A FUNCTIONAL LANGUAGE MACHINE BASED ON QUEUES

Guang Liang Xiong

November 1984
CSD-840047

UNIVERSITY OF CALIFORNIA

Los Angeles

A Functional Language Machine Based on Queues

A thesis submitted in partial satisfaction of the

requirement for the degree Master of Science

in Computer Science

by

Guang Liang Xiong

1984

Table of Contents

page

ABSTRACT OF THE THESIS

A Functional Language Machine Based on Queues

by

Guang Liang Xiong

Master of Science in Computer Science

University of California, Los Angeles, 1984

Professor Milos D. Ercegovac, Chair

A machine organization based on queues is proposed to execute the Berkeley FP programs in the format of Q-code.

The proposed machine consists of an instruction queue, a data queue, an execution unit and memory modules.

The overall organization and each part of the machine are described. Several alternatives for the main components are suggested and compared. The potentials of concurrent executions are explored. A simulation package of the machine and its performance are reported.

The characteristics of the Q-machine include the simplicity of the organization, the beneficial implicit address management of the queues, the doubled bandwidth of memory access and the simplicity of the compilation.

CHAPTER 1

Introduction

In 1978, Backus, in his well-known Turing Award lecture [1], discussed a new style of programming language - functional programming language. The functional programming language breaks the circle of conventional programming languages and leads to a new direction of programming style and computer architecture. There are numerous research projects, trying to develop a suitable computer architecture to execute programs in functional programming languages[10]. As another attempt in this direction, a functional programming language machine based on queues, is discussed in this thesis.

1.1 Functional Programming Languages

The conventional programming languages, such as COBOL, Fortran, Pascal, etc., are characterised by Backus in [1] as "von Neuman" languages, because they are are mainly based on the von Neuman computer model. These languages "use variables to imitate the computer's storage cells, control statements elaborate its jump and test instructions; and assignments imitate its fetching, storing, and arithmetic." [1]

Along with the rapid advancement of hardware and the decrease in its cost, the concurrent executions, which can enormously increase the efficiency of data processing, become not only possible but also necessary. However, the conventional "von Neuman" programming languages impede the progress of concurrent architecture due to their inherent difficulties of partitioning programs to parallel processors.

Moreover, the conventional programming languages are partially responsible for the difficulties in software development and maintenance, due to the difficulties of building up a program using existing subroutines, the lack of mathematical foundations, the complexities, and the limited expressive power.

The functional programming languages, which are suggested by Backus in [1], arise as a new type of programming languages attempting to eliminate some of the drawbacks of conventional languages.

The Berkeley FP [2], is one of a few existing functional programming language implementations closely following [1]. We now use Berkeley FP as the example to illustrate the characteristics of functional programming languages. The specifications of primitive (system-defined) functions in Berkeley FP are listed in Appendix.

FP, unlike conventional programming languages, does not have the concept of "variable." It deals with functions instead of values and has no assignment statements.

The single FP operation, application (:), evaluates an FP function and its argument. For example, +:<1,2> means the application of the function + to its argument <1,2>.

Functional programs consist of the expressions which are a functional-level combination of primitive functions. For example, we write the function returning the sin of the sum of its input, i.e. sin(x+y), as sin@+. This functional expression is the composition of "sin" and "+". The symbol "@" is the compose operator, and "sin" and "+" are the functional arguments of the functional form composition.

Functional forms can take functions or other functional forms as arguments and return functions as their results. Functions may be directly applied to objects, e.g.

sin@+:<1,2>. Functions may also be taken as a functional argument in another functional expression, e.g. cos@sin@+, where sin@+ is taken as functional arguments by the function cos.

Another important feature of FP is to use certain functional forms to avoid the explicit specification of control information. For example, the functional form Apply to All (&f) can be used to apply the functional argument f to all elements in its input. It replaces the loop control statement in the programs written in conventional programming languages.

Certain functional forms in FP also give the potential of parallelism. For example, the functional form Tree Insert (|f), recursively applies f to the two sequences split from the original input until it reaches a null sequence or a sequence which has only one element. The splitting procedure forms a binary tree, and execution starts from the leaves of the tree. If there are multiple processors in the system, the nodes in the same level of the tree can be executed in parallel.

The composition of the functional form Apply to All and the function distr (or distl) also has the potential of parallelism.

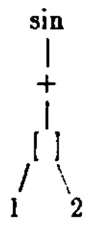
A more detailed discussion of the Berkeley FP language is given in [2]. The style of programming is discussed in [1,9].

In order to execute FP efficiently, we need an architecture, which should require simple translation from FP to executable machine instructions, be able to seperate the accesses to instructions and data as well as have the potentials of concurrent executions.

Because of its tree structures and the absence of variables, we found that FP program can be translated to a string of instruction representation - Q-code, which then can be executed by a machine based on queues - Q-machine. Q-code and Q-machine are discussed in the next section.

1.2 Q-code and Q-machine

The following tree represents the graph of FP expression sin@+@[1,2].



If we traverse the above tree from bottom-up by level-order and from right-to-left by node-order, we get the following string formed by the encountered nodes:

2 1 [ ] + sin

This string form is called the Q-code (Q-notation in [5]), which is the string representaion of the above FP program tree. Q-code was first defined by Z. Pawlak and A.J. Blikle and called by them "cross order." [3,4]

A queue machine organization is proposed by M. Feller in [5,6] to execute Q-notation. The proposed Model S queue machine has an execution unit, a queue, which contains the operands of the next instruction in its front and the result of the last instruction in its end, and a memory module, where the instructions and data are stored. It has the advantage of transparent working store management, no addressable registers, and the potential of concurrent executions.

In this paper we develop a modified version of the queue machine, which can execute the Q-code compiled from Berkely FP programs. It consists of an execution unit and a data queue, which are similar to the ones in the queue machine in [5,6], a instruction queue, which handles the instructions and two seperate memory modules to store data and instructions.

We load the above Q-code from FP program sin@+@[1,2] to the instruction queue, and load the data sequences, which are applied by the two legs of the FP tree, to the data queue. We then connect the front of the data queue and the front of the instruction queue to an execution unit. The execution unit applies the instruction in the first word of the instruction queue to the data in the front of the data queue and enqueues the result to the end of the data queue. The machine organization is shown in Figure 1.1.

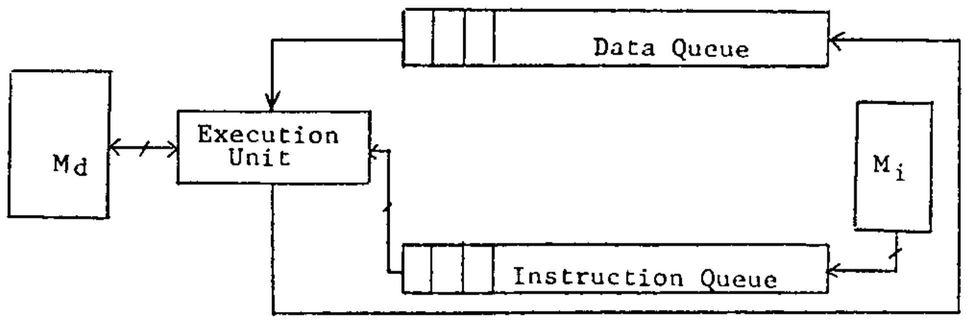


Figure 1.1 Q-machine

We call the above machine organization "Q-machine."

In Chapter 2, we discuss the details of the Q-machine organization and the executions of FP programs on the Q-machine with a single processor. In Chapter 3, we explore the possible parallel executions of FP programs on the Q-machine with multiple processors. In Chapter 4, we report the simulation of the Q-machine and evaluate its performance by running several FP programs. Chapter 5 is the conclusion of this thesis.

CHAPTER 2

The Single Processor Q-machine

In this chapter we develop a queue machine (Q-machine for short) architecture, which can execute FP programs compiled in Q-code, with a single Q-machine processor. We illustrate the Q-machine organization, structure, and function of its components. We also discuss some alternatives and perform comparisons between them.

2.1 System Organization

The Q-machine consists of four main components: Data Queue (D-queue), Instruction Queue (I-queue), Execution Unit (E-unit), and Memory modules (M_i and M_d) -- as Figure 2.1 shows below.

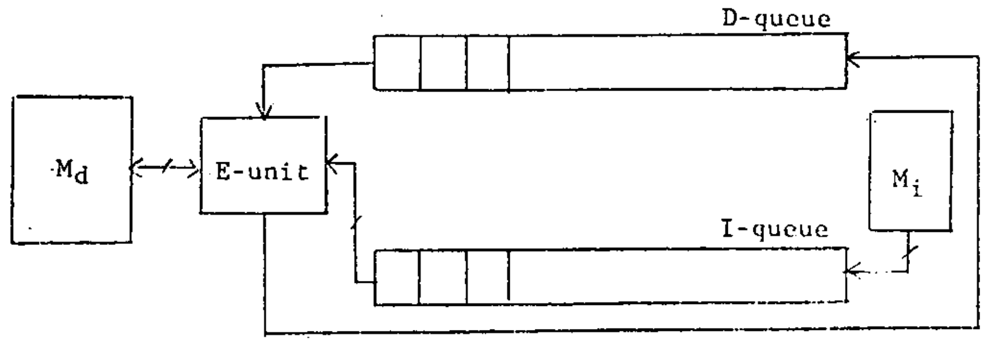


Figure 2.1 Q-machine Organization

The D-queue consists of a sequence of words, each word containing a data representation (we will elaborate its structure in Section 2.2). The front word of the

D-queue contains the representation of the data to be processed by the E-unit during the next machine cycle. The end word of the D-queue contains the data obtained from the E-unit at the last machine cycle. The D-queue performs all queue operations, such as removing the front data from the queue, enqueuing data to the end of the queue, etc. In the beginning of the execution of an FP program, the data applied by the FP program are enqueued to the D-queue (because the D-queue is empty in the beginning of the execution, thus the data are in the front of the D-queue after enqueued). In the end of the FP program execution, the data contained in the front of the D-queue, which are the only data in the D-queue, are the final result. At each machine cycle the result data from the E-unit are enqueued to the D-queue, the front data in the D-queue are fetched to the E-unit, and each word in the D-queue is moved one word ahead. (Except for functional form CONSTRUCTION, the details of which are in Section 2.6).

In the I-queue, each word contains an instruction of Q-code. Introduced in Chapter 1, Q-code is a string consisting of FP instructions in the order of traversing the FP program graph tree from right to left on node-order and and bottom-up on level-order. Each node in the tree, such as +, -, trans, etc., is an FP instruction, which we call Q-instruction. In the beginning of an FP program execution, the Q-instructions of Q-code compiled from the original FP program are enqueud to the I-queue. The front instruction in the I-queue is the one to be executed by the E-unit at the next machine cycle. At each machine cycle, the front Q-instruction in the I-queue is fetched to the E-unit and other Q-instructions in the I-queue are moved one word ahead. The details of I-queue structure will be given in Section 2.3.

The E-unit performs the operations according to the front data in the D-queue and the front Q-instruction in the I-queue. It applies the Q-instruction to the data, accesses and/or allocates memory if needed, and enqueues the execution result data to

D-queue. It halts the entire operation when an END instruction is met in the front of the I-queue. We will discuss the structure and functions of the E-unit in Section 2.4.

The FP program in the Q-instructions with their associated user-defined names is stored in memory module M_p. The data are stored in memory module M_d.

We now give a simple example to illustrate how this system works. Assume the following FP program:

+@[1,2]

Its equivalent tree is shown in Figure 2.2:

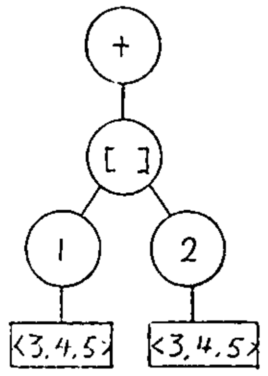


Figure 2.2 Tree Graph of +@[1,2]:⟨3,4,5⟩

In the Q-instructions this program is:

2 1 [ ] +

The data to be applied is sequence <3,4,5>. After the data and the Q-instructions are loaded into the system, the system appears as in Figure 2.3:

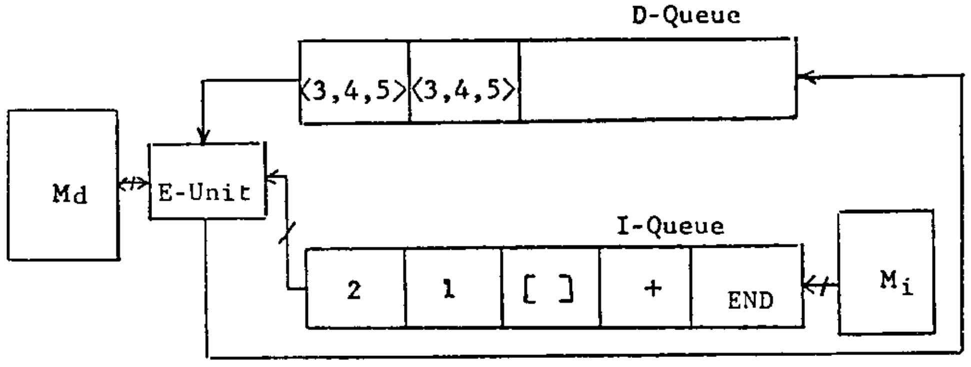


Figure 2.3 Execution of 2:<3,4,5>

At the first machine cycle, function 2 (Select) is applied to <3,4,5> and the result, 4, is enqueued to the D-queue. The state of the system after the first machine cycle is shown in Figure 2.4.

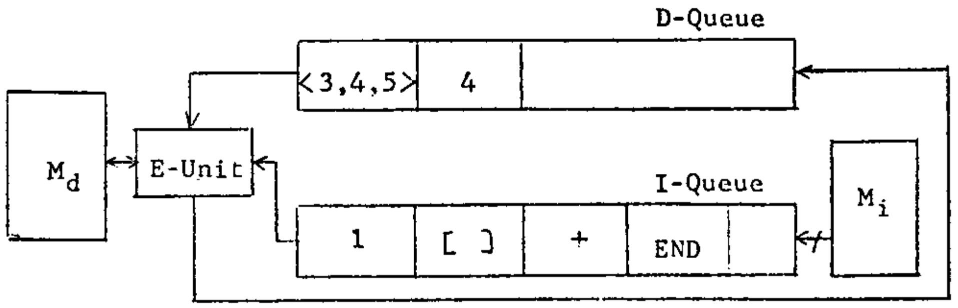


Figure 2.4 Execution of 1:<3,4,5>

At the second machine cycle, function 1 (Select) is applied to <3,4,5> and the result, 3, is enqueued to the D-queue. The system after the second machine cycle is shown in Figure 2.5.

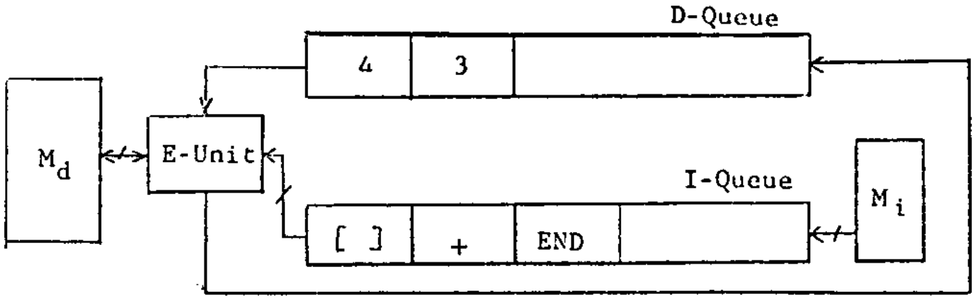


Figure 2.5 Execution of [ ]

At the third machine cycle, functional form "[ ]" (Construct) is applied to the first two words in the D-queue (the number of data elements to be constructed into a sequence is stored in the Q-instruction CONSTRUCTION at compilation time) and the resulting sequence, <3,4>, is enqueued to the D-queue. The system after the third machine cycle is shown in Figure 2.6.

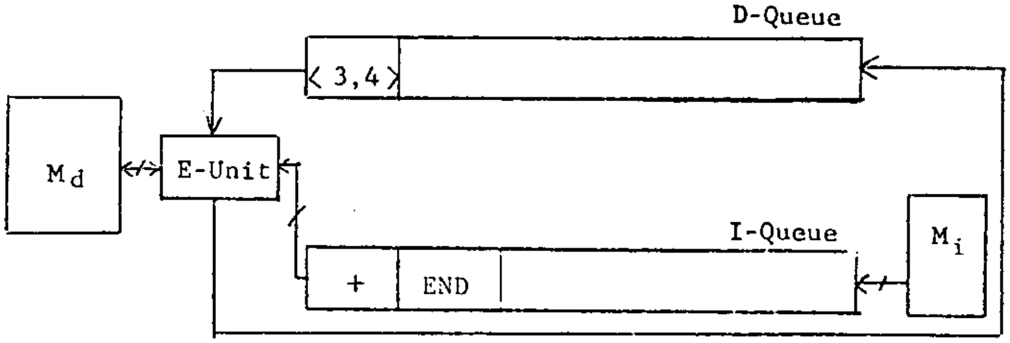


Figure 2.6 Execution of +:<3,4>

At the fourth machine cycle, function + is applied to <3,4> and the result, 7, is enqueued to the D-queue. The system after the fourth cycle is shown in Figure 2.7:

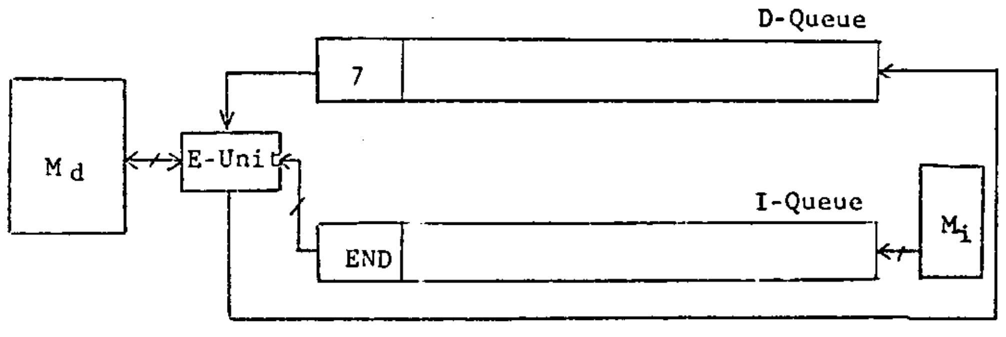


Figure 2.7 Result of +@[1,2]:<3,4,5>

At the fifth machine cycle, the instruction END is encountered in the front of the I-queue and the data, 7, in the front of the D-queue is the final result. The entire operation is finished at this machine cycle.

For the purpose of illustrating the Q-machine operation, the data in the D-queue and the Q-instructions in the I-queue are simplified in this example.

2.2 D-queue Organization and Functions

The D-queue is a very important component in the Q-machine. It holds the data that will be fed to the E-unit and the data generated by the E-unit. Therefore, it is necessary that the D-queue be able to handle various data structures of FP objects and perform necessary data moves. An atom in FP can be either a number (real or integer), a character string, a Bottom (?), an empty set (<>), T(rue) or F(alse). A Sequence in FP is an ordered list of atom(s) and/or sequence(s) enclosed in angle brackets (<>), such as <1,<2,3>> and <<4,5>,<8,9>,<>>.

The word in the D-queue must be able to handle both the atom and sequence, and allow queue operations to be performed quickly. We discuss three alternatives of the data structure for the D-queue word: with fixed length and pointers to the memory; with fixed length and multi-level queue; and with variable length.

2.2.1 D-queue With Fixed Length and Pointers to the Memory

In this structure, each D-queue word has always a fixed length. An atom is directly represented by the word. A sequence is represented by a descriptor.

The atom is represented by the format shown in Figure 2.8.

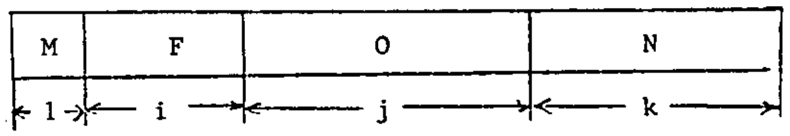


Figure 2.8 Data Format of Atom

M field: 1 bit, is always 0 for atoms;

F field: i bits, indicates the type of atom, such as integer, real, etc.;

O field: j bits, holds the data, such as 1.23, 567, etc.;

N field: k bits, is the pointer to the next object if the atom is an element of a sequence; otherwise it is 0.

Thus, the object of integer atom 789 (not belonging to a sequence) is represented as shown in Figure 2.9.

