PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

CARMEL-2: A SECOND GENERATION VLSI ARCHITECTURE FOR FLAT CONCURRENT PROLOG

Arie Harsat and Ran Ginosar

Department of Electrical Engineering
Technion - Israel Institute of Technology
Mt. Carmel, Haifa 32000, Israel

ABSTRACT

CARMEL-2 is a high performance VLSI uniprocessor, tuned for *Flat Concurrent Prolog* (FCP). CARMEL-2 has 13-fold speedup over its predecessor, CARMEL-1, and it achieves 2,400 KLIPS executing *append*. This tremendous execution rate was gained as a result of an optimized design, based on an extensive architecture-oriented execution analysis of FCP, and the lessons learned with CARMEL-1. CARMEL-2 is a RISC processor in its character and performance. The instruction set includes only 29 carefully selected instructions. The 10 special instructions, the prudent implementation of all other instructions, the unique dual-memory system architecture and pipeline scheme, as well as sophisticated mechanisms such as *intelligent dereference*, distinguish CARMEL-2 as a RISC processor for FCP.

1 INTRODUCTION

This paper presents the architecture of CARMEL-2, a high performance VLSI processor for *Flat Concurrent Prolog* (FCP) (Shapiro 1986). FCP, like other parallel logic programming languages such as Parlog (Clark and Gregory 1986) and GHC (Ueda 1985), was designed for parallel and distributed computers. We have selected the former as the language for our parallel logic programming computer CARMEL (Computer ARchitecture for Multiprocessing Execution of Logic programs). In this paper we describe the uniprocessor component of that parallel architecture.

We have employed a structured methodology to obtain the optimal design of CARMEL-2 (Harsat and Ginosar 1987). This methodology extends the well known RISC concept (Katevenis 1984, Gimarc and Milutinovic 1987), by adding new concepts, more appropriate for logic programming languages like Prolog and FCP. Essentially, the methodology consists of the following steps: First, a representative benchmark is analyzed to discover its execution bottlenecks. A *novel* intermediate language is defined to focus the analysis at the desired architectural level. Second, an architecture is designed to accelerate execution efficiently. Third, the result is analyzed, and the architecture is tuned and improved. This last step may be applied repeatedly until converging to a satisfactory final design.

CARMEL-1 (Ginosar and Harsat 1987) was based on the analysis of an experimental early software prototype of FCP. As such, that architecture unavoidably suffered from the disadvantages and inefficiencies of the original software environment. CARMEL-2 processor, presented in this paper, is the second generation. Its architecture was improved based on the lessons learned with CARMEL-1. Furthermore, novel compilation techniques of FCP were employed. CARMEL-2 is estimated to operate 13 times faster than CARMEL-1. It achieves 2.4 MLIPS executing *append*. This tremendous execution rate was gained as a result of an optimized design. The paper presents the architecture of CARMEL-2 and some design decisions.

In Section 2 we discuss the main characteristics of FCP and the findings of FCP execution analysis. The system architecture is described in Section 3. Data types are discussed in Section 4. Section 5 presents the instruction set. The pipeline is discussed in Section 6. The data path is the subject of Section 7. Performance is estimated and compared with some related works in Section 8. Section 9 concludes the paper.

2 FCP AND ITS EXECUTION ANALYSIS

FCP is translated into a program for the sequential FCP *abstract machine* (FAM) (Shapiro 1987). Novel compilation techniques, employing decision trees, are applied (Kliger 1987). FAM is different from the well known Warren Abstract Machine (WAM) (Warren 1983), mainly in being multiprocessing oriented: a process is created in FAM for every new goal, and multiple processes may be active concurrently. Unlike WAM, no environment stack is maintained and there is no backtracking in FCP. In addition, FCP introduces a synchronization mechanism of read-only shared variables; a process suspends when trying to unify an unbound read-only variable.

The FAM architecture includes 16 special purpose registers, and several data structures for process- and memory-management. Data memory is dynamically allocated within a single heap structure. The structure of CARMEL-2 reflects this organization to a certain extent.

An architecture-oriented execution analysis of FCP is presented in (Harsat and Ginosar 1987). The results described here affect the architecture of CARMEL-2, as explained throughout the rest of this paper. The analysis reveals that most of the time is spent performing very few operations. Dereference operations take 22% of the total execution time. Various pointer manipulations account for almost 20% of the time. Type identification consumes over 15%. Call and return overhead of system predicates (called *guards*), takes about 9% of the time. We have found that FCP programs demonstrate a characteristic behavior, independent of parameters like the type of computation, the size of the program, run-time memory requirements, and others. In addition, garbage collection (which is a system service rather than a FCP inherent activity), does not affect significantly the characteristic behavior.

