we are closer to hardware, there is a large semantic gap and instructions are quite simple. An example of a RISC architecture is MIPS.
Irrespective of whether we use CISC or RISC, an app's instruction count is defined by the compiler *and* the ISA.
x86 is slightly complicated since it converts CISC operations to RISC operations, and generates *microoperations*. It is an intelligent CISC-RISC decoder. This consumes 2% of the chip area.
In any ISA, there are various factors we must take into account:

- Simplicity

- Performance, power, cost, time to market
- RISC v. CISC
- Fixed v. Variable encoding
- Endianness
- Number of registers per instruction
- Code size
- Open sourcedness

#### 1.5.2. ISAs other than MIPS

There are various common ISAs other than MIPS. For example,

- *x86*, which is used by Intel, AMD in laptops, desktops, and servers.
  x86 has thousands of instructions. Some differences are:
  - A single operand can act as both a source and destination. For example `add $s0 $s1` adds `$s0` and `$s1`, then stores the result in `$s0`.
  - An operand can be in memory as well. For example, `add $s0 Mem[$s1]` can be done.
  - Fixed length instructions are not there any more.

  In variable width instructions, different isntructions can take a different number of bytes. It occupies a smaller code footprint and is compact as a result. However, fixed length instructions (like those in MIPS, which occupy 4 bytes) lend themselves to simple decoding despite occupying a larger memory footprint.
- *ARM*, which is used by Arm, Qualcomm, Apple, and Samsung.
  The primary reason one would look at ARM with the *thumb instruction set* is that it is useful in embedded devices (which have low memory), where we require low code density. Instead of using normal 32-bit instructions, we use "thumb" instructions that are only 16 bits. 32-bit registers have 16-bit counterparts.
- More recently, *RISC-V*, an open-source ISA, has been gaining popularity.

RISC was created to enable better compiler control. It was motivated by memory stalls (no work is done during a complex instruction when there is a stall). It enables the compiler to optimize the code, and allows discovery of fine-grained parallelism to reduce the stalls.

## 1.6. Endianness

*Endianness* refers to the byte ordering within a word (or the bit ordering within a byte).

- In the *big endian* notation, the address of the most significant byte is the word address. This is used in MIPS.
- In the *little endian* notation, the address of the least significant byte is the word address. This is used in x86.

To understand this better, consider the following snippet of code.

```
unsigned int i = 1;
char *c = (char*)&i;
printf("%d", *c)
```

If the ISA follows the big endian notation, the output is 0 and if it follows the little endian notation, the output is 1.

## 1.7. Alignment

Ideally, any address should be aligned. For example, suppose we want to access the 4th byte starting from an address `y`. This requires that `y%4` is 0.
For example, MIPS does not allow unaligned access. On the other hand, x86 does not enforce any alignment. It is the compiler's duty to generate aligned or unaligned access.
Alignment is desired to enable faster transfer of data. Indeed, if a 4-byte word is not aligned, then we need to access the memory twice to get the entire word (once for each of the two parts split across). This will become clearer once we understand caches later.
`LOAD`s and `STORE`s need an alignment network that ensures the data loaded/written is aligned. This alignment network differs based on the endianness.