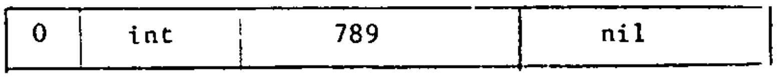


Figure 2.9 Atom 789

A sequence is represented by a sequence descriptor, which has the format as shown in Figure 2.10.

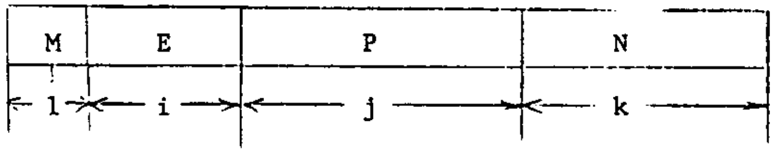


Figure 2.10 Data Format of Sequence Descriptor

M field: 1 bit, is always 1 for sequences;

E field: i bits, is an integer number which indicates the number of elements in the sequence it represents;

P field: j bits, is the pointer to the first element in the sequence it represents;

N field: k bits, is the pointer to the next object if the represented sequence itself is an element of another sequence; otherwise it is 0.

The sequence <10,<20,30>> is represented by the descriptor shown in Figure 2.11.

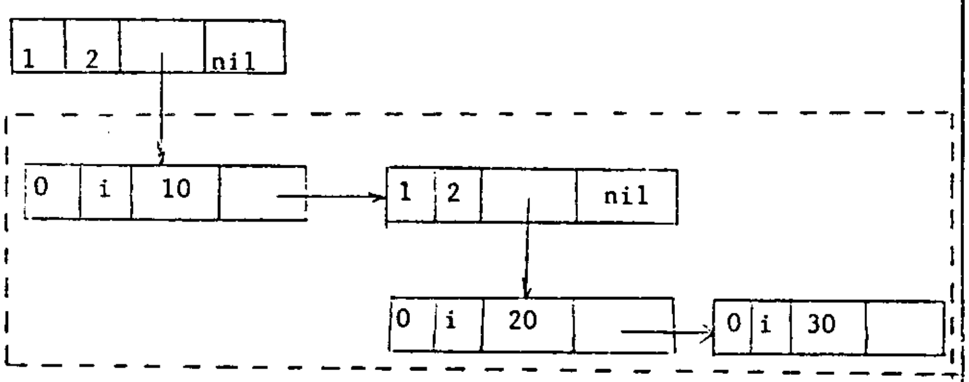


Figure 2.11 Data Structure of Sequence ⟨10,⟨20,30⟩⟩

With the above structure, if an object is an atom, it will be directly enqueued to the D-queue; if an object is a sequence, only its descriptor will be enqueued to the D-queue. All its elements (surrounded by dashed lines in Figure 2.11) will be stored in the memory and fetched by the E-unit when needed. Figure 2.12 illustrates the relationship between these components.

D-queue can be implemented in a group of fast registers. There are two methods for generating the movement of a queue word.

In the first method, all queue words are shifted ahead one word as Figure 2.13 shows.

This method is simple, but involves considerable data movements if the length of the D-queue is relatively long.

In the second method, two indicators are used to indicate the front and end of the D-queue respectively. Therefore only the indicators are advanced to implement the movement of queue words. Figure 2.14 shows the second method.

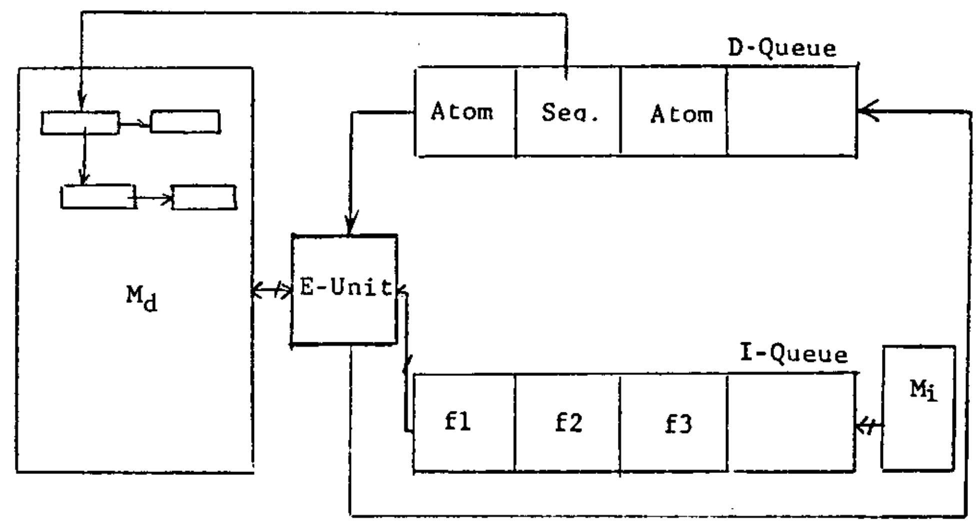


Figure 2.12 Sequence Elements Stored in the Memory

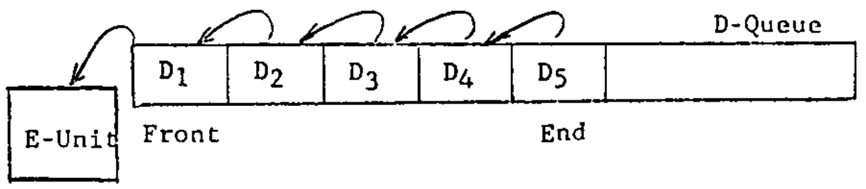


Figure 2.13 Data Movement in D-queue

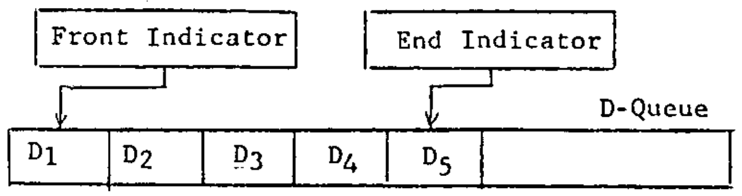


Figure 2.14 Data Movement Using Indicators

This method eliminates data movements inside the D-queue. However, it needs a circular register group to hold the data.

The strong point of the above D-queue organization is its addressless operation, implicit register allocation and deallocation, as well as simplicity and ease of implementation. The weak point is its use of the random-access memory to handle sequences and need of garbage collection.

In order to estimate the access time to the memory, we define two different levels of memory access. One is called Vertical Access, which locates the first element of a sequence according to the pointer stored in the P field in the descriptor of the sequence. The other is called Horizontal Access, which locates the ith (i>1) element in a sequence according to the pointer stored in the N field in the representation word of the (i-1)th element in the same sequence.

For example, in Figure 2.15, to get atom 10 in the sequence <5 <6 7 10>>, according to its descriptor, requires 2 vertical accesses (VA_1 and VA_2) and 3 horizontal accesses (HA_1, HA_2 and HA_3).

Therefore, to locate an element of a sequence requires

$$n + \sum_{i=1}^{n} D_i$$

access time. Here n is the nesting level at which the element to be located resides, and D_i is the number of horizontal accesses in level i.

For the example in Figure 2.15, n is 2, D_1 is 1, and D_2 is 2. The total access time is $5 = 2+1+2$.

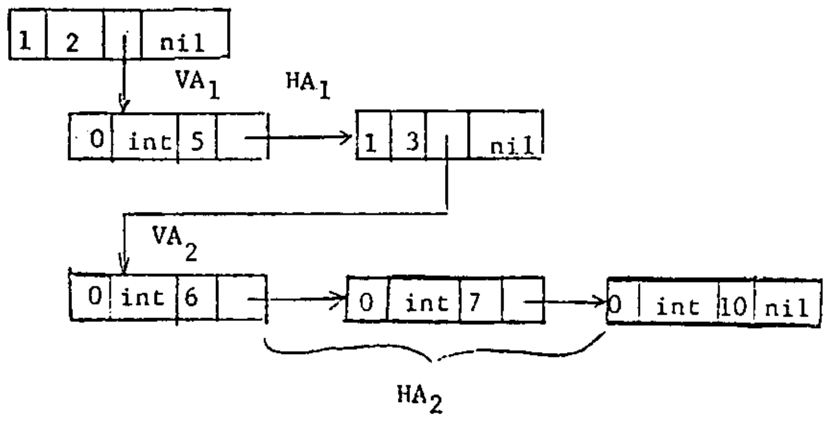


Figure 2.15 Access Path of Atom 10

A suitable garbage collection scheme for the Q-machine using the above D-queue organization is needed. We discuss how the scheme of reference count [11] can be used.

The data which are valuable to the system, and therefore need to be kept in the memory, are the data which are referred by the descriptors currently residing in the D-queue. The memory cells which hold the data no longer referred by any descriptor in the D-queue can be freed for future use. We use an integer tag field in each memory cell to indicate the number of references that are currently pointing to this cell. This tag field is called reference counter. If the reference counter of a cell is 0, there is no reference pointing to this cell and it can be freed.

The algorithm of the reference counting is:

1. Before an object is enqueued to the D-queue, it is checked whether it is an atom. If so, nothing has to be done. If it is a sequence descriptor, all the reference counters of the cells that are referred by this descriptor and/or the descriptor of its descendents are increased by 1.

2. After an object is fetched from the D-queue, if it is a sequence descriptor, all the reference counters of the memory cells that are referred by this descriptor and/or the descriptor(s) of its descendents are decreased by 1.

3. The cell whose reference counter is 0 can be collected for use. The collection can be carried out at any time except during the period that the reference counters of the fetched data have been decreased but have not been increased by the E-unit, if such increase is necessary.

We now use the example of applying FP program +@[1,2] to sequence <4,5,6> to illustrate how this garbage collection scheme works.

The sequence <4,5,6> and its descriptor are shown in Figure 2.16.

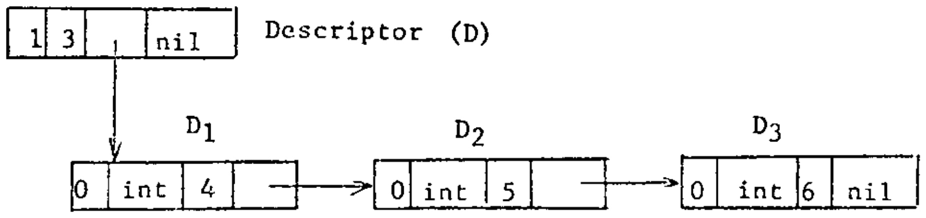


Figure 2.16 Sequence ⟨4,5,6⟩

After the descriptor of the sequence is enqueued to the D-queue twice (because the sequence will be applied by both select functions 1 and 2), and the Q-instructions are loaded to the I-queue, the state of the system is shown in Figure 2.17. Notice that the reference counters (the leftmost field) in the memory cells which hold atoms 4, 5 and 6 are all 2's.

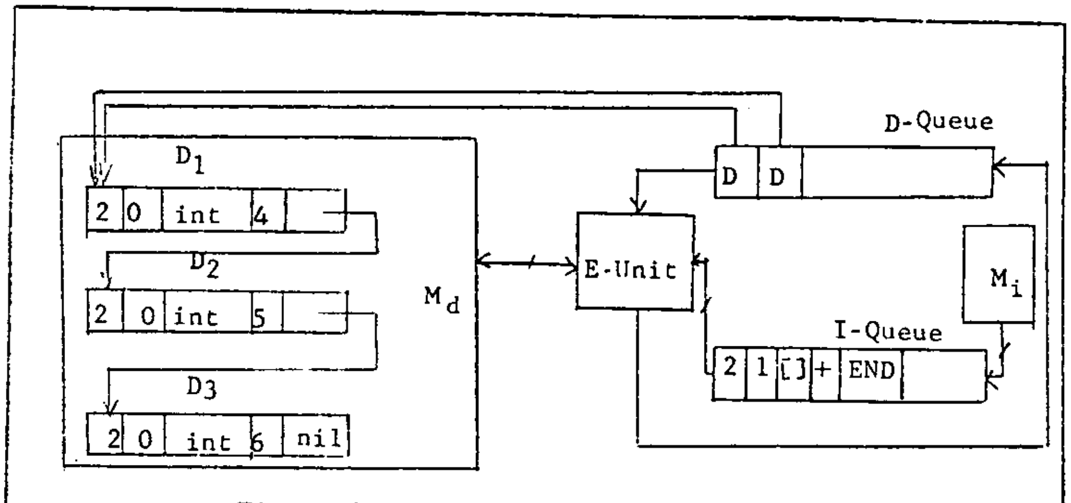


Figure 2.17 Reference Counters of <4,5,6>

During the application of function 2 (Select) to <4,5,6>, the descriptor in the front of the D-queue is fetched to the E-unit, and the reference counters of atoms 4, 5 and 6 are all decreased by 1. The result of 2:<4,5,6> is atom 5, and it is directly enqueued to the D-queue. The state of the system is shown in Figure 2.18.

After the application of function 1 (Select) to <4,5,6>, the reference counters of the cells holding atoms 4, 5 and 6 are reduced to 0s as Figure 2.19 shows. Hence these 3 cells can be freed.

After the application of Function "[ ]" (Construction) to atoms 4 and 5, a new sequence <4,5> is formed. The descriptor of <4,5> is enqueued to the D-queue. Atoms 4 and 5 are stored in the memory and each has a reference counter of 1. Figure 2.20 shows the state of the system after applying the Construction.

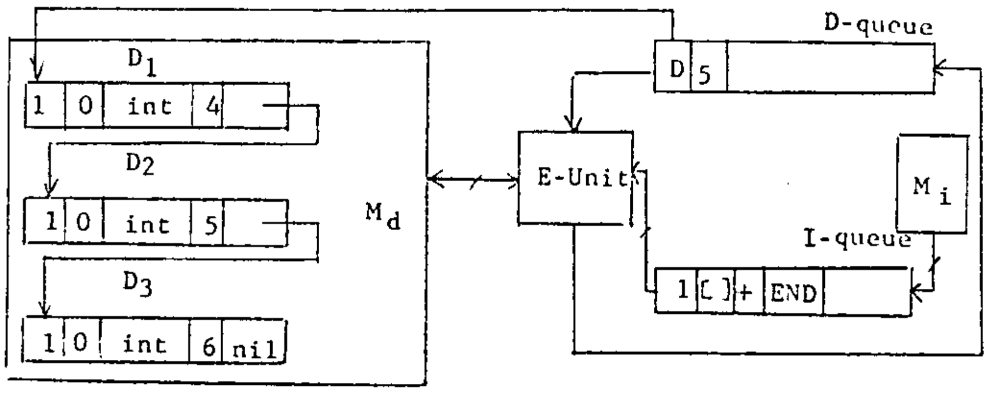


Figure 2.18 Reference Counters of ⟨4,5,6⟩

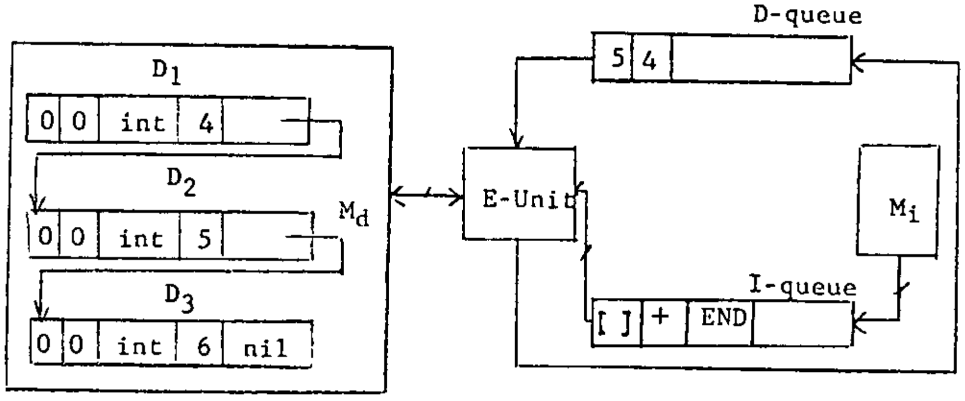


Figure 2.19 Reference Counters of ⟨4,5,6⟩

During the application of function + to <4,5>, the reference counters of 4 and 5 in the memory are decreased to 0. The cells holding them can thus be freed. The result of +:<4,5>, 9, is enqueued to the D-queue and the operation halts due to the END in the front of the I-queue. The state of the system is shown in Figure 2.21.

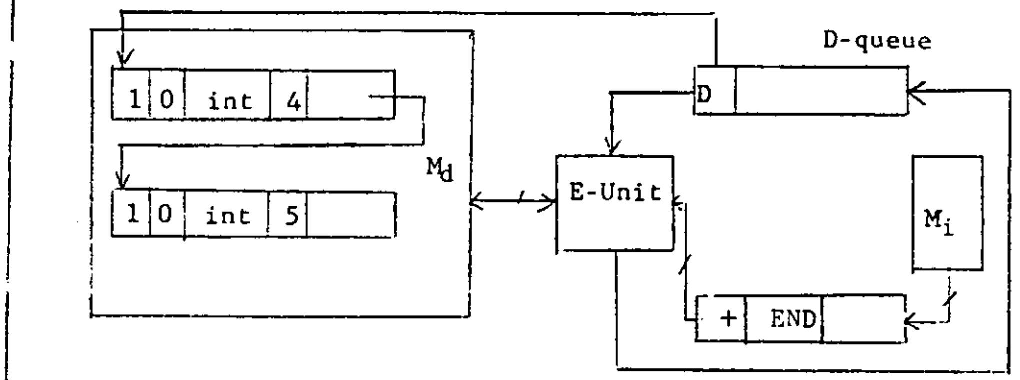


Figure 2.20 Reference Counters of <4,5>

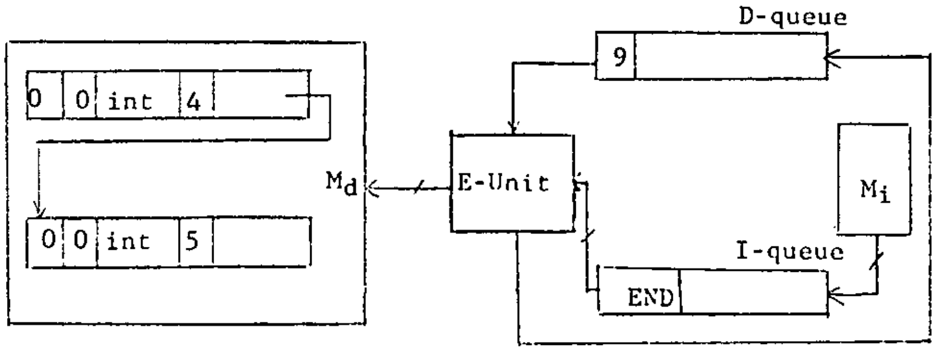


Figure 2.21 Reference Counters of <4,5>

2.2.2 D-queue With Fixed Length and Multi-level Queue

Here we consider an alternative D-queue organization with a fixed length and multiple queues, which we call the multi-level queue.

An object can be represented by a tree. A node in the representing tree could be an atom or a sequence descriptor. The level of the root of the tree is 1, and the level of a node is the sum of 1 and the distance between the node and the root. For example, the sequence, <10,<30,40> >, is the following tree:

```
   < >          level 1
  /   \
10    < >        level 2
      /  \
    30   40     level 3
```

If we put the objects (either an atom or the descriptor of a sequence) on the same level of the tree to the corresponding queue with the same level, we will have H parallel queues forming the D-queue. (H is the height of the object tree; e.g. for the above object <10,<30, 40>>, H is 3).

With this structure, an atom is represented by the format in Figure 2.22:

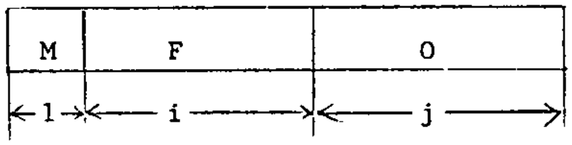


Figure 2.22 Data Format of Atom

M field: 1 bit, is always 0 for atoms;

F field: i bits, indicates the type of atom;

O field: j bits, is the data itself.

A sequence descriptor is represented by the format shown in Figure 2.23.

M field: 1 bit, is always 1 for sequence;

E field: k bits, is the position of the beginning element in the sequence which this descriptor represents in the queue at the next level;

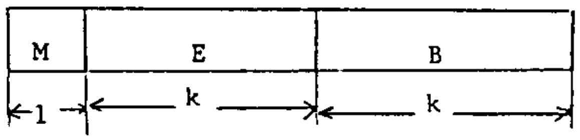


Figure 2.23 Data Format of Sequence Descriptor

B field: k bits, the position of the ending element in the sequence which this descriptor represents in the queue at the next level.

The sequence <10,<30, 40>> is stored in the following format in the D-queue with a multi-level queue as Figure 2.24 shows.

