# IBM RISC System/6000 processor architecture

by R. R. Oehler
R. D. Groves

**This paper describes the hardware architecture of the IBM RISC System/6000* processor, which combines basic RISC principles with a *partitioning of registers by function into multiple* ALUs. This allows a high degree of parallelism in execution and permits a compiler to generate highly optimized code to manage the interaction among parallel functions. Floating-point arithmetic is integrated into the architecture, and floating-point performance is comparable to that *of many vector processors.***

## Introduction

The first RISC machine, the 801, was invented at the IBM Thomas J. Watson Research Center in 1975. Since that time, within IBM, there have been several implementations based on the original ideas. Each of these implementations extended the ideas in different ways, including virtual addressing, coprocessor extensions, and I/O control. Although most of these implementations never became part of any IBM product, some did—the most visible RISC-based product has been the IBM RT System.

*RISC System/6000 is a trademark of International Business Machines Corporation.



Starting in 1985, most of the original 801 research team again considered the issues of machine architecture, examining not only System/370, but also factoring in the experience gained from the 801 and its follow-ons. Studies were performed on floating-point organization and performance, the effectiveness of the architecture as a compiler target, and, most significantly, reexamining the effects of machine organization and architecture on parallel execution and pipelining.

There are three components in the processor performance equation: number of instructions (path length), cycles per instruction, and cycle time. Since there already existed a dataflow model of the original architecture that had concentrated on reducing the levels of logic required to perform a cycle, it was felt that as long as this model was not significantly changed, the benefits of newer technology with higher levels of integration would reduce cycle time. Consequently, no architectural effort was focused on reducing cycle time. Instead, emphasis was placed on reducing both number of instructions and number of cycles per instruction. To reduce the number of instructions required to perform a task, high-leverage compound-function instructions were explored which could replace two or more of the original 801 instructions. The original design target for the 801 was to execute one instruction every cycle. The emphasis of this research study was to define an architecture whose implementations could easily execute more than one instruction per cycle, otherwise known as a superscalar architecture.

The result of this activity led to a second-generation RISC architecture, the so-called "AMERICA architecture." A small effort was initiated at the Watson Research Center to design a system based on this

architecture. This work progressed very rapidly, and it soon became clear that such a system was feasible and could realize the expected performance.

In 1986, the IBM development laboratory in Austin, Texas, which developed the RT System, accepted the AMERICA architecture and began developing new products based on it. This original research idea has now been implemented in the IBM RISC System/6000* (RS/6000) processor version of IBM's POWER architecture. What follows is a general description of the new architecture, including many examples of how this second-generation RISC architecture achieves significant performance improvements over earlier RISC architectures.

## RISC System/6000 architecture

One of the most notable features of the RS/6000 architecture is the separation of the components of the processor into functional units. There are three major units: fixed-point, floating-point, and branch. Each of these components can process instructions in parallel, with the branch unit in overall control and responsible for the integrity of program execution.

Obviously, all CPU architectures provide for fixed-point and branch functions, and many even provide for floating-point. What makes the RS/6000 architecture different is how it applies the original 801 philosophy, which is to increase the role of the compiler and operating system in managing the hardware, thereby simplifying the hardware design. Unlike many other CPU designs in which a significant amount of hardware is devoted to trying to provide maximum parallelism among functional units by elaborate register-dependency checks, branch prediction, branch-history tables, store/load interlocks, etc., the RS/6000 architecture has avoided the need for most of this hardware by assigning registers to functional units, separating the setting of conditions from normal instruction activity, and requiring program management of some store/load interactions. This exposes the latent parallelism of the processor to the compiler and operating system, requiring explicit management of the various units and their interactions in order to fully exploit the hardware. And, like the original 801, it sometimes even *requires* this management for programs to function properly.

The second distinguishing feature of the RISC System/6000 architecture is its emphasis on floating-point performance. Very early in the project, an advanced floating-point multiply-add design was conceived that would permit a 64-bit multiply-and-add operation to be performed every cycle. It was also observed that floating-point performance is in very many cases governed by fixed-point performance. For instance, most matrix operations require a significant number of address computations for loads and stores. In addition, there are many repetitious instructions to be fetched and executed. Therefore, to fully exploit the multiply-add design, the RS/6000 architecture was designed to perform all storage operations, including address computations, on the fixed-point unit, and to perform program fetch and branch execution on the branch unit, with all three units overlapped.

From these two features of RS/6000 architecture, the basic design principle was devised: Seek a system organization that offers maximum overlap of the functional units, eliminating all dead cycles, while holding to instructions which can usually be executed in a single cycle.

An example of the possible overlap to be found in this architecture can be seen in the following 2D graphics transform problem:

$$V' = A*V + b,$$

or

$$x' = a11*x + a12*y + b1,$$

$$y' = a21*x + a22*y + b2.$$

If $x$ and $y$ are considered to be an array of points, then after initialization the following code computes a new array of points $x'$ and $y'$:

```
loop: lfdu   fp0, x(i)          # load and update
                                  floating-point register
      fma    fp3, fp1, a11, b1  # floating-point multiply-
                                  add
      lfdu   fp2, y(i)          # another floating-point
                                  load and update
      fma    fp4, fp2, a22, b2  # floating-point multiply-
                                  add
      fma    fp3, fp2, a12, fp4 # floating-point multiply-
                                  add
      stfdu  fp3, x'(i)         # store and update float-
                                  ing-point register
      fma    fp4, fp1, a21, fp4 # floating-point multiply-
                                  add
      stfdu  fp4, y'(i)         # another floating-point
                                  store and update
      bc     loop               # branch and count
```

While the workings of the instructions are not important at this point, note that there are a total of nine instructions in the loop consisting of four storage instructions which can execute in the fixed-point unit, four floating-point instructions which can execute in the floating-point unit, and one branch instruction which can execute in the branch unit. All of the instructions are compound-function instructions (multiply and add, load and update, store and update, branch and count).

Considering that the multiply-add instruction performs two floating-point operations, these nine instructions result in the execution of thirteen operations. However, an overlapped implementation will only require four instruction times for each iteration of the loop. This, then, is an example of the potential of the RS/6000 POWER architecture.

## System organization

The RISC System/6000 architecture defines separate instruction and data caches. The benefits of this organization are discussed by Radin [1] and Hopkins [2]. As in the original 801, these caches are of a "store-in" design and are managed by a combination of hardware (for cache loads and cast-outs) and software (for synchronization). The instruction cache is associated with the branch unit, and the data cache is shared between the fixed-point and floating-point units for data access. The branch unit manages the instruction cache, and the fixed-point unit manages the data cache. The role of these caches in address translation is discussed in the section on the addressing model.

**Figure 1** shows the RS/6000 system organization. Note that the figure also shows how the registers are partitioned among the various units. The reasons for this partitioning will become apparent in the following sections.

