A Type Driven Hardware Engine for Prolog Clause Retrieval over a Large Knowledge Base

Kam-Fai Wong[†] *and M. Howard Williams*[††]

† Software Engineering Department, Unisys Ltd., Livingston, EH54 7AZ, Scotland.

†† Dept. of Computer Science, Heriot-Watt University, Edinburgh EH1 2HJ, Scotland.

Abstract

Whereas existing Prolog systems are very effective at handling small knowledge bases, they are not very efficient at and often incapable of handling large sets of clauses. Large knowledge bases which may comprise millions of clauses and are shared by a number of users, may need to reside in secondary memory. In such cases exhaustive search is inordinately slow. Various approaches have been put forward for handling the problem, most of which involve *coupled* systems (loosely or tightly coupled). A Prolog data/knowledge based system which provides an *integrated* solution to the problem is being developed. An essential element in this system is the CLAuse Retrieval Engine, CLARE, which is a special purpose hardware engine designed to perform selective retrieval of data from disk in order to identify all potential clauses which will be required for full unification during a query. The engine consists of two separate hardware components, which together form a two-stage filtering configuration. This paper concentrates on the second stage filter which is concerned with partial test unification.

1. Introduction

One of the important application areas of Prolog is knowledge base implementation. Most existing systems for handling large knowledge bases [3,4] are constructed by *coupling*[1] a Prolog translator to a relational database manager. In a coupled knowledge based system, clauses are usually stored in one of two different databases : an Extensional Database (EDB) consisting of ground facts equivalent to tuples in a relational database and an Intensional Database (IDB) consisting of non-ground facts and rules. Such a system is usually based on the following assumptions :

(1) The EDB constitutes the bulk of the knowledge base.

(2) The IDB is small enough to fit into main memory.

(3) Mixed relations consisting of both ground facts and non-ground facts or rules do not occur.

An exception to this is the EDUCE system [1] which stores facts and rules in a relational database which is tightly coupled to a Prolog system. One way to increase the performance of a coupled system is to employ a database machine to provide for rapid searching of disk files. The major role of the database machine is to expedite the data retrieval process by matching the argument terms of the disk resident facts against the argument terms of a query on-the-fly. Term matching in conventional database machines is based on equality testing which is a sufficient test only in the case of clauses which are ground facts whose arguments are purely atomic.

Equality testing is, however, inadequate for handling rules and non-ground facts which may contain variables and possibly also complex structures as their arguments. Evaluation of a Prolog query is based on *resolution* which invokes *unification* - a complex and time consuming term matching process. Compounding this with a low disk transfer rate, unification applied to a large set of disk resident rules would be impractically slow. For this reason, the decision is usually taken to separate rules and non-ground facts and store them separately in the IDB under the management of a Prolog system. On the other hand, retaining the IDB in main memory incurs a serious size restriction to the entire knowledge based system.

Another problem in a coupled knowledge base arises from the fact that in the Prolog system there is an ordering

associated with clauses which is used to effect the user in writing Prolog programs, whereas in the database part the ordering of clauses is determined by the system. For this reason mixed relations cannot be catered for in a uniform way and are usually disallowed. This difference in philosophy is quite acceptable for some applications but is a serious problem for a general purpose knowledge base.

In the future an increasing amount of semantic information will be included in knowledge based systems. For example, medium-size knowledge based systems are envisaged by D.H.D. Warren to be ".... of the order of 3000 predicates, 30000 rules, 3000000 facts, and 30 Mbytes total size" [5]. Clauses with rules and structures will not be uncommon. Rules and facts will coexist in the same file. Ordering of clauses, which contributes to the interpretation of the knowledge based system, will be important. For these reasons, the idea of coupling a Prolog system to a relational database will be inadequate, and purpose built systems will be required.

One way of providing a knowledge based system suitable for the future is to adopt the *integrated* implementation approach. In an integrated system, users have a uniform view of the knowledge base : rules and facts of a predicate are kept together in a user-specified order and they are managed by a single Prolog system. Compared with a coupled knowledge base, an integrated knowledge base is more flexible and can be engaged in a much wider scope of applications. However, existing systems are restricted to small applications due to their inefficient way of handling disk resident clauses†. One of the chief problems with Prolog is that when it searches for a clause, it uses the process of unification to match the goal against each clause in turn. This is a complicated process of matching functor and arguments according to certain rules, and takes up a considerable proportion of query processing time. For a large knowledge base stored on disk, the time taken to perform this could well be intolerable.

In order to provide for high-speed retrieval of clauses from large clause sets stored on disk, a special purpose Prolog data/knowledge base machine, CLARE - CLAuse Retrieval Engine, has been designed and is under development. A two-stage filtering process is used by CLARE which employs two pieces of dedicated hardware to search disk resident data on-the-fly. The first stage filter (FS1) is a hardware realisation of index searching based on *superimposed codeword plus mask bits* [2], while the second stage filter (FS2) is responsible for *partial test unification*. In the next section, an overview of the Prolog Data Base Machine (PDBM) project - a project which is aimed at developing the type of system to handle large knowledge bases - is outlined. The architecture of the second stage filter (FS2) of the CLARE hardware is described in detail in section three which also includes the timing calculations of the various hardware operations supported by the FS2.

† A set of benchmarks have been devised for evaluating the efficiency of various Prolog systems in handling data bases of different sizes [6]. The benchmark programs were run on a Sun3/160 workstation, equipped with 4MB of main memory, and revealed that the Prolog systems were unable to cope with more than about 60k clauses and even then the overhead of loading these clauses into main memory was very high [7].

2. The PDBM Project - A Brief Overview

The aim of the Prolog Data Base Machine (PDBM) project is to produce a Prolog system which will handle large sets of clauses efficiently [8]. This is achieved by a combination of software and hardware development. The software component is based on a C version of Prolog-X. Prolog-X is a Prolog compiler originally developed by Clocksin[9] and which now runs on ICL3900 and SUN systems. It uses Edinburgh Prolog syntax but with extension to handle modules. Using Prolog-X, clauses are compiled and stored in modules, each module containing one or more procedures. Modules are then classified into two types depending on their size, viz small modules which are loaded into main memory when required, and large modules which are disk resident.

The hardware component of the system consists of a special purpose Clause Retrieval Engine (CLARE) which is designed to increase the performance of the system in retrieval over large clause sets. CLARE selectively retrieves clauses on the fly from a disk. As clauses stream from the disk, they are subjected to a two stage filtering process, as follows :

2.1. Superimposed Code Word plus Mask Bits (SCW+MB)

Predicates with the same functor names and arities are stored in a compiled clause file. For fast searching in large files, codewords are generated for facts and rule heads and these are maintained in a secondary file. The secondary file is effectively an index table associating codewords with clause addresses. When a query is given, the first stage filtering hardware scans through the secondary file and extracts the addresses of those clauses whose codewords successfully match with the codeword corresponding to that query. The size of a secondary file is generally much smaller than that of a compiled clause file [10], thereby enabling quicker retrieval to be achieved by scanning the former than by searching the latter exhaustively. The superimposed codeword plus mask bits indexing scheme (SCW+MB) [11] is an extension of the basic superimposed codeword method which takes account of complex structures and variables which arise in Prolog clauses by the use additional field comprising a set of mask bits. Index matching is performed in parallel, using standard PLAs and MSI components [2], in the first stage of the CLARE engine.