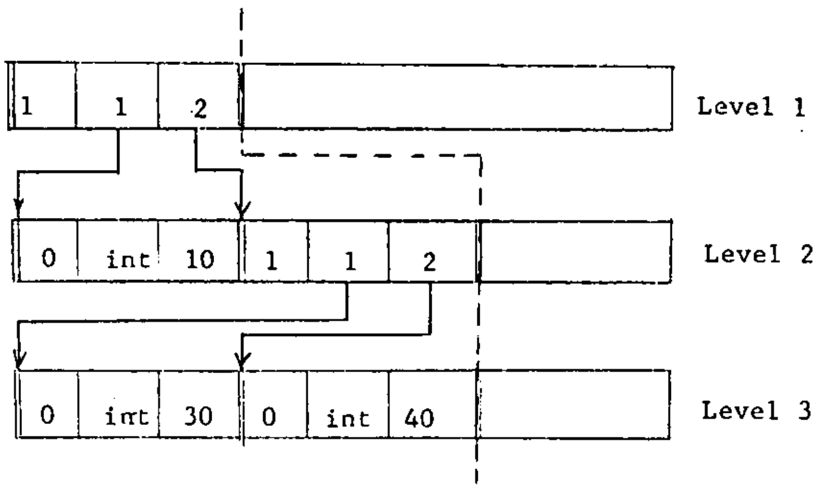


Figure 2.24 <10,<30,40>> Stored in Multi-level Queue

When data are fetched to, or enqueued from, the E-unit, the words to the left of the dashed lines will be fetched or enqueued together.

There are also two methods for implementing the data movement in the multi-level queue structure: 1) moving the data, or 2) advancing the indicators. However, multi-level indicators are needed to indicate the different fronts and ends in the multi-level queue.

This organization is good for eliminating memory access and avoiding garbage collection. However it has two main drawbacks: 1) frequent update of position pointers, and 2) many data moves.

Because the descriptor of a sequence on a certain level contains the front position and the end position pointers to its elements in the next level, whenever the positions of its elements in the next level change due to the queue operations, the position pointers in the descriptor of the sequence must be changed as well. For example, the D-queue containing sequences <10,<30,40>> and <<5,6>,8> is shown in Figure 2.25.

In Figure 2.25, the data on the left side of the dashed line are <10,<30,40>> and the data on the right side are <<5,6>,8>.

If the data in the front of the D-queue, sequence <10,<30,40>>, are removed from the D-queue due to queue operation, the data next to the front, sequence <<5,6>,8>, would be moved to the front. Meanwhile, the position of the descriptor of sequence <<5,6>,8> at level 1 is changed from 2 to 1, the position of descriptor of sequence <5,6> at level 2 is changed from 3 to 1, the position of atom 8 at level 2 is changed from 4 to 2, and the positions of atom 5 and 6 at level 3 are changed from 3 and 4 to 1 and 2 respectively. All the position pointers stored in the descriptors at the upper level have to be updated correspondingly as well. The D-queue after the removal of the front data, <10,<30,40>>, is shown in Figure 2.26. Notice all the changes in

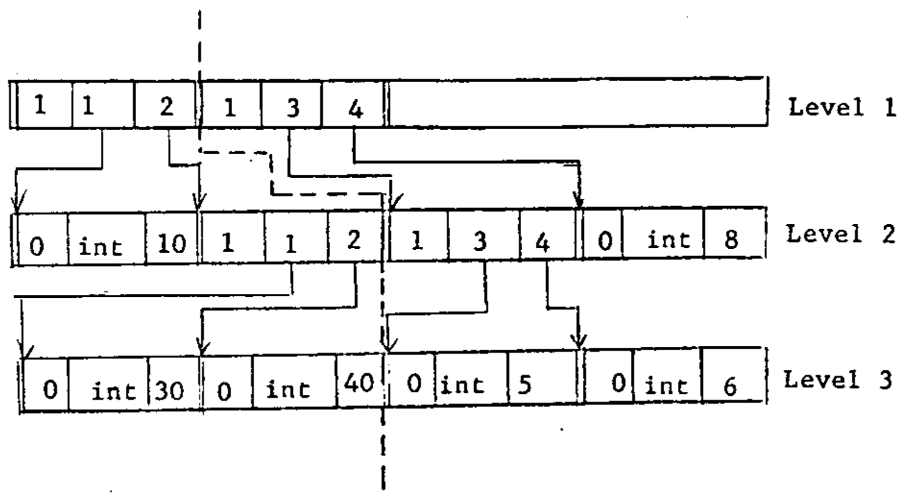


Figure 2.25 ⟨10,⟨30,40⟩⟩ and ⟨⟨5,6⟩,8⟩Stored in Multi-level Queue

the position pointer areas in the data representation of sequence <<5,6>,8>.

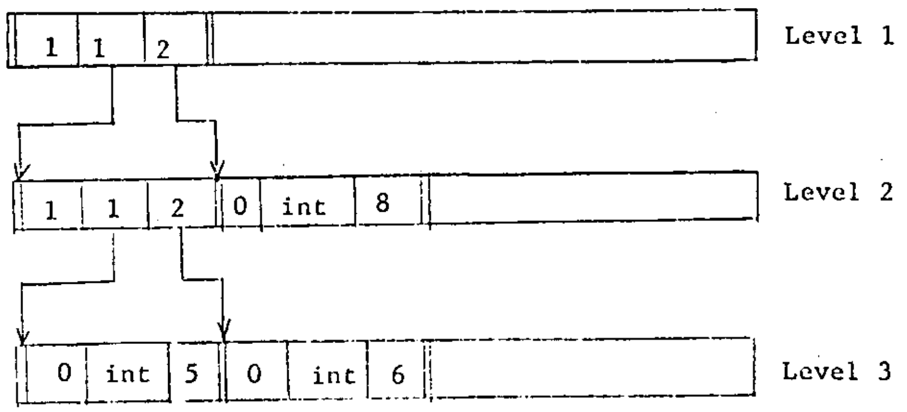


Figure 2.26 Changes of Position Pointers

The second drawback of this multi-level queue organization is that all data at all levels must be moved during queue operations. For example, in the queue organization of a fixed length and pointers to the memory, which is described in section 2.2.1, applying select function 2 to the sequence <10,<30,40>>, we need only to enqueue the descriptor of the resulting sequence <30,40> to the D-queue without moving atom 30 and 40 in the memory. On the other hand, if the multi-level queue organization is used, not only the descriptor of the sequence <30,40>, but also atoms 30 and 40 need to be enqueued to the D-queue. If the data to be applied have many nesting levels, the number of all data moves at all levels will be fairly considerable.

2.2.3 D-queue With Variable Length

The third structure option of the D-queue is to store all atoms and sequences in one level, but to use different delimiters to separate the words of the D-queue and to separate the elements of the sequence.

An atom is represented by the following format shown in Figure 2.27.

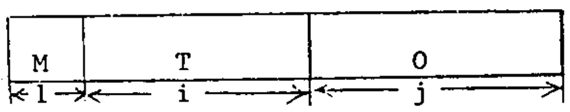


Figure 2.27 Data Format of Atom

M field: 1 bit, is always 0 for atoms;

T field: i bits, indicates the atom type;

O field: j bits, contains the data.

A delimiter is represented by the format shown in Figure 2.28.

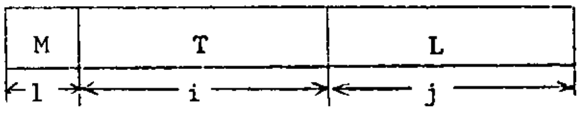


Figure 2.28 Data Format of Delimiter

M field: 1 bit, is always 1 for delimiter;

T field: i bits, indicates the delimiter type; such as '<', '>', ',', etc.

L field: j bits, is the level number of the delimiter.

Sequence <10,<20,<30,40>,50>> can be stored in the D-queue as Figure 2.29 shows.

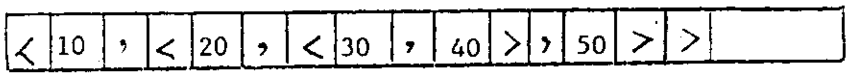


Figure 2.29 Sequence <10,<20,<30, 40>,50>>

This D-queue organization also eliminates the memory access and avoids garbage collection. However, the distance of data movement in this organization depends on the length of the first data in the front. Therefore, during the queue operation of removing the front data, we have to search the boundary of the front data. Further-more, having a sequence fetched from the D-queue, the E-unit has to search and/or separate the elements according to the delimiters. This search and/or separation will be a time consuming sequential operation.

In summary, each of the above three D-queue structures has its strong points and weak points.

For the D-queue word with a fixed length and pointers to the memory, the strong point is its simplicity and ease of implementation. The weak point is its access to memory and need of garbage collection.

The D-queue word with a fixed length and multi-queues is good for eliminating memory access and avoiding garbage collection. However, it involves more data moves during queue operations and all position pointers must be updated after every queue operation.

The D-queue word with a variable length does not have the problem of memory access and garbage collection, but because searching for the boundary of the front word of the D-queue and separating the elements of a sequence must be done at run-time, it considerably degrades the system performance.

The following table gives the comparisons of memory allocation and data movement between the above three D-queue word structures during a single machine cycle. Assume, during the machine cycle, a select function k is applied to a sequence, $<x_1, x_2, ..., x_i, ..., x_m>$, where the elements of sequence x_i ($m \geq i > 0$) are atoms only and N_i is the number of elements in x_i. I represents the D-queue word structure with fixed length and pointers to memory, II represents the structure with fixed length and multi-level queues, and III represents the structure with variable length.

Compared Item	I	II	III
Memory Words Required	$m+\sum_{i=1}^{m} N_i$	0	0
Registers Required	1	$m+1+\sum_{i=1}^{m} N_i$	$m+1+\sum_{i=1}^{m}(2N_i+1)$
D-queue Words Discarded	1	$m+1+\sum_{i=1}^{m} N_i$	$m+1+\sum_{i=1}^{m}(2N_i+1)$
D-queue Words Enqueued	1	$1+N_k$	$2N_k+1$
Memory Accesses	k	0	0
Garbage Collection Needed	Yes	No	No
Boundary Searching Needed	No	No	Yes

From now on, we use the D-queue word with a fixed length and pointers to the memory as the D-queue organization in this paper. This organization needs less data movement than the multi-level queue. It also avoids expensive boundary searching and element separation at run-time, which are needed in the organization with variable length.

2.3 The Organization of the I-queue

The I-queue consists of i registers, each of them being m bits long.

Figure 2.30 shows the format of an I-queue word.

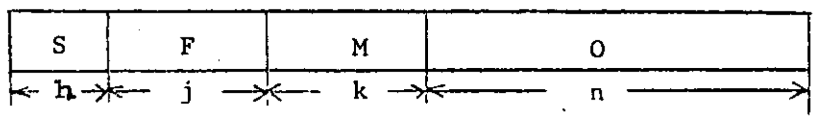


Figure 2.30 Instruction Word Format

S field: h bits, indicates the type of the FP functional form, such as Regular, Right Insert, Tree Insert, Apply to All, While, Condition, Construction and Constant;

F field: j bits, represents FP function, such as +, -, apndr, trans, etc;

M field: k bits, is the number of times required to enqueue the result from the E-unit to the D-queue;

O field: n bits, is used for different purposes with different FP functions. For the functional form of Constant (%number), it stores the constant number; for the functional form Construct ([ ]), it stores the number of elements to be constructed in a sequence; and for a user-defined function, it stores the name string of the user-defined function.

The I-queue connects to the memory and the E-unit. It is controlled by the E-unit. Accessing I-queue is faster than accessing the memory.

Before the execution of an FP program starts, the Q-instructions of the FP program are stored in the main memory. In the beginning of the execution, the first i Q-instructions of the invoked FP program are enqueued to the I-queue, if the number of Q-instructions of the invoked program is greater than or equal to i. If the number of Q-instructions is less than i, all the Q-instructions are enqueued to the I-queue. At each machine cycle, the E-unit fetches the first Q-instruction in the I-queue and applies it to the data in the front of the D-queue. The first Q-instruction in the I-queue is discarded after being fetched to the E-unit. The rest of the Q-instructions in the I-queue are each moved one word ahead. When the memory access is available and more Q-instructions in the memory need to be applied, the I-queue filling process begins.

In the execution of programs in conventional languages on conventional machines, the data and the program code are closely coupled through address references after they are loaded to the memory. Fetching data and fetching instructions share the same memory access, which is the bottle-neck.

In the execution of FP programs in the Q-machine, an advantage is that the address of Q-instructions in the momery and the address of the data in the memory are totally independent of each other after they are loaded into the main memory. Therefore, we have the freedom to have the Q-instructions of an FP program loaded into the same memory module where the data are loaded, or into a totally separate memory module.

In a case where the data and the Q-instructions share a single memory module, the I-queue filling process can be carried out when the E-unit is not accessing data from, or storing data to, the memory module.

In a case where the Q-instructions are loaded to a separate memory module, which has its own access path, the I-queue filling process can be carried out whenever it is needed.

In the rest of the paper, we use an independent memory module to store the Q-instructios of an FP program.

2.4 The Organization of the E-unit

The Execution Unit applies the function instruction (the front of the I-queue) to the data (the front unit of the D-queue), then enqueues the result to the end of the D-queue.

The E-unit consists of three main components: Pre-processor, Main-processor, and cache memory units. Figure 2.31 shows the structure of the E-unit.

At each machine cycle, the E-unit carries out the following operations:

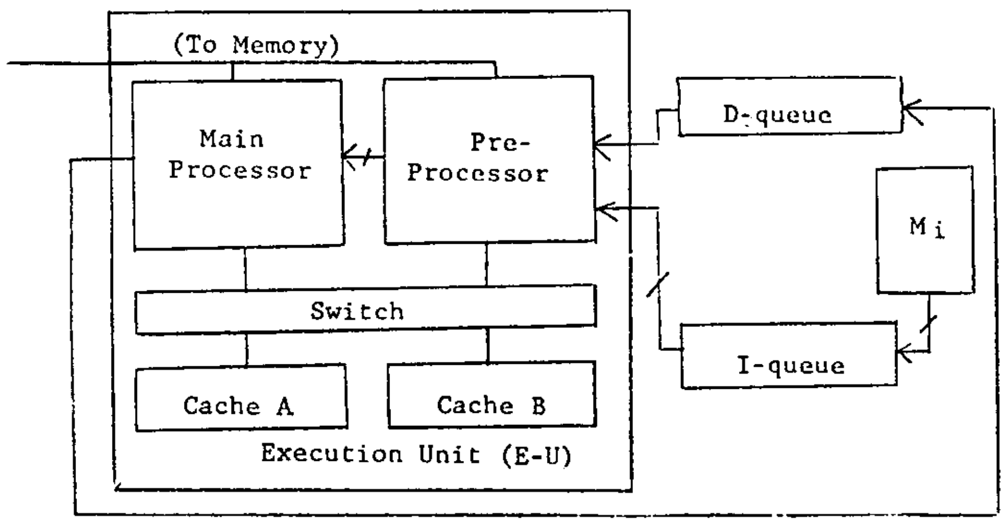


Figure 2.31 E-unit Organization

1. It fetches data from the front word of the D-queue.

2. It fetches the next Q-instruction from the I-queue.

3. If the data from the front of the D-queue is the descriptor of a sequence, and the Q-instruction from I-queue operates on the elements of the sequence, the E-unit fetches the elements of the sequence from the memory.

4. If the data from the front of the D-queue is the descriptor of a sequence, the E-unit decreases the reference count of each element and their descendants of the sequence.

5. It applies the Q-instruction to the data.

6. If the result is a newly-formed sequence, the E-unit stores its element(s) to the memory.

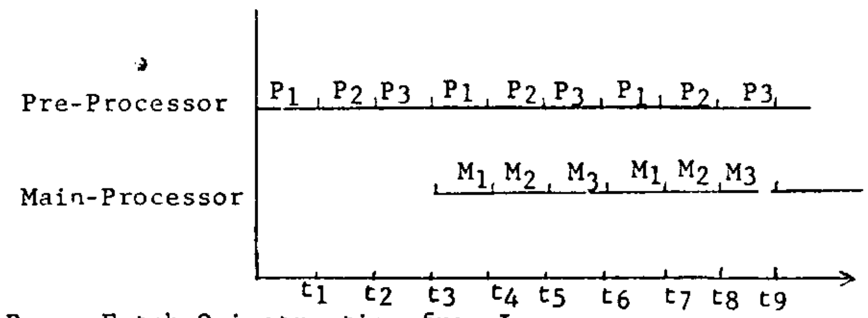
7. If the resulting data is a sequence, the E-unit increases the reference count of each element and their descendants of the sequence.

8. It enqueues the result to the D-queue.

In order to make it possible for the operations to overlap, hence reducing the time for each machine cycle, the above operations are divided into two groups: the first group includes operations 1-4, the second group includes operations 5-8. The Pre-processor carries out operation 1-4 and the Main-processor carries out operation 5-8. The Pre-processor and the Main-processor are pipelined. When Main-processor is carrying out operations 5-8 of the ith Q-instruction, the Pre-processor is carrying out operations 1-4 of the (i+1)th Q-instruction. The time cycle required for the Pre-processor or Main-processor to complete an execution is called an E-unit cycle. The communications between the Pre-processor and the Main-processor are through the buffer (cache A and cache B) via a switch. Figure 2.32. is the time diagram of the E-unit cycle.

2.4.1 The Memory

There are two cache memory units in the E-unit. At any moment, one of them is connected to the Pre-processor and the other is connected to the Main-processor by the switch. After one E-unit cycle is completed, the two cache memory units switch the connections to the two processors. The one connected to the Pre-processor will be connected to the Main-processor, so that the Main-processor can apply the function instruction to the data prepared by the Pre-processor. The one connected to the Main-processor will be connected to the Pre-processor, so that the Pre-processor can have available space to store the data being fetched in the next E-unit cycle.



P_1 -- Fetch Q-instruction from I-queue
P_2 -- Fetch data from D-queue
P_3 -- Fetch data from memory, if needed
M_1 -- Execute the Q-instruction
M_2 -- Store data to memory, if needed
M_3 -- Enqueue resulting data to D-queue

Figure 2.32 Time Diagram

2.4.2 The Pre-processor of the E-unit

The basic task of the Pre-processor is to fetch the front Q-instruction from the I-queue and the front data from the D-queue, as well as to have the data ready for execution in the Main-processor. The Pre-processor is connected to the D-queue, the I-queue, the memory and one of the two cache memory units at any moment.

For example, the current front in the I-queue is the Q-instruction "+", and the current front of the D-queue is the descriptor of sequence <3,4>.

The Pre-processor fetches the front Q-instruction from the I-queue and stores it in a special location in the cache memory to which it is currently connected. Meanwhile, the Pre-processor fetches the two elements of the sequence, according to its descriptor in the front of the D-queue, from the main memory and puts them in one of the cache memories that is currently connected to the Pre-processor, so that the Main-processor can apply the Q-instruction "+" to them at the next E-unit cycle.

If there is no data available in the D-queue (i.e. D-queue is empty) for the next instruction, the Pre-processor will wait for the result from the Main-processor. In this case, the result of the current execution in the Main-processor will remain in the cache and the Pre-processor will fetch it from there in the next cycle instead of fetching from the D-queue.

The Pre-processor can access memory, but does not have the capability of allocating memory nor of writing data to memory.

2.4.3 The Main-processor of the E-unit

The task of the Main-processor is to apply the current function to the data prepared by the Pre-processor, allocate memory if needed, and enqueue the result to the D-queue.

The Main-processor is connected to the D-queue, the memory and one of the two cache memory units. It does not have a connection with the I-queue and only takes Q-instruction from the Pre-processor through one of the two cache memory units.

We continue using the example set forth in Section 2.4.2 above. After the E-unit cycle, during which Pre-processor fetches function instruction "+", elements 3 and 4, and stores them in cache unit A, the Main-processor is connected to cache unit A. From cache unit A, the Main-processor takes the function instruction "+", as well as operands 3 and 4, and processes the addition to get the result 7. Because 7 is an atom, there is no need to allocate memory, nor to form sequence. Therefore, the Main-processor just enqueues the result to the D-queue.

The Main-processor can access memory, allocate memory and write data to memory as well.

The Main-processor is required only to be able to execute these system-defined functions. (The execution of user-defined functions is actually the executions of a group of system-defined functions, the details of which will be given in Section 2.6.2). Hence, the Main-processor can be implemented in hardware or firmware, which will make the execution of functions more efficient.

