A Parallel Architecture for High Speed Data Compression

James A. Storer, *Computer Science Dept., Brandeis University, Waltham, MA 02254*
John H. Reif, *Computer Science Dept., Duke University, Durham, NC 27707*

October 15, 1990

Abstract: Data compression is becoming an essential component of high speed data communications and storage. *Lossless* data compression is when the decompressed data must be identical to the original. *Textual substitution* methods are among the most powerful approaches to lossless data compression, where repeated substrings are replaced by pointers into a dynamically changing dictionary of strings. We present a massively parallel architecture for textual substitution that is based on a systolic pipe of 3,839 identical processing elements that forms what is essentially an associative memory for strings that can "learn" new strings based on the text processed thus far. Key to the design of this architecture is the formulation of an inherently "top-down" serial learning strategy as a "bottom up" parallel strategy. A custom VLSI chip for this architecture that operates at 300 million bits per second has been fabricated.

1. Introduction

Lossless data compression is the process of encoding ("compressing") a body of data into a smaller body of data which can at a later time be uniquely decoded ("decompressed") back to the original data. Not all data is compressible. However, data that arises in practice typically contains redundancy of which compression algorithms may take advantage. The two most common applications of data compression are:

Data Communications: A sender can compress data before transmitting it and a receiver can decompress the data after receiving it, thus effectively increasing the data-rate of the communication channel.

Data Storage: Data is compressed before it is stored and decompressed when it is retrieved, thus increasing the effective capacity of the storage device.

There has traditionally been a tradeoff between the benefits of data compression versus the computational costs of the encoding and subsequent decoding. Massively parallel architectures have the potential for real-time performance over high bandwidth channels where data travels at millions or billions of bits per second. The key contribution of this paper is the

Preprint of paper appearing in Journal of Parallel and Distributed Computation, 13, 1991, pp 222-227.

design of a systolic architecture (a very practical model of massively parallel computation where each processor communicates only with its left and right neighbor in a linear array) for *dynamic textual substitution*, a powerful approach to data compression that prior to this work, seemed to require complex global communication.

Our approach is to modify one of the best textual substitution methods in a way that yields equivalent compression but admits a practical systolic implementation. The key technical idea is the formulation of an inherently "top-down" serial learning strategy as a "bottom up" parallel strategy.

We have designed and fabricated a prototype VLSI chip based on this architecture. Each chip contains 128 processing elements and thirty of these chips comprise a complete encoding / decoding pipe of 3,839 processing elements that is capable of operating at 300 million bits per second (that is, using a 40 meg hertz clock, in encoding mode the system can read a new byte on every clock tick and in decoding mode the system can write a byte on every clock tick). These chips, together with some control logic, can be placed on a single board to form a complete system. To speed completion time and reduce costs, the design process has been kept simple; higher performance is clearly possible with current technology.

2. Textual Substitution Methods

To simplify our discussion we henceforth assume all *data* to be a sequence of 8-bit *characters* and we refer to the set of all possible characters (the integers 0 through 255) as the *input alphabet* Σ. In addition, we assume that all handling of the data while it is in compressed form is *noiseless* (no bits are added, lost, or changed). We make this assumption for convenience; a host of techniques are available in the literature for error detection and correction.

We employ an *on-line* model for data compression where the *encoder* and *decoder,* each have a fixed (finite) amount of local memory, which we refer to as a *local dictionary*, D. We assume that the two local dictionaries may be initialized at the beginning of time to be empty. What distinguishes this model from an *off-line* model is that neither the sender or the receiver can see all of the data at once; data must be constantly passing from the sender through the encoder, through the decoder, and on to the receiver.

