KCM: A Knowledge Crunching Machine

H. Benker, J.M. Beacco, S. Bescos, M. Dorochevsky, Th. Jeffré, A. Pöhlmann,
J. Noyé, B. Poterie, A. Sexton, J.C. Syre, O. Thibault, G. Watzlawik

ECRC, Arabellastr. 17, 8000 Muenchen 81, West Germany

E-mail: {hans, jacques, jclaude}@ecrcvax.uucp

ABSTRACT

KCM (Knowledge Crunching Machine) is a high-performance back-end processor which, coupled to a UNIX desk-top workstation, provides a powerful and user-friendly Prolog environment catering for both development and execution of significant Prolog applications. This paper gives a general overview of the architecture of KCM stressing some new features like a 64-bit tagged architecture, shallow backtracking and an original memory management unit. Some early benchmark results obtained on prototype machines are presented. They show that KCM, which runs at a peak speed of 833 Klips on list concatenation, compares favorably with other dedicated Prolog machines and available commercial systems running on fast general purpose processors.*

Keywords: Prolog engine, tagged architecture, memory system

1 Introduction

KCM (Knowledge Crunching Machine) is intended to be a single user, single task high-performance back-end processor which, coupled to a UNIX* desk-top workstation, provides a powerful and user-friendly Prolog environment catering for both development and execution of significant Prolog applications. Conducted at ECRC, this project is a joint effort involving ECRC 's shareholders Bull, ICL and Siemens who intend to build KCM pilot machines in 89. Until now and unlike its predecessors ICM3 [12] and ICM4 [1] which did not go beyond chip level simulations, prototype KCM machines have been built. A complete software environment is currently being written, based on SEPIA [8], a second generation Prolog system also developed at ECRC.

Starting from the ICM3 design the general architecture was simplified and the functionality of the machine was extended (FPU, MMU supporting paging, hardware stack overflow check). Much care has been taken to keep the machine as flexible as possible. Though KCM is dedicated to Prolog, it is not restricted to Prolog and can be seen as a tagged general purpose machine with support for Logic Programming in general.

This paper is organised as follows; section 2 explains the main design decisions. Then section 3 describes the architecture of the machine. In section 4, some early benchmark results are presented. Comparisons are made with other dedicated Prolog machines and Prolog systems running on general purpose computers. We conclude by sketching future work.

We assume some familiarity with the programming language Prolog [16] and its implementation techniques [20, 5].

2 Design Decisions

2.1 System Environment of KCM

The KCM machine is intended to supply a workstation type of machine with high performance for processing logic programming languages. Our first approach to this goal was to design a Prolog co-processor, following the example of existing numerical co-processors. This is the easiest way to connect to existing software written under UNIX*. Unfortunately, as opposed to numerical co-processors, the memory behaviour of Prolog programs is very dynamic and they need random access to all symbol tables and to the entire run-time environment. Therefore either the Prolog co-processor has to share all data with the host or to support its own run-time environment.

Sharing all data between the host and the co-processor means that all data need to be in identical formats on both processors. Due to the architectural differences between the machines such a format can only be a non-optimal compromise for at least one of the machines.

The high memory bandwidth requirement is easier to match if the memory is not shared with another processor.

It is very inefficient to frequently switch the execution of a program between the host and the co-processor. Unlike numerical co-processors it may ask for help from the host, e.g. to execute a built-in predicate in Prolog. In this case the complete internal state of the co-processor has to be sent to the host. This is very costly, since the internal state consists of all internal registers and the dirty entries in the cache (if it is store-in, which is reasonable to assume).

For these reasons KCM was designed as back-end processor with its own private memory. The complete language sub-system runs on KCM which is designed as a single-task machine. It uses the host with its operating system (UNIX) as server for I/O including tasks like e.g. interaction with the user (tty, window system etc.), file management, paging etc.

*UNIX is a registered trademark of AT&T.

Figure 1 shows the system environment of KCM and how it is connected to a host machine. Physically the CPU of KCM is implemented on one large printed circuit board. The interface with the communication memory and the main memory of KCM are separate boards of double-euro format. All boards together fit into the cabinet of the diskless desktop workstation that is used as host.

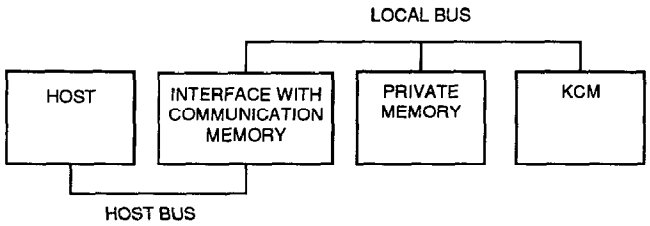

Figure 1: KCM System Environment

2.2 Technology

For the implementation of KCM we choose to use standard TTL and CMOS technology. Although top-of-the-range ASIC technology certainly gives performance advantages over the use of off-the-shelf components, it was not considered worthwhile for a research project. It is easier to experiment with standard components and undertake small design changes if one does not fix the design in silicon. Nevertheless two small ASIC chips are used which make it easier to fulfill the requirement to fit into a workstation.

2.3 Word Format

The model of computation for KCM is derived from the WAM, the abstract machine defined by D.H.D. Warren [20]. This model supposes tagged data words, i.e. a basic entity consists of a value plus an additional tag field that gives information on its type.

