MARS- A RISC-based Architecture for LISP

Hung-Chang Lee, Feipei Lai, Jenn-Yuan Tsai, Tai-Ming Parng, and Yu-Fang Li

Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan, R. O. C.
Tel: 886-2- 3635251 ext 241

Abstract

A RISC-based chip set architecture for Lisp is presented in this paper. This architecture contains an instruction fetch unit (IFU) and three processing units -- integer processing unit (IPU), floating-point processing unit (FPU), and list processing unit (LPU). The IFU feeds instructions to the processing units and provides the branch handle mechanism to reduce branch penalty; the IPU is optimized for integer operations, string manipulation, operand address calculations, and some cooperation affairs for constructing a multiprocessor architecture; the FPU handles the floating point data type, which conforms to IEEE standard 754; and the LPU handles Lisp runtime environment, dynamic type checking, and fast list access. In this architecture, not only the critical path of complex register file access and ALU operation is distributed into LPU and IPU, and the tracing of a list can be done fast by the non-delayed *car* or *cdr* instructions of LPU. But also, by using a new branch control mechanism (called branch peephole), this architecture can achieve almost-zero-delay branch and super-zero-delay jump. Performance simulation shows that this architecture would be about 4.1 times faster than SPUR and about 2.2 times faster than MIPS-X.

I. Introduction

Lisp, due to its extensibility and flexibility, has gained popularity these days. Nevertheless, Lisp programming language has some features that are difficult to implement efficiently on conventional computers. These features include frequent function calls, slow list traversal, scope issue of special variable, polymorphic operations, and automatic garbage cell recovery[1,2,6].

The Lisp machines, according to Pleszkun's[1] classification, can be divided into three classes. First, the unspecialized stack-based microcoded Lisp processors (e.g. Symbolics 3600[2], Lambda[3]). Second, multiprocessor architectures where each processor serves a specialized function (e.g. Fairchild FAIM-1[4]). Third, multiprocessor systems composed of pools of identical processing elements aiming for high performance through concurrent evaluation of different parts of a Lisp program on separate processors (e.g. EM-3[5]). Another class of Lisp machine designed recently is RISC-like architecture with some enhancements to support Lisp such as SPUR[6], or by appealing to compiler to reduce the hardware complexities such as MIPS-X[7-9].

A limited instruction set suitable for Lisp execution are presented, and a RISC-based architecture, based on this instruction set, is designed. In fact, the architecture model of *MARS* has three folds. The first fold is the Lisp environment administrator and list traveling access. The second one is the general computational unit, and the final one is instruction feed and control transfer unit. Each fold has respective chip to carry its task. These synchronized chips take advantage of instruction format parallelism and get parallel execution whenever possible.

The next section gives an overview of the systems. This follows by a description of the micro-architecture and instruction pipeline of *MARS* in section III and IV. Section V follows with a performance evaluation using the simulation tools. Conclusion and status are stated in the last section.

II. System overview

MARS [19] is a VLSI processor board for Lisp processing. Inside each board, shown in figure 1, there are CPU chips, i.e., IFU, (Instruction Fetch Unit) and IPU (Integer Processing Unit) as well as special chips, FPU (Floating-point Processing Unit) and LPU (List Processing Unit), Each processor board separate Instruction, Address, and Data buses.

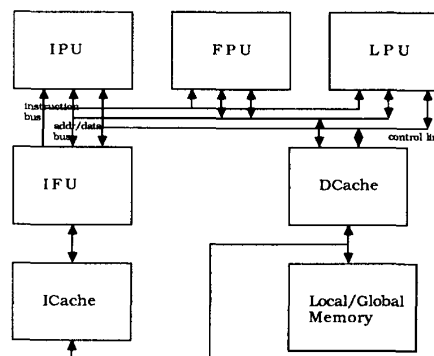

Figure 1. MARS board level block diagram

IFU, built on a single chip together with a 32-kilobyte instruction cache, is the buffering, controlling mechanism between the instruction cache and the datapath chips (IPU, FPU, LPU). It is designed to interleave instruction fetch and execution and achieve coordinated execution among IPU, FPU, and LPU. The block diagram is given in figure 2, in which there are a remote PC (Program Counter) chain, a displacement adder, a return address stack (RAS) to store PC for call/return instruction pair and dual instruction buffers for holding sequential and branch instruction streams.

IPU, shown in figure 3, retains the integer datapath and some control part of a common RISC CPU[10], performing integer arithmetic, shift, logical operations, and address calculation for data operands of all datapath chips. There are a flat 32 word register file, a 2-level internal forwarding latch, and a shifter. FPU conforms to IEEE standard 754. It has

1984/89/0000/0198$01.00 © 1989 IEEE

separate, pipelined Add/Sub and Mul/Div units to provide spatial and temporal parallelisms, a hardwired control unit to directly support for hardware format conversion, a synchronous interface protocol to tightly couple with other chips, and a 32-word 64-bit register file.

Figure 2. The IFU block diagram

Figure 3. The IPU block diagram

LPU, shown in figure 4, provides hardware primitives for list processing, such as *car, cdr, cons, rplaca* (replacea), *rplacd* (replaced). It is featured with a big windowed register file to expedite procedure call, a tag manipulation datapath, and a control register for shallow binding.

CCMMU (Cache Controller and Memory Management Unit) is responsible for the operation of local data cache on each processor board, addressing translation, and data coherency protocol among processors. Local data cache will be as large as 128 KB and data will be heuristically prefetched in the face of pointer or list. Lock-up free cache design together with the cache coherency protocol- phoenix- are proposed in [17].