Indexing via SCW+MB is a partial matching technique. Clauses satisfying the matching process are only potential unifiers for a query. Some of these satisfiers are false drops (or 'ghosts') and will fail at the subsequent full unification. False drops are derivable from two main sources [12]:

(1) *Non-unique encoding* in which the same codeword is shared by one or more clauses.

(2) *Restrictive codeword representation* which is caused by truncation of clauses with arities larger than the allowable hardware limit. (In the present system, only 12 arguments of a query is encoded).

Another problem which cannot be handled by the superimposed codeword scheme is that of shared variables. Variables are ignored in the encoding process. This would create no problem if each variable only occurs once in a predicate. However, if a variable occurs more than once in a predicate, a large proportion of false drops could be generated. For example, the query married_couple(Same_surname,Same_surname) would result in the retrieval of the entire predicate from the knowledge base, whilst in reality the resolution set should be very small.

2.2. Partial Test Unification

Further refinement of the clauses which satisfy the first stage is provided by partial test unification using the second stage hardware (FS2). Using the clause references from the secondary index file, compiled clauses are extracted from the knowledge base. Facts and rule heads are compiled into pseudo in-line formats (PIF) ready for partial test unification. In the PIF format, an argument is represented by an 8 bit type tag followed by a 24 or 32 bit content field with an optional 32 bit extension. At present, 107 data types are supported under the PIF format, see appendix 1. The FS2 filter matches clause arguments in PIF against those of the query following a set algorithm. Five levels of matching have been investigated which depends on the depth of the two terms being compared:

- *Level 1* - type only.
- *Level 2* - type and content, ignoring complex structures.
- *Level 3* - type and content, catering for first level structures.
- *Level 4* - type and content, including full structures.
- *Level 5* - type, content with full structures and variable cross binding checks†.

Since the cost and complexity of the matching hardware to cater for levels four and five are high, a level three partial test unification algorithm is being adopted. In addition to the original algorithm, variable cross binding checks have been incorporated. A simplified version of the matching algorithm is shown in figure 1. After the second stage, the percentage of false drops will be reduced significantly, resulting in a manageable clause set for full unification.

An independent software module, the Clause Retrieval Server (CRS), is being developed which links CLARE with the PDBM Prolog system. In practice, there will be four searching modes during a clause retrieval:

(a) By software only - the CRS performs all the search operations itself.

(b) Using FS1 only - the superimposed codeword hardware.

(c) Using FS2 only - the partial test unification hardware.

(d) Using both FS1 and FS2 - a two-stage hardware filter.

One of these modes will be selected depending on the nature of a query (e.g. whether it contains cross bound variables) and the knowledge base (e.g. whether it is rule or fact intensive). The CRS will also support simultaneous access by multiple clients which involves procedures for concurrency control and transaction handling.

At the hardware level, CLARE is incorporated in a SUN3/160 workstation. The communication between CLARE and the host takes place via the VMEbus interface. CLARE is memory mapped into the /dev/vme24d16, SUN's user space, using the mmap() system call [13]. Both filtering stages, FS1 and FS2, appear in the form of plug-in circuit boards. A common address space from ffff7e00(hex) to ffff7fff(hex) - 128k bytes in total - is shared by FS1 and FS2. The two filters are mutually exclusive. The selection between the two is governed by the third least significant bit, b_2, of an 8-bit control register - A 0 in b_2 selects FS1 and a 1 selects FS2.

3. The Second Stage Hardware (FS2)

Once a filtering stage is enabled, the setting of the first two significant bits of its control register determines the operation mode of the hardware. FS2 can be set to work in one of 4 operational modes :

Operational Mode	b0	b1
Read Result	0	0
Search	0	1
Microprogramming	1	0
Set Query	1	1

When a query is posed, it is translated into microprogram instructions. These instructions are loaded into the FS2 while it is set to Microprogramming mode. The FS2 is then switched to Set Query mode and all query arguments are written into the FS2 query memory. After performing the two operations, the FS2 is ready to perform a search. The DMA begin and end addresses of the disk transfer command block, where retrieved data will be placed, is specified to be the FS2 address space. A search is then initiated by setting the FS2 to Search mode. From then on the FS2 is, effectively, connected to the disk controller. At the end of the search, if a match has been found, the most significant bit (b_7) of the control register will be set. At this stage, in order to extract the potential answers to the query, the FS2 will be set to Read Result mode.

The overall architecture of the FS2 hardware is depicted in figure 2. It consists of four functional units. The Writable Control Store (WCS) provides the microprogram instructions to coordinate all the other units. Data streaming from the disk is retained in one of the two memory banks in the Double

† A variable cross binding occurs when a database variable is bound to a query variable. Checking is essential to ensure that the cross binding relationship is consistent i.e. the variables are not instantiated to different terms. Referring to superimposed codeword index matching, cross binding is in fact an alternative form of shared variable and has the effect of creating undesirable false drops.

Buffer (DB) while data stored previously in the other bank are subjected to partial test unification by the Test Unification Engine (TUE). The results of the test will decide which clause is to be captured in the Result Memory (RM).

3.1. The Writable Control Store (WCS)

The construction of the WCS, figure 3, is based extensively on AMD (Advanced Micro Devices, Inc) bit-slice components. The WCS consists of a bank of fast bipolar RAM which holds the microprogram instruction for coordinating the overall FS2 hardware during a query. A *one level pipeline based architecture* [14] is adopted by the WCS - the microprogram instruction appears one instruction ahead of the microprogram memory address. The RAM can hold a maximum of 2048 microprogram instructions, each 64 bits wide. Normally, the fast RAM is read only and is addressed by the Micro Program Controller (MPC). When the FS2 is operating in Microprogramming mode, the fast RAM is connected to the VMEbus. It will appear as normal memory to the SUN host and loading of microprograms is enabled. Two counters, one for the database input and one for the query, are used to keep track of the number of elements remaining during the matching of lists and structures. An 8 MHz clock is used to synchronise the various parts in the WCS.

At the beginning of a retrieval and after matching a clause, the MPC is engaged in a polling routine This routine repeatedly monitors the zeroth bit of the conditional code (CC) which is asserted when a new clause has been read into the Double Buffer and is ready to be examined. The output of the MPC which is the address in the fast RAM, can derive either from the MPC's internal counter or externally from the branch address field of the microprogram instructions when a branch or jump instruction occurs. Another external source comes from the output of the Map ROM. The Map ROM stores a list of jump vectors and its address port is connected to the db-data and Q-data bus from the Test Unification Engine. Only the type fields of the db-data and Q-data are effective. Depending on the combination of the type fields, different microprogram routines are invoked. The data types supported by the FS2 are classified into 3 categories (see appendix 1). Each category is handled differently :

Simple terms require simple matching.