2.5 The Stack of Queues

In the above description of the Q-machine, the D-queue and I-queue are one-dimensional queues. In this organization of the Q-machine, the FP programs which call user-defined functions cannot be executed efficiently. In order to solve this problem, we develop a two-dimensional queue structure, or Stack of Queues (S-Q), to deal with user-defined function calls and certain functional forms. The structure of the Stack of Queues is shown in Figure 2.33.

When the execution of an invoked user-defined FP function begins, a new I-queue which contains the Q-code instructions of that user-defined function is pushed to the top of the I-queue stack (In the very beginning of the execution, the I-queue stack is empty). Meanwhile, a new D-queue whose front unit contains the data to be applied by that user-defined function is pushed to the top of the D-queue stack. (As with the I-queue stack, the D-queue stack is empty in the very beginning). During the execution, only the tops of the I-queue and D-queue stacks are connected to the E-unit. After the execution of the user-defined function is finished, the top I-queue and top D-queue will be popped off from the I-queue stack and the D-queue stack, respectively. The E-unit will then be connected to the new top of the I-queue stack and the new top

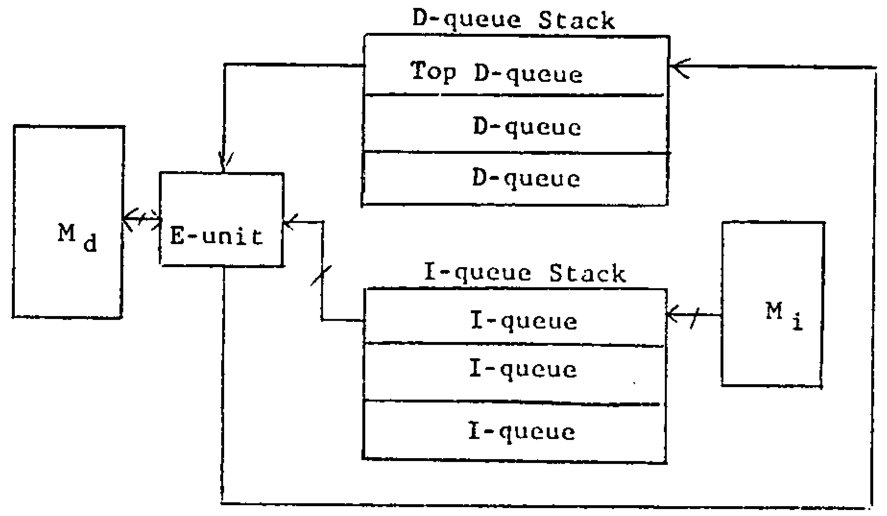


Figure 2.33 Queue Stacks

of the D-queue stack.

2.6 Execution of FP Programs on the Q-machine

In this section, we discuss the execution of Berkeley FP programs on the Q-machine including the execution of regular FP system-defined functions, the execution of user-defined functions, and the execution of functional forms.

2.6.1 Execution of FP System-Defined Functions

Berkeley FP has 39 system-defined functions, which are listed in Appendix.

Here we use a function "select" as an example to show how a FP system-defined function is executed on the Q-machine.

A select function is defined as follows:

$\mu : x \equiv$
$x = <x_1, x_2, \dots x_k> \wedge 0 < \mu \leq k \rightarrow x_\mu;$

$x = <x_1, x_2, \ldots x_k> \wedge \text{-k} \leq \mu < 0 \rightarrow x_{k+\mu+1};$?

A select function is represented by an integer. For example, a single 2 in FP represents selecting the second element from the applied data sequence. So if function "2" is applied to sequence <10,20,30,40>, the result will be object 20.

When the Pre-processor encounters a select function in the front of the I-queue, it checks first whether the object in the front of the D-queue is a sequence. If the object is not a sequence, the Pre-processor will put a "?" (bottom) to cache. If the object is a sequence and the select number is within the range of the elements in that sequence, the Pre-processor will fetch the element from the memory to the cache memory connected to it. In the above select function 2:<10,20,30,40>, 20 will be stored in a cache.

At the next E-unit cycle, the Main-processor is connected to the cache that was connected to the Pre-processor during the last E-unit cycle and takes the select instruction passed from the Pre-processor. Because the selected element is already in the cache, the Main-processor simply enqueues the selected element to the D-queue.

2.6.2 Execution of User-Defined Functions

The FP allows a user to define a function which consists of a user-given name and FP expressions.

For example, a function called MEAN can be defined by a user to compute the average of the number in an input sequence. Function MEAN is defined as follows:

{MEAN /@[!+,length]}

The curly brackets are delimiters of user-defined functions.

A user-defined function can be directly applied by a user or called by other user-defined functions in the same program. For example, a function named DEVIATION is defined by the user to compute the deviations of each element in the applied sequence from their mean. The function DEVIATION is defined as follows:

{DEVIATION &-@distl@[MEAN,id]}

It calls MEAN to get the mean of elements in an applied sequence, then distributes the mean to each element and applies - (subtraction) to all of them.

In order to execute these user-defined functions, their compiled Q-instructions have to be loaded to the memory in the Q-machine first. Figure 2.34 shows the Q-instructions of DEVIATION and MEAN.

Function Name	Q-instructions							
MEAN	id	length	!+	[ ]	/	END		
DEVIATION	id	id	MEAN	[ ]	distl	&-	END	

Figure 2.34 Q-instructions of MEAN, DEVIATION

When DEVIATION is applied by a user to a sequence, e.g. <4.0,5.0,6.0>, the Q-code of DEVIATION are pushed to the top of the I-queue stack, and the sequence <4.0,5.0,6.0> is pushed to the top of the D-queue stack. The system is shown in Figure 2.35.

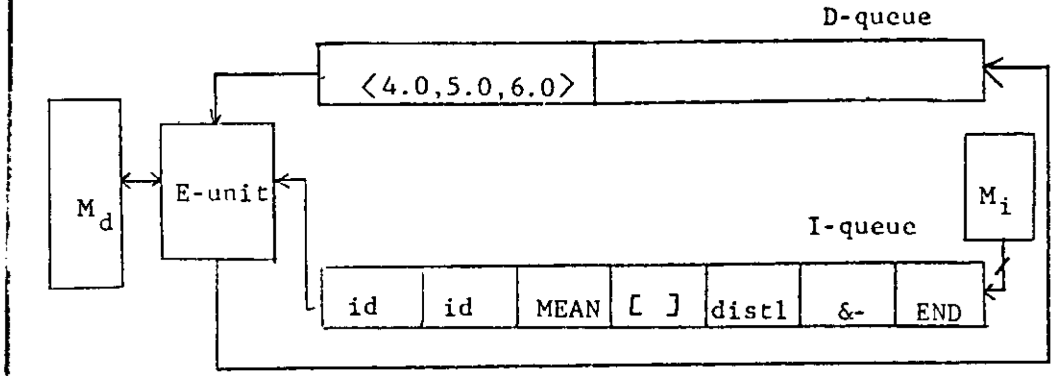


Figure 2.35 Execution of DEVIATION

The system as it appears after the first two machine cycles is shown in Figure 2.36.

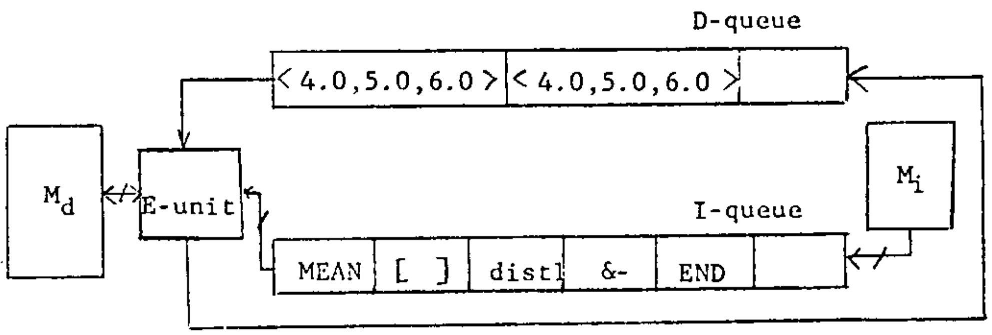


Figure 2.36 MEAN Encountered

From Figure 2.36, we can see that the E-unit encounters the user-defined function MEAN in the front of the I-queue and the sequence <4.0,5.0,6.0> in the front of the D-queue. Then the E-unit searches for the Q-instruction which is associated with the name "MEAN". After the Q-instruction of "MEAN" is located, it is pushed to the top of the I-queue stack. Meanwhile, the object sequence <4.0,5.0,6.0> is pushed to the top of the D-queue stack. The snapshot of the system at this moment is shown in Figure 2.37.

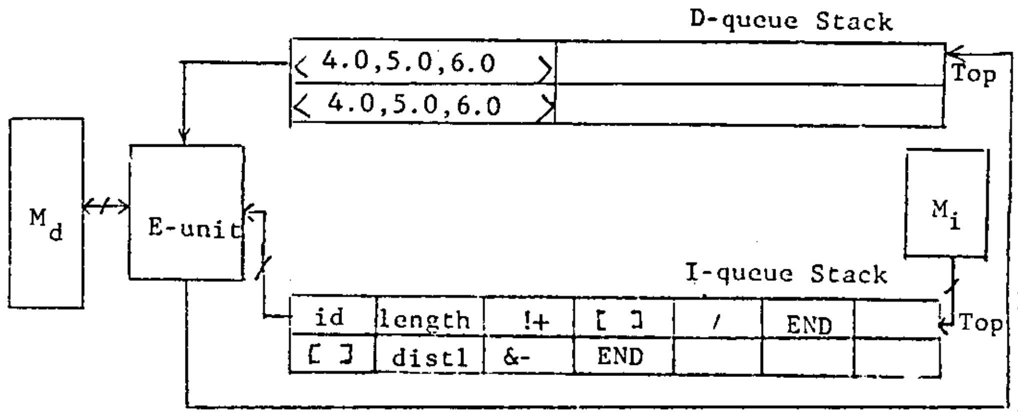


Figure 2.37 Execution of MEAN

In Figure 2.37, the top of the I-queue stack is the current active I-queue and the top of the D-queue stack is the current active D-queue. The E-unit is connected to the D-queue as well as the I-queue, and begins the execution.

After 5 machine cycles, the front of the active I-queue is END and the front of the active D-queue is the result object 5.0, which is the mean of 4.0, 5.0 and 6.0. The E-unit sees the END in the front of the I-queue, and keeps the result, 5.0, in its cache. The top of the I-queue stack and the top of the D-queue stack are popped off. The system snapshot afterward is shown in Figure 2.38.

The state of the system returns to the state that existed when the user-defined function MEAN was invoked. Then the new tops of the I-queue and D-queue stacks become tops and connect to the E-unit again. The result object 5.0 (the result of MEAN:<4.0,5.0,6.0>) carried by the E-unit is enqueued to the presently active D-queue, and the execution of the remaining Q-instructions in the I-queue resumes.

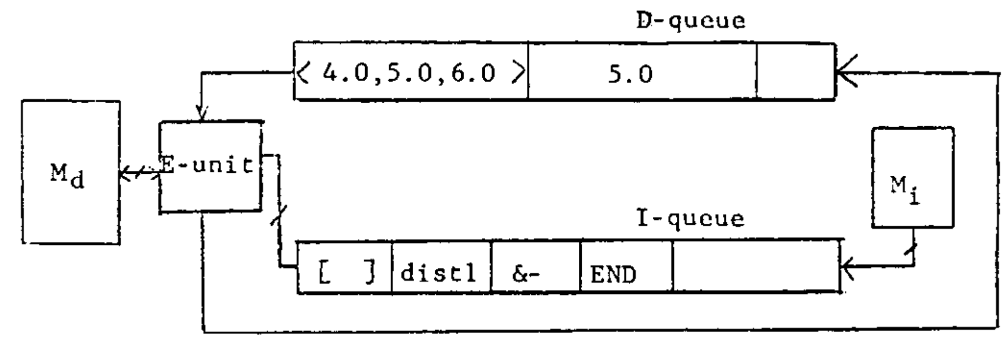


Figure 2.38 Result of MEAN Enqueued to D-queue

2.6.3 Execution of Functional Forms

In an FP programming language, besides the basic system-defined functions, there are also functional forms that define new functions by operating on function and object parameters of the form [1]. Functional forms considered here include Composition (f@g), Construction ([f_1, f_2, ..., f_n]), Condition (f→g,h), Constant(%n), Right Insert(!f), Tree Insert(|f), Apply to All (&f) and While (while f g). We now discuss the executions of these functional forms in the Q-machine.

2.6.3.1 Composition

In the Q-machine, the functions connected by the Composition symbol, "@", are dealt with as two levels of the FP tree. For example, "f@g" is a tree in Figure 2.39. Therefore, f@g is translated into Q-code as

g f

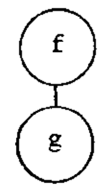


Figure 2.39 Tree Graph of f@g

Here function g is executed first, then the result of g will be enqueued to D-queue and used by f later. The compiler takes care of putting the functions on both sides of "@" in the proper order of execution, so there is no need for the E-unit to deal with it in a particular way.

2.6.3.2 Construction

The functional form Construction is dealt with as a special function in the Q-machine. Form Construction ([ ]) is the root of the function tree, and the functions inside the Construction brackets are the leaves of the tree. Figure 2.40 show the tree structure for Construction form [f_1, f_2,..., f_n]. (n is the number of functions inside the square brackets). The number of the functions inside the Construction brackets is known at the compilation time and is stored in the O field of the instruction representing the Construction.

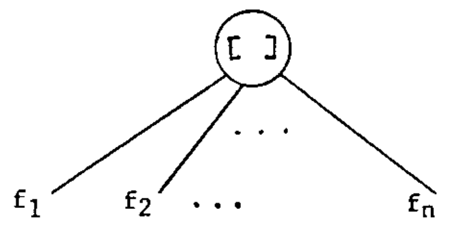


Figure 2.40 Construction

Because of the way of generating Q-instruction, when the instruction of the Construction reaches the front of the I-queue, the values r_n, r_{n-1}, ..., r_2 and r_1, resulting from executing f_n, f_{n-1}, ..., f_2 and f_1, respectively, are in the first n words in the front of the D-queue. Figure 2.41 gives the system snapshot at the moment Construction is encountered.

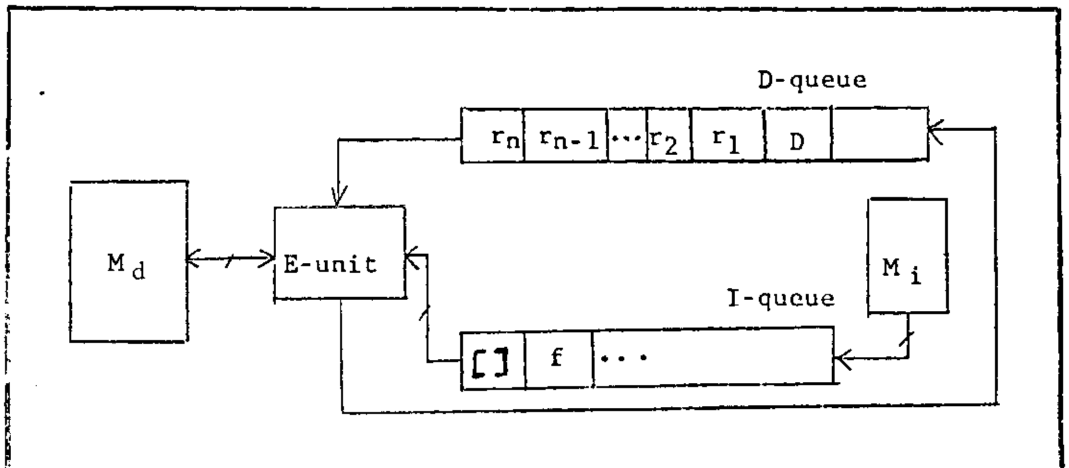


Figure 2.41 Execution of Construction

At this point, the E-unit fetches and removes the data in the first n words from the front of the D-queue. The E-unit creates a sequence which consists of these n objects as its elements. The sequence then is enqueued to D-queue as the result of Construction. Figure 2.42 shows the system after the result sequence is enqueued into the D-queue.

Observing FP program trees, we can find out that the functional form Constructions are the only nodes that may have more than one descendants.

For example, we have the following FP program:

{ CONSTR f1@[f2αf3, f4, f5@f6αf7] @f8 }

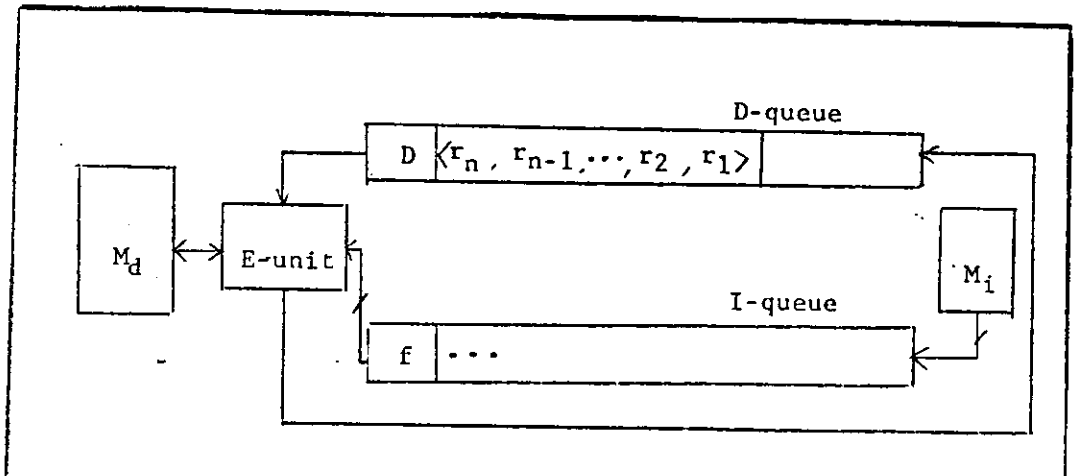


Figure 2.42 Result of Construction

where f1-f8 are either FP system-defined or user-defined functions.

Its tree representation is shown in Figure 2.43.

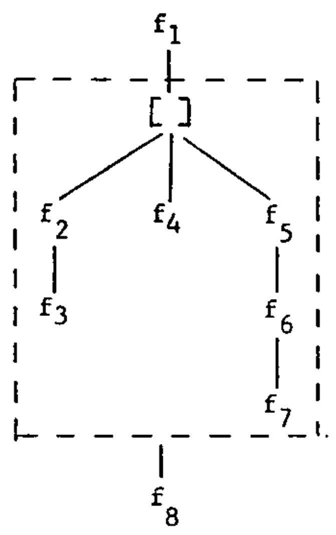


Figure 2.43 Tree Graph of CONSTR

In Figure 2.43, the part which is surrounded by dashed lines is the functional form Construction. Apparently, it is an unbalanced tree. If the tree is directly translated into Q-instructions, the execution of the Q-instructions will fail to perform correctly. The data resulted from f8 will not be able to reach f3 and f4, which is required by the above FP program CONSTR.

In order to overcome this problem, two steps need to be taken:

1. The result of executing f8 will be enqueued to D-queue n times, where n is the number of legs of the Construct tree. In the above example, n is 3. The number n is detected at compilation time and is stored in the N field of the Q-instruction representing the function f8 by the compiler.

2. FP system-defined function "id" will be stuffed into the unbalanced tree by the compiler to make it balanced. In the above example, three ids will be stuffed in.

After the above two steps are taken, the tree in Figure 2.43 appears as shown in Figure 2.44. The Q-instructions compiled from the FP program CONSTR are:

```
f8 (3)  f7 (1)  id (1)  id (1)  f6 (1)  id (1)  f3 (1)
f5 (1)  f4 (1)  f2 (1)  [ ] (1)  f1 (1)
```

The number inside the parethesis following the Q-instruction is the number of the times that the result of the Q-instruction would be enqueued by the E-unit to the D-queue.

2.6.3.3 Condition

The functional form Condition is defined as:

$$\begin{aligned}(\mathrm{f}\rightarrow \mathrm{g;h}) : x \equiv & \\ & (f{:}x) = \mathrm{T} \rightarrow \mathrm{g{:}x}; \\ & (f{:}x) = \mathrm{F} \rightarrow \mathrm{h{:}x};\ ?\end{aligned}$$

Here "f" is the predicate function, "g" is the true part function and "h" is the false

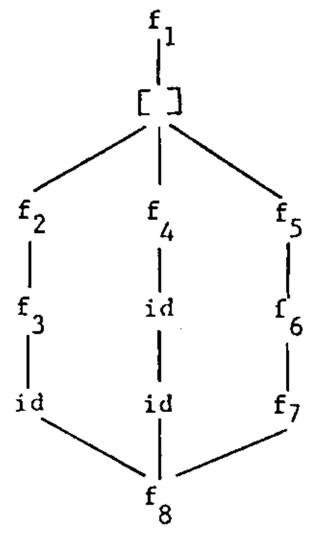


Figure 2.44 Balanced Tree of CONSTR

part function. If the result of applying f to object "x", i.e. (f:x), is T(rue), then the true part function g will be applied to x, and the result, g:x, will be returned as the result of this Condition. However, if (f:x) turns out to be F(alse), then the result of applying h to x, i.e. h:x, will be the result of the Condition.

