Computer Architecture

William Schultz

September 19, 2022

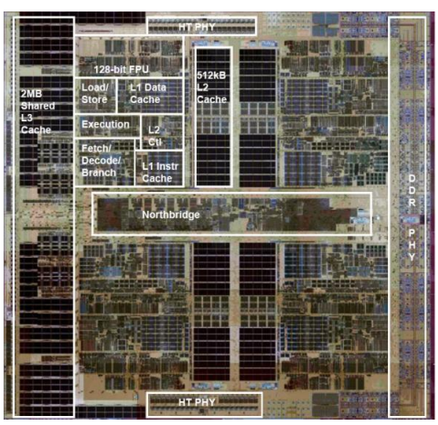
1 Overview: What is a computer?

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)

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].

As a concrete 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.

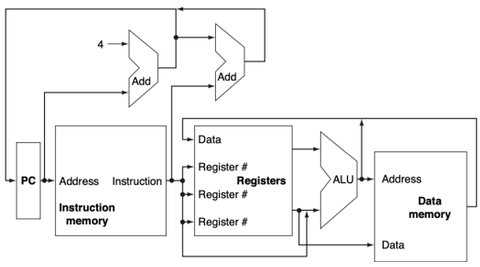


FIGURE 4.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them. All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register 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

MIPS instruction execution classically takes five steps:

1. **Fetch** instruction from memory.
2. **Read** registers while **decoding** the instruction.
3. **Execute** the operation to calculate an address.
4. Access an operand in data **memory**.
5. **Write** the result into a register.

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*. That is, a progressively larger and slower series of caches that serve as memory for the processor.

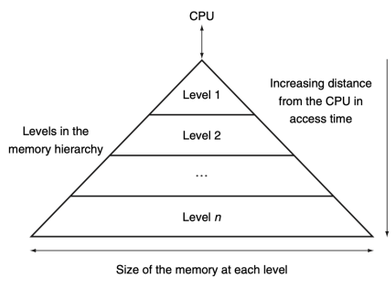


FIGURE 5.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size. This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have 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 use 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.

A memory hierarchy may consist of multiple levels, but data is copied only between two adjacent levels at a time, so we can consider only two levels when describing the caching mechanisms. There is an *upper level* (faster and closer to the processor) and a *lower level*. The smallest unit of information that can be either present or absent from any cache level is typically referred to as a cache *block* or *line*.

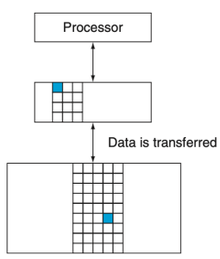


FIGURE 5.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level, the unit of information that is present or not is called a *block* or a *line*. Usually we transfer an entire block when we copy something between levels.

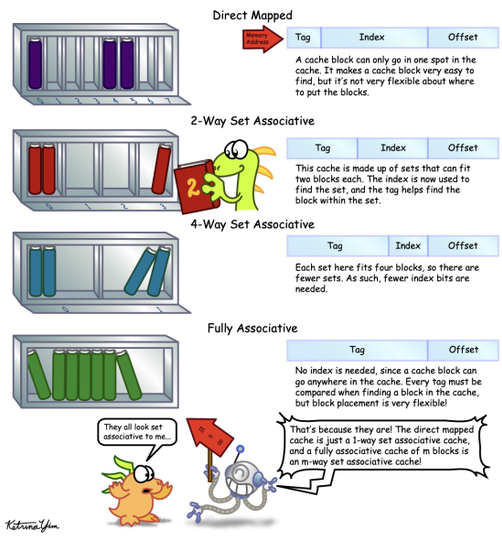
5.1 Cache Basics

In a basic scenario, we can imagine that a processor issues memory requests that are each one word (e.g. 32 bits) and the blocks in the cache are also one word. If a cache contains words $X_1, \ldots, X_{n-1}$ prior to a request for a word X_n not in the cache, then a cache miss occurs and X_n is brought into the cache.

