IBM RISC System/6000: Architecture and Performance

The IBM RISC System/6000 realizes the idea of a superscalar microprocessor. The architecture of this processor has its instruction set specifically designed for a superscalar machine containing three independent units—branch, fixed-point, and floating-point. Besides the emphasis on superscalar, the design also emphasizes high-performance floating-point operations.

Richard R. Oehler

Michael W. Blasgen

IBM Research Division

Two features distinguish the RISC System/6000 from other reduced instruction-set computing microprocessors: superscalar organization and high-performance, floating-point capability.

We designed the architecture to be realized as a superscalar machine. By *superscalar* we mean a machine that, like a simple scalar machine, executes one nonvector instruction stream but achieves high performance by dividing the work among independent functional units that operate in parallel. Thus, with an execution unit in the pipeline that permits one-cycle execution, a scalar machine can execute three instructions in three cycles. In comparison, a superscalar machine with three independent units can execute the same three instructions in one cycle if they are independent (ignoring pipeline latencies). Superscalar systems of this sort can improve cycles per instruction by a substantial amount.

To achieve this superscalar characteristic, we have separated the RISC System/6000 (called system hereafter) processor into three major execution units: fixed-point, floating-point, and branch. Each of these units operates in parallel, with the branch unit in overall control and responsible for ensuring the integrity of program execution.

We have seen other machines with such superscalar organizations, the most notable being several early IBM machines like the Assembly Control System project described by John Cocke in his Turing lecture[1] and the 360/91.[2] More recently, developers of the Intel 860 and Apollo DN10000 have used a superscalar approach, although the approaches used in these machines are somewhat different. For example, the 860 has an explicit mode for dual-instruction execution, which is a type of very large instruction word format. The system does not define a mode, and all instructions run in superscalar mode.

The realization of the system architecture in a full superscalar organization does require substantial hardware—more than could fit on one chip in the available complementary metal-oxide semiconductor (CMOS) technology.

For example, we assigned registers to functional units, and most instructions execute entirely within a unit. When the compiler requires interaction (for example, when the fixed-point unit generates condition codes that the branch unit uses), it schedules the code explicitly for maximum performance. Thus, the system exposes its latent parallelism to the compiler and expects the compiler to deal with the parallelism. And, like the original 801 (see box), the system sometimes even requires software intervention in the management of data and instruction cache cross-consistency to ensure properly functioning programs.

(The machine does not, however, require scheduling of instructions to function correctly—

0272-1732/91/0600-0014$01.00 © 1991 IEEE

the pipeline is fully interlocked, and the superscalar aspects of the machine are automated. Scheduling is only required to achieve high performance.)

The second distinguishing feature of the system is the emphasis on floating-point performance. Very early in the project we conceived of an advanced floating-point unit design that would permit a 64-bit multiply-and-add operation to be completed every cycle. To ensure access to storage did not limit floating-point performance, we designed the machine to complete a load or store operation every cycle as well, in parallel with the floating-point operations.

The fixed-point unit performs all storage operations including address computations, which permits full overlap with the floating-point unit. To further improve path length, we carefully reviewed the 801 instruction set and made many changes. In particular, we incorporated new compound instructions like multiply-add, branch-on-count, and update-load.

These two features of the system—superscalar organization and high-performance floating point—led to the basic design principles. The principles are to offer maximum overlap of the three functional units, avoid dead cycles, and define instructions that can (for the most part) be completed at a rate of one per cycle.

Branch cycle

An independent functional unit handles branches and manipulates condition registers, instruction fetch, and instruction caching. The branch unit has its own register set to contain condition codes, target addresses, and loop counts. It executes the instructions that manipulate the condition registers.

The branch unit fetches instructions from memory (caching them in the instruction cache), executes the branches, and dispatches the remainder to the floating- and fixed-point units. The branch unit conducts these operations until either the queue of instructions at the other units is full or a conditional branch is encountered with an unsatisfied dependency on one of the branch unit's local registers.

When the branch unit sees an unconditional branch, it simply executes it. As a result, unconditional branch execution overlaps with other computations, producing zero-time unconditional branches. Since procedure call linkage is important, the branch unit has a link register to contain the target and/or return instruction address for subroutine linkage. The branch instructions contain a field that controls whether the branch instruction sets the link register; this can be used to save return addresses.

To handle conditional branches, the branch unit uses the condition register and the count register.