Atoms, integers and floating point numbers are included in this category. Apart from the case of an integer constant (integer in-line) in which the content field is the value of the integer itself, the content fields of these data items are symbol table references. Equality is the testing condition when two arguments are compared against each other in this type.

Variable terms may require skipping, storing or fetch then match operation.

There are five variable types. An anonymous variable is effectively a don't care object and it causes a skip in the matching process. When an anonymous variable is encountered on either side during a match, the match will succeed immediately irrespective of the type of the other side. The other variable types account for the origin of a variable - whether it occurs in a query (QV) or in a data/knowledge base (DV). QV and DV are further divided into two groups in order to record their modes of occurrence. For example, 1st-QV will be the type of a variable when it is being referred to in the first time during the partial test unification process; if the same variable occurs again in the same query, it will be typed as Sub-QV (subsequent query variable). At compile time, the subsequent occurrences and the first occurrence of a variable have the same content field which is a pointer to the internal memory of the Test Unification Engine. First occurrences of QVs and DVs will invoke an instantiation routine which stores, respectively, the database and the query argument terms in the Q-Memory and DB-Memory of the Test Unification Engine (see later). QVs and DVs which occurred subsequently are handled specially. It involves two microprogram instruction cycles : firstly, the location of the internal memory of the Test Unification Engine, referenced to by the content field of the variable, is read; and secondly, that value is subject to partial test unification.

Complex terms require repetitive matching

A complex term can be a list or a structure. In-line complex terms can contain one to 32 constituent terms. During the matching process, the arities of the in-line complex terms of both the database and query (which need to be the same) will be loaded into respective counters. The matching process will be performed upon each constituent term pair. At the end of a comparison, both counters are decremented. This process will be repeated until the counters reach zero. Structures and lists with arities greater than 32 are represented, respectively, by structure and list pointers. Comparison of these types is identical to simple term matching. As a subset of lists, unlimited lists are defined. They are lists which contain a tail variable, e.g. [a,b|Tail]. The arities of the terms being compared may not be equal in this case. The arities are loaded into two counters and matching is repetitively carried out until the value of either counter is zero.

3.2. The Double Buffer and the Result Memory

The Double Buffer is separated physically into two identical halves. Each half consists of a bank of memory and an output register. During a search, one half is selected for input while the other is selected for output, and these roles alternated whenever the input is filled. The half which is selected for input accepts the data streaming in from disk. The data from the output register is the clause stored in the corresponding memory in the previous cycle. The output data is fed into the Test Unification Engine for matching with the pre-loaded query data. Selection control is basically generated by a toggle flip flop. Both inverting and non-inverting outputs of the flip flop are utilised in order to produce two non-overlapping clocks. The clock period is variable and is equal to the time taken for the Double Buffer to read in 2 clauses.

The Result Memory has a capacity of 32K bytes which is large enough to contain all clause satisfiers of one disk track - the worst case of a single FS2 search call. While disk data is transferring to the Double Buffer, a copy of the data is written into the Result Memory in parallel. The address of the Result Memory is derived from the Address Generator. The Address Generator consists of two counters. One is 6 bits wide and contributes to the upper 6 bits of the Result Memory address (A_9-A_{14}). This counter is incremented whenever a clause satisfier is found. The value of this counter at the end of a retrieval indicates the number of clause satisfiers. The other counter is 9 bits wide and forms the lower 9 bits of the Result Memory address (A_0-A_8). The second counter is always reset to zero after a clause has been examined. The schematic block diagram depicting the Double Buffer and the Result Memory is shown in figure 4.

3.3. The Test Unification Engine (TUE)

The Test Unification Engine (TUE), figure 5, consists of two banks of memory (DB Memory & Query Memory) a comparator (Comp), three registers (Reg1-3) and 6 selectors (Sel1-6). The DB Memory is a dual port memory which is used for storing bindings of database variables at run time. It is reset to pointing to itself at the beginning of each clause input. The Query Memory is pre-loaded at query time and it contains the query argument terms. The Comparator (Comp) is a standard ALS 8-bit comparator which generates a 1-bit HIT signal according to the values of its A and B inputs. The A and B inputs correspond to the data/knowledge base and query argument terms respectively. HIT is fed into the conditional code register (CC) of the WCS which will select of the next microroutine. When the FS2 is in Set Query mode, only the Query Memory is enabled. The right hand branches of Sel4 are selected to provide, respectively, the data and address to the Query Memory. The Query Memory is directly connected to the VMEbus interface and appears to the host as ordinary memory.

While the FS2 is in Search mode, operation of the TUE is controlled by microcode. Depending on the types of the terms being compared, different microprogram routines are invoked. The main objective of the microprogram routines is to ensure that the right data appears at the memory and the comparator. This is done by selecting the appropriate paths of the six selectors and enabling the appropriate registers. Seven basic hardware operations are provided in realising the partial test unification algorithm listed in figure 1.

3.3.1. MATCH

This is the simplest operation which is applied when the data/knowledge base and the query arguments are integers, atoms, floating point numbers, structures or lists (corresponding to cases 1 to 4 in the algorithm shown in figure 1). The routes in which query and knowledge base terms are passed to the Comparator are shown by the thick dotted lines in figure 6. Whenever a hardware operation is invoked, data/knowledge base and query arguments travel in two separate routes in parallel. The routes taken during the MATCH operation are :

(1) The database route originates from the Double Buffer, along the In-bus, through the left branch of Sel1 and finally appears at the A-port of the Comparator. From the timing specifications of individual devices, the time taken for data to appear at the A-port is calculated to be 40ns (see bottom of figure 6).

(2) The query route starts from the left branch of Sel6, from which micro bits 13-20 provide the address of the Query Memory. Data derived from the Query Memory enters the right branch of Sel3 and ends up at the B-port. The estimated propagation time is 75ns.

It takes 30ns for the comparator to generate a result. Comparison can not take place until both A and B inputs are ready. Therefore, although information travels on both routes in parallel, the longest routing time of the two should be taken in calculating the execution time of the MATCH operation. This gives an execution time of 105ns (75+30).

3.3.2. DB_STORE

This operation is applied when the data/knowledge base argument's type is first occurrence variable (1st-DV) - corresponding to case 5a of figure 1. The aim is to store the query argument in the location of the DB Memory which is addressed by the content field of the data/knowledge base argument. Figure 7 shows the routes taken by the database and query arguments. On the database side, data is extracted from the Double Buffer; it streams via the In-bus to the left branch of Sel1. Travelling via the left branch of Sel2, the data appears on the A address port of the dual ported DB Memory. There is only one input port in the DB Memory and it is connected to the output of Reg3. The content of Reg3 is taken from the output of the Query Memory whose location is specified by the left branch of Sel6. The time taken to execute this operation is 95ns.