Figure 4. The LPU block diagram

Why Separate IFU, IPU, and LPU

At first, we intended to adopt integrated LPU architecture like Symbolics 3600. but owing to the unique configuration and some other consideration inside *MARS* (*MARS* can run conventional language, for instance C, in different way)[19], an on-chip instruction cache inside LPU will not provide any speed advantage when all the chips wait for instructions coming from the instruction cache and then decode their own instructions. Besides, some extra problems incurring with the design of on-chip cache has been deeply discussed in [24-25]. Therefore, we decide to separate the instruction fetch unit for conventional LPU, and build IFU to accommodate the remote PC unit and the necessary logic for buffering and control of instruction access. This decision did cause some problems, thereby influenced our designs for instruction set and microarchitecture, but it provided us valuable experiences to deal with such arrangement and the result can still meet our initial requirement.

III. Micro-Architecture

A. Instruction set

The instruction set of LPU[23], shown in Table 1, was carefully designed to speed up the execution of Lisp program and to reduce the time wasted by traffic between LPU and IPU/FPU. In order to reach this goal, there are some instructions executed parallel by LPU, IPU and/or FPU.

Parallel execution of IPU and LPU happens when IPU executes some ALU operation and LPU checks the corresponding tag of source registers and generals exception if data type of operands are neither fix number nor character. Another parallel execution happens when LPU loads or moves data, used by IPU or FPU, into a frame register. Besides IPU and FPU, LPU also writes this data into the corresponding register. With this parallel execution, we need not take extra instructions to transfer data from LPU to IPU or to FPU. The instructions of LPU with this kind of parallel execution are *car*, *pop*, *load* and *mov*.

A.1 List primitives

The hardware implemented list primitives are *car*, *cdr*, *cons*, *rplaca* and *rplacd*. The *car* and *cdr* instructions are similar to load instruction except that the least two significant bits of address are masked with 00 or 01, respectively. In LPU pipeline stage, there is no delay for load instruction, so the trace of a list could be done fast by *cdr* and *car* instructions. The *rplaca* and *rplacd* instructions are similar to the *store* instruction except masking the address bits the same as in *car* and *cdr*. The *cons* instruction executes the action of *rplaca* by using global register R7 (free cons cell pointer) as address and moves R7 to destined register. A complete cons primitives in Lisp could be done by a *cons* instruction following a *rplacd* instruction to replace the cdr of this cons cell and a *car* instruction to update the free cons cell pointer. All above instructions are executed with parallel type checking. If the data type of source address register is not *cons* or *nil*, it will result in an exception.

A.2 Stack operation

The stack operation can be done in one instruction cycle by the *push* and *pop* instruction of LPU. Stack pointer register (SP) in LPU which decreases by 2 (distance of double-word) before executing the push instruction and increases by 2 after executing the pop instruction automatically. The *push* and *pop* instructions are mainly used when binding or unbinding the special variables and when saving or restoring frame windows. The *pop* instruction, as *car* and *cdr*, is a non-delay load instruction which can supply data for the next LPU instruction without delay. The content of stack pointer register can be read or written by *rd_sp* or *wr_sp* instructions.

A.3 Data and tag transfer

The data transfer instructions include: *load* instruction which loads data from memory, *mov* instruction which moves data from a register to another register, *load_f* and *store_f* instructions which generate address for FPU and *f_to_l* instruction which transfers data from FPU to LPU. Note that the *load* and *mov* instructions are parallel executed by IPU, FPU and LPU when the destination is a frame register and the action of transferring data from LPU to IPU or to FPU can be done by the *mov* instruction.

The tag value of a register can be loaded with the immediate tag value packed in instruction or moved from another register. The data field of destined register could be the destined register itself or another register which is the second source register specified in the instruction.

A.4 Compare & branch

In the *MARS* system, comparison of two operands and branch are executed in one instruction cycle with zero or one delay. The *compare & branch* instruction of LPU compares two operands with eight kinds of conditions encoded in 3-bit condition code and sends the compared result to the other processing units.

A.5 Function call and return

The actions of jumping to target address and saving of program counter of function call and return instructions are done by IFU, while LPU updates the pointer of current frame window and checks whether control register file being overflow/underflow or not. If an overflow or underflow happens, LPU causes an exception.

A.6 Special instructions

There are some special instructions of LPU which could be executed only in kernel mode. The *rd_lpsw* and *wr_lpsw* instructions transfer data between registers and LPU processing status word which contains the current window pointer, saved window number and some system status. The *lpu_wake* and *lpu_sleep* instructions set LPU to be active or inactive. When LPU is inactive, the *MARS* system is acting as a general purpose computer without hardware support for Lisp.

B. Tagged List and Data representation and storage

MARS represents List pointer or immediate data with a 38-bit tagged word consisting of a 6-bit tag and a 32-bit pointer or data. The data types represented by tag are shown in Table 2.

There are three immediate data types: character, fix number and short floating-point number. The first two are used by IPU and the last one is used by FPU. We decided to represent short floating-point number by an immediate type because all the IPU, FPU and LPU have a user-view of 32-word control register file. The register files are monitored by LPU as a frame window and saved in the LPU's frame window while executing function call. The 32-bit short floating-point number has the same width as the data field of the LPU's register; therefore, it can be used as an immediate data and stored in the LPU's frame window.