Most commercial Prolog systems on general-purpose computers pack tag and value into a 32-bit word, thus reducing the range of virtual addresses and integers. Many tagged architectures are based on a word of 36 - 40 bits to store a 32 bit value together with the tag, causing problems to the communication with standard machines, due to the non-standard word length. Therefore some AI-machines actually use 40 bits internally but 64 bits externally, i.e. in memory and on disc ([7]). There are also software systems using 64 bits to store a Prolog entity. In particular the system developed at ECRC is designed that way, see [8]. This was adopted for KCM which is built as a 64-bit machine, 32 bits for the value part and 32 bits for the tag part (figure 2 shows the KCM data word).

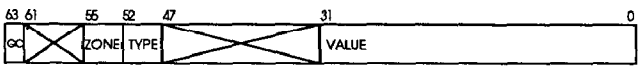

Figure 2: KCM Data Word Format

For simplicity of the design it was decided to use identical size for both code and data words. RISC supporters have proved that a fixed instruction length saves a lot of decoding hardware and also time in the critical path. A 64-bit instruction word allows encoding register addresses etc. always in the same fields of the instruction. The savings in decoding hardware make up for a large amount of the additional memory used. Also a 64-bit instruction width provides room for a lot of flexibility allowing a more regular and simple, but yet powerful instruction set. Figure 3 shows the two basic instruction formats of KCM.

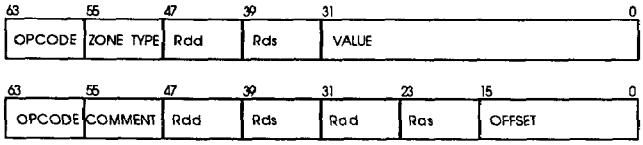

Figure 3: KCM Instruction Word Formats

2.4 Memory System

KCM has two separate access paths to memory, one for code and one for data. There are two independent caches, but the physical memory is shared between instructions and data. KCM shares this property with most RISC architectures as well as some of the recent CISC designs (e.g. 68030).

Both caches of KCM are logical caches, i.e. they operate on the basis of virtual addresses. Logical caches are faster than physical caches, because there is no need to translate addresses for every memory access, but only for cache misses. The main reason why logical caches are rarely used is that they need to be flushed on a context switch. This is no problem to KCM as it is designed as a single-task backend processor.

To improve locality on stacks it was decided to use a version of the WAM known as the split-stack model, i.e. there are two separate stacks for environments and choice points to control execution instead of one.

2.5 Control

The basic operation in unification is a kind of pattern matching which involves the comparison of the type and value parts of two words in parallel. Multiple different reactions may be required, according to the result of the analysis of the two types involved. Due to its microcoded control KCM can quickly change the flow of control. It is equipped with special data paths to calculate entry points into microcode according to the combination of two types.

The semantics of some instructions of KCM are not context-free. According to the value of a number of flags such instructions may be executed in different modes. The mode bits are taken into account when decoding instructions and therfore do not need to be tested explicitly.

KCM is an entirely synchronous machine, controlled by a single central microsequencer. Simulation studies revealed that typically the gain achievable by asynchronous operation of an instruction prefetch unit is compensated by the overhead of synchronisation.

2.6 Top-Level Architecture

Starting from the Harvard architecture the top-level block diagram of the machine is easily developed. The basic parts are the two caches with the data processing units connected to them: the execution unit to the data cache and the instruction prefetch unit to the code cache. These units are controlled by the control unit and the memory management

connects the physical memory. The memory management is in between the caches and the main memory, not in between the CPU and the caches, i.e. logical caches are used. The block diagram in figure 4 shows these units with the main buses of KCM.

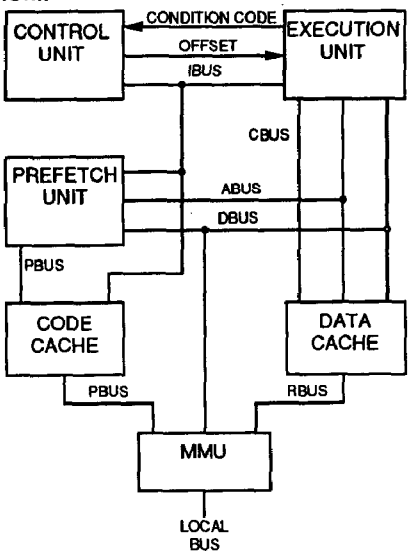

Figure 4: KCM Top Level Architecture

3 Architecture of KCM

According to the design decisions developed in the previous section we can now present the main features of KCM:

- 64-bit tagged architecture
- conventional technology (TTL/CMOS) plus two CMOS-ASICs
- 4-phase clock system, 80 ns cycle time
- separate logical code and data caches, 8K words each
- private memory (32 Mbytes on one board)
- hardware support for unification and backtracking
- verification of access rights to virtual addresses

3.1 The CPU

3.1.1 Basic Data Manipulation

Source and destination for all data manipulation instructions are registers in the 64 x 64 bit register file. The instructions have a four address format; two source and two destination registers. The addresses are supplied to the register file by the Register Address Calculator RAC which is implemented in an 1.5 µm CMOS ASIC.

Using the four address format KCM supports a move instruction that moves the contents of two 64 bit wide registers in one cycle. Figure 5 shows the basic data paths used: the source data are put on the buses ABUS and BBUS. They are transferred via the ALUs ALU_C and ALU_D and written back to the register file on the buses CBUS and DBUS respectively.

For arithmetic and other data manipulation the ALU_D or the FPU are used. Such instructions use only three of the four register address fields in the instruction format: two source and one destination register. Both the ALU and FPU only treat the data part of a word; 32 bit IEEE data format is used. The tag part of the data word usually just bypasses the ALU or FPU using the Tag-Value-Multiplexer TVM.