3.3.3. QUERY_STORE

This operation is applied when the query type is a first occurrence variable (1st-QV) - case 6a of figure 1. The objective is to store the data/knowledge base argument in the location of the Query Memory which is addressed by the content field of the query argument. Referring to figure 8, the database route travels from the Double Buffer through the left branch of Sel1, then through the right branch of Sel5 and passes the left branch of Sel4 to appear at the input port of the Query Memory. The route which the query data takes is derived from the left branch of Sel6 and terminates at the address port of the Query Memory. The total execution time for this operation is 115ns.

3.3.4. DB_FETCH

When a subsequently occurring data/knowledge base variable (Sub-DV) is encountered during a search - case 5b of

figure 1, the DB Memory is accessed at the B-port in order to extract the binding of the variable for subsequent comparison. The routes taken by the data/knowledge base and the query arguments in this operation are depicted in figure 9. The database route starts from the Double Buffer which appears at the B address port of the DB Memory. The corresponding data is extracted from the DB Memory and it travels via the right branch of Sel1 to appear at the A-port of the Comparator. The query route is identical to the query route taken by the simple MATCH operation. The total execution time for this process is 105ns.

3.3.5. QUERY_FETCH

This operation is applied when the query type is a subsequently occurring variable (Sub-QV) - case 6b of figure 1. It is slightly more complicated to implement and requires two cycles for completion. In the first cycle, the query route (from the left branch of Sel6 → output of the Query Memory → right branch of Sel3 → right branch of Sel2 → A address port of DB Memory) provides the address to the A port of the DB Memory, figure 10. From the DB Memory the binding associated with the query variable is extracted and appears at the B-port of the Comparator via the left branch of Sel3 in cycle two. Running concurrently with the first and second cycles, the data/knowledge base argument is placed at the A-port of the Comparator. The time taken to execute this operation is 170ns.

3.3.6. DB_CROSS_BOUND_FETCH

DB_CROSS_BOUND_FETCH is required in situations when a data/knowledge base variable is initially bound to a query variable and the former is being used again in the same clause - cross binding (case 5c of figure 1). For example, it will be called for the second occurrence of the data/knowledge base variable A if the query f(X,a,b) is matched against the clause f(A,a,A). In the query side, data takes the simple route, i.e. ub13-20→left of Sel6→ output of Query Memory→right of Sel3→ A-port of Comp. Two cycles are required to establish the database route. In the first cycle, figure 11a, the data from the Double Buffer appears on the B address port of the DB Memory. From the DB Memory B data port, the reference to the cross bound variable is extracted and stored in Reg1. This reference is placed at the B address port in the second cycle, figure 11b. From the DB Memory the ultimate association is extracted and placed at the A-port of the Comparator via the right branch of Sel1. The total execution time is calculated to be 170ns.

3.3.7. QUERY_CROSS_BOUND_FETCH

The route taken by this operation is shown in figure 12. It is invoked when a query variable is initially bound to a data/knowledge base variable and the former is being used again in the same clause (case 6c of figure 1). This is the most complicated operation and requires 3 microprogram cycles. In the first cycle, the A-port of the Comparator is set up via the database route (Double Buffer→left of Sel1) and concurrently, the A port of the DB Memory is addressed by the data presented on the query route (i.e. ub13-20→left of Sel6→Query Memory→right of Sel3→right of Sel2). In the second cycle the query variable binding is extracted from the DB Memory A data port. This binding is recycled and is put back at the A address port (via left of Sel2→right of Sel2). The ultimate value for comparison is routed to the B-port of the comparator (via left of Sel3). The time required to perform the operation is calculated to be 235ns.

4. Conclusion

A special purpose hardware engine, CLARE, is being developed which will provide Prolog with a rapid clause retrieval rate in dealing with large data sets. CLARE consists of two hardware components and together they form a 2-stage filtering architecture. The hardware is designed to be linked to a M68020 host, via the VMEbus interface, in a SUN3/160 workstation. The prototype of the first stage filter (FS1) hardware has been fully developed. It can search data at a rate of up to 4.5Mbyte/sec. The design of the second stage filter (FS2) is complete and the partial test unification algorithm has been verified. The construction of the FS2 is well under way. Once the CLARE hardware is fully developed, it will be subjected to benchmark tests similar to the ones devised in [7].

Execution times for the various FS2 hardware operations are calculated and they are summarised in table 1. From the table it can be seen that, QUERY_CROSS_BOUND_FETCH is the most time consuming operation which takes 235ns. This gives the worst case execution rate of the FS2 to be approximately 4.25Mbytes/second. The SUN3/160 workstation, the target platform of CLARE, can be mounted with either a SCSI based disk system, e.g. Micropolis 1325, or a SMD+ based disk system, e.g. Futjitsu M2351A. Even if the second type of disk (which offers a faster transfer rate) is used and it is tuned to operate at its peak rate (circa 2Mbytes/second), the FS2 hardware can still filter disk data at a faster rate than it can be delivered from the disk.

Acknowledgement

The material described herein is based on work carried out in the Prolog Database Machine project. This project, is a collaborative project involving the Department of Computer Science, Heriot-Watt University and International Computers Ltd (ICL, UK), and is funded by the Science and Engineering Research Council (SERC) and the Department of Trade and Industry (DTI) of the United Kingdom, under the Alvey initiative.

Reference

[1] Bocca, J., "EDUCE A Marriage of Convenience: Prolog and a Relational DBMS", Internal report KB-9, European Computer-Industry Research Centre, Munich, Sept.'85.

[2] Wong, K.F. & Williams, M.H. "Partial Matching Hardware for Clause Retrieval in Large Prolog Databases", Technical Report 88/2, Dept. of Computer Science, Heriot-Watt University, Edinburgh, Scotland, 1988.

[3] Li, D., A Prolog Database System, Research Studies Press, London 1984.

[4] Berra, P.B., Chung, S.M. & Hachem N.I., "Computer Architecture for a Surrogate File to a Very Large Data/Knowledge Base", IEEE Computer, pp25-32, March'87.

[5] Warren, D.H.D., "Logic Programming and Knowledge Bases", Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems, pp69-72, Islamorada, Florida, February'85.

[6] Williams,M.H., Massey, P.A. & Crammond, J.A., "Benchmarks for Prolog from a Database Viewpoint", Proceedings of 2nd Alvey Workshop for SIGKME, ppC1-C8, Brunel University, England, 1987.

[7] Williams,M.H., Massey, P.A. & Crammond, J.A., "Benchmarking Prolog for Database Applications", Prolog and Databases: Implementations and New Directions, Ellis Horwood Series in AI, edited by P.M.D. Gray and R.J. Lucas, pp161-187, 1988.

[8] Williams, M.H., "The Design of a Prolog Database Machine", Proceedings of the 1st Workshop for SIGKME, ppL1-L5, Reading University, England, Jan'87.

[9] Clocksin, W.F., "Design and Simulation of a Sequential Prolog Machine", New Generation Computing, vol. 3, pp101-120, 1985.

[10] Wong, K.F. "Comment on *A Comparison of Concatenate and Superimposed Code Word Surrogate Files For Very Large Data/Knowledge Bases*", Technical Report 88/6, Dept. of CS, Heriot-Watt University, Edinburgh, Scotland, 1988.