Further findings, not presented in (Harsat and Ginosar 1987), show that while the dereference operation is frequently used, the reference chains are short: the average length is about 1.0, while 99% are less than three-element long. There is no single dominating argument type in FCP. We have also measured the typical (maximal) memory requirements for each of the different system data structures, in order to be able to size our memory efficiently.

FCP, as a non-procedural language, supports no user subroutines. Subroutines are used only to implement system services and guards. They have no local variables, and very few arguments are passed. Hence all calls are to predetermined locations. Similar to Call instructions, most branches are also directed at absolute addresses. This is due to the fact that FCP is free of the concept of programmable control flow: there is no GOTO, etc.

Program execution consists of multiple processes. Each process is described by a record in memory, which identifies its code and data. A ready ("active") queue of processes is maintained. Processes are identified by pointers to these records. The process state consists of merely four CARMEL-2 registers, and process switches are very frequent. The processes are "light weight," in the sense that they switch often and fast.

These findings are the basis for the FCP support in CARMEL-2. It includes an optimized instruction set, and other architectural features, as described in the following sections.

3 CARMEL-2 SYSTEM ARCHITECTURE

Figure 1 describes the unique system architecture of CARMEL-2. Data memory is separate from instruction memory. Both memories may be accessed within a single machine cycle. CARMEL-2 places the instruction and data addresses on the address bus one after the other, in the beginning of the cycle, and each address is captured by the corresponding address latch. At the beginning of the next cycle the instruction is fetched, followed by the required data word, on the data bus. This memory interleaving scheme allows the matching of a fast CARMEL-2 processor with slower memories, without having to slow down the processor.

Additional, relatively small, memory is used as a *stack* to keep return addresses generated by *Call* and *Trap* instructions. An external Stack Pointer (SP) register keeps the top of the stack (data) always ready for reading. There is also an internal SP inside the processor. The Jump/Call/Return detector unit is used for fast decoding and target prefetch of unconditional Jump, Call and Return instructions, and is explained in Section 6.

The *data buffer* buffers data written to memory during *store*, and stack arguments during *push*. It is also used during initialization to write into the instruction memory. Data memory is divided, in a flexible manner, into six main segments. It also contains an initialization segment. See Figure 2.

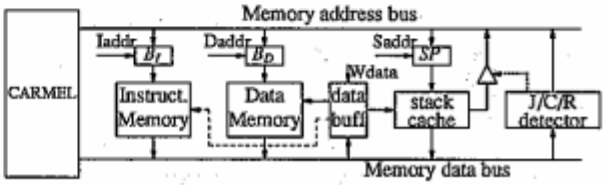

Figure 1: CARMEL-2 System Architecture

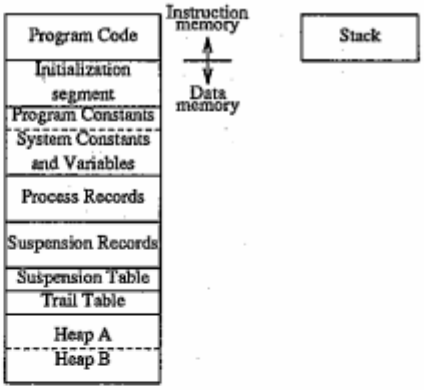

Figure 2: CARMEL-2 Virtual Memory Space

4 DATA FORMATS AND TYPES

There are ten types of data arguments in FCP. Each argument is accompanied by a two- or four-bit tag. Two types, *integer* and *list-integer*, are real data; *nil* has no value, and the remaining seven types are pointers, see Table 1. Our tag allocation allows 30-bit integers, and 28-bit addresses.

The specific tag allocation supports *cdr-coding* of lists. The difference between an element (e.g. integer) and a list of same elements (e.g. list-integer) is just one tag bit (bit 30). Similarly, single bits determine the reference and read-only attributes. Integers are also distinguished on a

basis of a single bit. As a result, a simple special tag identification and mapping hardware is required.

Table 1: Data Formats and Types

argument type	tag	# of data bits
1. *integer*	10	30
2. *list-integer*	11	"
3. *variable*	0010	28
4. *tuple*	0110	"
5. *string*	0100	"
6. *reference*	0001	"
7. *read-only reference*	0011	"
8. *list-reference*	0101	"
9. *list-ro-reference*	0111	"
10. *nil*	0000	0

5 CARMEL-2 INSTRUCTION SET

CARMEL-2 instruction set is optimized according to the execution analysis carried out in (Harsat and Ginosar 1987), and further analysis of CARMEL-1. It also reflects pipeline dependencies, as described in Section 6. The instructions are defined in Table 2.

