The Design of the 88000 RISC Family

Advances in silicon technology and RISC design bring cost-effective minimainframe performance with potential upward compatibility to the engineering workstation.

Charles Melear
Motorola, Inc.

The semiconductor industry is currently investigating the value of reduced instruction set computers, or RISCs, for computationally bound applications such as engineering workstations, Unix-based multitasking/multiuser systems, and sophisticated embedded controllers. However, RISCs are not new. Through their use in mainframes, RISCs have long demonstrated performance that is superior to that of superminicomputers and PCs.

In very simple terms, a RISC has a small, streamlined instruction set that operates at higher speeds than other techniques. Dividing a job into small parcels that can be efficiently executed by these simple instructions provides a performance advantage over conventional microprocessor techniques.

Actually, this explanation is a gross oversimplification because the performance of a RISC depends upon its entire support system. For the sake of analogy, consider a simple adder circuit built with emitter-coupled-logic (ECL) technology. The circuit can perform an add operation in just a few nanoseconds—perhaps less (a nanosecond is one billionth of a second). However, problems arise in the process of adding the two numbers to the adder circuit and dealing with the result once it is calculated. If feeding the adder circuit takes a lot longer to perform than the add, you don't gain anything by making a very fast adder circuit. Likewise, if a processor has a very rapid processing speed—but the memory system causes wait states due to access-time restrictions—processor performance becomes a moot point.

Note that the discussion of any RISC system involves much more than an examination of a very high speed computing machine. Designing an economical RISC platform requires integrating the processor into the memory-system architecture as well as generating very efficient, highly optimized machine code.

With these principles in mind, Motorola designed and implemented the RISC 88000 system in high-speed, complementary metal-oxide semiconductor (HCMOS) technology. This technology now allows cost-effective economical fabrication of chips for RISC devices. The 88000 family makes small mainframes economically available as individual engineering workstations.

0272-1732/89/0400-0026$01.00 © 1989 IEEE

Performance

Actual 20-MHz 88000 systems turn in performance ratings equivalent to 14 to 17 VAX million instructions per second (MIPS). They also achieve benchmarks of 38,000 Dhrystones/second. The system's floating-point unit (FPU) is rated at 16.5 million Whetstones/second. Both integer and floating-point instructions can achieve burst rates equal to the clock frequency.

Multiprocessing

Systems with more than one processor offer additional performance without significantly impacting the bandwidth requirements of the memory system. The 88000 architecture specifically includes hardware to facilitate the implementation of multiprocessor systems. Figure 1 is a block diagram of an 88000 system, which functions as an individual processing element.

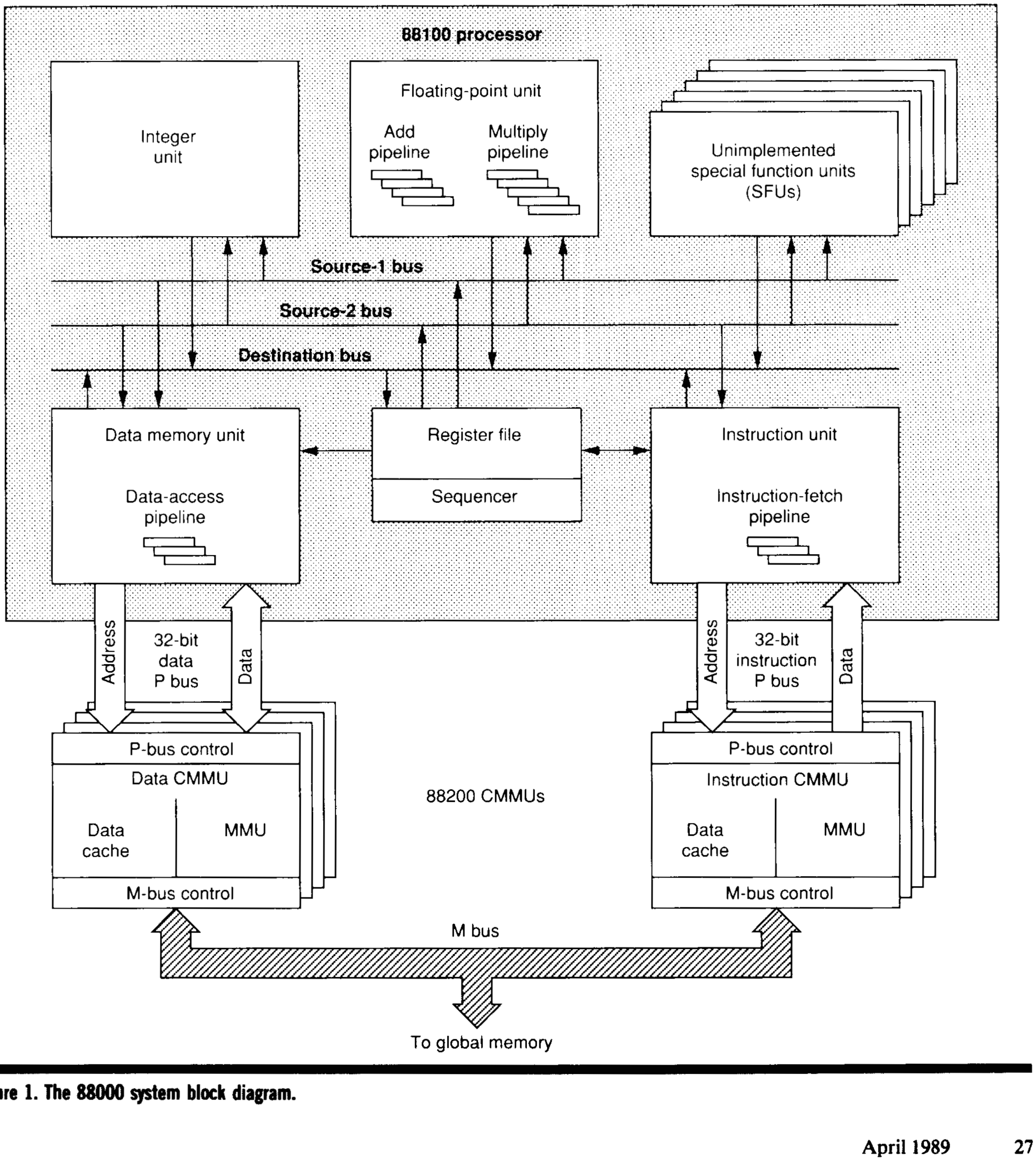


Figure 1. The 88000 system block diagram.