For operations on 64 bit words the TVM is used. It can manipulate the garbage collection bits in the data word and swap value and tag parts of a word.

Most data manipulation instructions execute in one cycle. Exceptions to this rule are: floating point operations, integer multiplication and division.

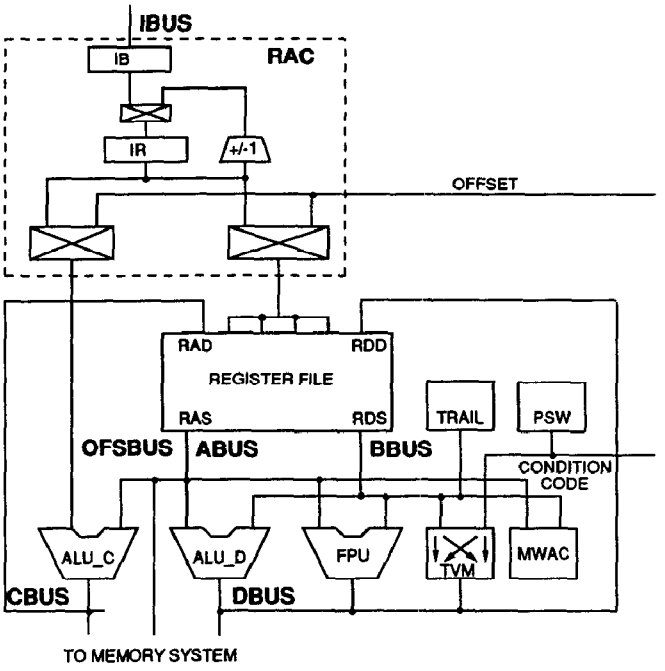

Figure 5: The Execution Unit

3.1.2 Memory Access

KCM supports three address modes: direct, pre-address calculation, post-address calculation.

The direct address mode uses an instruction format that comprises an absolute address and a source or destination register for a store or load instruction respectively.

The pre- and post-address calculation instructions use an instruction format with three register addresses and a 16-bit signed offset. The register addresses are:

- Ras - address source register
- Rad - address destination register
- Rds - data source register for store instructions
- Rdd - data destination register for load instructions

The contents of Ras is put on the ABUS. The ALU_C adds the 16 bit signed offset from OFSBUS to the address and the result of the computation is stored into register Rad via the CBUS.

The data cache can use either ABUS or CBUS as address source, i.e. the address can be taken directly from register Ras or include the offset. The data input/output of the data cache is connected to the DBUS.

The addressing modes allow implementation of stacks that grow in either direction. Pre-increment, post-increment, pre-

decrement, post-decrement, as well as an offset from the top-of-stack are all supported.

3.1.3 Instruction Sequencing

Figure 6 shows a diagram of the instruction prefetch unit of KCM. It is based on a three stage pipeline: the register P contains the address of n+2; the register IB contains the instruction n+1 and SP the address from where it was fetched; the register IR contains the instruction n that is currently executed and TP its address in the code space.

During normal execution P is incremented every cycle and the registers SP, TP, IB, and IR are all clocked. This allows to fetch and execute instructions at a rate of 1 instruction/cycle.

A special instruction predecoding hardware switches the multiplexer for P to use IB as input if the currently fetched instruction is a branch. Thus immediate jump and call instructions take two cycles. All branches in KCM have absolute addresses as branch targets.

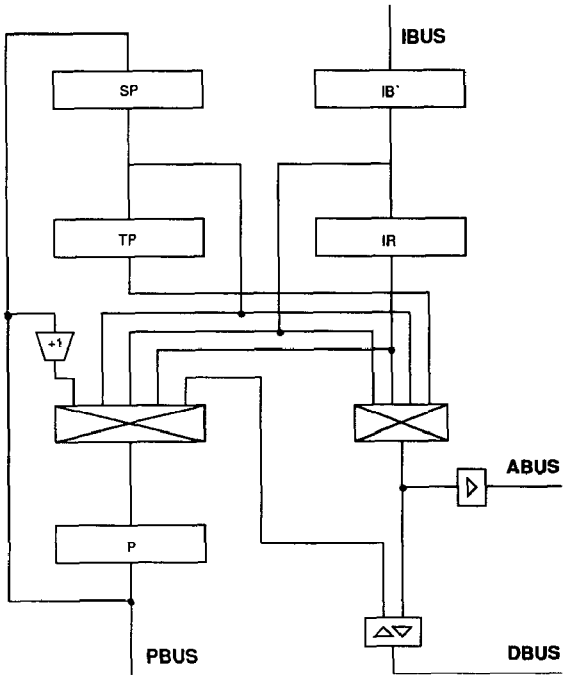

Figure 6: The Prefetch Unit

Conditional branches operate on the status bits of the ALU and FPU as well as other status bits in the machine that are all stored in the Processor Status Word PSW. Conditional branches take only one cycle if the branch is not taken and four cycles if the branch is taken.

The prefetch unit is implemented as two 32-bit slices using 1.5 μm CMOS ASIC technology.

3.1.4 Support for Unification

The basic elements supporting unification in KCM are the Multi-Way-Address-Calculator MWAC together with the microcoded control.

Following the basic ideas of the WAM [20] the compiler splits unification into a number of smaller steps implemented as unification instructions. The corresponding algorithms are to first determine the types of the input parameter(s) and then take appropriate action. In order to implement these unification steps efficiently it is therefore necessary to quickly react, according to the combination of types of the input parameters.