In the table, we use *tag()* and *value()* to represent the tag separation hardware. *Tag()* returns the tag part of an argument. *Value()* returns a 30-bit integer, if the tag is *integer* or *list_integer*, and a 28-bit value sign-extended to 30 bits otherwise. '||' represents concatenation. If the ALU output is stored in a register, it is concatenated with the tag of the first argument, to produce a 32-bit result. In all

Table 2: CARMEL-2 Instruction Set

S_2 is either a second register or a 16-bit 2's complement immediate operand. Y is a 22-bit 2's complement displacement for PC-relative addressing mode. The Condition Code (CC) register contains four standard flags: N, Z, V and C. R_0 is a hardwired zero. *Tag()*, *value()* and '||' are explained in the text.

Group	CARMEL	Instructions	Function (see Sections 5 and 6 for more details)
Arith-metic	*ADD*	R_s, S_2, R_d	$R_d \leftarrow [value(R_s) + value(S_2)] \parallel tag(R_s)$, *set CC*
	SUB	R_s, S_2, R_d	$R_d \leftarrow [value(R_s) - value(S_2)] \parallel tag(R_s)$, *set CC*
	XOR	R_s, S_2, R_d	$R_d \leftarrow [value(R_s) \oplus value(S_2)] \parallel tag(R_s)$, *set CC*
	AND	R_s, S_2, R_d	$R_d \leftarrow [value(R_s)\ \&\ value(S_2)] \parallel tag(R_s)$, *set CC*
	OR	R_s, S_2, R_d	$R_d \leftarrow [value(R_s) \mid value(S_2)] \parallel tag(R_s)$, *set CC*
Shift	*SLL*	R_s, R_d	$R_d \leftarrow logic\_shift\_left(value(R_s)) \parallel tag(R_s)$, *set CC*
	SRL	R_s, R_d	$R_d \leftarrow logic\_shift\_right(value(R_s)) \parallel tag(R_s)$, *set CC*
	SRA	R_s, R_d	$R_d \leftarrow arith\_shift\_right(value(R_s)) \parallel tag(R_s)$, *set CC*
Load and Store	*LOAD*	$S_2(R_x), R_d$	$R_d \leftarrow M[value(R_x) + value(S_2)]$
	LOADr	Y, R_d	$R_d \leftarrow M[PC + Y]$
	STORE	$Imm<16>(R_x), R_m$	$M[value(R_x) + Imm<16>] \leftarrow R_m$
	STOREr	Y, R_m	$M[PC + Y] \leftarrow R_m$
Flow Control	*JMP*	*Address*	$PC \leftarrow Address$
	JC	*COND*, $S_2(R_x)$	*if (COND) {* $PC \leftarrow value(R_x) + value(S_2)$ *}*
	JCr	*COND*, Y	*if (COND) {* $PC \leftarrow PC + Y$ *}*
	CALL	*Address*	$M[SP] \leftarrow PC, PC \leftarrow Address, SP{+}{+}$
	TRAP	*COND*, Y	*if (COND) {*$M[SP] \leftarrow PC, PC \leftarrow PC + Y, SP{+}{+}$*}*
	RET	R_s, S_2, R_d	$R_d \leftarrow [value(R_s) + value(S_2)] \parallel tag(R_s)$, *set CC*, $PC \leftarrow M[SP], SP{-}{-}$
FCP Special	*InsTag*	R_s, Tag, R_d	$R_d \leftarrow value(R_s) \parallel Tag$
	IfTag	R_s, Tag, S_2	*if* $(tag(R_s) = Tag)$ $PC \leftarrow [value(R_2)$ *or* $PC + Imm<16>]$
	IfNotTag	R_s, Tag, S_2	*if* $(tag(R_s) \neq Tag)$ $PC \leftarrow [value(R_2)$ *or* $PC + Imm<16>]$
	BRonTag	R_s	*10-way branch by* $tag(R_s)$ *The 10 addresses are in* $PC+1, PC+2, \ldots, PC+10$
	Build	R_s, S_2, R_d	$R_d \leftarrow [value(R_s) + value(S_2)] \parallel RefTag$, *set CC*
	BuildM	$R_s, Imm<16>, R_m$	$M[value(R_m)] \leftarrow [value(R_s) + Imm<16>] \parallel RefTag$
	STOvar	$R_m, Imm<16>$	$M[value(R_m) + Imm<16>] \leftarrow value(R_0) \parallel VarTag$
	Deref	Rd_2, F, R_s, Rd_1, Tag	R_s *holds initial pointer* *if (F) stop dereference whenever RoRef is met* $Rd_1 \leftarrow$ *final pointer* $Rd_2 \leftarrow$ *dereferenced value* *continue as BRonTag* *The 10 addresses are in* $PC+2, PC+3, \ldots, PC+11$
	DerefI	$Rd_2, F, N, R_s, Rd_1, Tag$	*similar to Deref* *if (N) continue as IfNotTag else as IfTag* *the branch address is in* $PC+2$
	IfTS	Y	$TS{-}{-}$, *if* $(TS \geq 0)$ $PC \leftarrow PC + Y$ *else* $TS \leftarrow$ *initial value* (set by *SetTS*)
	SetTS	$Imm<8>$	$TS \leftarrow Imm<8>$

other cases, the 30-bit ALU output is a calculated address, which is truncated to 28 bits.