The memory bus (M bus) interfaces two 88200 cache memory management units (CMMUs) to the memory system. The M bus also externally connects to other processing nodes and allows them to share a common memory. Since one CMMU can share a common block of data in a multiple-node system, designers used a hardware method to ensure that all nodes have only the most current data. If one node modifies shared data, all other nodes automatically invalidate the same entry.

Another case arises when data contained in just one cache has been modified. The corresponding external memory location may not have been updated. If another processor tries to access this "stale" data, the node containing the current data automatically updates external memory before allowing the second processor to proceed. Once memory is updated, the original access proceeds correctly. This procedure solves the problem of passing stale data between nodes. Because stale data is automatically invalidated in hardware, the software operating system can devote its attention to controlling the system—rather than checking multiple copies of data for currency.

The CMMUs also provide an arbitration network on the M bus so that multiple-memory bus masters can gain access to global memory.

Object-code compatibility

The designers of the 88000 system established object-code compatibility for future versions of the 88100 processor as a fundamental goal. RISC architectures have a regular nature that tends to make upwardly compatible object code easier to achieve than in architectures that use variable-length instructions. For instance, all RISC instructions, operations, and registers are 32 bits in width. It follows that instruction-set or hardware enhancements will strictly adhere to the internal 32-bit data paths and registers. To allow variable-length instructions with extension words in a RISC architecture would greatly increase the circuit complexity without significantly increasing performance. Maintaining upward object-code compatibility is very important to many users since their most significant investment is in software. When source code is not available for recompilation, object-code compatibility becomes even more important.

Total hardware

As stated, RISC performance depends not only on the speed of the processor but also on its interface with the memory system. Performance also depends on the memory system itself. The total system consists of the 88100 processor and the two CMMUs, which interface with the global memory system. The processor has a Harvard-type architecture, which means that separate processor data/address-bus (P-bus) structures interface to the instruction and data CMMUs. Instructions can only be fetched in the code address space that is addressed by the instruction unit (see Figure 1). No data manipulation can occur in the code address space, primarily because of a read-only instruction-unit data bus. The processor operates on the data contained in the data address space. The data unit cannot fetch instructions from the data memory.

The processor generally requires a new instruction on each clock cycle of 20 MHz or more. This fact places very stringent demands on the memory system. RISC systems do not tolerate memory wait states. One wait state per instruction fetch degrades the machine performance by 50 percent. Therefore, one must either use fast—and very expensive—static RAMs in the memory system or employ some type of caching technique. For the sake of economy, designers chose a caching technique for the 88000 system. This approach allows the use of slower—but more dense—dynamic RAMs for the main memory. A properly sized cache memory can hide most of the wait states from the bulk memory.

Special function units

The 88000 architecture can accommodate more than one functional block that independently executes instructions. The FPU is an example. Special function units (SFUs) sit on the four internal buses. Instructions and operands can be dispatched to and results returned from any one of the execution units as selected by a 3-bit field in the instruction. The field allows logical room for eight SFUs. The integer unit, although it is addressed like an SFU, does not technically fall into that category. But because one of the eight possible codes is taken up by addressing the integer unit, only seven SFUs can be implemented.

Designers chose the FPU as the first and only implemented SFU in the 88000 architecture (Figure 1). All internal registers physically reside in this functional block. Therefore, the FPU could be removed from the 88100 without affecting any other portion of the device. The sole consequence would be the loss of floating-point instructions. This methodology implements a building-block strategy. In the future, system designers can make selections from an SFU library to include in their versions of the 88000 microprocessing unit.

Overview of the 88100

Let's develop an understanding of the internal workings of the processor before further exploring the system aspects of the devices.

The 88100 uses HCMOS logic. It qualifies as a RISC because of its general attributes of single-cycle instruction-execution times and fixed-instruction lengths. It also has a smaller instruction set than conventional