Using the count register speeds up looping. A defined compound instruction, branch-and-count, decrements the count register by one and branches on the resulting value. In most cases, fixed- and floating-point operations can overlap the branch-and-count function, achieving zero-time, loop-closing branches. Other conditional branches refer to a condition code, placed in the condition register by previously executed fixed- and floating-point operations. If early scheduling of the condition code setting is possible, conditional

RISC development

Under the intellectual guidance of John Cocke,[3] researchers at the IBM T.J. Watson Research Center invented and constructed the first RISC machine—the 801—in the 1970s. Since then the same team developed several variations. Newer machines extended the original ideas to include virtual addressing[4] and enhanced I/O.

Starting in 1985, most of the original 801 team, including Cocke, embarked on a new effort, the America project, to reconsider the issues of machine architecture with the benefit of 801 experience and its follow-ons.

(Cocke suggested the name, America. He recited the story of Queen Victoria watching the completion of the first America's Cup race in England in the 19th Century. The Queen asked: "Who is in first?" The reply came, "America." "And who is in second?" she asked. Answered the Queen's aide, "There is no second.")

The team performed studies on floating-point organization and performance, the effectiveness of the architecture as a compiler target, and, most importantly, the possibilities of instruction-level parallelism through the use of dense VLSI capabilities.

Three components determine processor performance: the number of instructions it takes to execute an algorithm, the cycles per instruction, and the cycle time. Reducing both the number of instructions and cycles per instruction was the objective of the design. The proposed approach used is a "superscalar" organization, a term coined by Tilak Agerwala and Cocke to describe the instruction-level parallelism envisioned in the new machine.[5] This activity led to a new-generation RISC. The research division initiated a minor effort to design an America machine. This work progressed very rapidly, and it soon became clear that such a system was feasible and offered important advantages. Rather than develop in research what could have only been a prototype, IBM's Research Division team approached the company's development laboratories.

In 1986, a group at IBM's Austin, Texas, laboratory accepted the America ideas. This decision eventually led to the RISC System/6000.[6]

branches can also be fully overlapped, again achieving zero-time branches.

More specifically, the 32-bit condition register divides into eight condition fields. Two of the eight condition fields are assigned to fixed- and floating-point units (one each) to contain the codes associated with arithmetic computations.

Both fixed- and floating-point arithmetic instructions contain a record bit that indicates whether the corresponding condition field should be set. Any of the eight fields can be explicitly set with compare instructions. The two permanently assigned fields permit the fixed- and floating-point units to set the conditions in parallel. The motivation for the record bit is to permit the compiler to move condition code-setting instructions to an earlier point, because subsequent instructions with the record bit off will not change the computed condition code.

Branch instructions have three types of address computations: program counter relative, absolute, and register. In the relative and absolute types, the addressing information is located in the instruction and can execute immediately. We use branches based on the link register when the target address is not known at compiling time or is too large to be contained in the instruction. For subroutine calls the compiler must move the subroutine address or the return address to the link register as early as possible to overlap the branch.

Part of the cost of separate branch registers is the additional instruction(s) required to move the contents of the branch unit registers to or from the general-purpose registers. Early in the development cycle, we considered eliminating the extra instructions by using a specific general-purpose register as the link register. In this proposal, the fixed-point unit would monitor all loads of that general-purpose register and ship the new value to the branch unit. We rejected this proposal because it would not have saved any time, since the coordination and transfer would still take at least one cycle. Furthermore, the compiler would not gain any advantage. In fact, by making the branch scheduling explicit, the compiler can use its general scheduling algorithms to deal with this more globally.

Fixed-point unit

Besides handling all of the normal integer and string instructions, the fixed-point unit accomplishes 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 floating-point unit's role is to either supply or accept the data between the data cache and its floating-point registers. Floating-point loads and stores are fixed-point operations and take fixed-point cycles.

The RISC fixed-point unit represents only evolutionary changes as compared with the early 32-bit 801. It has thirty-two 32-bit general-purpose registers, three operand operations (RT = RA + RB) and a powerful rotate and mask facility. All instructions are 32 bits long. Load and store instructions also have an update mode for use in automatically incrementing the address. An MQ register processes multiply and divide as well as extended-precision computation. Each arithmetic operation computes the three standard conditions of less than, equal, and greater than. The generation of condition codes occurs on each arithmetic instruction, but the condition field (back in the branch unit) is not set unless the record bit in the arithmetic instruction is set.