The instruction set includes only 29 instructions. Almost all instructions execute effectively in a single cycle. There are only four addressing modes: register, absolute (immediate), register-based, and PC-relative.

5.1 General Instructions

The general instructions are similar to those of other RISC processors (Gimarc and Milutinovic 1987). However, their specific implementation is unique, and represents the fact that CARMEL-2 is designed for FCP, as explained below.

5.1.1 Arithmetic and Shift Instructions

All arithmetic and shift instructions operate on either two registers or a register and a 16-bit immediate operand. The ALU operates on the value part of the two arguments. The *shift* instructions provide only a single position shift. There are no explicit multiple bit shifts in FCP, and the compiler hardly needs any. Consequently, the data path does not include a *barrel shifter* (Katevenis 1984).

5.1.2 Load/Store Group

It is extremely important in CARMEL-2 to provide fast load and store. The ability to exploit a large register file in order to minimize memory traffic is very limited in FCP, because there are very few local scalar variables that can be manipulated inside registers. Most arguments are global and structured. Thus, many data accesses in CARMEL-2 are addressed to the external memory. Moreover, as mentioned in Section 2, FCP employs multiple "light-weight" process switches. Each process switch includes two register and two memory accesses. To support fast memory accesses we have devised highly efficient Load and Store instructions. Dual memory system architecture is utilized, which allows to execute Load and Store in a single cycle.

For Load and Store instructions, the address calculation components reside in registers, or are specified by a register and a 16-bit immediate. LOADr and STOREr instructions use the PC and a 22-bit 2's complement displacement for address calculations.

5.1.3 Flow Control Instructions

JMP and Call

The unconditional jump (JMP) and Call instructions contain a 28-bit absolute target address. Utilizing the external Jump/Call/Return detector during their instruction fetch cycle, they perform branch in a single cycle. Call exploits the detector in the execution cycle to push the return address onto the external Stack.

Return

Observations of CARMEL-1 code show that an Add instruction always precedes the Return instruction. In most cases, this is due to the "Test" instruction (which is implemented by Add), where Test is used to set condition code flags before the actual return from a guard function (to indicate *success/failure*). In the other cases, the Add serves as "Add," "Move," "Test," "Clear," "Load small (16-bit) immediate," "Increment" or "Decrement" (by any number). We combine each Add and the following Return instructions into a single Add-and-Return instruction. The opcode field contains the opcode of Return, while the remaining fields contain the Add instruction arguments.

Return employs the external detector to perform fast branches. The return address resides at the top of the external Stack, and pointed at by SP. Thus *Ret* can be executed in a single cycle. Since a useful Add instruction is also executed in parallel with Return, *Ret* is effectively executed in `zero-cycles`.

Conditional Jumps

The conditional jump instructions, *JC* and *JCr*, are delayed instructions (Katevenis 1984). Register-based and the PC-relative addressing modes are supported. *TRAP* is another delayed instruction, similar to *JCr*, which effectively executes in a single cycle. *TRAP* is conditional, and employs the PC-relative addressing mode. As in the case of *Call*, the return address is pushed onto the Stack.

5.2 Special Instructions

The special FCP instructions, in addition to the other special architectural features, distinguish CARMEL-2 as an optimized RISC processor for FCP.

5.2.1 InsTag

This is a single cycle instruction. The actual operation performed by *InsTag* is determined by the immediate tag specification. The *InsTag* instruction either inserts the full two- or four-bit tag into the argument, or changes a single bit of the tag. As was explained in Section 4, this single-bit manipulation supports *cdr-coding* by changing an element into a list and vice versa. It also retains flexibility to support other similar operations in the future.

5.2.2 IfTag and IfNotTag

These instructions check whether a certain argument contains the specified immediate tag. The target branch address is generally an "error" address, and in most of the cases the branch will not be taken. Thus the processor predicts that the branch is not taken, and the following instruction is always fetched. Consequently, in most of the cases, IfTag and IfNotTag are single cycle instructions. When the prediction fails, they take two cycles.

IfTag and IfNotTag can also check for group membership: *list, read_only,* and other attributes. The branch address is either in a register, or is computed by adding PC with a 16-bit immediate offset. The latter mode

is the most useful. In CARMEL-1, IfTag was a *skip* instruction. The new CARMEL-2 branch-based IfTag was found more effective.