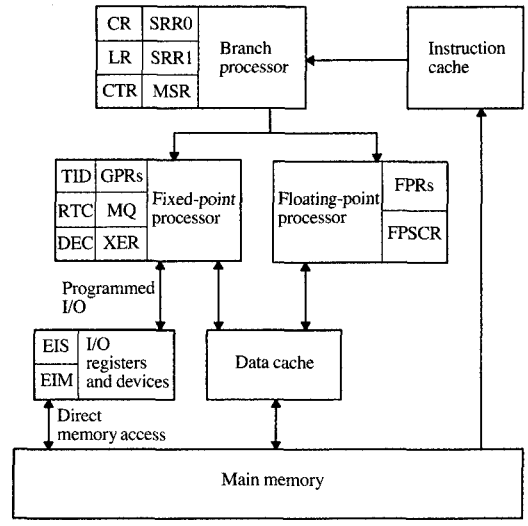


Figure 1

Logical view of RISC System/6000 architecture.

## Branch unit

Instruction fetch (including instruction address generation) and instruction address translation (including protection checking) are performed by the branch unit. All management of interrupts is handled by the branch unit. Thus, machine functions such as the appearance of sequential execution of instructions, synchronization at interruption, and switching of state are all handled by the branch unit.

The principal registers of the branch unit are the link register, the count register, the condition register, and the machine-state register. The link register is used to contain the target and/or return instruction address for subroutine linkage. The count register is used to control loop iteration. There is a branch instruction that reduces the count register by one and branches on the resulting value. The machine-state register controls system states such as enable/disable and relocate on/off. The condition register is explained later in this section.

From these registers, it is possible for the branch unit to fetch the appropriate next instruction, decode it, and either execute it, if it is a branch-unit instruction, or dispatch it to the fixed-point or floating-point units. The branch unit can continue to do this until either the queue of instructions at the other units is full or there is a data dependency on one of the local registers of the branch unit requiring data from the fixed-point or floating-point unit and the data is not yet available. The architecture assumes that the branch unit will be able to fetch at least three instructions in one cycle, one for itself and one for each of the other two units. It also assumes that the branch unit is capable of dispatching one fixed-point and one floating-point instruction every cycle. Since the branch unit is assumed to be processing the instruction stream in advance of the fixed- and floating-point units, the unnecessary complication of a "delayed" branch or branch with execute, as in the original 801, is no longer required.

The following code sequence illustrates the use of the branch-unit registers (and the overlap of the branch and fixed-point units). It is a simple implementation of the C subroutine "void bzero(addr,len);". The purpose of this code is to set to zero an area of memory. There are two parameters to the subroutine: r3 (GPR 3) contains the address of the string to be zeroed, and r4 contains the length of the string. The LR (link register) contains the return address.

```
cmpi   0, r4, 0      # test for length <= 0
mtspr  CTR, r4       # set number of bytes to zero
cal    r5, 0(r0)     # value to store = 0
bler                 # return if length <= 0
ai     r3, r3, -1    # decrement addr by 1
```

```
zloop: stbu   r5, 1(r3)     # incr r3 and zero a byte
       bc     16, 0, zloop  # dec CTR and jump if CTR
                              ! = zero
       br                   # return
```

The *cmpi* (compare immediate) instruction tests the length for zero. The *mtspr* (move to special-purpose register) instruction moves the count to the CTR (count register). The *cal* (compute address lower) instruction is an easy way to put a zero into r5. The *bler* is a pseudo-instruction for a "conditional branch on less than or equal," where the branch target is in the link register. The *ai* (add immediate) instruction decrements the starting address (because the next instruction will pre-increment it). The *stbu* (store byte with update) instruction stores one byte and updates the target address (by one in this case). Updating is discussed in the section on the fixed-point unit. The *bc* (branch count) pseudo-instruction decrements the CTR and, if not zero, goes to the branch target (in this case the previous instruction).

Note that the code is "scheduled"; that is, the *cmpi* is separated from the *bler* that uses the resulting condition, and the *mtspr* is also separated from the *bc*. Also note that these separations are maximum (for this code). The architecture-implied timing of the code in the loop is one iteration per cycle; that is, the branch unit does the *bler*, the fixed-point unit does the *stbu*, and both units are completely overlapped. Another aspect of this "scheduled" code is that the test for zero length is not done immediately; that is, the *mtspr* and the *cal* are inserted before the *bler*. The (correct) assumption here is that the normal path is for nonzero length, and that with this ordering, this path has minimum cycle time; i.e., the *bler* is overlapped and therefore does not take a cycle.

From studies of compiled code, it is clear that the proportions of branch, fixed-point, and floating-point instructions are not equal. (Branches usually constitute less than 20 percent of all instructions executed, and most programs do not have a high percentage of floating-point instructions.) It is also clear that the three instruction types are not uniformly distributed in the code. However, to achieve optimum fixed- and floating-point performance, the compiler should intermix the fixed- and floating-point instructions. Some amount of instruction queuing with wider instruction fetch and dispatch could reduce the level of this requirement on a compiler.

Branch instructions have three types of address computations: instruction address relative, address absolute, and register. The fields required for the instruction address relative and address absolute types are part of the instruction and can be executed immediately. When there is no condition to be tested, the size of this instruction field is 26 bits; that is, a $\pm 2^{25}$ relative or absolute address can be generated. For conditional branches, the relative and absolute address field is 16 bits; that is, a $\pm 2^{15}$ relative or absolute address can be generated. The last type, register, is used when the target address is not known at compile time or is larger than 26 bits. The register in this case is always a branch-unit register, typically the link register. For maximum performance, the compiler needs to move the address to the link register as early as possible, so that the target instruction may be fetched by the branch unit while the instructions between the load of the link register and the register branch are still in the instruction queues. For unconditional branches, the branch unit may be significantly ahead of the fixed-point and floating-point units with respect to which instructions are currently being executed.

For branches that depend on the condition register (conditional branches), the situation is somewhat different. The condition register is composed of eight condition fields. Two of the eight are assigned to the fixed-point and floating-point units (one to each) to contain the results of arithmetic computations. Both fixed- and floating-point arithmetic instructions contain a bit, the record bit (Rc) that indicates whether or not the corresponding condition field should be set. All eight fields can be explicitly set by fixed- or floating-point arithmetic compare instructions. In addition, there are special branch-unit instructions that either manage (move/save) the fields or perform logical operations on the fields.

