Contents

1 Background

1.1 Symbols / Notation

I assume a working familiarity with binary. Knowledge of modular arithmetic is not required but familiarity with the notation will be of use.

1.2 Machine model

We are citing the computer organization and design book by Hennessy&Patterson in general.

We want a general model of a computer to reason about. But modern computers are both complicated and vary significantly in their implementation of low level operations. Instead we want to reason about a general idealized computer, and then claim relevance to the real world via the argument that real computers are refinements of the general high level abstractions.

This follows the pattern of the field, as these abstract computers predated physical machines that operated accord to those principles. {needed?}

We consider a general Von Neumann architecture computer {cite}. For the purposes of this thesis, we will ignore input and output from the computer and all non-deterministic features. What remains is naturally divided into three interconnected subsystems.

1.2.1 Byte Sequences

Before we can discuss the systems of a computer, we will benefit from a formalization of data in general. We consider the fundamental unit of meaningful data to be a byte sequence. A byte sequence is defined as a linear sequence of bytes, where each byte can also be undefined, in which case we do not know what it contains, and we cannot safely use it. A byte sequence can be a sequence of both defined and undefined bytes. A byte sequence has the natural property of length, defined as the number of bytes in the sequence. We may abbreviate byte sequence as b-sequence or b-seq.

Let B be the set of all b-sequences.

$$\text{len}\colon B \to \mathbb{N}$$

where

$$\text{len}(b) = \text{ number of bytes in } b$$

We will write B for the set of all b-seqs, and B_i for the set of all b-seqs of length i. These are mostly useful as terms to define operations on qualified length sequences without regard for the contents of the byte sequence, as below. Byte sequences are also equipped with two other natural operations, concatenation and splitting. We define concatenation as

$$+\!\!+ : B_i, B_j \to B_{i+j}$$

Generally written

$$a +\!\!+ b = c$$

Where the resulting b-seq is simply the input b-seqs placed end to end in the order of the arguments. this operation naturally extends to more than two arguments, as it is associative.

We define *split* to be the operation that undoes concatenation, however we must include one additional piece of information that was lost in concatenation, namely where the split should be.

$$/\!/ : b_{i+j}, \mathbb{N}_{\leq i+j} \to (b_i, b_j)$$

generally written

$$/\!/(a, i) = (b, c) \text{ when } a = b +\!\!+ c \text{ and } \mathrm{len}(b) = i$$

This operation returns a pair of the first i bytes and the remaining bytes in order. This operation also extends naturally to taking a set of split points and returning a tuple with as many entries as 1 plus the number of split points.

It is reasonable to assume that a fully defined byte sequence represents one of more pieces of semantically meaningful data, i.e. data that holds meaning to a programmer familiar with the surrounding context. The most common and central interpretation of the contents of a byte sequence would be as an integral number, either in the context of an unsigned natural number or a signed integer proper. The details of that representation will not be covered, but we can assume that an arbitrarily large finite integer can be represented by a byte sequence of sufficient length.

Note that these operations (concatenation and splitting) operate irrespective of the contents of the bytes or whether the bytes are defined. Those internal details are preserved under these operations. The specifics of any particular encoding are not assumed. So particularly one could concatenate a defined sequence with an undefined one to produce a mixed sequence, and then later split the mixed sequence to produce the original pair of sequences, assuming that one kept track of the lengths so as to specify the correct split point. The insight here is that both operations, but particularly split, are well defined on all possible inputs in their domain. This hinges on the fact that split is only defined with indices that fall in the meaningful range. As a reminder, even undefined or sequences of mixed defined and undefined bytes still have a concrete length. A sequence being undefined simply means we don't know anything about the contents, and thus cannot reason about them. That uncertainly does not extend to the length of the sequence, nor is it the same as a byte being known but unspecified. This will become apparent later. It is the responsibility of the user to apply these operations in a manner that preserves semantic meaning if that is desired.

With our notion of byte sequences in hand, we can proceed.