The MWAC is implemented as a PROM. Its inputs are the two type fields of the source operands on ABUS and BBUS. Depending on the current unification instruction it maps the two input types onto a 4 bit offset. The microcode sequencer branches to a microcode address to which it adds this offset, i.e. it does a 16-way branch according to the input types.

The algorithms for the instructions unifying lists or structures are different for the read and write case. The *get_list* and *get_structure* set the read/write mode flag. In KCM this flag is directly used for the decoding of the unification instructions. The appropriate microcode can therefore start immediately and no time is lost for a test.

The register file holds the source operands for unification instructions. These may be the objects to be unified themselves or they may be references to objects in memory. The objects in memory may again be references to other objects. Some hardwired control in addition to microcoded control of the data cache supports the detection of references. It is possible to start a dereferencing operation in the data cache even if the object sent to the data cache is not an address. If it is an address, then the data cache will perform a read, if it is not then it will abort the read (this is important, because random data - like e.g. a floating point number - used as an address may cause a cache-miss and even a page fault which certainly is not tolerable). This allows following reference chains at the rate of one reference per cycle.

3.1.5 Support for Backtracking

In general, Prolog execution relies heavily on backtracking. Each time there are at least two clauses matching the current goal, the state of the computation has to be saved. In case of failure of the first alternative, the execution can then backtrack and try the following alternative.

In the standard WAM, a special frame called choice point has to be pushed onto memory each time such a situation occurs. The size of a choice point varies with the number of arguments but its typical size is about 10 words. Saving and restoring such frames was found to be a major source of memory traffic. According to [17], it amounts to about 50% of all memory references.

In KCM, the creation of choice points is delayed so that in case of shallow backtracking saving and restoring a choice point is avoided. Shallow backtracking is very frequent (see [21] or [17]). It corresponds to a failure in the head or the guard of a clause for which there are still alternatives. The guard is a possibly empty series of goals following the head which is known not to modify the Prolog state of execution (with the exception of the continuation).

The scheme used (inspired by [13]) relies on two flags. In case of failure the *shallow flag* indicates the type of the failure. It is set each time a clause with some remaining alternatives is entered and reset at the neck of the clause (i.e. after the guard). The *choice point flag* indicates whether there is already a choice point for the current clause. If a clause is entered in deep mode, it is set, otherwise it is reset.

When trying the first alternative, only three state registers are saved into shadow registers in the register file. No choice point is generated until the neck of the clause (or the neck of some of its alternatives in case of shallow backtracking) is encountered. The execution of many predicates is then made deterministic, i.e. the head and the guard suffices to select a unique matching clause without any choice point creation.

Both flags are treated as the read/write mode flag at decode time and thus never tested at execution time.

The creation of choice points itself is supported by the Register Address Calculator (RAC). It supplies the register file with addresses. On choice point creation/restoration a number of consecutive registers, depending on the arity of the predicate, need to be stored/loaded. The RAC can increment and decrement register addresses and therefore a microcode loop can store/load one register per cycle.

When unification binds a variable that is older than the last choice point, it has to push an item onto the trail stack in order to unbind the variable upon the next fail. Up to three comparisons of the address of the variable with the contents of special registers are required in order to determine whether trailing is necessary or not. The Trail hardware in the execution unit of KCM performs these comparisons in parallel with dereferencing.

3.2 The Memory System of KCM

3.2.1 Virtual Address Spaces

Code and Data Space
Prolog and other AI languages allow some kind of self modifying code and incremental compilation. To overcome resulting consistency problems between data and code cache KCM has two different address spaces for code and data along with two sets of instructions to access each of them. Incrementally generated code is written directly to the code cache. This however preserves the possibility for a compiler/assembler/linker/loader to run in batch mode and generate large blocks of instructions in the data space (where a write access is more efficient). When the compilation is finished, the memory management system can invalidate the virtual data page and attach the physical page to the code space.

Available Virtual Address Space
All addresses in KCM are word addresses, i.e. they address a 64-bit entity. In the current implementation of the KCM architecture only the 28 least significant bits of the value part of the address are used. Due to the two separate virtual address spaces the total amount of virtual memory is the equivalent of 32 bits, i.e. KCM can address as much virtual space as most 32-bit processors but the address format has room for future extensions.

3.2.2 Semantics of an Address

There are only two different types of words in KCM that need to be distinguished by context: code and data. Instructions are distinguished by opcodes and data contain a type field encoding different types. Data words can be used as addresses. Figure 7 shows the format of a data word used as address.

The bits 31 to 0 are the actual value of the address. These bits correspond to an address in a non-tagged architecture.

The four bits 51 to 48 encode 16 possible types such as *integer*, *floating point*, *variable*, *list*, *data pointer*, or *code pointer*.

The four bits 55 to 52 encode a zone in virtual memory. Stacks, heaps, and other data areas are mapped to zones. Thus the zone bits encode information like e.g. local stack, global stack, heap, and static data area.

The bits 63 to 56 and 47 to 32 are currently not used for addresses.

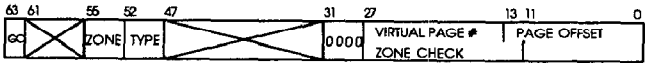

Figure 7: KCM Address Format

3.2.3 Check of Access Rights at the Logical Level

The MMU provides protection mechanisms for physical memory. There are three reasons which led us to introduce an additional protection scheme called zone check, working at the level of virtual addresses.