The type of data which is one of the following types: an immediate number, an indirect number, or a list is recognized by a type-checking hardware in parallel with the execution of data itself -- usually under the assumption of integer type, or with the comparison of LPU-compare-branch instruction. The other sixteen tag values are defined by compiler to identify pointed object such as string, vector and function.

Poor data density

MARS architecture was design with emphasis on speed and simplicity rather than on data or hardware density. Board level data bus are 64 bits because of the intended one data fetch per cycle for the double floating point. The memory implementation particularly has poor Lisp data density since a 64-bit system is used instead of a brand new 38-bit one.

Several approaches exist for handling 38-bit data type:

(1) build the whole system with 38-bit words,

(2) allow unaligned cache access, or

(3) place 38-bit words in aligned 64-bit words.

We adopt the last one because of three considerations. First, many off-the-shelf subsystems use 32 bit words as an integral unit. Second, unaligned cache accesses would increases the cache cycle time and therefore, decreases the system performance. Third, since MARS runs other conventional languages (e.g. C and Fortran), a 32-bit words integral unit memory system is preferred.

MARS stores the 38-bit tagged word in memory by two 32-bit words aligned in a double-word boundary (data in even word and tag in odd word) with 26 bits unused. The data bus connecting four processing units and cache memory is 64-bit wide -- 32 bits are needed by the IFU and IPU, 64 bits by the FPU and 38 bits by the LPU. This means that we can load/store one or two 32-bit words in one memory cycle.

A *list cons cell* consisting of car's tagged word and cdr's tagged word is represented by four contiguous 32-bit words

aligned in the quad-word boundary of memory. The first half of the double-word contains the car's tagged word and the second one contains the cdr's tagged word. We can access the car's tagged word of a cons cell by setting the least four significant bits of the cell's address to 0000 and access the cdr's tagged word by setting these four bits with 1000.

C. Registers organization

The registers organization in LPU plays the role as a runtime environment administrator. In Lisp, the arguments, local variables and special variables are accessed frequently. These variables are allocated in register file and maintained in a fast scheme described later. There are two kinds of register files in LPU, one is the control register file and the other is the binding register file. The control register file organized as an overlapping frame window structure is used to keep the activation records of callers and callees. The 32-word register file of IPU and LPU are mapped to one of the frame windows, so user can view the control register as a 32-word register frame window whose data may be either integer (in IPU), floating-point number (in FPU), or pointer (in LPU). The binding register file is used to provides the dynamic scope of special variables and keep the binding value of theirs. We use shallow binding scheme to bind and restore the special variables.

C.1 Control register file

The control register file is organized as multiple, overlapping, fixed-size frame windows (shown in figure 5), similar to the multi-window register file of RISC I&II [10]. However, the control register file differs to that of RISC II in that it has to monitor the 32-word register files of IPU/FPU and keep their contents in the corresponding frame windows in function calling. We designed a mechanism, which will be described later, to map the registers of IPU/FPU into the control register file of LPU across function call/return. There are 8 frame windows and totally 136 registers in the control register file, but only 32 registers (one frame window) can be seen by user at a time. The user's view of 32-word registers is further partitioned into four 8-word register sub-frames -- i.e., global, input, output, and local. The partition of the 32-word registers differ to that of RISC II, which has 10 registers for global, 6 for input, 6 for output and 10 for local, for the reasons that Lisp uses more arguments and fewer local variables than C or Pascal, and this arrangement makes the mapping of IPU/FPU's registers to the control register file easier. The global frame shared by all windows is used to hold some environment variables such as return value and pointer to the top of heap memory. The input frame is intended to place input arguments from parent function (caller). On the other hand, the output frame is used to hold and send arguments to child function (callee). The output frame of caller is overlapped with the input frame of callee. Once a function is called, the window viewed by user switches from caller to callee and the output frame of caller now becomes the input frame of callee. The local registers frame which does not overlap with other window is used to store local lexical variables or temporary values.

Register mapping between IPU and LPU

In the *MARS* system, only LPU has frame-window register structure and there are merely 32 registers in IPU and FPU. How do we keep IPU's and FPU's register data when executing function call? We solve this problem by the following mechanisms. First, the 32-word register file of IPU/FPU is partitioned into four 8-word register groups which are mapped into the four sub-frames of LPU's current frame window. Figure 5 shows the mapping of IPU/FPU's 32-word register into LPU's frame window. *A* group registers (R0 - R7) and *C* group registers (R16 -R23) of IPU/FPU are always mapped into the global frame and local frame of LPU's current frame window. In contrast, *B* group registers (R8 - R15) and *D* group registers (R24 - R31) of IPU/FPU are mapped into the input frame and output frame or vice versa according to the current window number of LPU being even or odd. Assuming that the window number of current function is 0, then *B* group registers are mapped into the input frame and *D* group registers are mapped into the output frame. After calling a child function, the frame window number is increased by one and now the *D* group registers of IPU/FPU are mapped into the input frame of window 1, which is the output frame of window 0 (see figure 6). This means that we do not have to save the 8 registers corresponding to the output frame of current window when executing function call. Likewise, *A* group registers mapped into the common global frame of all frame windows do not have to be saved, so only the remaining two 8-word register groups mapped into local frame and input frame need to be saved into or restored from the corresponding frames of LPU's current window when executing function call or return.

The translation from IPU/FPU's register number to the sub-frames of LPU's current frame window can be implemented easily by the circuit shown in figure 7. The *turn* signal sent from LPU is reset to 0 when the number of LPU's current frame window is even and is set to 1 when frame window number is odd. When *turn* is 0 the translation is an identical one; when *turn* is 1 the translation maps the registers number of *B* group into output frame and maps the registers number of *D* group into input frame.

