COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY STANFORD CA 94305-4055

MIPS: A VLSI Processor Architecture

John Hennessy, Norman Jouppi,
Forest Baskett, and John Gill'

Technical Report No. 223

November 1981.

The MIPS project has been supported by the Defense Advanced Research Projects Agency under contract #MDA903-79-C-0680.

MIPS: A VLSI Processor Architecture

John Hennessy, Norman Jouppi,
Forest Baskett, and John Gill

Technical Report No. 223

November 1981

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

MIPS is a new single chip VLSI processor architecture. it attempts to achieve high performance with the use of a simplified instruction set, similar to those found in microengines. The processor is a fast pipelined engine without pipeline interlocks. Software solutions to several traditional hardware problems, such as providing pipeline interlocks, are used.

Key Words and Phrases: Instruction set design, VLSI, computer architecture, pipelining,

An earlier version of this report appears in the Proceedings of the CMU Conference on VLSI Systems and Computations, 1981

1 Introduction

MIPS (Microprocessor without Interlocked Pipe Stages) is a general purpose processor architecture designed to be implemented on a single VLSI chip. The main goal of the design is high performance in the execution of compiled code. The architecture is experimental since it is a radical break with the trend of modem computer architectures. The basic philosophy of MIPS is to present an instruction set that is a compiler-driven encoding of the microengine. Thus, little or no decoding is needed and the instructions correspond closely to microcode instructions. The processor is pipelined but provides no pipeline interlock hardware; this function must be provided by software.

The MIPS architecture presents the user with a fast machine with a simple instruction set. This approach is currently in use within the RISC project at Berkeley [4]; it is directly opposed to the approach taken by architectures such as the VAX. However, there are significant differences between the RISC approach and the approach used in MIPS:

1. The RISC architecture is simple both in the instruction set and the hardware needed to implement that instruction set Although the MIPS instruction set has a simple hardware implementation (i.e.. it requires a minimal amount of hardware control), the user level instruction set is not as straightforward, and the simplicity of the user level instruction set is secondary.

2. The thrust of the RISC design is towards efficient implementation of a straightforward instruction set In the MIPS design, high performance from the hardware engine is a primary goal, and the microengine is presented to the end user with a minimal amount of interpretation. This makes most of the microengine's parallelism available at the instruction set level.

3. The RISC project relies on a straightforward instruction set and straightforward compiler technology. MIPS will require more sophisticated compiler technology and will gain significant performance benefits from that technology.

MIPS is designed for high performance. To allow the user to get maximum performance, the complexity of individual instructions is minimized. This allows the execution of these instructions at significantly higher speeds. To take advantage of simpler hardware and an instruction set that easily maps to the microinstruction set, additional compiler-type translation is needed This compiler technology makes a compact and time-efficient mapping between higher level constructs and the simplified instruction set. The shifting of the complexity from the hardware to the software has several major advantages:

- The complexity is paid for only once during compilation. When a user runs his program on a complex architecture, he pays the cost of the architectural overhead each time he runs his program.

- It allows the concentration of energies on the software, rather than constructing a complex hardware engine, which is hard to design, debug, and efficiently utilize. Software is not necessarily easier to construct, but the VLSI environment makes hardware simplicity important.

The design of a high performance VLSI processor is dramatically affected by the technology. Among the most important design considerations are: the effect of pin limitations, available silicon area, and size/speed tradeoffs. Pin limitations force the careful design of a scheme for multiplexing the available pins, especially when data and instruction fetches are overlapped. Area limitations and the speed of off-chip intercommunication require choices between on- and off-chip functions as well as limiting the complete on-chip design. With current state-of-the-art technology either some vital component of the processor (such as memory management) must be off-chip, or the size of the chip will make both its performance and yields unacceptably low. Choosing what functions are migrated off-chip must be done carefully so that the performance effects of the partitioning are minimized In some cases, through careful design, the effects may be eliminated at some extra cost for high speed off-chip functions.