The most significant departure from the original 801 is the inclusion of more complex instructions, reflecting the increased very large-scale integration (VLSI) capability available in the 1990s. The system supports integer multiply and divide, floating-point arithmetic, and string operations. Studies justify the string operations by indicating that moving strings takes as much as 15 percent of the cycles during key integer tasks. Given the wide data paths envisioned in the system's processor, we need to define string operations. Such operations move unaligned data from storage to consecutive registers, or from consecutive registers to storage.

The most significant departure from the original 801: The inclusion of more complex instructions, reflecting the increased VLSI capability.

To achieve high performance on subroutine linkage, system designers adopted load multiple and store multiple instructions. Since these instructions define a sequence of registers to be loaded or stored, they permit an implementation to achieve its maximum transfer rate between the registers and storage. The operating system also uses this approach during the process switch. Both the load/store multiple and the string instructions are multicycle operations.

Floating-point unit

The floating-point unit's architecture has thirty-two 64-bit floating-point registers. (The hardware has 38 registers.) The floating-point status and control register contains exception indicators, default exception masks, and floating-point conditions. The floating-point unit supports the ANSI/IEEE Standard 754-1985.[7] The architecture defines the basic arithmetic set of add, subtract, multiply, and divide operations. It also

defines the usual floating-point register move, and negate and absolute value operations. It provides for round to single and floating compare. To fully conform to the IEEE standard, the system requires software support for some of the extended functions required by the standard such as square root.

Since the state associated with thirty-two 64-bit registers is substantial, the system includes instructions to set and release a floating-point register lock. When set, this lock prevents the execution of floating-point instructions. An attempt to execute any of these instructions will cause an interruption identifying the attempted use. This interruption permits the software to save the floating-point registers during a process switch for only those processes that are actually accomplishing floating-point work.

An important feature of the floating-point unit is this: It can execute a multiply-add instruction on every cycle. The multiply-add instruction has four operands (two multiply, one add, and a fourth register for the result). For high precision, the multiply-add develops the full 106-bit precision in the multiply, conducts the add at a full 162-bit precision, and then scales the result to a 53-bit mantissa. Thus multiply-add is somewhat more powerful than the two-instruction sequence. We believe this extension to the IEEE precision rules is consistent with the intent of the standard, and in fact this precision is required to get accurate results in certain computations.

To exploit the level of computational power offered by the multiply-add operations, we must give and take data from the floating-point unit by means at least as fast as these instructions can execute. We accomplish this exploitation by using the fixed-point unit as the address generator and the data mover for the floating-point unit. A large number of floating-point registers act as a buffer. These facilities, when applied to standard math library functions, have lead to some remarkable results. For instance, studies of various matrix operations have shown that the system achieves performance usually associated with vector machines. Additionally, if the matrix operations understand the geometry of the data cache and have significant reuse (as do the IBM Engineering and Scientific Subroutine Library scientific subroutines that are available for the machine), users obtain even better performance.

Cache management

The architecture specifies a copy-back cache. In such a cache, results stored to memory by the processor are placed in cache, and then at some later time a cache line is copied back to main memory. Thus main memory and cache do not reflect the same state, which is not a problem for the processor alone, since it always views memory through the cache. However, it is a problem for I/O, since data moved by I/O comes from or goes to main storage directly, bypassing the data cache. Additionally, the instruction cache does not automatically coordinate stores in the data cache. This is a problem for programs, like the loader, that create other programs from data.

Table 1. Models of the RISC System/6000.

Models	Processor	
	Clock cycles (ns)	Data cache (Kbytes)
320, 520	50	32
320H	40	32
530, 730, 930	40	64
540	33	64
550, 950	24	64

This approach to cache memory is the same as that used in the original 801. A consequence of this approach is that software must manage the synchronization between the cache and main memory (for I/O) and between the caches (for program management).

The realization

A multichip CMOS processor made the system architecture a reality. Three processing chips—fixed, floating, and branch—directly correspond to the three functions. The fixed-point chip contains general-purpose registers along with the integer arithmetic unit and the string operations. The floating-point chip holds floating-point registers along with the high-performance, floating-point, multiply-add array.

The branch chip contains the 8-Kbyte instruction cache. A 16-Kbyte, custom static RAM, data cache chip is replicated two or four times in the processor. The set rounds out with a memory control chip and an I/O control chip.