The motivation for two standard condition fields should be obvious: Fixed-point and floating-point computations involving the setting of conditions are completely independent of each other and can proceed in parallel. The motivation for explicit control by the Rc bit of the setting of the standard fields is twofold: Only the conditions that are to be tested need be saved (subsequent arithmetic will not change the saved condition), and explicitly identifying an instruction which sets the condition makes it easier for the compiler to schedule this instruction as far away as possible from the testing of the condition. The motivation for the additional condition fields is the recognition that comparisons are independent operations themselves and should not have to interfere with each other by setting a common condition code. By giving comparisons their own condition field, the comparisons can be scheduled earlier, repeated comparisons can be eliminated, and invariant comparisons can be moved out of inner loops. By having the compiler target condition codes and comparisons to different fields in the condition register, the hardware can exploit simple scoreboarding algorithms to detect dependencies without significantly affecting performance.

**Figure 2(a)** illustrates the use of multiple condition fields, showing how pre-testing of the conditions enables the branch code to be no more than just the branch logic itself. **Figure 2(b)** shows the instruction stream.

On entry to this routine, r3, r4, and r5 contain the addresses of x, y, and z, respectively. The *cmp* (compare) identifies the target condition field and is a fixed-point instruction. The nonregister form of branch (*bf*—branch false pseudo-instruction) is an instruction to perform a relative branch. The *lr* (load register) is a pseudo-instruction for moving the contents of one register to another.

The setting of a condition field by an arithmetic or comparison operation and its subsequent testing represent one major requirement for coordination between the fixed- and floating-point units and the branch unit. It was recognized very early that it would be extremely difficult to have the setting operation immediately followed by the testing branch and not lose cycles. A possible implementation that probabilistically reduces this dependency, but increases hardware complexity, is for the branch unit to conditionally dispatch fixed- and floating-point instructions. This then becomes an area where an implementation is faced with the more traditional trade-off between significant hardware complexity and better performance.

Even when predicting whether a branch will be taken, the compiler can help. There are two cases of conditional branches that can be analyzed: if-then-else logic and loop closure. In general, without additional language help, it is not obvious which path of the if-then-else logic should be taken. For this case, the best guess is often the fall-through case. On the other hand, the branch-back for loop closure is almost always taken. An RS/6000 processor does not have the same information that a compiler does in analyzing the cases; however, it does have the displacement. (In general, branch addresses are obtained from an immediate field in the branch instruction.) If the displacement is negative, it is a good assumption that the branch instruction is part of a loop closure and will be taken. Otherwise, the fall-through case is the preferred path for the conditional dispatch of instructions.

Part of the cost of separate branch-unit registers is the additional instruction(s) required to move branch-register contents to or from the general-purpose registers (GPRs). Early in the development cycle, the obvious choice of using a specific GPR as the link register was considered, thereby eliminating the instructions required to save and restore an explicit link register. It was assumed that the fixed-point unit could monitor all loads of the selected GPR and ship the new value to the branch unit. While this was certainly possible, this proposed mechanism did not save any cycles because the coordination and transfer still took at least one cycle. Furthermore, the hardware to support this implied transfer would be more difficult to implement and more error prone than that required for explicit transfer instructions. Explicit register transfer instructions also expose this dependency to the compiler, which can then schedule the transfer as early as possible so that the content will be available to a branch instruction. For these reasons, the implied link register was rejected.

```
/* This procedure returns the middle of three numbers */

y2: procedure (x, y, z) returns (fixed bin(31)) reorder;
   declare
      (x, y, z) fixed bin(31),
      mid       fixed bin(31);
   if x<y then
      if y<z then
         mid = y;
      else
         if x<z then
            mid = z;
         else
            mid = x;
      else
      if y>=z then
         mid = y;
      else
         if x>=z then
            mid = z;
         else
            mid = x;
         return (mid);
end y2;
```

(a)

```
*                                  Start of Procedure
*                                  (note r3 will contain answer on return)
        l       r0, X (r3)         load X, Y, Z into r0, r3, r4
        l       r3, Y (r4)
        l       r4, Z (r5)
*                                  Precompute conditions
        cmp     cr6, r0, r3        Compare X and Y, saving result in cr6
        cmp     cr0, r0, r4        Compare X and Z, saving result in cr0
        cmp     cr1, r3, r4        Compare Y and Z, saving result in cr1
*
        bf      cr6, lt, %8        Branch to Label %8 if X ge Y
        btr     cr1, lt            Conditional return (Y lt Z)  (return Y)
        lr      r3, r4             Move Z to return register
        btr     cr0, lt            Conditional return (X lt Z)  (return Z)
        lr      r3, r0             Move X to return register
        br                         Return                       (return X)
%8:
        bfr     cr1, lt            Conditional return (return Y)
        lr      r3, r4             Move Z to return register
        bfr     cr0, lt            Conditional return (return Z)
        lr      r3, r0             Move X to return register
        br                         Return (return X)
```

(b)

**Figure 2**

Illustration of the use of multiple condition fields in determining the middle of three numbers: (a) program code; (b) instruction stream.

```
The calling routine:
        bl    .glue.foo              # branch to foo (actually glue)
        l     rTOC, 20(rDSA)         # restore my TOC

The glue routine:
.glue.foo l   r12, foo.DESCR(rTOC)   # address of descriptor for foo
        st    rTOC, 20(rDSA)         # save caller's TOC on stack
        l     r0, 0(r12)             # get code address of "callee"
        l     rTOC, 4(r12)           # get TOC address of "callee"
        mtspr CTR, r0                # set target to count reg
        bc                           # branch through count reg

The callee routine:
.foo    mfspr LR, r0                 # move LR if foo not leaf
        mfspr CR, r12                # move CR if foo alters it
        stm   r13, -144-76(r1)       # save GPRs 13--31 if foo alters them
        st    r0, 8(r1)              # save LR if foo not leaf
        st    r12, 4(r1)             # save CR if foo alters it
        stu   r1, -szdsa(r1)         # decrement stack ptr and save back
                                       chain
        .
        .
        .
        l     r0, szdsa+8(r1)        # get LR if saved
        l     r12, szdsa+4(r1)       # get CR if saved
        ai    r1, r1, szdsa          # restore stack ptr
        mtspr LR, r0                 # put back into LR if saved
        lm    r13, -144-76(r1)       # restore GPRs 13--31 if saved
        mtspr CRF, '00111000'B, r12  # restore CR2, CR3, CR4 of CR if
                                       saved
        br                           # return through LR
```

**Figure 3**

Using "glue" code in cross-object linkage.

However, one observation about program linkage did cause a change in the architecture. It was assumed that one link register was sufficient (and obviously it is). If nested routines are to be called, the link register (along with the other conventional registers) will be saved at the point of call and restored at the point of return. At the outermost ("leaf") level, no registers need be saved or restored.