Speed/complexity/area tradeoffs are perhaps the most important and difficult phenomena to deal with. Additional on-chip functionality requires more area, which also slows down the performance of every other function. This occurs for two equally important reasons: additional control and decoding logic increases the length of the critical path (by increasing the number of active elements in the path) and each additional function increases the length of internal wire delays. In the processor's data path these wire delays can be substantial, since thy accumulate both from bus delays, which occur when the data path is lengthed, and control delays, which occur when the decoding and control is expanded or when the data path is widenced. In the MIPS architecture we have attempted to control these delays; however, they remain a dominant factor in determining the speed of the processor.

2 The microarchitecture

2.1 Design philosophy

The fastest execution of a task on a microengine would be one in which all resources of the microengine were used at a 100% duty cycle performing a n&redundant and algorithmically efficient encoding of the task. The MIPS microengine attempts to achieve this goal. The user instruction set is an encoding of the microengine that makes a maximum amount of the microengine available. This goal motivated many of the design decisions found in the architecture.

MIPS is a load/store architecture, i.e. data may be operated on only when it is in a register and only load/store instructions access memory. If data operands are used repeatedly in a basic block of code, having them in registers will prevent redundant load/stores and redundant addressing calculations; this allows higher throughput since more operations directly related to the.computation can be performed. The only addressing modes supported are immediate, based with offset, indexed, or base shifted. These addressing modes may

require fields from the instruction itself, general registers, and one ALU or shifter operation. Another ALU operation available in the fourth stage of every instruction can be used for a (possibly unrelated) computation. Another major benefit derived from the load/store architecture is simplicity of the pipeline structure. The simplified structure has a fixed number of pipestages, each of the same length. Because, the stages can oc used in varying (but related) ways, the result is that pipline utilization improves. Also, the absence of synchronization between stages of the pipe, increases the performance of the pipeline and simplifies the hardware. The simplified pipeline eases the handling of both interrupts and page faults (see Section 4.2).

Although MIPS is a pipelined processor it does not have hardware pipeline interlocks. The five stage pipeline contains three active instructions at any time; either the odd or even pipestages are active. The major pipestages and their tasks are shown in Table 1.

Table 1: Major pipestages and their functions

Staae	Mnemon i c	Task
Instruction Fetch	IF	Send out the PC, increment it
Instruction Decode	ID	Decode instruction
Operand Decode	OD	Compute effective add ress and send to memory if load or store, use ALU
Operand Store/ Execution	OS/ EX	Send out operand it store/ Execution cycle. use ALU
Operand Fetch	OF	Receive operand if load