The different models of the system use different clock speeds and data cache sizes (see Table 1). In each cycle, the branch unit can fetch four instructions from the instruction cache. Then the branch unit executes two instructions (provided one is an instruction that manipulates condition registers and the other is a branch instruction). Finally, it sends out two instructions to the arithmetic units. Both instructions are sent to both arithmetic units, and each arithmetic unit discards the instructions that are not appropriate to it. Thus the ideal instruction stream consists of one CR-logic instruction, one branch instruction, one fixed-point instruction, and one floating-point instruction in each group of four consecutive instructions (in any order within each group). This mix can execute at a rate of four instructions per cycle (ignoring pipeline delays and multicycle instructions).

In practice that mix is seldom achieved. There are almost never that many CR-logic instructions. In addition, fixed-point unit operations dominate many programs. But many floating-point-intensive jobs can achieve three instructions

continued on p. 56

continued from p. 17

per cycle. By considering the floating-point add as two operations, the processor can achieve four operations per cycle.

The arithmetic units can each hold six instructions in buffers, and this makes the machine somewhat forgiving if the sequence departs from the ideal alternatives. Furthermore, the floating-point unit has an additional stage for renaming and a second stage of execution. These stages act as two additional buffers.

In more detail, performance aspects include:

- All fixed-point unit instructions take one cycle, except for multiply (three), divide (19), and load- and store-multiple (variable number of cycles) operations.
- Fixed-point load followed by use incurs a one-cycle delay; an independent operation should be inserted between the load and its use. A floating-point load followed by use normally incurs no delay.
- Support of unaligned operations is available, but this support should be avoided for maximum performance.
- The floating-point unit has a two-cycle latency, with a two-stage pipeline. It retires one floating-point operation (including multiply-add) per cycle, unless the operation depends on the immediately preceding operation. Hardware fully supports basic floating-point operations, including cases that involve denormalized numbers, infinities, and so on.
- Unconditional branches take zero time in most cases.
- Conditional branches take zero time if there are three independent instructions between the compare and the branch. In the absence of intervening instructions, a compare followed by a branch-taken incurs a three-cycle delay, and a compare followed by a branch-not-taken is zero (because of conditional dispatching of the next instruction).

Warren[8] provides more details on performance prediction.

While the architecture specifies 32 floating-point registers, the implementation actually has 38, as does the dynamic register-renaming function. Renaming prevents delays by avoiding apparent conflicts. For example, consider the sequence:

```
Reg 3 = Reg 2 + Reg 1
Store Register 3
Load Register 3 with new value
```

The load following the store is presumably the next iteration of the loop. The register-renaming hardware will note that register 3 is not the same physical register as the one named in the preceding store. The hardware will, for purposes of the load, rename some other physical register as register 3. This step permits the load to proceed without interlocking with the preceding store. In general, any instruction that redefines a register's contents in the presence of its preceding use is a candidate for renaming.

We envision other implementations of the architecture besides the present system realization.

Performance

The benefits of the system's superscalar organization is shown in the SPEC benchmarks.[9] Table 2 shows the SPEC ratios and Specmarks for various systems. (IBM announced these results, which reflect new versions of the Fortran and C compilers, in November 1990.)

Different effects contribute to the fast performance of the top machines. The system owes its performance to a superscalar organization implemented in CMOS. The fast Mips Computer Systems machine has a cycle time of 17 ns achieved through the use of bipolar technology.

Benchmark details

At IBM Research, we wrote an instruction-trace simulator that traces the execution of a program, producing the sequence of instructions executed. We used this tool to capture the execution of the individual SPEC programs. We then examined the traces, looking at the frequency and sequences of various instructions.

We chose three of the SPEC benchmarks (LI, Matrix300, and TOMCATV)—two extremes and a midpoint. LI does not show the advantages of the independent units; at the other extreme TOMCATV achieves a very high degree of instruction overlap. Matrix300 is an interesting case because it offers a significant opportunity for compiler optimization.

LI benchmark. Table 3 (on page 58) summarizes the results of tracing the LI benchmark. Of the three benchmarks mentioned in this article, the LI program achieves the least instruction-level parallelism of any of the SPEC benchmarks. On the LI program, the system gains nothing from having an independent floating-point unit. Its independent branch unit sometimes helps and sometimes hurts. (In Table 2, a scalar processor like the Mips R3000—included in the 40-ns DECstation 5000 Model 200—matches the performance of the superscalar, 40-ns RISC System/6000 Model 530 on the LI benchmark.)

The principal function in LI is the searching of a linked list for a match. Since the average depth of search is very shallow, it is very difficult to schedule this code. Figure 1 on page 59 details this search.