One particular case did cause unnecessary cycles. As a system strategy, software has the notion of bound objects, that is, separately compiled units linked (bound) together, with all internal references resolved. Software also requires that such objects be relocatable and sharable. To accomplish sharing across bound objects, a more complicated linkage convention is required. This convention uses "glue" code, that is, code introduced to manage cross-object linkage. Instead of calling an entry in another object directly, the glue code is called first. The glue code loads the appropriate addresses and then calls the requested entry.

As originally defined, this double call makes use of the link register four times: once as the target for the appropriate glue code, once for the return address of the calling program, once as the target for the original entry called, and once for the return address of the glue code. To accomplish this, the original implementation of the glue code had to save the contents of the link register and then load the address of the called entry into the link register. On the return path, the called entry returns to the glue code, which reloads the saved address into the link register and returns to the calling routine. Other than reloading the saved address, the glue code need not be returned to, and the original calling program can be given direct control.

To accomplish this, it was necessary to make available another branch-unit register to contain a branch-target address. Because this observation was made late in the design cycle, a separate register could not be used, so the count register was chosen to perform this additional function. With this new architecture, the calling program loads the address of the appropriate glue code into the link register and branch-and-links through the link register, setting its return address. The glue code now loads the count register with the address of the called entry and branches to it. On return, the called routine branches through the link register directly to the calling program.

**Figure 3** shows an out-of-bound-object linkage sequence with these conventions: Each object maintains two pointers: one to the DSA (dynamic save area) and the other to the TOC (table of contents). The DSA is the program stack, and the TOC is the linkage segment. The link register (LR) is used to call the "glue." The glue called is specific to the actual target routine. The calling module is responsible for reloading its pointer to the TOC.

Note that for leaf procedures most of these instructions are not needed—for simple leaf procedures there is no need to save or restore registers. Even the DSA need not be updated. By convention, not all of the CR is saved, just the third, fourth, and fifth nibbles. The *mfspr* (move from special-purpose register) copies data from the identified special-purpose register to a GPR. This code does not deal with FPRs (floating-point registers).

## Fixed-point unit

Besides handling all of the normal fixed-point arithmetic instructions, the fixed-point unit does all data-address computations for both itself and the floating-point unit. In this role, the fixed-point unit schedules the movement of data between the floating-point unit and the data cache. The role of the floating-point unit is to either

supply the data going to the cache or accept the data coming from it, with the source or destination being its floating-point registers. This operation of data transfer should be viewed as not taking a floating-point unit cycle. Floating-point loads and stores are fixed-point operations and take fixed-point cycles.

The fixed-point unit is a traditional RISC, and in many respects represents only a modest change from the original 801. It has thirty-two 32-bit registers, three operand operations (RT = RA + RB), and a powerful rotate and mask facility. All instructions are 32 bits long, with a 16-bit immediate field where appropriate. The immediate field can be used as a displacement, which is relative to a register. Load and store instructions also have an address auto-increment feature. An MQ register is used for multiply and divide operations as well as extended-precision computation. Each arithmetic operation computes the three standard conditions of less than, equality, and greater than. Overflow is the fourth condition, under control of an overflow control bit. Controlling overflow is necessary when extended-precision operations are being performed. Even though the conditions are generated on each arithmetic instruction, the first condition field (back in the branch unit) is not set unless the record bit in the arithmetic instruction is set.

Other than the inclusion of full multiply and divide instructions (instead of multiply step and divide step), the most significant departure in the fixed-point unit from the original 801 is for the handling of misaligned data. The original 801 asserted that all programs and data were aligned or could be aligned. While this is true of programs, some of the data represented by various declarations in old programs is not aligned. To cover this case, the 801 had an alignment-check mode. If the mode was off, all addresses were truncated to the natural boundary of the operands. If alignment-check mode was on, all addresses were checked for conformance of the address to the natural boundary of the operand. If the address conformed, the operation proceeded. If it did not, the instruction caused an alignment-check interrupt. Software could then fix up the register, if the operation was a load, or the storage location, if it was a store, and then resume the interrupted program at the next instruction. This scheme is reasonable as long as the frequency of use of misaligned data is very low. In the case of the data found in some programs (usually old FORTRAN programs), it is not.

The RS/6000 architecture has adopted the following strategy for dealing with misaligned data. It calls for the implementation of misaligned loads and stores, including special string operations. It is relatively easy to conceive how this might work when all of the data is contained within one cache line. To implement this, the hardware can appropriately adjust the alignment on the way in or out of the cache with a simple byte multiplexor. A more difficult question is what to do with misaligned data that straddles cache-line boundaries. If one line is in the cache but the other is not, the hardware must pause in the middle of an instruction and wait for the missing cache line to cycle. Such hardware is more complicated. An even more difficult situation occurs when the misaligned data crosses a protection boundary. (While paging and protection have not yet been discussed, think of a protection boundary as one where access is allowed on one side but may or may not be allowed on the other. The other side may be "protected.") In this case, if the instruction is allowed to start, some of the misaligned data will be moved; the rest might not. In general, there is no way to tell ahead of time without "dry-running" the instruction, that is, probing the data accesses within each protection boundary. If the result of the dry run is successful, the instruction is started; if it is not, an interrupt is generated. This procedure is very costly both in hardware and in time.

To simplify the hardware requirements, RS/6000 architecture only requires hardware to handle misaligned data within a cache line for the all-non-string instructions. The architecture allows for the partial completion of an operation and the generation of an alignment-check interrupt when the data crosses a cache-line boundary. System software can then complete the instruction by fixing up the affected registers or memory locations. This definition will lead to various implementations. At a minimum, all data will be accessible within a cache line. On some machines, all data except that crossing a protection boundary will be accessible. Other machines might allow all data to be accessible (if not protected). System software must implement the alignment-check interrupt, which will execute more or less frequently depending on the machine on which it executes.

Character string operations represent a significant part of the mix of functions being performed and are rarely aligned on word boundaries. Our studies have indicated that as much as 15 percent of the cycles spent during compilation are spent moving strings. The original 801 used special instructions that shifted characters in a combination of the MQ and a general-purpose register, stored the contents of the general-purpose register, and updated (auto-incremented) the storage address. Thus, it was possible to write a subroutine that, after initial alignment, could move four bytes every two cycles (if the loop was "unrolled"). While this may have been good on a machine that had a four-byte data flow, it prevented the use of the much wider data flow possible in today's technology. These instructions also required significant overhead in setting up the inner loop. Since more than

```
.
.
.
# ------------------------------------------------------------
# Move section–R3 Target address, R4 Move length, R5 Source address.
# ------------------------------------------------------------
#
# String length here is always at least 21. Scheme is to move chunks of
# 16 bytes using LSX/STSX with R4 as the index reg and use the count
# reg as loop control. Move uses R9–R12. Remainder count in R8.
#
        lil     r9, 16          #Chunk length for move and remainder.
        cmpl    cr1, r5, r3     #Test if long fwd move might destruct.
        mtspr   XER, r9         #Set move length.
        rlinm.  r8, r4, 0, 0x0000000f
#                               #R8 = remainder,
                                CR bit 2 (eq) = "no remainder."
        mtctr   r6              #Set loop count–Number chunks to move.
        crand   27, 26, 2       #CR bit 27 = "No pad and no
                                remainder."
        blt     cr1, backward   #B If A(source) < A (target) logically.
#
forward:

# Start chunk move at first 16 bytes of target area.
        lil     r4, 0           #Beginning Index.
loopf1: lsx     r9, r4, r5      # Get next chunk of string
        stsx    r9, r4, r3      #Store it away
        ai      r4, r4, 16      #Bump index up
        bdn     loopf1          #Bump count, Br if chunk(s) left to do.
        bbtr    27              #Return if no pad and no remainder.
        bz      cr0, pad        #Go do pad if no remainder.
#
        mtxer   r8              #Set move count for remainder.
        lsx     r9, r4, r5      #Get the string.
        stsx    r9, r4, r3      #Store the string.
        bzr     cr6             #Return if pad not required.
        b       pad             #Go pad.
        .
        .
        .
```