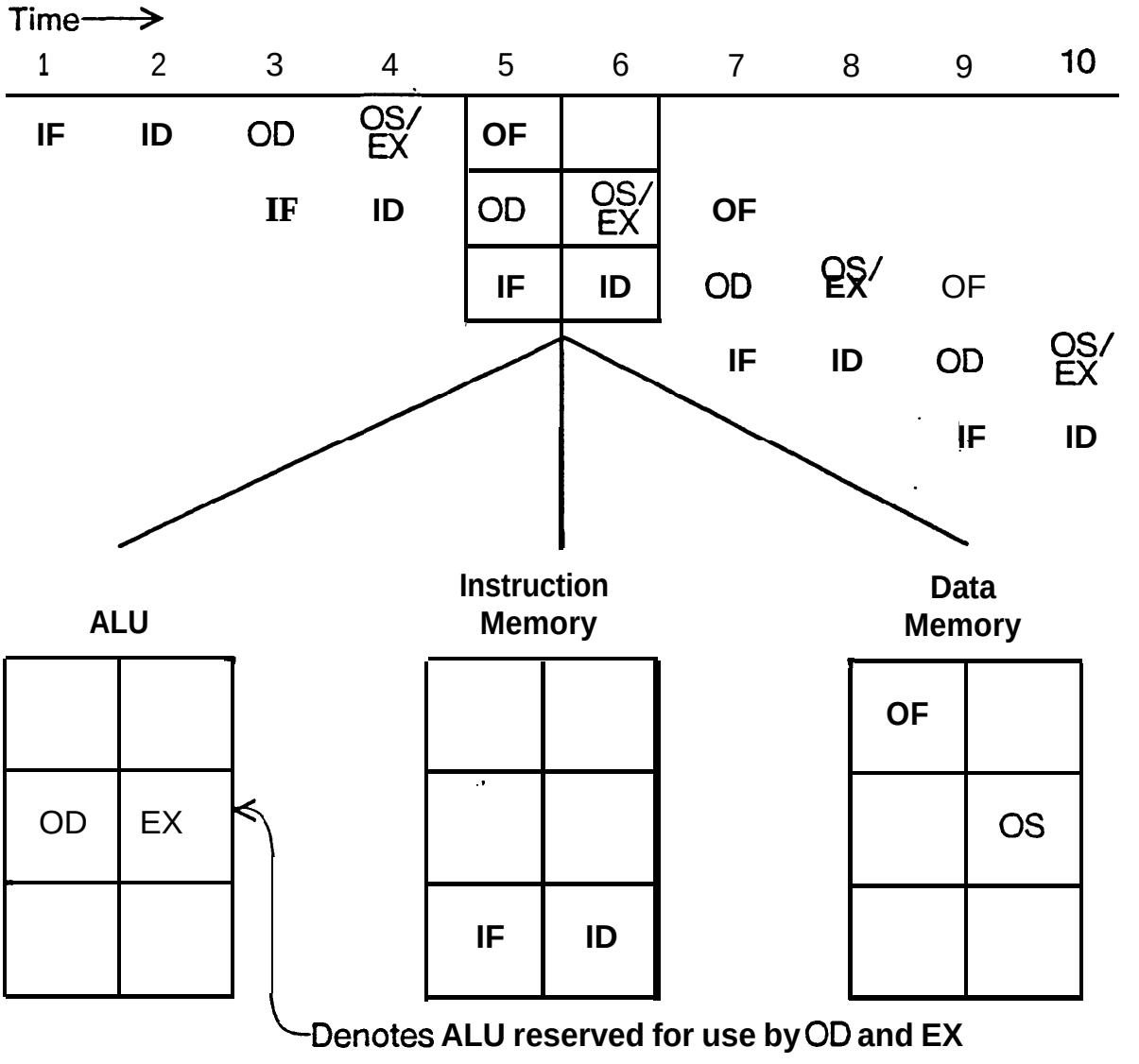
Interlocks that are required because of dependencies brought out by pipelining are *not* provided by the hard ware. Instead, these interlocks must be statically provided where they are needed by a ***pipeline reorganizer.*** This has two benefits:

1. A more regular and faster hardware implementation is possible since it does not have the usual complexity associated with a pipelined machine. Hardware interlocks cause small delays for all instructions, regardless of their relationship on other instructions. Also. interlock hardware tends to be very complex and nonregular [3, 5]. The lack of such hardware is especially important for VLSI implementations, where regularity and simplicity is important

2. Rearranging operations at compile time is better than delaying them at run time. With a good pipeline reorganizer, most cases where interlocks are avoidable should be found and taken advantage of. This results in performance better than a. comparable machine with hardware interlocks, since usage of resources will not be delayed. In cases where this is not detected or is not possible, no-ops must be inserted into the code. This does not slow down execution compared to a similar machine with hardware interlocks, but does increase code size. The shifting of work to a reorganizer would be a disadvantage if it took excessive amounts of computation. It appears this is not a problem for our first reorganizer.

In the MIPS pipeline resource usage is permanently allocated to various pipe stages. Rather than having pipeline stages compete for the use of resources through queues or priority schemes, the machine's resources are dedicated to specific stages so that they are 100% utilized (see Figure1).

Figure 1: Resource Allocation by Pipestage



To achieve 100% utilization primitive operations in the microenginc (e.g., load/store, ALU operations) must be completely packed into macroinstructions. This is not possible for three reasons:

1. Dependencies can prevent full usage of the microcngine, for example when a sequence of register loads must be done before an ALU operation or when no-ops must be inserted.

2. An encoding that prescrved all the parallclism (i.e., the microcontrol word itself) would be too large. This is not serious problcm since many of the possible microinstructions are not uscful.