[11] Ramamohanarao, K. & Shepherd, J. "A Superimposed Codeword Indexing Scheme for Very Large Prolog Database", Proceedings of 3rd International Logic Programming Conference, London, England, July'86.

[12] Wong, K.F. & M.H. Williams, "Design Considerations for a Prolog Database Engine", Proceedings of the 3rd International Conference on Data and Knowledge Bases, pp111-120, Jerusalem, Israel, 28-30 June'88.

[13] Sun Microsystems, Inc., "Writing Device Drivers for the Sun Workstations", Sun Workstation Manual, February'86.

[14] Advanced Micro Devices, Inc, "Architectures Using the 2910A", Bipolar Microprocessor and Logic Data Book, p(5)175 -177,1985.

Appendix I: Data Types Supported in the Pseudo In-line Format

Table A1: CLARE Data Type Scheme

Items	Type Tag (1b)	Content (b/3b)	Extension (4b)
Variables:			
Anonymous Var	0010 0000 (0x20)	-	-
First Query Var	0010 0111 (0x27)	Variable Offset (b)	-
Subsequent Query Var	0010 0101 (0x25)	Variable Offset (b)	-
First DB Var	0010 0110 (0x26)	Variable Offset (b)	-
Subsequent DB Var	0010 0100 (0x24)	Variable Offset (b)	-
Simple Terms:			
Atom Pointer	0000 1000 (0x08)	Symbol Table Offset	-
Float Pointer	0000 1001 (0x09)	Symbol Table Offset	-
Integer In-line	0001 nnnn (0x1N)	Least Significant Value -	
	(nnnn = most significant nibble)		
Complex Terms:			
Structure In-line	011a aaaa	Functor Symbol Table Offset	-
	(aaaaa=arity, ≤31)	Structure Elements Follow	
Structure Pointer	010a aaaa	Functor Symbol Table Offset	Pointer to Structure
	(aaaaa=arity, ≤31)		
Terminated List In-line	111a aaaa	-	-
	(aaaaa=arity, ≤31)	List Elements Follow	
Unterminated List In-line	101a aaaa	-	-
	(aaaaa=arity, ≤31)	List Elements Follow	
Terminated List Pointer	110a aaaa	Pointer to List (4b)	-
	(aaaaa=arity, ≤31)	Only Use in DB Argument	
Unterminated List Pointer	100a aaaa	Pointer to a List (4b)	-
	(aaaaa=arity, ≤31)	Only Use in DB Argument	

1 **if** both types are integers **then**
compare their contents (numbers)

2 **if** both types are atoms **OR** float **then**
compare their contents (hashed values to the symbol table)

3 **if** both types are structures **then**
compare their contents (functor names and arities &
contents of top level elements)

4 **if** both types are lists **then**
compare their contents (length of list and types &
contents of top level elements)

5 **if** the database type is a variable **then**
check for the existence of the clause variable
5a **if** it does not exist **then**
create a new entry in the DB variable store
associate it with the query term

5b **if** it exists **then**
extract its association from the DB variable store
5c **if** it is a variable itself **then**
fetch the ultimate association from the DB store
repeat the comparison operations again

6. **if** the query type is a variable **then**
check for the existence of the query variable
6a **if** it does not exist **then**
create a new entry in the Query variable store
associate it with the database term
6b **if** it exists **then**
extract its association from the Query variable store
6c **if** it is a variable itself **then**
fetch the ultimate association from the DB store
repeat the comparison operations again

Figure 1: The simplified partial test unification algorithm.

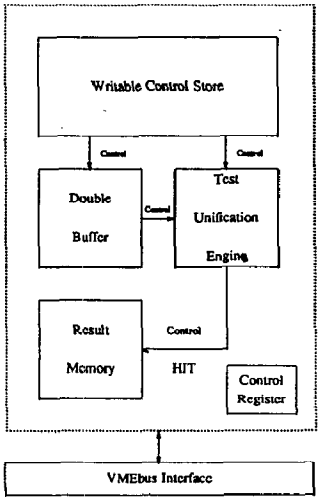


Figure 2: The overall architecture of the second stage filter hardware (FS2).

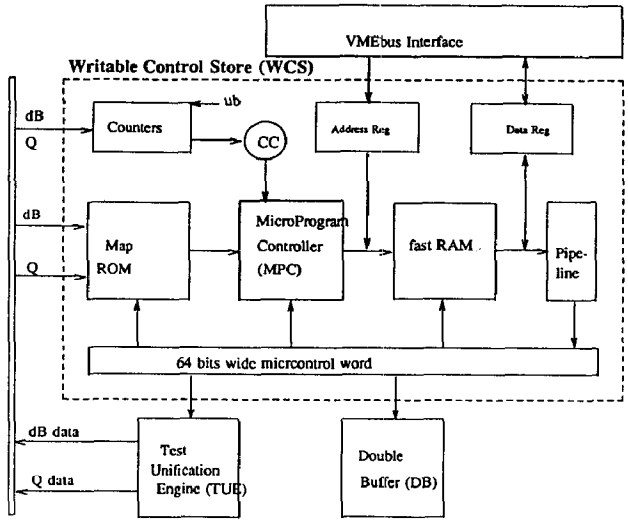


Figure 3: The Writable Control Store (WCS).

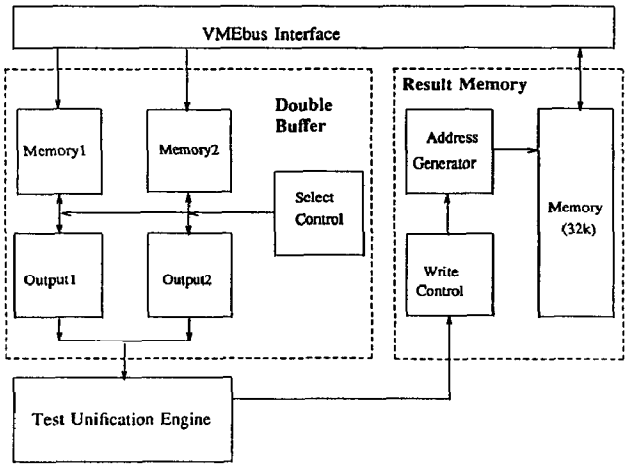


Figure 4: The Double Buffer and the Result Memory.