**Figure 4**

Example of a string move, assuming no string overlap.

half the strings moved are less than eight bytes, this overhead represents a significant portion of the execution cycles for moving strings.

Given the wider data paths implied by the RS/6000 architecture, it was reasonable to define string operations. Such operations move misaligned data from storage to consecutive registers, or from consecutive registers to storage. Of course, these instructions have to deal with the problems of alignment and protection and are therefore defined to allow partial completion of the operation and complete restartability. Since their main advantage is high-speed transfer of long character strings, these instructions cannot take alignment-check interrupts. The fact that these instructions load and store misaligned data significantly reduces the overhead required to enter the inner loop. With these instructions, the architecture can achieve data-move performance equal to the maximum rate implied by the data flow without very complicated hardware and without dry-running instructions.

**Figure 4** is an excerpt from the C subroutine "void bcopy (to_addr,from_addr,len);". The purpose of this code is to move the "from" string to the "to" string. It will move the string "backward" if there is a possibility that the strings overlap. The string length can be zero. The small section of code shown here will move a small number of bytes (fewer than 16) or move bigger chunks until there are only a small number of bytes to move.

The new instructions (*blt*, *bdn*, *bbtr*, *bz*, *bzr*, and *b*) are all pseudo-instructions for the RS/6000 branch instructions. The *rlinm* instruction rotates and ANDs under mask. The period (·) following *rlinm* is the indication to return the condition code to the branch unit at the end of this instruction. The *crand* instruction logically ANDs two condition-register bits, putting the result into a designated condition-register bit. This instruction is executed on the branch unit. The *bbtr* pseudo-instruction branches on the specified condition-register bit. The *lsx* and *stsx* instructions respectively load and store strings. They start from a designated GPR and use consecutively higher-numbered GPRs. The number of bytes moved is controlled by the value in the fixed-point exception register (XER), a special-purpose fixed-point register which contains fixed-point arithmetic exceptions and control fields for the string instructions.

The null-terminated strings supported by the C language presented a particular problem in the RS/6000 architecture. The following represents the most straightforward loop for implementing *strcpy* (source,target) with r3 containing the address of the source and r4 the address of the target:

```
        ai      r3, -1          # Prepare addresses for pre-
                                  increment
        ai      r4, -1
loop:   lbzu    r5, 1(r3)       # Get byte of string and update
                                  pointer
        cmpi    cr0, r5, 0      # Compare with null
        stbu    r5, 1(r4)       # Store byte away and update
                                  pointer
        bne     cr0, loop       # Loop if not null
```

Since the compare immediate (*cmpi*) is dependent upon the load byte and zero with update (*lbzu*), the one-cycle

data cache access will be fully exposed, resulting in at least four fixed-point cycles per byte transferred. In addition, the setting of cr0 by the *cmpi* instruction is only one instruction before the branch not equal (*bne*) which depends upon it. Therefore, some implementations may not be able to completely overlap the execution of the branch instruction.

Several alternatives to this problem were examined. One proposed solution was to add a compare-bytes instruction. This instruction would compare all the bytes in a register and return the number of the first byte from the left which contained a null (or any other arbitrary character). This solution allowed for an inner loop which would load four bytes in one instruction, compare those four bytes with the compare-bytes instruction, and store four bytes if no null was found.

This alternative was rejected for two reasons: First, this particular approach cannot easily be extended to support wider data buses to the cache. This approach is basically limited to four bytes every three or four cycles. The second reason for rejecting the compare-bytes instruction was the overhead required for set-up and termination. Since the inner loop loads four bytes before checking for the terminating null character, the loop is likely to load bytes beyond the end of the string. In order to avoid spurious storage protection exceptions caused by accessing storage that the program would not normally touch if the original one-byte-at-a-time loop were executed, the new inner loop must be preceded by code which copies enough of the string to force the inner loop to be word-aligned, which guarantees that no protection boundaries will accidentally be crossed. The loop also must be followed by some "fix-up" code which stores only the bytes up to and including the null character. Character string statistics indicate that most strings are eight bytes or less, so the significant overhead before and after the inner loop significantly reduces the benefit associated with the compare-bytes instruction.

The solution selected for the RS/6000 architecture was to define a load-string-and-compare-bytes (*lscbx*) instruction. This instruction loads misaligned data into consecutive registers and compares all incoming bytes to a comparison character contained in the XER (usually a null). If a match is found, the Equal bit in cr0 will be set. The byte count of the first match is returned in the XER count field. The instruction is defined such that any storage after the first matching byte must appear to have been "untouched" by the instruction. Implementations can either "pause" at storage protection boundaries to check for the end of the string, or proceed across the boundary and cancel any protection exceptions generated after the match.

With this instruction, implementations can move null-terminated character strings at nearly the full bandwidth

```
        lil     r0,16           #Load byte count and null into XER
        mtspr   XER,r0
        lil     r0,0            # Initialize index to zero

loop:   lscbx.  r5,(r3+r0)      #Get 16 bytes of source string
        stsx    r5,1(r4)        #Store up to 16 bytes at target
        ai      r0,16           #Update index
        bne     cr0,loop        #Loop if not null
```

**Figure 5**

Example of coding for *strcpy*.

of the processor/cache interface. Also, multiple-byte comparisons are performed in parallel without having to involve the branch unit. The instruction definition also minimizes the set-up required for the inner loop so that it is efficient for short and long strings. An example of the coding of *strcpy* (source,target) with r3 containing the address of the source and r4 containing the address of the target using *lscbx* is shown in **Figure 5**. This code sequence is simple and compact enough that the compiler can generate it in-line whenever *strcpy* is encountered.