3. The encoding of the microenginc presented in the instruction set sacrifices some functional specification for immediate data. In the worst case, space in the instruction word used for loading large immediate values takes up the space normally used for a base register, displacement, and ALU operation specification. In this case the memory interface and ALU can not be used during the pipe stage for which they arc dedicated.

Nevertheless, first results on microengine utilization are encouraging. Many instructions fully utilize the major resources of the machine. Other instructions, such as load immediate which use few of the resources of the machine, would mandate greatly increased control complexity if overlap with surrounding instructions was attempted in an irregular fashion.

MIPS has one instruction size, and all instructions execute in the same amount of time (one data memory cycle). This choice simplifies the construction of code generators for the architecture (by eliminating many nonobvious code sequences for different functions) and makes the construction of a synchronous regular pipeline much easier. Additionally, thc fact that each macroinstruction is a single microinstruction of fixed length and execution time means that a minimum amount of internal state is needed in the processor. The absence of this internal state leads to a faster processor and minimizes the difficulty of supporting interrupts and page faults.

2.2 Resources of the microengine

The major functional components of the microengine include:

- ALU resources: A high speed, 32-bit carry lookahead ALU with hardware support for multiply and divide; and a barrel shifter with byte insert and extract capabilities. Only one of the ALU resources is usable at a time. Thus within the class of ALU resources, functional units can not be fully used even when the class itself is used 100%.

- Internal bus resources: Two 32-bit bidirectional busses, each connecting almost all functional components.

- On chip storage: Sixteen 32-bit general purpose registers.

- Memory resources: Two memory interfaces, one for instructions and one for data. Each of the parts of the memory resource can be 100% utilized (subject to packing and instruction space usage) because either one store or load form data memory and one instruction fetch can occur simultaneously.

- A multistage PC unit: An incrementable current PC with storage of one branch target as well as six previous PC values. These are required by the pipelining of instructions and interupt and exception handling.

3 The instruction set

All MIPS instructions are 32-bits. The user instruction set is a compiler-based encoding (i.e. code generation efficiency is used to choose alternative instructions) of the micromachine. Multiple simple (and possibly unrelated) instruction pieces are packed together into an instruction word The basic instruction pieces are:

1. ALU pieces - these instructions are all register/register (2 and 3 operand formats). They all use less than 1/2 of an instruction word Included in this category are byte insert/extract, two bit Booths multiply step, and one bit nonrestoring divide step.

2. Load/store pieces - these instructions load and store memory operands. They use between 16 and 32 bits of an instruction word When a load instruction is less than 32 bits, it may be packaged with an ALU instruction, which is executed during the Execution stage of the pipeline.

3. Control flow pieces - these include straight jumps and compare instructions with relative jumps. MIPS does not have condition codes, but includes a rich collection of set conditionally and compare and jump instructions. The set conditional instructions provide a powerful implementation for conditional expressions. They set a register to all l's or 0's based on one of 16 possible comparisons done during the operand decode stage. During the Execution stage an ALU operation is available for logical operations with other booleans. The compare and jump instructions are direct encodings of the micromachine: the effective operand decode stage computes the address of the branch target and the Execution cycle does the comparison. All branch instructions have a delay in their effect of one instruction; i.e., the next sequential instruction is always executed

4. Other instructions - include procedure and interrupt linkage. The procedure linkage instructions. also fit easily into the micromachine format of effective address calculation and register-register computation instructions.

MIPS is a word-addressed machine. This provides several major performance advantages over a byte addressed architecture. First, the use of word addressing simplifies the memory interface since extraction and insertion hardware is not needed This is particularly important, since instruction and data fetch/store are in a critical path. Second, when byte data (characters) can be handled in word blocks, the computation is much more efficient, Last, the effectiveness of short offsets from base register is multiplied by a factor of four.

MIPS does not 'directly support floating point arithmetic. For applications where such computations are infrequent, floating point operations implemented with integer operations and field insertion/extraction sequences should be sufficient. For more intensive- applications a numeric co-processor similar to the Intel 8087 would be appropriate.