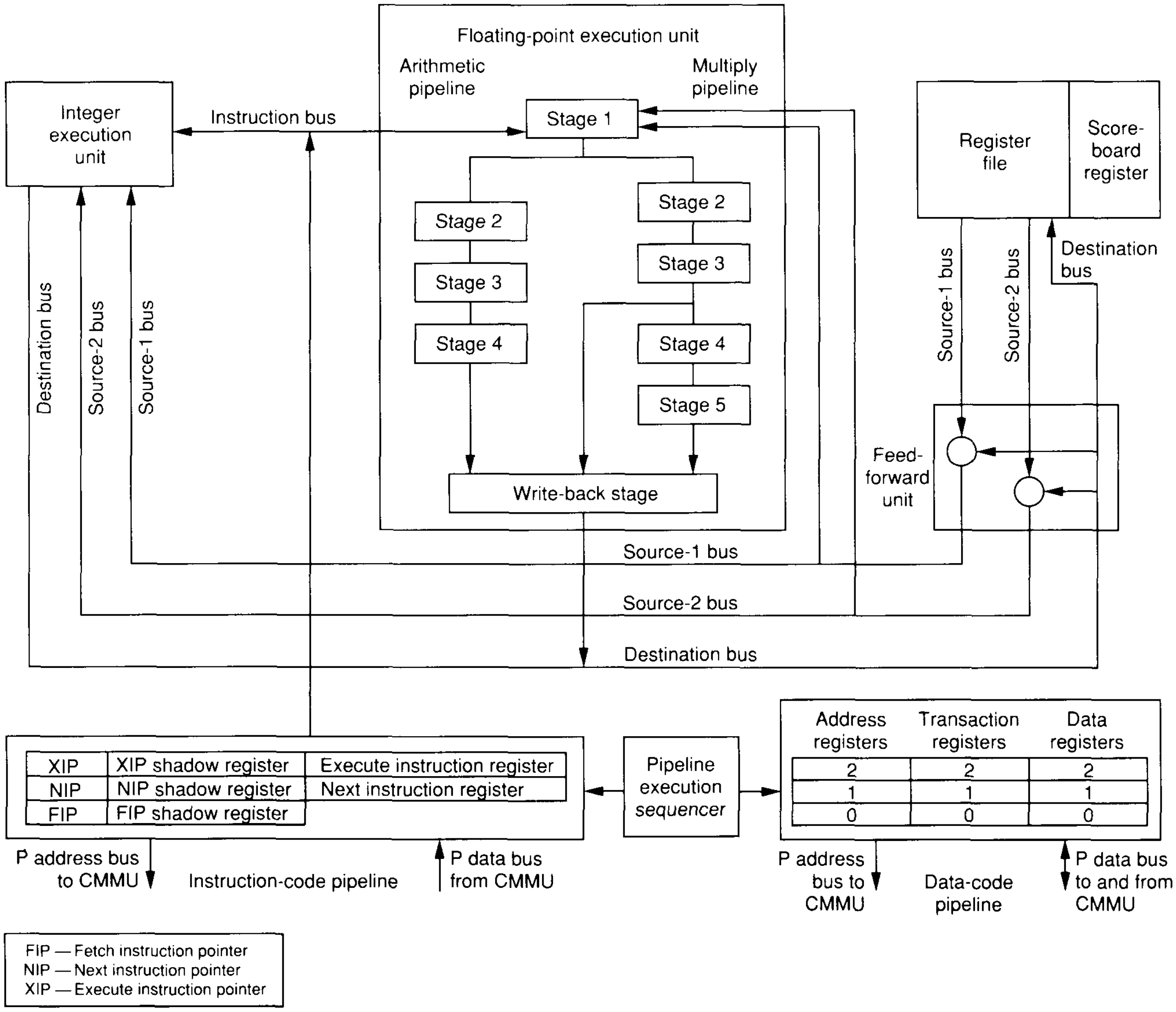


Figure 2. The 88100 processor block diagram.

microprocessor techniques as well as a greatly reduced number of memory addressing modes. In addition to these common RISC attributes, the 88100 implements the just-discussed floating-point arithmetic unit on chip. The FPU is an actual execution unit for the machine that shares its chip real estate.

Figure 2 provides a detailed block diagram of the processor, which employs a dual P-bus system. One P bus serves the instruction memory under instruction-unit control. The second P bus serves the data memory under data-memory unit control. This methodology allows simultaneous instruction fetches along with data-memory transactions.

The machine is parallel in nature, that is, it can work on up to 15 instructions simultaneously. All execution units can perform useful work at the same time. Three memory transactions can progress in the data-memory unit, the instruction unit can fetch one instruction while decoding another, the integer unit can execute an instruction, and the FPU can be executing up to nine instructions—all at once.

Integer-execution unit

The integer unit shown in Figures 1 and 2 executes single-cycle instructions, which essentially include all instructions except Multiply, Divide, Memory-Access, and Floating-Point. Like other execution units, the integer unit connects to four separate buses: instruction,

Table 1.
Exception vectors.

Number	Address	Definition
0	0	Reset (the VBR is cleared before vectoring)
1	VBR + $8	Interrupt
2	VBR + $10	Instruction access exception
3	VBR + $18	Data access exception
4	VBR + $20	Misaligned access
5	VBR + $28	Unimplemented opcode
6	VBR + $30	Privilege violation
7	VBR + $38	Bounds check violation
8	VBR + $40	Integer divide error
9	VBR + $48	Integer overflow
10	VBR + $50	Error
11-113	— —	Reserved for supervisor and future hardware use
114	VBR + $390	SFU1 precise—floating-point precise exception
115	VBR + $398	SFU1 imprecise—floating-point imprecise exception
116	VBR + $3A0	SFU2 precise*
117	VBR + $3A8	Reserved
118	VBR + $3B0	SFU3 precise*
119	VBR + $3B8	Reserved
120	VBR + $3C0	SFU4 precise*
121	VBR + $3C8	Reserved
122	VBR + $3D0	SFU5 precise*
123	VBR + $3D8	Reserved
124	VBR + $3E0	SFU6 precise*
125	VBR + $3E8	Reserved
126	VBR + $3FO	SFU7 precise*
127	VBR + $3F8	Reserved
128-511	— —	Supervisor call exceptions—reserved for user definition

* SFU2 through SFU7 are not implemented. Executing an instruction that is coded for these SFUs causes a precise exception for that SFU.

source-1 and -2 operand, and destination. The source-1 and -2 buses carry operands from the register file or the embedded field of an instruction to an execution unit or SFU. The destination bus returns results to the register file. Instructions dispatch along with the associated operands to the integer unit, and the result returns on the destination bus in one cycle. The system can dispatch a new instruction and receive the result in one cycle. Thus, the integer unit can achieve an execution rate equal to the clock rate.

The overall function of the integer unit is to execute instructions that are dispatched to it by the instruction unit. The integer unit contains dedicated hardware that performs specific functions to complete "difficult" instructions in one clock cycle.

One section calculates numerical results from instructions such as Add and Subtract. Another section is used specifically for bit-field instructions that set, clear, extract, and rotate register fields. A dedicated add unit calculates target addresses for Branch and Jump instructions. As each instruction is fetched, branch-target-calculation circuitry uses part of the instruction operand to calculate a branch target address, whether the just-fetched instruction is a Branch or not. On the next cycle, the sequencer (which controls all instruction and data flow) determines whether or not the instruction is a Branch. If it is, a precalculated target address waits to be used as a fetched instruction pointer (described later). If the instruction is not a Branch, the sequencer simply discards the resultant address calculation.

