Implementing
Non–Numerical Algorithms
On A Decoupled Architecture
Supporting Software Pipelining

Andreas I. Moshovos

Submitted in Partial Satisfaction of the Requirements for the Degree of
Master of Science
in
Computer Science

Heraklio, August 1992
Implementing
Non-Numerical Algorithms
On A Decoupled Architecture
Supporting Software Pipelining

A Thesis Submitted by
Andreas I. Moshovos
in Partial Satisfaction of the Requirements for the Degree of
Master of Science

Heraklio, August 1992

Author:
Department of Computer Science

Thesis Committee:
Manolis Katevenis, Associate Professor, Supervisor
Apostolos Traganitis, Associate Professor, Member
Nikos Alvertos, Assistant Professor, Member

Approved:
Stelios Orphanoudakis, Professor,
President of the Committee on Graduate Studies
Abstract

Access decoupling tries to reduce execution delays due to memory access delays. Software pipelining is a scheduling technique in which successive iterations of a loop, are continuously initiated at constant intervals, before preceding iterations complete. Through the use of software pipelining, better utilization of multiple, pipelined functional units is possible.

Both techniques can be easily applied to most of the numerical algorithms, as these algorithms produce data independent fetch/store sequences and their iterations are simple. On the contrary, it is not easy to apply these two techniques to non-numerical algorithms, since these algorithms produce data dependent fetch/store sequences and their iterations often contain conditional constructs.

The problems arising when trying to execute non-numerical algorithms on an architecture which uses both of the above techniques are discussed in this thesis and some possible implementations are shown. Mainly, sorting algorithms are considered. Initially, processing of elementary data types is discussed. Such data types are easily manipulated, as they can be stored within a register of the target machine. Later, we discuss the additional problems encountered when operating on larger data structures.

The implementations shown were validated through the use of a simulator of our architecture. Execution performance is compared to that of an ideal architecture with a main memory with no access delays, and it is shown that the loss of performance in our architecture is minor when compared to the very low speed of the memory that we assume. Application of these two techniques are significant.
Acknowledgements

First of all I must thank Prof. Manolis Katevenis, my advisor, for convincing me to work on these ideas, for his valuable help and most of all for helping me recognize what was important. I would also like to thank Prof. Apostolos Traganitis for his guidance and support during my studies. He was a great person to talk to about everything. I would also like to thank both of them as well as Prof. Nikos Alvertos for participating in my thesis committee.

I would like to thank the Computer Science Institute of F.O.R.T.H.\textsuperscript{1} which provided both financial and technical support during my graduate studies. This work was supported by the ESPRIT “AMUS” (contract 2716) and “SHIPS” (contract 6523) research projects.

G. Dimitriadis provided valuable information on selecting text processing tools during the preparation of this document. Also P. Vatsolaki was there to help when system problems occurred.

Although they are not directly involved with this work, I would like to thank K. Tsioubarakis and S. Leventis for being both good friends and great persons to work with. Their help was invaluable.

Finally I send my depeest thanks to my family and to G. Siabakos and S. Danas for their love and support.

\textsuperscript{1}Foundation for Research and Technology–Hellas (Greece)
Contents

1 Introduction .................................................. 7
   1.1 Access Decoupling ...................................... 8
   1.2 Software Pipelining ................................... 13
   1.3 Using both access decoupling and software pipelining ......... 22

2 Our Architecture .............................................. 25
   2.1 An access decoupled architecture with three address generators .......... 25

3 Nonnumerical algorithms ...................................... 31
   3.1 Sorting algorithms .................................... 31
      3.1.1 Insertion sort .................................... 31
      3.1.2 Shellsort ......................................... 37
      3.1.3 Selection sort .................................... 37
      3.1.4 Bubblesort ....................................... 40
      3.1.5 Mergesort ......................................... 41
      3.1.6 Heapsort .......................................... 43
      3.1.7 Quicksort ......................................... 44
   3.2 Pattern matching ....................................... 55
      3.2.1 Brute-Force algorithm .............................. 55
      3.2.2 Knuth-Morris-Pratt algorithm .................. 55

4 Sorting when keys are strings ................................. 59
   4.1 Using insertion sort with fixed length keys ..................... 59
      4.1.1 Using insertion sort with variable length keys ........... 61
   4.2 Using quicksort with strings ................................ 61
   4.3 Providing a small data store local to the processing unit ....... 66

5 Measurements .................................................. 67
   5.1 Simulating the architecture ................................ 67
   5.2 Partitioning $10^5$ elements ............................... 68
   5.3 Adding the elements of two vectors ......................... 71

6 Conclusions,Summary .......................................... 73

A Processor model .............................................. 75

B Instruction set .............................................. 77
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Simulator Usage</td>
<td>81</td>
</tr>
<tr>
<td>D</td>
<td>Queues implementation</td>
<td>83</td>
</tr>
<tr>
<td>D.1</td>
<td>Data queues</td>
<td>83</td>
</tr>
<tr>
<td>D.2</td>
<td>Synchronization queues</td>
<td>84</td>
</tr>
<tr>
<td>E</td>
<td>Programs</td>
<td>87</td>
</tr>
<tr>
<td>E.1</td>
<td>Partition process of quicksort</td>
<td>87</td>
</tr>
<tr>
<td>E.2</td>
<td>KMP string searching</td>
<td>90</td>
</tr>
</tbody>
</table>
Introduction

The incredible progress made in computer technology in the past half century was motivated by the demand for even-greater computing power. It’s not likely that this demand will subside in the near future. Even faster processing units and vast amounts of memory are required as the complexity and the size of problems, computers are called to handle, are continuously increasing.

Although advances in integrated circuit technology made it possible to build even faster processing units and larger memories, a tremendous research on improving parallelism within a processor is leading to an increasing performance gap between processors and main memory. As the size of data sets that programs operate on, increases, main memory access time has become a serious bottleneck in improving the overall performance of computer systems.

Extending parallelism within a processor is also an important issue in improving the overall performance of computer systems. Both single processor and multiprocessor machines will benefit from higher levels of parallelism within single processors.

Pipelining, being used in virtually all high-performance architectures, significantly improves parallelism within processors. Increasing the depth of the pipeline and providing more functional units are two methods of augmenting the parallelism in pipelined architectures. But there certain tradeoffs in applying these two methods. Increasing pipeline depth leads to increased execution delays due to pipeline hazards and higher levels of fine-grain code parallelism are required in order to adequately utilize more functional units. Scheduling techniques are called to overcome pipeline hazards and better utilize the system resources.

In this work we are concerned with the viability and usability of architectures that combine the techniques of access decoupling [1] and software pipelining [2], [3]. Access decoupling tries to reduce or hide execution delays due to memory accesses, while software pipelining is a mixed software/hardware scheduling method. Not all algorithms can execute efficiently if one or both of the techniques are applied. An algorithm executes efficiently on access decoupled architectures only if it has data independent fetch/store sequences, while - partial or total - interdependence between successive iterations of the loop constructs is required on architectures that employ software pipelining.

So far there has been a significant work in applying these methods to numerical algorithms, as most of them have data-independent fetch/store sequences and as the successive iterations of their loop constructs are independent. Although most of the scientific/engineering problems are solved by numerical algorithms, nonnumerical algorithms are also usually required. Applying access decoupling and software pipelining to nonnumerical algorithms is not a trivial task, as most of them do not possess the required properties. Usually these algorithms contain
data-depedent fetch/store sequences, while the iterations of their loop constructs depend on the outcome of the execution of previous iterations.

In this work we discuss mainly the problems of implementing sorting algorithms on decoupled architectures, with possible additional support of software pipelining. Whenever this is possible, viable implementations are shown. We also discuss the implementation of a few other non–numerical algorithms.

This thesis is organized as follows. An introduction to access decoupled architectures and software pipelining is first presented. Having explained the basic ideas of software pipelining and access decoupling it is then possible to discuss the benefits gained by the application of these two techniques, their limitations and the problems raised in applying them on nonnumerical algorithms. After that an overview of the architecture developed during this work is presented. The general ideas and the problems of implementing sorting algorithms on that architecture are discussed. A detailed description of these implementations follows. The problems raised in processing elements of non–elementary type are then addressed, along with the architectural extensions required. A simulator developed during this work was used to collect the measurements that follow. Based on these measurements, the efficiency of the proposed implementations and architecture is discussed. Finally some usefull conclusions are drawn, and ideas on futher extending this work are discussed. Overviews of the simulated components and the instruction set used can be found in the appendices.

1.1 Access Decoupling.

Access decoupling tries to reduce execution delays due to memory accesses. In contrast with cache memory techniques, which are based on the locality of reference property, the concept of access decoupling is the following:

Request the data required from memory before they are actually required. If the request of a datum precedes its use by at least by \( n \) cycles, where \( n \) cycles is the memory access delay, then no execution delays occur due to memory delays.

On access decoupled architectures, program execution is seperated into two parts:

- access the memory to fetch operands and declare where the results should be stored.
- arithmetic operations on the operands to produce results.

Such architectures are composed of two major functional units, as figure 1.1 illustrates:

- the address generator (marked as AG) and
- the processing unit (marked as PU)

These units are actually two processors executing seperate instruction streams. The address generator performs all operations necessary for transferring data to and from main memory. It generates the addresses of the data required for or produced by the processing unit and communicates these to the main memory. The processing unit operates on the data fetched from memory and produces the desired results. Each of the two processors uses its own distinct register set and functional units.

Communication between the two processors and main memory is done through three queues. Addresses for data fetching are placed into the "Address Read–Data Queue" (ARDQ) by the
address generator unit. Data received from memory are placed into the “Data Read–Data Queue” (DRDQ). The processing unit removes data from the DRDQ when it needs to process them, and the generated results are placed into the “Store–Data Queue” (STDQ) were they are paired with addresses previously generated by the address generator unit. These pairs are stored back to main memory.

The two processors can communicate with each other either through main memory or through the use of two additional queues. The first holds information produced by the processing unit and consumed by the address generator, while the second contains information generated by the address generator and used by the processing unit.

Execution flow, on such architectures, is as follows: Initially the address generator unit generates the addresses of data, required by the processing unit and places them into the ARDQ, from which the memory subsystem receives them whenever it is ready to do so. Concurrently, the address generator generates the addresses of the memory locations were the processing results will be stored and places them into the STDQ. Whenever previously requested data are produced by the memory, they are placed into the DRDQ. The processing unit removes data from the DRDQ, operates on them, and places the produced results into the STDQ, where they are paired with their previously generated addresses. These pairs are then removed from the STDQ and actually stored to memory.

Consider, for example, the execution of the C code fragment shown in figure 1.2.

\[
\text{int a[16,1000000],b[1000000]};
\]
\[
\text{for (i=0; i<1000000; i++)}
\]
\[
\text{b[i]=a[0,i]*3;}
\]

**Figure 1.2:** multiply the elements of a column of an array by 3

This code multiplies the elements of the first column an array a[] by 3 and stores the results into an array b[]. The program executed in the address generator produces the addresses of the
elements of the first column of array a[] and those of the b[] array elements. First, the addresses of elements a[0,0] and b[0] are produced, then those of a[0,1] and b[1], and so on until the addresses of all the elements are generated. The addresses that refer to the a[] array are placed into the ARDQ and used for fetching data. The addresses that refer to the b[] array are placed into the STDQ. The program executed in the processing unit multiplies the data received from memory (through the DRDQ) by 3, and places the results into the STDQ, where they are paired with the addresses of the array b[] elements.

<table>
<thead>
<tr>
<th>time</th>
<th>Address generator action</th>
<th>Processing unit action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>request a[0,0], generate &amp;b[0]</td>
<td>wait</td>
</tr>
<tr>
<td>1</td>
<td>request a[0,1], generate &amp;b[1]</td>
<td>wait</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>request a[0,n+1], generate &amp;b[n+1]</td>
<td>a[0,0] * 3</td>
</tr>
<tr>
<td>n+1</td>
<td>request a[0,n+2], generate &amp;b[n+2], store b[0]</td>
<td>a[0,1] * 3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2n+2</td>
<td>request a[0,2n+3], generate &amp;b[2n+3], store b[n+2]</td>
<td>a[0,n+3] * 3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 1.1: Execution of the program of figure 1.2 on the simple access decoupled architecture presented

Assuming a memory access delay of n cycles, the execution flow is shown in table 1.1. Initially, at time 0, all queues are empty. As there are no data available in the DRDQ, the processing unit waits. At the same time, the address generator generates the addresses of a[0,0] and b[0] and places them into the ARDQ and STDQ respectively. The processing unit continues to wait for n - 1 cycles, and, at time n, the first element of array a[] is received from memory and placed into the DRDQ. During these cycles the address generator continued to generate the addresses of the a[] and b[] array elements. As a result, as soon as the processing unit multiplies the first element and places it into the STDQ, the second element is received from memory and can be multiplied. Execution continues in the same way, and, at time 2n + 2, 2n + 4 elements have been requested from memory, n + 4 elements have been multiplied, and, n + 3 among them have been sent to memory.

The execution of the same algorithm on a conventional architecture that employs a simple cache memory is shown in table 1.2. At time 0, the conventional architecture requests the first element of the a[] array from the cache memory. As none of the elements of the array a[] are initially into the data cache, a cache miss occurs. So, the processing unit waits for n cycles, until at time n, that element is received from memory. At time n + 1, the multiplication of a[0,0] completes and a store into b[0] is initiated. Assuming that the cache has a fairly deep write buffer for write misses, at time n + 2, the second element from the array a[] is requested. Assuming that the array a[] is stored in a row major form and that it is not possible to store all the elements of an row of this array within a cache block, as before a read miss occurs, and again n cycles are lost, waiting for the memory. As shown, at time 2n + 2, only 3 elements have been requested from memory and 2 elements have been multiplied and sent to it.

Note that, the benefits gained by a superscalar conventional architecture wouldn’t be great. Although in such an architecture it would be possible to request an element of array a[] concurrently with the multiplication of a previous one, it will still be required to wait for the memory between successive access requests.
1.1. Access Decoupling.

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>request a[0]</td>
</tr>
<tr>
<td>1</td>
<td>wait</td>
</tr>
<tr>
<td>..</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>b[0] = a[0] * 3</td>
</tr>
<tr>
<td>n+1</td>
<td>request a[1]</td>
</tr>
<tr>
<td>..</td>
<td>...</td>
</tr>
<tr>
<td>2n+1</td>
<td>b[1] = a[1] * 3</td>
</tr>
<tr>
<td>2n+2</td>
<td>request a[2]</td>
</tr>
</tbody>
</table>

**Table 1.2:** Execution of the program of figure 1.2 on conventional architectures

The access decoupled architecture executes the program of figure 1.2 in about \( n + 1000000 \) cycles while the execution in a conventional architecture requires about \( n \times 1000000 \) cycles. As a function of both memory delay and array size, execution time becomes \( n + \text{elements} \) and \( n \times \text{elements} \) respectively. The ratio of the cycles required on the two architectures is approximately:

\[
\frac{\text{Cycles}_{\text{AccessDecoupled}}}{\text{Cycles}_{\text{Conventional}}} = \frac{1}{\text{elements}} + \frac{1}{n}
\]

Clearly the access decoupled architecture is faster, and the performance difference increases with the size of the data set and the memory access delay. The cache memory technique fails for this example since none of the elements processed is referenced more than once and the access stride is larger than the cache block size. In contrary, access decoupling is proved efficient, as the address generator manages to request the data that the processing unit requires, at least \( n \) cycles before they are actually required. As a result of applying access decoupling, the processing unit sees only an initial \( n \) cycle delay in the arrival of the first element and no delays in the reception of the rest of them. Figure 1.3 illustrates the execution flow in access decoupled architectures in a general, ideal situation. Initially the processing unit waits while the address generator unit generates addresses and requests data from main memory. As soon as the first data are available the processing units starts execution and proceeds executing without any additional delays, since new data are continuously received from memory.

---

**Figure 1.3:** Execution progress on access decoupled architectures

If the elements of a row of array \( a[] \) were to be multiplied then the performance of a cache memory based architecture will be better. In fact if \( m \) elements can be prefetched within a cache block then the execution time would be \((n - 1) \times \frac{\text{elements}}{m} + \text{elements}\), that is the full \( n \) cycles of delay are seen only every \( m \) elements. Looking at the factor of the execution time on
this architecture and on the access decoupled architecture, we see that the latter is faster when
the number of elements is large enough (at least \(m+1\)). For a small number of elements, the
two architectures have similar performance. There is a similarity between data prefetching and
access decoupling for this example. This holds only for data fetch sequences that refer to data
stored in consecutive memory locations. If the required data are not stored in that way, as for
example the elements of multidimensional arrays, data prefetching cache memories fail while
access decoupling remains efficient.

As the next example will illustrate, it is not always possible to benefit by the application
of access decoupling. In fact the execution performance of access decoupled architectures is
remarkably lower than that of cache memory based architectures for some algorithms. Consider,
for example, the execution of the C code fragment shown in figure 1.4.

```c
binary_tree t;

while (t.key!=search_key)
    if (t.key<search_key) t=t.right_subtree;
    else t=t.left_subtree;
```

**Figure 1.4**: Search in a binary tree for a key value

The execution flow on a conventional architecture is shown in table 1.3, while table 1.4
illustrates the execution flow on an access decoupled architecture.

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>request t</td>
</tr>
<tr>
<td>1</td>
<td>wait</td>
</tr>
<tr>
<td>(n)</td>
<td>...</td>
</tr>
<tr>
<td>(n+1)</td>
<td>compare t.key with key</td>
</tr>
<tr>
<td>(n+2)</td>
<td>(t = \text{request right or left subtree} )</td>
</tr>
<tr>
<td>(n+3)</td>
<td>wait</td>
</tr>
<tr>
<td>(2n+2)</td>
<td>...</td>
</tr>
<tr>
<td>(2n+3)</td>
<td>compare t.key with key</td>
</tr>
<tr>
<td>(2n+4)</td>
<td>(t = \text{request right or left subtree} )</td>
</tr>
<tr>
<td>(3n+4)</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>compare t.key with key</td>
</tr>
</tbody>
</table>

**Table 1.3**: Executing a binary tree search on conventional architectures

The execution performance on the two architectures is similar. It is expected that cache
memory based architectures will actually execute this algorithm faster than access decoupled
architectures as whenever two consecutively fetched tree nodes are placed at adequately near
memory locations cache misses will not occur. On the other hand, on access decoupled
architectures the address generator spends most of its time waiting for the processing unit to
compare previously fetched data, as the location of the next to fetch tree node depends on the
outcome of that comparison. If the key value of the current processed tree node is larger than the value searched then the left child should be fetched next. But if it is smaller then the right child node should be fetched.

<table>
<thead>
<tr>
<th>Time</th>
<th>Address Generator</th>
<th>Processing Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>request t</td>
<td>wait</td>
</tr>
<tr>
<td>n</td>
<td>wait</td>
<td>Rx = compare t.key with search key</td>
</tr>
<tr>
<td>n+1</td>
<td>if (Rx) request right subtree else request left subtree</td>
<td>wait</td>
</tr>
<tr>
<td>n+2</td>
<td>wait</td>
<td>wait</td>
</tr>
<tr>
<td>2n+2</td>
<td>wait</td>
<td>Rx = compare t.key with search key</td>
</tr>
<tr>
<td>2n+3</td>
<td>if (Rx) request right subtree else request left subtree</td>
<td>wait</td>
</tr>
<tr>
<td>2n+4</td>
<td>wait</td>
<td>wait</td>
</tr>
<tr>
<td>3n+4</td>
<td>wait</td>
<td>Rx = compare t.key with search key</td>
</tr>
</tbody>
</table>

Table 1.4: executing a binary tree search on access decoupled architecture

As the two examples presented illustrate, it is not possible to efficiently execute all algorithms on access decoupled architectures. The means by which an algorithms is evaluated as suitable for execution on access decoupled architectures are the data fetch/store sequences produced. Algorithms that fetch and store data at locations that are independent of the actual data values are suitable for execution on access decoupled architectures. If the location of the next access is decided based on the values of previously processed data, it is not likely that the algorithm will benefit from access decoupling. In the first case, the address generator can continue generating addresses for consecutive memory accesses concurrently with the processing of "older" data in the processing unit. On the other hand, in the later case, the address generator has to wait for the processing unit to operate on previously requested data, since the location of the next access depends on the outcome of that processing.

Concluding, it can be said that access decoupling can reduce or hide executions delays due to memory access. Even algorithms that have poor locality of reference on their data space, can benefit from this technique. Unfortunately, not all algorithms can be executed efficiently on access decoupled architectures: only those having data independent data fetch and store sequences will. The execution performance of access decoupled architectures is comparable to that of cache memory based architectures for algorithms that have heavily data dependent fetch/store sequences; in addition, if the algorithm exhibits some locality of reference, the cache–based architecture will perform significantly better.

1.2 Software Pipelining

Software pipelining [2], [3] is a scheduling technique in which iterations of a loop are continuously initiated at constant intervals, before the preceding iterations complete. It is aimed for pipelined architectures that have multiple functional units available. Software pipelining tries to better utilize the system resources in order to increase the loop throughput.
The main idea of this technique can be illustrated through the following example. Consider the C code fragment shown in figure 1.5. This code multiplies all the elements of array a[] by three.

\[
\begin{align*}
\text{for}(i=0; i<1000; i++) \\
a[i] &= a[i]*3.0;
\end{align*}
\]

**Figure 1.5:** A C program suitable for software pipelining

Assuming that 2 cycles are required for the multiplication (the multiplier is two stage pipelined), the minimum sequence of instructions for this code is:

1: Load  
2: Mul  
3: Noop  
4: Store

Without software pipelining, the maximum throughput of the above loop is 1 multiplication per 4 cycles. In the case of a superscalar architecture in which the three instructions can be served by different functional units, the available resources are very poorly utilized. As the outcome of each iteration does not depend on those of preceding iterations, a new iteration can be initiated immediately after initiating the previous one. And, as there is no competition for using the system resources, it is possible to initiate an iteration every cycle. This is the optimal throughput for the loop of figure 1.5. Figure 1.6 shows the code necessary for initiating a new iteration every cycle.

\[
\begin{align*}
1 & : \text{Load} \\
2 & : \text{Mul Load} \\
3 & : \text{Mul Load} \\
4 & : \text{Store Mul Load brtop 4} \\
5 & : \text{Store Mul} \\
6 & : \text{Store} \\
7 & : \text{Store}
\end{align*}
\]

**Figure 1.6:** Initiating a new iteration every cycle

Execution using software pipelining is separated into three parts or stages as usually called. In the Prolog stage a new iteration is initiated and executed concurrently with preceding iterations. During this stage none of the initiated iterations completes its execution. In the above example, the Prolog consists of instructions 1 to 3. In the Steady stage that follows, as in the Prolog stage, a new iteration is initiated and executed concurrently with previously initiated iterations. The difference is that the initiation of a new iteration coincides with the completion of a previous initiated one. In the above example, the steady stage is reached at the fourth instruction. During this stage four iteration are concurrently executed and the loop throughput is the maximum possible. The brtop [3] branches back to the fourth instruction, without delays,
1.2. Software Pipelining

until the end of the a[] array is reached. The semantics of this instruction will be explained later in the discussion of the hardware assistance required for the application of software pipelining. In the third and last stage (instructions 5–7), which is called Epilog, no new iterations are initiated. In this stage the execution of previous initiated iterations completes.