4 Systems issues

The key systems issues are the memory system, and internal traps and external interrupt support

4.1 The memory system

The use of memory mapping hardware (off chip in the current design) is needed to support virtual memory. Modem microprocessors (Motorola 68000) are already faced with the problem that the sum of the memory access time and the memory mapping time is too long to allow the processor to run at full speed. This problem is compounded in MIPS; the effect of pipelining is that a single instruction/data memory must provide access at approximately twice the normal rate (for 64k RAMS).

The solution we have chosen to this problem is to separate the data and instruction memory systems. Separation of program and data is a regular practice on many machines; in the MIPS system it allows us to significantly increase performance. Another benefit of the separation is that it allows the use of a cache only for instructions. Because the instruction memory can be treated as read-only memory (except when a program is being loaded), the cache control is simple. The use of an instruction cache allows increased performance by providing more time during the critical instruction decode pipe stage..

4.2 Faults and interrupts

. The MIPS architecture will support page faults, externally generated interrupts+ and internally generated traps (arithmetic overflow). The necessary hardware to handle such things in a pipclined architecture usually large and complex [3, 5]. Furthermore, this is an area where the lack of sufficient hardware support makes the construction of systems software impossible. However, because the MIPS instruction set is not interpreted by a microengine (with its own state), hardware support for page faults and interrupts is significantly simplified

To handle interrupts and page faults correctly, two important properties are required. First, the architecture must ensure correct shutdown of the pipe, without executing any faulted instructions (such as the instruction which page faulted). Most present microprocessors can not perform this function correctly (c.g. Motorola 68000, Zilog 28000, and the Intel 8086). Second, the processor must be able to correctly restore the pipe and continue execution as if the interrupt or fault had not occurred.

These problems are significantly cased in MIPS because of the location of writes within the pipe stages. In MIPS all instructions which can page fault do not write to any storage, either registers or memory, before the fault is detected. The occurrence of a page fault need only turn off writes generated by this and any instructions following it which are already in the pipe. These following instructions also have not written to

any storage before the fault occurs. The instruction preceding the faulting instruction is guaranteed to be executable or to fault in a restartable manner even after the instruction following it faults. The pipeline is drained and control is transferred to a general purpose exception handler. To correctly restart execution three instructions need to be reexecuted. A multistage PC tracks these instructions and aids in correctly executing them.

5 Software issues

The two major components of the MIPS software system are compilers and ***pipeline reorganizers.*** The input to a pipeline reorganizer is a sequence of simple MIPS instructions or instruction pieces generated without taking the pipeline interlocks and instruction packing features into account. This relieves the compiler from the task of dealing with the restrictions that are imposed by the pipeline constraints on legal code sequences. The reorganizer reorders the instructions to make maximum use of the pipeline while enforcing the pipeline interlocks in the code. It also packs the instruction pieces to maximize use of each instruction word Lastly, the pipeline reorganizer handles the effect of branch delays.

Since all instructions execute in the same time, and most instructions generated by a code generator will not be full MIPS instruction set, the instruction packing can be very effective in reducing execution time. In fully packed instructions, e.g. a load combined with an ALU instruction, all the major processor resources (both memory interfaces, the alu, busses and control logic) are used 100% of the time.

The example in Figure 2 illustrates the techniques: where possible, short instructions are moved together into one word. As this is a very short segment, not too many compactions are possible. Once a basic block has been treated for compaction, the effects of the delayed branch arc processed. In this case it is possible to remove the no-ops, required because of pipeline dependencies and branch delays, completely.

Note that the code with no-ops was also of reasonable quality: the loading of the array base addresses is hoisted up, and the store of S is moved out of the loop. (Initialization of S is done outside the segment considered.) The no-op following "ld (r5,r1), r7" is necessary to take care of the missing pipeline interlock.

The optimal packing of instructions is obviously a hard problem (at least NP-complete); however, we are investigating heuristics that we believe will have acceptable running times, yet will produce nearly optimal code in most cases.