The first reason is to monitor the size of the multiple stacks. This allows detection of stack overflows and prevention of stack collisions. The monitoring of stack size can be used to trigger garbage collection as well as to implement an adapted paging strategy. In this respect our scheme is very similar to [15].

The second reason is additional security and debugging support. The zone check of KCM uses the type field of an object to prevent the programmer from using e.g. the result of a floating point operation to address a memory cell.

The third reason is that the MMU is not involved when writing to the logical cache. Suppose a write-access occurs to an invalid or write-protected virtual address. Without protection on the level of the logical caches the data will simply be stored in the cache. It is not until the same cache cell is used otherwise that the MMU can detect the error.

Each stack and memory area in KCM is mapped to a zone which is defined by a start and an end address. Using the zone number given with every address it is checked that the value of that address actually points to in between the minimum and maximum address of that zone. The limits of the zones may be changed dynamically.

Each of the different stack pointers has a different zone number. If the stack pointer is incremented or decremented beyond the minimum or maximum address of its zone then the next access using it will cause a trap.

The zone check helps developing system (assembler) programs by forcing and verifying the proper use of addresses: each zone may only allow a limited number of types as addresses and each zone may be write-protected. In the following we give examples of types that are or are not allowed as addresses in different zones. Any number type like *integer* or *floating point* is not allowed as address pointing into any zone. Lists and structures are constructed on the global stack. Therefore the types *list* and *structure* are allowed as addresses into the global stack as well as the types *reference* and *data pointer*. On the local stack, however, only *reference* and *data pointer* are allowed, since lists and structures are not

constructed there. Other stacks like the choice point stack allow only *data pointer* to be used as an address, since no *reference* may ever point into that stack.

The zone check operates on virtual addresses when accessing the data cache. It verifies that the most significant 4 address bits not used in the current implementation are zero. Then it checks whether the 16 bits from 27 to 12 (see figure 7) are within the range stored in a special RAM field. This allows the definition of zones with a granularity of 4K words.

3.2.4 The Caches

The Data Cache
Studies of Prolog memory behaviour show that many items get pushed onto the stacks that are never accessed again [17]. This is due to the fact that environments and structures are allocated but never read because of failure. As result the ratio of reads to writes in Prolog is about 1:1 which is much smaller than in conventional programming languages. Therefore the data cache in KCM is a so called store-in or copy-back cache where data is written to memory only when the cache cell is needed otherwise, but not at every single write. The access time for both read and write is 80 ns.

Prolog stacks show a high locality near the top [17]. Thus, a direct-mapped cache works very well for a single stack. For a stack which is only accessed from the top it has exactly the same behaviour as a top-of-stack circular buffer, but it is easier to implement. A direct mapped cache can also cache items deep in the stack, if required, for stacks with random accesses. The problem with direct mapped caches is what happens if accesses to multiple stacks occur and the top-of-stack pointers modulo the cache size are close to each other. We ran a number of small programs in a simulator of a direct mapped cache with two different initialisations: In the first run the top-of-stack pointers were initialised to values such that they used different cache locations. For the second run the top-of-stack pointers were initialised such that they all pointed to the same cache cell. The hit ratios were very good in the first run and dropped quite dramatically in the second.

In a direct mapped cache such collisions between stacks are bound to occur at some stage, because stacks grow at different rates. A set associative cache solves this problem, but at a high cost in hardware - and even worse - a slower cycle time. KCM uses a direct mapped data cache, but it is split into 8 sections of 1K x 64 bits each. The sections are selected by the zone field of the address word.

The line (block) size of the data cache is one.

The Code Cache
Unlike the data cache the instruction cache almost always is accessed to read an instruction, but only very rarely to write. Therefore it is designed as a write-through cache. The read access time is 80ns.

The size of the code cache is 8K x 64 bits.

The line size of the code cache is one. Since it is a write-through cache the line size does not prevent the code cache from using the page mode of the memory and fetching a few words ahead when a miss occurs.

3.2.5 Address Translation and Allocation of Physical Memory

The address translation hardware is designed for speed and simplicity, i.e. a simple RAM is used to hold the entire page table rather than storing the page table in main memory and use an associative cache for fast access to recent translations. This design works because KCM is a single-task machine that does not need to do context switches. There is no need to exchange valid page tables between one process and another and so the size of the page table held in the machine does not matter. There is no need either to swap pages out because another process would need them. Therefore pages can be quite large.

The bits 27 to 14 of an address give the virtual page number and the bits 13 to 0 the offset into one page (see figure 7), i.e. the page size is 16K words.

The address translation is done using a RAM organised as 32K x 16 bit. It contains one entry for each virtual page (16K virtual pages for code and data each). Each entry consists of 5 status bits plus 11 bits physical page number.

3.2.6 Main Memory

The main memory is implemented on a separate board of double-euro format. Using SMD technology with components mounted on both sides one such board holds 32 MBytes. This is using 1 Mbit chips, but the layout of the board allows the use of 4 Mbit chips to obtain 128 MBytes. Two such boards can be plugged into the workstation.

The memory is implemented with a 32 bit wide data bus. A fast page mode is used to access two 32 bit words in order to form a 64 bit KCM word. The page mode is also used to prefetch data for the code cache. The page mode cycle time is 120 ns.

4 Early Results

Since the first prototype started working in July 1988, a number of benchmark programs have been run on KCM in order to verify the correctness and measure the performance of the machine. Currently there are three KCM machines available on which the benchmark programs are run, using a first software environment including:

- code generation tools (Prolog compiler, macro assembler, linker)
- a loader
- an emulator
- monitors (at microcode, macrocode, and Prolog levels)
- a message passing system connecting the tools to the KCM systems (KCM and its driver or the emulator) in a plug-compatible way.