In this scheme, utilization of the available pipelined functional units is better than in the traditional program execution, since the pipelines are not emptied at iteration boundaries. Instead, the pipelines are filled during the Prolog stage, their utilization remains in a steady state during the Steady stage, and finally they are drained during the Epilog stage. This is illustrated in figure 1.7. Optimal utilization of the available resources and of the loop throughput is possible with software pipelining.

![Figure 1.7: Execution stages using software pipelining](image)

The objective of software pipelining is to minimize the interval at which iterations are initiated, as this Initiation Interval (II) determines the throughput of the loop.

Two lower bounds exist on the initiation interval and they are determined by the existing resource and precedence constraints. The most heavily utilized resource determines the first. If a new iteration is initiated every \( n \) cycles and if the body of each iteration consists of \( i \) instructions, then in the Steady stage \( \left[ \frac{i}{n} \right] \) iterations execute concurrently. The resource requirements of the concurrently executing iterations cannot exceed the available resources. If each iteration uses a resource \( r \) \( t_r \) times during each loop iteration, and if there are \( n_r \) copies of this resource, then [3]:

\[
\text{Resource Constraint minimum } II = \max \text{ over all resources } \left( \frac{t_r}{n_r} \right)
\]

If the execution of an iteration depends on the outcome of the execution of a preceding iteration, then a new iteration cannot be initiated before the instructions of that previous iteration
complete. Also, instructions of the same iteration that depend on each other cannot execute concurrently. Consider for example the code of figure 1.8.

```c
int a[100], min;
min = a[0];
for (i = 0; i < 100; i++)
  if (a[i] < min) min = a[i];
```

**Figure 1.8:** Minimum element of a vector

The precedence constraints that exist on this loop are the following: First, the value of a[i] must be accessed, before the comparison can take place; Updating the minimum value has to wait for the comparison to complete; and finally, the comparison of an iteration cannot be initiated before the minimum value of the immediately preceding iteration has been updated.

The recurrence minimum initiation interval (RMII) for a loop can be derived from its *data dependence graph* (DDG). This is a graph representation of the loop body in which nodes correspond to operations and edges correspond to dependencies between operations. Edges are labeled with pairs ($l_i, d_i$), where $l_i$ is the latency of the parent operation and $d_i$ is the distance, measured in iterations, of the two operations connected via this edge. Cycles represent dependencies between iterations, while simple paths represent dependencies within an iteration. Figure 1.9 shows the DDG for the example of figure 1.8. The compare instruction cannot be initiated concurrently with the Load instruction of the same iteration. Also, updating the minimum value must wait for the comparison to complete. Finally the comparison of an iteration must be at least 2 cycles apart from the updating of the minimum of its immediately previous iteration.

![DDG Example](image)

**Figure 1.9:** A data dependence graph example

A closed formula for the computation of the Recurrence Constraint Initiation Interval (RCII) exists. If a dependence edge $i$ of the DDG has latency $l_i$ and connects operations that are $d_i$ iterations apart, then [3]:

1.2. Software Pipelining

Recurrence minimum \( II = \operatorname{Max. over all cycles} \left( \frac{\sum_{\text{cycle edges}} l_i}{\sum_{\text{cycle edges}} d_i} \right) \)

The minimum initiation interval is the maximum of the resource constraint minimum initiation interval and of the recurrence minimum initiation interval.

\[ MII = \min(RCMI, RMII) \]

A loop body on which software pipelining is applied is divided into stages, each consisting of \( II \) cycles. The stage count (SC) is the number of stages. If the length of the loop schedule is \( n \) cycles, then the stage count, is \( SC = \lceil \frac{n}{II} \rceil \). Both the length of the prolog stage and the epilog stage can be expressed by means of the stage count and of the initiation interval. The first \((SC - 1) \times II\) cycles constitute the prolog stage. The last \((SC - 1) \times II\) constitute the epilog stage.

Although software pipelining can be applied without special hardware support, software-only implementations have certain disadvantages. Without hardware support, additional instructions are required for both the prolog and the epilog stages. Also, applying software pipelining to loops that include conditional constructs or have multiple exits is difficult, because control transfer instructions produce different delays when taken or not taken.

With hardware support, a single piece of code can be used to generate all iterations of a loop and conditional constructs can be executed in a uniform way without variable delays. Although various implementations may exist, we adopted an approach similar to that presented in [3]. In this approach, a simple machine model is used. As figure 1.10 shows, several pipelined functional units are available. An instruction can initiate one operation on each of these functional units. The control unit handles control transfer and loop control operations. Conventional machines usually have only one register file available. In this machine two register files are available, the static register file and the rotating register file. References to the static register file are absolute, that is static register \( sr0 \) always refers to the first register of the static register file. On the other hand, references to the rotating register file are relative to the current value of the \( MCP \) (multiconnect pointer) register. That is, the offset of the register is added to the value of MCP modulo the size of the file and the result is used as an index to the rotating register file. The rotating register file can be viewed as a continuous cycle of registers.

A predicate file is also available. Predicates are single bit values that enable or disable operations issued to functional units. Each operation within an instruction has its own predicate. An operation whose predicate has a zero value is not initiated while those that have predicates whose values are '1' are initiated and executed by the functional units. This way of instruction execution validation is called conditional execution.

Predicates, also referred to as guards, are useful in implementing conditional constructs. Consider the previous example of minimum element computation. The code using conditional execution is shown in figure 1.11.

It is assumed that \( g1 \) always contains '1', so all instructions guarded by \( g1 \) are always executed. The instruction guarded by \( g2 \) is executed only if min is greater than \( a[i] \), since the value of \( g2 \) is updated by the compare instruction. The advantage of conditional execution is that no control transfer instructions are required for the implementation of conditional constructs. This leads to similar execution traces, independent of the condition status, which is a requirement for software pipelining. Also, execution delays due to branches are eliminated, which is extremely useful in heavily pipelined architectures where branch costs are high.
**Figure 1.10:** Machine model for hardware support of software pipelining

- **g1:** Min = Read a[0]
- **L:**
  - **g1:** Read a[i]
  - **g1:** g2 = (a[i] < min)
  - **g2:** min = a[i]
  - **g1:** branch L

**Figure 1.11:** Code for the minimum element computation using conditional execution
Two special loop control operations are supported. The *brtop* operation is used to overlap loops whose iteration count is known, such as "for" loops and *wtop* is used for loops whose iterations count is decided during loop execution, such as "repeat – until" or while loops. The semantics of the two instructions are shown at figure 1.12. Two special purpose registers are used. The loop count (LC) register is used by the brtop operation and holds the number of loop iterations that remain to be executed. The Epilog stage counter (ESC) is used by both wtop and brtop operations during the epilog stage execution of a loop. Both instructions initially decrement the MCP pointer in order to provide a new register environment for the next iteration. Brtop initiates a new iteration by asserting the predicate of the next iteration (next_pred=1) only if the LC register has not reached zero. Wtop initiates a new iteration only if the loop’s condition holds and if the Epilog stage has not been reached. After LC reaches zero or if the wtop operation is used, when the loop’s condition doesn’t hold, the epilog stage execution occurs. No new iterations are initiated during this stage (next_pred=0). As ESC reaches zero, the epilog stage completes and so does the loop execution.

![Figure 1.12: Semantics of brtop and wtop operations](image)

Execution on this machine can be illustrated through the example code of figure 1.13. Figure 1.14 shows the machine code.

```c
float a[100];

for (i=0; i<100; i++)
    if (a[i]>0) a[i]=3.0*a[i];
```

*Figure 1.13: A simple C code*

First the LC register is loaded with the number of iterations required. This is 99, since the first iteration is always executed and the loop should be repeated 100 times. Assume that this code is to be executed on a machine that has an integer unit with a 2 cycles latency, a floating point unit with a 3 cycles latency and a memory port with 5 cycle latency. A
g1 : LC = 99
g1 : ai = &a[0]
g1 : gi = true

LOOP:

\[
\begin{align*}
  &gi : vi = \text{read}(ai) \\
  &gi : ai+1 = ai + 4 \\
  &gi : ti = ai > 0 \\
  &gi : ti = ti \text{ and } gi \\
  &ti : vi = vi \times 3 \\
  &ti : \text{write } vi \text{ to } ai \\
  &g1 : gi+1 = \text{brtop}
\end{align*}
\]

**Figure 1.14:** Machine code for the program of figure 1.13.

A viable schedule for this code is shown in table 1.5.

<table>
<thead>
<tr>
<th>time</th>
<th>Integer unit</th>
<th>Floating point Unit</th>
<th>Memory unit</th>
<th>Control unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>add ai,4</td>
<td>compare ai,0</td>
<td>read ai</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>and gi,ti</td>
<td>mul vi,3</td>
<td>write vi</td>
<td>brtop</td>
</tr>
</tbody>
</table>

**Table 1.5:** Schedule for the code of figure 1.14

The overlapped schedule, where a new iteration is initiated every 3 cycles, is shown in table 1.6. Although each iteration issues at most two operations to each functional unit and there are no recurrences, it is not possible to initiate a new iteration every two cycles, as in such a case resource competition occurs; for example the add operation of a new iteration will be called to execute concurrently with the and operation of a previous iteration. The minimum initiation interval for this loop is bounded by resource constraints. As the loop body execution requires 14 cycles and the initiation interval is 3, the stage count becomes 5. So the ESC register is initially loaded with that value.

The prolog stage consists of cycles 0 to 11; new iterations are continuously initiated every three cycles but none of these completes its execution during those cycles. Cycles 12 to 299 constitute the Steady stage. During these cycles five iterations execute concurrently. As LC reaches zero at cycle 300, Epilog stage begins. During the cycles 300 to 314 no new iterations are initiated and the last five iterations complete their execution.

This overlapped schedule is produced by a simple piece of code. The key in overlapping the execution of iterations lies on the guard selection for each operation of the loop. The code for this example is shown in table 1.7. The schedule of table 1.5 is used to determine the guards of each operation. An operation that executes in the ith stage of the loop body is guarded by the ith guard; for example the add and read operations are executed in the first cycle of the loop and they are guarded by the first guard (offset +0 from were MCP points) while the write operation is guarded by the fifth available guard (offset +4 from were MCP points) as it is executed in the 14th cycle of the loop's body execution and there are 5 stages. The code used
<table>
<thead>
<tr>
<th>cycle</th>
<th>Integer unit</th>
<th>Floating point unit</th>
<th>Memory unit</th>
<th>Control unit</th>
<th>brtop guard</th>
<th>LC</th>
<th>ESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>add $a_{0,4}$</td>
<td>read $a_0$</td>
<td>brtop</td>
<td>000001</td>
<td>99</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>000001</td>
<td>99</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>000001</td>
<td>99</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add $a_{1,4}$</td>
<td>compare $a_{0,0}$</td>
<td>read $a_1$</td>
<td>brtop</td>
<td>000011</td>
<td>98</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>000011</td>
<td>98</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>000011</td>
<td>98</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>add $a_{2,4}$</td>
<td>compare $a_{1,0}$</td>
<td>read $a_2$</td>
<td>brtop</td>
<td>000111</td>
<td>97</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>000111</td>
<td>97</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>000111</td>
<td>97</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>add $a_{3,4}$</td>
<td>mul $a_{0,3}$</td>
<td>read $a_3$</td>
<td>brtop</td>
<td>001111</td>
<td>96</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>compare $a_{2,0}$</td>
<td></td>
<td>001111</td>
<td>96</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>001111</td>
<td>96</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>add $a_{4,4}$</td>
<td>mul $a_{1,3}$</td>
<td>read $a_4$</td>
<td>write $a_0$</td>
<td>011111</td>
<td>95</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>compare $a_{3,0}$</td>
<td></td>
<td>brtop</td>
<td>011111</td>
<td>95</td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td>011111</td>
<td>95</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>$i^3$</td>
<td>add $a_{i,4}$</td>
<td>mul $a_{i-3,3}$</td>
<td>read $a_i$</td>
<td>write $a_{i-4}$</td>
<td>111111</td>
<td>99-i</td>
<td>5</td>
</tr>
<tr>
<td>$i^3+1$</td>
<td></td>
<td>compare $a_{i-1,0}$</td>
<td></td>
<td>brtop</td>
<td>111111</td>
<td>99-i</td>
<td>5</td>
</tr>
<tr>
<td>$i^3+2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>111111</td>
<td>99-i</td>
<td>5</td>
</tr>
<tr>
<td>300</td>
<td></td>
<td>mul $a_{96,3}$</td>
<td>write $a_{95}$</td>
<td>brtop</td>
<td>111110</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>301</td>
<td></td>
<td>compare $a_{98,0}$</td>
<td></td>
<td></td>
<td>111110</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>302</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>111110</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>303</td>
<td></td>
<td>mul $a_{97,3}$</td>
<td>write $a_{96}$</td>
<td>brtop</td>
<td>111000</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>304</td>
<td></td>
<td>compare $a_{99,0}$</td>
<td></td>
<td></td>
<td>111000</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>305</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>111000</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>306</td>
<td></td>
<td>mul $a_{98}$</td>
<td>write $a_{97}$</td>
<td>brtop</td>
<td>110000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>307</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>110000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>308</td>
<td></td>
<td>mul $a_{99}$</td>
<td>write $a_{98}$</td>
<td>brtop</td>
<td>110000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>309</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>110000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>310</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>110000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>311</td>
<td></td>
<td>mul $a_{99}$</td>
<td>write $a_{99}$</td>
<td>brtop</td>
<td>100000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>312</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>313</td>
<td></td>
<td></td>
<td></td>
<td>brtop</td>
<td>100000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>314</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1.6: Overlapped schedule
to generate the iterations of the loop consists of $II$ instructions. An operation that executes in the $i$th cycle of the loop body is placed into the $(i \ MOD \ SC)$ instruction of this code.

<table>
<thead>
<tr>
<th>instruction</th>
<th>Integer unit</th>
<th>Floating point Unit</th>
<th>Memory unit</th>
<th>Control unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>$g_{i+0} : \text{add}$</td>
<td>$g_{i+3} : \text{mul}$</td>
<td>$g_{i+0} : \text{read}$</td>
<td>$g_{i+1} = \text{brtop}$</td>
</tr>
<tr>
<td>2:</td>
<td>$g_{i+2} : \text{and}$</td>
<td>$g_{i+1} : \text{compare}$</td>
<td>$g_{i+4} : \text{write}$</td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1.7: The machine code for the overlapped schedule

As shown, software pipelining is an efficient scheduling technique which permits the concurrent execution of multiple iterations of loop constructs. It is aimed to be used on heavily pipelined architectures that contain multiple functional units. The main effort in applying software pipelining is done by the compiler. Simple hardware mechanisms can also help in the execution of software pipelined code. Such techniques lead to compact code, eliminating the need for additional instructions for the implementation of the prolog and epilog stages, while, through the use of conditional execution, even loops that include conditional constructs in their body can easily benefit from software pipelining.

Software pipelining imposes certain constraints to the loops on which it can be applied on. The schedule for each individual iteration must be the same in order to ensure a compact steady stage, while it is difficult to apply software pipelining on loops that have complex conditional constructs. Also, loops that contain recurrences are not likely to benefit the most from software pipelining.

Finally, it can be said that benefits gained by the application of software pipelining, as with access decoupling, increase with the size of the data set of the algorithm that they are applied on. More data need additional iterations to be processed, and this means that more iterations will be executed in the steady stage during which the system resources are better utilized. As the length of prolog and epilog stages – during which the utilization of the system resources is not so good as it is during the steady stage – doesn’t change as the iteration count increases, this means that the fraction of execution time during which the system resources are better utilized is also increased.

1.3 Using both access decoupling and software pipelining

As it was shown, the potential benefits gained by the application of access decoupling or software pipelining are significant and increase with the size of the data set that the programs operate on. Both techniques can be applied simultaneously as access decoupling reduces memory delays and software pipelining increases the throughput of loop execution. Access decoupling is needed as in software pipelining it is not possible to treat memory just like another functional unit, with long latency, because unlike the other functional units memory systems exhibit variable delays. Also access decoupling can be used to increase the execution performance of code that doesn’t belong to loops. It is expected that the performance of architectures that combine both techniques will heavily depend on the algorithm that they are called upon to execute, since we saw that both of them are only beneficial in certain cases. Algorithms that have data independent fetch/store sequences are suitable for access decoupled architectures.

The maximum loop throughput possible through the use of software pipelining is limited by resource and precedence constraints. The resource constraints can be satisfied by increasing
the available functional units. It can be said that resource constraints are actually imposed by the architectural implementations. In contrast, the precedence constraints are imposed by the algorithm itself and they are the same independently of the architecture on which the algorithm will be implemented. So, a general classification of algorithms to suitable or not for the application of software pipelining should be based only on the precedence constraints they contain.

So far, there has been a significant work in applying both techniques to numerical algorithms. One reason for this is that these algorithms are usually suitable for the application of both techniques, because their data fetch/store sequences do not depend on the data values and their loops are without complex conditional constructs and with few recurrences. One other reason is that most of the scientific/engineering problems are solved through the use of numerical algorithms.

In this work we consider the application of these techniques to non-numerical algorithms. Usually, these algorithms have data dependent fetch/store sequences and loops that include complex conditional constructs with many recurrences. Most of the non-numerical algorithms, in the form that they are usually expressed, are not likely to benefit from the application of access decoupling or software pipelining. Possible modifications of these algorithms, in order to achieve our goals, are considered. Also, in order to extend the set of algorithms appropriate for execution on such architecture, simple architectural extensions are discussed.
Our Architecture

The previously shown access decoupled architecture model is of limited use. As this chapter shows, few non-numerical algorithms can be executed efficiently on that architecture. Simple architectural extensions can significantly increase the set of such algorithms that can be executed efficiently and also improve the overall performance of such architectures. An architecture that incorporates such extensions is presented.

2.1 An access decoupled architecture with three address generators

The architecture used through this work is shown in figure 2.1. Compared with the simple access decoupled architecture model shown in the previous chapter, this architecture mainly differs in the number of address generators and memory data queues available. Instead of one address generator which was responsible to generate addresses for both data fetching and storing, three address generators are present. Each of them is dedicated to one of the three data queues available. Two of these queues service data fetching, while the third one is used for data storing purposes. As before, there is one data processing unit.

All four available processors support software pipelining through the simple hardware mechanisms presented in the previous chapter. This choice simplifies the code generation process. Although it would be possible not to support software pipelining on the address generators this would make it difficult to produce code of similar flowchart to that of the processing unit.

Each of the address generators has both a data and an instruction cache memory. The data cache is used to hold informations required in the process of address generation such as loop count, array base address etc. As we showed in the previous chapter, the performance of cache memories is limited by the cache miss ratio. This is because during a cache miss the execution of the address generator is delayed until the memory delivers the required data or instructions, so the address generator sees all the memory access delay. Having a adequately small number of cache misses is significant to the overall performance of the architecture, as an increased number of misses can reduce the benefits gained by the application of access decoupling. Usually, instruction caches have large hit ratios as the locality of reference property is strong in the code address space. In contrary this property is not so strong in the data address space. For loops that have data-independent data fetch/store sequences, the information required for address generation is known before loop execution and can be loaded in the data cache before
Figure 2.1: A decoupled architecture supporting software pipelining

loop initiation. So data cache misses are likely to occur only between loop boundaries. It is also expected that the number of data elements required during the execution of a loop for address generation purposes will be small, and hence the data cache misses will also be few. So, for loops that operate on large data sets, the execution delays due to data cache misses will be a small fraction of the total loop execution time. The way by which the cache memories communicate is not examined in this work.

As figure 2.1 illustrates, the data processing unit has only an instruction cache and no data cache. The only means of communication between this processor and main memory is through the three data queues. Each of the three data queues is serviced through a dedicated memory port. The two memory read ports are used by the two data fetch queues and the one memory write port is used by the data store queue.

Communication between the three address generators and the processing unit is done either through the use of six additional queues or via main memory. These six queues are referred to synchronization queues, because their purpose is to synchronize the execution of the four processors whenever required. There are two synchronization queues between each address generator and the processing unit — one of them passes synchronization information from the processing unit to the address generator, while the other one is used in the opposite direction. Each element of the synchronization queues is one-bit wide. When a processor writes to a synchronization queue, '1' is pushed into it. The value returned when reading from a synchronization queue depends on previous write accesses to that queue: a '0' value is returned if no data are available, while a '1' value is returned and the corresponding element is removed from the queue if data are available. Since the information contained in a synchronization queue is irrelevant (their semantics are very much like to that of semaphores), just keeping a count of
2.1. An access decoupled architecture with three address generators

how many tokens are in a queue at each time is adequate. As it can be seen in the appendices the synchronous queues were implemented as counters. In no circumstance does a read access to a synchronization queue block the execution of the processor that performs it. Writes to a synchronization may temporarily stall execution of the processor that performs them, when no space is available in the queue. The access is granted and execution resumes as soon as space becomes available.

It is expected that the synchronization queues will be used mainly during “while” type loops, i.e. in loops whose iteration count is decided upon during their execution. Consider for example the code of figure 2.2.

```c
char *s;

while (*(++s)!=0)
    process *s;
```

**Figure 2.2:** A simple while loop

One of the address generators will be used to generate the addresses of the characters of string s. Although it is possible to generate these addresses, it is not possible to decide where the string ends without having knowledge of the actual characters, since the end of the string is marked by a zero–valued character. So the address generator cannot decide when it should stop generating more addresses. On the other hand, the processing unit can decide where the string ends by looking at the data received from memory. The synchronization scheme used is as follows: each time the address generator generates the address of another character, it also polls the synchronization queue. As long as there are no data available within that queue the address generator continues to generate addresses for character fetching. In the meanwhile, the processing unit consumes data from the data queue and process them. As soon as it sees the terminating null character, it writes it into the synchronization queue directed to the address generator. The next time when the address generator polls the synchronization queue a ‘1’ value is returned, and owing to that it stops the generation of addresses of string characters. An additional synchronization phase is required in ordered for the additional characters fetched from memory to be flushed off the data queue. The address generator informs the processing unit, via the synchronization queue, that it has stopped generating addresses for the string. Then the processing unit has to flush all the data from the data queue and finally after that to inform the address generator that it can proceed with its execution. Note that the address generator cannot proceed to generate new addresses for consecutive accesses even if partial flushes are supported by the data queue; the processing unit does not know how many elements it has to discard from the data queue. The code for both address generator and processing unit is shown in figure 2.3.