The instruction pipeline fetches and partially decodes instructions before they are actually dispatched to the appropriate execution unit. During each clock cycle, the pipeline can fetch one instruction, partially decode another one, prefetch any needed operands, and dispatch a third instruction to an execution unit. The pipelined structure is necessary because there is not enough time within one clock cycle to fetch, decode, and execute an instruction. However, dividing the job into sections and using a pipeline technique moves instructions through the instruction pipeline at the rate of one per clock cycle.

The feed-forward unit also speeds program execution. When an instruction dispatches to an execution unit that requires the result of the previous instruction, a problem occurs. There is no time to write the previous result into the register file and make that result available as an operand for the next instruction. When a result is needed, the feed-forward unit solves the problem by taking the result of the previous instruction and routing it on a source-operand bus on the next cycle.

The internal buses of the device carry the instruction, both operands, and the result—all in one cycle. At the speeds that RISCs require, it is not feasible to multiplex internal buses. Therefore, designers implemented separate buses to carry the instruction, operands, and results from the instruction pipeline to the execution units and the register file. The 32-bit bus consumes a great deal of silicon area, but no other acceptable method exists for transferring four 32-bit values in one cycle, as RISC technology requires.

Exception processing

Exceptions can come from a number of sources. Table 1 presents the exception vectors. The exception-vector address can be formed in one of two ways. Concatenating a particular value to the vector base register (VBR) of the integer unit forms hardware exception addresses. For example, say a data-access exception occurs. The fetched-instruction pointer points to the address formed by concatenating the 20-bit VBR with $18, which contains the first instruction of the exception routine. The exception-vector address for instructions such as Trap on Bit Set is formed by taking the 20-bit value in the VBR and concatenating the low-order 9 bits of the Trap instruction followed by three zeros to form a 32-bit address. This address is the location of the first instruction of the exception routine.

Exception processing begins when an external interrupt or any enabled hardware exception occurs. The shadow-freeze (Sfrz) bit of the processor-status register (PSR) sets. Table 2 shows the integer-unit control registers. All trap-time and shadow registers freeze during this cycle, including

- the trap PSR,
- the shadow scoreboard register,
- the shadow registers for the Execute, Next, and Fetch instruction pointers, and
- the shadow registers for the data-memory unit.

All SFUs freeze, that is, instruction processing stops in place. The instruction unit fetches the appropriate instruction in the exception-vector table.

A Trap instruction also initiates exception processing. However, a Trap allows the machine to synchronize itself. That is, before the Trap is actually issued, all memory transactions and floating-point instructions can complete. Then the shadow registers freeze and exception processing continues by fetching the instruction pointed to by the exception vector. (This vector is formed by concatenating the lower 9 bits of the Trap with the VBR.)

Multiple exceptions require some additional processing. Once an exception happens, the Sfrz bit sets and the shadow registers freeze. All SFUs are disabled. If an exception is taken while the Sfrz bit is set, the shadow registers do not reflect the values of the runtime registers. All necessary shadow and general-purpose registers must be stored in external memory to allow nested exceptions. The exception handler software then reenables the SFUs when appropriate, depending upon the cause of the exception. Then software clears the Sfrz bit. This procedure reenables the shadow mode, in which the shadow registers update on a cycle-by-cycle basis and become mirror images of the runtime registers again. This process repeats each time a nested exception occurs.

To return from an exception condition, the shadow registers must contain the appropriate machine context for program return. Setting the Sfrz bit, loading the shadow registers with the desired values, and executing a return from exception (Rte) instruction accomplishes the return. The Rte automatically writes the shadow registers to the runtime registers and clears the Sfrz bit.

Table 2.
Register model of the 88100 integer unit.

Control register no.	Mnemonic	Description of register
0	PID	Processor identification
1	PSR	Processor status
2	TPSR	Trap processor status
3	SSBR	Shadow scoreboard
4	SXIP	Shadow Execute instruction pointer
5	SNIP	Shadow Next instruction pointer
6	SFIP	Shadow Fetch instruction pointer
7	VBR	Vector base
8	DMT2	Transaction 2
9	DMD2	Data 2
10	DMA2	Address 2
11	DMT1	Transaction 1
12	DMD1	Data 1
13	DMA1	Address 1
14	DMT0	Transaction 0
15	DMD0	Data 0
16	DMA0	Address 0
17	SR0	Supervisor storage 0
18	SR1	Supervisor storage 1
19	SR2	Supervisor storage 2
20	SR3	Supervisor storage 3

If further exceptions are prevented by not clearing the Sfrz bit while the current exception processes, it is not necessary to save the shadow registers. They cannot be overwritten. In this case, an Rte automatically returns to the normal context without saving or restoring the shadow registers.

FPU execution

The FPU executes all floating-point instructions as well as Integer Multiplies and Integer Divides. As shown in Figure 3 on the next page, the FPU is a pipelined structure. Thus, the result for the instruction is not ready for several cycles. However, if a result is ready from a prior instruction, it can return to the register file via the destination bus during this cycle. The pipelined nature of the FPU allows a new instruction to