A powerful (and practical) approach to compression of a string of characters is *textual substitution*. For a general treatment of textual substitution techniques as they apply to both on-line and off-line algorithms, as well as further references on the subject, see the book of Storer [1988]. The basic idea is to use the local dictionary D to store a constantly changing set of strings. Data is compressed by replacing substrings of the input stream that also occur in D by the corresponding index into D; we refer to such indices as *pointers*. Thus, the input to an on-line data compression algorithm employing textual substitution is a sequence of characters and the output is a sequence of pointers (where typically, most of the pointers specify strings of length greater than 1 but a few are essentially character codes).

The following page presents generic encoding and decoding algorithms for on-line dynamic textual substitution. Most on-line textual substitution algorithms can be viewed as instances of one general approach. The generic encoding algorithm reads a stream of characters and writes a stream of bits, and the generic decoding algorithm receives a stream of bits and outputs a stream of characters. We use the notation D_{max} to denote the maximum number of entries that D may contain. The chip that has been fabricated for our architecture uses $D_{max} = 4,096$ (indices 0 to 255 correspond to the basic single byte values and are not actually stored).

We have intentionally talked about the maximum number of entries that D may contain without any reference to how long individual entries may be. This is because all of the algorithms to be discussed in this section (and the parallel algorithm to be presented in the next section) can organize the local dictionary D with a data structure that represents each dictionary entry by a constant amount of space that is independent of the actual length of the corresponding string. Experimental simulations show that in practice, $D_{max} = 4,096$ provides a nice compromise that achieves most of the compression achievable with the underlying algorithm while at the same time keeping the number of entries reasonable.

The generic encoding and decoding algorithms work in lock-step to maintain identical copies of D (which is constantly changing). The encoder repeatedly finds a match between the incoming characters of the input stream and the dictionary, deletes these characters from the input stream, transmits the index of the corresponding dictionary entry, and updates the dictionary with some method that depends on the current contents of the dictionary and the match that was just found; if there is not enough room

```
(1) Initialize the *local dictionary* $D$ to have one entry for each character
    of the input alphabet.
(2) **repeat forever**
    (a) {*Get the current match string s:*}
        Use a *match heuristic* MH to read $s$ from the input.
        Transmit $\lceil log_2|D|\rceil$ bits for the index of $s$.
    (b) {*Update D:*}
        Add each of the strings specified by an *update heuristic* UH to
        $D$ (if $D$ is full, then first employ a *deletion heuristic* DH to make
        space).
```

Generic Encoding Algorithm

```
(1) Initialize the local dictionary $D$ by performing Step 1 of the encoding
    algorithm.
(2) **repeat forever**
    (a) {*Get the current match string s:*}
        Receive $\lceil log_2|D|\rceil$ bits for the index of $s$.
        Retrieve $s$ from $D$ and output the characters of $s$.
    (b) {*Update D:*}
        Perform Step 2b of the encoding algorithm.
```

Generic Decoding Algorithm

left in the dictionary, some deletion heuristic must be performed. Similarly, the decoder repeatedly receives an index, retrieves the corresponding dictionary entry as the "current match", and then performs the same algorithm as the encoder to update its dictionary.

Left out of the generic encoding and decoding algorithms is the specification of the following three heuristics:

The match heuristic, MH: A function that removes from the input stream a string s that is in D. Since the characters of Σ are implicitly in D and can never be deleted, there is always at least one such s.

The update heuristic, UH: A function that takes the local dictionary D and returns a set of strings that should be added to the dictionary if space can be found for them.

The deletion heuristic, DH: A function that takes the local dictionary D and returns a string of D that may be deleted.

Our architecture is based on the following choices:

MH: The *greedy heuristic* is to always take the longest possible match between the input stream and a string in D. Key to our architecture will be a "bottom up" method for approximate determination of the longest match.

UH: The *identity* (ID) heuristic is to add the previous match concatenated with the current match. Key to our construction will be a modified version of the ID heuristic to be discussed later.