Figure 2: Source code, original machine code, and reorganized machine code

Source code	Correct Code with No-Ops	Reorganized Code	ALU use 00	ALU use EX	Data Memory use
(* A,B,C: global	ld #A. r3				
N,R,S: local *)	ld #B, r4				
	ld #C, r5				
For i:= 0 To N Do	mv #0, rl	ld N(sp),r2;mv #0,r1	x	x	x
	ld N(sp),r2	ld #C, r5			x
	bgt rl. r2. L30	bgt rl, r2, L30	x	x	
	no-op	ld #B, r4			x
	no-op	ld #A, r3			x
Begin					
A[i]:=B[i]+C[i];	L20:ld (r4,r1),r6	L100:ld (r4,r1),r6	x		x
	ld (r5,r1), r7	ld (r5,r1),r7;add r6.r8	x	x	x
	no-op				
	add r7, r6, r9	add r7,r6,r9;add r7,r10	x	x	
	st r9, (r3,r1)	st r9,(r3,r1);add #1,r1	x	x	x
R:= R + B[i];	add r6, r8				
S:= s + C[i];	add r7, r10				
	add #1, rl				
	ble rl. r2, L20	ble rl, r2, L100	x	x	
	no-op	st r8, R(sp)	x		x
	no-op	st r10, S(sp)	x		x
End	st r8, R(sp)				
	st r10, S(sp)				
	L30:....	L30:....			
Size	21 Words	12 Words			
Time	120 Units	75 Units			