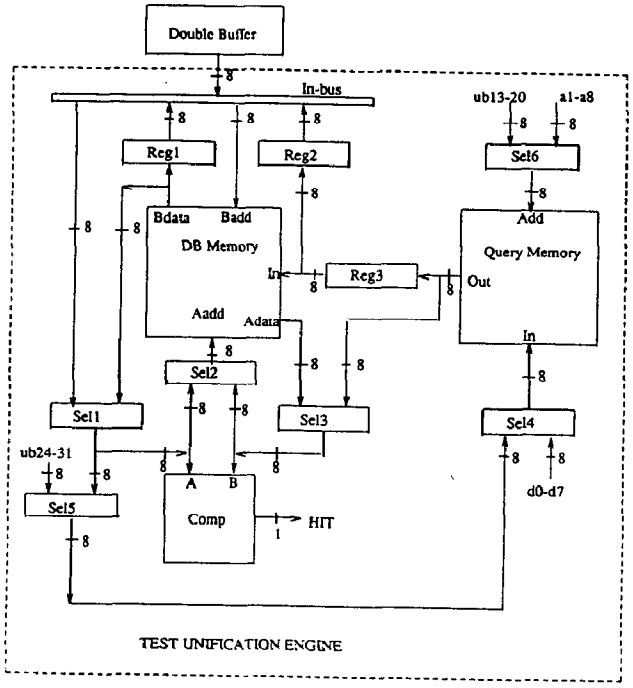
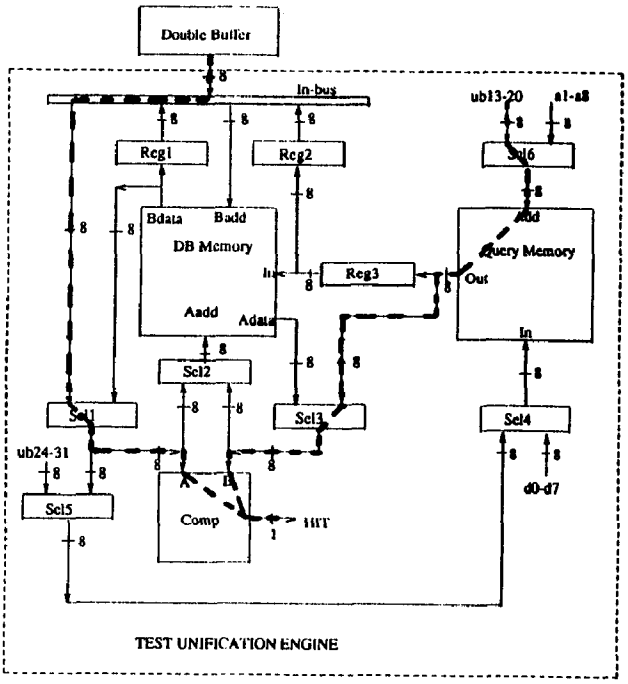


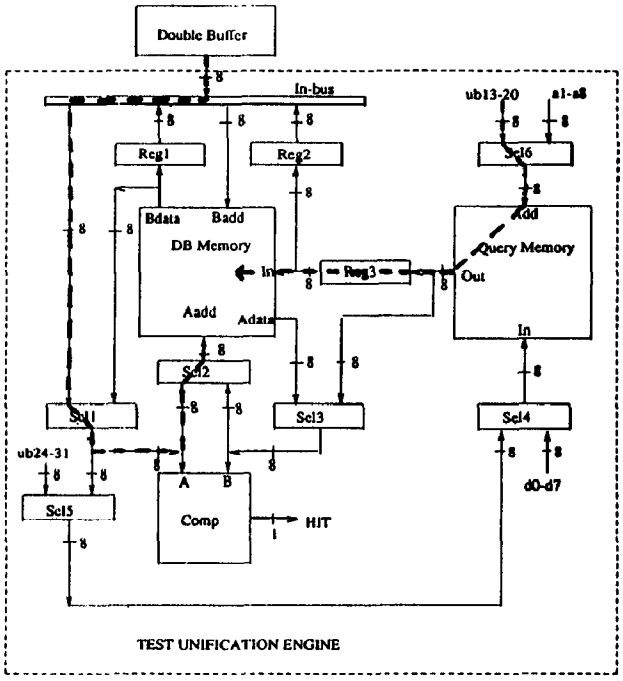
Figure 5: The Test Unification Engine.



Timing Calculation for the MATCH Operation						
database route :	Double Buffer	→	Sel1			
	20		20		(=40)	
query route :	Sel6	→	Query Memory	→	Sel3	
	20		35		20	(=75)
Comparison :	(=30)					

execution time = query route (75) + comparison (30) = 105ns.

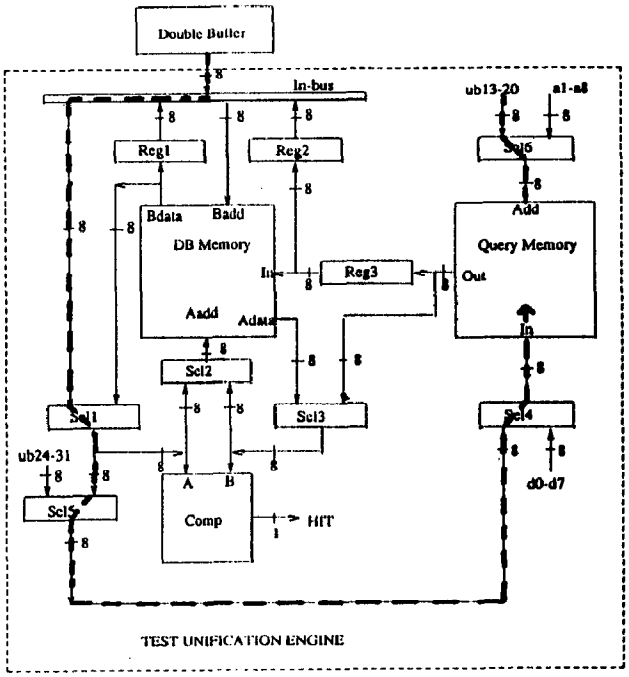
Figure 6: The MATCH Operation



Timing Calculation for the DB_STORE Operation						
database route :	Double Buffer	→	Sel1	→	Sel2	
	20		20		20	(=60)
query route :	Sel6	→	Query Memory	→	Reg3	
	20		35		20	(=75)
DB Memory write :	(=20)					

execution time = query route + DB Memory write = 95ns

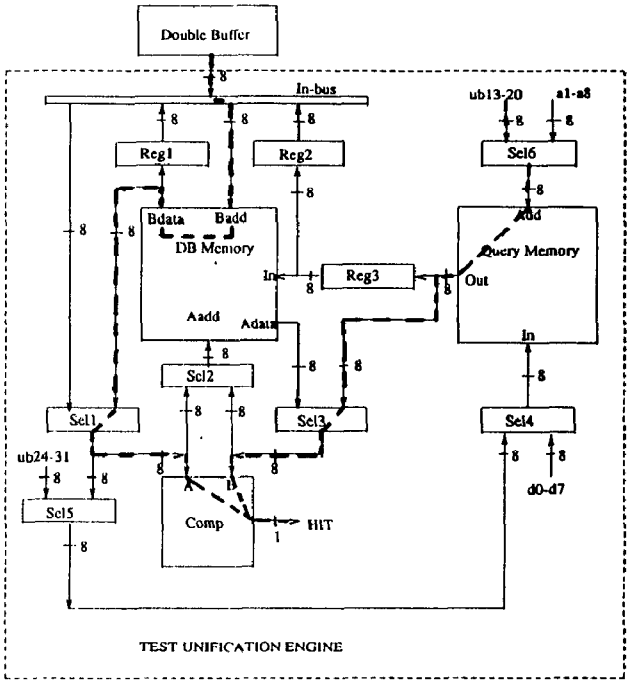
Figure 7: The DB_STORE Operation



Timing Calculation for the QUERY_STORE Operation								
database route :	Double Buffer	→	Sel1	→	Sel5	→	Sel4	
	20		20		20		20	(=80)
query route :	Sel6							
	20							
Query Memory write :	(=25)							