Though many more benchmark programs have been run, we will restrict ourselves to the PLM suite. This suite was gathered by the PLM team at U.C. Berkeley in order to evaluate the performance of the PLM [4]. It is an extension of the initial set of benchmarks written by D.H.D. Warren [19]. It has the advantage of having been widely distributed and used so that there is no ambiguity on the benchmark programs and queries.

The programs were compiled and assembled on the host

with integer arithmetic and static linking*. Each program was linked together with a small runtime library (I/O, timing predicates to control a timer residing on the KCM-host interface card...). Note that this library did not include any assert/retract facilities which made it impossible to run one of the programs of the suite. The programs were finally downloaded and run on KCM.

4.1 Static Code Size Comparisons

Table 1 compares the static code sizes for KCM, PLM and SPUR. SPUR is a general-purpose RISC architecture that supports tagged data developed at U.C. Berkeley [7]. The figures on PLM and SPUR are taken from [2]. Note that in the three cases the values indicated do not include the code of the runtime library which has to be linked with each program.

	PLM		SPUR		KCM			KCM/PLM		SPUR/KCM	
Program	Instr.	Bytes	Instr.	Bytes	Instr.	Words	Bytes	Instr.	Bytes	Instr.	Bytes
con1	28	87	414	1656	33	31	248	1.18	2.85	12.55	6.68
con6	32	106	430	1720	39	41	328	1.22	3.09	11.03	5.24
divide10	213	661	3988	15952	214	234	1872	1.00	2.83	18.64	8.52
hanoi	52	183	385	1540	56	59	472	1.08	2.58	6.88	3.26
log10	207	625	4040	16160	198	208	1664	0.96	2.66	20.40	9.71
mutest	141	468	1703	6812	162	172	1376	1.15	2.94	10.51	4.95
nrev1	71	260	761	3044	64	70	560	0.90	2.15	11.89	5.44
ops8	205	633	3804	15216	206	216	1728	1.00	2.73	18.47	8.81
palin	178	565	2556	10224	230	240	1920	1.29	3.40	11.11	5.32
pri2	132	383	1933	7732	141	151	1208	1.07	3.15	13.71	6.40
qs4	121	456	1230	4920	184	192	1536	1.52	3.37	6.68	3.20
queens	242	723	3636	14544	212	224	1792	0.88	2.48	17.15	8.12
query	273	1138	3942	15768	305	357	2856	1.12	2.51	12.92	5.52
times10	213	661	3988	15952	214	224	1792	1.00	4.68	18.64	8.90
average								1.10	2.96	13.61	6.43

Table 1: Static code size comparison

As expected, the static instruction KCM/PLM ratios are close to 1, though a little bit higher. Both machines rely on the same kind of high-level WAM-like instructions to run Prolog. The main difference is due to the use of cdr-coding in PLM. This allows PLM to compile a statically known list cell in one instruction rather than two in KCM. This notably results in high ratios for nrev1 and qs4 which include long input lists (respectively 30 and 50 elements). However, as discussed in [18], cdr-coding is only effective, as far as code space is concerned, for statically built lists. It does not bring a significant benefit on average.

The static byte KCM/PLM ratio is about 3. The average PLM instruction is 3.3 bytes long, whereas the average KCM instruction is slightly more than 8 bytes long (because of the *switch* instructions which are the only multi-word instructions together with some specialised byte manipulation instructions). This gives an idea of the cost of our instruction encoding in terms of memory space. This cost is quite reasonable compared to the cost of a low level instruction set such as SPUR, which produces, on average, code which is more than 6 times bigger. The code cache is expected to be large enough to keep the incurred traffic ratio low. This however requires further evaluation.

*In the final system, compilation will take place on KCM with generic arithmetic and dynamic linking by default

4.2 Benchmark Execution Times

Table 2 compares the PLM machine with KCM. The PLM figures are first evaluation figures extracted from [4]. For each program two timings were given, one for hand compiled code and one for automatically compiled code. The best figure is retained.

Benchmark		PLM		KCM		PLM/KCM
Program	Inferences	ms	Klips	ms	Klips	ms/ms
con1	6	0.023	261	0.007	857	3.29
con6	42	0.137	307	0.059	712	2.32
divide10	22	0.380	58	0.091	242	4.18
hanoi	1787	7.323	244	2.795	639	2.62
log10	14	0.109	128	0.039	359	2.79
mutest	1365	12.407	110	4.644	294	2.67
nrev1	499	2.660	188	0.650	768	4.09
ops8	20	0.214	93	0.059	339	3.63
palin25	325	3.152	103	1.221	266	2.58
pri2	1235	10.000	124	5.240	236	1.91
qs4	612	4.854	126	1.316	465	3.69
queens	687	4.222	163	1.205	570	3.50
query	2893	17.342	167	12.610	229	1.38
times10	22	0.330	67	0.082	268	4.02
average						3.05

Table 2: Comparison with PLM

It is important to note that the PLM timings result from a simulation of the benchmark programs and, as such, rely on some simplifying assumptions. In particular, built-in functions implemented via the escape mechanism (i.e. resorting to the host) were not accurately timed; they were allocated a standard 3 cycles.