Another aspect of including string operations in the architecture is its effect on subroutine linkage. The original 801 used a sequence of stores to save state at subroutine call and a sequence of loads to restore state at subroutine return. These sequences were themselves part of the "prolog" and "epilog" code and were called at entry to and exit from every non-"leaf" routine. While the additional branch overhead of this linkage can be eliminated by the ability of the branch unit to fetch ahead, this convention would have required the link register to be saved before calling the prolog and restored in the epilog.

The sequences of stores and loads also presented a problem. Without special hardware that recognized such sequences, the maximum transfer rate to storage was one word every instruction, or four bytes every cycle. To achieve better performance without special hardware, we adopted load multiple (LM) and store multiple (STM) instructions. Since these instructions define a sequence of registers to be loaded or stored, they permit an implementation to achieve its maximum transfer rate.

LM and STM do require some of the mechanism of string operations. While the storage address of these instructions is defined to be word-aligned, the full length of the operands may cross a protection boundary. Thus, LM and STM are defined to allow a page fault during

```
      SUBROUTINE MATMPY (A, B, C, n, n1, n2, n3) C
      REAL*8 A(0:N, 0:N), B(0:N, 0:N), C(0:N, 0:N) C
      INTEGER I, J, K, N C C C

      DO 30 I = 0, N1
       DO 20 J = 0, N2
        C(I, J) = 0
        DO 10 K = 0, N3
          C(I, J) = C(I, J) + A(I, K) * B(K, J)10   CONTINUE 20   CONTINUE
30  CONTINUE

      END
      .
      .
      .


      lfd       fp0 = +matmpy(r8, 0)
      cmpi      cr1 = r30, 0
      ai        r9 = r30, 1
      ai        r5 = r5, -8
      a         r5 = r7, r5
CL.0:
      li        r29 = 0
      bt        CL.3, cr0, 1t
      lr        r6 = r3
      lr        r8 = r4
CL.2:
      stfdux    r8, c(r8, r28, 0) = fp0
      lrf       fp1 = fp0
      bt        CL.5, cr1, 1t
      lr        r11 = r7
      lr        r10 = r6
      lctr      r9
      cmp       cr6 = r0, r30
CL.4:
      lfdux     fp3, r11 = a(r11, r31, 0)
      lfdu      fp2, r10 = b(r10, 8)
      fma       fp1 = fp1, fp3, fp2
      bctf      CL.4, cr6, gt
      stfd      c(r8, 0) = fp1
CL.5:
      ai        r29 = r29, 1
      cmp       cr6 = r29, r27
      a         r6 = r12, r6
      bf        CL.2, cr6, gt
CL.3:
      c         cr6 = r7, r5
      ai        r4 = r4, 8
      ai        r7 = r7, 8
      bf        CL.0, cr6, gt
CL.1:
```

**Figure 6**

Optimized matrix multiply operation.

their operation without requiring a dry run. Similarly, they are permitted to perform a partial operation and yet be restartable from the beginning.

By including LM and STM in the RS/6000 architecture, it became reasonable to generate in-line prolog and epilog code. See Figure 3 for use of the LM and STM instructions.

## Floating-point unit

The floating-point unit has thirty-two 64-bit floating-point registers. There is also a floating-point status and control register that contains exception indicators, default exception masks, and floating-point conditions. The floating-point unit supports ANSI/IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic" [3]. The architecture defines the basic arithmetic set of add, subtract, multiply, and divide operations. It also defines the usual floating-point register move, negate, and absolute-value operations. It provides for round-to-single and floating-point compare. Like the original 801, the RS/6000 architecture requires software support for many of the extended functions in order to fully conform to the IEEE standard.

RISC System/6000 floating-point processing is organized for double-precision computations. This means that data held in the floating-point registers is always represented in double-precision format. Thus, when single-precision data is loaded, it is expanded into double-precision format. Similarly, when a floating-point register is stored as single-precision data, it is first converted to single-precision format and then stored. To achieve the same results as a processor which supports an IEEE conforming single-precision mode requires that a round-to-single instruction be performed after each floating-point arithmetic operation; otherwise the results will be in extended single-precision format.

One interesting extension to RS/6000 floating-point operation is the definition of a floating-point register lock. When set, this lock prevents the execution of floating-point instructions. An attempt to execute any of these instructions causes an interrupt identifying the attempted use. The floating-point register lock and the associated interrupt can be used by software to manage the saving and restoring of floating-point registers during process switching or at system call so that the FPRs need only be saved and restored for those tasks which are actually executing floating-point code.

As was previously noted, the single most important feature of the floating-point unit is its ability to do a multiply-add instruction every cycle. Four multiply-add instructions were added, to cover the four possible signs of the operations (not the operands). An interesting side effect of these new instructions is the new precision gained from their use. It is now possible to obtain a more precise result by doing a multiply-add than by doing a multiply followed by an add, since the full precision of the intermediate result is used during the multiply-add but lost during the two-instruction sequence.

To exploit the level of computational power offered by the multiply-add operations, data must be given to and taken from the floating-point unit by a means at least as fast as the multiply-add execution time. This has been accomplished by using the fixed-point unit as the address generator and data mover for the floating-point unit and by including a large number of floating-point registers, which act as a buffer. These facilities, when applied to standard math library functions, have led to some remarkable results.

The paper by P. W. Markstein [4] in this issue illustrates the benefits of this approach. For instance, studies of various matrix operations have shown that the RS/6000 processor achieves performance usually associated only with vector machines. Additionally, if a particular matrix operation algorithm is sensitive to the geometry and replacement algorithm of the data cache, significantly improved performance is obtainable. **Figure 6** shows a matrix multiply operation that achieves near-optimum performance.

**Figure 7** illustrates the case of inner product.

The *lfd* instruction loads a floating-point register. The *lfdu* instruction loads a floating-point register and updates the address. The *stfd* instruction stores a floating-point register. The CL.0 loop in this code contains four instructions or five operations. There are two fixed-point instructions, one floating-point instruction, and one branch instruction, so an implementation of this architecture could perform this loop in two instruction times.

Another very important aspect of fully exploiting floating-point performance is the method of presentation of floating-point exceptions and the precision in identifying the instructions that cause floating-point exceptions. Exceptions are a natural and perhaps an expected consequence of floating-point operations, and most can be handled by default rules. (Default exception handling is defined by the IEEE standard.) These default rules can be managed completely in hardware and require no program intervention after initialization.

The IEEE default rules do not always provide the desired result, however. Since the standard allows for program fix-up after an exception, the architecture problem then is how to define a mechanism to permit program fix-up. The most straightforward approach is to specify that a floating-point interrupt at the failing instruction will occur whenever there is a floating-point exception that is not defaulted. The hardware implication of this is that all instructions after a floating-point instruction must be conditional until it is known that no exceptions are possible on that instruction. Some floating-point instructions take many cycles, and exceptions may not be known until the last cycle of the instruction. Therefore, most implementations would serialize on floating-point instructions—if not the first, then the second; if not all, then some. The inclusion of a floating-point interrupt would sacrifice much of the potential floating-point performance.

