Computer Architecture

William Schultz

September 17, 2022

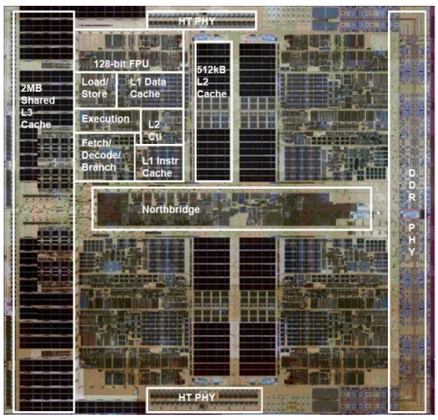
1 Overview: What is a computer?

At a high level, any computer can be viewed as consisting of the following abstract components:

1. **Input**
2. **Output**
3. **Memory**
4. **Processor** = (Datapath + Control)

where the processor can be viewed as consisting of two distinct sub-components. *Datapath* is the hardware responsible for performing all required operations (e.g. ALU, registers, internal buses), and *Control* is the hardware that tells the datapath what to do e.g. in terms of switching, operation selection, data movement between ALU components, etc. [1].

For example, below is a photograph of the quad-core AMD Barcelona processor chip, originally shipped in 2007, with overlaid diagram describing the various subcomponents.



2 Instructions: Language of the Computer

To command a computer you must speak its language. The words of a computer language are called *instructions*, and its vocabulary called an *instruction set*. The *stored-program concept* is the idea that instructions and data of many types can be stored in a computer's memory as numbers.

2.1 Instructions for Making Decisions

Conditional branch instructions are analogous to `if` and `goto` statements in a programming language e.g. the following "branch if equal" instruction

```
beq register1, register2, L1
```

goes to the statement labeled `L1` if the value in `register1` and `register2` are equal.

3 Arithmetic for Computers

Addition, substraction, multiplication, division, floating point, ALU, etc.

4 The Processor

To understand the basics of a processor implementation, we can look at the construction of the datapath and control path for an implementation of the MIPS instruction set. This includes a subset of the core MIPS instruction set:

- The memory reference instructions load word (`lw`) and store word (`sw`)
- The arithmetic-logical instructions `add`,`sub`,`AND`,`OR`, and `slt`
- The instructions branch equal (`beq`) and jump(`j`)

Overall, much of what needs to be done to implement these instructions is the same regardless of the exact class of instruction. For every instruction, the first two steps are identical:

1. Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read.

For example, the diagram below shows a high level, abstract outline of a MIPS processor implementation.

Figure 4.1 shows the high-level view of a MIPS implementation, focusing on the various functional units and their interconnection. Although this figure shows most of the flow of data through the processor, it omits two important aspects of instruction execution.

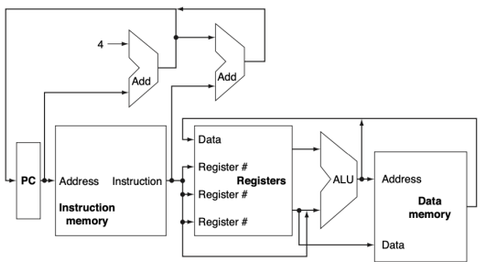


FIGURE 4.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them. All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

4.1 Logic Design Conventions

Note that the datapath elements of a MIPS implementation consists of two different type of logic elements:

- **Combinational**: elements that operate on data values, where their outputs always depend only on the current inputs (i.e. think of them as implementing pure functions)
- **Sequential**: elements that contain some internal *state*. These elements have at least two inputs and one output, where the inputs are:

  1. The data value to be written.
  2. The clock.

  The output from a sequential logic component provides the value that was written in an earlier clock cycle.

A *clocking methodology* defines when signals can be read and when they can be written. We can assume an *edge-triggered clocking* methodology, which means that any values stored in a sequential logic element are updated only on a clock edge.

Since state (i.e. sequential) elements are the only ones that can store values, any collection of combinational logic must have its inputs come from a set of state elements and its outputs written into a set of state elements The inputs are values that were written in a previous clock cycle, while the outputs are values that can be used in a following clock cycle.

4.2 Pipelining

TODO.

5 The Memory Hierarchy

In an ideal world, we would have an infinitely large and fast memory for our computer, but this is not feasible in practice, since fast memory is costly. So, instead, we simulate the illusion of an infinite memory by using a *memory hierarchy*. Essentially, a progressively larger and slower series of caches that serve as memory for the processor.

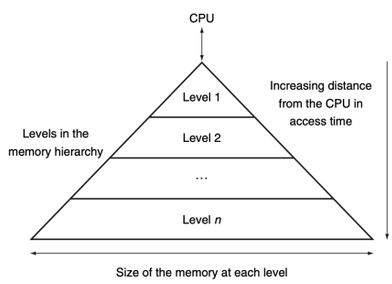


FIGURE 5.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size. This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level *n*. Maintaining this illusion is the subject of this chapter. Although the local disk is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy.

Note that the *principle of locality* underlies the way that programs operate. That is, the assumption is that programs access a relatively small portion of their address space at any instant of time. There are two different types of locality:

- **Temporal locality**: if an item is referenced at some point in time, it is likely to be referenced again soon.
- **Spatial locality**: if an item is referenced, other items that are nearby are likely to be referenced soon.

We make us of this to construct the memory hierarchy from a series of *caches*, that get progressively faster and smaller as they get closer to the actual processor.

References

[1] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Revised Fourth Edition, Fourth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2011.