execution time = database route + Query Memory write = 105ns

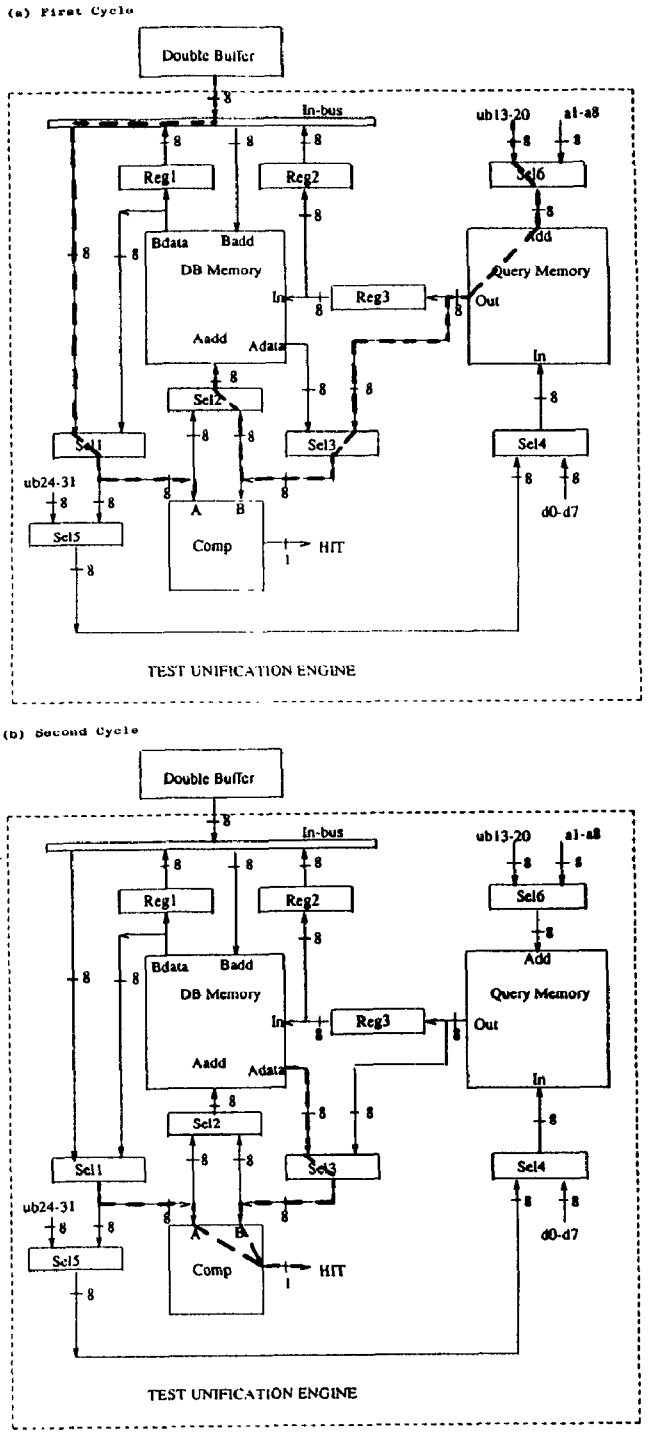
Figure 8: The QUERY_STORE Operation



Timing Calculation for the DB_FETCH Operation						
database route :	Double Buffer	→	DB Memory	→	Sel1	
	20		25		20	(=65)
query route :	Sel6	→	Query Memory	→	Sel3	
	20		35		20	(=75)
comparison:	(=30)					

execution time = query route (75) + comparison (30) = 105ns.

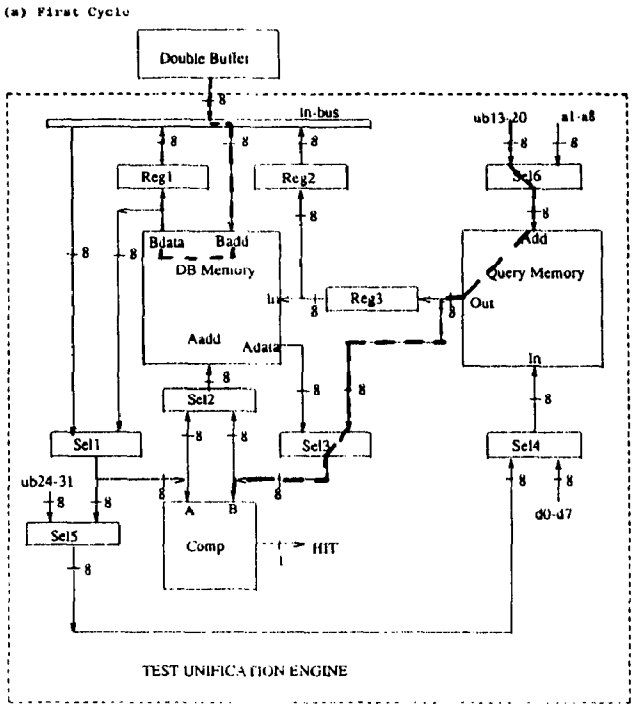
Figure 9: The DB_FETCH Operation

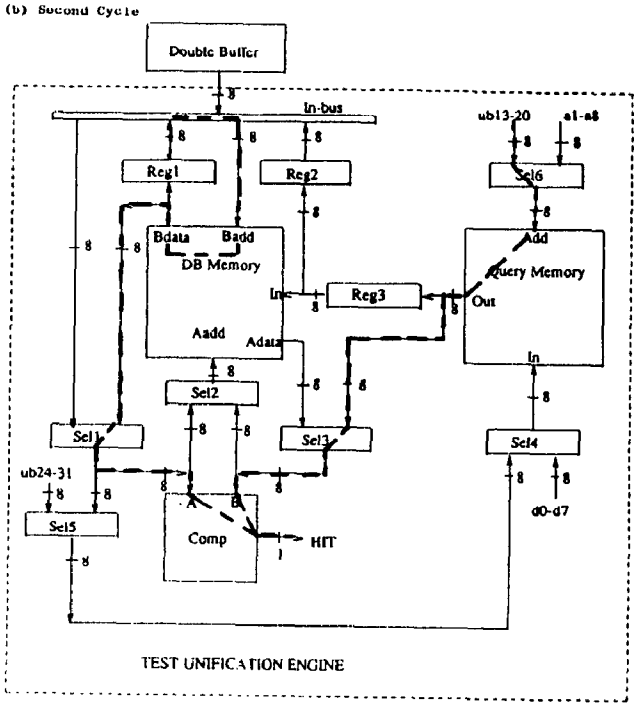


Timing Calculation for the QUERY_FETCH Operation										
First cycle										
database route :	Double Buffer	→	Sel1							
	20		20	(=40)						
query route :	Sel6	→	Query Memory	→	Sel3	→	Sel2	→	DB Memory	
	20		35		20		20		25	(=120)
Second cycle										
database route :	set at 1st cycle									
query route :	Sel3									
	20									
Comparison :	(=30)									

execution time = 1st cycle query route (120) + 2nd cycle query route (20) + comparison (30) = 170ns.