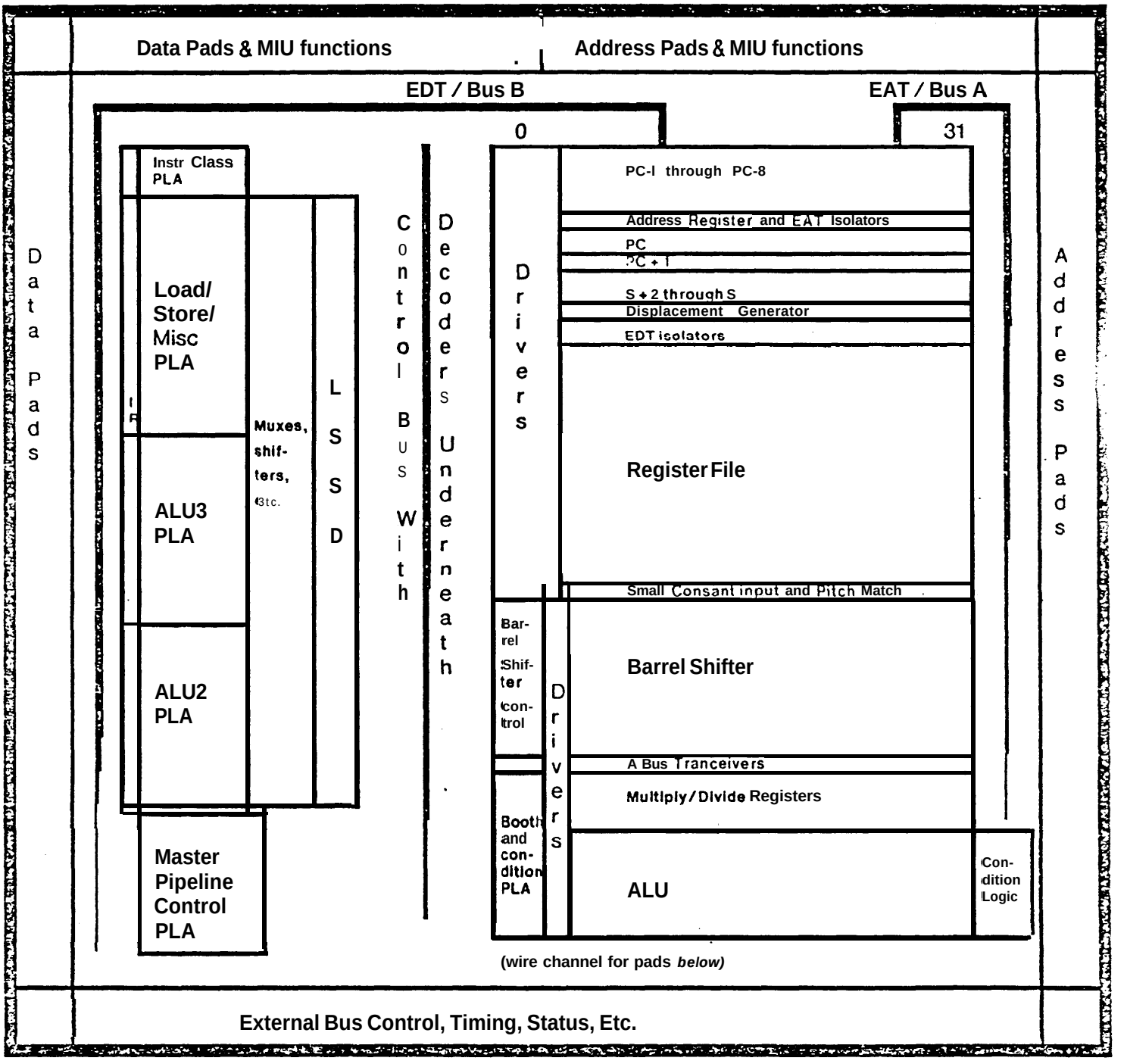
6 Present status and conclusions

The present status of the MIPS project is:

- Data path components: completely designed at the transistor level; approximately 50% laid out. The ALU has been fabricated and performs as simulated, with less than 100ns required for addition.
- Control: a SLIM [2] program for designing the control PLA's has been written and the PLAs have been generated.
- Software: code generators have been written for both C and Pascal. These code generators produce simple instructions, relying on a pipeline reorganizer. A first version of the pipeline reorganizer is running and an instruction level simulator are also in use.

Figure 3 shows the floorplan of the chip. The dimensions of the chip arc approximately 6.9 by 7.2 mm with a minimum feature size of 4 μ (i.e. $\lambda = 2\ \mu$). The chip area is heavily dedicated to the data path as opposed to control structure, but not as radically as in RISC implementation. Early estimates of performance seem to indicate that we should achieve approximately 2 MIPS (using the Puzzle program [1] as a benchmark} compared to other architectures executing compiler generated code. We expect to have more accurate and complete benchmarks available in the near future.

Figure 3: MIPS Floorplan



The following chart compares the MIPS processor to the Motorola 68000 running the Puzzle benchmark written in C with no optimization or register allocation. The Portable C Compiler (with different target machine descriptions) generated code for both processors. The MIPS numbers are a close approximation.

	Motorola 68000	MIPS
Transistor Count	65,000	25,000
Clock speed	8 MHz	8 MHz[1]
Data path width	16 bits	32 bits[2]
Static Instruction Count	1300	647
Static Instruction Bytes	5360	2588
Execution Time (sec)	26.5	6.5

Acknowledgements

Many people have contributed to the MIPS project. Among the most important other contributors are: Jim Clark, VLSI circuit ideas; Cary Komfeld, Pascal code generators; Glenn Trewitt, resource usage simulator and unified approach for exception handling, and Wayne Wolf, redesign of the barrel1 shifter.

References

1. Baskett, F. Puzzle: an informal compute bound benchmark. Widely circulated and run..

2. Hennessy, J.L. A Language for Microcode Description and Simulation in VLSI. Proc. of the Second Caltech Conference on VLSI, Caltech, January, 1981.

3. Lampson, B.W., McDaniel, G.A. and S.M. Omstein. An Instruction Fetch Unit for a High Performance Personal Computer. Tech. Rept CSL-81-1, Xerox PARC, Jan, 1981.

4. Patterson, D.A. and Sequin C.H. RISC-I: A Reduced Instruction Set VLSI Computer. Proc. of the Eighth Annual Symposium on Computer Architecture, Minneapolis, Minn., May, 1981.

5. Widdoes, L.C. The S-l Project: Developing high performance digital computers. Proc. Compcon, IEEE, San Francisco, Feb, 1980.

[1] The 68000 IC-technology is much better, and the 68000 performs across a wide range of environmental situations. We do not expect to achieve this clock speed across the same range of environmental factors.

[2] This advantage is nor used in the benchmark, i.e. the 68000 version deals with 16 bit objects while MIPS uses 32 bit objects