MARS -- *M*ULTIPROCESSOR *A*RCHITECTURE *R*ECONCILING *S*YMBOLIC WITH NUMERICAL PROCESSING
A CPU *ENSEMBLE* WITH ZERO-DELAY BRANCH/JUMP

Gia-Shuh Jang, Feipei Lai, Hung-Chang Lee, Yeong-Chang Maa, Tai-Ming Parng, Jenn-Yuan Tsai

Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.
Tel: (02)3635251, 3625252 Ext. 241

Abstract

The design of CPU chips (IFU, IPU and LPU) for the *MARS* project is described in this paper. IFU is devised to interleave instruction fetch and execution, and thus to achieve coordinated execution among datapath chips. IPU is the main computing engine for integer operations and operand address calculation. By using dual instruction buffers, a reserved phase for Branch/Jump target fetch and instruction decode peeping, our architecture can support almost-zero-delay branching and super-zero-delay jump. LPU handles Lisp runtime environment, dynamic type checking and fast list access. In this architecture, 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.

I. Introduction

MARS (*M*ultiprocessor *A*rchitecture *R*econciling *S*ymbolic with Numerical Processing) is a multiprocessing system. Each processor board is linked together via an interconnection network (presently, a single system bus). Inside each board, 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), dedicated for floating point and list operations, Instruction Cache, Data Cache, Local Memory and separate Instruction, and Address/Data buses.

IFU, which is built on a single chip, is the buffering, control mechanism between the instruction cache and the datapath chips (IPU, FPU, LPU). It is designed to interleave instruction fetch and execution, and to achieve coordinated execution among IPU, FPU, and LPU. The block diagram is given in Figure 1, 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 retains the integer datapath and some control parts of a common RISC CPU [1-3], performing integer arithmetic, shift, logical operations, and address calculation for data operands of all datapath chips. The functional block is displayed in Figure 2, where we can see a flat 32-word register file, a 2-level internal forwarding latch, and a shifter.

FPU conforms to IEEE Standard 754 [16]. It has separate, pipelined Add/Sub and Mul/Div units to provide spatial and temporal parallelisms, a 32-word 64-bit register file, a hardwired control unit, direct support for hardware format conversion and a synchronous interface protocol to couple tightly with other chips.

LPU 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, address 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. Coherence protocol will be similar to that of the *Dragon* Project [4].

II. Why Separate IFU and IPU ?

At first, we intended to adopt CPU architecture like MIPS-X which incorporates a 2 KB on-chip instruction cache [5], but owing to the unique configuration inside our processor board, an on-chip instruction cache inside CPU will not provide any speed advantage when CPU, FPU, and LPU all wait for instructions coming from the instruction cache and then decode their own instructions. Therefore, we decide to separate the instruction cache from the CPU, 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 an arrangement and the result can still meet our initial requirement.

III. Microarchitecture

Instruction Set

The instruction set (see Figure 3) for IFU and IPU follows that of a typical 801/MIPS-style RISC processor with some minor changes fitting for our own scheme to achieve fast and simple instruction decode [1,3]. By careful encoding, the burden of instruction decoding is well distributed into the datapath chips. There are 4 categories of IFU/IPU instructions, i.e., Compute, Load/Store, Branch/Jump, and Miscellaneous (see Figure 3). They are carefully designed with a view to exploring the controllability and observability of the hardware and thus can be used as a good target for an optimizing compiler. There are no variable-length instructions to prevent instruction addresses from crossing the page boundaries. All Compute operations are register-oriented, that is, only Load/Store instructions are allowed to access the memory. The most attractive merit of our architecture is the minimization of delay slots for Branch/Jump Instructions. We will give a detailed description of these instructions in the following sections.

The instruction set of LPU is carefully designed to speed up the execution of Lisp programs. There are 4 principal types of instructions, List primitives, Stack operation, Data/tag transfer, and Special. Hardware-implemented list primitives are *car*, *cdr*, *cons*, *rplaca* and *rplacd*. The stack operation includes *push* and *pop*, which are mainly used for binding/unbinding special variables and saving/restoring of frame windows. The content of stack pointer register can be read or written by *rd_sp* or *wr_sp* instructions. The data transfer instructions include: *load*, *mov*, *load_f*, *store_f* (the latter two generate address for FPU) and *f_to_l* . The tag value of a register can be loaded with the immediate tag value packed in instruction or moved from another register. 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 current window pointer, saved window number and some system status. The *lpu_wake* and *lpu_sleep* instructions control LPU to be active or not. When LPU is inactive, the *MARS* system is acting as a general purpose computer without hardware support for Lisp.