IPU/FPU		LPU	
A	R0 -R7	GLOBAL	
B	R8 -R15	W0.IN	W7.OUT
C	R16-R23	W0.LOCAL	
D	R24-R31	W0.OUT	W1.IN
C	R16-R23		W1.LOCAL
B	R8 -R15	W2.IN	W1.OUT
C	R16-R23	W2.LOCAL	
D	R24-R31	W2.OUT	W3.IN
C	R16-R23		W3.LOCAL
B	R8 -R15	W4.IN	W3.OUT
C	R16-R23	W4.LOCAL	
D	R24-R31	W4.OUT	W5.IN
C	R16-R23		W5.LOCAL
B	R8 -R15	W6.IN	W5.OUT
C	R16-R23	W6.LOCAL	
D	R24-R31	W6.OUT	W7.IN
C	R16-R23		W7.LOCAL

Figure 5. Frame-window structure of control register file and mapping of corresponding register groups in IPU/FPU

Apart from the above mapping scheme to reduce the overhead of saving and restoring IPU/FPU register data, we save them into LPU's current frame window parallel with the execution of IPU's instruction. LPU monitors all the instructions executed by IPU. When IPU executes an operation and writes the result back to the register file, it also puts this result on the data bus at the memory cycle. Meantime, LPU receives the data and writes back into the corresponding register of the current frame window. With this mechanism, we need

not save any IPU registers data into LPU while executing a function call, we only have to restore the necessary IPU register data from LPU which would be used before the execution of the next function call or before the end of the current function when the called function returns. This overhead would be about two or three instruction cycles per function call on average.

By using above mechanisms, the register data of IPU could be kept in LPU's control register file with little overhead while executing function call or return. The multiple, overlapping frame window structure of control register file in LPU updates runtime environment very fast. Because LPU does not execute ALU operations, it can spend more time in accessing the complex register file. On the other hand, the IPU which must spend time in executing ALU operations has a simple 32-word register file and can access the register faster.

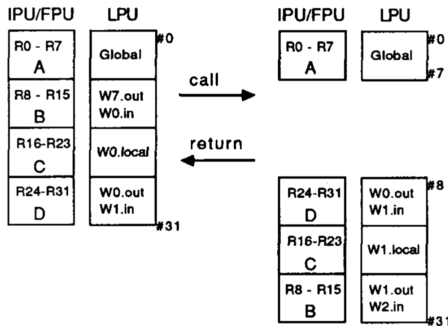

Figure 6. Mapping of frame window before and after executing function call and return

	R(4)	R(3) =>	R(4)	R(3)	
TURN = 0	0	0	0	0	A
	0	1	0	1	B
	1	0	1	0	C
	1	1	1	1	D
TURN = 1	0	0	0	0	A
	0	1	1	1	D
	1	0	1	0	C
	1	1	0	1	B

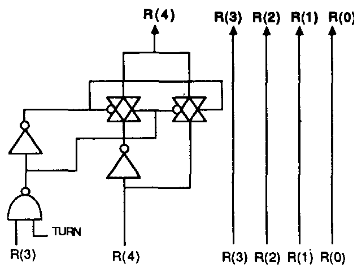

Figure7. Translation of IPU/FPU Register number

C.2 Binding register file

Variables binding in MARS Lisp

There are two kinds of variables in Lisp, the lexical (or static) variables and the special (or dynamic) variables. The lexical variables are known at compile time and is constructed as a stack frame. In contrast, the binding of the special variables are known at run time. Two popular ways of binding are deep binding and shallow binding. Deep binding implementations store the binding value of special variable in stack. When looking up a variable, the stack must be searched until the value is found. In shallow binding, each variable is assigned a global-value cell to store the binding value, while old values are pushed on a stack. In this scheme, variable lookup is very quick. For this reason, most uniprocessor Lisp system use shallow binding.

Binding register for shallow binding

The 32-word binding register file which has no counterpart register file in IPU/FPU is used to store special variables in Lisp. Each special variable corresponds to one register allocated at loading time. We use binding registers in shallow binding scheme to handle the special variables. When a special variable is bound to a new value, the old value in the corresponding register has to be pushed into the restore stack, but when this special variable is unbound, the old value is popped from the restore stack and restored to the corresponding register. An example of binding and unbinding of special variables is shown in Figure 8. The binding registers, which does not have the corresponding register file in IPU/FPU, can only be used by LPU's instructions. If they are to be executed IPU or LPU instructions, they should be moved to the global frame registers. By using the binding registers, we can speed up the access of special variables.

```
(let ((x 3)
      (y 5)
      (z 7))
  (foo x y z))
```

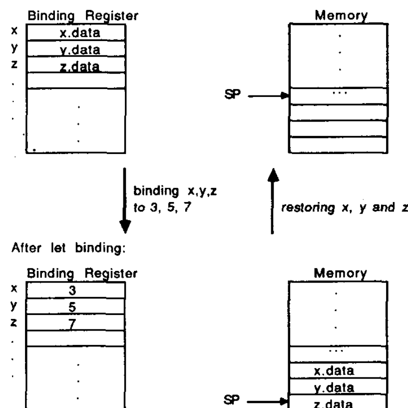

Figure 8. Binding and Unbinding of Special Variable

Binding register file used in multiprocessor environment

MARS is a multiprocessor project. Parallel processing for building multiprocessor environment is currently under intensive research. For multiprocessor Lisps, shallow-binding implementation poses a problem: in the event that multi process es try to change a special-variable binding, how does *MARS* Lisp resolve the conflicting requirements of the different processes without serializing processes which result in the accessing conflict. *MARS* Lisp uses the declaration of special variable primitive (e.g. defvar) to define the special variable. Instead of assigning a global-value cell to store the current binding value, MARS Lisp uses the binding register file for keeping the information. Different processes can get the updated value directly from the register file instead of from the

global shared memory. Therefore, serializing processes which get conflict for the same special variable is not necessary (shown in Figure 9). In the case of the number of special variables excesses the number of binding register file, the extra special variables are kept in the restore stack and copied to the next stack if another function call executes.