Since this search includes no floating-point code, the only overlap possible is between the branch and fixed-point units. To achieve overlap with the fixed-point and branch units, the compiler should schedule code to obtain separation be-

Table 2. SPEC benchmarks, arranged according to decreasing Specmark.

System	SPEC ratios										
	Gcc	Espresso	Spice 2g6	Doduc	Nasa7	LI	Eqntott	Matrix300	Fpppp	TOMCATV	Spec-mark
IBM Model 550*	30.2	34.7	47.7	45.6	77.9	33.8	39.9	78.3	89.8	132.6	54.3
Mips RC6280	47.5	43.8	38.0	39.3	43.9	45.1	38.6	51.9	51.8	42.5	44.0
IBM Model 540*	22.4	25.0	34.7	32.8	54.8	24.1	28.5	49.1	66.7	95.8	38.7
IBM Model 530*	17.4	20.8	28.8	27.4	45.6	20.1	23.7	41.9	55.3	79.8	32.0
Stardent 3010	18.1	20.8**	14.2**	19.9	64.9	18.5**	18.7**	109.6	30.1	61.3	29.4**
Alacron AL860 accelerator	15.1**	22.3**	18.1	18.8	57.6	21.7**	18.9**	28.1	25.7	43.8	24.7**
IBM Model 520*	13.4	16.3	20.4	21.2	33.3	15.8	18.6	34.2	43.2	59.1	24.6
Intel Star 860	13.5**	20.5**	14.7	15.6	45.2	17.9**	18.0**	21.6	21.2	36.3	20.8**
Solbourne 5E/901	23.9	19.7	18.5	14.1	21.5	21.7	22.0	25.9	22.5	16.5	20.3
Silicon Graphics 4D/320 S	21.5	23.6	15.3	19.8	18.2	25.5	20.2	13.3	24.7	16.5	19.5
Sun Sparc-server 490	21.1	16.6	16.5	17.2	23.7	22.0	19.6	24.6	19.0	15.6	19.4
HP Apollo Series 10000	13.6	13.4	11.7	23.9	26.7	11.7	11.4	22.0	38.9	31.2	18.6
DECstation 5000 Model 200	17.3	18.5	13.7	18.2	22.6	21.8	18.4	17.0	22.0	17.3	18.5
Mips RC3260	19.0	18.9	13.8	17.2	18.7	23.8	18.4	14.0	23.4	17.7	18.3
Mips M/2000	19.3	18.9	13.9	17.3	18.7	24.0	18.5	14.0	23.3	17.7	18.3
Mips RC3230 Server	18.4	18.1	14.3	16.7	20.9	23.1	18.8	15.5	15.9	18.5	17.9

* RISC System/6000 **Portability changes made to source code to execute benchmark

tween the loads and the subsequent use (including compare) and between the compares and the branches. As can be seen in the "Load to use distance" (Table 3), an instruction does not occur between the load and the use of the loaded data in 53 percent of the cases. The current implementation will lose one cycle in this situation.

As can be seen in the "Condition to branch distance" (Table 3), an instruction does not occur between the test and the branch in 61 percent of the cases; of these, 41 percent are taken. One instruction occurs in 21 percent of the cases, and of these 33 percent are taken. Two instructions occur in 10 per cent of the cases, with 17 percent taken. Finally, in 3 percent of the cases, three instructions occur, with 10 percent taken. This condition creates many delays in the pipe, since up to three cycles can be used on taken branches. This penalty reduces every intervening instruction by one cycle. Thus the LI benchmark does not exploit the superscalar features of the machine. Various improvements in compiler technology enhanced scheduling of the branch.[10]

LI is the only benchmark that has a moderate level of subroutine linkage. As can be seen from the "No. of registers loaded" data (Table 3), the compiler successfully reduces the number of registers that need to be saved/restored.

Matrix300 benchmark. Table 4 on page 59 summarizes the results of tracing the Matrix300 benchmark. Just one loop completely dominates the Matrix300 program. This program performs well on the system, but the data cache is limited. Figure 2 shows the subroutine that contains that loop.