Pipeline Scheme and Data Flow

We choose an unbalanced pipeline scheme (see Figure 4) to eliminate redundant bubbles occurred during pipeline execution. The IFU pipeline has 2 stages, PD/IAC (Partial Decode/Instruction Address Calculation), and ICA (Instruction Cache Access). IAC and ICA overlap temporally with the first 2 or 3 stages of the datapath pipelines. On the other hand, the IPU pipeline consists of 5 stages, IF (Instruction Fetch), ID/RF (Instruction Decode/Register Fetch), ALU (ALU operation), MA (Memory Access), and WB (register Write Back), but in fact, there are at most 4 instructions (see Figure 5) to be executed in the IPU pipeline.

Due to the high instruction bandwidth provided by the IFU, the whole system uses only one phase to latch the prefetched sequential instruction, leaving a bubble of one phase in the IF pipeline stage. This bubble is used to fetch the target instruction for all branch and jump instructions. Accompanied by a fine tuned microarchitecture, this pipeline scheme does achieve a zero-delay fast compare-and-branch.

We allocate one cycle for memory access, which will be tight for normal cache memory systems. To loosen the timing constraints, a late-miss signal[9] used to check whether the memory access is successful is verified at the beginning of the next cycle. If it is activated, the whole system will be stalled and then be forced to reexecute the second phase of the memory access cycle until the correct data is fetched.

Owing to a simple instruction set and a symmetric 32-word register file, the ID/RF stage needs one phase only. Although the ALU stage occupies 2 phases, yet it is well designed [15] to get the correct results settled almost within one phase. The additional phase will serve as the reserved time slot for memory access if a slow memory system is of interest.

A double internal forwarding mechanism is implemented in hardware. It reduces delay slots of Load instruction from 2 to 1, and makes the pipeline free of interlocks caused by data dependency.

The pipestages of the LPU is given in Figure 6, which is somewhat different from Figure 4. First, ID is no more overlapped with register fetch. Second, instead of taking an ALU operation in the IPU, LPU performs tag comparison and overlaps it with memory access. Despite these different operations, the two pipeline stages get synchronous at the memory access stage.

Most Lisp programs execution spend most of the time doing list access. List structure is usually constructed with two parts, header of list (*car*) and tail of list (*cdr*). And each of the two parts contains a tag field to identify the data type and a data field to tell what the data is or where it is. When a *car* or *cdr* instruction issues, tag field check and data field access carry out simultaneously. By the delayed RF mechanism of LPU, registers can be fetched with shortcut and incur no internal interlock to the ensuing instruction. We call this **Non-delayed list access**, whose details are illustrated in Figure 7.

Special Design Issues for IPU

The general form of microarchitecture is a direct consequence of the chosen instruction set and pipeline scheme. Like other MIPS-like CPUs, the IPU microarchitecture (see Figure 2) bears all the advantages resulting from a simple instruction set, a flat register file, and a streamlined system design.

A flat 32-word, 32-bit register file provides a large storage template for the optimizing compiler to allocate global variables, frequently used temporary variables, and passed parameters. By dint of graph coloring algorithm [17,18] and a rich register file, the optimizing compiler can decrease the variable swapping overhead caused by procedure calls. A Byte Inserter/Extractor is included to support manipulation of characters and strings. Finally, we determined to implement a simple shifter that supports 1 to 3-bit logical left shift and 1-bit logical and arithmetic right shift. The small shift amount is chosen to partially support the address calculations of **array** and **struct** data types.

Microarchitecture for IFU

As shown in Figure 1, the IFU is featured with its dual instruction buffers, the partial decode unit, the return address stack (RAS), the displacement adder, and the remote PC chain.

The dual (sequential and branch target) instruction buffers will have 8 entries each, with a 2-word wide bus directly from the instruction cache. Originally, the instruction buffer was designed as an instruction queue with a single inlet and outlet, but after considering the need of instant access of branch target from cache to the instruction registers of datapath chips, we decide to use parallel load scheme instead of single inlet plus a delicate control logic for initial filling/bypassing of buffers. The outlet is controlled by a counter/multiplexer to sequentially enable the normal outgoing instruction to the pads. Jump instructions, on the other hand, are intercepted within the IFU and will not be released to the datapath chips.