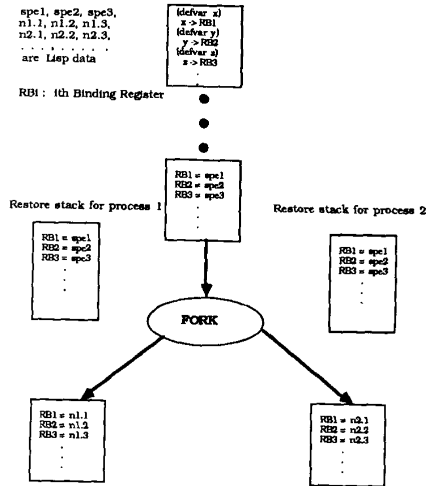

Figure 9. Binding register used in MARS Lisp

IV. Pipeline scheme & Branch handle strategy

MARS uses a fronted instruction fetch stage plus the following two independent four-stage pipelines, shown in figure 10, attempting to issue and complete an instruction every cycle. The instruction fetch stage issued by IFU fetches the following instruction after a non-compare instruction and two instructions (plus control transfer target address) after a control transfer instruction. The two four-stage stages are independent but synchronously executed by IFU and LPU respectively to meet the execution requirements that these two chips demand. For example, one duty of IPU is to determine the branch condition, so that the register value which determine the condition should be fetched as early as possible. On the other hand, LPU does not take this responsibility so that register fetch stage is delayed as far as possible to wait the result of external data reference.

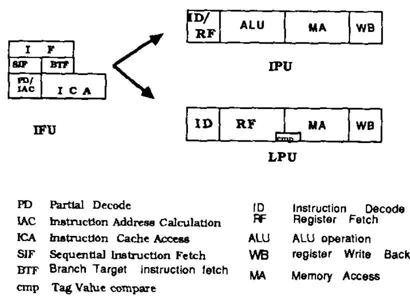

Figure 10. Three kinds of pipeline stage in MARS

Through this kind of pipeline stage arrangement, *MARS* can support almost zero-delay branch, super-zero jump, and a non-delayed list access.

A. Branch peephole -- a Almost-zero-delay branch and super-zero-delay jump control handling strategy

Branch instructions have a considerable effect on the performance of pipelined machines[11-13]. Conventional architectures employ additional hardware to deal with this problem. They detect the presence of a branch instruction and put off prefetching until the branch condition has been evaluated, or use some branch prediction techniques to reduce the number of control flow breaks[14,15]. Recently, a Branch Folding was proposed in the design of CRISP[16], which folded a non-branch and the following branch instruction to get zero-delay branch. A well designed IFU adopt the combination of delayed branch, multiple prefetch and early resolution method to reduce the branch penalty. IFU executes the first two stages of our total 6-stage pipeline and issues instructions to the datapath chips (IPU, FPU, and LPU). Some intelligence exists in the IFU when issuing the instructions flow. The partial decode unit of IFU, executing at the PD (partial decode) stage, can peep out the existence of an incoming jump instruction, calculate the address and access the instruction ahead of time. IFU can absorb that jump instruction and send out the jump target address simultaneously. This mechanism makes a super-zero-delay jump instruction. Moreover, a conditional jump instruction is known in the PD stage, the IFU unit extracts the offset field of the instruction, adds this value to the PC (program counter), and then fetches the branch target to the datapath chips. If *compare* is a fast compare (fcb), we can resolve the *compare* at the early beginning of the ALU stage, that is, settle the branch before IF stage of the next instruction. Therefore, we can obtain a zero-delay branch. In some cases, however, a full compare is necessary, delayed or squashable compare and branch (dcb and scb) are addressed to reduce the penalty of pipeline drain. Experience has pointed out that only 10% of the slots are filled with no-operation instruction[17]. With the combination of fast and full compare and branch (fcb, dcb, and scb) schemes, almost-zero-delay branch effect can be obtained.

B. Non-delayed list access

Most Lisp programs execute list access frequently[18]. List structure is usually constructed with two parts, header of list (*car*) and tail of list (*cdr*). Each of the two parts contains a tag field to identify the data type and a data field. When a *car* or *cdr* instruction is issued, tag field check and data field access are carried out simultaneously. Under LPU delayed RF mechanism, register can be fetched with short cut and incur no internal interlock to the following instruction, illustrated in figure 11. Detailed timing about non-delayed load work is that tag comparison stage (cmp) is executed at the falling edge of phase 2 and the memory access stage (MA) of the previous instruction is also ready at this period so that an internal forwarding from MA stage to cmp stage can be done.