To get a fair comparison, the same kind of assumption was made, on KCM side, for write/1 and nl/0 which were compiled as unit clauses, so that a call to these predicates costs only 5 cycles (the minimum for a call/return sequence which creates two prefetch pipeline breaks). We cannot however guarantee that both assumptions match completely. This has however no influence on mutest which does not include any I/O and very small influence on most of the programs for which I/O are used only to report the final result. Exceptions are con1 and con6 because of their very small size as well as hanoi which reports each move executed on the towers.

The Klips (Kilo logical inferences by seconds) figures use our own definition of an inference. A logical inference is taken to be the invocation of a goal at the source level. Independently whether a goal is compiled in line or not, it is counted as an inference. For instance the evaluation of an arithmetic expression (predicate is/2) is counted as one inference whatever the complexity of the expression. More generally, calls to built-in predicates are counted as one inference*. This definition has the advantage of being implementation independent. This is not the case for the definition used in [4].

According to these figures, KCM is between two and four times faster than the initial PLM.

*The cut operation is not counted as an inference

Table 3 compares KCM with one of the best commercial systems, QUINTUS 2.0, running on a SUN3/280 workstation (M68020 25MHz, FPU 20MHz, 16Mbytes of main memory).

The benchmark programs are not exactly the same as in the previous paragraph. All the I/O predicates (used to print the solutions) have been removed in order to measure the pure inferencing capabilities of both systems. This gives less inferences as well as less Klips. The Klips were, in the previous table, somehow artificially inflated because of the assumptions made on built-in predicate timings.

The holes in the table correspond to programs which were too small to get significant results. On the remaining programs, some important discrepancies were also sometimes noted between successive runs of the same program. For each program, the figure given here is the best figure obtained on 4 successive runs on a quiet system (benchmarking was the only user activity).

According to these measurements, KCM is on average almost 8 times faster than QUINTUS. The ratios range from 5.08 (nrev1) to 10.17 (query). As expected, the lower ratios are obtained for intrinsically deterministic programs for which no backtracking is required. As soon as the execution backtracks, higher ratios are observed. The highest ratio is actually obtained on query, a small database-like query model, showing the efficiency of KCM indexing.

Note however that these figures are slightly biased since Quintus does not allow the integer arithmetic and static linking optimisations. However we do not expect generic arithmetic and dynamic linking to slow down programs significantly, thanks to the use of multi-way branching for generic arithmetic and fast indirect calls via memory (4 cycles). Actually, some programs, e.g. query, will even be speeded up with generic arithmetic (floating arithmetic is significantly faster than integer arithmetic on multiplications and divisions).

Benchmark		QUINTUS		KCM		Q/KCM
Program	Inferences	ms	Klips	ms	Klips	ms/ms
con1*	4			0.006	666	
con6*	12			0.046	261	
divide10*	20			0.090	222	
hanoi*	767	11.600	66	1.264	607	9.18
log10*	12			0.039	308	
mutest*	1365	41.500	33	4.644	294	8.94
nrev1*	497	3.300	151	0.649	766	5.08
ops8*	18			0.058	310	
palin25*	323	9.330	35	1.220	265	7.65
pri2*	1233	30.500	40	5.239	235	5.82
qs4*	610	11.000	55	1.315	464	8.37
queens*	657	9.010	73	1.182	556	7.62
query*	2888	128.170	23	12.605	229	10.17
times10*	20			0.081	247	
average						7.85

Table 3: Comparison with QUINTUS/SUN

4.3 Comparisons with other Prolog Machines

Let us sacrifice to the Klips tradition (we could not escape it in the abstract either), mainly to say that it has not much sense. Table 4 compares the peak performance of a number of major Prolog machines, CHI-II [6], DLM-1 [14], IPP [10], AIP [9], KCM, PSI-II [11] and the successor of PLM, X-1 [3]. The first figure gives the performance of a con1-like program (concatenation of two lists), the second of a nrev1-like (naive reversal of a list).

Machine	By	Klips	Word	Comment
CHI-II	NEC C&C	490 - ?	40	Back-end - multi-processing
DLM-1	BAe	800? - ?	38	Back-end - physical memory
IPP	Hitachi	1360 - 1197	32	Integrated in super-mini (ECL)
AIP	Toshiba	? - 620	32	Back-end
KCM	ECRC	833 - 760	64	Back-end
PSI-II	ICOT	400 - 320	40	Stand-alone - multi-processing
X-1	Xenologic	400?	32	SUN co-processor

Table 4: Comparison with other dedicated Prolog machines

The problem is that there is no standard way to compute these figures. It is especially noticeable on list concatenation. There are two ways to compute Klips there. Either, the whole execution is taken into account, including the generation of input data, or only the basic inferencing step, i.e. the concatenation of one more element, is taken into account. In the first case, further questions are: What is the length of the lists? Is the top-level goal counted as an inference? In Table 4, we used, as CHI-II, the second method (one concatenation step is 15 cycles). Needless to say, it does not give the worst figure.

In spite of these problems, it is clear that KCM compares favorably with other dedicated machines in terms of peak performance. Not taking into account IPP which is implemented in ECL, DLM-1 is the only machine which is claimed to reach the same level of performance as KCM. It is also clear that first, peak performance is not an accurate measure of the efficiency of a system, and second it does not give any hint on its usability. With respect to these points, we think that KCM has definitive assets over DLM like a much simpler design, virtual memory, support of SEPIA (including modules, coroutining, events...).

In all cases, a fair comparison (outside the scope of this paper) should also take into account such points as the system environment of the machine (back-end, co-processor or stand-alone, single or multi-user, multi-processing), its size (chip, desktop, deskside), its memory system (caches, main memory, paging), its technology, the status of the project (paper design, prototype, product, software available)...