At the time of compilation, the compiler generates one instruction to represent the entire Condition functional form. The instruction has a special tag to indicate that it is the functional form Condition, and has a name given by the compiler in the O field of instruction. The given name is unique in the function names of this FP program file. During the compilation time, the compiler compiles the predicate function (f) part, the true action function (g), and the false action function (h) separately as user-defined functions. The compiler gives a name, which is correlated to the name given to the entire functional form Condition above, to each of these functions. Later when the Q-machine executes the instruction representing the Condition, it can therefore fetch its predicate and true or false action functions.

To illustrate the execution of Condition, we give the following FP program as an example:

```
{EXAMPLE
  f1@ (<@[id,%0]→ -@[%0,id]; id) @f2
}
```

This FP program is to perform f2 first, then get the absolute value of the result from f2 and turn it to f1. The result from f1 will be the final result of function EXAMPLE. Because we are interested in the functional form Condition, f1 and f2 are not of our concern.

The FP expression inside the parentheses is a functional form Condition. The predicate part, "<@[id,%0]", tests whether the applied number is negative. If it is true (the number is negative), the true action part, "-@[%0,id]", will be applied to the number to get its absolute value. If the predicate function part returns a F(alse), which means the number is positive, the false part function, which is function "id", will be applied to the number.

The Q-code compiled from the above FP program EXAMPLE are listed in Figure 2.45. (Assume the name given by the compiler to the Condition is "COND1").

Function Name	Q-instructions						
EXAMPLE	f_2	COND1	f_1	END			
COND1?	id	%0	id	[ ]	<	END	
COND1#	id	id	%0	[ ]	-	END	
COND1~	id	END					

Figure 2.45 Q-instructions of EXAMPLE

Here "COND1?" is the name given by the compiler to the predicate part of the Condition. The name is formed by appending a "?" to the name given to the Condition. "COND1#" is the name given to the true part of the Condition. The name is formed by appending a "#" to the name given to the Condition. "COND1~" is the name given to the false part of the Condition. The name is formed by appending a "~" to the name given to the Condition.

The system snapshot at the time the Q-machine executes the functional form to the atom -2 is shown in Figure 2.46.

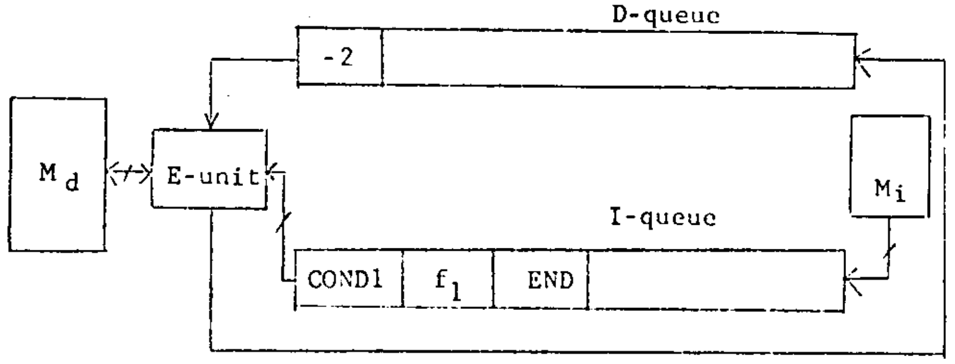


Figure 2.46 Execution of EXAMPLE

When the Q-machine encounters the Condition instruction with name "COND1", it fetches the Q-instruction of the predicate part function, which is stored in the memory with its name "COND1?", and executes it as a regular user-defined function. The system after the Q-code of "COND1?" is pushed to the I-queue stack and the atom -2 is pushed to the D-queue stack as shown in Figure 2.47.

The result of executing the predicate function part "COND1?" to -2 is T(rue) because -2 is a negative number. Then the Q-machine fetches the true part function "COND1#" and executes it to -2 as a user-defined function as well. The system after

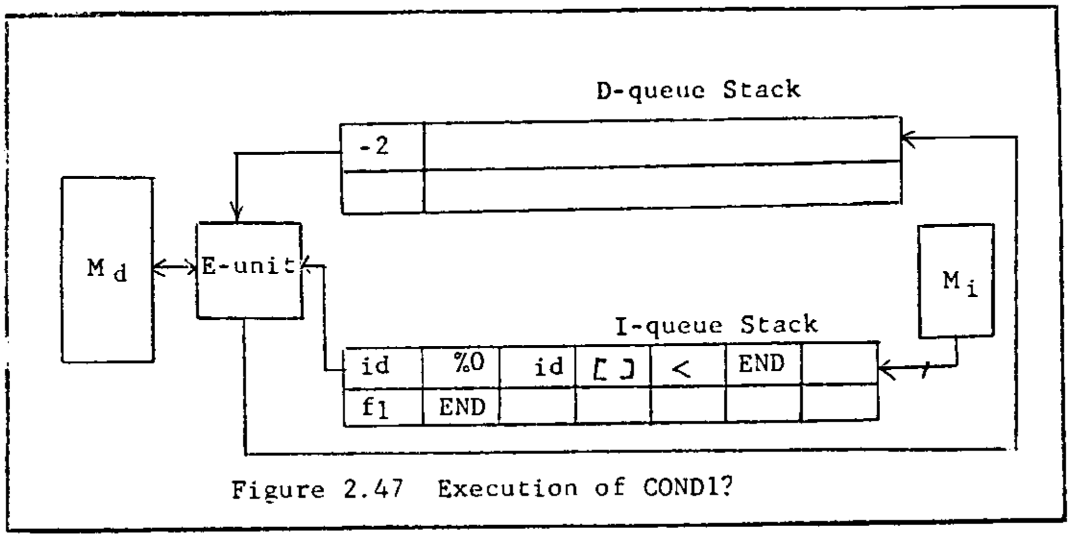


Figure 2.47 Execution of COND1?

the Q-instructions of "COND1#" and atom -2 are pushed to the I-queue stack and the D-queue stack, respectively, as shown in Figure 2.48.

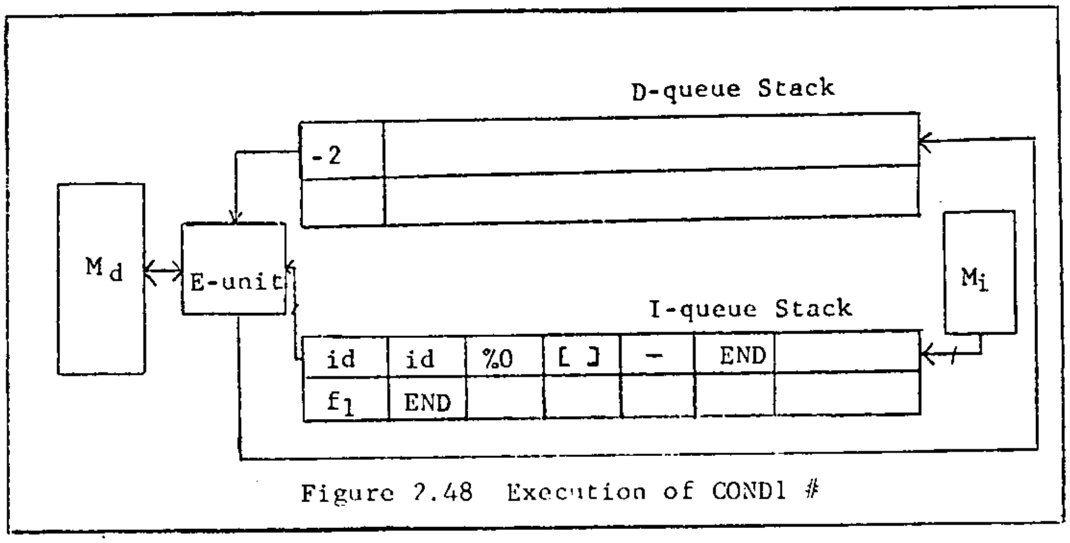


Figure 2.48 Execution of COND1 #

The result of executing "COND1#" is 2, which is the absolute value of -2. It is enqueued to the original D-queue when the top I-buffer and D-queue, after "COND#" has been executed, are popped off from their respective stacks. The system snap shot is shown in Figure 2.49.

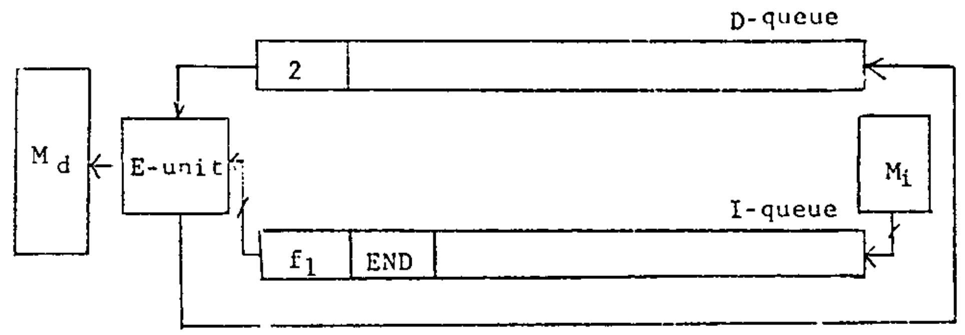


Figure 2.49 Result of COND1# Enqueued to D-queue

2.6.3.4 Constant

The definition for Constant in FP is

%x:y ≡ y = ? → ?; x, for every x being an atom.

The number x following "%" is the object parameter. The Constant form always returns x as result when it is applied to any objects other than bottom (?). In the Q-machine, Constant is dealt with as a function. Its value, x, is obtained at compilation time and attached to its instruction during compilation time. When the functinal form Constant is applied to the data in the front of D-queue, the constant, x, which is stored in the O field of instruction by the compiler will be enqueued to D-queue, if the applied object is not bottom (?). If the applied object is bottom (?), a bottom (?) will be enqueued to D-queue.

2.6.3.5 Right Insert

Right Insert is defined as follows:

!f : $x \equiv$
  $x = <> \rightarrow e_f$: x;
  $x = <x_1> \rightarrow x_1$;
  $x = <x_1, x_2, ..., x_k> \wedge k > 2 \rightarrow$ f:$<x_1$, !f:$<x_2,..., x_k>>$; ?

Because of the nature of Right Insert, the E-unit in the Q-machine has to handle it differently from regular functions.

The Pre-processor in the E-unit fetches the elements of the sequence to a cache memory inside the E-unit. There is an internal pointer, IP, pointing to the last element in the sequence.

When the Main-processor takes over, it picks up the element IP pointing to, in this case, x_k, and applies function f to x_k. The result, f:x_k, replaces x_n in the cache after this application. Then IP is decreased by one, and points to x_{k-1}. The Main-processor picks up x_{k-1} and f:x_k to form a new sequence, and then applies function f to this new sequence, $<x_{k-1}$,f:$x_k>$. The iteration of operation continues until x_1 is picked up and the last result is generated. The last result then is enqueued to the D-queue.

If f is a system-defined function, the Main-processor can execute it directly. If f is a user-defined function or a system-generated function, the Main-processor will execute it in the same way it executes other user-defined functions.

2.6.3.6 Apply to All

The functional form Apply to All is defined as follows:

&f : $x \equiv$
  $x = <> \rightarrow <>$;
  $x = <x_1, x_2, ..., x_k> \rightarrow <$f:$x_1$, f:$x_2$, ..., f:$x_k>$; ?

Apply to All applies the function following the symbol "&" to all elements of the sequence and returns a sequence consisting of the results of applying f to each element.

At execution time, the Pre-processor gets the functional form &f from the I-queue and the data sequence from the D-queue. Then the Pre-processor fetches the elements x_1, x_2, ..., x_k of the sequence x to a cache.

At the next E-unit cycle, the Main-processor takes over. It starts applying f to x_1, and puts the result, f:x_1, to a newly created sequence y. It goes through from x_1 to x_k and puts the results one by one to sequence y. Finally, after all elements are applied, the Main-processor enqueues the descriptor of sequence y to the rear of the D-queue.

If f is a system-defined function, the Main-processor applies it to each element in x as we described above.

If f is a user-defined function, the Main-processor will invoke the function and perform the I-queue and D-queue stack operations.

The functional form of Apply to All can be executed in parallel in a system with multi-processors. Details will be given in Chapter 3.

2.6.3.7 While

The functional form While is defined as follows:

(while f g) : $x \equiv$
    $f{:}x$ = T → (while f g) : (g:x);
    $f{:}x$ = F → x; ?

Both f and g are FP expressions. We call f the predicate function and g the action function of While.

Applying While to object x, we apply function f to x first. If its result is T(rue), we will apply the While to the result of g:x. If the result of f:x is F(alse), we will return x as the result of the While functional form.

At compilation time, the compiler generates one instruction to represent the entire While functional form. The instruction has a special tag attached to indicate it is a While functional form. Similar to the compilation of Condition, a unique name is given by the compiler to the While. Meanwhile, the compiler compiles the predicate part and action part of While separately as two user-defined functions. Each of the functions is given a name which is correlated to the name given to the While functional form.

To illustrate the procedure, we give an example of the FP program FACTORIAL:

```
{FACTORIAL
  1@ (while >@[2,%1] [*@[1,2], -@[2,%1]) @[%1,id]}
}
```

After compilation, the Q-instructions of FACTORIAL are shown in Figure 2.50.

Function Name	Q-instruction									
FACTORIAL	id	id	%1	[ ]	WHILE1	1	END			
WHILE1?	id	%1	2	[ ]	>	END				
WHILE1#	id	%1	2	2	1	[ ]	[ ]	-	*	END

Figure 2.50 Q-instructions of FACTORIAL

The name "WHILE1" is given by the compiler to the While functional form. The name "WHILE1?" is formed by appending "?" to the name "WHILE1", and given by the compiler to the predicate part function. The name "WHILE1#" is formed by appending "#" to the name "WHILE1", and given by the compiler to the action part function.

When FACTORIAL is executed on the Q-machine, the E-unit meets the instruction of the While functional form associated with the name "WHILE1" after 4 machine cycles. The E-unit recognizes that it is a functional form While according to the tag attached to the instruction during compilation. The E-unit keeps the object D from the front of the D-queue, applied by the While, and fetches the Q-instructions of "WHILE1?", which are the predicate part of "WHILE1", and applies them to the data D in the same way it executes regular user-defined the functions. If the result of the execution is T(rue), the E-unit fetches the Q-instructions of "WHILE1#" and applies them to the data D as a regular user-defined function. The result of the execution, D_1, will replace D in the E-unit. Then the E-unit applies the predicate function "WHILE1?" to D_1 again. The iteration continues until the result of predicate function "WHILE1?" applied to the object D_n becomes F(alse). After the E-unit gets the result of F(alse), it returns the object D_n as the result of functional form While - "WHILE1".

In the next chapter we discuss several alternatives in the Q-machine organization which allow the exploitation of parallelism in FP programs.

CHAPTER 3

Q-machine With Multiple Processors

The characteristics of the FP language and the Q-machine allow us to explore possible parallel executions in different levels if the Q-machine has multiple processors.

In this chapter, we discuss several alternative organizations for parallel executions of FP programs. In FP, certain functional forms, such as Apply to All and Tree Insert, explicitly provide the opportunities for parallel executions. Another functional form, Construction, can also be executed in the Q-machine in parallel. In addition, the organization of the Q-machine also has the potential of using parallel E-units connecting to the D-queue and I-queue to carry out concurrent executions. In the following sections, we discuss the parallel execution of Apply to All, Tree Insert and Construction in the Q-machine with multiple E-units or multiple Q-machine units. We will also discuss the parallel executions using parallel E-units connecting to the D-queue and the I-queue.

3.1 Parallel Execution of Apply to All

In FP, Apply to All (&f) is a functional form. Its function parameter f could be a system-defined function, or a user-defined function. In Section 3.1.1, we will discuss the parallel execution of &f where f is a system-defined function. In Section 3.1.2, we will discuss the parallel execution of &f where f is a user-defined function.

3.1.1 Parallel Execution of Apply to All Using Multiple E-units

Assume that the Q-instruction in the front word of the I-queue of the Q-machine is a functional form Apply to All (&f), and that the data in the front of the D-queue is a sequence, D, having n elements. If the Q-machine has only a single E-unit, it has to apply the parameter function f to each element in D sequentially. The time cost then would be nt (where t is the time cost for applying f to one element of sequence D). We can improve the performance by having n E-units apply the Q-instruction f to the n elements of the sequence D in parallel.

In the Q-machine with parallel execution using multiple E-units, when the original system E-unit encounters Apply to All and the data sequence D, which has n elements, it will invoke a group of E-units, where the number of the E-units in the group equals the number of the elements in the applied sequence D.

We call the invoking E-unit "Master E-unit" and the invoked E-units "Slave E-units." Figure 3.1 shows the relationship between a master E-unit and its slave E-units.

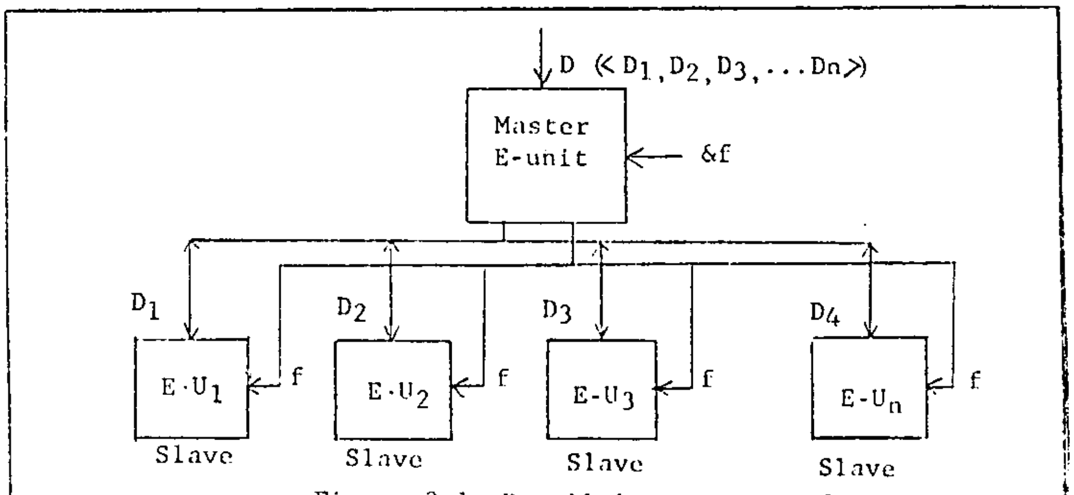


Figure 3.1 Parallel Execution of &f

The master E-unit broadcasts the function parameter f of the functional form Apply to All, &f, to every slave E-unit. It also passes the first element in the sequence, for instance D_1, to E-U_1; the second element D_2 to E-U_2, ..., and the nth element D_n to E-U_n. Each Slave E-unit will apply the function f to the received data and return the result to the master E-unit. After receiving all results from these slave E-units, the master E-unit forms a sequence which includes all the results as elements in an order corresponding to their original data and enqueues the formed sequence to the D-queue, to which the master E-unit is connected.

The Q-machine with the capability of executing the Apply to All in parallel using multiple E-units has a speedup of n, (n is the number of elements in the sequence to which Apply to All will be applied), versus the Q-machine using a single E-unit executing Apply to All sequentially.

3.1.2 Parallel Execution of Apply to All Using Multiple Q-machines

The scheme introduced in section 3.1.1 for executing Apply to All (&f) in parallel is only good if f is a system-defined function, because the E-units used to carry out the parallel executions, unlike a complete Q-machine, does not have its own D-queue and I-queue, which are necessary to execute user-defined functions.

In order to execute Apply to All (&f) in parallel when f is a user-defined function, each slave E-unit in the configurations in section 3.1.1 used for parallel executions must be replaced by an entire Q-machine unit.

The following Figure 3.2 shows the configuration of using multiple identical Q-machine units to execute Apply to All &f in parallel, where f is a user-defined function.