An alternative strategy is not to report an interrupt at all, but simply to set a bit indicating that a floating-point exception has occurred. It is then up to a program to test for floating-point exceptions. Different compiler strategies can be used as to where it is appropriate to test for these exceptions. Since the definition of the exception also

```
C                    LOOP3–INNER PRODUCT
      SUBROUTINE LOOP3(N)
      REAL*8     X(1000), Z(1000), Q
      IJK = 1
      DO 3 K = IJK, N
      Q = Q+Z(K)*X(K)
3     CONTINUE
      END
```

(a)

```
*                                    Subroutine for Inner Product
      l       r3,n(r3,0)             Load N from argument pointer
      l       r5,.&loop3$(r2,0)      Addressability local data (Q)
      cmpi    cr0,r3,1               Test N, result to cr0
      bt      CL.1,cr0,lt            See if N<1, exit if so
      ai      r4,r5,8                set address of start of X
      lfd     fp0,q(r5,0)            fetch Q into fp0
      mtspr   CTR,r3                 set ctr = N
CL.0:
      lfd     fp2,z(r4,8008)         load Z(K) into fp2
      lfdu    fp1,x(r4,8)            load X(K) into fp1, bump address
                                     of X
      fma     fp0,fp0,fp2,fp1        Q = Q+Z(K)*X(K)
      bctf    CL.0,cr0,gt            Branch on Count
      stfd    fp0,q(r5,0)            Store Q
CL.1:
      li      r3,0                   Set return value
      br                             Return
```

(b)

**Figure 7**

Example of an inner-product loop for a matrix operation: (a) program code; (b) instruction stream.

includes the setting of summary information, it is possible to test at the end of a program, at the end of a subprogram, or at the end of a statement where a floating-point operation was used. This level of precision can be controlled by linker/compiler option. None of these tell exactly where the exception occurred; they simply identify that it occurred. In most cases, this information is sufficient.

However, if the exact failing instruction must be known, there are two possible strategies. One can insert a test for the exception after each floating-point instruction, or one can tag each queued and/or executing floating-point instruction with its address. Inserting code to test for every possible exception is yet another mode for the compiler to manage, necessitates recompilation, and can significantly expand execution time. Address tagging of "active" floating-point instructions identifies the failing instruction exactly. However, it does require that the

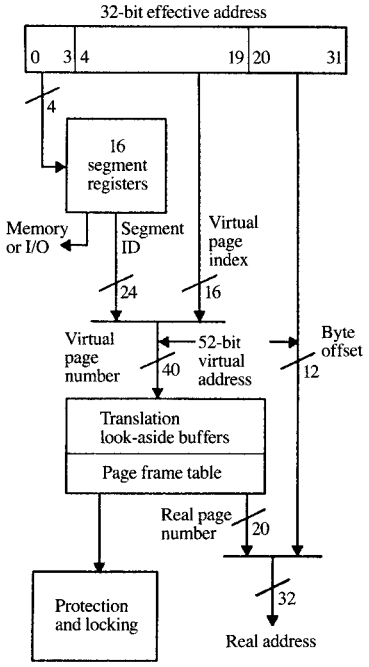


**Figure 8**

Virtual address generation and translation.

implementation keep track of the address tags. Moreover, it is not synchronous; that is, if an exception occurs, the location of the failing instruction is reported, but not before the program has gone beyond that point. Fix-up may still be possible, but in general this method only permits localization of the failing instruction. Consider the case of the inner-product loop described in Figure 7. This loop consists of two floating-point loads, one floating-point multiply-add, and one branch. The "active" floating-point instructions will all be instances of the same multiply-add instruction. If an exception occurs, what is known is the address of the instruction, not the iteration number. The benefit of this approach is speed; floating-point performance is not limited by exception recognition. The drawback, as outlined above, is the precision with which the fault is determined.

RISC System/6000 architecture adopted a two-part strategy. The principal approach would be test-code insertion, with the compilers able to insert such code at the statement or (sub)program level. The linker also supports the enabling of test code at program exit, ensuring the ability to report a floating-point exception if it occurs anywhere within the program.

To avoid recompilation in order to identify the failing operation exactly, the architecture also adopted a synchronize mode, in which an interrupt can be generated, identifying the failing instruction by running the machine with one floating-point instruction dispatched at a time. This technique has the same weakness as code insertion; that is, floating-point performance is greatly reduced. However, it may not be as bad as code insertion, because the synchronization can be managed by hardware rather than by extra code inserted by the software. It is expected that the mode will only be used by certain programs, and then only to debug their algorithms.

## Storage model

The storage model of RS/6000 architecture is an extension of the 801 (and the RT System) model. All computed addresses (called effective addresses) are 32 bits wide. Excluding the special case of programmed I/O, the model has two addressing modes, real and virtual. If the machine is in real mode, the effective address equals the real address, and the full 32 bits are used to access real storage. If the machine is in virtual mode, a translation step must be performed. The overall approach to translation can be found in [5, 6]. The differences between RISC System/6000 and the RT System are these: The segment identifier (SID) in the RS/6000 is 24 bits wide, resulting in a 52-bit virtual address. The real page size is 4096 bytes, and a full 32-bit real address is supported. For special segment processing, the transaction identifier (TID) had been extended to 16 bits, and each lock bit spans 128 bytes (or 32 lock bits per 4-Kbyte page). **Figure 8** graphically illustrates the data flow of this translation process.

RISC/6000 architecture has extended the special segment locking to grant locks to transactions without generating lock faults. When the system software can determine that there is only one active transaction in a particular page, the translation mechanism can be enabled to automatically grant access and turn on the appropriate lock bit for each new lock line access. Of course, if another transaction wishes to access the page, software must then capture the lock information and return to the standard method of lock processing. In addition, under very special circumstances the RS/6000 storage model permits read access to data in special segments without checking either TIDs or lock bits. This access is not recorded. The situation most appropriate for this is during journal writing, where the system, not the transaction, is outputting the data.

RISC System/6000 architecture has an interesting strategy for managing the caches. In the effective address resolution path, it is important that the TLB lookup be done in parallel with the cache-directory lookup. (If it is

done sequentially, which is the most straightforward way, an extra cycle is added to all loads, and perhaps to stores as well.) Doing the lookups in parallel requires either that the hash functions be identical or that the bits used be invariant with respect to translation. Keeping the hash functions identical means that functions which use multiple parts of the address (such as the SID and some of the virtual offset) cannot be used, since looking up a real address may result in a very different congruence class from that which results from looking up the virtual address currently mapped at that real address. This problem, sometimes called the cache-aliasing problem, forces many implementations to serialize the lookups, or to guess for the correct class. The penalty for an incorrect guess is another cycle in the load path. The alternate approach (choosing invariant bits for cache lookup) leads either to limited cache sizes or to requirements on virtual-equals-real (V = R) software mapping.