The partial decode unit first distinguishes control transfer instructions from other types of instructions and then extracts some field (*target* field for Jump instructions, e.g., *jumpa, jumpb, cal_jmp*, and *offset* field for Branch instructions) within the instruction to decouple the instruction execution of IFU from all other datapath chips and thus get more parallelism. The return address stack (RAS) contains the return address (NEXTPC) for procedure call instruction pair *cal_jmp* and *ret_jmp*. It is planned to have at least 16 entries to satisfy deep procedure calls of Lisp and to reduce overflow. Overflow/Underflow conditions are resolved through *load_ras* and *store_ras* instructions.

The PC chain is a chain of shift registers holding all the PCs currently in the pipeline. It also includes an incrementer for the most 30 significant bits of current PC to access the next instruction block. The displacement adder generate address for Branch/Jump instructions, either an offset is added to current PC or a 25 bit *target* field is concatenated with the most 7 significant bits of the current PC. Meanwhile there is a base register to facilitate register indirect mode Jump, the *jumpb* instruction.

Registers Structure for LPU

The register structure in LPU plays the role as a runtime environment administrator. In Lisp, arguments, local variables and special variables are accessed with high frequency. These variables are allocated in register file and maintained in a fast scheme described later. There are two kinds of register file in LPU, one is the control register file and the other is the binding register file. The control register file organized as an 8-frame, 136-word overlapping frame window is used to keep the activation records of callers and callees. The 32-word register file of IPU and LPU is mapped to one of the frame window, so users can view the control register as a 32-word register frame window whose data may be integers (in IPU), floating-point numbers (in FPU), or pointers (in LPU). The binding register file is used to keep the binding value of special variables. We use shallow binding scheme to bind and restore the special variables.

In the *MARS* system, only LPU has frame-window register structure and there are merely 32 registers in IPU and FPU. The correspondence between LPU control registers and IPU/FPU register file is obtained by a switched partition mapping mechanism. Fig. 8 also shows the mapping of IPU/FPU's 32-word register into LPU's frame window. The *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, the *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.

Besides the above mapping scheme to reduce the overhead of saving and restoring IPU/FPU register data, we save the register file data into LPU's current frame window during the execution of an IPU 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 them back into the corresponding registers. With this mechanism, we do not need to save any IPU registers data into LPU when executing a function call, but 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 current function when the called function returns.

By using above mechanisms, the register data of IPU could be kept in LPU's control register file with little overhead when 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 accessing the complex register file. On the other hand, the IPU, which must spend time executing ALU operations, has a simple 32-word register file and can access the registers faster.

The 32-word binding register file, which has no counterparts in IPU/FPU, is used to store special variables in Lisp. Each special variable corresponds to one register allocated at loading time. We use shallow binding scheme to handle the binding registers. When a special variable is bound to a new value, the old value in the corresponding register has to be pushed into the binding stack, but when this special variable is unbound, the old value is popped from the binding stack and restored to the corresponding register. An example of binding and unbinding of special variables is shown in Fig. 9. The binding registers can only be used by LPU's instructions. If they are needed to execute 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.

IV. Control Transfer and Exception Handling

Control Transfer

Branches have a considerable effect on the performance of our deeply-pipelined architecture because they interrupt the flow of the pipeline. After studying various branching schemes of pipelined processors [6-8, 12], we decided to combine dual instruction fetch paths with fast/delayed/squashable compare-and-branch to obtain an almost-zero-delay branch. By employing dual instruction buffers in IFU and dual instruction registers in each datapath chip along with a reserved phase within the IF stage, both the sequential and branch target instructions are ready for selection by the branch result at the end of the IF stage. If the *compare* is a **fast** *compare* (*fcb*), i.e., test if one operand is equal to, less than or greater than zero, or test if two operands are equal or unequal, we can resolve the *compare* at the early start of the ALU stage of the current instruction, that is, settle the branch before the end of the IF stage of next instruction. Therefore, we can obtain a zero-delay branch.