1.2.2 Memory

The first is Memory, which we conceptualize as a function of addresses to values

$$\begin{aligned} M &: \mathbb{N} \to B_1 \\ &: a \mapsto b \end{aligned}$$

Let $\mathcal{M}$ be the set of all such functions.

However it is cumbersome to talk about bytes individually when often we want to operate on b-sequences, most commonly words, which will be defined later. We will extend our notation to

$$\begin{aligned} M &: \mathbb{N}, \mathbb{N} \to B \\ &: a, l \mapsto b \text{ such that } \operatorname{len}(b) = l \end{aligned}$$

where b is the iterated concatenation of sequential addresses starting at a.

$$b = m(a) \mathbin{+\!\!+} m(a+1) \mathbin{+\!\!+} m(a+2) \mathbin{+\!\!+} ... \mathbin{+\!\!+} m(a+l-1)$$

thus we have b as the sequence of l bytes starting at a. Once again, the existence of semantic meaning of b depends wholly on the linear sequence of bytes being fetched being in the same order as is desired in the resulting b-seq. Once again this definition is endianness agnostic.

We define the operation called a *read* as invoking the function for our current memory state at the given address and for the given length. thus we write

$$read(m, a, l) = b$$

exactly when

$$m(a, l) = b$$

Most of the time we will not write the memory function, as it will be clear from context.

We define an operation called a *write* as taking a memory function, and returning another memory function such that the output of the memory function at the given address is altered, and all others remain the same.

we write

$$writebyte(m, a, b) = m'$$

where len$(b) = 1$ such that

$$read(m', a, 1) = b$$

and

$$read(m', a', 1) = read(m, a', 1)\ \forall a' \neq a.$$

We can then extend *writebyte* naturally to

$$write(m, a, b) = m'$$

without the restriction on the length of b. this is equivalent to

$$\begin{aligned} writebyte(m, a, b) = & writebyte_1(...(\\ & \quad writebyte_1(\\ & \quad\quad writebyte_1(m, a, b_0), \\ & \quad\quad a + 1, b_1), \\ & \quad ...), \\ & a + l - 1, b_{l-1}) \end{aligned}$$

where b_i is the i th byte of b, and l = len(b). This is the composition of changing each byte one at a time. Once again, the value starting at a and extending for l bytes is altered, and all others are preserved.

We say a memory access is s aligned if the address is a multiple of s.

$$a \equiv_s 0$$

We say an access is naturally aligned if the address is l aligned where l is the length of the access.

Our model is not limited to only naturally or s aligned accesses and will operate in a defined and reasonable way even for unaligned accesses. However this is not true for all computers. Many computers limit defined behavior to access that are naturally aligned or aligned to a minimum of a fixed value. Our model is strictly stronger and can represent those access patterns, as they are a subset of the unrestrained pattern.

We define these two operations that are thin wrappers around the behavior of our memory function in order to more conveniently model the way in which real computers interact with their memory. The memory functions as we have described them are simply a shorthand way to talk about the entire state of memory (all data that has been stored), in a compact form. It is equally useful to think about memory as a giant lookup table of values, as in fact they are exactly that in reality.

This formalism is meant to capture the intuition that memory holds onto data indexed by an address. We can ask for the data again without destroying it, and we can change the data so that further requests will return the new value.

1.2.3 Arithmetic

The second subsystem of a general Von Neumann computer is the arithmetic logic unit, commonly abbreviated to ALU. The ALU is responsible for all simple operations of the computer. The most general formulation is a black box that knows about some fixed finite number of elementary operations. This box can be queried with an operation and the appropriate number of arguments and will return the operation applied to said arguments. We will not create a formal object to talk about the ALU as a whole, but for a given elementary operation $\star$ on byte sequences of appropriate length, the existence of the ALU allows us to write

$$b \star b' = b''$$

and perform computations on the contents of byte sequences. This is what differentiates the ALU from other systems. In this simple model, it is the only system that acts according to the semantic meaning of a byte sequence rather than moving, copying, storing, retrieving, or otherwise doing some operation on the entire sequence.