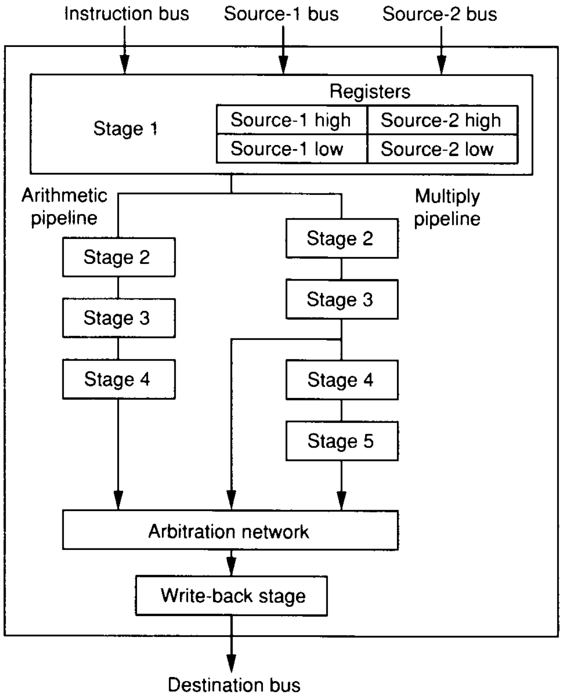


Figure 3. FPU block diagram.

be dispatched on each clock cycle as long as a pipeline stall does not occur. The system can dispatch a double-precision, floating-point instruction on every second clock cycle.

The FPU has two pipelines: arithmetic and multiply. The arithmetic pipeline is used for most floating-point instructions, including Integer Divide and Floating-Point Divide. The multiply pipeline is designed for Integer and Floating-Point Multiplies. Both pipelines begin and end in a common stage. The type of instruction determines which pipeline is used. A single-precision, floating-point instruction cycles to the next stage of the appropriate pipeline on each clock cycle. In the case of double-precision, floating-point instructions, the calculations break into upper and lower words. Each stage in the FPU must first operate on the upper word of the operands and—on the next cycle—on the lower word. For this reason, a new double-precision, floating-point instruction can only start on every second clock cycle.

Write-back arbitration. A result can reach the write-back stage of the FPU in three ways. Integer Multiply instructions flow through stage 3 of the multiply pipeline and then directly to the write-back stage. Floating-point instructions move through stage 5 of the multiply pipeline before reaching the write-back stage. All floating-point arithmetic instructions use all four stages of the floating-point arithmetic pipeline. Three instructions can attempt to deliver results to the write-back stage in the same cycle. In this case, the arbitration network gives priority as follows: Integer Multiply instructions, Floating-Point Multiply instructions, and the floating-point arithmetic pipeline. A long string of consecutive Integer Multiply instructions stalls stage-4 instructions in the arithmetic pipeline and stage-4 and -5 instructions in the multiply pipeline. The arbitration network does not grant access to the write-back stage until the Integer Multiplies complete. Carefully written software generally minimizes the stalling effect of one pipeline on another when data-dependent code sequences occur.

Destination-bus priority. The FPU connects to the destination bus, as do the integer and data-memory units. All three units can have a result ready to drive on the destination bus during the same cycle. In this case, the integer unit has first priority, the FPU second, and the data-memory third. When a long string of integer-unit instructions occurs, the other two units do not get a write slot on the destination bus. Their results have to sit until a write slot occurs.

Various mechanisms exist for assigning write slots to the three units that sit on the destination bus. When an integer-unit instruction issues, it always receives a write slot. If this unit doesn't need the slot—say an integer-unit instruction does not execute during this cycle or the instruction does not generate a result—the FPU can use that slot. The issuing of each floating-point or data-memory-unit instruction creates a write slot if a floating-point result is ready in the write-back stage. Otherwise, this slot goes to the data-memory unit. Whenever the instruction pipeline (discussed later) stalls—causing no instruction to be issued—the system grants a write slot to the highest priority unit that has data waiting to go onto the destination bus.

Floating-point registers. Table 3 illustrates the register model for the FPU. The first nine floating-point control registers (Fcr0-Fcr8) can only be accessed in the supervisor mode and are used for exception processing.

The floating-point user-status and -control registers (Fcr62-63) can be accessed in either user or supervisor mode. The floating-point control registers primarily hold information on instructions that cause exceptions in the FPU. By saving the state of the FPU for the instruction that caused the exception, software can attempt to correct the condition needing service and restart the FPU.

The processor hardware updates the floating-point exception-cause register to indicate the following floating-point exceptions:

- a conversion to integer overflow,
- an unimplemented floating-point instruction,
- a control-register-privilege violation (attempt to access in user mode),
- a floating-point reserved-operand check, and
- a divide by zero.

The following floating-point imprecise exceptions can be signaled as well:

- an underflow,
- an overflow, and
- an inexact condition.

Very simply put, a precise exception is one that signals as soon as the instruction reaches the integer unit or any SFU. For instance, the FPU knows immediately when it receives an unimplemented opcode. In this case,

- the appropriate flag in the exception register sets,
- the exception register points to the offending instruction, and
- the source-1 and source-2 operand high and low registers of the FPU store the operands that were issued with the offending instruction (see Table 3).

User-supplied software routines can handle exception recovery. For instance, an unimplemented opcode can be deliberately inserted in the user's code. The exception handler can decode the opcode portion of the instruction and perhaps run a synthesized instruction (such as trigonometric or hyperbolic) in software.

Imprecise exceptions, on the other hand, do not signal until the instruction has nearly completed. For instance, an underflow (a result with an exponent of -127) does not signal until the result is actually calculated. An underflow is not always fatal. For instance, the number 1101×10^{-127} can also be represented as 11010×10^{-126}. The uncertainty in the last digit may be acceptable for a particular application. It does allow program execution to continue. However, by the time that the offending instruction generates an exception, the instruction pointers no longer point to the instruction. This condition makes it impossible to identify the actual instruction that caused the imprecise exception.

The imprecise operation-type register contains the information that determines what action to take—along with the appropriate instruction information to continue execution. That is, the register contains

- the exponent of the inexact result,
- whether the result is single or double precision,
- the 5-bit opcode that identifies the instruction type,
- which exception handlers became enabled, and
- the destination register for the result.

The floating-point-result high and low registers store the mantissa of the actual inexact result. The floating-point high register also contains information about the rounding modes and the guard, round, and sticky bits that can provide additional bits of accuracy in the result. From this information, software can complete an instruction that caused an imprecise exception.

