CS 305: Computer Architecture

Amit Rajaraman

Last updated July 26, 2021

§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 such as Microsoft, Amazon, and Facebook 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 $1, $2, $3` (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 $0, $1, $2`, `add` is the operation, `$0` is the destination, and `$1` and `$2` are the sources. Each of these three (`$n`) 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 "immediate". 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 can replace `subi` through a properly chosen constant when using `addi`.
`$0` or `$zero` is a special register that contains zero. In particular, `a=b` becomes `add $s1, $s2, $zero`. This begs the question, why would we add when can move instead? This leads to the *pseudo-instruction* `move $s1, $s2`. This is not an actual instruction and is merely used for programming convenience.
Let us next look at a couple of logical operations:

- `sll` shifts to the left,
- `srl` shifts to the right, 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
```

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 "word" by a 4-byte binary string. 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.