Significant improvements in the Matrix300 numbers have been made since their initial presentation at the August 1990 Hot Chips Conference.[11]

We can directly attribute these improvements to the changes

Table 3. LI SPEC benchmark summary.*

Instruction frequencies and branches

Types	Percent
Instruction frequencies	
Branch	22.8
Load	25.4
Store	15.5
Compare	13.2
Add/Sub	7.1
Logical	6.2
Rotate/Shift	1.5
Move	4.7
Floating	0.0
Branches	
With links (local calls)	2.2
With links (X-module calls)	0.3
Unconditional	38.3 of total
Conditional	61.7 of total
Taken	34.8
Not taken	65.2

Moving between GPRs and branch unit SPRs

Direction	Percent
From (SPR to GPR)	2.2
To (GPR to SPR)	2.1

Load and link register conditions

Load to use distance		Link register to branch	
Distance	Percent	Distance	Percent
0	53.1	0	6.1
1	19.7	1	34.6
2	5.0	2	50.1
3	2.8	3	5.3
4+	19.4	4+	3.0

LM and STM (linkage)

No. of registers loaded by LM	Percent	No. of registers stored by STM	Percent
2	34.7	2	34.9
3	23.5	3	24.2
4	3.2	4	3.2
5	4.8	5	4.7
6	5.7	6	5.6
7	16.2	7	15.8
8	11.5	8	11.2
9+	0.4	9+	0.4

Condition to branch distance

No.	Count	Taken	Not taken
0	60.6	41.0	59.0
1	20.8	32.7	67.3
2	10.0	17.1	82.9
3	3.4	9.5	90.5
4+	1.5	22.7	77.3

Basic block lengths**

No.	Percent
1	14.2
2	9.8
3	24.6
4	16.2
5	9.1
6	9.1
7+	17.0

* Percentages are based on a trace of 300 million instructions. Neither string operations nor CTR registers were used.
** Average of 4.4 percent

in the Fortran compiler. This compiler modifies the main (and only significant) loop in two ways. First, it reorders the computation removing a floating-point load from the loop. This reordering reduces the basic block from six instructions to five. Since this load was not overlapped, its removal leads to a significant reduction in time. Second, and perhaps more important, the compiler changed the loop's array reference from a second-dimension variation to a first-dimension variation (and rewrote the equation). The change significantly improved data cache reuse, since the way the compiled program now references storage is sequential. Referencing the data by the second dimension results in good data cache reuse only if all the data fits in the cache. This is not the case with Matrix300, where the matrix occupies 720 Kbytes.

The new compiler reduces the running time for Matrix300 from 170.9 to 107.9 seconds as measured on a Model 530. This reduction emphasizes the importance of compiler technology in RISC performance. Significant performance gains

```
/* xlygetvalue - get the value of a symbol (no instance variables)
   */
NODE *xlygetvalue(sym)
  NODE *sym;
{
    register NODE *fp,*ep;

    /* check the environment list */
    for (fp = xlenv;  fp; fp = cdr(fp))
     for (ep = car(fp); ep; ep = cdr(ep))
      if (sym == car(car(ep)))
       return (cdr(car(ep)));

  /* return the global value */
  return (getvalue(sym));
}
```

In assembler:

```
loop:
   l      r11,8(r11)    # fixed load (displacement format)
   cmpi   cr0,r11,0     # compare with null (0)
   beq    cr0,eol.202   # branch if end of list
   l      r12,4(r11)
   l      r0,4(r12)     # two levels of indirection
   cmpl   cr1,r31,r0    # compare for match
   bne    cr1,loop      # branch if no match
   ...
eol.202:
```

Figure 1. Linked list search in LI SPEC benchmark.

```
      SUBROUTINE SAXPY(N, A, X, INCX, Y, INCY)
      IMPLICIT REAL*8(A-H,O-Z),INTEGER*4(I-N)
      DIMENSION X(INCX,N), Y(INCY,N)
      IF (N.LE.0) RETURN
      DO 10 I=1,N
          Y(1,I) = Y(1,I) + A*X(1,I)
   10 CONTINUE
      RETURN
      END
```

The assembler code for this loop is as follows:

```
loop:
   lfdx      fr0,r11,r9      # load float (Y)
   lfdux     fr1,r12,r10     # load float with update (X)
   fma       fr0,fr0,fr2,fr1 # floating multiply add
   stfdux    fr0,r11,r9      # store float with update (Y)
   bdn       loop            # branch-and-count
```

Figure 2. The Matrix300 loop.

come by exposing many of the machine resources to a compiler. As compilers continue to improve, so will the performance numbers. In fact, we know of techniques involving strip mining (block for cache), jamming, and unrolling that will provide substantial improvements in programs like Matrix300. These techniques have already been used in carefully coded Fortran programs like IBM's Engineering and Scientific Subroutine Package. This approach is known to be amenable to automation in a compiler.

In May this year, IBM announced improvements to the Fortran compilers based on the previously mentioned techniques. Matrix300 execution time on a Model 530 dropped to 10.4 seconds. This is 435 times faster than the reference of VAX 11/780.