DH: The *swap* ($SWAP$) heuristic works as follows: When the *primary* dictionary first becomes full, an *auxiliary* dictionary is started, but compression based on the primary dictionary is continued. From this point on, each time the auxiliary dictionary becomes full, the roles of the primary and auxiliary dictionaries are reversed, and the auxiliary dictionary is reset to be empty. Although this heuristic does not fit directly into the generic algorithms, they can be modified to accommodate it.

Different combinations of choices for the above three heuristics than we have used here give rise to a variety of methods, many of which have been studied extensively in the literature. However, we shall not be concerned with other heuristics here. The book of Storer [1988] considers tradeoffs between different heuristics (in terms of both computing resources and amount of compression) and the relationship between various heuristics and work by past authors. Most authors in the past have used for MH the *greedy heuristic.* Other choices for UH include the *first character* (FC) heuristic (concatenate the last match and the first character of the current match) and the *all prefixes* (AP) heuristic (concatenate the last match with each of the prefixes of the current match). Using the traditional serial RAM model of computation, the FC and AP heuristics can be easily implemented in real time using constant space per entry by representing the dictionary with a standard trie data structure (dictionary entries correspond to nodes of the trie). The ID heuristic requires a much more complicated data structure that has some undesirable worst-case properties. However, the situation changes with a massively parallel model of computation (with some appropriate modifications to the ID heuristic). Other choices for DH include FREEZE (once the dictionary is full, it remains the same from that point on) and LRU (delete that string in D that has been matched least recently); note that SWAP can be viewed as a discrete version of LRU. The FREEZE heuristic can be "unstable" in practice when the data characteristics change after the dictionary is full. The LRU and SWAP heuristics typically perform equivalently in practice. The serial nature of a doubly-linked list use by LRU is "natural" for a serial implementation whereas SWAP appears to be more natural for parallel implementations. The FC-FREEZE heuristic can be viewed as a practical interpretation of the second model proposed by Lempel and Ziv (Lempel and Ziv [1976], Ziv and Lempel [1977,1978], Welch [1984]), and the FC-LRU is a natural improvement to FC-FREEZE. The ID heuristic was considered by Seery and Ziv [1977,1978] and the ID-LRU heuristic models the work of Miller and Wegman [1985]. The AP-LRU heuristic is discussed in Storer [1988].

3. A Parallel Implementation

The model of parallel computation employed by our algorithm is a *systolic pipe*; that is, a linear array of processors, each connected only to its left and right neighbors.

A real-life example of a systolic pipe is an automobile assembly line may produce a new car every 20 minutes even though each car is in the assembly line for a day. Although each station in the automobile assembly line performs a different task, the stations are at least conceptually identical, if we view them all as taking as input a partially built car, performing an elementary operation (such as welding), and then outputting a partially completed car to the next station.

From the point of view of VLSI, a systolic pipe has the following desirable properties:

- All processors are identical and the length of connections between adjacent processors can be bounded by a constant.
- The structure can be laid out in linear area and power and ground can be routed without crossing wires.
- The layout strategy can be independent of the number of chips used. A larger pipe can be obtained by placing as many processors as possible on a chip and then, using the same layout strategy, placing as many chips as possible on a board.

We have intentionally left out of our definition of a pipe the specification of what constitutes a "processor". In principle, any computational device, including a main-frame computer, could be used. However, it is typical for processors to be extremely simple. Here a processor consists of only a few registers, a comparator, and some finite-state logic.

Figure 1 depicts a standard "snake" layout for a systolic pipe, Figure 2 shows an individual processor, and Figure 3 shows how power and ground for the processors can easily be added to the snake.

Gonzalez and Storer [1985] and Storer[1988] present a systolic pipe implementation for the static dictionary model (where the dictionary is supplied in advance and never changes) that effectively forms an associative memory where a given entry may be formed with pointers to other entries. Here we describe how to generalize such a structure to maintain a dynamically changing dictionary based on a variant of the ID update heuristic, the $SWAP$ deletion heuristic, and a "bottom up" parallel version of the greedy match heuristic.