However, in some cases, the compares are not to be or can not be converted to be **fast**, thus a **full** compare must be addressed by **delayed** or **squashable** compare-and-branch (*dcb* and *scb*). A delayed compare-and-branch is used when there are available surely-executed instructions to fill the delay slot, whereas a squashable one is used when only probably-executed instructions coming from the beginning of the branch target are available. If the branch is not taken as expected, the instruction filled in the delay slot will be squashed (i.e., turned into nop). Statistics have shown that, through compiler techniques, about 80% of branches can be changed into **fast** compare [2,9]. Accordingly, we believe *fcb* will enjoy the lion share of Branch instructions.

Since Jump instructions only involve target address calculation within IFU and need no data operation of the datapath chips, they are intercepted by the IFU and will not be released to the datapath. With dual partial decode masks, the partial decode unit of IFU can peep out the existence of an ensuing Jump instruction, calculating the address, and accessing the instruction cache ahead of time, so IFU can send out the jump target in the original instruction slot for Jump. From the viewpoints of datapath chips, the IFU absorbs the Jump instruction and directly issues the target instruction. We would like to call it a **super-zero-delay** Jump because the delay is *de facto* minus one.

Exception Handling

Exception is another source of pipeline breakage, which needs very careful treatment to reduce its harm to performance. Though there have been many schemes proposed to implement precise exception (interrupt) [9-11], the hardware complexity and the precision of exception recovery still pose to be a dilemma. We often have to trade off one for the other. In our system, we determine to favor the former, that is, to reduce the hardware complexity reasonably to quicken our cycle time and hence increase our performance. Once an exception happens, either in IFU or any of the datapath chips, the exception-stuck chip will serve as a ***floating master***, sending encoded exception condition to the IFU, and notifying all other chips to purge all the instructions not yet finished in their pipelines. The IFU also takes this information to decide the starting address of the exception handling routine. Worthy of note, the mechanism for pipeline purging can be shared with that of branch delay (instruction) squashing, thus getting a uniform design for hardware.

V. Status and Performance Expectation

Up to now, with the help of the GDT M language, intraboard functional simulation has been finished and logic-level simulations for the chips are under way. Performance evaluations by means of queueing models and instruction-level trace-driven simulation for a uniprocessor, system bus, and overall system are also being undertaken.

Five benchmarks - 20 queen, ackermann(3,8), shell sort, and Hanoi tower (n = 18 and n = 24) - have been executed on SPARC, mips R2000 and our *MARS* instruction-level simulator. The results are shown in table 1. Column 1 gives the results for *MARS*, excluding the effects of cache miss, while column 2, 3, 4, 5 give those for SPARC, mips R2000, SPARC (optimized) and mips R2000 (optimized). Column 6 through 9 give the ratios of execution time for SPARC and R2000 to that of our system. From these data, we can see that *MARS* executes C about 3.24 times as fast as SPARC (with optimization) and about 1.66 times as fast as mips R2000 (with optimization).

According to Katevenis' thesis [2], branch/jump occupies at least 30% of the dynamically executed instructions which often incur 1 to 2 *nops* or delay slots to be filled. Our pipeline arrangement can drastically reduce slots due to control transfer by *fcb* (compared with mips R2000 [13,14], when there is no program rewriting, the figures are 1/6 for non-numeric and 3/8 for numeric programs, respectively); leaving the compiler more chances to fill the delayed load slots, thus accomplishing our goal of single-cycle instruction execution. According to the above estimation, our architectural performance gain over mips R2000 will be 12%. In addition, our clock rate is 50 ns, which is 7/6 faster than that of R2000. Thus *MARS* should outperform R2000 by about 30%, which is quite close to our preliminary results (if the effect of cache miss or other kinds of overhead is taken into account).

Three Gabriel benchmarks - a set of programs which test the speed of various aspects of Lisp systems- have been carried out to evaluate the power of *MARS* against other architectures, shown in table 2. The first column shows the results for *MARS*, excluding the consideration of cache misses. Column 2, 3, 4, and 5 give the results for three other architectures; the results for VAX-11/780 are from Gabriel's book [19], SPUR's results are from Patterson's paper [20], MIPS-X's results are from Pater's paper [21], with and without optimization. *MARS* executes Lisp programs about 27.8 times as fast as the VAX-11/780, almost 4.1 times as fast as SPUR, and about 2.3 times as fast as MIPS-X. However, the performance advantage varies significantly among all these benchmarks.