Table 4. Matrix300 SPEC benchmark.*

Instruction frequencies and branches

Types	Percent
Instruction frequencies	
Branch	20.0
Load	39.5
Store	19.9
Multiply-add (float)	19.7
Branches	
Unconditional	0.4 of total
Conditional	99.6 of total
Taken	99.0
Not taken	1.0

Condition to branch distance

No.	Count	Taken	Not taken
1	0.1	63.3	36.7
2	99.1	0.4	99.6

Basic block lengths**

No.	Percent
2	0.3
3	0.2
4	0.3
5	98.4
16	0.8

* Percentages are based on a trace of 300 million instructions. No movement between GPRs and branch unit SPRs occurs. The CTR register is used significantly. Almost no branches and links result.

** Average of 5.0 percent

The assembler code in Figure 1 shows that two loads and one store in the loop dominate the Matrix300 code. It appears that a stall occurs between the load of floating-point Register 1 and the floating-point multiply-add using this register, and that loop would help eliminate this stall by producing increased opportunities for scheduling. This is not the case, however, because of the register renaming. Only one stall (at the beginning of the loop) takes place because the fixed- and floating-point units operate in parallel, and because the floating-point unit renames registers. The floating-point unit postpones the store and continues the loads of the next iteration. The data loaded is available when the floating-point unit starts the multiply-add. Both of these characteristics (postponing stores and the one-cycle slip between the fixed- and floating-point units) are typical dynamic execution time traits of the system as it approaches peak performance.

Table 5. TOMCATV SPEC benchmark.*

Instruction frequencies and branches

Type	Percent
Instruction frequencies	
Branch	6.1
Load	33.1
Store	9.6
Compare	1.7
Add/sub (float)	15.0
Multiply (float)	10.7
Multiply-add (float)	18.4
Divide	0.8
Branches	
Unconditional	0.3 of total
Conditional	99.7 of total
Taken	99.3
Not taken	0.7

Condition to branch distance

No.	Count	Taken	Not taken
0	14.1	98.7	1.3
2	0.1	2.3	97.7
30+	85.8	99.9	0.1

Basic block lengths**

No.	Percent	No.	Percent
1	14.0	9	13.9
2	0.1	10	0.1
3	0.2	12	1.1
4	0.1	13	13.8
5	14.1	14	0.1
6	14.0	17	13.8
8	0.1	18+	14.6

* Percentages based on a trace of 740 million instructions. Almost all loads/stores and compares are floating-point units. Movement is minimal between GPRs and branch unit SPRs, and the CTR register is used a little. Almost no branches and links result.

** Average of 16.5 percent

TOMCATV benchmark. Table 5 summarizes the results of tracing the TOMCATV benchmark. The TOMCATV program is the type of program in which the RS/6000 performs best. The loops are reasonably long; there is good use of both fixed- and floating-point instructions, which can be almost completely overlapped. The compiler can utilize the branch-and-count instruction to close loops, thereby assuring no lost cycles. While there is no single "hot spot" in the code, the loop shown in Figure 3.

A mixture of floating- and fixed-point operations (the floating-point loads and stores, the fixed-point stores, and the add immediate) occurs as well as a balance between them. A branch-on-count (not shown) controls this loop. We use multiple condition fields of the condition register. The only performance penalty in the loop is the partially scheduled branch true on cr0. Overall, the compiler and machine together achieve remarkable performance on TOMCATV because this program has both vector and scalar operations, a good mixture of fixed- and floating-point operations, and its basic blocks are large and therefore allow ample instruction scheduling.

THE RISC SYSTEM/6000 REPRESENTS A significant advance over the original 801 ideas: We integrated floating-point features into the architecture and concentrated on the parallel aspects of execution. But many of the improvements are in the details learned over many years by writing compiler and system code for 801-style machines. Overall, this development has led to an architecture that offers remarkable floating-point performance—comparable to that

```
C
C     DETERMINE MAXIMUM VALUES OF RESIDUALS
C
      DO 270 J = 1,M
      DO 270 I = I1P,I2M
      IF(ABS(RX(I,J)).LT.ABS(RXM)) GOTO 262
      RXM = RX(I,J)
      IRXM = I
      JRXM = J
  262 IF(ABS(RY(I,J)).LT.ABS(RYM)) GOTO 270
      RYM = RY(I,J)
      IRYM = I
      JRYM = J
  270 CONTINUE
```