The elements of the three data queues have three fields. The first is used to hold the address, the second field holds the corresponding data, and the third field is used to indicate the state of the element. Elements of these queues can be completely empty, which means that neither address nor data fields hold valid information; or else, only the address field may hold valid information; finally, both the address and data fields may hold valid information. Additionally, elements of the read queues that have a valid address field may or may not have been requested from memory, or they can also have been flushed but not yet received from memory. A detailed
Our Architecture

<table>
<thead>
<tr>
<th>Address generator 0</th>
<th>Processing unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>while synch_from_PU == '0'</td>
<td>while data from queue 0 != 0 process;</td>
</tr>
<tr>
<td>fetch s++;</td>
<td>write to synch_to_AG0;</td>
</tr>
<tr>
<td>write to synch_to_PU;</td>
<td>while synch_from_AG0 == '0' wait;</td>
</tr>
<tr>
<td>while synch_from_PU == '0'</td>
<td>flush all from data queue 0; write to synch_to_AG0;</td>
</tr>
<tr>
<td>wait;</td>
<td>...</td>
</tr>
</tbody>
</table>

---

Figure 2.3: Code for the program of figure 2.2

The state transition diagram can be found in the appendices.

Contrary to read accesses to the synchronization queues, a read access to a data queue may temporarily freeze the execution on the processing unit: the pipeline of the processing unit is stalled \(^1\) whenever data are requested that are not yet available. Execution resumes as soon as the memory delivers the requested data. Write accesses to a data queue may also temporarily freeze the execution of the processor that performs them if space is not available within the queue. Both address generators and the processing unit do write accesses to the data store queue. The address generators write to the address field and the processing unit write the data field of the data store queue.

In the architecture presented the manipulation of the data and synchronization queues is done by an additional unit, the queues controller. This unit handles access to queues requested by the four processors and also interacts with the main memory for data transfers. One additional task performed by this unit is the comparison of addresses for data fetching with the addresses of data to be stored. As the three address generators execute their programs concurrently it is possible that one of them will request a data element that has not yet been stored to memory. In such a case the queues controller delays its request until the store is actually performed.

Supporting two read queues cycle instead of only one has two main advantages:

- on superscalar implementations of the processing unit, it would be possible to efficiently handle the data access requirements as usually the number of data loads is twice that of data stores. For this purpose, only one address generator is sufficient, provided it has a throughput of two (or three) addresses per cycle.

- the set of the algorithms that can be executed efficiently is extended, as explained below. For this purpose, independent address generators are required.

Consider for example the addition of two vectors. During each iteration two data elements are fetched and one is stored. Having two read ports and one write port it is possible to implement the loop with only one instruction and achieve a maximum throughput.

---

\(^{1}\) All the pipeline is frozen if the code currently executing uses software pipelining. On the other hand, if software pipelining is not currently used, not all the pipeline is freezeed as in such a case deadlocks may occur. Only the instruction fetch, the decode and register file access stages are stalled. This permits the completion of previous issued instructions.
2.1. An access decoupled architecture with three address generators

One algorithm that cannot be implemented efficiently if only one address generator is available is the **merging of two sorted sequences**. The code is shown in figure 2.4. If two read queues are available they can be used to hold the elements of the two sequences, one of them holding the elements of array $a[ ]$ and the second one to hold the elements of array $b[ ]$. If only one data queue were available, it wouldn’t be possible to efficiently execute that algorithm, since the next access depends on the outcome of the comparison. This can be illustrated through the use of the following example. Consider that the array $a[ ]$ is the $(1, 3, 4, 9)$ while the array $b[ ]$ contains the $(2, 6, 7, 8)$ and that the in the first iteration the $a[0]$ element is read before the $b[0]$ element. In this case the algorithm will fetch the following sequence of elements: $(1, 2, 3, 4, 6, 7, 8, 9)$. If the array $a[ ]$ was the $(1, 2, 3, 6)$ and the array $b[ ]$ was the $(4, 9, 8, 10)$ then the fetch sequence would be the: $(2, 1, 3, 4, 6, 9, 8, 10)$. In the first example the data queue should contain the $(a_0, b_0, a_1, a_2, b_1, b_2, b_3, a_3)$ sequence, while in the second it should contain the $(a_0, b_0, b_1, b_2, a_1, b_3, a_2, a_3)$. It can be seen that the fetch sequence depends on the values of the fetched data.

```c
int a[], b[], c[];

ia=0; ib=0; ic=0;
while there are elements
    if (a[ia]>b[ib])
        c[ic++]=a[ia++];
    else c[ic++]=b[ib++];
```

**Figure 2.4:** Merge two sorted sequences

In a more formal way it can be said that if $n$ data read queues and $m$ data store queues were available then it would be possible to efficiently execute an algorithm if it were possible to find $n$ data read sequences and $m$ data store sequences such that their combinations include all the possible data read or data store sequences that the algorithm produces. In such a case the data queues are filled with those sequences. So, if more data queues were available the set of the algorithms that can be executed efficiently on the architecture will also be larger. As the cost of each data queue is significant the inclusion of more data queues remains an open issue.

Having dedicated address generators for each data queue, also costs. But it has the following advantage: it permits different address generation rates for each data queue. When using access decoupled architectures with numerical algorithms, the relative rate of data consuming from the data queues is known. So one address generator can generate the addresses for all the data queues, as it knows how many addresses it should generate and when it should place them to each of the data queues. On the other hand, non–numerical algorithms exhibit different and variable data consuming rates from each of the data queues. If only one address generator were provided for the three queues then it would have to poll each of them and whenever space becomes available to generate the corresponding addresses. It would then be required to support in hardware a way in which the address generator can get knowledge of the queue status. Also a portion of its execution time and its instruction bandwidth will be consumed in polling the data queues. Thus, we would need to increase the instruction bandwidth of the address generator in order to sustain an adequately large address generation throughput. Finally generating code for such an architecture will be difficult, as it
would be required to generate one program that merges and produces an overlapped schedule out of three, possibly different, programs that in our architecture execute on the three address generators.
3

Nonnumerical algorithms

As it was previously discussed, the existence of strong recurrences and of data fetch/store sequences that highly value-dependent makes it difficult to efficiently execute nonnumerical algorithms on decoupled architectures that support software pipelining. One way to overcome these difficulties is to modify the algorithms in order to eliminate or reduce the recurrences and produce data independent fetch/store sequences. One can expect that this is not a trivial task. The difficulties arising in modifying and implementing some nonnumerical algorithms, mainly sorting ones, are discussed in this chapter. Sorting algorithms are mainly discussed as they operate on large data sets and also because they constitute time consuming components of important applications such as large data bases.

Initially, the problems of implementing these algorithms are discussed and whenever this is possible, modifications are presented. For this initial discussion it is assumed that the algorithms operate on elementary data types, which can be stored within a register. Later, these implementations will be extended to handle more complex data types.

3.1 Sorting algorithms

In this section we are considered with implementing some well known sorting algorithms on machines that follow the architectural model presented in the previous chapter. Some elementary sorting algorithms are initially considered, followed by advanced sorting methods. Elementary sorting methods are suitable only for small input sizes as they take about $N^2$ steps to sort $N$ randomly arranged items. As normally elementary sorting methods consume less time than advanced ones when sorting a small number of input elements, it is expected that it would be preferable to apply advanced sorting methods only when the number of input elements is adequately large and to switch to simpler methods when the input size is small or in the later stages of the application of an advanced sorting method.

3.1.1 Insertion sort

The pascal-like code for this sorting method is shown in figure 3.1. The input elements are conceptually divided into a sorted and an unsorted sequence. Initially, the sorted sequence consists of only one element which can be chosen to be the first element of the input array. During each step of the algorithm one element is removed from the unsorted sequence and inserted in the appropriate position within the sorted sequence. This insertion is done by
shifting all elements that are larger than the new element by one position. After each step the size of the sorted sequence is increased by one element while the size of the unsorted sequence is reduced again by one element. The above method is applied until the unsorted sequence becomes empty, at which time the sorted sequence contains all the elements.

```
procedure insertion;
    var i,j,u:integer;
    begin
    for i:=2 to N do
    begin
    u:=a[i]; j:=i;
    while a[j-1]>u do
    begin
    a[j]:=a[j-1]; j:=j-1 end;
    a[j]:=u;
    end
    end;
```

Figure 3.1: Insertion sort

Both the data fetch and store sequences, as expressed, are data dependent. Each step of the algorithm initially fetches the next element from the unsorted sequence, namely the \( a[i] \). Then, during the execution of the inner while loop, the elements of the sorted sequence are fetched one after the other until we encounter the first that is smaller than the element new to insert. The data fetch sequence is value-dependent since it is not known in advance how many elements the inner while loop will need or in other words how many elements of the sorted sequence are larger than the element under insertion. The inner while loop also produces the data store sequence. During each of its iterations either an element of the sorted sequence is moved by one position or the element under insertion is placed at its appropriate position. As it is not either known in advance how many times the inner while loop will iterate, it is not known how many elements will be written by it. That makes the store sequence data dependent.

Two alternative methods to produce data independent fetch and store sequences are presented. The first method can be used on architectures that support, in hardware, partial flushes of the data read and store queues. The other method does not require additional architectural support and it is based on a modification of the inner while loop so that it operates on all the elements of the sorted sequence rather than only on part of them. In both methods during each iteration of the outer loop, the address generators fetch all the elements of the sorted sequence and produce all the addresses of the new sorted sequence.

**Method 1:** If partial flushes are supported by the hardware of the data queues, the processing unit can flush the additional elements off the data read queue and discard the excess addresses from the store queue as soon as the new element is inserted. This is possible because it knows how many elements are in the sorted sequence \((i - 1\) elements) and how many elements it has processed before encountering the first element that is smaller or equal than the one to be inserted. As soon as the new element is placed at its appropriate position, the processing unit has to flush \(j - 1\) elements from the data read queue and \(j\) elements from the data store queue. Consider for example that after 4 iterations the input array \(a[]\) contains the following numbers: 2, 3, 6, 8, 9, 4, 10, 1, .... During the fifth iteration, the sixth element, namely "4", has
to be inserted into the sorted sequence which is 1, 2, 3, 8, 9. The address generators will fetch the elements of the sorted sequence and produce the addresses of the first six elements of the array a[]. During its execution the inner while loop will compare “4” with 9 and “8”, and it will copy them one position to the right, since they are larger. When the number “3” is read from the input queue, the processing unit places the new element into the output queue, as the appropriate position to insert it was found. The insertion process has now been completed, and the additional elements fetched into the data read queue and the additional addresses placed into the store queue have to be discarded. Since three elements have been read from the input queue the processing unit knows that it has to discard two additional elements from that queue, namely the “2” and “1”. Also, three addresses from the store queue have to be discarded, as the new sorted sequence consists of six elements and only three of them have actually been written, namely “9”, “8” and “4”. Table 3.1 illustrates the above process.

<table>
<thead>
<tr>
<th>operation</th>
<th>input queue</th>
<th>output queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy 4 into a register</td>
<td>(4, 1, 2, 3, 8, 9)</td>
<td>()</td>
</tr>
<tr>
<td>compare 1, 4</td>
<td>(1, 2, 3, 8, 9)</td>
<td>()</td>
</tr>
<tr>
<td>copy 1</td>
<td>(2, 3, 8, 9)</td>
<td>()</td>
</tr>
<tr>
<td>compare 2, 4</td>
<td>(2, 3, 8, 9)</td>
<td>(1)</td>
</tr>
<tr>
<td>copy 2</td>
<td>(3, 8, 9)</td>
<td>(1)</td>
</tr>
<tr>
<td>compare 3, 4</td>
<td>(3, 8, 9)</td>
<td>(1, 2)</td>
</tr>
<tr>
<td>copy 3</td>
<td>(8, 9)</td>
<td>(1, 2)</td>
</tr>
<tr>
<td>compare 8, 4</td>
<td>(8, 9)</td>
<td>(1, 2, 3)</td>
</tr>
<tr>
<td>insert 4</td>
<td>(9)</td>
<td>(1, 2, 3)</td>
</tr>
<tr>
<td>discard 8 and 9</td>
<td>(9)</td>
<td>(1, 2, 3, 4)</td>
</tr>
</tbody>
</table>

Table 3.1: Insertion to a sorted sequence when partial flushes are supported

Method 2: Alternatively, if partial flushes are not supported by the hardware, they can be simulated by replacing the inner while loop with a for type loop. During this loop, all the elements of the sorted sequence are compared to the element under insertion. Again, those elements that are larger than it are copied to the store queue. As soon as the first element that is smaller (or equal) than the element under insertion is encountered, the new element is placed into the data store queue. All other elements are just copied to the store queue.

Consider the previous example of inserting a “4” into the sorted sequence 1, 2, 3, 8, 9. Again, the address generators fetch the elements of the sorted sequence into one of the read queues and produce the addresses of the first six elements of array a[]. The processing unit first copies the elements “9” and “8” to the store queue, as they are larger than “4”. As soon as the “3” is encountered the element under insertion, “4”, is copied to the data store queue, followed by the “3”, and then the remaining elements of the sorted sequence are copied to the data store queue. With this modification, each step of the algorithm reads all the elements of the sorted sequence and writes all the elements of the new sorted sequence. Thus during the ith step, the first i + 1 elements of the input array are accessed and rewritten. The above process is illustrated in figure 3.2.

It is possible to maintain the same data fetch and store scheme in the case that only global flushes are supported by the hardware of the queues. But, in such a case, the delays between successive iterations will be comparatively large, since the address generators will have to wait
<table>
<thead>
<tr>
<th>operation</th>
<th>input queue</th>
<th>output queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy 4 into a register</td>
<td>(4, 1, 2, 3, 8, 9)</td>
<td>()</td>
</tr>
<tr>
<td>compare 1, 4</td>
<td>(2, 3, 8, 9)</td>
<td>()</td>
</tr>
<tr>
<td>copy 1</td>
<td>(2, 3, 8, 9)</td>
<td>(1)</td>
</tr>
<tr>
<td>compare 2, 4</td>
<td>(3, 8, 9)</td>
<td>(1)</td>
</tr>
<tr>
<td>copy 2</td>
<td>(3, 8, 9)</td>
<td>(1, 2)</td>
</tr>
<tr>
<td>compare 3, 4</td>
<td>(8, 9)</td>
<td>(1, 2)</td>
</tr>
<tr>
<td>copy 3</td>
<td>(8, 9)</td>
<td>(1, 2, 3)</td>
</tr>
<tr>
<td>compare 8, 4</td>
<td>(9)</td>
<td>(1, 2, 3)</td>
</tr>
<tr>
<td>insert 4</td>
<td>(9)</td>
<td>(1, 2, 3, 4)</td>
</tr>
<tr>
<td>copy 8</td>
<td>(9)</td>
<td>(1, 2, 3, 4, 8)</td>
</tr>
<tr>
<td>compare 9</td>
<td>()</td>
<td>(1, 2, 3, 4)</td>
</tr>
<tr>
<td>copy 9</td>
<td>()</td>
<td>(1, 2, 3, 4, 9)</td>
</tr>
</tbody>
</table>

Table 3.2: Insertion to a sorted sequence when partial flushes are not supported

for the processing unit to discard the additional elements from the data queues before fetching the data for the next iteration. An unpredictable number of elements would be lost if the address generators proceeded with the next iteration before the processing unit actually discards the additional elements fetched and generated for the current iteration.

It is expected that implementations of the insertion sort will benefit from the application of software pipelining. A new iteration of the inner while loop can be initiated before preceding iterations complete, as the two elements that have to be compared are known long before. The first is the element under insertion while the other is the next elements from the current sorted sequence. If partial queue flushes are supported by the hardware then each iteration of the inner while loop is independent of its preceding iterations, or in other words no recurrences exist in the data dependency graph of the algorithm and, owing to that, the execution of succeeding iterations can be overlapped. If the second method is adopted, succeeding iteration depend on previous initiated iterations. Actually a new iteration cannot decide if it should copy both the element under insertion and the element compared to when the later happens to be smaller or equal than the new element. It has to be informed if some of the previously initiated iterations has located the appropriate position for the element under insertion; only one of all the iterations should copy the new element to the data store queue. Fortunately the comparison, which is the first operation of each iteration can be initiated independently of the outcome of previous iterations. Only the store operations of each iteration have to be delayed until the previous iterations decide if they should copy or not the new element into the store queue. The iteration body in such a case can be though as being composed of two basic blocks: the Compare and the Copy elements blocks.

In the first block the element under insertion and an element one from the sorted sequence are compared. In the second block, the iteration copies the necessary elements to the data store queue. Those may be both the element under insertion and the element from the sorted sequence, or only the second one. To decide wether it must copy both elements or only one of them the iteration has to use both the outcome of its own comparison and that of the comparison of its preceding iteration. If the current element from the sorted sequence is larger or equal than the element under insertion, while the previous element of the sorted sequence were not, then
3.1. Sorting algorithms

both elements have to be copied to the store queue. If both the current and the previous element of the sorted sequence happen to be larger or equal than the element under insertion then the current iteration has to only copy the element from the sorted sequence, since the element under insertion have been placed in its appropriate position by one of the previous iterations. And finally if the element from the sorted sequence is smaller that that under insertion then it has to be copied to the data store queue. The code for the inner loop is shown in figure 3.2. In this figure the element under insertion is denoted by u, while the element from the sorted sequence is denoted by a[i].

\[
1: \text{smaller}_i = (u > a[i]) \\
2: \text{if not smaller}_i \text{ and smaller}_i-1 \text{ then store } u \\
\text{store } a[i]
\]

Figure 3.2: Iteration body of the insertion process

Both the implementations shown suffer from inefficiencies. Although it is possible to efficiently execute the insertion process, it is not possible to initiate a new insertion before the previous one completes and creates the new sorted sequence. The initiation of a new insertion processes has to wait until the previous insertion completes and the new sorted sequence is actually stored to main memory. This in turn may cause long delays between successive insertions, as the sorted sequence is updated during each iteration and cannot be prefetched. In the worst, case these delays may be twice the access delay of the main memory, which happens when an element currently produced by the processing unit is concurrently requested by one of the address generators.

The efficiency of the method when partial queue flushes are not supported, depends on the input data, as some of the execution time is spent in rewriting part of the sorted sequence over itself. If most of the elements of the sorted sequence happen to be smaller or equal than the element under insertion then most of the execution time is lost during this rewriting process.

As mentioned, insertion sort has a complexity of $N^2$ which make it useful only for small input sizes. Considering that between successive insertions significant delays occur, it can be said that the implementations shown are not so useful when only one array has to be sorted, as these delays it may turn out to be comparable or even longer than the time spent during the execution of the iterations. Insertion sort on access decoupled machines is useful, when more than one small arrays have to be sorted. In such a case, the execution of multiple insertions operating on different arrays can be interleaved. As figure 3.3 shows the first insertion of the first array may be followed be the first insertion of the second array, that may be followed by the first insertion of the third array, and so on. As these successive iterations operate on different arrays, no delays occur due to data not yet being stored back to memory. The elements for the next insertion can be fetched during the execution of the current insertion.

Figure 3.3: Interleaving the execution of mutiple insertion sorts
For better utilization of multiple functional units the possibility of initiating more than one iterations each time has to be explored. Clearly, it is not possible to concurrently initiate more than one iterations of the same insertion process, since when using the first method, the actual number of iterations is not known in advance, while, when using the second approach successive iterations depend to each other. So, the possibility of concurrently initiating multiple iterations, each from a different insertion process has to be explored. As in our architecture there are two data read queues available, the execution scheme of figure 3.4 may be useful. Unfortunately this scheme cannot be used with either of the two methods discussed above unless the second method is modified. The problem is that the sequence of data stores produced when two insertions are concurrently executing is not predictable and thus cannot be generated by the address generators. Using the first method, all iterations write one element each to the store queue, except for the last iteration which flushes a number of elements out of the queue. As the number of iterations is not known, it is not possible to predict the sequence of stores produced when two iterations, each from a different insertion, are concurrently executed. When using the second method all the iterations write one element to the output queue except for one which writes two elements, when the first element larger than or equal to the element under insertion, is encountered.

![Figure 3.4: Concurrent execution of multiple insertions](image)

A small modification of the second method can be used with this execution scheme. As figure 3.5 illustrates, the inner while loop is modified to write only one element in each iteration. Initially the element under insertion is loaded into a register, namely the u register. During the execution of the loop, the value of the u register is compared with elements from the sorted sequence. If the element from the sorted sequence happens to be less than the u value it is copied to the data store queue. If it is larger or equal the u value is copied to the data store queue and the u register is updated with the value of the element from the sorted sequence. In this way only one data store occurs during each iteration. Note that although there seems to exist a recurrence between the comparison of a new iteration and the updating u register of the previous iteration, it is not necessary for the comparison to wait for the updating of the previous u to complete. Even if the comparison uses the old value of u, there is no problem, as the input sequence is sorted. But the recurrence between storing u and updating the u of the previous iteration is important. Consider for example the insertion of “4” into the sorted sequence of (1, 5, 6). The first iteration writes out “1” while the u value remains 4. The second iteration writes out “4” and sets u to “5”. Even if the third iteration is initiated before u takes its new value, “5”, it will compare “4” with “6” and decide that u has to be stored and then updated to “6”. Since the storing of u has to wait for its previous update, “5” will be actually written and not “4”. If the storing were initiated before the completion of the previous one then the “4” will be written twice, while “5” will be lost.

It is worth noticing that only one of the data read queues is necessary for implementing
3.1. Sorting algorithms

\[ u = \text{next element of the unsorted sequence}; \]
for all the elements of the sorted sequence
\[ e = \text{element of sorted sequence}; \]
if \( e < u \) then store \( e \);
else begin store \( u \); \( u = e \); end;
store \( u \);

**Figure 3.5:** A modified insertion code to be used when initiating multiple iterations of different insertions

insertion sort. Consequently, to fully utilize the two read queues in our architecture, multiple arrays have to be sorted. Such arrays may have been produced by the application of an advanced sorting method.

3.1.2 Shellsort

*Shellsort* is an extension of insertion sort. This method rearranges the input array to give it the property that taking every \( h \)th element yields a sorted sequence. The input array is conceptually divided into \( h \) smaller groups, each consisting of the elements that are \( h \) positions apart into the input array. By repeatedly reducing the value of \( h \) and applying insertion sort on the \( h \) element groups the input array is sorted. Shellsort gains speed over insertion sort by allowing exchanges of elements that are far apart.

As the core of this method is the insertion sort, the methods previously presented can be used. Notice that the execution schemes that interleave the execution of multiple insertions are directly applicable when shellsort is used, as by definition multiple arrays are available when \( h \geq 2 \). When \( h \) is reduced to one, there is only one array available and it is not possible to benefit of those execution schemes. In that case, the inefficiencies of insertion sort cannot be overcome.

It is expected that the programs necessary to implement this sorting method will be complicated, as they have to interleave the execution of multiple insertions by software means. On the other hand, the possibility of applying the execution schemes of figure 3.3 and 3.4 make this algorithm an attractive alternative to insertion sort for architectures such as ours.