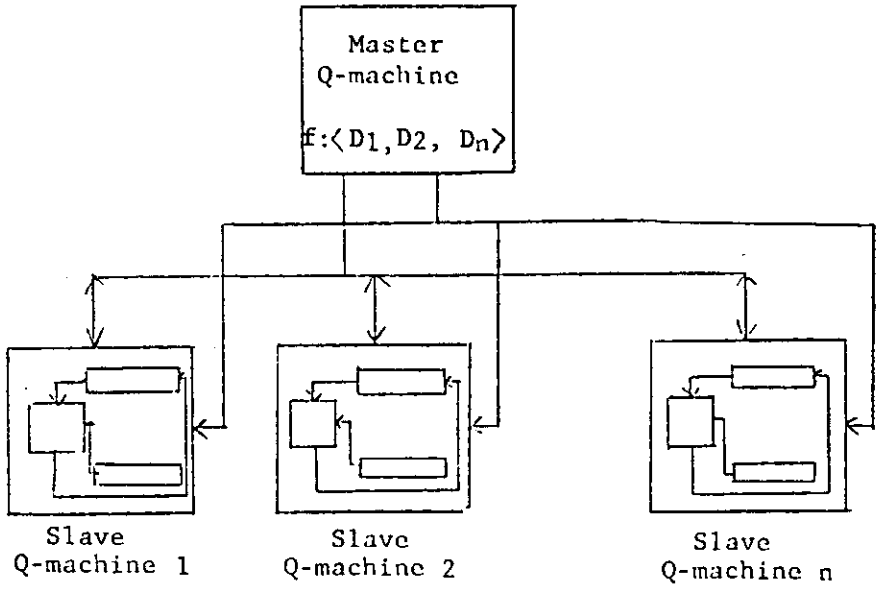


Figure 3.2 Palallel Executions Using Mutiple Q-machines

When the Q-machine encounters an &f in the front of its I-queue, and the data, D, which is a sequence consisting of n elements D_1, D_2, ... D_n, in the front of its D-queue, it invokes n Q-machine units Q_1, Q_2, ..., Q_n, which are identical to the invoking Q-machine unit, to apply f to each element of D in parallel.

The invoking Q-machine is called the master Q-machine and the invoked Q-machines are called slave Q-machines.

The master Q-machine broadcasts the Q-instructions of f to all slave Q-machines. (If the scheme is implemented in shared memory, only the pointer to the starting instruction of Q-code of f needs to be broadcasted to each slave Q-machine). The master Q-machine passes the data D_1 to Q_1, D_2 to Q_2,..., D_n to Q_n. After receiving the Q-instructions and data, each slave Q-machine begins execution like an independent Q-machine. After completing the execution, the slave Q-machine returns the result to its master Q-machine and is detached from its master. Having received all the results from its slave Q-machines, the master Q-machine forms a sequence to include these results as elements and enqueues the sequence to its D-queue. The execution of the rest of the Q-instruction continues.

3.2 Parallel Execution of Tree Insert

In FP, Tree Insert (&f) is a functional form. Its function parameter f could be a system-defined function, or a user-defined. In Section 3.2.1, we will discuss the parallel execution of |f where f is a system-defined function. In Section 3.2.2, we will discuss the parallel execution of |f where f is a user-defined function.

3.2.1 Parallel Execution of Tree Insert Using Multiple E-units

Tree Insert is defined in FP as:

$$\begin{array}{l} |\mathrm{f} : x \equiv \\ \quad x = \,<> \rightarrow e_f : \mathrm{x}; \\ \quad x = \,<x_1> \rightarrow x_1; \\ \quad x = \,<x_1, x_2, \ldots, x_k> \wedge \mathrm{k} > 1 \rightarrow \\ \qquad \mathrm{f} : < |\mathrm{f} : <x_1, \ldots, x_{\frac{k}{2}}>, |\mathrm{f} : <x_{\frac{k}{2}+1}, \ldots, x_k>>; \, ? \end{array}$$

To execute a Tree Insert |f, the master E-unit splits the input sequence D into two sequences, D_1 and D_2, then invokes two slave E-units, E-U_1 and E-U_2. The master E-unit passes the Tree Insert |f to both slave E-units, but passes D_1 to E-U_1 and D_2 to E-U_2. If D_1 has more than one element, E-U_1 will split D_1 again and invoke another two slave E-units, E-U_{11} and E-U_{12}. The same applies to E-U_2. The procedure will go on until a slave E-unit has an input sequence which is a null sequence or a sequence has just one element. Figure 3.3 shows the splitting procedure for executing |+ : <1,2,3,4,5,6,7,8>. When the splitting stops, the E-unitss at the bottom level, such as E-U_{31}, E-U_{32}, E-U_{33}, E-U_{34}, E-U_{35}, E-U_{36}, E-U_{37} and E-U_{38}, begin to apply function + to their data, then return the results to their master E-units, such as E-U_{21}, E-U_{22}, E-U_{23}

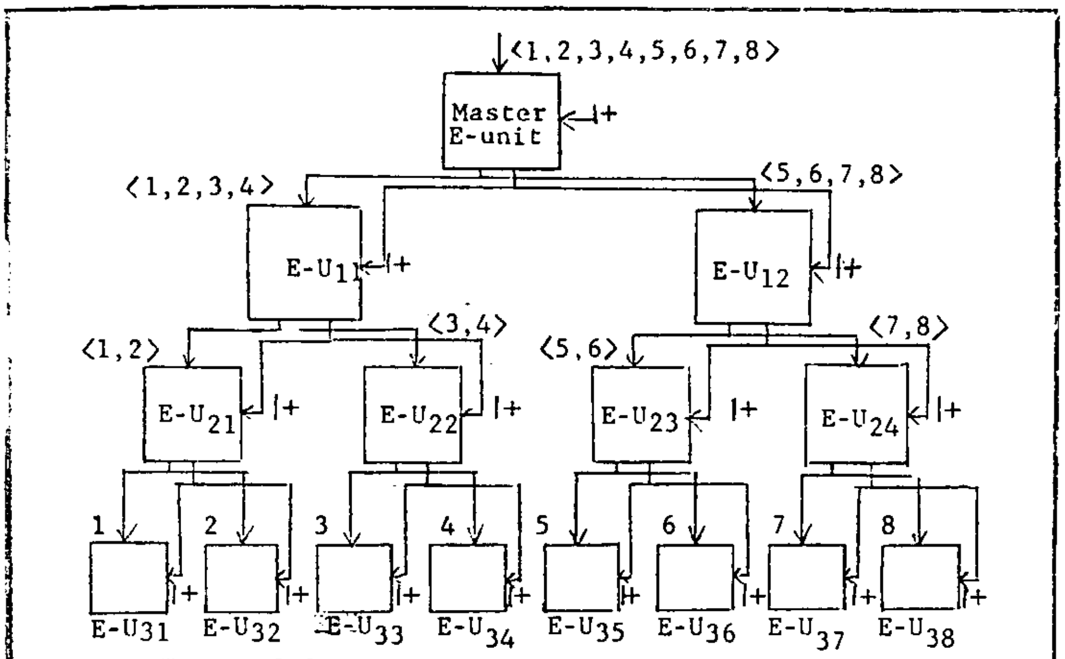


Figure 3.3 Parallel Executions of |+:<1,2,3,4,5,6,7,8>

and E-U_{24}. The E-units that complete their executions are detached from their master E-units. Figure 3.4 shows the system after the bottom level E-units have completed their executions.

Now, the E-units in the third level, E-U_{21}, E-U_{22}, E-U_{23} and E-U_{24}, begin to apply + to the sequence formed by the results of the lower level, and return the results of execution to their master E-units, which in this example are E-U_1 and E-U_2. Figure 3.5 shows the system after executions in the third level are finished.

The procedure goes on until the top level Master E-U finishes the execution of applying + to its data, which is a sequence formed by the data returned from the second level executions. The top master E-units will enqueue the final result to the D-queue, to which it is connected. Figure 3.6 shows the system state after the executions of |+ are finished.

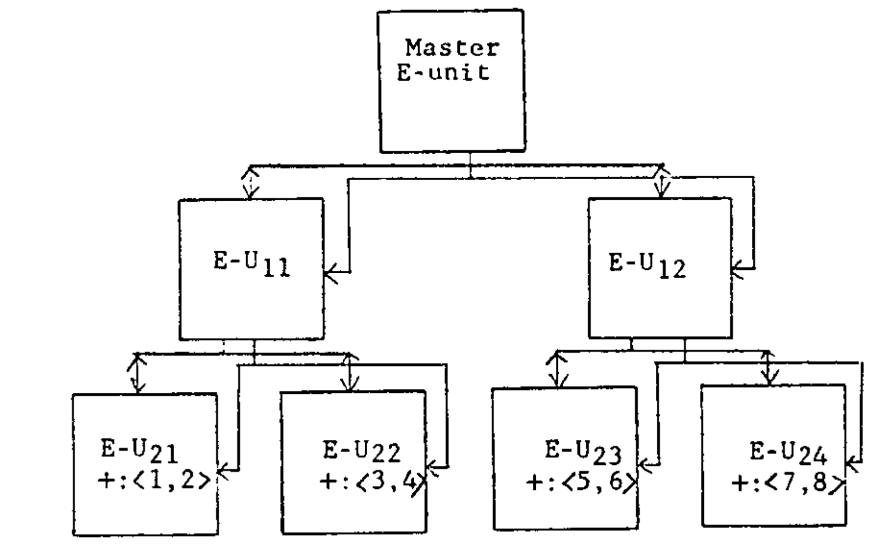


Figure 3.4 Parallel Executions of |+

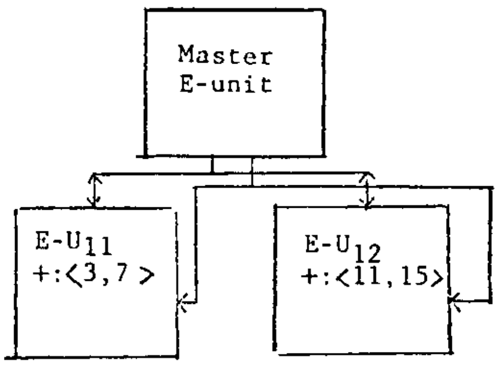


Figure 3.5 Parallel Executions of |+

The E-units in the same level can execute in parallel. The time cost of executing Tree Insert |f is

$$\lfloor lg_2 N \rfloor \times T_f + T_{split}$$

where N is the number of elements in the applied sequence, T_f is the time cost to apply

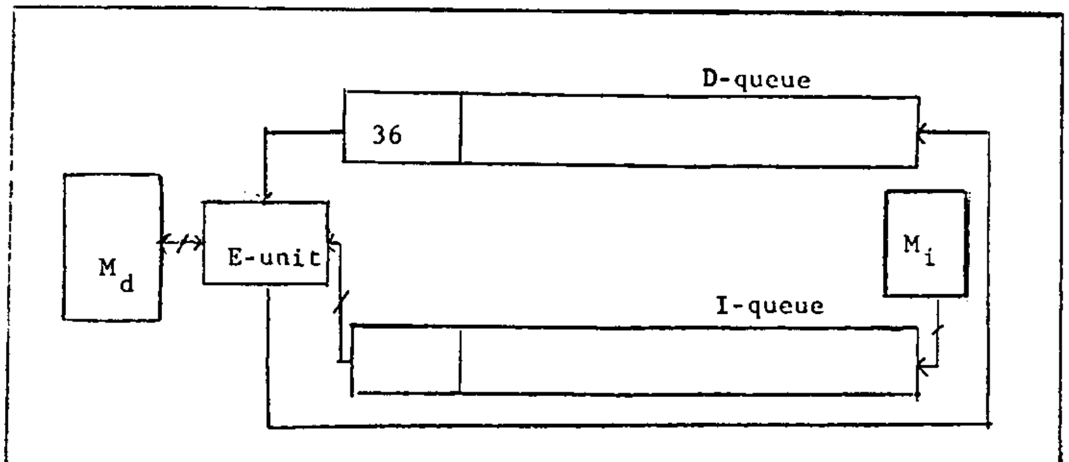


Figure 3.6 Result 36 Enqueued to D-queue

f to data, T_{split} is the time cost to split the applied sequence.

The speedup for Tree Insert executed in parallel using multiple E-Us versus one executed in a single E-unit is

$$\frac{N-1}{\lceil lg_2 N \rceil + \frac{T_{split}}{T_f}}$$

3.2.2 Parallel Execution of Tree Insert Using Multiple Q-machines

The parallel execution of the Tree Insert (|f), when f is a user-defined function is similar to the parallel execution of Tree Insert when f is a system-defined function. Only the slave E-units in Figure 3.3 will be replaced with slave Q-machines.

3.3 Parallel Execution of Construction

The execution of Construction in the Q-machine with a single processor is described in Section 2.6.3.2. The solution is suitable for the executions of Constructions in the Q-machine with a single processor.

If we use the scheme of invoking multiple Q-machines, we will not only be able to avoid stuffing ids to an unbalanced tree, but also be able to execute the legs of the Construction in parallel.

To illustrate how to use multiple Q-machines to execute Construction, we recall the example used in Section 2.6.3.2. It is the FP program CONSTR, which was defined as the following:

{CONSTR f1@[f2@f3, f4, f5@f6@f7]@f8}.

Assume there are multiple Q-machines which can be invoked by a master Q-machine.

At the compilation time, the functional form Construction will be translated into a single Q-instruction, [], of which the O field contains a system-given name, for instance, LEG. The n legs of the Construction will be compiled separately to n Q-codes having names given by the compiler. These names, LEG1, LEG2 and LEG3 in this example, are associated with the name in the O field of the Construction Q-instruction. The Q-instructions of the FP program CONSTR appears as Figure 3.7 shows.

Function Name	Q-instructions				
CONSTR	f_8	[ ]	f_1	END	
LEG1	f_3	f_2	END		
LEG2	f_4	END			
LEG3	f_7	f_6	f_5	END	

Figure 3.7 Q-instructions of CONSTR

When the master Q-machine encounters the instruction of Construction in the front of the I-queue and the data D, which is the result of f8, in the front of the D-queue, it will invoke 3 slave Q-machines to execute the three legs of the Construction tree in parallel. The master Q-machine broadcasts the data D to the D-queue of each slave Q-machine. It passes the Q-instructions of LEG1 to the slave machine 1, the Q-instructions of LEG2 to the slave machine 2, and the Q-instructions of LEG3 to the slave machine 3. Figure 3.8 shows the configuration.

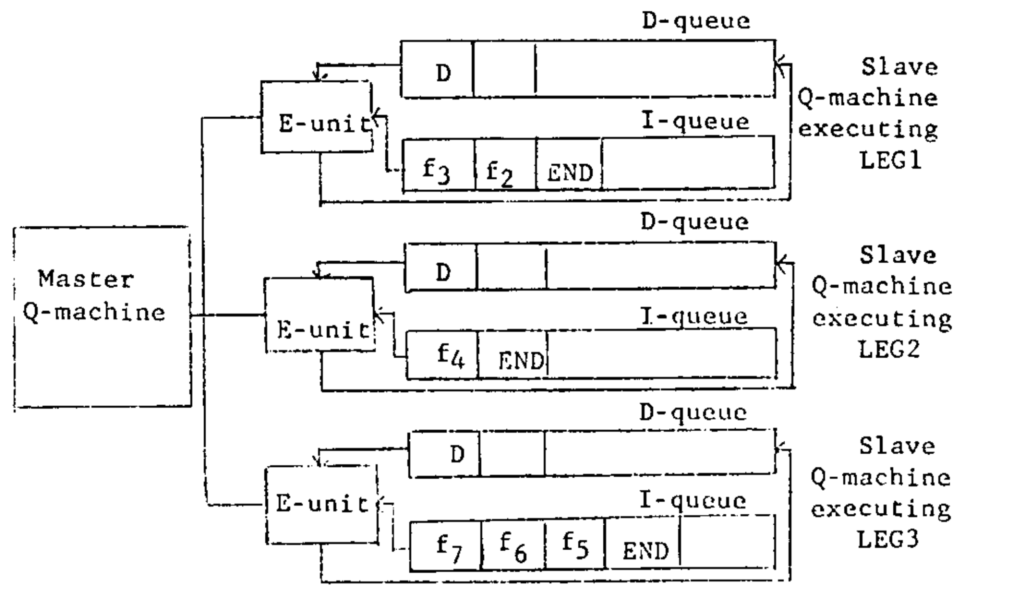


Figure 3.8 Parallel Execution of Construction

Having received the data and Q-code from the master Q-machine, each of the slave machines begins execution in parallel. After all three slave machines return their results to the master machine, the master machine puts the results into a sequence. The sequence, which is the result of the execution of Construction, then is enqueued to the the D-queue of the master Q-machine. The operation in the master Q-machine resumes.

The speedup of having multiple Q-machines execute n legs of a Construction tree in parallel versus having a single Q-machine execute the balanced Construction tree sequentially is about n.

3.4 Parallel Execution Using Parallel E-units Connecting to D-queue and I-queue

In the Q-machine with a single processor discussed in Chapter 2, there is one E-unit connected to the D-queue and the I-queue, and it only operates on the front of the D-queue and the I-queue. Figure 3.9 shows the snapshot of the system when the data in the front word of the D-queue, D1, and the Q-instruction in the front word of the I-queue, I1, are executed by the E-unit.

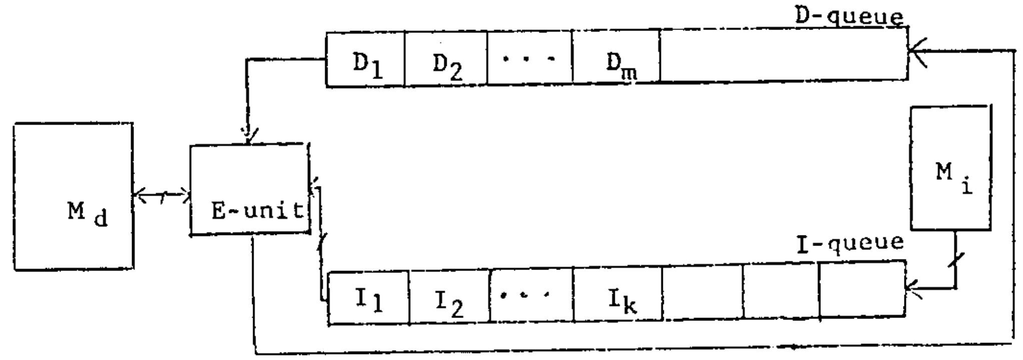


Figure 3.9 Q-machine With A Single E-unit

In the execution of FP programs on the Q-machine, if the front Q-instruction in the I-queue is any Q-instruction other than Construction, only one operand in the front of the D-queue is required for applying the Q-instruction. If the front Q-instruction in the I-queue is Construction, then n front operands in the D-queue are required to form the resulting sequence, where n is the number of elements in the sequence to be formed and n is attached to the Construction instruction during compilation time.

Assume there are a total of m operands in the D-queue at a moment, and each instruction, I_i, in the I-queue, needs O_i operands from the D-queue. (As indicated above, O_i is 1 for all Q-instructions other than Construction, O_i is a non-negative integer for Construction). We can find a number k such that

$$\sum_{i=1}^{k} O_i \leq m$$

The number k indicates that all the operands needed for executing instructions I_1, I_2, ..., I_k, are already in the D-queue. Therefore, with a single E-unit, at the time that I_1 is applied to D_1, D_2, ..., D_{O_1}, all Q-instructions I_2, ..., I_k in the I-queue and operands D_{O_1+1}, D_{O_1+2}, ..., D_m, are idle. If there are k E-units available in the Q-machine, we can have all these idle (k-1) Q-instructions to be executed concurrently with the execution of I_1. We call this parallel execution using parallel E-units connecting to the D-queue and the I-queue. Figure 3.10 shows the organization of modified Q-machine to carry out the above parallelism.

In Figure 3.10, each word of the D-queue connects to the interconnection network. The network also connects to an array of E-units, which are connected to the k front words of the I-queue. The control unit of the interconnection network detemines the executable Q-instructions (I_1,I_2, ..., I_k), the operands required by each instruction and the number of E-units required (which is k as well). Indicated in [5] and [6], the shuffle-exchange network [7] is suitable for the interconnection network.

The possible parallel executions using parallel E-units connected to the D-queue and the I-queue are determined by the tree structure of the FP program. For example, we have the following FP program tree as Figure 3.11 shows.