MARS Lisp performance edge over MIPS-X may be attributed to its direct hardware support for tag handling, type checking on lists, binding registers and uses frame windows to reduce the cost of register saving and restoring. The frame windows do not really pay off for the Gabriel benchmarks and their average effect is neutral. This is because these benchmarks use only a few arguments and local variables, and have a very deep call-depth.

VI. Conclusion

A design for CPU (IFU, IPU and LPU) chips of our *MARS* system 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. Preliminary performance evaluations have shown that our *MARS* achieves a remarkable edge over SPARC and mips R2000. Also, with deliberate register structure for environment updating and variable binding, direct hardware support for list primitives, distributing register fetch critical path to LPU and other chips and non-delayed *car* and *cdr* instructions, our system will outperform VAX, SPUR and MIPS-X on Lisp execution to a remarkable degree.

VII. Acknowledgements

The *MARS* project is partly supported by the National Science Council under contract NSC78-0404-E002-45. We would like to thank Jiun-Lu Lee for his programming effort on our instruction-level simulator, and Jiun-Loong Tsai of ERSO, ITRI for his help on running benchmarks on mips R2000.

References

1. G. Radin, "The 801 Minicomputer," *Proc. SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, Palo Alto*, Mar. 1982, pp. 39-47.
2. M. Katevenis, "Reduced Instruction Set Computer Architectures for VLSI," Ph.D. thesis, Computer Science Division(EECS) UCB/CSD, University of California, Berkeley, Oct. 1983.
3. S. Przybylski *et al* "Organization and VLSI Implementation of MIPS," *Journal of VLSI and Computer Systems*, Vol. 1, No. 2, Dec. 1984, pp. 170-208.
4. J. Archibald and J.-L. Baer, "Cache Coherence Protocol: Evaluation Using a Multiprocessor Simulation Model," ACM Trans. Computer Systems, Vol. 4, No. 4, Nov. 1986, pp. 273-298.
5. 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.
6. S. McFarling and J. Hennessy, "Reducing the Cost of Branches," *Proc. 13th Symposium on Computer Architecture*, Jun. 1986, pp. 396-403.
7. J. E. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th Symposium on Computer Architecture*, May 1981, pp. 135-148.
8. J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, Jan. 1984, pp. 6-22.
9. P. Chow and M. Horowitz, "Architectural Tradeoffs in the Design of MIPS-X," *Proc. 13th Symposium on Computer Architecture*, Jun. 1986, pp. 300-308.
10. J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. 12th Symposium on Computer Architecture*, Jun. 1985, pp. 36-44.
11. 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.
12. D. J. Lilja, "Reduced the Branch Penalty in Pipelined Processors," *Computer*, Jul. 1988, pp. 47-55.
13. J. Moussouris *et al*, "A CMOS RISC Processor with Integrated System Functions," *The Proceedings of COMPCON Spring 86*, IEEE, Mar. 4-6, 1986, pp. 126-137.
14. Craig Hansen *et al*, "A RISC Microprocessor with Integral MMU and Cache Interface," *Proc. ICCD*, IEEE, Oct. 6-9, 1986.
15. Tackdon Han, David A. Carlson, "Fast Area Efficient Adder," *Proc. 8th Symposium on Computer Arithmetic*, May 1987.
16. IEEE Computer Society Microprocessor Standards Committee Task P754, "A Proposed Standard for Binary Floating Point Arithmetic, draft 10.0," Jan. 1983.
17. G. J. Chatin *et al.* "Register Allocation via Coloring," *Proc. SIGPLAN Symposium on Compiler Construction*, Jun. 1982.
18. F. Chow and John Hennessy, "Register Allocation by Priority-Based Coloring," *Proc. SIGPLAN Symposium on Compiler Construction*, Jun. 1984.
19. Richard P. Gabriel, "Performance and Evaluation of Lisp System," *Reading*, MIT Press, Cambridge, Mass., 1985.
20. Dave Patterson, "A Progress Report on SPUR," *ACM Computer Architecture News*, Feb. 1987, pp. 15-21.
21. Peter Steenkiste, and John Hennessy, "Lisp on a Reduced Instruction Set processor: Characterization and Optimization," *Computer*, June 1988, pp. 34-45.