Table 3.
FPU control registers.

Floating-point control register no.	Mnemonic	Floating-point registers
0	FPECR	Exception cause
1	FPHS1	Source-1 operand high
2	FPLS1	Source-1 operand low
3	FPHS2	Source-2 operand high
4	FPLS2	Source-2 operand low
5	FPPT	Precise-operation type
6	FPRH	Result high
7	FPRL	Result low
8	FPIT	Imprecise-operation type
9-61	—	Unimplemented
62	FPSR	User status
63	FPCR	User control

Register file

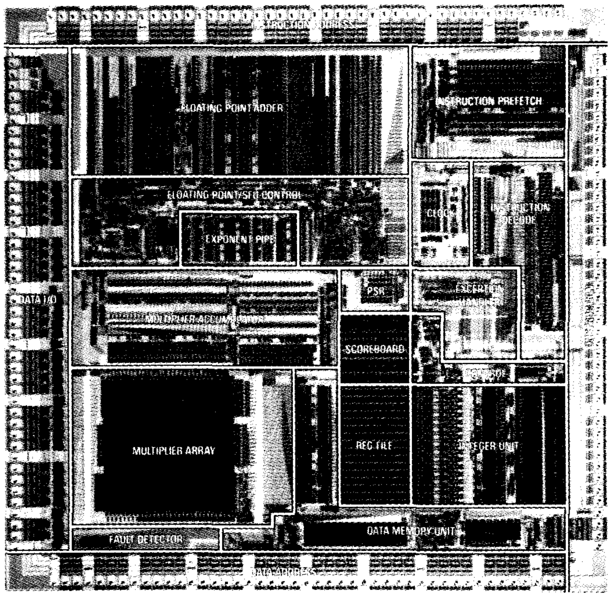
The register file (shown in Figure 2) consists of thirty-two 32-bit, general-purpose registers that source instruction operands and receive the calculated results. The first register, R0, is unique in that the system always reads it as zero and cannot write to it. This condition creates a constant of zero to be used as a source operand, which is convenient for such things as synthesizing single-cycle register-to-register moves. Adding any register to R0 and storing the result in the chosen destination register accomplishes this task.

The R1 general-purpose register also has a special property. The system automatically stores the return address for a Branch or Jump to Subroutine in R1.

The remaining 30 general-purpose registers serve as the source or destination of instruction operands and results. No hardware conventions exist for the use of these registers. Recall that all source operands must come either from embedded fields in the instructions or the register file. Data in external memory must first be loaded into a register in the register file before an instruction can use the data as an operand.

Scoreboard register

This register ensures that a source operand is not fetched from a register that is currently waiting for a result. If operands are available, this hardware scheme



Photomicrograph of the 88100 microprocessor.

lets instructions be dispatched to idle execution units while other instructions are in progress.

The scoreboard register contains 32 bits; one bit corresponds to each register in the register file. When a multiple-cycle instruction is issued, a bit sets in the scoreboard register that corresponds to the register that receives the results of the instruction. The bit clears when the result of the instruction writes to the destination register. Once the scoreboard bit is set, a subsequent instruction cannot use that register for a source operand until the bit clears, which indicates the result of a previous instruction has been delivered. If an instruction reaches the execution stage of the instruction pipe—and tries to fetch an operand from a register with a set scoreboard bit—the instruction pipeline stalls. The instruction does not move to the appropriate execution unit until the scoreboard bit clears. All integer-unit instructions execute in a single cycle. Therefore, integer-unit instructions cannot stall the instruction pipeline.

While instruction-pipeline stalls are inevitable to some degree in any code, properly written software takes advantage of the machine's architecture and arranges instructions in the best possible order to maximize the throughput rate.

Alternatively, the compiler or software writer can deliberately install NO-OPs after multicycle instructions to ensure their completion before another instruction is fetched.

Scoreboarding provides absolute protection from problems that could otherwise arise from out-of-order execution models.

Instruction unit

The instruction unit shown in Figure 1 fetches and partially decodes instructions. All instruction-unit registers are accessible through the integer-execution unit. The instruction unit is a three-stage, pipelined structure consisting of Fetch, Next, and Execute stages.

The Fetch stage consists of the fetched instruction pointer (FIP) and its shadow register (see Figure 2). At the beginning of each cycle, assuming no pipeline stalls or memory wait states occur, the FIP issues a new address to the memory system. This address is either the previous address plus 4 bytes or the target address of the *currently executing flow-control instruction.*

The second, or Next, stage of the instruction pipeline consists of its instruction pointer (NIP), the NIP shadow register, and the Next instruction register. The FIP of the previous cycle shifts to the NIP, and the corresponding instruction from the memory system returns to the Next instruction register. At this time, the instruction is partially decoded, and any needed operands from the register file are prefetched and prepared for transfer to the appropriate execution unit.

The third, or Execute, stage consists of the its instruction pointer (XIP), the XIP shadow register, and the Execute instruction register. During this stage, the instruction dispatches to the appropriate execution unit. If an exception occurs during any cycle, the shadow registers that maintain real-time copies of their corresponding runtime registers freeze, maintaining the value at the time of the exception as well. This process saves the state of the machine and allows exception processing to begin immediately.

Branch-execution enhancement

The instruction unit handles a special problem associated with flow-control instructions. The sequence for a flow-control instruction is the same as any other instruction through the instruction pipeline. The Branch or Jump is fetched in cycle 1. During cycle 2, the Branch or Jump shifts to the Next-instruction slot and the branch target address is calculated. A new instruction is also fetched during this cycle. During cycle 3, the Branch or Jump goes to the Execute slot and the calculated target address writes into the FIP pointer, which outputs the address onto the external code-address bus. A problem arises: The instruction immediately following the Jump has already been fetched and partially decoded. This instruction normally would not be needed because the program has just been directed to another spot. Therefore, the instruction in the Next slot would normally be invalidated, causing a "hole" in the instruction pipeline.