3.1.3 Selection sort

The pascal-like code for this sorting algorithm is shown in figure 3.6. *Selection sort* is simple and works as follows: First find the smallest of all the input elements and exchange it with the first element. Then find the second smaller and exchange it with the second element and continue in this way until all elements are considered. Then the entire array is sorted.

The code of figure 3.6 has two unpleasant characteristics:

- the position of one of two elements exchanged in each iteration is decided upon at execution time.
- the element used in the comparison in the inner *for* loop changes during the execution of the loop.
procedure selection;
    var i,j,min:integer;
    begin
        for i:=1 to N-1 do
            begin
                min:=i;
                for j:=i+1 to N do
                    if a[j]<a[min] then min:=j;
                exchange(a[min],a[i]);
            end;
    end;

Figure 3.6: Selection sort

Due to the first characteristic, it is not possible to efficiently execute this algorithm on access decoupled architectures. The address generator responsible for the data stores does not know where the minimum element is located, so it cannot produce its address before the processing unit locates it.

As can be seen in the code of figure 3.6, the minimum element is located in the following way: a local variable is used and it is initially set to point to the first input element. Then all elements are considered in turn. Each time an element is found to be smaller than the element where the local variable points to, the local variable is updated to point to that element. So during execution time, the local variable gives the index of the smaller element encountered so far. In this process, a recurrence exists between successive iterations: it is not possible to initiate a new comparison before the previous iteration decides and updates the value of the local variable.

We will consider these two problems separately. We will first show a method to produce a value-independent store sequence. As indexed accesses are not efficient on access decoupled architectures the method of finding the minimum element has to be modified. The local variable can be used in a slightly different way, to hold the value of the smaller element encountered so far, instead of pointing to it. By modifying the inner for loop, it is possible to generate data-independent data sequence as figure 3.7 illustrates.

\[
\begin{align*}
\min &:= a[i]; \\
\text{for } j &:= i+1 \text{ to } N \text{ do} \\
    &\text{if } a[j] < \min \text{ then exchange}(a[j], \min) \\
    &\text{else } a[j] := a[j]; \\
\min &:= \min;
\end{align*}
\]

Figure 3.7: A modified version of the inner for loop of selection sort

This code produces fetch and store sequences that are both data independent. During each iteration of the for loop, the \text{jth} element of the input array is fetched and then overwritten either by itself or by the value of the local \text{min}. Before the for loop initiation, the \text{i}th element is
3.1. Sorting algorithms

fetched, while after the completion of the for loop this element is overwritten by the local min, which, that time, holds the global minimum element.

This code does not produce the same data exchanging as the initial code of figure 3.6, but as far as the placement of the ith smaller element in the ith position is considered, the same results occur if this code is used. And now it is possible to implement and execute this code on access decoupled architectures. An unpleasant effect of this method is that the number of data stores is significantly increased. This must not be considered as an inefficiency as it was assumed that the memory subsystem can accept one store request per cycle.

Unfortunately in the code of figure 3.7 a recurrence exists; a new iteration cannot be initiated before the updation of the local min by the currently executing iteration completes. So it is expected that this code will not be able to benefit from software pipelining. As the inner for loop of the algorithm of figure 3.6 is used to locate the minimum element of a \( a[j], \ldots, a[N] \) we will consider the algorithm which finds the minimum of a sequence of numbers \( (a_0, a_1, \ldots, a_N) \). A modification will be shown in which software pipelining can be used efficiently.

**Finding** \( \text{min}(a_0, a_1, \ldots, a_N) \)

The code with the modifications necessary to make it appropriate for decoupled architectures but not yet for software pipelining is shown in figure 3.8.

```plaintext
min=a_0;
for i=1 to N
  if a_i<min then exchange(a_i,min)
  else a_i=a_i;
  a_0=min
```

---

**Figure 3.8:** Finding the minimum of a sequence \( (a_0, \ldots, a_N) \)

As previously discussed, a recurrence exists between successive iterations: the compare operation of an iteration has to wait for the previous iteration to update the min value. To overcome this problem, we can use the following property of the min() operator:

\[
\text{min}(\text{set}_A) = \text{min}(\text{min}(\text{subset}_1), \text{min}(\text{subset}_2), \ldots, \text{min}(\text{subset}_x))
\]

We can conceptually partition the input sequence in an adequately large number of subsets and find the minima of each of them. Software pipelining can now be applied, interleaving the execution of these processes in the way illustrated in figure 3.9.

An additional stage is required at the end in order to determine which is the minimum of all the elements. In this stage, the minima of the subsets are compared in pairs. The larger of the two elements compared is discarded each time and this proceeds until only one element is left. Note that the number of comparisons required during this stage will be small and that they can be executed concurrently. For example if 4 subsets were used then the final stage can be scheduled as follows:

The number of subsets required for efficient execution depends on the architecture used and mainly on the pipeline delays for the compare operation. It is expected that for typical pipelines
with latencies of 3 to 5 cycles, 3 to 5 subsets will suffice and this means that the additional stage to find the global minimum will be kept small.

Returning to the selection-sort discussion, it is now clear that the inner for loop of the algorithm can benefit from software pipelining if the above method of finding the minimum element is used.

Its $N^2$ complexity makes selection-sort usefull only for small input sizes. As with insertion sort, this method is only of interest when used in the last stages of the application of an advanced method.

Selection-sort is also appropriate for sorting multiple arrays concurrently, in order to achieve better utilization of the available functional units. There is no problem in producing both store and fetch sequences that are necessary for interleaving the execution of multiple selection-sort; each iteration always reads and writes the same number of elements. Also, in such a case, it is not necessary to split the input arrays into subsets in order to efficiently compute the minimum element. It is possible to interleave multiple $\min()$ operations, each operating on a different array.

### 3.1.4 Bubblesort

Bubblesort is one of the simplest sorting methods. The pascal-like code is shown in figure 3.11. The problems arising in implementing this algorithm on our architecture are similar to those found in selection-sort:

- two elements are possibly exchanged in each iteration of the inner for loop.
- the value of one of the elements used by the comparison operation of an iteration may change during the execution of a previous iteration.

The first point makes the store sequence data dependent, while the second point imposes a recurrence between successive iterations. The methods that can be used to modify the code of bubblesort in order to overcome these problems are somewhat similar to the methods used in selection sort. We can modify the inner for loop so that each iteration of it reads only the
procedure bubble;
var i,j:integer;
begin
for i:=N downto 1 do
  for j:=2 to i do
    if a[i-1]>a[j] then
      exchange(a[i-1],a[j])
end;

Figure 3.11: Bubblesort

A closer look to the bubble sort code reveals that it is similar to the implementation method discussed for the selection-sort. In fact, after each step of bubble sort the maximum of the "remaining" elements is placed at its final position. The implementation of selection sort discussed above is similar as in each step it places the minimum element at its final position. So, the implementation seen can also be thought as a modification of bubblesort (in fact it is closer to bubblesort than it is to selection-sort).

A theoretical analysis of bubble sort shows that it does much more work than selection-sort to get each element into position. This is because bubblesort performs more exchanges during its execution. The performance of the implementations shown differs from the performance of the two algorithms when executed in conventional architectures. This can be explained if one thinks that in a decoupled architecture the cost of memory accesses can not be counted in a way similar to that used in conventional architectures. In a conventional architecture it is preferable to avoid memory accesses wherever possible. In contrary, on decoupled architectures, in most of the cases, it is preferable to do additional memory accesses in order to maintain a data independent fetch and store sequence.

3.1.5 Mergesort

Mergesort is the first advanced $N\log N$ sorting method considered in this work. There are two major alternatives of this algorithm, namely straight mergesort and natural mergesort.

Straight mergesort works as follows: the input array is considered as composed of many small sorted subarrays. Initially each subarray consists of only one element. Pairs of those subarrays are merged into larger sorted arrays. This merging process is repeated until only one array is left.

Natural mergesort differs only in the way that the input array is initially partitioned into sorted subarrays: Instead of using subarrays of only one element the input array is first scanned to locate already sorted sequences. These sorted sequences are then merged in pairs until only one is left.

Both methods employ the process of merging, which is illustrated in figure 3.12. An efficient implementation of this algorithm is essential for both methods of mergesort.
int a[], b[], o[];
int ai, bi, oi;

ai = 1; bi = 1; oi = 1;
while there are elements in either a[] or b[]
  if a[ai] > b[bi] then o[oi++] = b[bi++];
  else o[oi++] = a[ai++];

Figure 3.12: Merge two sorted arrays

The store sequence is fortunately data independent, as at each iteration a new element is stored into the next position of the output array. On the contrary, the data fetch sequence is value-dependent. Each iteration may read a new element from either input array a[] or from input array b[] and the decision is made by the previous iteration: if an element from the array a[] is copied to the output array then the next iteration has to read a new element from the array a[], and the other way around for the b[] array.

The desire to implement algorithms such as this one on our architecture led us to include two independent read-address generators rather than just one in our architecture, as described in chapter 2. Using these two address generators and the corresponding read queues, the elements of array a[] can be fetched into one of the queues while the elements of array b[] are fetched into the other data queue. The implementation of the merging process is then straightforward, as figure 3.13 shows.

u = read from data queue 0;
from2nd = true;
while not all elements considered
  begin
    if from2nd then x = read from data queue 1;
    else x = read from data queue 0;
    if u > x then write x to the store queue;
    else begin
      write u to the store queue;
      from2nd = not from2nd;
      u = x;
    end;
  end;

Figure 3.13: Implementing the merging process on our architecture

A local variable u is used to hold the last element that a previous iteration has read, but not yet stored, while the local variable from2nd indicates from which of the two data queues the next read should be done. Multiple recurrences exist between successive iterations. The values of both u and from2nd are updated in each iteration. So, it is expected that this implementation can not benefit from software pipelining techniques, since a new iteration cannot be initiated
before preceding ones complete their execution.

It was not possible to find an alternative implementation that eliminates these dependencies. An alternative technique is to try to interleave the execution of multiple merging processes. As the data sets they operate on are distinct, it is possible to initiate independent iterations before preceding ones complete. In all but the last step of the mergesort algorithm, there are multiple pairs available to be merged. An alternative method can be used during the last stage, as only one pair of subarrays are available to be merged: the process of merging is modified to alternatingly operate on the two top and on the two bottom elements of the two arrays. This method can be better explained through the use of the following example: consider the merging of the two sorted arrays (1, 4, 5, 6) and (2, 3, 7, 8). Table 3.3 illustrates how this method operates.

<table>
<thead>
<tr>
<th>compare</th>
<th>operation</th>
<th>first subarray</th>
<th>second subarray</th>
<th>output sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt; 2</td>
<td>write smaller</td>
<td>(4, 5, 6)</td>
<td>(2, 3, 7, 8)</td>
<td>(1, x, x, x, x, x, x)</td>
</tr>
<tr>
<td>6 &lt; 8</td>
<td>write larger</td>
<td>(4, 5, 6)</td>
<td>(2, 3, 7)</td>
<td>(1, x, x, x, x, x, 8)</td>
</tr>
<tr>
<td>4 &gt; 2</td>
<td>write smaller</td>
<td>(4, 5, 6)</td>
<td>(3, 7)</td>
<td>(1, 2, x, x, x, x, 8)</td>
</tr>
<tr>
<td>6 &lt; 7</td>
<td>write larger</td>
<td>(4, 5, 6)</td>
<td>(3)</td>
<td>(1, 2, x, x, x, 7, 8)</td>
</tr>
<tr>
<td>4 &gt; 3</td>
<td>write smaller</td>
<td>(4, 5, 6)</td>
<td>()</td>
<td>(1, 2, 3, x, x, 7, 8)</td>
</tr>
<tr>
<td>6</td>
<td>copy</td>
<td>(4, 5)</td>
<td>()</td>
<td>(1, 2, 3, x, 6, 7, 8)</td>
</tr>
<tr>
<td>4</td>
<td>copy</td>
<td>(5)</td>
<td>()</td>
<td>(1, 2, 3, x, 5, 6, 7, 8)</td>
</tr>
<tr>
<td>5</td>
<td>copy</td>
<td>()</td>
<td>()</td>
<td>(1, 2, 3, 4, 5, 6, 7, 8)</td>
</tr>
</tbody>
</table>

Table 3.3: Interleaving the execution of one merging process

Unfortunately it is not possible to implement those methods if only two data read queues are available. The sequence of data fetching produced by the interleaved execution of multiple merging processes is not predictable, since it depends on the actual data processed, and thus it cannot be generated by the address generators. Notice that the sequence of data stores produced is predictable, as each iteration always stores to an alternating end of the output array. If there were more data queues available, then it would be possible to use this execution scheme and benefit from both access decoupling and software pipelining.

We conclude that merge-sort is applicable to our architecture if comparisons are fast and thus no software pipelining is needed. In the opposite case, only the first stages of merge-sort can be efficiently implemented on our architecture.

3.1.6 Heapsort

Heapsort operates on tree data structures. The core of this sorting method is the shift process, in which a tree node is placed in its appropriate position within a sorted binary tree. This process is highly data dependent and the interested reader is referred to any elementary data structures text (e.g. [4] or [5]) for its details. It is almost impossible to efficiently execute this algorithm on access decoupled architectures. However, by taking this opportunity, we can make some useful comments about manipulating tree structures on access decoupled architectures.

It is likely that the most convenient implementation of binary tree structures on access decoupled architectures is that in which the tree is implemented through the use of an array. The root element of the tree is placed in the first position of the array and for a tree node placed in the $i$th position of the array the left and right child nodes are placed in the $(2 \times i)$th and the $(2 \times i + 1)$th positions of the array, respectively (like heap sort does). The advantage of this
method is that by knowing the position of a node it is possible to compute the positions or, in other words, the addresses of the child nodes, which is not possible when the pointer based representation of tree structures is used. In the latter case the address generators would need to read the node contents before they can generate the addresses of the child nodes.

Bottom-up traversals are easily implementable on access decoupled architectures, since there is only one possible path of movement. On the contrary, top-down traversals are not likely to be implementable on decoupled architectures since they involve making a “routing decision” on every step during the traversal.

3.1.7 Quicksort

Quicksort is one of the most efficient sorting methods available. The pascal-like code (using the last partition element as pivot) is illustrated in figure 3.14. In this implementation, the variable \( u \) holds the value of the pivot element and \( i \) and \( j \) are the left and the right scan pointers, respectively [4]. The three assignment statements following the outer repeat-until loop put the pivot element at its appropriate position.

```
procedure quicksort(l,r:integer);
  var u,i,j,t:integer;
  begin
    if r>l then
      begin
        u:=a[r]; i:=l-1; j:=r;
        repeat
          i:=i+1; until a[i]>=u;
          j:=j-1; until a[j]<=u;
          exchange(a[i],a[j]);
        until j<i;
        a[j]:=a[i]; a[i]:=a[r]; a[r]:=t;
        quicksort(i,i-1);
        quicksort(i+1,r);
      end;
  end;
```

Figure 3.14: Quicksort

As presented, the code has that following unpleasant characteristics:

- nested repeat-until loops.
- multiple element exchanges are done at data dependent positions.
- it is recursive.

A closer look to the code of figure 3.14 shows that the idea behind quicksort is this: select an element as pivot and then scan the input array and place the elements that are smaller (or equal) than the pivot “to the left” and the elements that are larger than the pivot “to the right”. In
this way, the initial array is partitioned into two smaller, arrays each consisting of elements that are smaller (left subarray) or larger (right subarray) than a certain pivot value. This algorithm is called partition. If this method is recursively applied until all subarrays consist of only one element each, then the input array is sorted. As the core of quicksort consists of the partition process we will first consider implementing this algorithm.

As presented in the figure 3.14, both the data fetch and store sequences produced by the execution of the partition process are value-depended. The input array is scanned from the left to right to locate the first element which is larger than the pivot; then it is scanned from right to left until the first elements smaller than the pivot is located. These two elements are exchanged, and the above process is repeated until all elements have been considered. The partition process can be modified in order to eliminate these dependencies. The modified code is shown in figure 3.15.

```
int in[], out[], ini, outi_left, outi_right;
ini=0; outi_left=0; outi_right=last element of out[];
pivot=in[0];
while ini < size of in[]
    if in[ini]<=pivot then out[outi_left++]=in[ini++];
    else out[outi_right]=in[ini++];
```

**Figure 3.15: Modified partition algorithm**

The input array `in[]` is scanned from the first element to the last. If an element if smaller that the pivot, it is copied to the next available position of array `out[]`, from the left to right. The `outi_left` variable keeps that index. The elements that are larger are copied to the next position of array `out[]`, from right to left; the `outi_right` index points to that location.

The data fetch sequence produced by this code is now independent of the actual data fetched: all the elements of the input array are considered once and in order. Unfortunately, the store sequence is still data dependent. The code of figure 3.16 eliminates this problem.

```
int in[], out[], ini, outi, pivot;
ini=0; outi=0; pivot=in[0];
for ini=0 to size of in[]
    if in[ini]<=pivot then out[outi++]=in[ini];
for ini=0 to size of in[]
    if in[ini]>pivot then out[outi++]=in[ini];
```

**Figure 3.16: A modified partition algorithm which produces data independent fetch and store sequences**

Now, the input array is scanned twice. During the first pass, the elements that are less than or equal to the pivot are copied to the output array; during the second pass, those that are larger
are copied into the rest of the output array. The data fetch sequence is again independent of the data. Now, the store sequence is also value-independent, since now the elements of the output array are filled in order from the first one to the last one.

Now that we have an appropriate implementation of the partition process, we can use it to implement the quicksort algorithm. As it was mentioned, quicksort is defined as a recursive algorithm. This makes it inappropriate for access decoupled architectures, since during the recursive calls, long execution delays will occur due to information exchange between the processors.

Fortunately, it is possible to implement quicksort without recursive calls. The methods used can be explained best through an example. Consider that initially array n[1] contains (59, 18, 23, 90, 12, 75, 89). Two copies of the input array are fetched, one to each of the two data read queues. At the end of these copies a special mark is placed to indicate that the arrays end there. As table 3.4 illustrates, during the first pass, the elements of the first data queue are read. The first element is selected as pivot and compared with all the others by the processing unit. Those that are smaller are copied to the output queue. As soon as the end mark is encountered, it is copied to the output queue to indicate that the first of the two partitions has been created.

During the second pass, the elements from the second data queue are read. Again the first, namely 59, is used as pivot and compared with all the other elements. But now, those elements that are larger than the pivot are copied to the output queue.

Now consider the execution example of table 3.5. In this, the input sequence consists of two partitions, assumed to have been created by a previous application of the algorithm. During the first step, the first copy of the first partition is read from one of the data queues. The elements that are smaller or equal than the pivot, namely 13, 49 and 50, are copied to the output queue. The pivot, for simplicity, is selected to be the first element of the partition. As soon as the end-of-partition mark is reached, it is copied to the output queue and the algorithm proceeds with the next step.

During the second step, the second copy of the first partition is read from the other data queue and now the elements that are larger than the pivot (again 50) are copied to the output queue. The second new partition is created in this way and it contains 51 and 59. Again, when the end-of-partition mark is found, it is copied to the output queue.

During the third stage, the first copy of the second partition is read from the upper queue and the elements that are smaller or equal than the pivot are copied to the output queue. These are 75, 80 and 84, as the pivot is again selected to be the first element of the partition, namely 90. As soon as the end-of-partition mark is found, it is copied to the output queue and the algorithm proceeds with the fourth and last stage, in which the second copy of the second partition is read from the lower queue, and during which the elements that are larger than the pivot are copied to the output queue.

In this example the two input partitions were split to four smaller partitions. Notice that the partitions generated can be again fed to the algorithm in order to repartition them. This method can be used to implement quicksort without using recursion. The execution flow when using this algorithm is illustrated in figure 3.17.