When servicing a processor request, we need to know (1) is the data item in the cache and, if so, (2) how do we find it? The simplest approach is to simply map each word to a locatino in the cache based on its address, which is known as *direct mapping*. This is essentially a simple hash based addressing scheme. With this approach, however, there may be collisions in addresses that map to the same cache location, so we need to deal with this. One approach is to add a set of *tags* to the cache. Basically, for each cache entry, we add a tag that contains the bits of the address that were not included in the hash function, so that we can disambiguate between two addresses that may map to the same cache location. In practice we may also include a *valid* bit that tells whether a particular cached value is currently valid to use or not.

Set Associative Caches

An alternative to the direct mapping approach is to make the cache *set associative*. So, instead of giving every address exactly one location in the cache, we give it n possible locations, which we call n-way set associative. If the cache can hold m entries then a m-way set associative cache is also referred to as *fully associative*. In this other extreme, it means that a block can be placed anywhere in the cache, but then comes with the tradeoff that finding a block is more expensive, since we may need to search through every block to find it.



Note that allowing for more potential locations for a block can decrease contention for blocks, since if there are only a few unused blocks, a new request is then free to pick any unused block. This is in contrast to a direct mapped scheme, where choice of block is completely determined by the address mapping function. Also, in an associative cache, we have a choice of where to place the new block, so potentially a choice of which block to kick out of the cache. The most commonly used scheme is *least recently used* (LRU) i.e. we remove blocks that were unused for the longest time.

5.2 Parallelism and Memory: Cache Coherence

For multicore processors, multiple processors likely operate on a common physical address space. If different processors have their own caches, though, then this means that the view of memory held by different processors may be mismatched i.e. two processors see different values for a given memory location. This issue is generally referred to as the *cache coherence problem*.

Informally, we might want to define a memory system as being coherent if any read of a data item returns the most recently written value of that data item. This is a bit too simplistic, though, and this definition contains two different aspects of memory system behavior. The first, *coherence*, defines *what values* can be returned by a read. The second, called *consistency*, determines *when* a written value will be returned by a read. Considering coherence first, we say that a memory system is coherent if

1. A read by processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.

2. A read by a processor P_1 to a location X that follows a write by another processor P_2 to X returns the value written by P_2 if the read and write are "sufficiently separated" in time and no other writes to X occurred between the two accesses.

3. Writes to the same location are *serialized*. That is, two writes to the same location by any two processors are seen in the same order by all processors.

The first property establishes a basic local ordering property for each processor i.e. it is what we would expect to hold true for a uniprocessor system. The second property defines a notion of what it means for multiple processors to have a coherent view of memory i.e. processors should observe the most recent effects of writes by other processors. The third property, *write serialization*, is also required, to ensure that all processors observe writes to a particular memory address in the same order.

Basic Schemes for Enforcing Coherence

In a cache coherent multiprocessor, the caches provide both *migration* and *replication* of shared data items:

- *Migration*: A data item can be moved to a local cache and used there in a transparent fashion.

- *Replication*: When shared data are being simultaneously read, the caches make a copy of the data item in the local cache. This reduces latency of access and contention for a read shared data item.

The protocols to maintain coherence for multiple processors are *cache coherence protocols*.

TODO: *Snooping* vs. *directory-based* cache coherence protocols.

6 Hardware Description Languages

Hardware description languages (HDLs) are languages for describing digital circuits at a higher level of abstraction, rather than directly describing every logic gate and their exact placement, connections, etc.

Verilog

Verilog is one of the most common HDLs, and it can be used to define both combinational and sequential circuits. As a simple example, we can describe a simple combinational circuit in Verilog as a *module*. For example, the following describes a circuit with one *input* and one *output*, where the output is the negation of the input.

```
module top_module(input a, output b);
    assign b = !a;
endmodule
```

When circuits become more complex, we can also declare *wires*, which are like internal connections that are not externally visible outside of the module. For example, the following declares an internal wire that takes on the negation of the input `a`, and this wire is then fed (i.e. connected to) the output. For this simple example the wire doesn't really serve a necessary purpose, but with larger circuits wires can help to decompose more complex bits of logic.

```
module top_module(input a, output b);
    wire w1; // wire declaration.
    assign w1 = !a;
    assign b = w1;
endmodule
```

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.