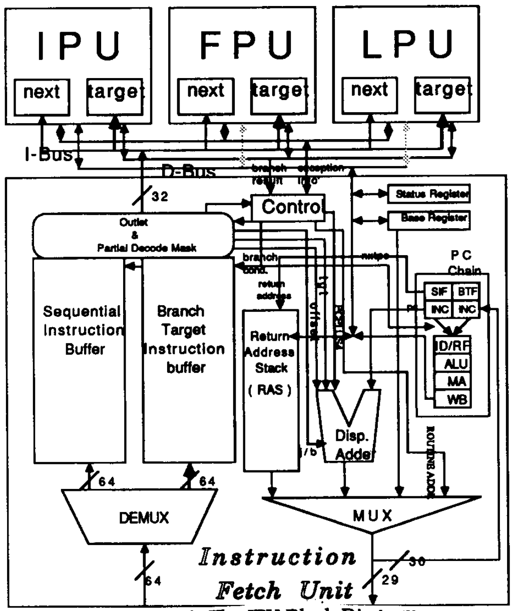


Figure 1. The IFU Block Diagram

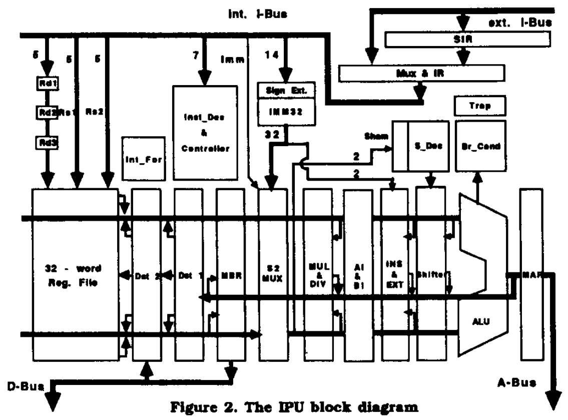


Figure 2. The IPU block diagram

Instruction	Operands	Action	Note	Delay
		COMPUTE		
add	Rd,Rs1,S2	Rd <- Rs1 + S2	Integer Addition	-
addu	Rd,Rs1,S2	Rd <- Rs1 + S2	Unsigned Integer Addition	-
sub	Rd,Rs1,S2	Rd <- Rs1 - S2	Integer Subtraction	-
subu	Rd,Rs1,S2	Rd <- Rs1 - S2	Unsigned Integer Subtraction	-
and	Rd,Rs1,S2	Rd <- Rs1 && S2	Logical AND	-
or	Rd,Rs1,S2	Rd <- Rs1 || S2	Logical OR	-
xor	Rd,Rs1,S2	Rd <- Rs1 xor S2	Logical Exclusive OR	-
sll	Rd,Rs1,S2	Rd <- Rs1 << S2<01:00>	Shift Left Logical	-
srl	Rd,Rs1,S2	Rd <- Rs1 >> S2<00>	Shift Right Logical	-
sra	Rd,Rs1,S2	Rd <- Rs1 >> S2<00>	Shift Right Arithmetic	-
extract	Rd,Rs1,imm	Rd<31:08> <- 0 Rd<07:00> <- byte imm<01:00> of Rs1	Byte Extraction	-
insert	Rd,Rs1,Rs2,imm	Rd<others> <- Rd<others> byte imm<01:00> of Rd <- Rs1<07:00>	Byte Insertion	-

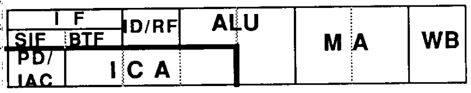
mult	Rs1,S2	LO <- bit <31:00> of Rs1 * S2
		HI <- bit <63:32> of Rs1 * S2
			Intiger Multiplication	-