3.1. Encoding

We start by describing a systolic pipe implementation only as it pertains to encoding; unlike the serial case, here, decoding will be the more complicated operation. All pointers are represented by the same number of bits; the bits for each input character are padded to the left with zeros to form the corresponding pointer and sent into the left end of the pipe (for our chip, each input character of 8 bits is padded to the left with 4 zeros to form a 12 bit pointer). The pipe consists of $D_{max} - |\Sigma| - 1$ processors numbered $|\Sigma|$ through $D_{max} - 2$ going from left to right; there are no processors for indices 0 through $|\Sigma| - 1$ since the characters of Σ are always implicitly in the dictionary and there is no processor numbered $D_{max} - 1$ (this pointer value is used for the nilpointer, to be discussed later). The processors are numbered so that if $i < j$, then processor i is to the left of processor j (and occurs earlier in the pipe than processor j).

The pipe stores a “pair forest” representation of the dictionary (each dictionary entry is represented by a pair of pointers to two other entries or to single characters) and implements a modified version of the ID heuristic. Each processor is capable of holding a pair of pointers (corresponding to the left and right pointers into the pair forest), but is initialized to be empty. The *leader* bit is initially 1 for the leftmost processor and 0 for all others. It is always the case that at most 1 processor is the leader, that the processors to its left contain dictionary entries, and that processors to its right are empty.

As data passes through the processors to the left of the leader as it is encoded. That is, whenever a pair of pointers enter a processor and matches the pair of pointers stored at that processor, this pair of pointers is removed from the data stream and replaced by a single pointer (the index of that processor). Data passes unchanged through the processors to the right of the leader. When a pair of pointers enters the leader, they represent adjacent substrings of the original data and can be viewed as the “previous” and “current” match. The leader can simply adopt this pair of pointers as its entry, set its *leader* bit to 0, and send a signal to the processor to its right to set its *leader* bit to 1. Note that unlike the serial algorithm, this is a “bottom up” approach to finding a longest match, since bigger matches are built from smaller ones. We do not want the new leader to adopt the same pair of pointers that was just adopted by the processor to its left. Hence, when a processor first becomes the leader, it must allow one pointer to pass through before proceeding to “learn”.

When the rightmost processor becomes the leader, it adopts a pair of pointers as its entry and then sends a signal to the right to pass on the *leader* bit; this signal indicates that the dictionary is full. At this point, the *SWAP* deletion heuristic can be implemented with a switch between two copies of the pipe that routes input/output lines appropriately as the dictionaries turn over.

3.2. Decoding

Consider now the operation of the decoding pipe. Although the same pipe is used for both encoding and decoding and when decoding data still enters from the left and leaves to the right, the numbering of processors is reversed; the rightmost processor is numbered $|\Sigma|$ (processor indices are sent down the pipe by the controller into the *index* registers upon power up or upon a restart that changes between encode and decode mode). Operation of the decoding pipe is essentially the reverse of the encoding pipe. Processor $|\Sigma|$ is initially the leader, and the remaining processors are empty. Compressed data enters from the left, passes through the empty processors unchanged, and is decoded in the non-empty portion of the pipe; that is, when a pointer arrives to a processor that is equal to the processor index, it is replaced by the two pointers stored by that processor. The leader bit propagates from right to left (the opposite direction from the encoder).

It is an inherent aspect of data compression there may be no reasonable bound on how many more decompressed bits there are than compressed bits. We address this issue with a *stop* bit that is be sent from a processor to the processor to its left to signal that no more data should be sent until the bit is turned off. A typical sequence of events for a decoding processor would be the following (each processor has a buffer capable of holding at most two pointers in its *BuffA* and *BuffB* registers):

1. The input buffer currently contains one pointer that is not the same as the processor index and the stop bit from the processor to the right is not set.
2. A clock tick occurs and the current pointer in the buffer is sent to the right and a new pointer is read into the buffer.
3. The new pointer in the buffer is the same as the processor index. This pointer is replaced by the first of the two pointers stored in the processor memory and the stop bit is set.
4. A clock tick occurs. The pointer in the buffer is transmitted to the right. Although a stop bit is now being transmitted, the processor to the left has not seen it yet and another pointer arrives. The second pointer in the processor memory is placed in the buffer before the pointer that just arrived. The buffer now contains two pointers.