The recursive implementation of quicksort has an treelike execution flow. At each step, one input partition is split to two smaller. The left most of the two is then considered and partitioned, all the way down, followed only the partitioning of the right most. The execution flow of the recursive implementation is a parent-left child-right child traversal of the tree. On the contrary, the nonrecursive implementation does multiple partitions at each step. In fact, in


<table>
<thead>
<tr>
<th>operation</th>
<th>queue 0</th>
<th>queue 1 output queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>pivot=59, copy</td>
<td>(18, 23, 90, 12, 75, 89, m)</td>
<td>(59, 18, 23, 90, 12, 75, 89, m)</td>
</tr>
<tr>
<td>18 &lt; 59, copy</td>
<td>(23, 90, 12, 75, 89, m)</td>
<td>(59, 18, 23, 90, 12, 75, 89, m)</td>
</tr>
<tr>
<td>23 &lt; 59, copy</td>
<td>(90, 12, 75, 89, m)</td>
<td>(59, 18, 23, 90, 12, 75, 89, m)</td>
</tr>
<tr>
<td>90 &gt; 59, discard</td>
<td>(12, 75, 89, m)</td>
<td>(59, 18, 23, 90, 12, 75, 89, m)</td>
</tr>
<tr>
<td>12 &lt; 59, copy</td>
<td>(75, 89, m)</td>
<td>(59, 18, 23, 90, 12, 75, 89, m)</td>
</tr>
<tr>
<td>75 &gt; 59, discard</td>
<td>(89, m)</td>
<td>(59, 18, 23, 90, 12, 75, 89, m)</td>
</tr>
<tr>
<td>89 &gt; 59, discard</td>
<td>(m)</td>
<td>(59, 18, 23, 90, 12, 75, 89, m)</td>
</tr>
<tr>
<td>place end mark</td>
<td>()</td>
<td>(59, 18, 23, 90, 12, 75, 89, m)</td>
</tr>
<tr>
<td>pivot=59</td>
<td>()</td>
<td>(18, 23, 90, 12, 75, 89, m)</td>
</tr>
<tr>
<td>discard 59</td>
<td>()</td>
<td>(18, 23, 90, 12, 75, 89, m)</td>
</tr>
<tr>
<td>18 &lt; 59, discard</td>
<td>()</td>
<td>(23, 90, 12, 75, 89, m)</td>
</tr>
<tr>
<td>23 &lt; 59, discard</td>
<td>()</td>
<td>(90, 12, 75, 89, m)</td>
</tr>
<tr>
<td>90 &gt; 59, copy</td>
<td>()</td>
<td>(12, 75, 89, m)</td>
</tr>
<tr>
<td>12 &lt; 59, copy</td>
<td>()</td>
<td>(75, 89, m)</td>
</tr>
<tr>
<td>75 &gt; 59, copy</td>
<td>()</td>
<td>(89, m)</td>
</tr>
<tr>
<td>89 &gt; 59, copy</td>
<td>()</td>
<td>(m)</td>
</tr>
<tr>
<td>place end mark</td>
<td>()</td>
<td>()</td>
</tr>
</tbody>
</table>

Table 3.4: An execution example of the modified partition process
<table>
<thead>
<tr>
<th>Operation</th>
<th>Queue 0</th>
<th>Queue 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Output Queue</td>
<td></td>
</tr>
<tr>
<td>pivot = 50, copy</td>
<td>$\langle 50, 13, 51, 49, 59, m, 84, 90, 75, 80, m \rangle$</td>
<td>$\langle 50, 13, 51, 49, 59, m, 84, 90, 75, 80, m \rangle$</td>
</tr>
<tr>
<td></td>
<td>($\lambda$, $\lambda$)</td>
<td>($\lambda$, $\lambda$)</td>
</tr>
<tr>
<td>13 &lt; 50, copy</td>
<td>$\langle 51, 49, 59, m, 84, 90, 75, 80, m \rangle$</td>
<td>$\langle 50, 13, 51, 49, 59, m, 84, 90, 75, 80, m \rangle$</td>
</tr>
<tr>
<td></td>
<td>(50)</td>
<td>(50)</td>
</tr>
<tr>
<td>51 &gt; 50, discard</td>
<td>$\langle 51, 49, 59, m, 84, 90, 75, 80, m \rangle$</td>
<td>$\langle 50, 13, 51, 49, 59, m, 84, 90, 75, 80, m \rangle$</td>
</tr>
<tr>
<td></td>
<td>(50, 13)</td>
<td>(50, 13)</td>
</tr>
<tr>
<td>49 &lt; 50, copy</td>
<td>$\langle 59, m, 84, 90, 75, 80, m \rangle$</td>
<td>$\langle 50, 13, 51, 49, 59, m, 84, 90, 75, 80, m \rangle$</td>
</tr>
<tr>
<td></td>
<td>(50, 13, 49)</td>
<td>(50, 13, 49)</td>
</tr>
<tr>
<td>59 &gt; 50, discard</td>
<td>$\langle 84, 90, 75, 80, m \rangle$</td>
<td>$\langle 50, 13, 49, m \rangle$</td>
</tr>
<tr>
<td>place end mark</td>
<td>(50, 13, 49, m)</td>
<td>(50, 13, 49, m)</td>
</tr>
<tr>
<td>pivot = 50, discard</td>
<td>$\langle 84, 90, 75, 80, m \rangle$</td>
<td>$\langle 13, 51, 49, 59, m, 84, 90, 75, 80, m \rangle$</td>
</tr>
<tr>
<td></td>
<td>(50, 13, 49, m)</td>
<td>(50, 13, 49, m)</td>
</tr>
<tr>
<td>13 &lt; 50, discard</td>
<td>$\langle 84, 90, 75, 80, m \rangle$</td>
<td>$\langle 51, 49, 59, m, 84, 90, 75, 80, m \rangle$</td>
</tr>
<tr>
<td></td>
<td>(50, 13, 49, m)</td>
<td>(50, 13, 49, m)</td>
</tr>
<tr>
<td>51 &gt; 50, copy</td>
<td>$\langle 84, 90, 75, 80, m \rangle$</td>
<td>$\langle 50, 13, 49, m, 84, 90, 75, 80, m \rangle$</td>
</tr>
<tr>
<td></td>
<td>(50, 13, 49, m, 51)</td>
<td>(50, 13, 49, m, 51)</td>
</tr>
<tr>
<td>49 &lt; 50, discard</td>
<td>$\langle 84, 90, 75, 80, m \rangle$</td>
<td>$\langle 59, m, 84, 90, 75, 80, m \rangle$</td>
</tr>
<tr>
<td>place end mark</td>
<td>(50, 13, 49, m, 51)</td>
<td>(50, 13, 49, m, 51)</td>
</tr>
<tr>
<td>59 &gt; 50, copy</td>
<td>$\langle 84, 90, 75, 80, m \rangle$</td>
<td>($\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$)</td>
</tr>
<tr>
<td>place end mark</td>
<td>(50, 13, 49, m, 51, 59, m)</td>
<td>(50, 13, 49, m, 51, 59, m)</td>
</tr>
<tr>
<td>pivot = 84, copy</td>
<td>$\langle 90, 75, 80, m \rangle$</td>
<td>$\langle 84, 90, 75, 80, m \rangle$</td>
</tr>
<tr>
<td></td>
<td>(50, 13, 49, m, 51, 59, m, 84)</td>
<td>(50, 13, 49, m, 51, 59, m, 84)</td>
</tr>
<tr>
<td>89 &gt; 84, discard</td>
<td>$\langle 75, 80, m \rangle$</td>
<td>$\langle 84, 90, 75, 80, m \rangle$</td>
</tr>
<tr>
<td></td>
<td>(50, 13, 49, m, 51, 59, m, 84)</td>
<td>(50, 13, 49, m, 51, 59, m, 84)</td>
</tr>
<tr>
<td>75 &lt; 84, copy</td>
<td>$\langle 80, m \rangle$</td>
<td>$\langle 84, 90, 75, 80, m \rangle$</td>
</tr>
<tr>
<td></td>
<td>(50, 13, 49, m, 51, 59, m, 84, 75)</td>
<td>(50, 13, 49, m, 51, 59, m, 84, 75)</td>
</tr>
<tr>
<td>80 &lt; 84, copy</td>
<td>$\langle \lambda \rangle$</td>
<td>$\langle 84, 90, 75, 80, m \rangle$</td>
</tr>
<tr>
<td>place end mark</td>
<td>($\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$)</td>
<td>(50, 13, 49, m, 51, 59, m, 84, 75, 80, m)</td>
</tr>
<tr>
<td>pivot = 84</td>
<td>$\langle \lambda \rangle$</td>
<td>($\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$)</td>
</tr>
<tr>
<td></td>
<td>(50, 13, 49, m, 51, 59, m, 84, 75, 80, m)</td>
<td>(50, 13, 49, m, 51, 59, m, 84, 75, 80, m)</td>
</tr>
<tr>
<td>90 &gt; 84, copy</td>
<td>$\langle \lambda \rangle$</td>
<td>($\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$)</td>
</tr>
<tr>
<td></td>
<td>(50, 13, 49, m, 51, 59, m, 84, 75, 80, m, 90)</td>
<td>(50, 13, 49, m, 51, 59, m, 84, 75, 80, m, 90)</td>
</tr>
<tr>
<td>75 &lt; 84, discard</td>
<td>$\langle \lambda \rangle$</td>
<td>($\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$)</td>
</tr>
<tr>
<td></td>
<td>(50, 13, 49, m, 51, 59, m, 84, 75, 80, m, 90)</td>
<td>(50, 13, 49, m, 51, 59, m, 84, 75, 80, m, 90)</td>
</tr>
<tr>
<td>80 &lt; 84, discard</td>
<td>$\langle \lambda \rangle$</td>
<td>($\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$, $\lambda$)</td>
</tr>
<tr>
<td>place end mark</td>
<td>(50, 13, 49, m, 51, 59, m, 84, 75, 80, m, 90)</td>
<td>(50, 13, 49, m, 51, 59, m, 84, 75, 80, m, 90)</td>
</tr>
</tbody>
</table>

Table 3.5: Execution example of the modified quicksort algorithm
its ith step, the nonrecursive implementation performs all the partitions which are located at depth i of the execution tree of the recursive implementation.

As far as software pipelining is concerned, successive iterations of the modified method are not independent. A new iteration has to wait for the previous one to decide from which queue it will read. Also it has to wait for pivot value updating done by previous iterations. When the details of the implementation will be discussed it will be shown that it is possible to do these two jobs in the early part of an iteration which will permit a partial interleaving of successive iterations.

Before discussing the implementation details, it is worth noticing that selecting the first element of a partition as the pivot degrades the performance of the partition process. Consider for example the partitioning of the sequence (4, 9, 3, 1, 2). In the first step all elements are compared with 4 and the left partition created will be (4, 3, 1, 2). If the first element is again selected as pivot it is clear that one of the two partitions that will be created will contain only one element, the pivot. So, if 4 is again selected as pivot the the two partitions produced are (4) and (3, 1, 2).

There are two ways to overcome this problem. The first is to place the pivot value at the end of one of the two partitions. Using this method, the initial partition of (4, 9, 3, 1, 2) is partitioned to (9) and (3, 1, 2, 4). Now, in the next step the first element is no more the same, so hopefully the performance of the partition process will be good enough. The disadvantage of this method is that additional bookkeeping is needed, requiring additional instructions to implement it, so the execution time will be lengthened.

The second method to preserve the performance of the partition process is to change the direction of scanning between successive steps. During the first step, the input array is scanned starting left to right, while in the second step the direction is changed from right to left, and so on. Using the previous example of partitioning (4, 9, 3, 1, 2), initially 4 is selected as pivot and
the partitions produced are (9) and (4, 3, 1, 2). In the next step, the partitions are scanned right to the left, so the first element encountered is not the same as in the previous scan — it is 2 and not 4. The disadvantage of this method is that execution delays will occur between successive steps. The last elements written by the previous iteration are exactly those that must be fetched first by the next iteration, as the direction of scanning changes. This means that their fetching will be delayed by the queue controllers until the results from the output queue are actually written to memory. However, if the input arrays are adequately large, it is expected that the overall execution time will be shorter than using the previous method. For large input sizes, the execution time lost between successive steps will be small compared to the execution time spend during the partitions.

Implementation details of quicksort

In this paragraph, the details of the implementation of quicksort on our architecture are discussed. Two arrays are used by each step of the algorithm and updated copies of them are produced.

The first is the array of elements which have to be sorted and it will be denoted as e[]. The second array will be denoted m[] and referred to the array of marks. The value of the ith element of the array of marks indicates whether the corresponding element of the e[] array is the last element of a partition. For example, if e[] contains (1, 12, 5, 20, 40, 90) and m[] contains (0, 0, 0, 1, 0, 1), then there are two partitions: (1, 12, 5, 20) and (40, 90). The last element of the array of marks must always be 1 since it is always an end-of-partition.

At each step of the algorithm, the two data read queues are both filled with interleaved copies of e[] and m[]. Thus, both of the two data read queues will be filled with the sequence (e0, m0, e1, m1, ..., eN, mN). For our previous example were the e[] array was (1, 12, 5, 20, 40, 90) and the array of marks was (0, 0, 0, 1, 0, 1) each of the two data read queues will be filled with (1, 0, 12, 0, 5, 0, 20, 1, 40, 0, 90, 1).

In each step, these two copies of e[] and m[] are partitioned in twice as many parts as they contained before. The first copy is used to find the elements that are less than or equal to than the pivot of their partition, and the other copy is used to find the elements that are larger than the pivot. This partition process produces updated copies of the two input arrays. The address generator responsible for data stores has to produce the addresses of the elements of these two arrays in a similarly interleaved way. Each step must not overwrite the e[] and m[] arrays, but create two new updated copies of these arrays. If the same copies were concurrently fetched and overwritten then a deadlock would occur due to the data queue mechanism; the store queue would contain addresses same to those found in the read queues and the queue controllers, assuming that the stores should precede the loads, would block data fetching until the store queue was filled. Even if there were a way to disable this interlock mechanism, it would be possible to overwrite an element whose second copy is not yet fetched.

So, each step of the algorithm produces two new arrays, the eNew[], which contains the new partitions, and the mNew[], which indicates their boundaries. These two arrays are used as input for the next step, and so on. Two copies of the e[] and the m[] array can be used in order to ensure that for each iteration, the input arrays are stored at different locations from those used by the output arrays. Note that similar methods must be used with the previously discussed implementations of other sorting algorithms.

The code executed by the processing unit is shown in figure 3.18. The outer repeat–until loop is executed until all partitions consist each of only one element then the elements are
sorted. This condition is tested through the use of a local variable, \texttt{AllPartitionsHaveSizeOne}, which is updated by the iterations of the inner do loop. This variable is initially set true and each time an iteration writes an element to a non-empty partition it is reset. Before processing the input elements, the other local variables are initialized. The \texttt{SearchingForGreater} is used to indicate whether an iteration searches for elements that are greater than the pivot or not. It also indicates from which of the two data queues we have to read data from. This variable is initially set false to indicate that the first pass over a partition will locate those elements that are less than or equal to the pivot. The iteration count is set to be the double of the number of the input elements as two copies of them have to be processed.

1: repeat  
2: \texttt{n=}2 \times \texttt{ElementsToSort}  
3: \texttt{SearchingForGreater=} \texttt{true}  
4: \texttt{AllPartitionsHaveSizeOne=} \texttt{true}  
5: \texttt{do} \texttt{n} \texttt{times}  
6: \texttt{begin}  
7: \quad \texttt{if} \texttt{SearchingForGreater}  
8: \quad \texttt{then} \texttt{EndOfPartition=} \texttt{Read From data queue 1}  
9: \quad \texttt{else} \texttt{EndOfPartition=} \texttt{Read From data queue 0}  
10: \quad \texttt{if} \texttt{EndOfPartition}  
11: \quad \texttt{then} \texttt{SearchingForGreater=} \texttt{not SearchingForGreater}  
12: \quad \texttt{if} \texttt{SearchingForGreater}  
13: \quad \texttt{then} \texttt{Element=} \texttt{Read From data queue 1}  
14: \quad \texttt{else} \texttt{Element=} \texttt{Read From data queue 0}  
15: \quad \texttt{if} \texttt{EndOfPartition} \texttt{then Pivot=} \texttt{Element}  
16: \quad \texttt{if} \texttt{EndOfPartition} \texttt{then FirstOfNewPartition=} \texttt{true}  
17: \quad \texttt{DoWrite=} \texttt{ Element} \leftarrow \texttt{Pivot}  
18: \quad \texttt{if} \texttt{SearchingForGreater} \texttt{then DoWrite=} \texttt{not DoWrite}  
19: \quad \texttt{if} \texttt{DoWrite} \texttt{then}  
20: \quad \texttt{begin}  
21: \quad \texttt{write Element to Store queue}  
22: \quad \texttt{write FirstOfPartition to Store queue}  
23: \quad \texttt{if} \texttt{not FirstOfPartition} \texttt{then AllPartitionsHaveSizeOne=} \texttt{false}  
24: \quad \texttt{FirstOfPartition=} \texttt{false}  
25: \quad \texttt{end}  
26: \quad \texttt{end}  
27: \quad \texttt{until} \texttt{AllPartitionsHaveSizeOne}  

\textbf{Figure 3.18:} Processing unit program for quicksort

Each iteration of the inner do loop initially reads the boundary mark into \texttt{EndOfPartitionMark} local (lines 7-9). This boundary mark refers to the previously read element and comes either from the first data read queue or from the second, depending on the value of \texttt{SearchingForGreater}. If the mark indicates that the end of partition was reached, then \texttt{SearchingForGreater} is toggled (lines 10-11) to indicate that the other copy of the partition should be used or that the processing of a new partition starts.
Then, the next element of the e[] is read from one of the data queues (lines 12-14) and if the previously read mark indicates that the end of a partition was reached this element is used as the new pivot (line 15) as it is the first element of its partition. In such a case FirstOfNewPartition is also set true. This variable is used to indicate whether a new partition has only one element or not. It is true until the first element of a new partition is written. It will be shown later that this variable helps in the creation of the new array of marks mNew[].

The element of the e[] array is then compared to the pivot (line 17) and the result of the comparison is stored in DoWrite. A true value of this local indicates that the element compared should be placed in the new partition. If this iteration searched for elements that are greater than the pivot then the DoWrite value is inverted (line 18).

Lines 19-25 write out the new partition element and its mark. In line 21, the element is copied to the new partition. Then, its mark is also written (line 22). Notice that the mark value written is not the one required by the definition of the mark array. This code marks the element that is the first of its partition and not the last one. Fortunately, in order to produce the correct marks, an additional step is not required, since a partition starts where the previous one ends. A slight modification of the programs of the address generators can fix the problem as it will be shown later. Then the AllPartitionsHaveSizeOne local is updated. If the element written is not the first of this new partition then at least one partition with two or more elements exists. Finally the FirstOfNewPartition is negated to indicate that the first element of a new partition has been written.

The above method can be best explained through the use of an example. Consider the array (59, 75, 80, 13, 84, 90, 51, 49, 50) to be sorted. The mark array is initially set to (0, 0, 0, 0, 0, 0, 0, 0, 1). The execution flow is illustrated in figure 3.6.

Notice that one of the two read queues contains an additional element at the top. This is a mark value indicating an additional partition end. The algorithm, as expressed, requires such a dummy end of partition mark that before the first partition is read.

The first iteration will read this mark and decide that it must use the second data queue and read from it the first element of the array, which is 59. This element is set to be the pivot value as it is the first element of the partition. Initially, this element is compared with itself, it is found to be equal to the pivot, and so it is stored to the output queue. The mark stored with it is '1', since it is the first element of a new partition currently under creation. The next iteration will read a mark value of '0' and continue with read the next element from the same data queue. This is 75 and currently SearchingForGreater is false, so this element is discarded. The same happens with the next iteration as its element is 80 which is larger than the pivot (50). The next iteration reads a '0' mark, and then 13, which is smaller than the pivot, and so it copies it to the output queue. Its mark is set to '0' as this is not the first element written to this partition while AllPartitionsHaveSizeOne is negated to indicate that there is a new partition which consists of at least two elements. The same process is repeated, and 51, 49 and 50 are copied to the output queue, all marked with '0'. After 50 is read from the current read queue, the next mark found is '1' which indicates that the first copy of the partition ended. Because of that SearchingForGreater is toggled, and the next element is read from the other read queue. Again, this is set to be the pivot, since it is the first element of its partition. Notice that, since the second copy of the same partition is read, this element is again 59. The elements of the second queue are read, but now those that are larger than the pivot are copied to the output queue. These are 75, 80, 84, 93 and 90. All are marked with '0', except the first one, whose
### 3.1. Sorting algorithms

<table>
<thead>
<tr>
<th>operation</th>
<th>queue 0</th>
<th>queue 1</th>
<th>output queue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1, 59, 0, 75, 0, 80, 0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (59, 0, 75, 0, 80, 0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) () (0, 75, 0, 80, 0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (59, 1)</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>pivot=59, copy</td>
<td>(59, 0, 75, 0, 80, 0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (0, 80, 0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (59, 1)</td>
<td>(59, 1)</td>
<td>()</td>
</tr>
<tr>
<td>75 &gt; 59, discard</td>
<td>(59, 0, 75, 0, 80, 0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (59, 1)</td>
<td>(59, 1)</td>
<td>()</td>
</tr>
<tr>
<td>80 &gt; 59, discard</td>
<td>(59, 0, 75, 0, 80, 0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (59, 1)</td>
<td>(59, 1)</td>
<td>()</td>
</tr>
<tr>
<td>13 &lt; 59, copy</td>
<td>(59, 0, 75, 0, 80, 0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (59, 1, 13, 0)</td>
<td>(59, 1, 13, 0)</td>
<td>()</td>
</tr>
<tr>
<td>50 &lt; 59, copy</td>
<td>(59, 0, 75, 0, 80, 0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (1) (59, 1, 13, 0, 51, 0, 49, 0, 50, 0)</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>pivot=59, discard</td>
<td>(0, 75, 0, 80, 0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (59, 1, 13, 0, 51, 0, 49, 0, 50, 0)</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>75 &gt; 59, copy</td>
<td>(0, 80, 0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (59, 1, 13, 0, 51, 0, 49, 0, 50, 0, 75, 1)</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>80 &gt; 59, copy</td>
<td>(0, 13, 0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (59, 1, 13, 0, 51, 0, 49, 0, 50, 0, 75, 1, 80, 0)</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>13 &lt; 59, discard</td>
<td>(0, 84, 0, 90, 0, 51, 0, 49, 0, 50, 1) (59, 1, 13, 0, 51, 0, 49, 0, 50, 0, 75, 1, 80, 0)</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>50 &lt; 59, discard</td>
<td>(1) (59, 1, 13, 0, 51, 0, 49, 0, 50, 0, 75, 1, 80, 0, 84, 0, 90, 0)</td>
<td>()</td>
<td>()</td>
</tr>
</tbody>
</table>

Table 3.6: An execution example of the quicksort method, 1st pass
mark is set to '1'.

After the inner do loop terminate, the output queue contains the following sequence: (59, 1, 13, 0, 51, 0, 49, 0, 50, 0, 75, 1, 80, 0, 84, 0, 90, 0). This means that the array of elements is now (59, 13, 51, 49, 50, 75, 80, 90), and the array of marks is (1, 0, 0, 0, 0, 1, 0, 0, 0). Instead of what was produced, the array of marks should be (0, 0, 0, 0, 1, 0, 0, 0, 1). It is very easy to correct this problem by shifting the elements of the array of marks by one position to the left in a cyclic way, or, even better, as a slight change in the store address sequence is enough to produce the same results. So, for the first step, the address generator produces the sequence \((e_0, m_0, e_1, m_1, ..., e_N, m_N)\), while for the second step the store sequence becomes \((e_0, m_1, e_1, m_2, ..., e_{N-1}, m_N, e_N, m_0)\), and for the i\text{th} step it is \((e_0, m_{i-1}, e_1, m_i, ..., e_N, m_{i-2})\).

Figure 3.7 illustrates the execution flow during the second step of the previously discussed example.

<table>
<thead>
<tr>
<th>operation</th>
<th>queue 0</th>
<th>queue 1</th>
<th>output queue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1,90,0,84,0,80,0,75,1,50,0,49,0,51,0,13,0,59,1)</td>
<td>(90,0,84,0,80,0,75,1,50,0,49,0,51,0,13,0,59,1)</td>
<td>()</td>
</tr>
<tr>
<td>pivot=90,copy</td>
<td>(90,0,84,0,80,0,75,1,50,0,49,0,51,0,13,0,59,1)</td>
<td>(0,84,0,80,0,75,1,50,0,49,0,51,0,13,0,59,1)</td>
<td>(90,1)</td>
</tr>
<tr>
<td>75 &lt; 90,copy</td>
<td>(90,0,84,0,80,0,75,1,50,0,49,0,51,0,13,0,59,1)</td>
<td>(1,50,0,49,0,51,0,13,0,59,1)</td>
<td>(90,1,84,0,80,0,75,0)</td>
</tr>
<tr>
<td>pivot=90,discard</td>
<td>(0,84,0,80,0,75,1,50,0,49,0,51,0,13,0,59,1)</td>
<td>(50,0,49,0,51,0,13,0,59,1)</td>
<td>(90,1,84,0,80,0,75,0)</td>
</tr>
<tr>
<td>75 &lt; 90,discard</td>
<td>(50,0,49,0,51,0,13,0,59,1)</td>
<td>(1,50,0,49,0,51,0,13,0,59,1)</td>
<td>(90,1,84,0,80,0,75,0)</td>
</tr>
<tr>
<td>pivot=50,copy</td>
<td>(49,0,51,0,13,0,59,1)</td>
<td>(50,0,49,0,51,0,13,0,59,1)</td>
<td>(90,1,84,0,80,0,75,0,50,1)</td>
</tr>
<tr>
<td>59 &gt; 50,discard</td>
<td>(1)</td>
<td>(50,0,49,0,51,0,13,0,59,1)</td>
<td>(90,1,84,0,80,0,75,0,50,1,49,0,13,0)</td>
</tr>
<tr>
<td>pivot=50,discard</td>
<td>()</td>
<td>(0,49,0,51,0,13,0,59,1)</td>
<td>(90,1,84,0,80,0,75,0,50,1,49,0,13,0)</td>
</tr>
<tr>
<td>59 &gt; 50,copy</td>
<td>()</td>
<td>(1)</td>
<td>(90,1,84,0,80,0,75,0,50,1,49,0,13,0,51,1,59,0)</td>
</tr>
</tbody>
</table>

Table 3.7: An execution example of the quicksort method, 2nd pass
3.2 Pattern matching

As can be seen in appendix D only 12 instruction are required to implement the inner do loop.