5.2.3 BRonTag

Another sort of type-identification is carried out by the 10-way Branch-on-Tag *(BRonTag)* instruction. The instruction spans eleven memory words, one for the opcode and the operand, and ten for alternative JMP instructions (one per each tag). It performs a computed branch, according to the tag of R_s. BRonTag takes three cycles.

5.2.4 Build and BuildM

Many a time, a pointer is generated in our codes by adding some offset to a base address. The proper instruction sequence to achieve that is "Add, InsTag." Due to its high frequency, we define the *Build* instruction for this purpose. *BuildM* is similar, except that the resulting pointer is also stored in memory (i.e. it replaces the sequence "Build, Store").

Although BuildM is an exception of the register-to-register instruction set, it is justified based on performance (Section 8), and especially on the fact that no new hardware is required for its implementation.

5.2.5 STOvar

Another memory oriented instruction is *STOvar*, which allocates a new (uninstantiated) variable. The argument is constructed by concatenating a *variable* tag with zero value. The result is stored in memory. STOvar is a simple and very useful single-cycle instruction. It replaces the sequence "InsTag, Store."

Incidentally, a *STOnil* instruction was also considered but discarded. It would store a *nil* argument (zero tag, zero value). Instead, *Store* R_0 achieves the same result at the same speed.

5.2.6 Intelligent Dereference

Deref performs linked list dereference. The number of cycles needed for completion of *Deref* is proportional to the length of the reference chain. In CARMEL-1, Deref required L+1 cycles for execution, where L was the length of the reference chain. For CARMEL-2, we have devised a more efficient dereference scheme, as follows.

Analyzing CARMEL-1 code, we have found that Deref is almost always followed by either BRonTag or IfTag. In both cases, the tested tag is that of the referenced argument. One possible improvement might have been to combine Deref with BRonTag (or IfTag) into one instruction. However, we get no speedup, since in any case we must first wait for the dereferenced value to be fetched from memory, and only then calculate the correct entry address into the BRonTag jump table. Dereference itself takes exactly L+1 cycles. Table-driven jump (BRonTag) takes 3 cycles. Deref+BRonTag as a single or as two consecutive instructions take L+4 cycles in both cases (Deref+IfTag take L+2, since IfTag takes one cycle).

We employ `tag prediction` to speed this process. Our simulations show that there are no dominating types in FCP (although we suspected it would be *list*). Therefore, we cannot gamble on any specific type (unlike SOAR (Ungar *et al* 1984), for example, which gambles on integers). However, we found out that in most cases the FCP compiler can predict the tag by simply testing the arguments of a clause. For example, consider the clauses:

f(X) ← X1=X+1 ...
: X is predicted to be integer.

f(X) ← g(X) ...
: X is most probably a variable.

f(7).
: The related argument is either an integer or a variable. Most often it is a variable.

f([X|Xs]) ← ....
: X is either *list_int, list_ref* or *list_ro_ref*. It is almost always the case that regardless of the list type, they all get the same treatment. In such cases, the compiler predicts "list" (i.e. one of the three, no matter which).

Our compiler predicts tags, and they are included as immediate operands within the Deref instruction. Deref performs dereference, followed by the equivalent of BRonTag. Deref is eleven-word long. The opcode is followed by ten JMP instructions, one for each of the ten tags (similar to BRonTag).

The processor fetches the target instruction according to the prediction, in parallel to following the reference chain. This is possible thanks to the dual instruction/data memory system. When the referenced value is fetched, it is checked for type. If the prediction was correct, the target instruction is executed immediately. Otherwise, the correct target is fetched and executed, at a cost of three additional cycles. Also, similar to Load, Deref is a delayed instruction. A useful instruction is inserted between the opcode (Deref) and the following JMP instructions.

If the tag of the referenced argument is as predicted, then instead of L+4 cycles, *intelligent dereference* takes L effective cycles (2 if L=1). This is significantly shorter than L+4 cycles, with separate Deref and BRonTag. On failure of the prediction, the instruction takes L+3 cycles effectively. Note that our simulations show that most of the reference chains are very short, with length two or less. Thus, the effective time of Deref is only two cycles.

Similar to Deref, DerefI is a combination of dereference and IfTag, or dereference and IfNotTag. A useful instruction follows Deref, and a branch address comes next. DerefI takes the same number of cycles as Deref.

There are still some very rare cases when dereference

is followed by neither BRonTag nor IfTag. We synthesize such dereference instruction by DerefI, specifying the address of the following instruction as the branch address. BRonTag, IfTag and IfNotTag are also used separately from Deref, and hence are included as well.