div	Rs1,S2	LO <- Rs1 div S2
		HI <- Rs1 mod S2
			Intiger Division	-

		BRANCH / JUMP		
fcb	cond,Rs1,S2,offset	if (Rs1 cond S2) then PC = PC + offset	Fast Compare&Branch with zero delay slot	yes
dcb	cond,Rs1,S2,offset	if (Rs1 cond S2) then PC = PC + offset	Delayed Compare&Branch with one delay slot	yes
scb	cond,Rs1,S2,offset	if (Rs1 cond S2) then PC = PC + offset else Squash the next instruction in the delay slot	Squashable Compare&Branch	yes
jumpb	base	PC = base register in IFU	Jump register indirect	-
jumpa	target	PC = target (within a segment)	Jump absolute	-
				-
cal_jmp	target	ras[++top] = PC + 1 PC = target	Push next PC to ras then Jump to target	-
ret_jmp		PC = ras[top--]	Pop top of ras to PC	-
		MISCELLANEOUS		
ipu_to_ifu	Rd,Rs1	IFU Rd <- IPU Rs1	IFU special register read	yes
ifu_to_ipu	Rd,Rs1	IFU Rd <- IPU Rs1	IFU special register write	yes
load_ras	addr	ras[++bottom] = M[addr]	Load ras upon stack underflow	yes
store_ras	addr	M[addr] = ras[bottom--]	Store ras upon stack overflow	yes
syscall		ras[++top] = PC +1 PC = exception handler modify STATUS register	System Call Trap	
rfe		restore STATUS register PC = ras[top--]	Return from Exception	
mfhi	Rd	Rd <- HI	Move contents of special register HI to Rd	
mflo	Rd	Rd <- LO	Move contents of special register LO to Rd	

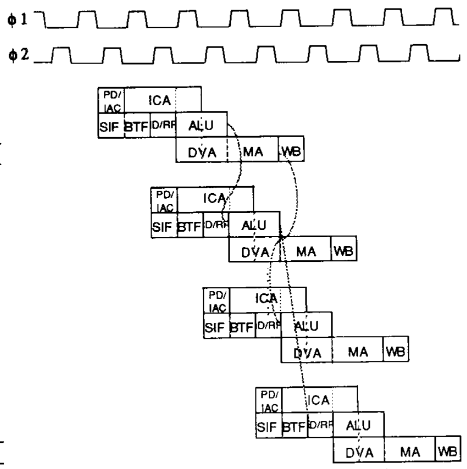
Instruction	Operands	Cache Op	Action	Note	Delay
			LOAD		
load	Rd,Rs1,S2	R[RO][RA]	Rd <- M[(Rs1+S2)&'03]	Load from data cache into Rd	yes
test_and_set	Rd,Rs1,S2	TS	Rd <- M[(Rs1+S2)&'03] M [(Rs1+S2)&'03] <00> <- 1	Test_and_set operation	yes
load_cs	Rd,Rs1,S2	any	Rd <- external cache defined by (Rs1 + S2)	Load cache status into Rd	yes
			STORE		
store	Rs2,Rs1,imm	W	Rs2 -> M[(Rs1+imm) &'03]	Store from Rs2 into data cache	-
store_cs	Rs2,Rs1,imm	any	Rs2 -> external cache state defined by (Rs1 + imm)	Store from Rs2 into data cache cache status register	-
to_fpu	Rd,Rs1,0	NA	Rs2_data -> fpu Rd <63..32>	Send data from IPU to FPU	yes
from_fpu	Rd,Rs1,0	NA	Rd <- FPU Rs1<63..32>	Send data from FPU to IPU	yes

Figure 3. Instruction Set for IFU and IPU



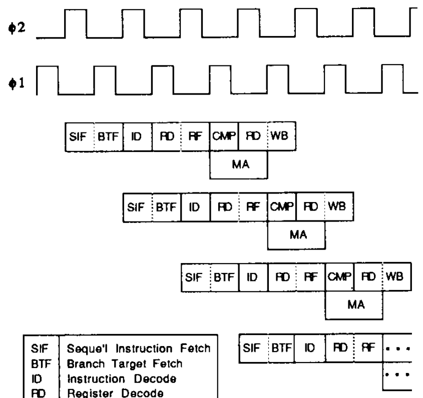
PD	Partial Decode in the IFU
IAC	Instruction Address Calculation in the IFU
ICA	Instruction Cache Access in the IFU
SIF	Sequential Instruction fetch from IFU
BTF	Branch Target instruction fetch from IFU
ID	Instruction Decode
RF	Register Fetch
ALU	ALU operation
MA	Memory Access
WB	register Write Back

Figure 4. The IFU/IPU pipeline



PD	Partial Decode in the IFU
IAC	Instruction Address Calculation in the IFU
ICA	Instruction Cache Access in the IFU
SIF	Sequential Instruction fetch from IFU
BTF	Branch Target instruction fetch from IFU
ID	Instruction Decode
RF	Register Fetch
ALU	ALU operation
DVA	Data Virtual Address calculation by ALU
MA	Memory Access
WB	register Write Back