The RS/6000 architecture has chosen to restrict any page that can be addressed both virtual and real to be identical in the cache hash-function bits in both spaces or for the operating system to perform the appropriate cache-line flushes to maintain consistency. Almost all of the RS/6000 software runs in virtual mode. This restriction affects only the low levels of interrupt handling and startup code.

As was previously noted, the data cache uses a "store-in," not a "write-through," algorithm. Additionally, there is no coordination of stores in the data cache with the possible contents of the instruction cache. (This is the same strategy that was used on the original 801.) A consequence of this strategy is that software must manage the synchronization between caches. Similarly, there is no coordination between the caches and I/O. Data moved by I/O comes from or goes to main storage directly. Thus, software must also manage this synchronization.

The RISC System/6000 architecture has added to this complexity. Because of the branch unit and its ability to fetch ahead, it is difficult to identify precisely which instructions have been prefetched and/or dispatched but not yet executed by the fixed-point or floating-point units. With software-managed caches, however, this must be known or controlled with certainty. Consider the case of program modification, e.g., a loader-generated piece of code such as glue code. After the instructions to generate this code are executed, the data cache line(s) containing these instructions must be stored back, and any line in the instruction cache that might contain old copies of the unmodified code must be flushed. These store-back and flush operations must be synchronized. RS/6000 architecture (and the 801) provide cache-synchronizing primitives to accomplish this function. In the former, however, it is conceivable that the branch unit will be looking far enough ahead that the unmodified instructions will have been prefetched and/or dispatched before the synchronizing primitives are executed. The architecture fix for this is to require a flush of the prefetch mechanism (and the instruction buffers) whenever a cache-synchronize instruction is executed. It is very easy (and safe) to overspecify when synchronization of this form is required. Consider the cases of various state changes associated with interrupts. Some, such as I/O, require synchronization because a new translation mode must be used. Others, such as supervisor calls, do not. Because of the performance consequences of this form of synchronization, RS/6000 architecture has chosen to minimize the number of instructions requiring synchronization. Careful attention is required by the hardware implementation and system software to neither miss places where synchronization is required nor include places where it is not required, so as to optimize between correct execution and maximum performance.

A good example of this type of situation is the changing of segment registers. On task switch, most of the segment registers must be saved and reloaded with the new task's contents. This is coded as many consecutive move-to-segment-register (*mtsr*) instructions. Since this code is also executing out of a segment, the "safest" definition is to synchronize the processor on every *mtsr* just in case it changes the code segment translation. For task switch, this would be an unnecessary burden, since the code segment of the task-switch routine is not being changed. Therefore, the architecture requires that software include an explicit synchronize instruction after an *mtsr* which changes code translation.

## Conclusions

The RISC System/6000 architecture has made significant improvements in implementing the ideas of the original 801. As is always the case, most of the changes are in the details learned over many years by writing compilers and system code for 801-style machines. The most important advance in the RISC System/6000 implementation has been to integrate floating-point arithmetic into the architecture and concentrate on the parallel aspects of execution. This has led to an architecture that offers remarkable floating-point performance, comparable to that offered by many vector processors. The architecture is rich enough to encourage the implementation of the multiple ALUs and much wider data paths that future technologies will probably make possible.

The real distinction of the RISC System/6000 approach over traditional RISCs, then, is that the architecture has partitioned the registers by function, allowing simultaneous execution with limited coordination. By exposing the areas of coordination

required, this architecture permits a compiler to generate highly optimized code which manages functional interaction to achieve parallelism that is close to the theoretical maximum for a given implementation.


## Acknowledgments

John Cocke is the inventor of most of the key concepts behind this architecture. Andrew Heller's vision made the product a reality. Also, Marc Auslander, Albert Chang, Greg Grohoski, Martin Hopkins, Peter and Vicky Markstein, Mark Mergen, Robert Montoye, and Daniel Prener, who were all part of the original AMERICA architecture development at the Thomas J. Watson Research Center, as well as Bill Hay from IBM Toronto and Jack and John O'Quin from IBM Austin, are thanked for their contributions to the RISC System/6000 implementation of the IBM POWER architecture.


## References


1. G. Radin, "The 801 Minicomputer," *SIGARCH Computer Architecture News* **10,** 39–47 (March 1982).
2. M. E. Hopkins, "A Perspective on the 801/Reduced Instruction Set Computer," *IBM Syst. J.* **26,** 107–121 (1987).
3. "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Standard No. 754*, American National Standards Institute, Washington, DC, 1985.
4. P. W. Markstein, "Computation of Elementary Functions on the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** 111–119 (1990, this issue).
5. Albert Chang and Mark F. Mergen, "801 Storage: Architecture and Programming," *ACM Trans. Comput. Syst.* **6,** 28–50 (February 1988).
6. R. O. Simpson and P. D. Hester, "The IBM RT PC ROMP Processor and Memory Management Unit Architecture," *IBM Syst. J.* **26,** 346–360 (1987).



*Received February 28, 1989; accepted for publication January 17, 1990*



**R. R. Oehler** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Oehler is manager of System Structures in the Advanced RISC Systems Department at the Thomas J. Watson Research Center. After seven years at the U.S. National Security Agency, where he worked on operating system design, and one year at RCA working on computer architecture, he joined the IBM Research Division in 1970. He has since had several assignments in computer architecture and operating system design, including manager of I/O Architecture in the IBM Data Systems Division from 1972 to 1974 and manager of Architecture for the System Products Division from 1981 to 1985. From 1986 to 1987 he was the lead architect for the RISC System/6000 development program. Mr. Oehler holds a B.A. in mathematics from St. John's University.

**R. D. Groves** *IBM Advanced Workstations Division, 11400 Burnet Road, Austin, Texas 78758.* Mr. Groves is a Senior Engineer currently on leave in the Management of Technology program at the MIT Sloan School of Management. He joined IBM in 1979 in Manassas, Virginia, and transferred to IBM Austin in 1982, joining the design team for the RT System. He was the logic design manager for the instruction cache and fixed-point units for the RS/6000 processor. Most recently, he was the manager responsible for the architecture of all the RISC-based processors in the Advanced Workstations Division. Mr. Groves received B.S. degrees in electrical engineering and business administration from Kansas State University in 1978 and 1979, respectively. He is a member of the IEEE, Tau Beta Pi, Eta Kappa Nu, Blue Key, and Phi Kappa Phi.