5.2.7 IfTS and SetTS

IfTS (If Time Slice) instruction employs a special 8-bit TS register. It supports the *tail recursion* mechanism, which reuses the "current process" record a certain number of times (Shapiro 1987). Whenever the father-process spawns its sons, the first son is allocated its father's record. However, to prevent starvation, there is a limit on the number of times tail recursion is allowed. *IfTS* decrements the internal TS register. As long as it is higher than zero, the branch is taken. Otherwise, TS is set to the initial value and the branch is not taken. It is a PC-relative delayed instruction, and executes effectively in a single cycle. IfTS is executed once per process spawn. It replaces the "Add, JCr" sequence.

SetTS is used to set the TS register whenever the current process is either suspended or halted, that is when the regular counting down should be discontinued. *SetTS* is first used by the initialization procedure.

6 CARMEL-2 PIPELINE

The processor cycle is divided into four (unequal) phases, ϕ_1 through ϕ_4. During ϕ_1, a new instruction is fetched from the data bus, while the address of the following instruction is produced on the address bus. Similarly, during ϕ_2, a data word can be either written onto the data lines (on store and push), or fetched in (on load or pop), and, simultaneously, the next data address is generated on the address lines. Arguments are read from the register file on ϕ_3, and all pre-ALU data path logic, such as sign extend and tag separation, complete preparing the arguments for the ALU. ϕ_4 is dedicated to ALU operation and tag mapping. The result can be stored in the register file during either ϕ_3 or ϕ_4 of the next cycle. We provide two examples on how the pipeline operates: Load and Jump.

6.1 Single-Cycle Load

During the first cycle, say C_1, of the LOAD instruction, the data address is computed. See Figure 3.

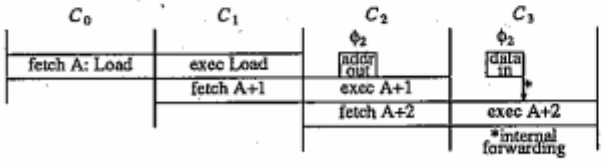

Figure 3: Single-Cycle Load of CARMEL-2; No cycles are lost, thanks to Instruction/Data memory interleaving.

The address is provided to the data memory during ϕ_2 of C_2. The operand is ready and fetched during ϕ_2 of C_3, and is stored into the register file during ϕ_4. Meanwhile, since data and instruction memories are separate, a normal flow of instructions continues to be fetched and executed. Hence, Load takes effectively only a single cycle.

6.2 Single-Cycle Unconditional Jump

JMP instruction contains a 28-bit absolute target address. The external Jump/Call/Return detector (see Figure 1) identifies the JMP as soon as it comes out of the instruction memory. See Figure 4.

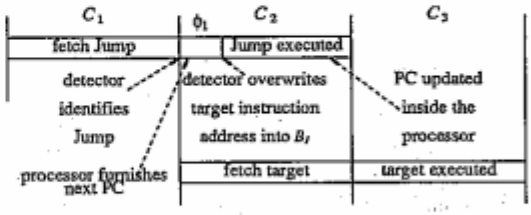

Figure 4: Single-Cycle Jump; No cycles are lost, thanks to the Jump/Call/Return detector.

The absolute address is latched into B_I, overriding the "next PC" which is sent simultaneously by the processor on the address bus. Consequently, the correct target instruction is fetched. Meanwhile, the instruction is also transferred to the processor, and the program counter is updated too. Hence, no cycle is lost, and the unconditional branch instruction is executed in a single cycle.

7 CARMEL-2 DATA PATH

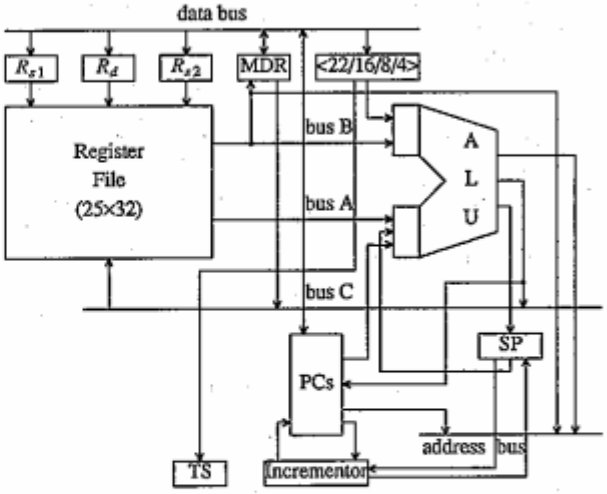

Figure 5: CARMEL-2 Data Path

Figure 5 shows the data path of CARMEL-2. The twenty five 32-bit registers file is organized as a single window. As explained in Section 2, FCP, unlike procedural languages, does not employ user subroutines. The system subroutines almost always use global data, and the number of arguments transferred by subroutine calls is close to zero, and very few local variables are used by sub-

routines. Consequently, multiwindow register file (Katevenis 1984) is unnecessary in CARMEL-2.