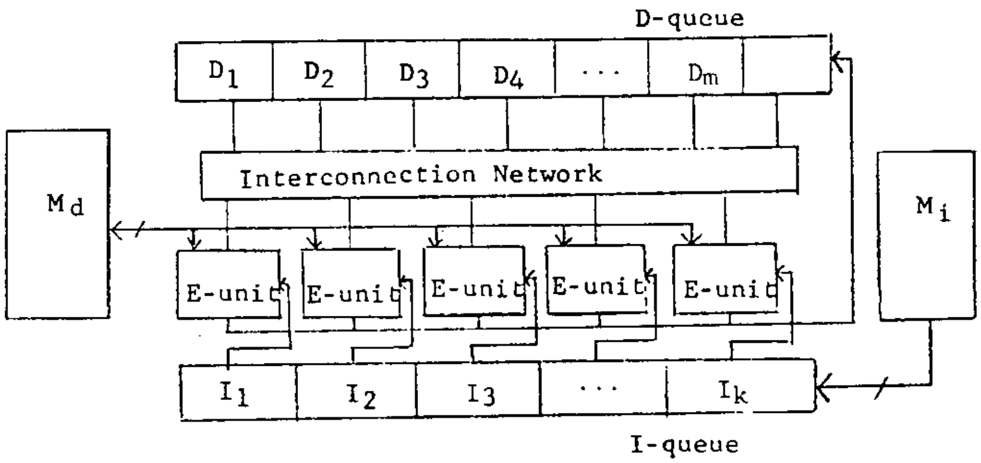


Figure 3.10 The Q-machine With Parallel E-units

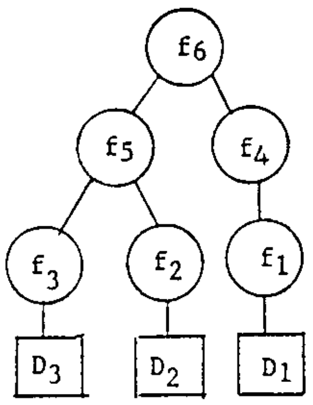


Figure 3.11 Tree of A FP Program

The compiled Q-instructions of this FP program are:

f1 f2 f3 f4 f5 f6.

The data to be applied to this program is D_1, D_2 and D_3.

After the Q-instructions and data are loaded to the I-queue and the D-queue, respectively, the Q-machine system appears as Figure 3.12 shows. (Here $O_1 = 1$, $O_2 = 1$, $O_3 = 1$, $O_4 = 1$, $O_5 = 2$, $O_6 = 2$).

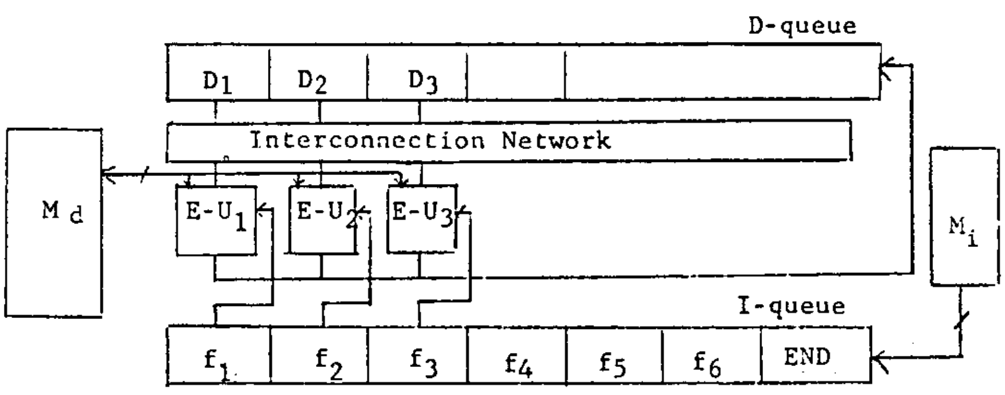


Figure 3.12 Parallel Excutions of f1, f2 and f3

In the above Figure 3.10, three front Q-instructions in the I-queue can be executed in parallel. (Here k=3 because m=3 and $O_1+O_2+O_3 = 3$)

After the first machine cycle of parallel execution, when the results R_1, R_2 and R_3 from E-U_1, E-U_2 and E-U_3, respectively, are enqueued to D-queue, the system becomes as the following Figure 3.13 shows.

At the second machine cycle, two front Q-instructions in D-queue can be executed in parallel. (Here k=2, because m=3 and O_1=1 and O_2=2). Data R_1 are fed to E-U_1 and will be applied by f4; data R_2 and R_3 are fed to E-U_2 and will be applied by f5. The result R_4 and R_5 from E-U_1 and E-U_2, respectively, are enqueued to D-queue like Figure 3.14 shows.

At the third machine cycle shown in Figure 3.15, only one Q-instruction, f6 in I_1, can be executed, because O_1=2 and m=2 at the moment. Data R_4 and R_5 are fed

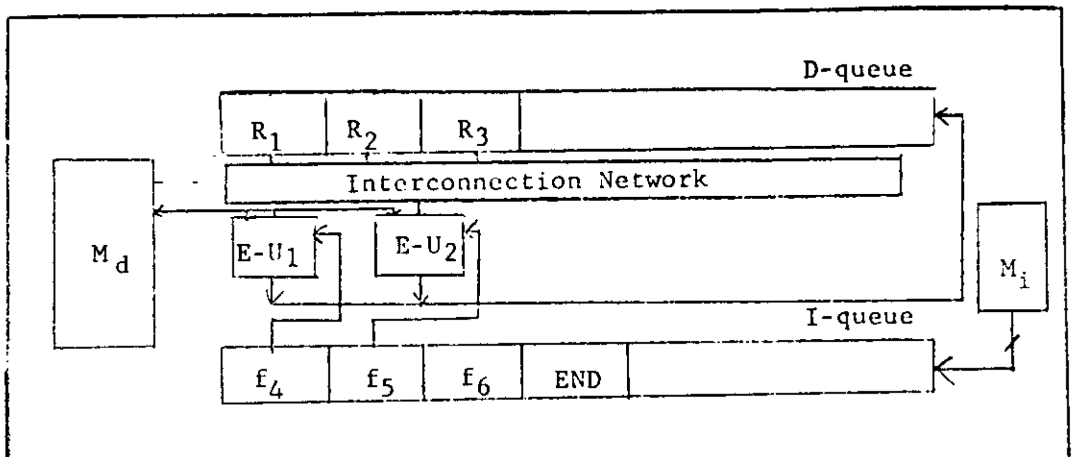


Figure 3.13 Parallel Executions of f_4 and f_5

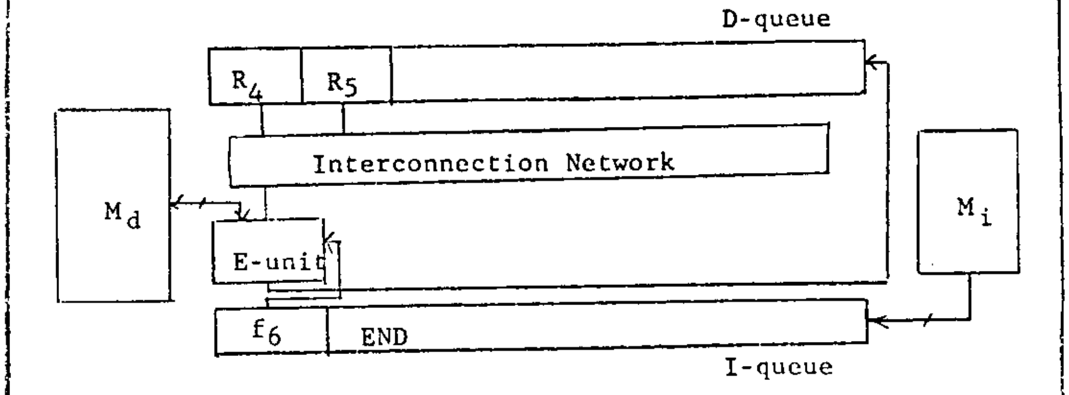


Figure 3.14 Execution of f_6

to E-U_1 and are applied by f6. The result, R_6, will be enqueued to the D-queue, which is the final result.

From the above illustration, we can find that the nodes in one level of the FP program tree can be executed in parallel using parallel E-units connecting to the D-queue and the I-queue. The execution time cost for one level is the maximum within the time costs of nodes in that level, which is:

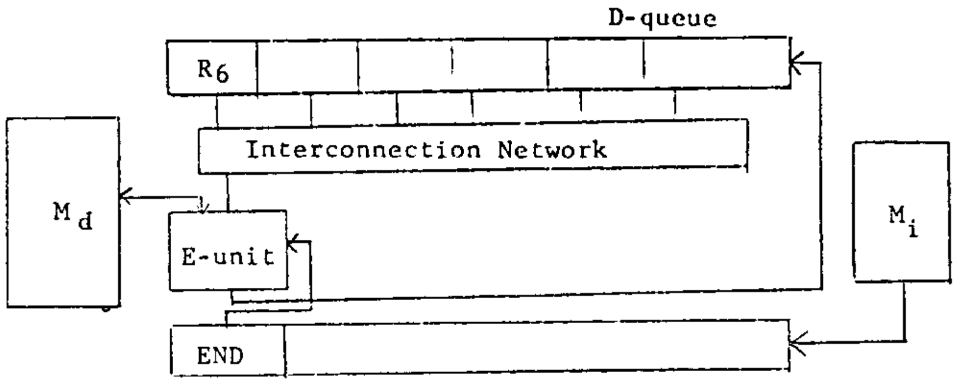


Figure 3.15 Result R_6 Enqueued to Dqueue

$$T_l = Max(T_{l_i})$$

where l=level, i = 1.., n, and n is the number of nodes in that level.

Therefore, for a program tree having a height H, to be executed in parallel using parallel E-units connecting to the D-queue and the I-queue, the time cost will be:

$$\sum_{l=1}^{H} T_l$$

where l is the tree level, H is the Height of the tree.

It is roughly $O(lg_k N)$ for a k-ary FP tree, where N is the number of the functions in the FP tree.

The time cost of executing the same FP program tree in a Q-machine with a single processor would be

$$\sum_{i=1}^{N} T_i$$

where T_i is the time cost of executing f_i, and N is the total number of functions in that FP program. The time cost is O(N). Executing an FP program with parallel E-units connecting to the I-queue and D-queue will save a considerable amount of time if the FP tree is well spread.

We discussed several schemes of the parallel executions in the Q-machine organization. The Q-machine organization is benefitted by certain functional forms in FP, such as Apply to All and Tree Insert, which provide the explicit opportunities for parallel executions. In addition to it, the Q-machine organization also has its own potential of parallelism, such as using parallel E-units connecting to the D-queue and I-queue to carry out concurrent executions. The different schemes that we discussed can be invoked individually or combined together to reach the maximum of the parallelism.

In order to check the feasibility of the Q-machine and the performance of FP programs running on the Q-machine, a Q-machine simulation package is needed. In the next chapter, we describe a simulation package of the Q-machine and analyze the performance of some sample FP programs, which have been run on the Q-machine simulator.

CHAPTER 4

The Simulation of the Q-machine and Performance Analysis

In order to verify the feasibility of the Q-machine design, gather statistics information about the executions of FP programs on the Q-machine, and explore the possible parallel executions in the Q-machine, a simulation package has been implemented.

Several FP programs have been run on the simulator of the Q-machine. The examples of the FP programs which have been run on the simulator and the performance of their executions are discussed in this chapter. They are DEVIATION, MATRIXMUL, QUICKSORT and TOWERHANOI programs.

4.1 The Simulator of the Q-machine

The simulation package includes three parts: a compiler, a data converter, and the Q-machine simulator. It is written in Pascal and runs under the UNIX system on VAX 11/780 at the Computer Science Department at UCLA. Figure 4.1 shows the configuration of the Q-machine simulation package.

The compiler reads in the program in Berkeley FP language from a file, which is specified by the user, and translates the FP program into executable Q-instructions. We call the compiler the Q-instruction compiler.

The Data Converter reads in the input data in FP format either directly from the user's terminal or from a data file given by the user, and converts the input data to the data format used in the Q-machine.

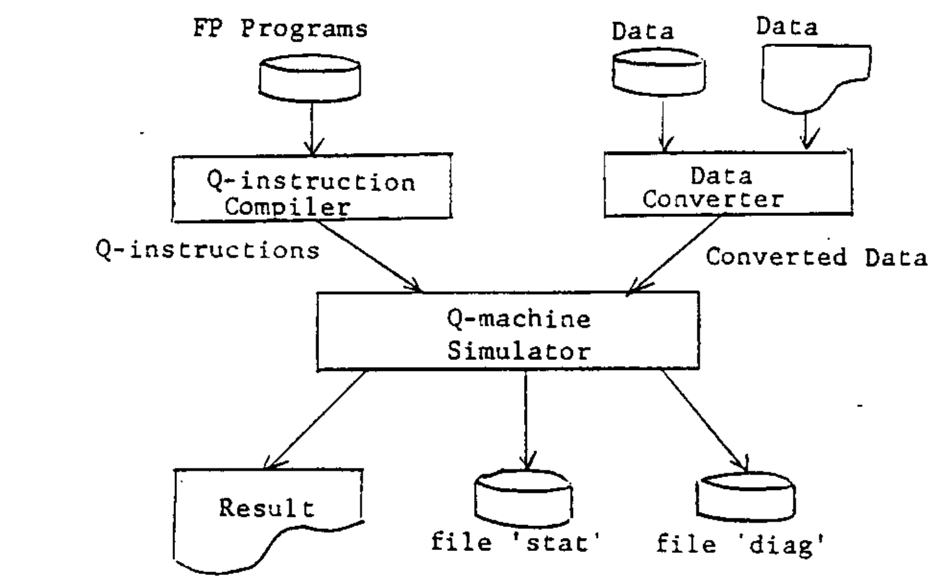


Figure 4.1 Organization of Simulation Package

The Q-machine simulator first loads the executable Q-instructions of the FP program, which is specified by the user, from the Q-instruction compiler to the Instructuion Queue (I-queue). It also loads the data, which are given by the user, from the Data Converter to the Data Queue (D-queue). Then the Q-machine starts executing the FP function. The result of the execution are printed on the screen. Meanwhile, two files are generated -- one is called 'stat', which contains the statistics data from the execution of the FP program; the other is called 'diag', which contains diagnosis data to help debug the Q-instruction compiler, Data Converter and Q-machine.

4.1.1 The Q-instruction Compiler

Compared with the conventional compilers which translate the programs in conventinal languages to ones in assembly languages, the Q-instruction compiler is simpler and more straightforward. The Q-instruction compiler used in the simulation package has about 1,000 lines of Pascal code. The natural tree structure of FP programs, the absence of variable in FP programs and the architecture of Q-machine make the Q-instruction compiler simple.

Most conventional programming language compilers consist of three phases:

1. Lexical analysis.

2. Parsing.

3. Code generation.

On the other hand, the Q-instruction compiler consists of two phases:

1. Top-down depth first, and then left-to-right FP program tree traversal.

2. Q-instruction generation.

Unlike conventional programming languages, FP does not have variables. Moreover, data and Q-instructions are totally independent of each other in the Q-machine. Therefore, the Q-instruction compiler does not need to bind attributes (such as types, scope, etc.) to variables or to allocate storage for variables.

Phase 1 of the Q-instruction compiler processes the character string of the FP program from left to right using recursive descent method. It puts all the Q-instructions (including the system-defined and user-defined functions in FP) in the order that they are encountered to a table, which is called the function table. Each Q-instruction in the function table occupies one entry, which has two fields: the Q-instruction, which is in the format of the word of the I-queue of the Q-machine; and the number of its level in the FP tree. Along with the traversal of the FP tree, necessary id stuffing (described in section 2.6.3.2 as the means of making the FP tree balanced) is carried out.

For example, the user-defined function MERGE

{MERGE distr@[1,trans@2]

is compiled by the Q-instruction compiler as follows.

During phase 1, the Q-instructions, the stuffed "ids", and the level of each Q-instruction are put into the function table shown below.

No.	Q-instruction	Level
1	distr	1
2	[ ]	2
3	1 (select)	3
4	id (stuffed)	4
5	trans	3
6	2 (select)	4

Phase 2 of the Q-instruction compiler scans the above table from the bottom up and put the Q-instruction with the deepest level first in the order which they are encountered to another table, which holds the executable Q-instructions associated with the name of the user-defined function. After n times of scanning, where n is the number of the deepest level (4 in the above case), the table which contains the executable Q-instruction is shown below. The first Q-instruction in the table, id, is added to distribute the applied data to the two legs of the FP tree of MERGE. (The reason of this was discussed in Section 2.6.3.2). The last Q-instruction, END, is added to indicate the end of the Q-instruction stream. (The level numbers are no longer needed in this table, but we put them in the table to show how they are rearranged).

No.	Q-instruction	Level
1	id	
2	2 (select)	4
3	id (stuffed)	4
4	trans	3
5	1 (select)	3
6	[]	2
7	distr	1
8	END	

Unlike the code generation phase in conventional compilers, there is no need for complex register allocation and assignment in the Q-instruction generation phase. This makes the Q-instruction generation phase of the Q-instruction compiler much simpler than the code generation phase of a conventional compiler.

4.1.2 The Data Converter

The Data Converter converts the input data to the data format used in the Q-machine. We chose the word with a fixed length and pointers to the memory as the word data structure of the D-queue in the Q-machine simulator. All input data are converted to this structure.

4.1.3 The Q-machine Simulator

The Q-machine is the main part of the simulator. It consists of three main componeets -- the Pre-processor, the Main-processor and the utility routine group.

Figure 4.2 shows the organization of the Q-machine simulator.

The Pre-processor processes the data from the D-queue. It calls ElementSep to separate the elements of the fetched object, if it is a sequence. It stores the processed data in a table (cache memory) inside the E-unit.

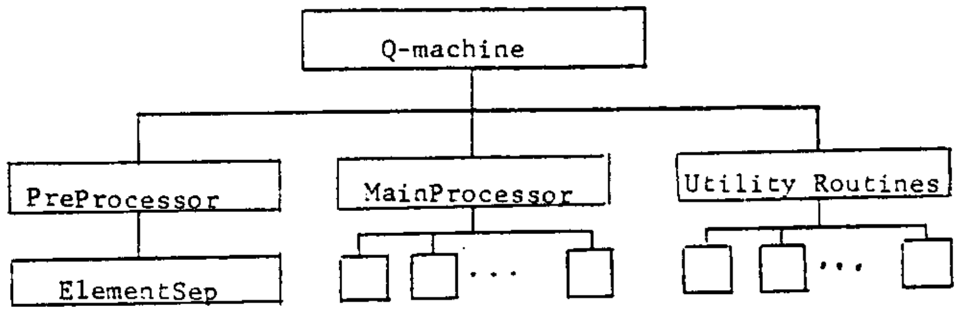


Figure 4.2 Software Organization of Q-machine Simulator

The Main-processor applies the Q-instruction passed-in by the Pre-processor to the data prepared by the Pre-processor. It then enqueues the result to the end of the D-queue. Inside Main-processor, there are 37 procedures to handle the executions of 47 FP system-defined functions.

The utility routine group handles the movements of the I-queue and the D-queue, the conversion of numbers, and the fetching of user-defined functions.

4.2 The Execution of Sample FP Programs

In this section we discuss the executions of four FP programs, DEVIATION, MATRIXMUL, QUICKSORT and TOWERHANOI. Assume one time unit is the time to execute a machine instruction. The estimated numbers of machine instructions needed for each FP system-defined function are listed in Appendix. Also assume one machine instruction is approximately equivalent to an assembly instruction in the conventional machine.

4.2.1 The Execution of DEVIATION Program

FP program DEVIATION finds the deviation of each element in the input sequence from the mean of the elements in the sequence.

FP program DEVIATION is:

```
{DEVIATION &-@distr@[id, MEAN]}
{MEAN      /@[|+, length]}
```

Its Q-instructions and estimated execution time cost are shown below. The number of elements in the sequence applied by DEVIATION is n.

Time Cost for Function MEAN							
Q-instruction	id	length	|+	[ ]	/	END	Total
Time Cost	2	1	n-1	1	1	0	n+4

Time Cost for Function DEVIATION								
Q-instruction	id	MEAN	id	[ ]	distr	&-	END	Total
Time Cost	2	n+4	2	1	n	n	0	3n+8

From the above tables, we can see that the total time cost for applying DEVIATION to a sequence consisting of n elements is 3n+8 time units.

The above time cost is based on the Q-machine with a single E-unit. If there are multiple E-units available, certain functional forms in the above DEVIATION program can be executed in parallel by multiple processors.

For the above execution, if there are as many processors as needed, the functional form "|+" (Tree Insert of Addition) in MEAN can be executed by n E-units in parallel. Assume that split time to build up the binary tree is lgn and the execution time of the additions is $\lfloor lgn \rfloor$, then the above |+ can be executed at the time cost of $2\lfloor lgn \rfloor$ instead of (n-1) time units which would be needed with a single E-unit.

With n E-units, the functional form &- (Apply to All of subtraction) in DEVIATION can be executed in one time unit instead of n time units with a single E-unit.