5. A clock tick occurs and the first of the two pointers in the buffer is transmitted to the right (no new pointer arrives from the left since the stop bit was set on the last clock tick). The stop bit is unset.
6. A clock tick occurs, the pointer in the buffer is sent to the right, and a new pointer arrives from the left.

If the stop bit is sent from the leftmost processor to the communication line, then this must handled with an xon / xoff protocol to the sender. Note that the use of the stop bit does not require any signal propagation; on each clock tick, the *only* form of communication is between a processor and the processors to its immediate left and right.

3.3. A Modified Pointer Adoption Strategy

We have not yet discussed in detail how the decoder "learns" entries. The naive method is to do essentially the same as the encoder. Suppose, for example, that four pointers are coming down the pipe: pointer x, followed by pointer w, followed by pointer v, followed by pointer u. Now consider what happens when pointers w and x have arrived at the leader processor. The leader processor adopts the entry wx and then passes leadership to the left. The processor to the left waits until pointer u and v have arrived and then adopts the entry uv. The encoder would have also learned the entries uv and wx; but in addition, it the encoder would have learned the entry vw. The entry vw has been "lost" by the above naive implementation of the decoder. The situation can be remedied by interspersing a nilpointer between every pointer in the input stream to the decoder. The effect is to slow down the operation of the decoder to allow time for the "overlapping" entries such as vw to be learned. The interspersion of nilpointers is necessary only during the learning period when not all processors have adopted an entry. Note also that the output data rate from the decoder is unaffected so long as the data has been compressed by at least a factor of 2.

We will not consider the details of such an implementation further because empirical tests indicate that learning every other entry achieves equivalent compression in practice (the full draft also gives theoretical justification). That is, our strategy is to use the naive decoder implementation described above and modify the encoder to also learn only every other entry. The old rule for a leader in the encoder was to adopt a pointer pair only if at least one of them was not adopted by the preceding leader. The modified rule is:

Adopt a pointer pair only if neither pointer was adopted by the preceding processor.

We implement his modified version of the ID heuristic by marking a pointer (by passing an extra bit along with each pointer) once it has been adopted by a leader. This could also be done by waiting for two pointers to pass by before trying to adopt; however, the mark bit proves useful for other purposes when the details of the finite state control of a processor are considered.

3.4. The Flush Operation

As mentioned earlier, the processor numbered $D_{max} - 1$ has been left out of the pipe. The corresponding pointer value (which is all ones when D_{max} is a power of 2) is called the *nilpointer*. Nilpointers are sent into the encoding pipe on every clock tick for which no input is present. Another role of the nilpointer is to implement *flush* operations and record them in compressed data. A flush operation is a signal sent down the encoding pipe to "push" all data out when there is a pause in the input or when it has ended. Note that it is not necessary to wait for a flush to make it through the pipe before starting a new input stream; after one clock tick, new data can "follow" the flush that is pushing out old data.

4. A Massively Parallel VLSI Chip

A preliminary custom VLSI chip for our architecture (patent pending) with a reduced number of processors was fabricated by MOSES in July of 1990 (and has passed all tests). Layout was done by the CDSR group at the Research Triangle Institute of North Carolina. The full version is currently being fabricated by MOSES and has the following characteristics:

- 1.0 micron double metal CMOS technology.
- 68-pin LCC package (48 pins for signals, 20 pins for power and ground).
- Die size = 9,650 by 10,200 microns (.38 by .40 inches).
- 350,000 transistors.
- Power consumption = 750mW (the consumption is very evenly distributed; the chip has no "hot" spots).
- 128 processors per chip (30 chips form a complete systolic pipe of 3,389 processors).
- 40 mhz clock, 8 bits per cycle, 320 million bits per second.