A disadvantage of the method presented so far is that even if only one of the partitions consists of more than two elements all the input elements will be passed over. It is expected that for input sequences of randomly arranged elements such situations will be rare. However, for partially sorted inputs, may often arise, degrading the performance of the method presented.

3.2 Pattern matching

In this section, pattern matching algorithms are considered. It is shown that most of them are inappropriate for implementation them on architectures such as ours; however one of them, the KMP pattern matching algorithm, is appropriate. The implementation method followed can be of general use, as it illustrates a method to implement algorithms using small lookup tables.

3.2.1 Brute-Force algorithm

The obvious method for pattern matching is to check at each possible position in the text where which the pattern could match. The algorithm uses two indexes $s_i$ and $p_i$. $s_i$ points to the current position in the input string and $p_i$ points to the current position in the pattern. The input string is scanned through the use of $s_i$. Each time a match is found, both $s_i$ and $p_i$ are incremented. When a mismatch occurs, $s_i$ is reset to the character immediately after the one the mismatched pattern started, and $p_i$ is zeroed. The process ends when either a match is found or the input is exhausted.

This method is not suitable for access decoupled architectures, since it requires that backup into the input string be possible. If the elements of the input array are fetched into one of the two data read queues backing up can only be implemented in the following way. Discard all the elements from the queue and communicate with the address generator to refetch the required elements. It is clear that this process will produce long execution delays.

One could fetch two copies of the input string one to each of the two data queues available, in order to overcome the problem of backing up: when a backup is required the second copy is used. This method has two major problems that make it inappropriate: first if only two data read queues are available it is not possible to concurrently fetch the pattern, and second even if three data read queues were available, this method would only allow a max of one backup $n$ each string part. If it happens that more than two mismatches occur in the same part of the string then more than two copies are required.

3.2.2 Knuth-Morris-Pratt algorithm

This algorithm takes advantage of the fact that, when a mismatch is detected, the mismatch consists of characters that are known in advance, since they were just read. By using this information, this algorithm works without ever backing up into the input string. None of the input string characters is read twice. The pascal–like code for this algorithm is shown in figure 3.19.

The algorithm uses a lookup table, `next[]`, of size equal to the size of the pattern searched for. The procedure `initnext` generates the elements of this array. The interested reader is referred e.g. to [4]. The $j$th element of this array gives the position in the pattern from which a
function kmpsearch: integer;
  var i,j:integer;

  begin
  i:=1; j:=1; initnext;
  repeat
    if(j=0) or (a[i]=p[j])
      then begin i:=i+1; j:=j+1; end;
    else j:=next[j];
  until (j>M) or (i>N);
  if j>M then kmpsearch:=i-M; else kmpsearch:=i;
  end;

Figure 3.19: The Knuth–Morris–Pratt pattern matching algorithm

A new pattern should be searched for if a mismatch in the jth position is detected. For example, if the pattern is 10100111, then the next array will be (1, 1, 1, 2, 3, 1, 2, 2). If a mismatch is detected in the fourth position then the sequence read was 1011, instead of the expected 1010. The array next indicates that the next search should start from the second element of the pattern. This is true as the last 1 of the mismatched 1011, matches the first character of the pattern. In order to implement this algorithm on decoupled architectures partial flushes of the read queues must be supported by the hardware. The input string is fetched into one of the two read queues. Since no backing up is ever needed, only one copy is enough. The second data read queue will be used to hold multiple copies of the pattern. Each time a mismatch occurs, the remaining part of the pattern is flushed off the data queue through the use of the partial flushing mechanism; then, a new pattern copy becomes available. It remains to find an efficient method to access the next array, whenever required. Clearly if the algorithm of figure 3.19 is used, long execution delays will occur each time the array next is accessed as the address generators have to wait for the processing unit to decide which element it wants, before fetching it. Instead of fetching only the elements of the pattern into the second data read queue we can fetch the elements of the array next as well, in an interleaved way. After fetching the i-th element of the pattern, the address generators also fetch the i-th element of the array next. The algorithm is also modified to read both the pattern element and the next [i] element. If mismatch is found, then the next [i] element can be used to compute the number of elements to flush in order to locate the next copy of the pattern element specified by the array next. If a partial match is found, then the next [i] element is simply discarded. However, instead of computing the number of elements that have to be discarded at execution time, this can be done in advance for each element of the array next. So, we change the definition of this array as follows: the i-th element of nextNew[i] gives the number of elements that must be flushed off the data queue in order to reach the next copy of the next[i] pattern element.

For the previous example, where the pattern was 10100111 and the next array was (1, 1, 1, 2, 3, 1, 2, 2), the nextNew array becomes (16, 14, 12, 12, 10, 6, 6, 4). For example, if a mismatch is found at the second character, then 14 elements have to be discarded from the data queue in order to reach the next copy of the first pattern element. These 14 elements are 7 pattern elements and 7 nextNew array elements.
3.2. Pattern matching

In order to determine when a full match is found, two special values are placed in the data queue between the copies of the pattern/nextNew array; they indicate that all the elements of the pattern were matched. For this reason the elements of the array nextNew shown above must be increased by 2. The nextNew element value of the last pair can be set negative to indicate that a full match was found, since normally all the values of the array nextNew must be positive. The modified algorithm is shown in figure 3.20.

```
nextNew=1;
while nextNew>0 and not all the elements of the input string were considered
begin
  patternC=read from data queue 1
  nextNew=read from data queue 1
  stringC=read from data queue 0
  if patternC!=stringC flush nextNew elements from the data queue 1
end
```

Figure 3.20: Modified KMP algorithm

Successive iteration of this algorithm depend on each other, so this algorithm is not appropriate for the application of software pipelining. An appropriate modification was not found.

An unpleasant characteristic of this method is its requirements on memory bandwidth. In most cases, mismatches will occur and this means that flushes on one of the read queues will be very frequent. For large patterns, the number of elements flushed each time will be large, leading to an increased requirement for memory accesses. This problem can be partially solved by modifying the code in order to avoid flushing if the array next indicates that the current pattern element has to be used again by the next iteration, which is the most frequent case of mismatches occurring at the first position of the pattern.

We can now conclude that it is possible to efficiently implement this algorithm on decoupled architectures but it is not possible to use software pipelining for significantly better performance. As it can be seen in the next chapter the implementation of this algorithm can be simplified if the processing unit has local memory. This also significantly reduces the memory bandwidth requirements the algorithm, since both the pattern and the array next can be stored in that local memory.
4

Sorting when keys are strings

The implementations discussed so far are capable to handle elementary data types. The elements of those types can be easily manipulated as they can be stored within a register of the target machine. In this chapter the problems arising when operating on larger data types are discussed, through the use of sorting algorithms that operate on strings. Initially strings of fixed length are considered, and then the additional problems encountered when using variable length strings are addressed. As it will be shown the inclusion of a small local data store, in the processing unit can dramatically increase the performance and ease the implementation of algorithms operating on nonelementary data types. The additional benefits gained by this architectural extension are significant, because more algorithms can be implemented and because the implementation of others is simplified.

4.1 Using insertion sort with fixed length keys

In order to sort fixed length strings using insertion-sort only slight modifications to the algorithms presented in chapter 3 are required. Consider for example the insertion of the “ox” string to the “ax, xi” sorted sequence. One of the two data read queues can be used to fetch multiple copies of the string under insertion while the other data read queue is used to hold double copies of each string of the sorted sequence. For our example, the first read queue will contain (ox, ox, ox), and the second queue will contain (ax, ax, xi, xi). The method works as follows: a string s from the sorted sequence is read and compared with the string i under insertion, consuming one of the two available copies of s. If s is smaller, its second copy is copied to the output queue. If it is larger, then i is copied to the output, followed by the second copy of s. For our example, the execution flow is as table 4.1 illustrates.

<table>
<thead>
<tr>
<th>operation</th>
<th>queue 0</th>
<th>queue 1</th>
<th>output queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>compare ax,ox</td>
<td>ax,ax,xi,xi</td>
<td>ox,ox,ox</td>
<td>ax</td>
</tr>
<tr>
<td>copy ax</td>
<td>ax,xi,xi</td>
<td>ox,ox</td>
<td>ax</td>
</tr>
<tr>
<td>compare xi,ox</td>
<td>xi,xi</td>
<td>ox,ox</td>
<td>ax,ox</td>
</tr>
<tr>
<td>copy ox</td>
<td>xi</td>
<td>ox</td>
<td>ax,ox</td>
</tr>
<tr>
<td>copy xi</td>
<td>xi</td>
<td></td>
<td>ax,ox,xi</td>
</tr>
</tbody>
</table>

Table 4.1: An insertion sort example
Double copies of each string of the sorted sequence are required, the first copy is compared with the element under insertion and the second copy is copied to the output queue. If only one copy was available then it would be impossible to compare each string and then copy it to the output queue. Notice that if the sorted sequence consists of $N$ strings then $N + 1$ copies of the string under insertion are required: $N$ of them are compared with the $N$ strings of the sorted sequence, and one is used to copy the new string to the output queue when the appropriate position is found. In a more formal way, we can say that the first read queue is filled with $(s_1, s_1, s_2, s_2, \ldots, s_N, s_N)$ and the second queue is filled with $N + 1$ copies of the string under insertion ($s_i$ denotes the $i$th string of the sorted sequence). The algorithm is shown at figure 4.1.

\begin{verbatim}
keyNotInserted=true
while not all strings are considered
  if first copy of $s_i$ > $s_{new}$ and keyNotInserted then
    write next copy of $s_{new}$ to output queue
    keyNotInserted=false
  write second copy of $s_i$ to output queue
\end{verbatim}

\textbf{Figure 4.1:} Insertion sort for fixed length strings

If the input sequence is large it may be preferable to stop comparing each string of the sorted sequence after the appropriate position for inserting the new string is found. When using elementary data types it was preferable to maintain a stable iteration body in order to benefit the most of software pipelining. In the case of strings, the compare operation is likely to cost significantly more than for elementary data types. The new algorithm is shown at figure 4.2.

\begin{verbatim}
while the $s_{new}$ is not inserted
  if first copy of $s_i$ > $s_{new}$ then write next copy of $s_{new}$ to output queue
  write second copy of $s_i$ to output queue
while there are more strings in the sorted sequence
  write first copy of $s_i$ to output queue
  discard second copy of $s_i$ through the use of a flush instruction
\end{verbatim}

\textbf{Figure 4.2:} Insertion sort : reducing the string compare operations

It is possible to implement the insertion sort method by only using one of the two data read queues. The data fetch sequence has to be modified to as follows:

$s_{1,1}, k_1, \ldots, s_{1, n}, k_n, k, s_{1, \ldots, s_N \ldots, 1, k_1, \ldots, s_N \ldots, k_n, k, s_N}$, where $k$ is the string under insertion, $s_{1, \ldots, s_N}$ are the strings of the sorted input sequence, $s_{i,j}$ indicates the $j$th character of the $i$th string and $k_j$ represents the $j$th character of the string under insertion. Again, two copies of each string of the sorted sequence are fetched, and plus $2N$ copies of the string under insertion. The characters of the first copy of each string of the sorted sequence are interleaved with the characters of a copy of the string under insertion. In this way, the processing unit can easily compare the two strings. Immediately after those two interleaved copies, a copy of the string under insertion follows. If the appropriate position for the insertion was found, this copy
is stored to the output queue. After that, the second copy of the string of the sorted sequence is fetched and used to copy the string to the output queue.

In our example, the read queues will contain \((a_0 x, o_x, a_x, x_0 i_x, o_x, x_i)\). In the first step, the \(a_x\) and \(o_x\) strings are compared; since \(a_x\) is smaller, the next copy of \(o_x\) is discarded and the second copy of \(a_x\) is copied to the output queue. Then the read queue contains \((x_0 i_x, o_x, x_i)\). Now the \(x_i\) string is compared with \(o_x\) and found to be larger, hence the next copy of the \(o_x\) string is copied to the output queue, followed by the second copy of the \(x_i\) string. Clearly this method consumes more memory bandwidth, since it fetches \(2N\) copies of the string under insertion instead of the \(N + 1\) copies fetched by the method that uses two read queues.

4.1.1 Using insertion sort with variable length keys

What made insertion-sort easy when fixed length strings were used was that the address generators could compute the string boundaries and easily fetch double copies of them. If variable length strings have to be sorted, it is not possible for the address generators to interleave double copies of the strings, since they do not know where the strings end.

Again, the data fetch sequence has to be modified. The second read queue is used to hold multiple copies of the string under insertion while the first read queue holds two copies of the sorted sequence. The strings of the sorted sequence are not fetched in an interleaved way; the second copy of the sorted sequence follows the first one in the read queue. The additional information required by the address generators is the length of the string under insertion and the length of the sorted sequence. Those can be computed in an initial step for each string of the input sequence. The algorithm consists of two main steps: during the first, all the strings of the sorted sequence are compared with the string under insertion and the result of these comparisons is recorded in a counter that points to the point of insertion. During the second step, this counter is used to insert the new string into the sorted sequence.

Consider for example the insertion of \("them\" string into the sorted sequence \("please, sort, too\""). For the first step of the algorithms the two read queues are filled with \((them, them,...)\) and \((please, sort, too)\). All the strings from the second queue are compared with the string under insertion the counter is 12, since the new string must be placed immediately after the second input string. The value of the counter gives the number of characters of the first two strings (the null characters are also counted).

During the second step of the algorithm, the two read queues are filled with \((them)\) and \((please, sort, too)\). The strings of the sorted are copied from the read queue to the output while decrementing the insertion counter. As soon as this counter reaches zero after \(sort\), the insertion point is reached, and the new string, \(them\), is copied to the output. The algorithm then copies the remaining strings to the output. The output produced is \((please, sort, them, too)\).

We could not find an implementation of insertion sort that uses only one read queue when variable-length string are to be sorted.

4.2 Using quicksort with strings

The implementation of quicksort discussed in the previous chapter uses both of the available read queues. This makes it impossible to fetch multiple copies of the pivots in a separate queue. Also, it is not possible to interleave a copy of the pivot with each copy of a string, since the pivots are different for each partition. The address generators don't know were each partition
ends in order to determine were the pivots of each partition are located. So, none of the data
fetching methods used when implementing insertion-sort is not applicable to quicksort.

It is possible to implement the quicksort algorithm if a small data store, local to the processing
unit, is included. This data store can only be accessed by the processing unit and it is in a
different address space of that used by the main memory. Two special operations can be used
to access this data store: one for loads and one for stores. Those two operations are adequate
for our purposes. It is convenient to think of this data store as an array which can be randomly
accessed by the processing unit.

If such a local data store is included, then a quicksort-like method that operates on fixed
length strings can be implemented as follows: again two copies of each string are fetched, back
to back. Each time a new partition starts, the pivot string is copied into the data store. That
copy is compared with each element found in the partition and the second copy of each string
is either copied to the output queue or discarded. Notice that for the case of fixed length strings
it is possible to fetch double copies of each string back to back and also fetch the elements
of the array of marks between successive strings. The address generators can compute the
boundaries of each string without needing to know its contents. Consider for example that the
input sequence is \((ba, ac, cb, bc)\) and that the array of marks is \((0, 0, 0, 1)\). The way the method
works for this example is illustrated in table 4.2.

As the string compare operation consumes significantly more execution time than the simple
integer/float compare operation, a slightly different method can be used in order to reduce the
number of string compares done. Each step of the algorithm is split into two parts. During the
first part all the strings are compared with the pivot of their partition only once and the results
are recorded in an array. Only one copy of the input sequence is fetched. During the second
part, the output sequence is produced. Two copies of the input sequence are fetched, one on
each of the two read queues. After each string, the result of the comparison of phase 1 is also
placed. The algorithm uses the first copy of a partition to generate the new subpartition that
consists of those elements that are smaller or equal than the pivot (comparison result '1') and
the other copy is used to produce the partition that consists of those elements that are larger than
the pivot (comparison result '0'). Consider for example the application of this method to the
input sequence \((ab, aa, ac, ba, bb, bc)\), at the stage when this sequence consists of two partitions
\((ab, aa, ac)\) and \((ba, bb, bc)\). The array of marks is \((0, 0, 1, 0, 0, 1)\). One of the two data read
queues is initially filled with \((ab, 0, aa, 0, ac, 1, ba, 0, bb, 0, bc, 1)\). The first pass creates the
\((ab, 0, 1, aa, 0, 1, ac, 1, 0, ba, 0, 1, bb, 0, 0, bc, 1, 0)\). Each string in this sequence is followed by
two numbers. The first number is the partition boundary mark and the second records the result
of the comparison: it is '1' for the strings that are less than or equal to the pivot of their partition,
and '0' otherwise. During the second pass, the two read queues are filled each with one copy of
that sequence, fetched in reverse order. The first copy is used to create the partition that consists
of elements that are less than or equal to the pivot, which for the first input partition \((ab, aa)\).
As soon as the first end of partition mark is encountered, the other copy of the sequence is used
to produce the partition that consists of elements that are larger than the pivot. These elements
are easily identified by the '0' value at their comparison result field.

The sequence produced during the first part is fetched in reverse order during the second
pass since the indicator of the comparison result for each string must be read before the
string it refers to. For our example, the second pass will create the following sequence:
\((ab, 0, aa, 1, ac, 1, ba, 1, bb, 0, bc, 0)\) in reverse order. Table 4.3 and table 4.4 show the details
of the application of the method discussed on the input sequence used in the previous example.
### Table 4.2: Example of the application of the quicksort–like method on fixed length strings

<table>
<thead>
<tr>
<th>Operation</th>
<th>Queue 0</th>
<th>Queue 1 Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy pivot to local store</td>
<td>((ba, ba, 0, ac, ac, 0, cb, cb, 0, ab, ab))</td>
<td>((1, ba, ba, 0, ac, ac, 0, cb, cb, 0, ab, ab, 1)), ()</td>
</tr>
<tr>
<td>copy ba</td>
<td>((ba, ba, 0, ac, ac, 0, cb, cb, 0, ab, ab))</td>
<td>((ba, 0, ac, ac, 0, cb, cb, 0, ab, ab, 1)), ((ba, 1, 1))</td>
</tr>
<tr>
<td>compare with ac</td>
<td>((ba, ba, 0, ac, ac, 0, cb, cb, 0, ab, ab))</td>
<td>((ac, 0, cb, cb, 0, ab, ab, 1)), ((ba, 1))</td>
</tr>
<tr>
<td>copy ac</td>
<td>((ba, ba, 0, ac, ac, 0, cb, cb, 0, ab, ab))</td>
<td>((0, cb, cb, 0, ab, ab, 1)), ((ba, 1, ac, 0))</td>
</tr>
<tr>
<td>compare with cb</td>
<td>((ba, ba, 0, ac, ac, 0, cb, cb, 0, ab, ab))</td>
<td>((cb, 0, ab, ab, 1)), ((ba, 1, ac, 0))</td>
</tr>
<tr>
<td>discard cb</td>
<td>((ba, ba, 0, ac, ac, 0, cb, cb, 0, ab, ab))</td>
<td>((0, ab, ab, 1)), ((ba, 1, ac, 0))</td>
</tr>
<tr>
<td>compare with ab</td>
<td>((ba, ba, 0, ac, ac, 0, cb, cb, 0, ab, ab))</td>
<td>((ab, 1)), ((ba, 1, ac, 0))</td>
</tr>
<tr>
<td>copy ab</td>
<td>((ba, ba, 0, ac, ac, 0, cb, cb, 0, ab, ab))</td>
<td>((1)), ((ba, 1, ac, 0, ab, 0))</td>
</tr>
<tr>
<td>copy pivot to local store</td>
<td>((ba, 0, ac, ac, 0, cb, cb, 0, ab, ab))</td>
<td>()((ba, 1, ac, 0, ab, 0))</td>
</tr>
<tr>
<td>discard ba</td>
<td>((0, ac, ac, 0, cb, cb, 0, ab, ab))</td>
<td>()((ba, 1, ac, 0, ab, 0))</td>
</tr>
<tr>
<td>compare with ac</td>
<td>((ac, 0, cb, cb, 0, ab, ab))</td>
<td>()((ba, 1, ac, 0, ab, 0))</td>
</tr>
<tr>
<td>discard ac</td>
<td>((0, cb, cb, 0, ab, ab))</td>
<td>()((ba, 1, ac, 0, ab, 0))</td>
</tr>
<tr>
<td>compare with cb</td>
<td>((cb, 0, ab, ab))</td>
<td>()((ba, 1, ac, 0, ab, 0))</td>
</tr>
<tr>
<td>copy cb</td>
<td>((0, ab, ab))</td>
<td>()((ba, 1, ac, 0, ab, 0, cb, 1))</td>
</tr>
<tr>
<td>compare with ab</td>
<td>((ab))</td>
<td>()((ba, 1, ac, 0, ab, 0, cb, 1))</td>
</tr>
<tr>
<td>discard ab</td>
<td>()</td>
<td>()((ba, 1, ac, 0, ab, 0, cb, 1))</td>
</tr>
<tr>
<td>operation</td>
<td>queue 0</td>
<td>queue 1 output</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td><strong>Part 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set ab as pivot</td>
<td>(0, aa, 0, ac, 1, ba, 0, bb, 0, bc)</td>
<td>()</td>
</tr>
<tr>
<td>pivot is equal to itself</td>
<td>(0, aa, 0, ac, 1, ba, 0, bb, 0, bc)</td>
<td>()</td>
</tr>
<tr>
<td>compare with aa</td>
<td>(0, ac, 1, ba, 0, bb, 0, bc)</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>compare with ac</td>
<td>(1, ba, 0, bb, 0, bc)</td>
<td>(1, 1, 0)</td>
</tr>
<tr>
<td>set ba as pivot</td>
<td>(0, bb, 0, bc)</td>
<td>(1, 1, 0)</td>
</tr>
<tr>
<td>pivot is equal to itself</td>
<td>(0, bb, 0, bc)</td>
<td>(1, 1, 0, 1)</td>
</tr>
<tr>
<td>compare with bb</td>
<td>(0, bc)</td>
<td>(1, 1, 0, 1)</td>
</tr>
<tr>
<td>compare with bc</td>
<td>()</td>
<td>(1, 1, 0, 1, 0)</td>
</tr>
</tbody>
</table>

Table 4.3: Reducing the number of compare operations of the quicksort–like method

It is worth noticing that the new method fetches only three copies of each string while the previous method fetched four copies.