Note that the ALU only takes as inputs and returns as outputs byte sequences of the appropriate length for the given operation. However, even if the data in the b-seq does not carry semantic meaning appropriate for the given operation, if the lengths are correct, then the ALU will return some byte sequence according to its internal logic. Thus the result will likely not carry appropriate or any semantic meaning. In addition we consider it to be an error to perform any ALU operation on an undefined byte. Treatment of errors will be discussed later.

Thus for some set of simple operations, we can now combine and manipulate semantically meaningful sequences of data. Common included operations are the elementary arithmetic operations (addition, subtraction, multiplication, and division) and binary operations, (bit flips, bit shifts, sign or zero extensions). Our model is only concerned with integer values, but computers that deal with floating point values or other fundamental types of data may have need of other operations.

1.2.4 Control

The final subsystem is the subsystem of Control. This system varies in form between real computers more than the other two, but in general this system connects the other systems, directs the operations of the computer, and (in general) holds intermediate values of complex calculations or the working set of data at a given time.

For the purposes of this section, fix some length l. This is referred to as the data width or word width of the computer, and is the unit of data that most of the operations work on. A byte sequence of length l is called a word. A computer of data width l can still operate on larger or smaller data, but in general they will be slower than on data of length l. In general, the ALU will have dedicated versions of common operations such as addition on various data lengths, and operations to extend or truncate a sequence to a different length while preserving the semantic meaning of the data.

The control system has some internal storage. This storage is called the register file, and it is separated into some fixed number of registers, each of length l. For the rest of the section, fix k as the number of registers in the register file. Each register can hold a single b-seq of length l, and as with memory, it will hold its value until explicitly changed. At a minimum, the Control system must have at least one register, the Program Counter register. Most computers will have a number of extra registers. The Program

Counter register always carries the semantic meaning of a natural number. The number's significance is that it is an address in memory that will be interpreted in the next step.

The Control system is critical in that it is the engine that drives the rest of the computer. The innermost loop on which all actions are performed is as follows:

1. The Control system reads an instruction from memory starting at the address stored in the Program counter.

2. The resulting b-seq is broken down into several sections and interpreted. This interpretation identifies one or more actions. These operations are often ALU operations, where the argument b-seq should come from, and where the result b-seq should go. However there are also memory operations, which take an address and read or write to it, and others, including altering the program counter subject to a condition on some other value. If more than one action is produced from the interpretation, the actions are equivalent of multiple singular actions specified in the program code in sequence. The difference is that the program counter is not updated between actions, and multiple memory fetches for each interpretation are not required.

3. The arguments are acquired (generally from memory or from the register file), and fed into the APU or interpreted by the Control system.

4. The resulting b-seq is acquired from the APU and the side effects are produced by the Control system. For instance, on a memory read, the b-sequence from memory is acquired.

5. The resulting b-seq is placed in the specified location as per the interpretation of the fetched value at the program counter. Again, this is often a register or in memory. In the event of a memory write such as $write(M, a, v)$ where M is the memory state before the instruction begins, then the resulting M' is now the memory state after this instruction, and will be the assumed contextual memory state up to and including the next write.

6. The program counter is incremented by one instruction. In the event of a branch or other non-linear control flow instruction, this step may be skipped by some implementations.

This loop is then repeated forever. The effect of this loop is that the computer performs operations one at a time, in order, according to the sequence of words starting at the location pointed to by the initial program counter. Note also that the Program Counter may not pass linearly over the code. In fact, most code heavily features control flow, where the Program Counter is manipulated with arithmetic operations to allow jumping over and into code that is not directly adjacent to the prior instruction. What separates the Von Neumann model from other models (namely the Harvard model) is that the instructions (the words that specify actions) are not segregated from the data being operated on. A memory read or write can touch a piece of data that is being operated on by the program in the same way that it can touch the set of instructions. This insight that code is data allows for incredibly flexible programs that can edit themselves while running, or other high levels of abstraction.

1.3 Instruction / asm model (P-Code)

1.4 Satisfiablility