C. Non-delayed function call, jump and return

Procedure calls occur frequently in Lisp program . There are several problems associated with such frequent procedure calls. First, local variables should be kept with each function call and restored from memory on every return. Second, arguments passing must be cross- referred to the memory to and fro. Frame windows are provided in the LPU to keep all these variables in registers to reduce the cost of external memory reference. The input frame in each procedure's frame window overlaps with the output frame of the procedure that

calls it, and the output frame overlaps with the input frame of the procedure that it calls. Local variables are kept in the local frame of the procedure's window, and global ones in the global frame, which is visible as well as shared by the frame windows. This frame-window scheme makes procedure call virtually free, and significantly speeds up their operation. However, the gain is not obtained without any price. Increasing the number of registers means that rather an amount of chip area will be used and register access time will be longer and process switching overhead will increase. Nevertheless, these prices are offset by a dedicated environment maintenance unit, that is, LPU, and an independent execution pipeline stage designed between IPU and LPU.

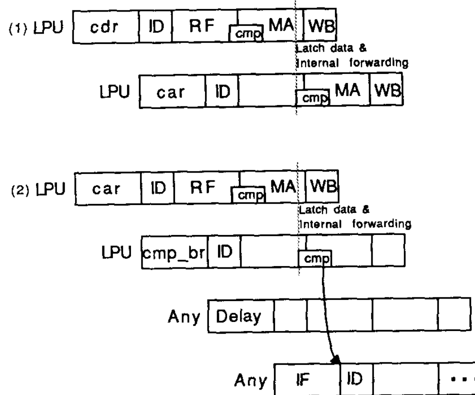

Figure 11. Non-delayed load execution (1) and one-delayed slot list compare and branch (2)

V. Performance Evaluation

Six Gabriel benchmarks - a set of programs which test the speed of Lisp systems in various aspects - have been carried out to compare the performance of *MARS* with other architectures, shown in Table 3. The first column show the results for *MARs*, excluding the effect of cache misses. Column 2, 3, 4, and 5 give the results for the other three architectures; the results for VAX-11/780 are from Gabriel's book[20]; SPUR's results are from Patterson's paper[21]; MIPS-X's results are from Steenkiste's paper [22], and distributed into two columns- one without optimization, the other with optimization. The last four columns give the ratio of the execution time of VAX-11/780, SPUR, and MIPS-X to the execution time of *MARS*. We have adjusted the results for SPUR for a 100-nanosecond cycle time instead of the original 150 nanoseconds because of a new version of SPUR report accordingly [28]. The reason why SPUR, when comparing with MARS, has such long cycle time is due to several reasons: SPUR decodes more instruction set in a chip, the tagging hardware is combined with ALU operation. *MARS* executes Lisp programs about 35.4 times as fast as the VAX-11/780, almost 4.1 times as fast as SPUR, and about 2.2 times as fast as MIPS-X. However, in all these cases, the perforamnce difference varies significantly across the benchmarks. The best one is *stack*, which binds the special variables in the binding register file and can be referenced directly from the binding register file. On the other hand, the benchmarks *iterative-div2* does not run so well as the other benchmarks because this benchmark has a very deep call-depth, window overflow and underflow occur in most of the function call/return and the overhead of saving and restoring frame window actually slows down the execution speed.

	Time in milliseconds					Ratios			
	MARS	VAX	SPUR	MIPS-X	MIPS-X●	VAX/ MARS	SPUR/ MARS	MIPS-X/ MARS	MIPS-X●/ MARS
tak	37	830	80	72	72	22.4	2.1	1.9	1.9
stak	70	7100	710	602	592	101.4	10.1	8.6	8.4
takl	325	5270	552	482	448	16.8	1.7	1.5	1.4
div-iter	55	3800	---	307	157	69.1	---	5.6	2.9
div-rev	340	3750	1950	284	196	11.0	5.8	0.8	0.6
deriv	110	8580	667	604	381	78.0	6.0	5.5	3.5
Geometric mean						35.4	4.1	2.9	2.2

Table 3. Execution times in milliseconds for the Grabriel benchmarks

Note: MIPS-X● mean that Lisp programs executie with optimization.

It is interesting to find the reasons for the performance difference between *MARS* and MIPS-X. Both *MARS* and MIPS-X are RISC processors, and of the same cycle time (i.e. 50 ns) but they differ in that *MARS* has a Lisp environment administrator (i.e. LPU). The LPU has hardware support for tag handling, type checking on lists, binding registers, and uses frame windows to reduce the cost of register saving and restoring, and so forth. The *MARS* hardware for tag handling would eliminate about 25 percent of the cycles on MIPS-X, binding register would also save about 50 percent of the cycles for load and store on MIPS-X, and others (e.g. super-zero-jump, almost-zero-jump, fast list access, etc.) account for the remaining 45 percent. The frame windows do not function well for the Gabriel benchmarks and their average effect is small. The reasons are that some benchmarks use only few arguments and local variables per frame window and have a call sequence straightly backwards and forwards, thus the overhead of saving and restoring frame windows for overflows and underflows which is 16 register-to-memory transfers instead of just several for MIPS-X actually slows down some programs. Neither do the non-delayed *car* and *cdr* instructions in these benchmarks work well when compared with MIPS-X since the delayed slot can always be filled. *MARS* will perform better on more realistic programs or cases without an optimization compiler involved because the frame windows and non-delayed *car/cdr* instructions will be more effective.