In the case of variable length strings the method gets more complicated. The first problem, found in insertion sort as well, is that double copies of each string cannot be fetched by the address generators. The solution is similar to the one used in for insertion sort. Each step of the algorithm now consists of two parts. During the first, comparisons take place and the results, paired with the strings compared, are recorded. During the second step, the actual partitioned sequence is produced. The method is similar to that used for sorting strings of fixed length.

One other problem found is that it is not possible for the address generators to fetch the array of mark elements at the appropriate position, since they do not know were each string ends. This problem can be overcome if the array of input strings and the array of marks are merged during separate step of initialization. For example, if the input array is (sort, them, please, too) then this merging process will produce (sort, 0, them, 0, please, 0, too, 0, 1). An additional problem found is that the method discussed in the previous chapter produces the elements of the array of marks at shifted positions. Placing the marks at their appropriate positions can be achieved in an additional step. This means that all the input strings will be read one more time. Instead of that, the generation method used for the array of marks can be changed. With some additional bookkeeping, it is possible to generate marks that indicate if the string which is currently being written to the output queue is the last of a new partition or not.

For our example, the first step would produce the output sequence (sort, 0, 0, them, 0, 1, please, 0, 0, too, 1, 1). After each string, two numbers follow: the first is the boundary mark and the second is the comparison result. The same method used on sorting
<table>
<thead>
<tr>
<th>operation</th>
<th>queue 0</th>
<th>queue 1</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1, 0, cb, 0, 0, bb, 0, 1, ab, 1, 0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(0, cb, 0, 0, bb, 0, 1, ab, 1, 0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>()</td>
</tr>
<tr>
<td>copy bc</td>
<td>(0, cb, 0, 0, bb, 0, 1, ab, 1, 0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(0, 0, bb, 0, 1, ab, 1, 0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(bc)</td>
</tr>
<tr>
<td>copy bb</td>
<td>(0, cb, 0, 0, bb, 0, 1, ab, 1, 0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(0, 1, ab, 1, 0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(bc, 1, bb, 0)</td>
</tr>
<tr>
<td>discard ba</td>
<td>(0, cb, 0, 0, bb, 0, 1, ab, 1, 0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(1, 0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(bc, 1, bb, 0)</td>
</tr>
<tr>
<td>discard bc</td>
<td>(0, 0, bb, 0, 1, ab, 1, 0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(bc, 1, bb, 0)</td>
</tr>
<tr>
<td>discard bb</td>
<td>(0, 1, ab, 1, 0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(bc, 1, bb, 0)</td>
</tr>
<tr>
<td>copy ba</td>
<td>(1, 0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(bc, 1, bb, 0, ab, 1)</td>
</tr>
<tr>
<td>copy ac</td>
<td>(0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(0, 1, aa, 0, 1, ba, 1)</td>
<td>(bc, 1, bb, 0, ab, 1, ac, 1)</td>
</tr>
<tr>
<td>discard aa</td>
<td>(0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(0, 1, ba, 1)</td>
<td>(bc, 1, bb, 0, ab, 1, ac, 1)</td>
</tr>
<tr>
<td>discard ab</td>
<td>(0, ca, 0, 1, aa, 0, 1, ba, 1)</td>
<td>(1)</td>
<td>(bc, 1, bb, 0, ab, 1, ac, 1)</td>
</tr>
<tr>
<td>discard ac</td>
<td>(0, 1, aa, 0, 1, ba, 1)</td>
<td>()</td>
<td>(bc, 1, bb, 0, ab, 1, ac, 1, aa, 1)</td>
</tr>
<tr>
<td>copy aa</td>
<td>(0, 1, ba, 1)</td>
<td>()</td>
<td>(bc, 1, bb, 0, ab, 1, ac, 1, aa, 1, ab, 0)</td>
</tr>
<tr>
<td>copy ab</td>
<td>(1)</td>
<td>()</td>
<td>(bc, 1, bb, 0, ab, 1, ac, 1, aa, 1, ab, 0)</td>
</tr>
</tbody>
</table>

Table 4.4: Reducing the number of compare operations of the quicksort–like method
strings of fixed length can now be used.

4.3 Providing a small data store local to the processing unit

As it was shown, the inclusion of a small data store local to the processing unit, permits the implementation of sorting algorithms that operate on variable length strings as well. Also, algorithms that use small look up tables, such as the KMP pattern matching algorithm, can be easily implemented if such a data store is provided. The look up table can be stored within that data store and be used throughout the execution of the algorithm. In this way, it is no longer necessary to fetch multiple copies of the look up table in one of the data queues. Also, note that if this data store is adequately large, then in the case of the pattern matching algorithm presented, both the look up table and the pattern can be stored in it. This will significantly reduce the memory bandwidth requirements and simplify the code that implements the algorithm.

Significant memory bandwidth reduction will be observed on algorithms that operate on fixed length strings as well, since it will not be necessary any more to fetch two copies of the input strings.

It is expected that the cost of such a data store will be small. Only the processing unit will access it. Another issue is the size of this data store. For the algorithms presented it is required that this data store can hold at least the largest string that may occur. Theoretically a string can be of infinite length, but it is expected that for most of the applications the strings will of some hundred characters. So, a small data store, of say some Kbytes, will be adequate for most of the applications.

Note that this data store is like a software managed data cache. The difference from a conventional cache is that now the way in which this memory is filled and drained are completely controlled to the program executed.

By providing such a data store, the programmers can use knowledge regarding the locality of reference likely to be observed within the data space of programs and try to benefit from it.
5

Measurements

In this chapter we present the simulator of our architecture that was implemented as part of this thesis work, and some measurements that were collected using it. Two algorithms were measured: (i) the partition process which constitutes the core of the quicksort algorithm, and (ii) adding the elements of two vectors. The performance of the architecture presented is compared to that of a superscalar architecture with a cache memory. It is assumed that this architecture is capable of issuing four instructions per cycle, roughly corresponding to the four scalar processors in our architecture.

Initially, it is assumed that the superscalar architecture employs a cache of ideal performance, i.e. one with no delays due to memory accesses, and then our architecture is compared to the superscalar assuming that the cache the worst possible performance, i.e. that it misses all the time.

The measurements that refer to our architecture were collected through the use of a simulator developed during this work. In the first section an overview of this simulator can be found and some additional information regarding its use can be found at the appendices.

The programs that were simulated for collecting these measurements did not use software pipelining, and similarly the programs for the conventional architecture did not use methods such as loop unrolling.

5.1 Simulating the architecture

In order to validate the methods discussed in the previous chapters and to get a measure of the performance of the proposed architecture, a simulator was developed.

Six concurrently operating, functional units are simulated:

- Three address generators
- the processing unit
- the queues controller
- the main memory subsystem

Functional simulation is done on a cycle by cycle basis. Structural models were designed for each of the six functional units. The detail level of each of these models varies. The models of the main memory and of the queues controller are rather behavioral, while the model used
for the 4 processors is detailed. The same scalar processor model was used for the address
generators and for the processing unit. This simplified the development and the debugging
process. Two memory models were included — one simulating a simple memory system with
a constant access delay, and the other simulating the operation of an interleaved main memory.

All the models were parameterized, so that the user can define the following characteristics
using of configuration file:

- **memory**: model, interleaving factor, access delay, request and output queues sizes.
- **processors**: pipeline delays for each instruction, pipeline depth.
- **queues controller**: data and synchronization queue sizes.

In order to provide an adequately large main memory space, a binary file was used to hold
the contents of main memory. A simple cache and paging mechanism is used to reduce the
accesses to that file.

An interactive execution trace facility was also included in the simulator. When using this
facility, the user can query the state of the functional units, execute a specified number of cycles,
and selectively freeze or restart the operation of the four processors.

This simulator was implemented in the C programming language. C was selected mainly
in order to achieve adequate simulation speed, and because it is available on many platforms.
Initially we considered developing a simulator in Verilog, but this idea was abandoned since
this software is not widely available and the simulation speed would be significantly lower.

The simulator consists of about 3,600 lines of code, out of which as many as 1,600 lines
implement user interaction and configuration information manipulation routines. This C code
was successfully compiled and executed on three systems: a Sun–SparcStation 2 running
SunOS v4.2, a 386 based machine running SCO 386ix v3.2, and a 386/40 machine running
MSDOS. In the first two systems, we used the C compiler offered with the operating system,
cc, and on the MSDOS machine we used the Borland Turbo C v2.0 environment.

A simple assembler was also developed using AWK. The C preprocessor (cpp), was used to
provide macro facilities to the assembler. Using a unified model for all four processors made it
possible to use a single assembler for all the processors.

The simulation speed is about 1.15K cycles/sec on the Sun–Sparcstation2 when the simulator
is compiled with no optimizations. The execution of the program that runs for 3.6M cycles
took about 52 mins on that machine. On the 386 system, the simulation speed was about half.
Additional informations regarding the simulated units and the simulator interface can be found
in the appendices.

### 5.2 Partitioning $10^5$ elements

The partition process is the core of the quicksort algorithm. During this process the input
sequence is partitioned into two subsets. The first of them consists of the elements that are
smaller or equal than an pivot and the second contains the elements that are larger than the
pivot.

The method used in implementing the partition process was discussed in chapter 3 as part
of implementation of a quicksort–like algorithm. As can be seen in the appendices, the code
necessary to implement this process on our architecture contains 18 instructions if its assumed
that all operations have a pipeline latency of 2 cycles. Only 12 of the 18 instructions perform
useful work, while the remaining 6 are noops placed before each instruction that uses the result
of its immediately preceding instruction. Of the 18 instructions, 2 are loads (one reads the
mark value and the other reads the element it refers to), so 2/18 or 11% of the instructions are
loads.

The minimum sequence of operations required by the implementation of the partition process
is illustrated in figure 5.1. It is assumed that two memory read ports and one memory write port
are available.

1: read mark and next element
2: update pivot
3: compare element with pivot
4: store element to one of the two new partitions
5: store mark to the same partition

Figure 5.1: Minimum sequence of operations of the partition process

Each of the operations 2 to 4 must wait for the preceding operation to complete; since we
assumed that the integer unit has a latency of 2 cycles, 8 cycles are required to execute the
sequence of figure 5.1. If an architecture does not provide a branch prediction mechanism
then the branch penalty should be added to this figure. In the following discussion, no branch
penalties are assumed to exist on the conventional architectures.

Since the cache based architecture supports indexed stores, it is possible to create the two
new partitions concurrently: each time a new element is compared to the pivot it can be placed
in the appropriate partition. Thus, in a conventional architecture only one pass over the input
sequence is required, while, as we saw, two passes are required on our architecture.

If the partition process is applied to $10^5$ input elements and a cache with ideal performance is
employed, then 800,000 cycles will be required on a conventional architecture. If on the other
hand the hit ratio of the cache is 0%, then each memory access will be delayed by the latency of
the main memory. Assuming a memory latency of 100 cycles, the loads of each iteration will
observe a 100 cycles delay. One of the 8 instructions or 12.5% of all the instructions performs
a load so if a 100 cycle delay occur each time a load is performed then the cycles required for
partitioning $10^5$ elements will be: $10^5 \times (7 \times 1 + 1 \times 100) = 10,700,000$ cycles. In other
words the conventional architecture with a cache of 100% miss ratio is about 13.4 times slower
than the conventional architecture with 100% hit ratio on the data cache.

The simulation showed that 3,600,115 cycles are required to partition $10^5$ elements, when
the memory system has an access delay of 100 cycles. This means that the conventional
architecture that employs an ideal data cache is only 4.5 times faster, although our architecture
has a memory system which is 100 times slower and 11% of the instructions executed access it.
In other words, our architecture is about 3 times faster than the conventional architecture when
the latter does not exploit its cache due to excessive (100%) misses.

Table 5.1 summarizes the results discussed in this section. The ratio entry indicates how
many times an architecture is slower than the architecture of best performance.

Figure 5.2 shows the execution time required to partition $10^5$ elements, as a function of
memory access delay and read queue size.
<table>
<thead>
<tr>
<th>architecture</th>
<th>cycles required</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional, with 100% hit ratio</td>
<td>800,000</td>
<td>1.0</td>
</tr>
<tr>
<td>Our architecture</td>
<td>3,600,115</td>
<td>4.5</td>
</tr>
<tr>
<td>Conventional, with 100% miss ratio</td>
<td>10,700,000</td>
<td>13.4</td>
</tr>
</tbody>
</table>

**Table 5.1:** Partitioning $10^5$ elements, execution times

---

**Figure 5.2:** Partitioning 1000 elements – impact of memory access delay and data read queues size
5.3. Adding the elements of two vectors

The x-axis gives the queue size used and the y-axis shows the execution time (in cycles). The y-axis is in logarithmic scale and the minimum point shown is 30,000 cycles. The curves shown were measured with a different memory access delay each. The curves are drawn for a main memory with 1,5,10,15,20,40,100 or 200 cycles access delay. As can be seen, if the read queues have room for only few elements, then execution time varies significantly with the memory access delay. When the size of the queues is increased, the differences in execution time decrease. It can be seen that even with read queues of size only 12 elements, the execution time for the 100-cycle memory reaches its minimum value. This, however, is an artifact of the specific algorithm, and should not be interpreted as suggesting that the queue should be so sort. In fact, since the partition process does 2 data reads every 18 instructions, it is expected that read queues of $\frac{18}{12} \times memory\; delay\; size$ will be adequate in order not to observe delays due to space unavailability within the queues. If the queues are smaller than that figure then the address generators cannot fetch data far enough ahead of the data processor. It can be seen that if the memory access delay is 100 cycles then $\frac{18}{18} \times 100 = 11.1$, which agrees with the measurements of figure 5.2.

In general, a program that performs $r$ data reads from a queue every $i$ instructions requires a read queue of $\frac{r}{i} \times memory\; delay$ elements in order not to observe additional delays due to space unavailability within the queue. In other words:

$$data\; read\; queue\; size = \left[ \left( \frac{reads\; from\; queue}{instructions} \right) \times memory\; access\; delay \right]$$

5.3 Adding the elements of two vectors

In this section we present measurements of the performance of our architecture when adding two vectors, of size $10^4$. The minimum sequence of operations required is shown in figure 5.3.

read $a_i$ and $b_i$  
add $a_i$ and $b_i$  
store result

Figure 5.3: Minimum sequence of operations to add two vectors

If a latency of two cycles is assumed on the integer unit, then 5 cycles are required to execute the above sequence. So, on a conventional architecture with a 100% data cache hit ratio, 50,000 cycles will be required to add two vectors of $10^4$ elements each. As only one of the five instructions is a memory load, the cycles required on a conventional architecture with a 0% hit ratio, and 100 cycles memory access delay, will be $10^4 \times (4 \times 1 + 1 \times 100) = 1,040,000$.

On our architecture the program of the processing unit consists of only 2 instructions:

The first instruction takes one element from the first read queue, adds it with an element from the second read queue, and stores the result to the output queue. Simulation showed that $30,117^1$ cycles were required to add two vectors of $10^4$ elements each. This means that

---

1. The scalar address generators cannot produce a new address every two cycles since 3 instructions are required to produce every address: write to queue, increment index, brcmp.
our architecture is about 1.6 times faster than the conventional architecture with an optimal performance data cache.

Through the use of loop unrolling it is possible to improve the performance of the conventional architectures. In the assumed case of a superscalar architecture capable of performing 4 operations concurrently, it is possible to produce one new element every cycle by using the schedule of table 5.2.

<table>
<thead>
<tr>
<th>op 1</th>
<th>op 2</th>
<th>op 3</th>
<th>op 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>load $a_0$</td>
<td>load $b_0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load $a_1$</td>
<td>load $b_1$</td>
<td>$r_0=\text{add } a_0,b_0$</td>
<td></td>
</tr>
<tr>
<td>load $a_3$</td>
<td>load $b_3$</td>
<td>$r_1=\text{add } a_1,b_1$</td>
<td>store $r_0$</td>
</tr>
<tr>
<td>load $a_4$</td>
<td>load $b_4$</td>
<td>$r_2=\text{add } a_2,b_2$</td>
<td></td>
</tr>
<tr>
<td>load $a_5$</td>
<td>load $b_5$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>load $a_i$</td>
<td>load $b_i$</td>
<td>$r_{i-2}=\text{add } a_{i-2},b_{i-2}$</td>
<td>store $r_i - 4$</td>
</tr>
</tbody>
</table>

Table 5.2: Ideal execution schedule on a 4 way superscalar architecture

Note that if only 4 operations can be issued per cycle, it is not possible to use the above schedule, since the branch instructions cannot be included. If we assume that a new element can be produced every cycle, then the conventional architecture with a 100% hit ratio would take about 10,000 cycles to add two vectors of $10^4$ elements each. This means that an ideal conventional architecture would be only 3 times faster than our architecture.
Conclusions, Summary

Both access decoupling and software pipelining can significantly improve the execution performance. Access decoupling can, partially or even totally, hide long memory delays. Software pipelining can be used to interleave the execution of code that may not be vectorizable and thus to better utilize the available functional units. Combining the two methods is straightforward.

Conditional execution can reduce the need for control transfer instructions in some cases and thus it can reduce the delays due to them. This technique is also attractive because it allows the application software pipelining even to loops that include complex conditional constructs.

Both access decoupling and software pipelining are aimed to be used on large input sizes, since performance remains high or even increases with the data set size, while the performance of cache memories degrades when the data set size grows.

Not all algorithms can be efficiently be executed on architectures that employ these techniques. The performance of access decoupling is significantly degraded if the data fetch or store sequence depends on the values of the data fetched, while the benefits gained from software pipelining are reduced by the recurrences that exist in the algorithm.

It is relatively easy to implement many numerical algorithms on access decoupled and software pipelined architectures, since the contain few recurrences and data independent fetch/store sequences. On the contrary, most of the non-numerical algorithms produce data dependent fetch/store sequences and have many recurrences, which makes their implementation on architectures such as the above not a trivial task.

Few nonnumerical algorithms are likely to be implemented without any modifications, these modifications are not likely to be doable automatically (i.e. by the compiler). Although some general guidelines were drawn on how to modify nonnumerical algorithms in order to make them appropriate for these architectures, it is expected that the effort will be left to the programmer, since algorithm rewriting is needed.

Most of the algorithms presented in this work, when modified for a decoupled architecture, use significantly larger memory bandwidth requirements than they did on conventional architectures, because in most cases additional data are fetched or stored just to maintain data independent fetch/store sequences.

In order to increase the set of algorithms that can be implemented on architectures such as ours, more data read and store queues can be provided, and a separate address generator can be incorporated for each of them. The code necessary to implement an algorithm on architectures that combine access decoupling and software pipelining is significantly simplified if separate address generators are provided for each data queue in the system. Also, the inclusion of a data store local to the processing unit can not only increase the set of appropriate algorithms but can
also simplify the required code and reduce the memory bandwidth requirements.

Table 6.1 summarizes the algorithms examined in this work.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>appropriate for access decoupling</th>
<th>appropriate for software pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>yes</td>
<td>only with multiple inputs sequences</td>
</tr>
<tr>
<td>shellsort</td>
<td>yes</td>
<td>only with multiple inputs sequences</td>
</tr>
<tr>
<td>bubble sort</td>
<td>yes</td>
<td>only with multiple inputs sequences</td>
</tr>
<tr>
<td>selection sort</td>
<td>yes</td>
<td>only with multiple inputs sequences</td>
</tr>
<tr>
<td>quicksort</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>heapsort</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>mergesort</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>KMP string search</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Brute force string search</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 6.1: Algorithms discussed in this work

The performance of architectures such as ours varies significantly with the program executed. Through the use of an simulator, it was shown that our architecture has an execution performance close to that of an ideal machine while operating with a significantly slower memory.

This work is far from being a complete study of the possible implementations of nonnumerical algorithms on decoupled architectures that support software pipelining and the architecture presented is also far from being usefull as its processing unit is only a scalar processor. The work presented here can be extended in several directions, two of which are:

- consider more nonnumerical algorithms.
- study the behaviour and performance of a superscalar processing unit.

The algorithms discussed in this work constitute a small subset of the existing non-numerical algorithms, since we have mainly considered sorting algorithms. Nevertheless, we provided some general guidelines on how to cope with the problems arising. A detailed consideration of more nonnumerical algorithms is required in order to obtain complete view of the problems that may occur. This will also help to qualify the usefullness of the architectural extensions discussed and form a basis decide which one to keep.
Appendix A

Processor model

Each of the four processors was simulated according to the model shown in figures A.1, A.2 and A.3.

The pipeline is separated in the following stages:

- **1st stage**: Instruction issue, i.e., logic that supports in-order instruction issuing and the $brtop$ and $wtop$ branch without delay instructions.
- **2nd stage**: Instruction decoding, register/guard/port file access, control transfer logic.
- A variable number of stages for ALU operations.
- A variable number of stages for accessing the local data cache on the address generators and the local data store on the processing unit.

---

**Figure A.1**: Pipeline overview, stage 1
Figure A.2: Pipeline overview, stage 2

Figure A.3: Pipeline overview, stage 3 to last
Appendix B

Instruction set

There are seven different types of instructions:

- Arithmetic
- Compare
- Logical
- Guard manipulation
- data cache access
- control transfer
- special operations

Table B.1 summarizes the arithmetic instructions available. Note that $Imm_{10}$ is always sign extended to 32-bits before its use.

<table>
<thead>
<tr>
<th>mmemonic</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_a = \text{add } r_b, r_c$</td>
<td>$R_a \leftarrow r_b + r_c$</td>
</tr>
<tr>
<td>$R_a = \text{addi } r_b, Imm_{10}$</td>
<td>$R_a \leftarrow r_b + Imm_{10}$</td>
</tr>
<tr>
<td>$R_a = \text{sub } r_b, r_c$</td>
<td>$r_d \leftarrow R_a - r_c$</td>
</tr>
<tr>
<td>$R_a = \text{subi } r_b, Imm_{10}$</td>
<td>$R_a \leftarrow r_b - Imm_{10}$</td>
</tr>
<tr>
<td>$R_a = \text{mul } r_b, r_c$</td>
<td>$r_d \leftarrow R_a \times r_c$</td>
</tr>
<tr>
<td>$R_a = \text{muli } r_b, Imm_{10}$</td>
<td>$R_a \leftarrow r_b \times Imm_{10}$</td>
</tr>
<tr>
<td>$r_d = \text{sc1 } R_a, r_b, r_c$</td>
<td>$R_d \leftarrow \text{if bit}(0, R_a) = 1 \text{ then } r_b \text{ else } r_c$</td>
</tr>
<tr>
<td>$r_a = \text{setlo } Imm_{16}$</td>
<td>$\text{bits}(0..15, r_a) \leftarrow Imm_{16}$</td>
</tr>
<tr>
<td>$r_a = \text{sethi } Imm_{16}$</td>
<td>$\text{bits}(16..31, r_a) \leftarrow Imm_{16}$</td>
</tr>
</tbody>
</table>

Table B.1: Arithmetic instructions

The notations used is as follows:

- $r_z$ refers to a register of the static, of the rotating or of the port file.
- $R_z$ refers to a register of the static, of the rotating, of the port file or of the guard file.
• \(g_x\) refers to a register that belongs to the guard file.