CARMEL-2 does not need a large number of general purpose registers for local scalar variables. As also mentioned in Section 2, scalar integer variables are not the dominating type in FCP. The number of general registers required is even smaller because return addresses are held in the external stack. Consequently, all register allocation is predetermined at compile time, and 25 registers suffice.

8 PERFORMANCE EVALUATION

The VLSI layout of CARMEL-1 was completed lately. Some logic and timing analysis have been carried out, and provide the basis for performance estimation. An instruction level simulator has been constructed. The simulator computes run-time statistics such as instruction frequency and execution time distribution, memory access patterns, register utilization, type distribution, and others. In addition to the simulator, a compiler, an assembler and a debugger were also written.

CARMEL-2 is designed for implementation in 1.25μm CMOS. Thanks to the very regular design, the critical path consists of only 12 gates. Using 2 nsec delay per gate as a conservative measure and leaving additional margins for phase transitions and driving internal busses and external loads, we estimate that the cycle will be 32 nsec. Since (almost) each instruction takes a single cycle, this amounts to 31 MIPS. Measuring logical inferences while executing *append*, we get 2400 KLIPS with our new optimizing compiler. This is 13-times faster than CARMEL-1. About 3-fold speedup is due to the novel compilation technique. Also, CARMEL-2 has a shorter critical path (32 nsec vs. 36nsec of CARMEL-1). The remaining speedup is due to optimization of the Call-Return mechanism, and the extension and modification of FCP special group instructions (see Table 2). A significant speedup is due to *intelligent dereference* mechanism. Reported performance for some other processors running *append* is shown in Table 6. Various features of the processors are shown as well.

We have also performed a partial analysis, to measure the contribution of special CARMEL-2 instructions and the tag separation and mapping circuits, to the execution rate. Table 5 shows the incremental growth in KLIPS and the speedup in percents by introduction of various groups of instructions, one at a time, as achieved for *append*. We also show the number of execution cycles for each of the cases. To arrive to these results, we have made six different compilations of *append*, as follows:

1. All instructions, excluding those of the "special FCP" group. See Table 2. The automatic tag separation and mapping hardware is not included as well.
2. As in 1, but the tag hardware is used.
3. As in 2, including: InsTag, IfTag, IfNotTag, BRonTag and CARMEL-1 Deref instruction.
4. As in 3, but with the intelligent dereference instructions (Deref and Derefl).
5. As in 4, and including Build, BuildM and STOvar instructions.
6. As in 5, and including the IfTS and SetTS instructions.

Table 5 summarizes the results.

Table 5: Evaluation of Incremental Contribution of special FCP support in CARMEL-2

Case #	Total cycles per reduction	Execution rate [KLIPS]	Incremental improvement
1.	63	496	
2.	40	781	57%
3.	24	1,302	67%
4.	19	1,644	26%
5.	16	1,953	19%
6.	13	2,403	23%

9 CONCLUSIONS AND FUTURE WORK

We have shown how the results of a structured analysis of FCP have led us to the design of an efficient processor optimized for the language. CARMEL-2 is the second generation processor, and is based on the lessons learned with CARMEL-1. CARMEL-2 shows performance of over 2400 KLIPS executing *append*.

Special support for FCP in CARMEL-2 includes: Tag separation and mapping hardware, a well-chosen set of special instructions and mechanisms such as *intelligent dereference* with *tag prediction* scheme, switch-on-type and single type identification instructions, cdr-coding, support for tail recursion, etc. Also, the pipeline supports fast process switches by utilizing fast load and store instructions. Smart control flow mechanisms avoid pipeline suspensions, and provide single-cycle jumps and calls, and zero-cycles return. Each and every processor feature is included only because it specifically supports some feature of the language, and it improves overall performance.

Meanwhile, a prototype of CARMEL-2 is being implemented in CMOS 1.25 μm. The simulations are also being used for analyzing CARMEL-2, in order to further improve its architecture and performance, together with improvements of the compiler.

Our most important goal is to investigate parallel architectures and the issues of distributed computation in FCP. We examine shared memory and message based schemes, exploiting AND-parallelism.

ACKNOWLEDGMENTS

The invaluable help of Ehud Shapiro, the designer of FCP, is gratefully acknowledged. We also want to thank Shmuel Kliger and Avi Natan for the fruitful discussions on FCP, and its compilation to CARMEL-2.