The packaged chips are less than 1 inch square. Due to the extremely simple pattern of interconnections (communication is only between a chip and the ones to its immediate left and right), a board area of approximately 5 by 6 inches suffices to accommodate a basic set of 30 chips that forms a complete systolic array of 3,389 processing elements. This chip set along with some control logic can easily be placed on a small board.

5. References

M. Gonzalez and J. A. Storer [1985]. "Parallel Algorithms for Data Compression", *Journal of the ACM* 32:2, 344-373.

A. Lempel and J. Ziv [1976]. "On the Complexity of Finite Sequences", *IEEE Transactions on Information Theory* 22:1, 75-81.

V. S. Miller and M. N. Wegman [1985]. "Variations on a Theme by Lempel and Ziv", *Combinatorial Algorithms on Words,* Springer-Verlag (A. Apostolico and Z. Galil, editors), 131-140.

J. B. Seery and J. Ziv [1977]. "A Universal Data Compression Algorithm: Description and Preliminary Results", Technical Memorandum 77-1212-6, Bell Laboratories, Murray Hill, N.J.

J. B. Seery and J. Ziv [1978]. “Further Results on Universal Data Compression”, Technical Memorandum 78-1212-8, Bell Laboratories, Murray Hill, N.J.

J. A. Storer [1988]. *Data Compression: Methods and Theory,* Computer Science Press, Rockville, MD.

T. A. Welch [1984]. “A Technique for High-Performance Data Compression”, *IEEE Computer* 17:6, 8-19.

J. Ziv [1985]. “On Universal Quantization”, *IEEE Transactions on Information Theory* 31:3, 344-347.

J. Ziv and A. Lempel [1977]. “A Universal Algorithm for Sequential Data Compression”, *IEEE Transactions on Information Theory* 23:3, 337-343.

J. Ziv and A. Lempel [1978]. “Compression of Individual Sequences Via Variable-Rate Coding”, *IEEE Transactions on Information Theory* 24:5, 530-536.