VI. Conclusion

A design of chips set for Lisp execution is proposed in this paper. By separating the IFU from the datapaths and our deliberate pipeline arrangement, we can not only get coordinated executions among IPU, FPU, and LPU but also drastically reduce slots due to control transfer; leaving the compiler more chances to fill the delayed load slots, thus accomplishing our goal of single-cycle instruction execution. What is more exciting, we can absorb the jump instructions within the IFU and directly issue the target to datapath chips to achieve what we call the super-zero-delay jump. An independent and separate LPU, playing the role as a Lisp runtime environment administrator, can accelerate Lisp programs with the following reasons. First, long complex register file access can be handled within LPU, without the company of a long ALU stage. Second, Two independent pipeline executions of IPU and LPU can separate the critical path of long register fetch plus integer processing into two independent parts. Furthermore, the LPU can put off the register fetch until the external memory access is ready, and thus no delayed slot is needed when referring the data cache. Third, because instruction decoding of IPU and LPU are local, some frequently used Lisp primitives can be hardwired without increasing the complexity and access time of instruction decode. Fourth, by excluding the arithmetic calculation within

the LPU, the LPU can offer more silicon resource to accommodate more registers, and thus cut down the need of external memory access to increase system performance. Fifth, hardwired primitives can reduce machine cycles needed when, implemented by the underlying machine instruction.

Status

The implement of LPU is in progress. We have described the LPU at the register-transfer level with M modeling language. The layouts of the custom chips will be finished later in this year.

REFERENCES

[1] A. R. Pleszkun, and M. J. Thazhuthaveetil, "The Architecture of Lisp Machines," *IEEE Computer*, Vol. 20, No. 7, Mar. 1987, pp. 35-44.

[2] D. A. Moon, "Architecture of the Symbolics 3600," *Proc. Twelfth Symposium on Computer Architecture*, Boston, June 1985.

[3] *LMI, The Lambda System: Technical Summary*, 1983, LISP Machines Inc.

[4] A. L. Davis and S. V. Robison, "The FAIM-1 Symbolic Multiprocessing System," *Spring 1985 Compcon Digest of Papers*, 1985, pp. 370-375.

[5] Y. Yamaguchi, K. Toda, and T. Yuba, "A Performance Evaluation of a Lisp-based Data-Driven Machine (EM-3)," *Proc. 10th Int'l Symp. Computer Architecture*, June 1983, pp.363-369.

[6] M. Hill, *et al.*, "Design Decisions in SPUR," *IEEE Computer*, Nov. 1986, pp.8-24.

[7] M. Horowitz, *et al.*, "MIPS-X: A 20-MIPS Peak, 32-bit, Microprocessor with On-Chip Cache," *IEEE Journal of Solid-State Circuits*,Vol. SC-22, No. 5, Oct. 1987, pp.790-799.

[8] P. Chow and M. Horowitz, "Architectural Tradeoffs in the Design of MIPS-X," *Proc. 13th Symposium on Computer Architecture*, Jun. 1986, pp. 300-308.

[9] P. Steenkiste, and J. Hennessy, "Tags and Type Checking in Lisp Hardware and Software approaches," *Proc. Second Int'l Conf. Architecture Support for Programming Languages and Operating Systems, ACM/IEEE*, Oct. 1987, pp. 50-59.

[10] M. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, Ph.D. dissertation, Computer Science Division (EECS) UCB/CSD, University of California, Berkeley, Oct. 1983.

[11] S. McFarling and J. Hennessy, "Reducing the Cost of Branches," *Proc. 13th Symposium on Computer Architecture*, Jun. 1986, pp. 396-403.

[12] J. E. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th Symposium on Computer Architecture*, May 1981, pp. 135-148.

[13] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, Vol. 17, No. 1, Jan. 1984, pp. 6-22.

[14] J. Hennessy, *et al.*, "Hardware/Software Tradeoffs for Increased Performance," *Proc. SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, Palo Alto*, Mar. 1982, pp. 2-11.

[15] D. J. Lilja, "Reduced the Branch Penalty in Pipelined Processors,"*IEEE Computer*, Vol. 21, No. 7, Jul. 1988, pp. 47-55.

[16] D. R. Ditzel and H. R. McLellan, "Branching folding in the CRISP microprocessor: Reducing branch delay to zero," in *Proc. 14th Annual Symp. Computer Architecture*, 1987, pp. 2-9.

[17] H.-C. Lee, and C.-E. Wu, "Lock-up free cache design and the phoenix protocol," *NTU-EE-CS memo No. 329-4*, Computer Science Division (EECS), National Taiwan University, Taiwan, R.O.C., Jan. 1989.

[18]. D. W. Clark, "Measurements of Dynamic List Structure Use in Lisp," *IEEE Trans. Software Engineering*, Vol. se-5, No. 8, Jan. 1979.

[19]. G.-S. Jang, F. lai, H.-C. Lee, Y. C. Maa, and T. M. parng, J.-Y. Tsai, "MARS-Multiprocessor Architecture Reconciling Symbolic with Numerical Processing," *International Symposium on VLSI Technology, Systems, and Applications*, 1989. pp. 365-370.

[20]. R. P. Gabriel, *Performance and Evaluation of Lisp System*, The MIT Press, Cambridge, Mass., 1985.

[21] D. Patterson, "A Progress Report on SPUR," *Computer Architecture News, ACM*, Mar. 1987, pp. 15-21.

[22]. P. Steenkiste, and J. Hennessy, "Lisp on a Reduce Instruction Set processor: Characterization and Optimization," *IEEE Computer*, Vol. 21, No. 7, June 1988, pp. 34-45.

[23] J.-Y. Tsai, *The Design of List Processing Unit (LPU) for the MARS system*. M.S. thesis, Computer Science Division, Department of Electrical Engineering (EECS), National Taiwan University, Taiwan, R.O.C., July 1989.