• \(sr\) refers to one of the special registers available. Those are the \(lc\) or loop count, the \(mcp\) or multiconnect pointer and the \(esc\) or epilog stage count.

• \(pr\) refers to a register that belongs to the port file.

• \(\text{bit}(i, x)\) denotes the \(i\)th bit of the operand \(x\).

The port file consists of the following ports:

• \(rp0\) or read port 0, it is read only, exists only on the processing unit and it used to access the first of the two data read queues.

• \(rp1\) or read port 1, it is read only, exists only on the processing unit and it used to access the second of the two data read queues.

• \(wp\) or write port, it is write only. Exists on all processors. On the processing unit it is used to refer to the data store queues. On the address generators it is used to refer to the corresponding data queue. Through this port the address generators store the addresses they produce in the appropriate queue.

• \(sp0\) or sync port 0, it is read/write. Exists on all the processors and used to access one of the synchronization queues. The processing unit access the synchronization queue from or to the address generator 0 through this port. The address generators access their synchronization queue that communicates with the processing unit.

• \(sp1\) or synchronization port 1, it is read/write. Exists only in the processing unit and used to access the synchronization queue from/to the address generator 1.

• \(sp2\) or synchronization port 2, it is read/write. Exists only in the processing unit and used to access the synchronization queue from/to the address generator 2.

• \(sp3\) or synchronization port 3, it is write only. Exists only in the processing unit and it is actually an alias to all the three synchronization ports.

Table B.2 shows the compare instructions available.

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Ra = \text{cmeq} \ r_b, r_c)</td>
<td>(Ra \leftarrow 1) if (r_b = r_c), 0 otherwise</td>
</tr>
<tr>
<td>(Ra = \text{cmeqi} \ r_b, \text{Imm}_{10})</td>
<td>(Ra \leftarrow 1) if (r_b = \text{Imm}_{10}), 0 otherwise</td>
</tr>
<tr>
<td>(Ra = \text{cmne} \ r_b, r_c)</td>
<td>(Ra \leftarrow 1) if (r_b &lt;&gt; r_c), 0 otherwise</td>
</tr>
<tr>
<td>(Ra = \text{cmnei} \ r_b, \text{Imm}_{10})</td>
<td>(Ra \leftarrow 1) if (r_b &lt;&gt; \text{Imm}_{10}), 0 otherwise</td>
</tr>
<tr>
<td>(Ra = \text{cmge} \ r_b, r_c)</td>
<td>(Ra \leftarrow 1) if (r_b &gt;= r_c), 0 otherwise</td>
</tr>
<tr>
<td>(Ra = \text{cmgei} \ r_b, \text{Imm}_{10})</td>
<td>(Ra \leftarrow 1) if (r_b &gt;= \text{Imm}_{10}), 0 otherwise</td>
</tr>
<tr>
<td>(Ra = \text{cmlt} \ r_b, r_c)</td>
<td>(Ra \leftarrow 1) if (r_b &lt; r_c), 0 otherwise</td>
</tr>
<tr>
<td>(Ra = \text{cmlli} \ r_b, \text{Imm}_{10})</td>
<td>(Ra \leftarrow 1) if (r_b &lt; \text{Imm}_{10}), 0 otherwise</td>
</tr>
</tbody>
</table>

**Table B.2:** Compare operations available

Table B.3 shows the logical instructions available.
<table>
<thead>
<tr>
<th>mnemonic</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_a = \text{and } r_b, r_c$</td>
<td>$R_a \leftarrow r_b$ binary and $r_c$</td>
</tr>
<tr>
<td>$R_a = \text{andi } r_b, \text{Imm}_{10}$</td>
<td>$R_a \leftarrow r_b$ binary and $\text{Imm}_{10}$</td>
</tr>
<tr>
<td>$R_a = \text{or } r_b, r_c$</td>
<td>$R_a \leftarrow r_b$ binary and $r_c$</td>
</tr>
<tr>
<td>$R_a = \text{ori } r_b, \text{Imm}_{10}$</td>
<td>$R_a \leftarrow r_b$ binary or $\text{Imm}_{10}$</td>
</tr>
<tr>
<td>$R_a = \text{xor } r_b, r_c$</td>
<td>$R_a \leftarrow r_b$ binary xor $r_c$</td>
</tr>
<tr>
<td>$R_a = \text{xori } r_b, \text{Imm}_{10}$</td>
<td>$R_a \leftarrow r_b$ binary xor $\text{Imm}_{10}$</td>
</tr>
<tr>
<td>$R_a = \text{shru } r_b, r_c$</td>
<td>$R_a \leftarrow r_b$ shifted right unsigned by $r_c$</td>
</tr>
<tr>
<td>$R_a = \text{shru } r_b, \text{Imm}_{10}$</td>
<td>$R_a \leftarrow r_b$ shifted right unsigned by $\text{Imm}_{10}$</td>
</tr>
<tr>
<td>$R_a = \text{shrs } r_b, r_c$</td>
<td>$R_a \leftarrow r_b$ shifted right signed by $r_c$</td>
</tr>
<tr>
<td>$R_a = \text{shrs } r_b, \text{Imm}_{10}$</td>
<td>$R_a \leftarrow r_b$ shifted left by $\text{Imm}_{10}$</td>
</tr>
<tr>
<td>$R_a = \text{shl } r_b, r_c$</td>
<td>$R_a \leftarrow r_b$ shifted left by $r_c$</td>
</tr>
<tr>
<td>$R_a = \text{shl } r_b, \text{Imm}_{10}$</td>
<td>$R_a \leftarrow r_b$ shifted right signed by $\text{Imm}_{10}$</td>
</tr>
</tbody>
</table>

Table B.3: Logical operations available

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_a = \text{mld } r_b, \text{Imm}_{10}$</td>
<td>$R_a \leftarrow \text{data mem}(\text{Imm}_{10} + r_b)$</td>
</tr>
<tr>
<td>$\text{mst } R_a, r_b, \text{Imm}_{10}$</td>
<td>$\text{data mem}(\text{Imm}_{10} + r_b) \leftarrow R_a$</td>
</tr>
</tbody>
</table>

Table B.4: Data cache/store access instructions

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{breq } r_a, \text{Imm}_{16}$</td>
<td>if $r_a = 0$ then $pc = pc + \text{Imm}_{16}$</td>
</tr>
<tr>
<td>$\text{bne } r_a, \text{Imm}_{16}$</td>
<td>if $r_a &lt; 0$ then $pc = pc + \text{Imm}_{16}$</td>
</tr>
<tr>
<td>$\text{bge } r_a, \text{Imm}_{16}$</td>
<td>if $r_a \geq 0$ then $pc = pc + \text{Imm}_{16}$</td>
</tr>
<tr>
<td>$\text{bgt } r_a, \text{Imm}_{16}$</td>
<td>if $r_a &gt; 0$ then $pc = pc + \text{Imm}_{16}$</td>
</tr>
<tr>
<td>$\text{bgt } r_a, \text{Imm}_{16}$</td>
<td>if $r_a &gt; 0$ then $pc = pc + \text{Imm}_{16}$</td>
</tr>
<tr>
<td>$\text{ble } r_a, \text{Imm}_{16}$</td>
<td>if $r_a \leq 0$ then $pc = pc + \text{Imm}_{16}$</td>
</tr>
<tr>
<td>$\text{brel } r_a, \text{Imm}_{16}$</td>
<td>if $r_a &lt; 0$ then $pc = pc + \text{Imm}_{16}$</td>
</tr>
<tr>
<td>$\text{brel } r_a, \text{Imm}_{16}$</td>
<td>if $r_a &lt; 0$ then $pc = pc + \text{Imm}_{16}$</td>
</tr>
<tr>
<td>$\text{callr } r_a$</td>
<td>$svy = pc$ $pc = pc + r_a$</td>
</tr>
</tbody>
</table>

Table B.5: Control transfer instructions
Table B.4 shows the data cache/store access instructions available. Table B.5 shows control transfer instructions available. Table B.6 shows guard manipulation instructions available.

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g_d = g_3 \text{and} R_a, g_b, g_c )</td>
<td>( g_d \leftarrow \text{bit}(0, R_a) \cdot g_b \cdot g_c )</td>
</tr>
<tr>
<td>( g_d = g_3 \text{andnot} R_a, g_b, g_c )</td>
<td>( g_d \leftarrow \text{bit}(0, R_a) \cdot g_b \cdot \text{not}(g_c) )</td>
</tr>
<tr>
<td>( g_d = g_3 \text{andnor} R_a, g_b, g_c )</td>
<td>( g_d \leftarrow \text{bit}(0, R_a) \cdot \text{not}(g_b) \cdot \text{not}(g_c) )</td>
</tr>
<tr>
<td>( g_d = g_3 \text{nor} R_a, g_b, g_c )</td>
<td>( g_d \leftarrow \text{not(\text{bit}(0,R_a) \cdot g_b \cdot g_c)} )</td>
</tr>
<tr>
<td>( g_d = g_3 \text{xor} R_a, g_b, g_c )</td>
<td>( g_d \leftarrow \text{bit}(0, R_a) \oplus g_b \oplus g_c )</td>
</tr>
</tbody>
</table>

**Table B.6: Guard manipulation instructions**

Finally table B.7 shows the special instructions.

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g_a = \text{brtop} )</td>
<td>explain in chapter 2</td>
</tr>
<tr>
<td>( g_a = \text{wtop} )</td>
<td>explain in chapter 2</td>
</tr>
<tr>
<td>( g_a = \text{espm} )</td>
<td>enter software pipeling execution mode</td>
</tr>
<tr>
<td>( sr = \text{setsr} )</td>
<td>( sr \leftarrow r_a )</td>
</tr>
<tr>
<td>( sr = \text{setsri} )</td>
<td>( sr \leftarrow \text{Imm}_{16} )</td>
</tr>
<tr>
<td>( sr = \text{flsh} )</td>
<td>flush ( r_a ) elements out of the ( sr ) queue</td>
</tr>
<tr>
<td>( sr = \text{flshi} )</td>
<td>flush \text{Imm}_{16} elements out of the ( sr ) queue</td>
</tr>
</tbody>
</table>

**Table B.7: Special instructions**

In order to flush all the elements from a queue a flush instruction is used with a negative parameter.

The three instruction formats are:

![Instruction formats](image-url)
Appendix C

Simulator Usage

The simulator is called *spadasim* standing for “software pipelining–access decoupling architecture simulator”. It can be found at plouton.csi.forth.gr in ~ moshovos/MSc/SPADASIM. It can be called through the following command:

```
% spadasim [-b I -i] [configuration-file]
```

The s (silent mode) option suppresses any messages during execution. The b (batch mode) option is used to suppress the interactive execution mode which is called by default. Finally *configuration-file* parameter specifies a configuration file other than the default, *spadasim.ini*.

The configuration file consists of lines that have the form:

```
specifier: value
```

Each line that begins with an # character is assumed to contain comments and ignored. The specifier field can be one of the following:

- mem.model
- mem.interleave
- mem.delay
- mem.bank.request.queue.size
- mem.output.queue.size
- qc.read.queue.0.size
- qc.read.queue.1.size
- qc.write.queue.size
- qc.sync.queue.0.size
- qc.sync.queue.1.size
- qc.sync.queue.2.size
- pu.cache.file
- ag0.cache.file
- ag1.cache.file
- ag2.cache.file
- mem.file
- stat.file
pu.stages
ag0.stages
ag1.stages
ag2.stages
opcode.delay.file

Character case is not significant in the configuration file. The mem.model can take two values: interleaved or noninterleaved. In the first case the interleaved memory model is used while in the second the constant delay memory model is used.

The mem.interleave takes a integer value which specifies the interleave factor when the interleaved memory model is used. The mem.delay takes an integer value which indicates the delay in cycles when accessing the memory subsystem. When the interleaved memory model is used this parameter specifies the cycle time of each memory bank.

The mem.bank.request.queue.size takes an integer as parameter which specifies the size of the pending request for each bank when the interleaved memory model is used.

The mem.output.queue.size takes an integer as parameter and specifies the length of the output queue of the memory subsystem. This queue is used to return the data of previous requested loads.

The QC.read.queue.1.size,QC.read.queue.0.size and the QC.write.queue.size all take an integer as parameter which specifies the length of the corresponding data queue.

The QC.sync.queue.0.size,QC.sync.queue.1.size and the QC.sync.queue.2.size are used to define the lengths of the three synchronization queues.

The pc.cache.file,ag0.cache.file,ag1.cache.file and the ag2.cache.file take a string as a parameter which is assumed to be the name of the file that contains the program, in binary form for the corresponding processor. Those code files are generated through the assembler and their format is the following:

address data

Both fields must be in hexadecimal form.

The mem.file specifier takes a string as parameter which it is assumed to the name of the binary file used for simulating the main memory. The main memory is only word addressed and each word is assumed to be of 4 bytes long.

The stat.file specifies the file in which the statistics should be written.

The pu.stages,ag0.stages,ag1.stages and ag2.stages defines the pipeline depth of the corresponding processor.

Finally the opcode.delay.file defines the file in which the pipeline latencies per instruction are defined. Such a file consists of lines that have the following form:

mnemonic: delay

where the mnemonic field specifies the instruction and the delay field is an integer which defines the pipeline latency in cycles of the corresponding instruction. Note that there is only one global table of instructions latencies for all four processors.

Note that no type checking is done when reading these files.
Appendix D

Queues implementation

The details of the implementation of both the data and the synchronization queues are shown.

D.1 Data queues

The three data queues have elements that consist of three fields:

- data field
- address field
- state field

The address field is written only by the address generators, the status and the data fields are manipulated by the queues controller. The data read queues the data received from memory are placed at the data field, while at the store queue the data produced by the processing unit are written to the data field and later they are send by the queues controller to main memory.

Each queue is access through three pointers. The agp, the pup and the qep. The first is used when an access is done by the address generator the second when is done by the processing unit and the third when is done by the queues controller.

The state field for the data read queues can have one of the following values:

- $E$ or empty, if the elements is free to be used.
- $AG$ or address generated, if the address field holds valid information but the memory request was not yet performed by the queues controller.
- $RQ$ or requested, if the address field is valid and the data was requested by main memory but not yet received.
- $C$ or complete, if both the data and address field are valid; the element can be processed by the processing unit.
- $RQF$ or requested but flushed. Is used to indicate that an element which was requested from main memory was requested to be flushed by the processing unit.
**Figure D.1:** State transitions of the elements of data read queue

Figure D.1 shows the state transitions possible. The state field of the elements of the data store queue can have one of the following values:

- **E** or empty.
- **AG** or address generated. Only the address field is valid.
- **C** or complete. Both the data and the address field are valid and the element can be sent to main memory.

Figure D.2 shows the state transitions possible.

**Figure D.2:** State transitions of the elements of data store queue

---

**D.2 Synchronization queues**

The synchronization queues are implemented by counters. Each time a processor writes to a synchronization queue its count is incremented. Each time a processor reads from a
synchronization queue an '1' is returned and the corresponding counter is decremented if it is larger than zero, otherwise a '0' is returned.

Flushes are implemented as subtrahends to the counter of the synchronization queue.
Appendix E

Programs

The programs necessary to implement the partition process used in quicksort and the KMP string matching algorithm are shown.

E.1 Partition process of quicksort

The program of the address generator 0 is:

```c
#include "qst.inc"

org 8
g1 : sr3 = setlo ELEMS_L
g1 : sr1 = setlo SOURCEM_L
g1 : sr3 = sethi ELEMS_H
g1 : sr2 = setlo SOURCEA_L
g1 : sr1 = sethi SOURCEM_H
g1 : sr2 = sethi SOURCEA_H
g1 : lc = setsr sr3
g1 : esc = setsri 0
g1 : g5 = espm
loop:
  .
g5 : wp = add sr0 sr1
g5 : wp = add sr0 sr2
g5 : sr2 = addi sr2 1
g5 : sr1 = addi sr1 1
g1 : g4 = brtop loop
stop
```

The program of the address generator 1 is:

```c
#include "qst.inc"

org 8
g1 : sr3 = setlo ELEMS_L
```
g1 : sr1 = setlo SOURCEM_L
g1 : sr3 = sethi ELEMS_H
g1 : sr1 = sethi SOURCEM_H
g1 : sr2 = setlo SOURCEA_L
g1 : sr1 = addi sr1 1
g1 : sr2 = sethi SOURCEA_H
g1 : lc = setsr sr3
g1 : esc = setsri 0
g1 : g5 = espm
loop:
g5 : wp = add sr0 sr2
g5 : wp = add sr0 sr1
g5 : sr2 = addi sr2 1
g5 : sr1 = addi sr1 1
g1 : g4 = brtop loop
stop

The program of the processing unit is:

#include "qst.inc"

#define Elem rr0
#define Mark rr01
#define Mark1 rr02
#define Key rr3
#define Key1 rr4

#define isGE g14
#define isGE1 g15
#define Larger g16
#define Larger1 g17
#define First g18
#define First1 g19
#define NOOP g1': sr0 = add sr0 sr0

org 8
init:
g1 : sr1 = setlo ELEMS_L
g1 : esc = setsri 0
g1 : sr1 = sethi ELEMS_H
NOOP
g1 : sr1 = add sr1 sr1
g1 : sr2 = addi sr0 1
g1 : sr1 = addi sr1 1
NOOP
g1 : lc = setsr sr1
g1 : g5 = espm
part:
g5 : Mark = sel Larger1 rp1 rp0
NOOP
g5 : Larger = g3xor Mark Larger1 g0
NOOP
g5 : Elem = sel Larger rp1 rp0
NOOP
g5 : Key = sel Mark Elem Key1
g1 : isGE = g3and g0 g0 g0
NOOP
g5 : isGE = cmge Elem Key
NOOP
g5 : isGE = g3xor Larger isGE g0
NOOP
g5 : First = g3or Mark First1 g0
isGE : wp = add sr0 Elem
isGE : wp = sel First sr02 sr0
isGE : First = g3and g0 g0 g0

stop

And the program of the address generator 2 (data stores) is:

#include "qst.inc"

org 8

g1 : sr4 = setlo 100
g0 : sr0 = add sr0 sr0
g1 : sr4 = sethi 100
g1 : sr3 = setlo ELEMS_L
g1 : sr1 = setlo TARGETA_L
g1 : sr3 = sethi ELEMS_H
g1 : sr2 = setlo TARGETM_L
g1 : sr1 = sethi TARGETA_H
g1 : sr2 = sethi TARGETM_H
g1 : lc = setsr sr3
g1 : esc = setsri 0
g1 : g5 = espm
loop:
g5 : wp = add sr0 sr1
g5 : wp = add sr0 sr2
g5 : sr1 = addi sr1 1
g5 : sr2 = addi sr2 1
g1 : g4 = brtop loop
stop

The “qst.inc” include file was:

#define ELEMS_L 34464
#define ELEMS_H 0001
#define SOURCEA_L 0000
#define SOURCEA_H 0000
#define SOURCEM_L 34464
#define SOURCEM_H 0001
#define TARGETA_L 3393
#define TARGETA_H 0003
#define TARGETM_L 37857
#define TARGETM_H 0004

E.2 KMP string searching

The programs shown implement the KMP string searching algorithm for the example shown in chapter 3.

The program of the address generator 0 is:

#include "kmps.inc"

#define Elements sr3
#define ElementsIdx sr1
#define NOOP g1 : sr0 = add sr0 sr0

org 8

org 8

The program of the address generator 1 is:
#include "kmps.inc"
#define NOOP g0 : sr0 = add sr0 sr0
#define LoopTimes rr1
#define KeySize sr1
#define KeyFirst sr2
#define KeyLast sr3
#define KeyIdx sr4
#define NextFirst sr5
#define NextIdx sr6

org 8

; Address generator 1:

label g1 : KeySize = setlo NEXTSIZE_L
label g1 : KeyFirst = setlo KEY_L
label g1 : KeySize = sethi NEXTSIZE_H
label g1 : KeyFirst = sethi KEY_H
label g1 : NextIdx = setlo SOURCENEXT_L
label g1 : LoopTimes = mul LoopTimes KeySize
label g1 : NextIdx = sethi SOURCENEXT_H
label g1 : KeyLast = add KeySize KeyFirst
label g1 : NextFirst = add sr0 NextIdx
label g1 : esc = setsri 0
label g1 : g5 = espm
loop:
label g5 : NextIdx = addi NextIdx 1
label g5 : wp = add sr0 NextIdx
label g5 : KeyIdx = addi KeyIdx 1
label g5 : wp = add sr0 KeyIdx
label g5 : g15 = cmge KeyIdx KeyLast
label g5 : g16 = g3xor sp0 g1 g0
label g5 : KeyIdx = sel g15 KeyFirst KeyIdx
label g5 : NextIdx = sel g15 NextFirst NextIdx
label g1 : g4 = wtop g5 g16
label g1 : sp0 = add sr0 sr0

stop

The program of the address generator 2 is:

#include "kmps.inc"

#define NOOP g1 : sr0 = add sr0 sr0

org 8

start:
label g1 : sr1 = setlo TARGET_L
NOOP
label g1 : sr1 = sethi TARGET_H
NOOP

g1 : wp = add sr0 sr1
stop

The program of the processing unit is:

#include "kmps.inc"
#define Element rr1
#define Pattern rr2
#define Next rr3
#define NotFinished g15
#define ADV g17

#define NOOP g1 : sr0 = add sr0 sr0

org 8

start:
g1 : sr1 = addi sr0 1
g1 : esc = setsri 0
g1 : g5 = espm
loop:
g5 : Element = add rp0 sr0
g5 : Next = add rp1 sr0
g5 : Pattern = add rp1 sr0
g5 : NotFinished = cmgei Next 0
g5 : ADV = cmne Element Pattern
NOOP

ADV : rp1 = flush Next

loop1: g1 : sp0 = breq loop1
g1 : rp0 = flushi -1
loop2:
g1 : rp1 = flushi -1

The file "kmps.inc" included was:
E.2. KMP string searching

#define ELEMS_L 0008
#define ELEMS_H 0000
#define SOURCENEXT_L 0017
#define SOURCENEXT_H 0000
#define KEY_L 0008
#define KEY_H 0000
#define NEXTSIZE_L 0009
#define NEXTSIZE_H 0000
#define SOURCE_L 0000
#define SOURCE_H 0000
#define TARGET_L 1000
#define TARGET_H 0000
Bibliography