Using as many processors as needed to carry out the above execution of DEVIATION, the time cost is (n+2$\lfloor lgn \rfloor$ + 5). The speedup of using as many processors as needed versus using a single processor is

$$\frac{3n+8}{n + 2\lfloor lgn \rfloor + 5}$$

The following table gives the comparison and speedup of using as many processors as needed and using a single processor to apply DEVIATION to the input sequences with different number of elements.

No. of Input Elements	Time Cost Using Single Processor	Time Cost Using k Processors	Speedup	$\frac{Speedup}{k}$
8	32	19 (k=8)	1.68	0.21
16	56	29 (k=16)	1.93	0.12
32	104	47 (k=32)	2.21	0.06
64	260	81 (k=64)	3.20	0.05

If there are only k processors available for parallel executions, the time cost for applying DEVIATION to a sequence having n elements is

$$\frac{2n - 1}{k} + n + 9$$

The ratio of speedup over the number of processors is

$$\frac{3n + 8}{2n - 1 + kn + 9k}$$

A program, equivalent to DEVIATION, in conventional programming language running on a conventional machine would need (8n + 1) assembly instructions. Compared with the number of machine instructions needed for running DEVIATION on the Q-machine, the extra (5n-7) time units of running the conventional program are spent on loop controls and variable assignments, which are not needed in the FP program running on the Q-machine.

A more extensive study is needed to make a fair comparison between the performances on the Q-machine and the conventional machines.

4.2.2 The Execution of MATRIXMUL Program

The FP program MATRIXMUL carries out the multiplication of two matrices.

The FP program MATRIXMUL is:

```
{MATRIXMUL &TOTAL@&distl@MERGE }
{MERGE distr@[1,trans@2]}
{TOTAL &INNER }
{INNER !+@&*@trans }
```

Assume that the multiplicand matrix has p rows and q columns, and the multiplier matrix has q rows and w columns. The value of p, q and w could be any positive integers. The Q-instructions compiled from the above FP program and the estimate of time are shown below.

Time Cost for Function MULTIPLY						
Q-instruction	id	MERGE	&distl	&TOTAL	END	Total
Time Cost	1	7+p+qw	pw	p(4qw)	0	4pqw+pw+qw+p+8

Time Cost for Function MERGE									
Q-instruction	id	2	id	trans	1	[ ]	distr	END	Total
Time Cost	2	1	1	qw	1	2	p	0	qw+p+7

Time Cost for Function INNER						
Q-instruction	id	trans	&*	!+	END	Total
Time Cost	1	2q	q	q-1	0	4q

Time Cost for Function TOTAL			
Q-instruction	&INNER	END	Total
Time Cost	w(4q)	0	4qw

The time for multiplying two matrices, which have p rows and q columns, q rows and w columns, respectively, is

$$4pqw + qw + pw + p + 8.$$

If there are as many processors as needed, we can execute the same program in the cost of

$$qw + p + 3q + w + 9.$$

with (pqw + p) processors.

The following table gives the information about the speed up and the ratio of the speedup over the number of processors used.

p	q	w	Time Cost Using Single Processor	Time Cost Using k Processors	Speedup	$\frac{Speedup}{k}$
4	3	3	177	34 (k=40)	5.2	0.13
4	4	4	300	45 (k=68)	6.67	0.098
7	7	7	1485	86 (k=350)	17.23	0.049
10	10	10	4218	159 (k=1010)	26.53	0.026

If there are k processors available for parallel execution, the k processor will be used to carry out the multiplication of two numbers, and the time cost will be

$$\frac{4pqw}{k} + pw + qw + p + 8$$

If an equivalent program in conventional language runs on a conventional machine, the number of assembly instructions executed would be

$$5pqw + 3pq + 2p.$$

The following table gives the comparison between the numbers of time units for running the MATRIXMUL on the Q-machine and a an equivalent program in conventional language on a conventional machine.

p	q	w	Time Cost on Q-machine Using a Single Processor	Time Cost on Conventional Machine	Speedup
4	3	3	177	224	1.28
4	4	4	300	376	1.25
7	7	7	1485	1876	1.26
10	10	10	4218	5320	1.26

The extra time units for running the equivalent program in conventional language on a conventional machine are also spent on the loop controls and variable assignments.

4.2.3 The Execution of QUICKSORT Program

The FP program QUICKSORT sorts the number in the input sequence into ascending order.

The FP program QUICKSORT is listed below:

```
{ NULLORONE or@[null,ONEELEMENT]}
{ ONEELEMENT =@[length, %1]}
{ PUTSMALL (>@id->[2];[])}
{ PUTLARGE (<=@id->[2];[])}
{ SMALLLIST concat@&PUTSMALL@distl@[first,tl]}
{ LARGELIST concat@&PUTLARGE@distl@[first,tl]}
{ QUICKSORT (NULLORONE->id;concat@[QUICKSORT@SMALLLIST,
                              [first],
                            QUICKSORT@LARGELIST])}
```

The Q-instructions compiled from the above FP program and the estimate of time are shown below. Assume that the recursively invoked QUICKSORT always sorts half of the input numbers of the invoking QUICKSORT routine.

Time Cost for Function NULLORONE							
Q-instruction	id	ONEELEMENT	null	[]	or	END	Total
Time Cost	2	7	1	2	1	0	13

Time Cost for Function ONEELEMENT							
Q-instruction	id	%1	length	[]	=	END	Total
Time Cost	2	1	1	2	1	0	7

Time Cost for Function PUTSMALL						
Q-instruction	id	>	2	[]	END	Total
Time Cost	1	1	1	1	0	4

Time Cost for Function PUTLARGE						
Q-instruction	id	<	2	[]	END	Total
Time Cost	1	1	1	1	0	4

Time Cost for Function SMALLLIST (n is the number of elements in the applied sequence)					
Q-instruction	id	tl	first	[]	distl (continued)
Time Cost	2	1	1	2	n (continued)
Q-instruction	&PUTSMALL	concat	END	total	
Time Cost	4n	n	0	6n+6	

Time Cost for Function LARGELIST (n is the number of elements in the applied sequence)					
Q-instruction	id	tl	first	[]	distl (continued)
Time Cost	2	1	1	2	n (continued)
Q-instruction	&PUTLARGE	concat	END	Total	
Time Cost	4n	n	0	6n+6	

Time Cost for Function QUICKSORT (n is the number of elements in the applied sequence) (T_n is the time for sorting n numbers.)			
Q-instruction	NULLORONE	LARGELIST	first
Time Cost	13	6n+6	1
Q-instruction	SMALLIST	QUICKSORT	[]
Time Cost	6n+6	$T_{\frac{n}{2}}$	1
Q-instruction	QUICKSORT	[]	concat
Time Cost	$T_{\frac{n}{2}}$	1	n
Q-instruction	END		Total
Time Cost	0		$2X_{\frac{n}{2}}+13n+30$

The following table gives the comparison of time and the speedup of running QUICKSORT in the Q-machine with a single processor and in the Q-machine with as many processors as needed.

No. of Numbers Sorted	Time Cost Using Single Processor	Time Cost Using k Processors	Speedup	$\frac{Speedup}{k}$
4	247	84 (k=8)	2.94	0.37
8	628	134 (k=16)	4.68	0.29
16	1478	208 (k=32)	7.10	0.22
32	3402	330 (k=64)	10.30	0.16
64	7666	548 (k=128)	13.19	0.11

The time cost on the Q-machine with a single processor looks relatively high, because we have to use two comparisons to detemine whether a given number should be placed either to the side where numbers smaller than the selected number reside, or to the side where numbers which are greater than or equal to the selected number reside. However, if 2n E-units are used in parallel to sort n numbers, these comparisons can be done in parallel.

4.2.4 The Execution of TOWERHANOI Program

The FP program TOWERHANOI simulates the disk movements in the well-known "Towers of Hanoi" problem.

The FP program TOWERHANOI is:

```
{Move ( =@[1,%1] -> [-@[1,%1],-@[2,%1],3,+@[4,%1]];
        [%0,3,2,4]@Move@[2,3@1,2@1,4@1]
                @[Move@[%1,2@1,4@1,3@1],2]
                @[Move@[-@[1,%1],2.4,3],-@[1,%1]]])}
```

We simulated the movements of 5 disks. The execution takes 2025 machine instructions (1598 functions).

In [8], Turner reported the numbers of reduction steps in the executions of the "Towers of Hanoi" problem with 5 disks in both Landin's SECD and his reduction machine implementation. The concept of the reduction step is equivalent to our machine instruction. The comparison between SECD, Q-machine and Turner's Reduction Machine is

Machine	SECD	Q-machine	Reduction
Time	1488	2025	3067
Time Factor	0.75	1	1.51

The time cost in the Q-machine is between the SECD and the Reduction machine. The reason of this is that Turner's reduction machine has to execute extra reduction steps generated by the compiler, on the other hand, the number of Q-instructions running on the Q-machine is very close to the number of FP primitive functions in the original program.

CHAPTER 5

Conclusion

In this thesis, we discussed a Q-machine architecture, which executes the Q-instructions compiled from an FP program. In addition, we gave the design of the Q-machine with a single E-unit.

The proposed Q-machine with a single processor is of interest because of the following advantages:

1. The simplicity of the Q-machine organization.

2. The beneficial implicit address management of the D-queue and the I-queue.

3. The separate data memory module and Q-instruction memory module virtually double the bandwidth of memory access.

4. The simplicity of the Q-instruction compiler.

We have also explored the potential parallelism in the Q-machine with multiple E-units and in multiple Q-machines.

We have simulated the Q-machine with a single E-unit and analyzed the performance of executing several FP programs on the Q-machine simulator. Compared with the performance of executing equivalent programs in conventional languages running on conventional machines, the Q-machine has the advantages of eliminating loop control and variable assignment.

The Q-machine is not a totally new architecture. Many of its components use the concepts of conventional machines, such as queues, stacks, shift registers, memory modules, etc. Therefore, it can be built up independently, or implemented (or emulated) as an alternative version in conventional machines, so that FP programs can run on conventional machines without losing efficiency.

The communication networks between multiple E-units and between multiple Q-machines will of course require further research to detemine the overhaed of the possible parallel executions, but the advantages of executing FP programs in the Q-machine in parallel are obvious.

An even more efficient Q-instruction compiler can possibly be generated through further study. Moreover, a challenging potential is brought to light through the generation of the Q-instruction compiler: the possibility of writing a Q-instruction compiler in FP, thereby enabling the Q-machine itself to compile FP programs into Q-instructions.

APPENDIX

Berkeley FP Primitive Functions

Selector Functions

$\mu : x \equiv$
$x = <x_1,x_2,...,x_k> \bigwedge 0 < u \leq k \rightarrow x_\mu;$
$x = <x_1,x_2,...,x_k> \bigwedge -k \leq \mu < 0 \rightarrow x_{k+\mu+1};$?

Estimated machine instructions : μ.

pick : $<n,x> \equiv$
$x = <x_1,x_2,...,x_k> \bigwedge 0 < n \leq k \rightarrow x_n;$
$x = <x_1,x_2,...,x_k> \bigwedge -k \leq n < 0 \rightarrow x_{k+n+1};$?

Estimated machine instructions : n.

last : $x \equiv$
$x = <> \rightarrow <>;$
$x = <x_1,x_2,...,x_k> \bigwedge k \geq 1 \rightarrow x_k;$?

Estimated machine instructions : k.

first : $x \equiv$
$x = <> \rightarrow <>;$
$x = <x_1,x_2,...,x_k> \bigwedge k \geq 1 \rightarrow x_1;$?

Estimated machine instruction : 1.

Tail Functions

tl : $x \equiv$
$x = <x_1> \rightarrow <>;$
$x = <x_1,x_2,...,x_k> \bigwedge k \geq 2 \rightarrow <x_2,...,x_k>$; ?

Estimated machine instructions : 2.

tlr : $x \equiv$

$x = <x_1> \rightarrow <>;$
$x = <x_1,x_2,...,x_k> \wedge k \geq 2 \rightarrow <x_1,...,x_{k-1}>$; ?

Estimated machine instructions : 2.

Distribute From Left and Right

distl : $x \equiv$

$x = <y,<>>$ -> $<>$;
$x = <y,<z_1,z_2,...,z_k>> \rightarrow <<y,z_1>,...,<y,z_k>>$; ?

Estimated machine instructions : k.

distr : $x \equiv$

$x = <<>,z>$ -> $<>$;
$x = <<y_1,y_2,...,y_k>,z> \rightarrow <<y_1,z>,...,<y_k,z>>$; ?

Estimated machine instructions : k.

Identity

id : x $\equiv$ x

Estimated machine instruction : 1.

out : x $\equiv$ x

Estimated machine instruction : 1.

Append Left and Right

apndl : $x \equiv$

$x = <y,<>> \rightarrow <y>$;
$x = <y,<z_1,z_2,...,z_k>> \rightarrow <y,z_1,z_2,...,z_k>$; ?

Estimated machine instruction : 2.

apndr : $x \equiv$

$x = <<>,z> -\!> <z>$;

$x = <<y_1,y_2,...,y_k>,z> \rightarrow <y_1,y_2,...,y_k,z>$; ?

Estimated machine instructions : k.

Transpose

trans : $x \equiv$

$x = <<>,...,<>> -\!> <>$;

$x = <x_1,x_2,...,x_k> \rightarrow <y_1,...,y_m>$; ?

where $x_i = <x_{i1},...,x_{im}> \bigwedge y_j = <x_{1j},...,x_{kj}>$,

$1 \leq i \leq k,\ 1 \leq j \leq m$.

Estimated machine instructions : $k \times m$.

reverse : $x \equiv$

$x = <> \rightarrow <>$;

$x = <x_1,x_2,...,x_k> \rightarrow <x_k,...,x_1>$; ?

Estimated machine instructions : k.

Rotate Left and Right

rotl : $x \equiv$

$x = <> \rightarrow <>$;

$x = <x_1> \rightarrow <x_1>$;

$x = <x_1,x_2,...,x_k> \bigwedge k \geq 2 \rightarrow <x_2,...,x_k,x_1>$; ?

Estimated machine instructions : k.

rotr : $x \equiv$

$x = <> \rightarrow <>$;

$x = <x_1> -\!> <x_1>$;

$x = <x_1,x_2,...,x_k> \bigwedge k \geq 2 \rightarrow <x_k,x_1,...,x_{k-1}>$; ?

Estimated machine instructions : k.

concat : $x \equiv$

$x = <<x_{11},...,x_{1k}>,<x_{21},...,x_{2n}>,...,<x_{m1},...,x_{mp}>>$

$\bigwedge$ k,m,n,p > 0

$\rightarrow <x_{11},...,x_{1k},x_{21},...,x_{2n},x_{m1},...,x_{mp}>$; ?

Estimated machine instructions : k+n+...+p.

pair : $x \equiv$

$x = <x_1,x_2,...,x_k> \bigwedge k>0 \bigwedge k$ is even

$\rightarrow << x_1,x_2>,...,<x_{k-1},x_k>>;?$

$x = <x_1,x_2,...,x_k> \bigwedge k>0 \bigwedge k$ is odd

$\rightarrow << x_1,x_2>,...,<x_k>>; ?$

Estimated machine instructions : k.

split : $x \equiv$

$x = <x_1> \text{ -> } <<x_1>,<>>;$

$x = <x_1,x_2,...,x_k> \bigwedge k>1$

$\rightarrow << x_1,...,x_{\frac{k}{2}}>,<x_{\frac{k}{2}+1},...,x_k>>; ?$

Estimated machine instructions : $\frac{k}{2}$.

iota : $x \equiv$

$x = 0 \rightarrow <>;$

$x \in N^+ \rightarrow <1,2,...,x>; ?$

Estimated machine instructions : x.

Predicate (Test) Functions

atom : $x \equiv$

$x \in$ atoms $\rightarrow$ T;

$x = ? \rightarrow$ F; ?

Estimated machine instruction : 1.

eq : $x \equiv$

$x = <y,z> \bigwedge y = z \rightarrow$ T;

$x = <y,z> \bigwedge y \neq z \rightarrow$ F; ?

Also less than (<), greater than (>), greater or equal (>=), less than or equal (<=), not equal (=); '=' is a synonym for eq.

Estimated machine instruction : 1.

null : $x \equiv$
$x = <> \rightarrow$ T;
$x \neq$? $\rightarrow$ F; ?

Estimated machine instruction : 1.

length : $x \equiv$
$x = <x_1, x_2, \ldots, x_k> \rightarrow$ k;
$x = <> \rightarrow$ 0; ?

Estimated machine instruction : 1.

Arithmetic and Logical

+ : $x \equiv$
$x = <$y,z$>$ $\wedge$ y,z are numbers $\rightarrow$ y+z; ?

Estimated machine instruction : 1.

- : $x \equiv$
$x = <$y,z$>$ $\wedge$ y,z are numbers $\rightarrow$ y-z; ?

Estimated machine instruction : 1.

* : $x \equiv$
$x = <$y,z$>$ $\wedge$ y,z are numbers $\rightarrow$ y*z; ?

Estimated machine instruction : 1.

/ : $x \equiv$
$x = <$y,z$>$ $\wedge$ y,z are numbers $\wedge z \neq 0 \rightarrow$ y/z; ?

Estimated machine instruction : 1.

And, Or, Not and Xor

and : $<x,y> \equiv$
$x = \text{T} \rightarrow \text{y};$
$x = \text{F} \rightarrow \text{F}; ?$

Estimated machine instruction : 1.

or : $<x,y> \equiv$
$x = \text{F} \rightarrow \text{y};$
$x = \text{T} \rightarrow \text{T}; ?$

Estimated machine instruction : 1.

xor : $<x,y> \equiv$
$x = \text{T} \wedge \text{y} = \text{T} \rightarrow \text{F};$
$x = \text{F} \wedge \text{y} = \text{F} \rightarrow \text{F};$
$x = \text{T} \wedge \text{y} = \text{F} \rightarrow \text{T};$
$x = \text{F} \wedge \text{y} = \text{T} \rightarrow \text{T}; ?$

Estimated machine instruction : 1.

not : $x \equiv$
$x = \text{T} \rightarrow \text{F};$
$x = \text{F} \rightarrow \text{T}; ?$

Estimated machine instruction : 1.

Library Functions

sin : x $\equiv$
x is a number $\rightarrow$ sin(x); ?

Estimated machine instruction : 1.

asin : x $\equiv$
x is a number $\wedge |x| \leq 1 \rightarrow \sin^{-1}(x); ?$

Estimated machine instruction : 1.

cos : x ≡

x is a number → cos(x); ?

Estimated machine instruction : 1.

acos : x ≡

x is a number $\wedge$ $|x| \leq 1$ → $\cos^{-1}(x)$; ?

Estimated machine instruction : 1.

exp : x ≡

x is a number → e^x; ?

Estimated machine instruction : 1.

log : x ≡

x is a positive number → ln(x); ?

Estimated machine instruction : 1.

mod : <x,y> ≡

x and y are numbers → $x - y \times \left\lfloor \frac{x}{y} \right\rfloor$; ?

Estimated machine instruction : 1.

References

[1] Backus, J., "Can Programming Be Liberated from the Von Neuman Style? A Functional Style and Its Algebra of Programs," CACM 21:8 (August 1978), 613-641.

[2] Baden, S., "Berkeley FP User's Manual. Rev. 4.1," (March 1983).

[3] Blikle, A. J., "Investigation in the Theory of Addressless Computers," Polska Akademia Nauk (Serie des science math., astr., et phys.) 14:14 (April 1966), 203-208.

[4] Pawlak, Z., "New Class of Mathematical Languages and Organization of Addressless Computers," (in) Colloquium on the Foundation of Mathematics, Budapest: Akademiai Kiado, 1965, 1966, 227-238.

[5] Feller, M., and Ercegovac, M. D., "The Queue Machine: An Organization for Parallel Computation,", Report (1980) of Computer Science Department, UCLA.

[6] Feller, M., "A Parallel Queue Organization for High-Speed Computing," (thesis) UCLA, Los Angeles, California, 1980.

[7] Lang, T., "Interconnections Between Processors and Memory Modules Using the Shuffle-Exchange Network," IEEE Trans. Computer, C-25:5 (May 1976), 496-503.

[8] Turner, D. A., "A New Implementation Technique for Applicative Languages," Software, Practice and Experience, vol. 9, 1979.

[9] William, J., "Notes on the FP Style of Functional Programming", Functional Programming and Its Applications, eds. Darlington, Henderson, and Turner, Cambridge University Press, Cambridge, 1982.

[10] Ercegovac, M. D., Patel, D. R. and Lang, T., "Functional Language and Data Flow Architecture," Proc. Summer Computer Simulation Conference, Vol.2, 1983, pp. 1007-1023.

[11] Knuth, Donald E., "The Art of Compuetter Programming "," Volume 1, Addison -Wesley Publishing Company, 1973, 412-413.