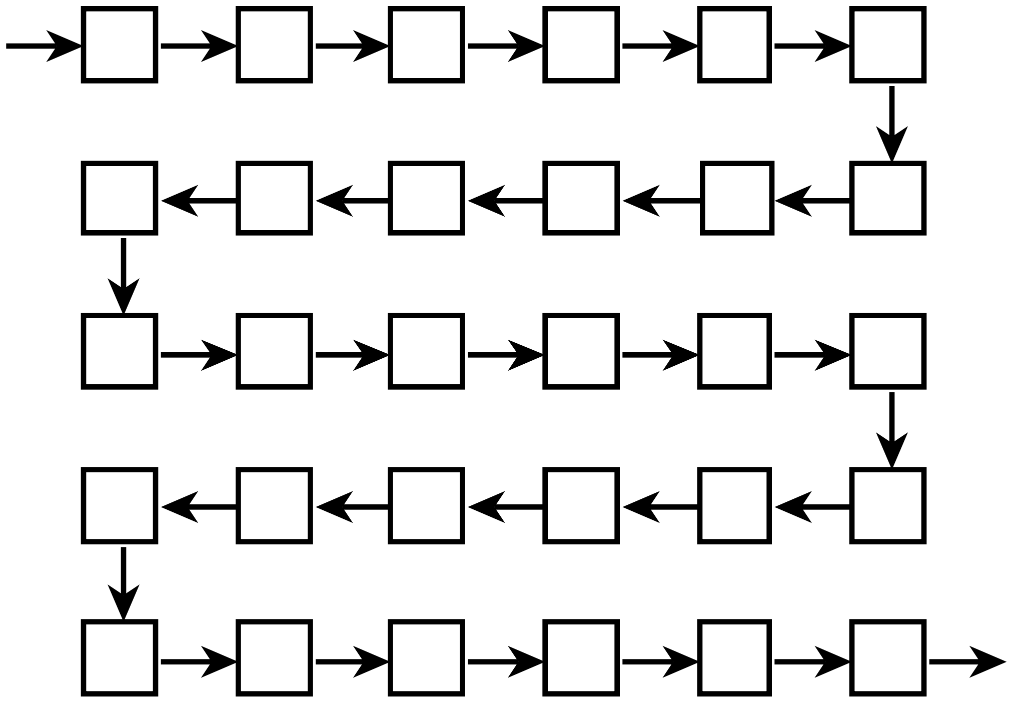

Figure 1: “Snake” Layout

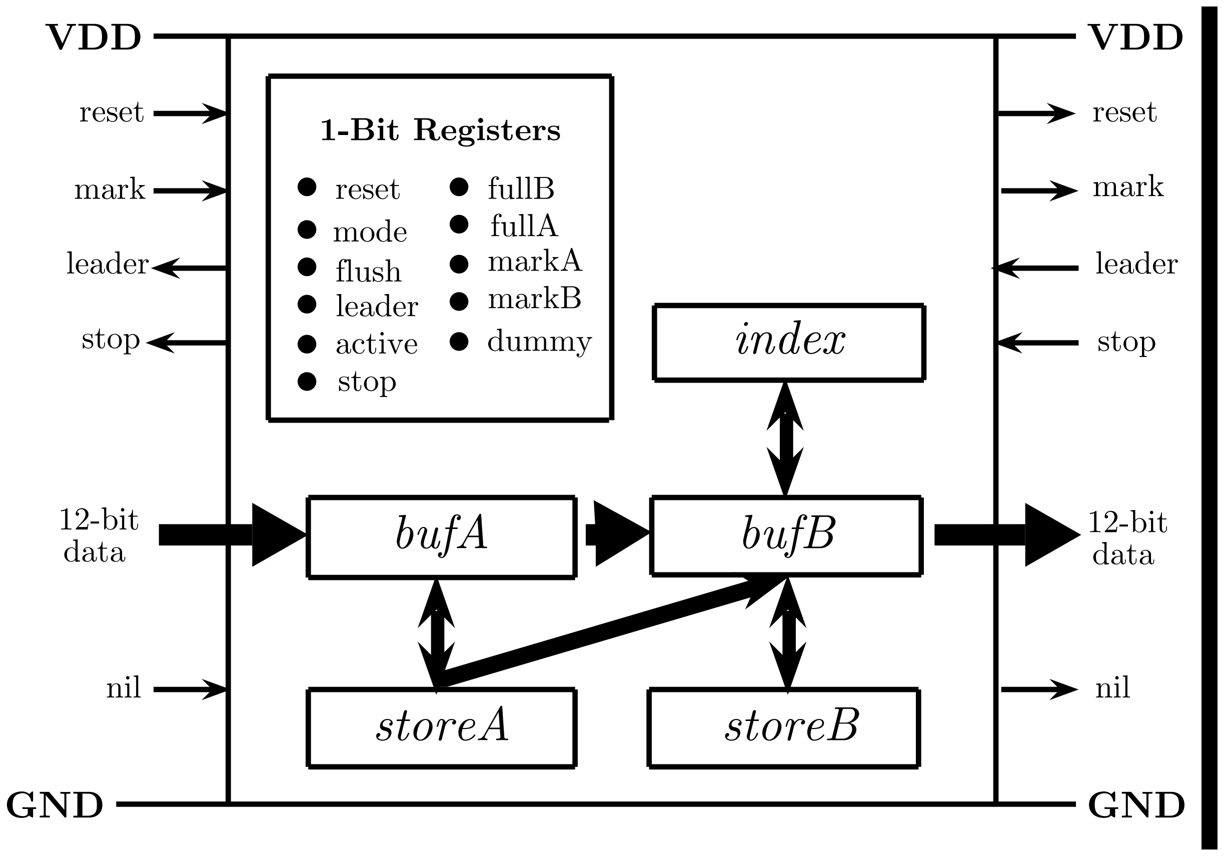