By using clever programming techniques, one can usually place a useful instruction immediately after a flow-control instruction and obtain a useful result. An "execute next" option provides this flexibility. This option can be enabled or disabled for each individual flow-control instruction (Branch or Jump). It causes the instruction immediately following the flow-control instruction to execute whether the Branch is taken or not.

Now consider the exception-vector table (Table 1). The vectors are aligned on double-word boundaries. Thus the table can hold two instructions per vector. When exception processing vectors to a particular location in the exception-vector table, a flow-control instruction is normally encountered that directs program execution. Placing the first instruction of the exception routine immediately after the flow-control instruction in the table and using the execute-next option lets the flow-control instruction point to the second instruction. Under these conditions, no hole occurs in the instruction pipeline.

Data-memory unit

This unit performs all data-memory transactions (see Figure 1).

When a Load, Store, or Exchange instruction is issued, the sequencer sends the instruction to the data-memory unit. Three register-indirect addressing modes address external memory:

- 16-bit immediate offset,
- indexed offset, and
- scaled-indexed offset.

A dedicated add circuit in the data-memory unit calculates the logical effective address for these addressing modes. This circuit can add a register to a 16-bit immediate value embedded in the instruction, add two registers together, or add two registers together after a scaling operation. It performs the last function after shifting the second register 0, 1, 2, or 3 places to the left to form the effective address. Loads and Stores result in multiple-cycle operations, but they dispatch at the rate of one memory-access instruction per clock cycle because of the pipelined nature of the data unit.

The data-memory unit is a three-stage structure. Each stage contains three registers: address, transaction, and data (see Figure 4). The address register contains the effective address of the memory transaction. The transaction register contains information such as the size of the transaction and its destination register. The data register contains the data to be stored.

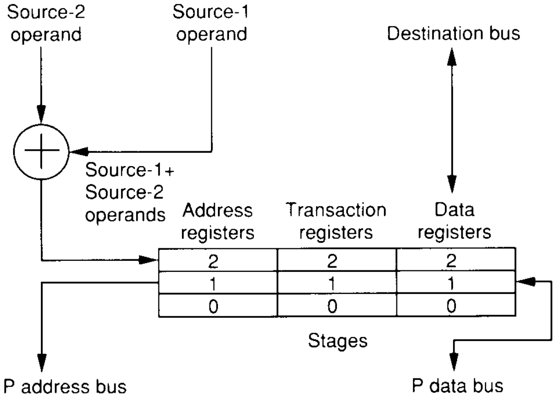


Figure 4. Data-memory unit organization.

In a typical sequence, a store transaction issues the indirect address plus the indexing value to stage 0 of the data pipeline in which the effective address is calculated. The contents of stage-0 registers shift to stage 1 on the next cycle, and the address and data register contents apply to the data-unit address and data buses.

Stage 1 is necessary because there is not enough time to calculate the effective address, apply the address to the external bus, and allow for any appreciable address and data setup times for memory. Stage 1 makes the entire cycle available for a memory access, which greatly reduces the bandwidth requirements of the memory system. During the third cycle, the contents of the stage-1 registers are shifted to stage 2. In the case of a Load instruction, the data returns to the data unit via the data-side data bus.

The only purpose of stage 2 is to maintain a copy of stage-1 information for one additional cycle. This feature allows the implementation of virtual memory systems. If a fault occurs for a memory transaction, the memory system returns the fault signal on the cycle following the access. Thus, when a memory fault or exception signals, the corresponding information about the memory access freezes in stage 2 of the data unit. All memory faults are not lethal. If the exception is caused by a page fault, the handler can find the appropriate section of the program on disk memory, read the required portion of the program into active memory, and modify the memory mapping registers as needed. The execution handler can examine the registers in stage 2 of the data unit and reconstruct the memory transaction that previously faulted. After retrieving the appropriate portion of the program, the memory access can now complete successfully.

Table 4.
Instruction-set summary.

Mnemonic	Description
Integer arithmetic instructions	
add	Add
addu	Add unsigned
cmp	Compare
div	Divide
divu	Divide unsigned
mul	Multiply
sub	Subtract
subu	Subtract unsigned
Floating-point arithmetic instructions	
fadd	Add
fcmp	Compare
fdiv	Divide
fldcr	Load from floating-point control register
fit	Convert integer to floating point
fmul	Multiply
fstcr	Store to floating-point control register
fsub	Subtract
fxcr	Exchange floating-point control register
int	Round floating point to integer
nint	Round floating point to nearest integer
trnc	Truncate floating point to integer
Logical instructions	
and	AND
mask	Logical mask immediate
or	OR
xor	Exclusive OR
cir	Clear bit field
Logical instructions (cont'd.)	
ext	Extract signed bit field
extu	Extract unsigned bit field
ff0	Find first bit clear
ff1	Find first bit set
mak	Make bit field
rot	Rotate register
set	Set bit field
Load/Store/Exchange instructions	
ld	Load register from memory
lda	Load address
ldcr	Load from control register
st	Store register to memory
stcr	Store to control register
xcr	Exchange control register
xmem	Exchange register with memory
Flow-control instructions	
bb0	Branch on bit clear
bb1	Branch on bit set
bcnd	Conditional branch
br	Unconditional branch
bsr	Branch to subroutine
jmp	Unconditional jump
jsr	Jump to subroutine
rte	Return from exception
tb0	Trap on bit clear
tb1	Trap on bit set
tbnd	Trap on bounds check
tcnd	Conditional trap

Instruction set

The RISC instruction set is relatively small in comparison to other kinds of computer architectures. RISC instructions are implemented in hardwired logic. One must add new instructions carefully. Any additions must be absolutely necessary because the logic needed to implement them also greatly impacts circuit density and size. One must evaluate a new instruction in terms of how it can improve overall processor performance. Writing compilers and simulation models of the processor—and evaluating the performance of the instruction set versus the compiler—accomplishes this purpose. Table 4 presents the 88100 instruction set.