Figure 5. Instructions executed in the IFU/IPU pipeline



SIF	Seque'l Instruction Fetch
BTF	Branch Target Fetch
ID	Instruction Decode
RD	Register Decode
RF	Register Fetch
CMP	Tag Compare
MA	Memory Access
WB	Register Write Back

Figure 6. LPU pipeline stages

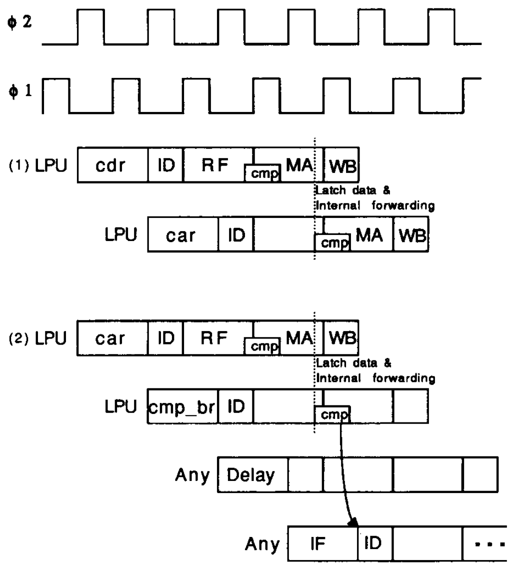
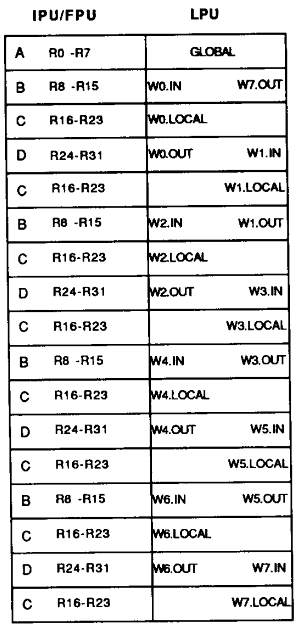


Figure 7. Examples of non-delay load



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

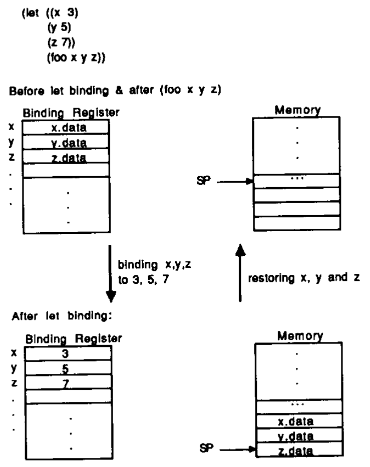


Figure 9. Binding and Unbinding of Special Variables

	Time in seconds					Ratios			
	MARS	SPARC	R2000	SPARC●	R2000●	SPARC/ MARS	R2000/ MARS	SPARC●/ MARS	R2000●/ MARS
queen	13.7	70.9	49.26	49.42	22.81	5.18	3.60	3.61	1.66
acker	1.36	21.19	3.42	16.20	2.42	15.58	2.51	11.91	1.78
shell	.0076	0.05	0.02	0.02	0.01	6.44	2.58	2.58	1.29
H18	0.1835	0.75	0.35	0.32	0.33	4.09	1.91	1.74	1.80
H24	11.7	48.98	22.01	21.62	21.16	4.19	1.88	1.85	1.81
Mean						6.17	2.42	3.24	1.66

Table 1. Execution time in seconds for five C benchmarks

Note: 1. ● means execution with optimization.
2. Mean is the geometric mean.

	Time in milliseconds					Ratios			
	MARS	VAX	SPUR	MIPS-X	MIPS-X●	VAX/ MARS	SPUR/ MARS	MIPS-X/ MARS	MIPS-X●/ MARS
tak	37	830	120	72	72	22.4	3.2	1.9	1.9
stak	70	7100	1060	602	592	101.4	15.1	8.6	8.4
takl	325	5270	825	482	448	16.8	2.5	1.5	1.4
div-iter	55	3800	—	307	157	69.1	—	5.6	2.9
div-rev	340	3750	2910	284	196	11.0	8.6	0.8	0.6
deriv	110	8580	990	604	381	78.0	9.0	5.5	3.5
Geometric mean						35.4	6.2	2.9	2.2

Table 2. Execution times in milliseconds for the Grabriel benchmarks

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