Figure 2: An Individual Processor

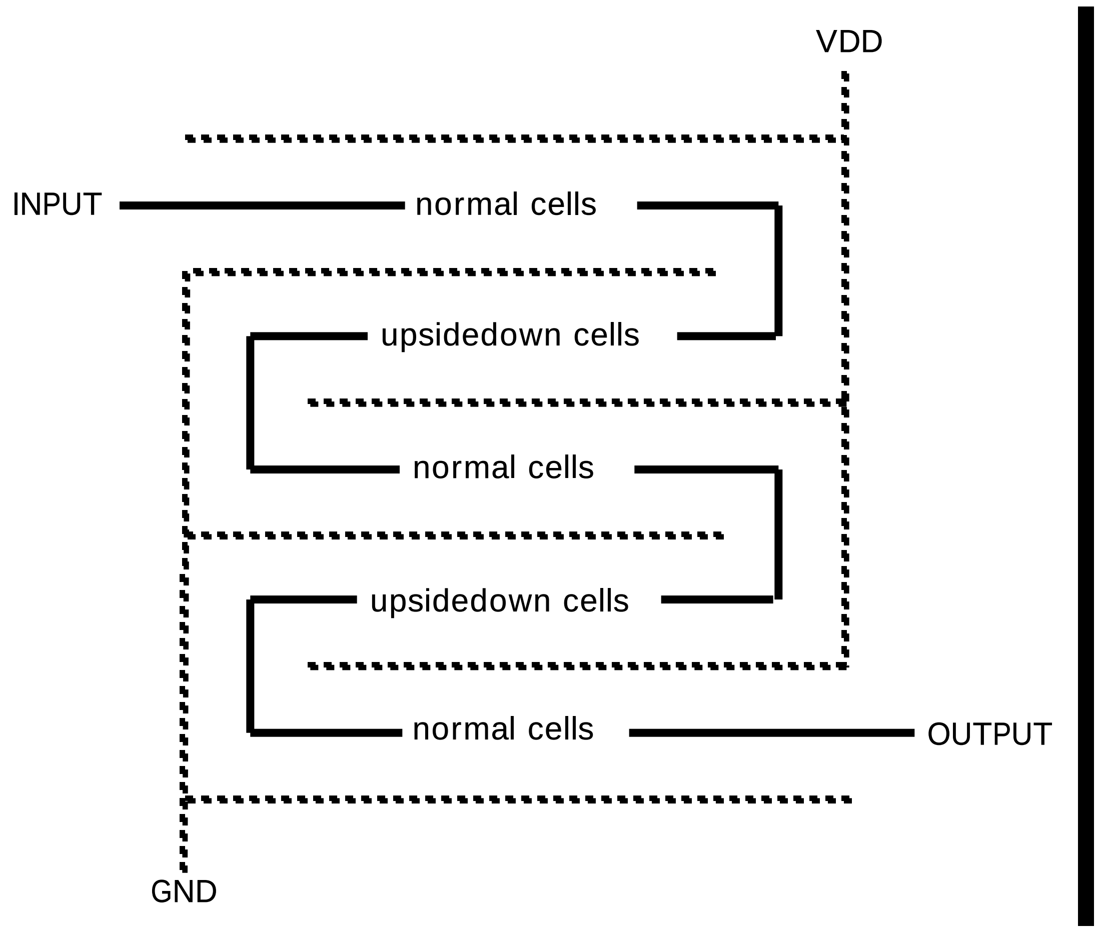

Figure 3: Power and Ground Distribution