Figure 10: The QUERY_FETCH Operation

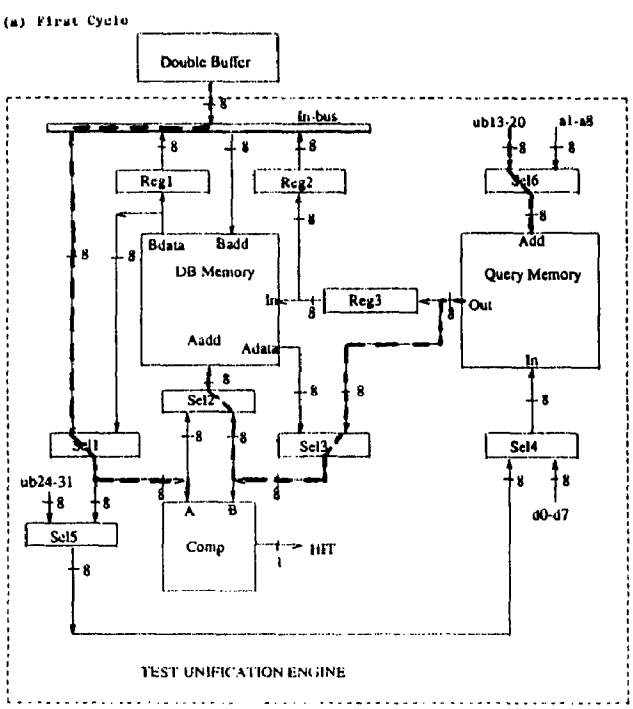




Timing Calculation for the DB_CROSS_BOUND_FETCH Operation						
First cycle						
database route :	Double Buffer	→	DB Memory	→	Reg1	
	20		25		20	(=65)
query route :	Sel6	→	Query Memory	→	Sel3	
	20		35		20	(=75)
Second Cycle						
database route :	Reg1	→	DB Memory	→	Sel1	
	20		25		20	(=65)
query route :	set at 1st cycle					
comparison :	(=30)					

execution time = 1st cycle query route (75) + 2nd cycle database route (65) + comparison (30) = 170ns.

Figure 11: The DB_CROSS_BOUND_FETCH Operation



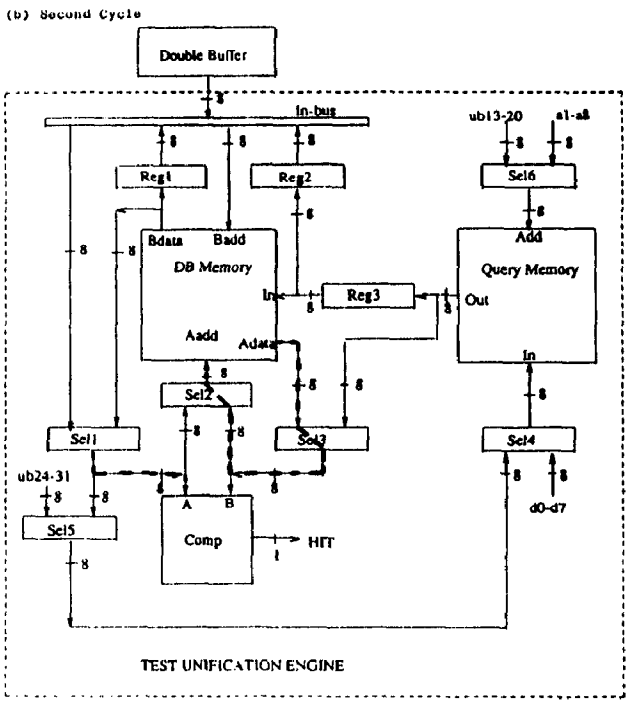
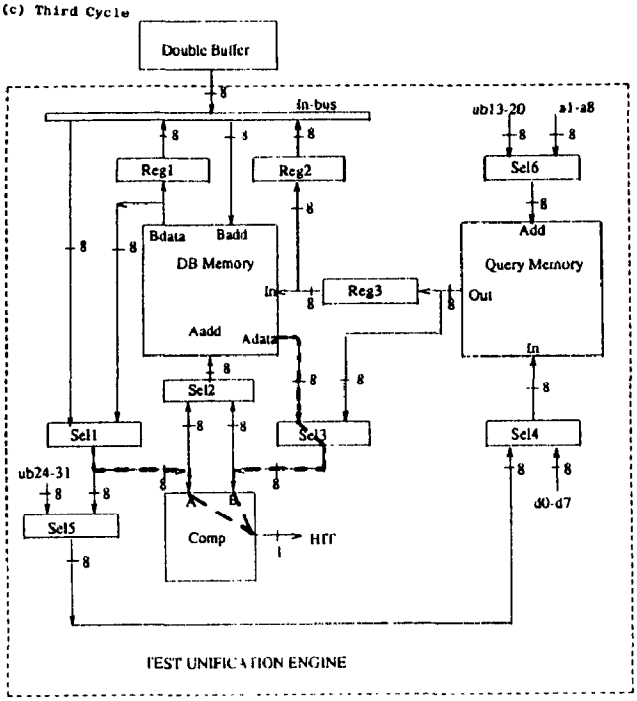


Figure 12: The QUERY_CROSS_BOUND_FETCH Operation



Timing Calculation for the QUERY_CROSS_BOUND_FETCH Operation								
First cycle								
database route :	Double Buffer	→	Sel1					
	20		20	(=40)				
query route :	Sel6	→	Query Memory	→	Sel3	→	Sel2	
	20		35		20		20	(=95)
Second cycle								
database route :	set in 1st cycle							
query route :	DB Memory	→	Sel3	→	Sel2			
	25		20		20	(=65)		
Third cycle								
database route :	set at 1st cycle							
query route :	DB Memory	→	Sel3					
	25		20	(=45)				
comparison :	(=30)							

execution time = query routes of (1st cycle (95) + 2nd cycle (65) + 3rd cycle (45)) + comparison (30) = 235ns

Figure 12: The QUERY_CROSS_BOUND_FETCH Operation

(Continuation)

Table 1: Execution Times of the FS2 Hardware Functions

Corresponding Figure	Operations	Execution times (ns)
6	MATCH	105
7	DB_STORE	95
8	QUERY_STORE	115
9	DB_FETCH	105
10	QUERY_FETCH	170
11	DB_CROSS_BOUND_FETCH	170
12	QUERY_CROSS_BOUND_FETCH	235