5 Future work

Much work remains to be done. On one hand, pilot machines are to be produced in 1989, as a joint effort of ECRC and its three shareholder companies Siemens, Bull, and ICL. On the other, a complete system is expected to be operational by the second quarter of 1989. This will include full host connection including paging and a complete Sepia environment running on KCM (incremental Prolog compiler, modules, coroutining...). The pilot machines will be delivered to selected customers in Europe.

Further evaluation studies will be conducted in parallel with this work, to get proper figures on the influence of each specialized unit (trail, dereferencing, RAC, double port register file...), on the overall performance and on the behaviour of the system on real-size programs.

In the longer term, addressing the problems of C compilation

on KCM and of the implementation of constraints is contemplated.

Acknowledgement

Special thanks are due to the Sepia team at ECRC for its extensive collaboration, and particularly Micha Meier for his time and expertise. We acknowledge the invaluable support of Bill O'Riordan at ICL, Chairman of the KCM Project Board. The KCM Project could not have reached its current state without the strong backing of Peter Mueller-Stoy and Walter Woborschil at Siemens, Francois Salle and Francois Anceau at BULL, and many other people in ECRC's Shareholder companies.

References

1. H. Benker, J. Noye, G. Watzlawik. ICM4. Technical report CA-25, ECRC, February, 1987.

2. Gaetano Borriello, Andrew R. Cherenson, Peter B. Danzig and Michael N. Nelson. RISC vs. CISCs for Prolog: A Case Study. ASPLOS II, IEEE Computer Society, October, 1987, pp. 136-145.

3. Tep Dobry. A Coprocessor for AI; LISP, Prolog and Data Bases. Proceedings of Spring Compcon' 87, IEEE Computer Society, February, 1987, pp. 396-402.

4. T.P. Dobry, A.M. Despain and Y.N. Patt. Performance Studies of a Prolog Machine Architecture. The 12th Annual International Symposium on Computer Architecture, IEEE/ACM, June, 1985, pp. 180 - 190.

5. John Gabriel, Tim Lindholm, E.L. Lusk, R.A. Overbeek. A Tutorial on the Warren Abstract Machine for Computational Logic. ANL--84-84, Argonne National Laboratory, June, 1985.

6. S. Habata, R. Nakazaki, A. Konagaya, A. Atarashi and M. Umemura. Co-Operative High Performance Sequential Inference Machine: CHI. Proceedings of ICCD'87, New York, 1987.

7. M. Hill, S. Eggers, J. Larus, G. Taylor, G. Adams, B.K. Bose, G. Gibson, P. Hansen, J. Keller, S. Kong, C. Lee, D. Lee, J. Pendleton, S. Ritchie, D. Wood, B. Zorn, p. Hilfinger, D. Hodges, R. Kerz, J. Ousterhout and D. Patterson. "Design Decisions in SPUR". *Computer 19* (November 1986), 8 - 22.

8. M. Meier, P. Dufresne, A. Herold, D. de Villeneuve. Sepia: An Extendible Prolog System. To appear in the proceedings of IFIP89.

9. S. Kawakita, M. Saito, Y. Hoshino, Y. Bandai, Y. Kobayashi. An Integrated AI Environment for Industrial Expert Systems. International Workshop on AI for Industrial Applications 1988, IEEE Computer Society, 1988, pp. 258-263.

10. K. Kurosawa/ S. Yamaguchi/ S. Abe/ T. Bandoh. Instruction Architecture for a High Performance Integrated Prolog Processor IPP. Proceedings of the 5th International Conference & Symposium on Logic Programming, Hitachi Research Lab., August, 1988, pp. 1507-1530.

11. Hiroshi Nakashima and Katsuto Nakajima. Hardware architecture of the sequential inference machine: PSI-II. Proceedings - 1987 Symposium on Logic Programming, IEEE Computer Society, September, 1987, pp. 104 - 113.

12. J. Noye, J.C. Syre, et al. ICM3: Design and evaluation of an Inference Crunching Machine. Database Machines and Knowledge Base Machines, October, 1987, pp. p. 3-16.

13. M. Meier. Shallow Backtracking in Prolog Programs. Technical Report TR-LP, ECRC, February, 1987.

14. A. Pudner. DLM - A Powerful AI Computer For Embedded Expert Systems. Frontiers in Computing, December, 1987, pp. 187 - 201.

15. M.L. Ross, K. Rammaohanarao. Paging strategy for Prolog based on dynamic virtual memory. TR 86/8, University of Melbourne, August , 1986.

16. Leon Sterling and Ehud Shapiro. *Advanced Programming Techniques.* Volume I: *The Art of Prolog.* The MIT Press, 1986.

17. Evan Tick. *Frontiers in Logic Programming Architecture and Machine Design.* Volume I: *Memory Performance of Prolog Architectures.* Kluwer Academic Publishers, 1988.

18. Herve Touati and Alvin Despain. An empirical study of the Warren Abstract Machine. Proceedings - 1987 Symposium on Logic Programming, IEEE Computer Society, September, 1987, pp. 114 - 124.

19. David H.D. Warren. Implementing Prolog. 49 and 50, University of Edinburgh, May, 1977.

20. David H. D. Warren. An abstract prolog instruction set. tn309, SRI, October, 1983.

21. A. Yamamoto, M. Mitsui, M. Yokota, K. Nakajiama. The Program Characteristics in Logic Programming Language ESP. OKI Electric, 1986.