The assembler code for some of this loop is as follows:

```
loop:
   fabs     fr10,fr24      # absolute value
   lfdu     fr8,8(r7)      # load rx
   fabs     fr13,fr23
   lfdu     fr11,8(r8)     # load ry
   fabs     fr9,fr8
   ai       r23,r4,1       # add immediate
   fabs     fr12,fr11
   fcmpu    cr0,fr9,fr10   # floating compare
   fcmpu    cr6,fr12,fr13
   blt      cr0,br1        # branch if less than
   fmr      fr24,fr8       # floating move register
   st       r4,120(r30)    # store irxm
   stfd     fr8,128(r30)   # store rxm
   st       r12,124(r30)   # store jrxm
br1:
```

Figure 3. The TOMCATV loop.

offered by many vector processors—as well as performance on integer tasks equal to the best of RISC processors. The first implementation of the system architecture is a high-performance, multichip processor. As technology evolves, we will shrink the current machine to one chip. A future multichip realization can use multiple fixed- and floating-point units to improve cycles per instruction even further.

Unlike earlier RISCs, the RISC System/6000 architecture anticipates a superscalar organization by partitioning the registers by function. The result allows instruction-level parallelism while requiring only limited coordination. By exposing the areas of coordination required, the system permits a compiler to generate highly optimized code that achieves, in some cases, close to the theoretical maximum parallelism possible from this machine.

Acknowledgments

The authors wish to acknowledge the many people who contributed to the system. Thanks to John Cocke for the seminal ideas and many subsequent contributions. Marty Hopkins, Dan Prener, Greg Grohoski, and Randy Groves contributed to the instruction set. Hank Warren, Peter Markstein, and Arvin Shepherd—together with the compiler team in Toronto and Haifa, Israel—produced the compiler. Special thanks to Ju-Ho Tang, who developed the instruction tracer.

References

1. J. Cocke, "The Search for Performance in Scientific Processors," *Comm. ACM*, Vol. 31, No. 3, Mar. 1988, pp. 250-253.
2. D.W. Anderson et al., "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM J. Research and Development*, Vol. 11, No. 1, Jan. 1967, pp. 8-24.
3. G. Radin, "The 801 Minicomputer," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, ACM Sigapl, New York, N.Y., 1982, pp. 39-47.
4. A. Chang and M. Mergen, "801 Storage: Architecture and Programming," *ACM Trans. Computers*, Feb. 1988, pp. 28-50.
5. T. Agerwala, "How Fast Can a Single Instruction Counter Machine Execute?" *Stanford Computer Forum Distinguished Lecture Videotape Series*, Computer Science Colloquium, Stanford University, Stanford, Calif., June 1983.
6. "Special Issue: RISC System/6000," *IBM J. Research and Development*, Vol. 34, No. 1, Jan. 1990.
7. *ANSI/IEEE Standard 754-1985, Standard for Binary Floating-Point Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1985.
8. H. S. Warren, Jr., "Predicting Execution Time on the IBM RISC System/6000," White Paper, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., Mar. 1991 (preliminary version).
9. *SPEC Newsletter*, System Performance Evaluation Cooperative, c/o Waterside Associates, 39510 Paseo Padre Parkway, Suite 350, Fremont, CA 94538.
10. M. Hopkins, "Compiling for the RISC System/6000 Branch Unit," *Proc. Hot Chips Symp.*, IEEE CS Press, 1990.
11. R. Oehler, "RISC System/6000 Architecture and Performance," *Proc. Hot Chips Symp.*, IEEE CS Press, 1990.

Richard R. Oehler is manager of systems architecture in advanced RISC systems at the IBM T.J. Watson Research Center, Yorktown Heights, N.Y. He has worked on computer architecture and operating systems design, managing the 801 architecture and efforts in simulators, tools, and operating systems. He was also lead architect for the RISC System/6000 for IBM's Advanced Workstation Division in Austin, Tex.

Oehler has a BA in mathematics from St. John's University, New York.

Michael W. Blasgen is director of advanced RISC systems at the same research center. He and his group conduct research in RISC architecture and software; he was responsible for many key contributions to the RISC System/6000.

Blasgen received a BS from Harvey Mudd College, an MSEE from the California Institute of Technology, and a PhD from the University of California, Berkeley. He is a senior member of the IEEE and a member of ACM.

Address questions regarding this article to Michael W. Blasgen, IBM T.J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate numbers on the Reader Service Card.

Low 156 Medium 157 High 158