[24] J. Cho, et al., "The Memory Architecture and the Cache and Memory Management Unit for the Fairchild CLIPPER Processor," *Tech. Rep. UCB/CSD 86/289*, Computer Science Division (EECS), University of California, Berkeley, CA, April 1986.

[25] A. J. Smith, "Cache Memories," *Computing Surveys*, ACM, Vol. 14, No. 3, Sep. 1982, pp. 473-530.

[26] G.-S. Jang, "The Floating Point Unit for MARS: Design and Specification," *NTU-EE-CS memo No. 329-3*, 1988.

[27] K.-C. Chen, H.-C. Lee, F. Lai, and Z.-W. Liao, "Concurrent MARS Lisp: Language feature and Implementation," *NTU-EE-CS memo No. 329-5*, Computer Science Division, Department of Electrical Engineering (EECS), National Taiwan University, R.O.C. Mar. 1989.

[28] D. Lee, et al., "A VLSI Chip Set for a Multiprocessor Workstation, PART I: A RISC Microprocessor with Coprocessor Interface and Support for Symbolic Processing," *Tech. Report No. UCB/CSD 89/500*, Computer Science Division (EECS), University of California, Berkeley, CA, April 1989.

Tag value	Data type
	Immediate:
000XXXB	imm. character
001XXXB	imm. fixed number
010XXXB	imm. short floating point (32 bits)
	Indirect number:
01100XB	big number
01101XB	ratio
01110XB	long floating point (64 bits)
01111XB	complex
	Other:
110000B	
ǀ	defined by compiler
111111B	

Table 2 Data type and corresponding tag value

Instruction	Operands	Action	Cycles(Exception)
List Primitives			
car_b	BRd, Rs1	BRd <- M[Rs1_data &~0F]	1 (E)
car_c	CRd, Rs1	CRd(ipu,fpu,lpu) <- M[Rs1_data&~oF]	1 (E)
cdr	Rd, Rs1	Rd <- M[Rs1_data<31:4> | 08]	1 (E)
cons	Rd, Rs1, Rs2	Rs2 -> M[Rs1_data & ~0F], Rd<-Rs1	1/2 (D)
rplaca	Rs1, Rs2	Rs2 -> M[Rs1_data & ~0F]	1/2 (D)
eplacd	Rs1, Rs2	Rs2 -> M[Rs1_data<31:4> | 08]	1/2 (D)
STACK			
push	Rs2	SP=Sp+8, Rs2 -> M[SP]	1/2 (-)
pop_c	BRd	BRd<-M[SP], SP=SP-8	1 (-)
pop_c	CRd	CRd(lpu,fpu,lpu)<-M[SP], SP=SP-8	1 (-)
Data & Tag Transfer			
mov_b	BRd, Rs2	BRd=Rs2	1 (-)
mov_c	CRd, Rs2	CRd(ipu,fpu)=Rs2_data; CRd(lpu)=Rs2	1 (-)
l_load_b	BRd, Rs1	BRd<-M[Rs1_data]	1 (-)
l_load_c	CRd, Rs1	CRd(ipu,fpu,lpu)<-M[Rs1_data]	1 (-)
l_store	Rs1, Rs2	Rs2->M[Rs1_data]	1/2(-)
f_to_l	Rd, Rs1	Rd_data(lpu)=Rs1(fpu)<single>	1 (-)
l_load_f	Rd, Rs1	Rd(fpu)<double> <-M[Rs1_data]	1 (-)
l_store_f	Rs1, Rs2	Rs2(fpu)<double> ->M[Rs1_data]	1/2(-)
load_tag	Rd, Rs1, imm_tag	Rd=imm_tag | Rs1_data	1 (-)
mov_tag	Rd, Rs1, Rs2	Rd = Rs2_tag | Rs1_data	1 (-)
Compare Branch			
l_dcb_rr	cp, Rs1, Rs2, target	cond=(Rs1cp Rs2)	1 (-)
l_dcb_ri	cp, Rs1, imm_tag, target	cond= (Rs1 cp imm_tag)	1 (-)
l_scb_rr	cp, Rs1, Rs2, target	cond= (Rs1 cp Rs2)	1 (-)
l_scb_ri	cp, Rs1, imm_tag, target	cond= (Rs1 cp imm_tag)	1 (-)
Call/Return			
cal_jmp	addr	cwp=cwp + 1	1 (A)
ret_jmp	addr	cwp = cwp -1	1 (B)
Special			
rd_sp	Rd	Rd_data = SP	1 (-)
wr_sp	Rs1	SP = Rs1_data	1 (-)
rd_lpsw	Rd, Lpsn	Rd = LPSW(Lpsn)	1 (-)
wr_lpsw	Rs1, Lpsn	LPSW(Lpsn) = Rs1	1 (-)
lpu_wake		lpu wake up	1 (-)
lpu_sleep		lpu sleep	1 (-)

Exception definition:

A	000B	window overflow
B	001B	window underflow
C	010B	illegal LPU opcode
D	011B	Rs1_tag != CONS
E	100B	Rs1_tag != CONS or NIL
F	101B	not both tag be FIXNUN
G	110B	not both tag be FIXNUM or CHAR
H	111B	none

Cycles:

1 means one cycle per instruction if no cache miss

1/2 means one cycle per instruction under uniprocessor environment and two cycles per instruction under multiproceessor environment both assmue no cache miss happen

Table 1. The Instruction set for LPU