Table 6: CARMEL-2 vs. Related Processors

Processor	Lang.	Perf.[1] [KLIPS]	R/M[2]	SA[3]/ CP	Imp.[4]	Word bits	Int bits	Tag bits	Multi way branch	Branch on single tag	Deref Inst.	Reference
Low Risc	Prolog		R	SA	No	32	28	3	Yes	No	Yes	Mills 1987
HPM	"	280 E	M	CP	Yes	36			Yes			Nakazaki *et al* 1985
PSI-II	KL0	400 M	M	SA	Yes	40	32	8		Yes		Nakashima and Nakajima 1987
PLM	Prolog	425 E	M	SA	Yes	32	26	2			Yes	Dobry *et al* 1985
CHI-II	"	490 E	M	SA	Yes	40	32	8		Yes	Yes	Habata *et al* 1987
RPM	"	526 E	R	SA	No	32	27	3	No	Yes	No	Cheng *et al* 1987
IPP (ECL)	"	1,000 M	M	CP	Yes	32	28	4	Yes	Yes		Yamaguchi *et al* 1987
CARMEL-2	FCP	2,400 E	R	SA	No	32	30	2,4	Yes	Yes	Yes[5]	

Note 1: Performance executing *append*. E-estimated, M-measured.
Note 2: Risc or Microcoded.
Note 3: Stand-Alone or Co-Processor.
Note 4: Implemented.
Note 5: CARMEL-2's *Intelligent Dereference*.

REFERENCES

Cheng, C. Y., Chen, C., and Fu, H. C., "RPM: A Fast RISC Type Prolog Machine," *CompEuro VLSI and Computers*, pp. 95-98, IEEE, Hamburg, 1987.

Clark, K. L. and Gregory, S., "Parlog: Parallel Programming in Logic," *ACM Tran. on Programming Languages*, vol. 8, No. 1, 1986.

Dobry, T. P., Despain, A. M., and Patt, Y. N., "Performance Studies of a Prolog Machine Architecture," *Proc. of the 12th Int'l Symposium on Computer Architecture*, pp. 180-190, 1985.

Gimarc, C. E. and Milutinovic, V. M., "A Survey of RISC Processors and Computers of the Mid-1980s," *IEEE Computer*, pp. 59-69, Sep. 1987.

Ginosar, R. and Harsat, A., "CARMEL-1: A VLSI Architecture for Flat Concurrent Prolog," *Proc. of the Int. Workshop on VLSI for Artificial Intelligence*, Oxford, July 20-22, 1988.

Habata, S., Nakazaki, R., Konagaya, A., Atarashi, A., and Umemura, M., "Co-Operative High Performance Sequencial Inference Machine: CHI," *Int. Conf. on Computer Design*, pp. 601-604, IEEE, Oct. 1987.

Harsat, A. and Ginosar, R., "Architecture-Oriented Execution Analysis of Flat Concurrent Prolog," *Technion EE pub. Technical Report*.

Katevenis, M. G. H., "Reduced Instruction Set Computer Architectures for VLSI," *ACM Doctoral Dissertation Award*, MIT Press, 1984.

Kliger, S., "Towards A Native-Code Compiler for Flat Concurrent Prolog," *Weizmann Institute MSc Thesis*, Rehovot, Israel, 1987.

Mills, J. W., "Coming to Grips with a RISC: A Report of the Progress of the LOW RISC Design Group," *Computer Architecture News*, vol. 15, pp. 53-67, SIGARCH, Mar 1987.

Nakashima, H. and Nakajima, K., "Hardware Architecture of The Sequencial Inference Machine: PSI-II," *Proc. of the Int. Symposium on Logic Programming*, pp. 104-113, IEEE, San Francisco, Sep. 1987.

Nakazaki, R., Konagaya, A., Habata, S., Shimazu, H., Umemura, M., Yamamoto, M., Yokota, M., and Chikayama, T., "Design of a High-speed Prolog Machine (HPM)," *Proc. of the 12th Int'l Symposium on Computer Architecture*, pp. 191-197, 1985.

Shapiro, E., "Concurrent Prolog: a Progress report," *IEEE Computer*, pp. 44-58, August 1986.

Shapiro, E., *Concurrent Prolog - Collected Papers*, vol. 2, pp. 511-574, MIT Press, 1987.

Ueda, K., "Guarded Horn Clauses," Technical Report 103, ICOT, 1985.

Ungar, D., Blau, R., Foley, P., Samples, D., and Patterson, D., "Architecture of SOAR: Smalltalk on a RISC," *11th Int'l Sym. on Computer Architecture*, pp. 188-197, ACM, June 1984.

Warren, D. H. D., "An Abstract Prolog Instruction Set," *SRI Technical Note 309*, SRI International, 1983.

Yamaguchi, S., Bandoh, T., Kurosawa, K., and Morioka, M., "Architecture of High Performance Integrated Prolog Processor IPP," *Proc. of Fall Joint Computer Conf.*, pp. 175-182, IEEE, Oct. 1987.