RISCs must use many hardware techniques to gain performance, even at the expense of creating larger circuit sizes. Remember that RISC instructions are very elemental; several RISC instructions generally equal one conventional instruction. Therefore, if a RISC is to obtain significant performance improvements, the machine must execute instructions on the highest possible percentage of clock cycles.

As stated, all 88000 instructions are 32 bits wide. The system contains no extension words or instructions shorter than 32 bits. Instruction-set implementation allows for streamlining the internal decoding circuitry. Figure 5 demonstrates the encoding pattern for an Add instruction. Instruction alignment circuitry is unnecessary. Bits 31-26 define which instruction executes.

Bit positions 25-21, 20-16, and 4-0 of an instruction always specify the destination source-1 and source-2 registers. The internal decoding circuitry does not have to locate and align a particular field in the instruction. These fields are always in the same place no matter what the instruction is. This method reduces the amount of circuitry needed to produce a high-performance RISC implementation.

Arithmetic instructions. These instructions include Add, Subtract, Compare, Divide, and Multiply. (Add and Subtract have signed and unsigned forms.) Other architectures can have several variations of particular instructions. For instance, an Add instruction can possess different forms that take advantage of the size of the data field or the location of the data. The RISC methodology does not allow as many forms of instructions because the large circuit size would make an IC unsuitable for manufacture.

Certain hardware techniques create a flexible instruction set. Consider the Add instruction, for which only two forms exist. The first adds the contents of two registers of the register file and delivers the result to a third register. The second adds the contents of a register to a 16-bit immediate field embedded in the instruction. A dedicated bit in the instruction enables/disables the overflow exception. Another bit causes the carry bit to be included/not included in the calculation. Yet another bit causes a carry bit to be generated/not generated. Implementing one instruction allows eight basic variations of Add: signed and unsigned, with or without underflow/overflow, and with or without carry.

The immediate form of Add is always without carry. This exact same scheme generates the eight basic forms of the Subtract instruction.

Condition codes. The Compare instruction calculates these codes. It compares two registers with one another or one register with a 16-bit immediate value embedded in the instruction modes. The results are placed in a destination register. Figure 6 shows the encoding pattern for the resultant condition codes.

Condition codes are not explicitly generated as each instruction executes. Condition codes need to be calculated only when they are used for a following conditional flow-control instruction such as a Branch on Condition. The circuitry needed to generate condition codes in machines with out-of-order execution models is quite complex. Therefore, designers implemented the Compare instruction to explicitly generate condition codes when needed and execute a Branch on Bit Set/Clear to emulate the desired Branch instruction.

111101	D	S1	011100	I	O	000	S2

Add R7, R8, and R1. Add contents of R8 to R1 and place results in R7.

111101	00111	01000	011100	1	1	000	00001

D	5-bit field specifying destination register
I	Enable/disable carry in
O	Enable/disable carry out
S1	5-bit field specifying source-1 register
S2	5-bit field specifying source-2 register

Figure 5. Add-instruction encoding.

111101	D	S1	011111	00000	S2

31	11	10	9	8	7	6	5	4	3	2	0
	hs	lo	ls	hi	ge	lt	le	gt	ne	eq	

Compare result written to destination register.

D	5-bit field specifying destination register
eq	Equal
ge	Greater than or equal
gt	Greater than
hi	Higher than
hs	Higher than or the same
lt	Less than
le	Lower than or equal
lo	Lower than
ls	Lower than or the same
ne	Not equal
S1	5-bit field specifying source-1 register
S2	5-bit field specifying source-2 register

Figure 6. Compare-instruction encoding.

Logical and flow-control instructions. Bit-field instructions extend RISC instruction sets designed for prior machines. Special hardware within the integer unit facilitates the execution of bit-field instructions. The fields on which these instructions operate can be of any width and located anywhere in the word. Bit-field hardware can clear, set, extract, and insert fields into registers. This hardware can essentially perform a single-cycle shift of any number of bits to a field of any width. The only limitation is that the amount of the shift plus the width of the affected field must be less than the width of the 32-bit register. Also, the Rotate instruction always rotates the entire contents of a 32-bit register, that is, the field width is always 32 bits.

Floating-point instructions. These instructions are also a modern extension of typical HCMOS RISC architectures. RISCs generally implement only the most basic arithmetic, logical, and flow-control instructions. Floating-point instructions vary from typical RISC techniques because of their multiple-cycle execution times. In this particular case, designers implemented enough hardware to perform floating-point arithmetic in a pipelined, sequential fashion. Placing the FPU on the internal silicon buses (which can provide a new floating-point instruction on every clock cycle) yields superior performance. In fact, the variance from standard single-cycle execution is well worth the additional cost that results from additional circuit size.

Although this article has described a data-processing machine that inhabits the very upward limit of HCMOS-microprocessor design and silicon-processing technology, remember that RISCs gain performance because developers fine-tune the entire system. The system must contain enough of the right instructions to allow compilers to generate efficient code. It means nothing if the speed of the processor doubles but requires four times as many instructions. A memory system that cannot supply instructions to the processor without wait states gains nil. Designers must interface each subsystem of the chip design as efficiently as possible with all other subsystems.

RISC systems have actually been around for some 25 years, but their primary application has been in mainframes built with ECL technology. Present silicon-wafer processing techniques allow very large systems to be built in HCMOS technology.

If history is any guide, some of the emerging RISCs will yield additional processing power at an increasingly cost-effective level to bring mainframe performance to the desktop market. Perhaps the real challenge is how to put that power to work in new and exciting applications.

Readers can direct questions concerning this article to the author at Motorola, Inc., 6501 William Cannon Drive West, MS/OE33, Austin, TX 78735-8598.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 153 **Medium** 154 **High** 155