The IBM RT PC ROMP processor and memory management unit architecture

by R. O. Simpson
P. D. Hester

The ROMP processor is the microprocessor used in the IBM RT PC. It is a 32-bit processor with an associated memory management unit implemented on two chips. ROMP is derived from the pioneering RISC project, the 801 Minicomputer at IBM Research. This paper describes some of the trade-offs which were made to turn the research project into a product. It gives an introduction to the architecture of ROMP, including the addressing model supported by ROMP's memory management unit. Some of the unique features of the programming model are explained, with high-level language coding examples which show how they can be exploited. ROMP's architecture is extensible, and the fact that almost all programming for the RT PC has been in high-level languages means that the RT PC hardware architecture can be extended as needed to meet future requirements while preserving the investment in existing software.

At the center of the IBM RT PC™ are the IBM-designed Reduced Instruction Set Computer (RISC) processor and its memory management unit. The processor is called ROMP, an acronym for Research/OPD Micro Processor. The name tells something of the origin of the processor.

The 801 Minicomputer project at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York, had defined an architecture for and built a prototype of a simple but very high-performance computer.[1,2] At about the same time (the late 1970s) the IBM Office Products Division (OPD) in Austin, Texas, was searching for a new microprocessor to be used in advanced office equipment. The ROMP project began as a joint effort between OPD and the Research Division, with the goal of adapting the architectural concepts of the 801 to an actual product.

ROMP's 801 heritage shows clearly in its architecture, and in fact the same highly optimizing compiler (PL.8)[3-6] is used for both. However, ROMP had a different set of design goals than the 801. While the 801 was an experiment in RISC architecture whose main goal was to demonstrate that a machine could be built which sustained a rate of one instruction per cycle, ROMP was to be part of a product and thus had constraints (primarily cost) that did not apply to the 801. These constraints strongly affected ROMP's design.

In this paper we first give some details of the implementation of the ROMP processor and its Memory Management Unit (MMU). The parallels between the 801 and ROMP are shown, and the differences between them are explained. ROMP's programming model is described and some examples are given of the use of its features.

Design goals

The RT PC ROMP processor was designed to

- Provide an architected address and data width of 32 bits
- Provide an efficient target for an optimizing compiler
- Support virtual memory

- Provide system integrity through separate user and supervisor states
- Provide improved error detection and reporting facilities

The first requirement dictated an architecture providing both 32-bit address and data quantities. As a result, it was decided that all registers and computations would support 32-bit quantities. However, the architecture provides for specific support of 8-bit and 16-bit quantities as well; individual 8-bit bytes and 16-bit halfwords can be loaded and stored and can be manipulated within the 32-bit registers.

The ROMP processor architecture was defined with the assumption that most software would be developed in a high-level language. A joint study between OPD and the IBM Research Division was conducted to evaluate the PL.8 optimizing compiler and the architectural requirements to take advantage of the compiler optimization techniques. The study indicated the need for a large number (16 or 32) of 32-bit general-purpose registers, and an instruction set closely matched to the compiler intermediate language.

During the architecture definition, it became clear that systems using processors of this class must provide virtual memory. ROMP saves sufficient machine state when a page fault occurs to identify the faulting instruction and address and to re-execute the load or store operation once the fault has been resolved. This virtual memory support is common on mainframes and some minicomputers, but had not appeared in a microprocessor prior to the design of the ROMP.

The need to provide protection of user programs and isolation of control program functions resulted in the definition of separate user and supervisor states. Only instructions which cannot be used to affect system integrity are valid in user state. Instructions associated with control program functions are valid in supervisor state only.

Certain requirements and facilities are provided for error detection and reporting, including parity checking on all external buses, bus time-out detection, and nonmaskable hardware error detection interrupts.

Cost constraints

The prototype 801 had been built from low-density but very high-speed circuitry. A VLSI version of the 801 in the technology available at the outset would have required many chips, would have dissipated power in excess of requirements, and would have exceeded the cost targets for a small system to be used in an office. ROMP's design was driven by the need to minimize the number of parts (VLSI chips).

Two chips. Existing technology did not allow functions as complex as the ROMP and its MMU to be combined into a single chip, so one chip was used for the processor and one for the MMU. The split is about even; the two chips are of comparable complexity (the MMU is somewhat larger than the ROMP).

ROMP's 2-byte instructions reduce the bandwidth required for instruction fetching.

High performance with inexpensive memory. The 801 had exceptionally high performance: 15.1 MIPS at a cycle time of 63 nanoseconds. However, much of its performance depended on its two caches, which could deliver an instruction word and a data word on each CPU cycle. Since such caches were prohibitively costly for small systems, pipelining techniques normally found in larger machines were adapted to the ROMP so that useful work may be done during the (comparatively) long time needed for memory operations. The techniques include asynchronous prefetching and partial decoding of instructions, a packet-switched channel between the ROMP and the MMU, execution of instructions beyond a "load" until the loaded data are actually needed, and delayed branches which overlap the execution of another instruction with the fetching of the branch target. In addition, the fact that many of ROMP's instructions are only 2 bytes long reduces the bandwidth required for instruction fetching; this bandwidth reduction is necessary because without the 801's two independent caches, the MMU can only supply one word per cycle to ROMP.

RAM size. All of the 801's instructions were 4 bytes long. This simplified instruction fetching and decod-

ing and allowed enough room to name three registers with 5-bit numbers in most instructions. Code density was considered more important for ROMP, which

The 2-byte instructions predominate in instruction mixes.

was to have a much smaller main memory than the high-performance 801. ROMP has both 2-byte and 4-byte instructions, with the 2-byte instructions predominating in both the static and dynamic instruction mixes.

Adapting the 801

ROMP's similarities and differences. The ROMP programming model and instruction set are derived from the 801 processor for which the PL.8 compiler was originally designed. With the implementation of the short instruction format, it is clear that it is not possible to provide three 5-bit register fields in a 16-bit instruction. To allow space for an op-code, the instructions were changed from three-address to two-address and the register field width reduced from 5 bits to 4. Thus ROMP has 16 general registers, rather than the 801's 32. Reducing the size of the register file also freed silicon area on the ROMP chip for implementing the rest of the CPU.

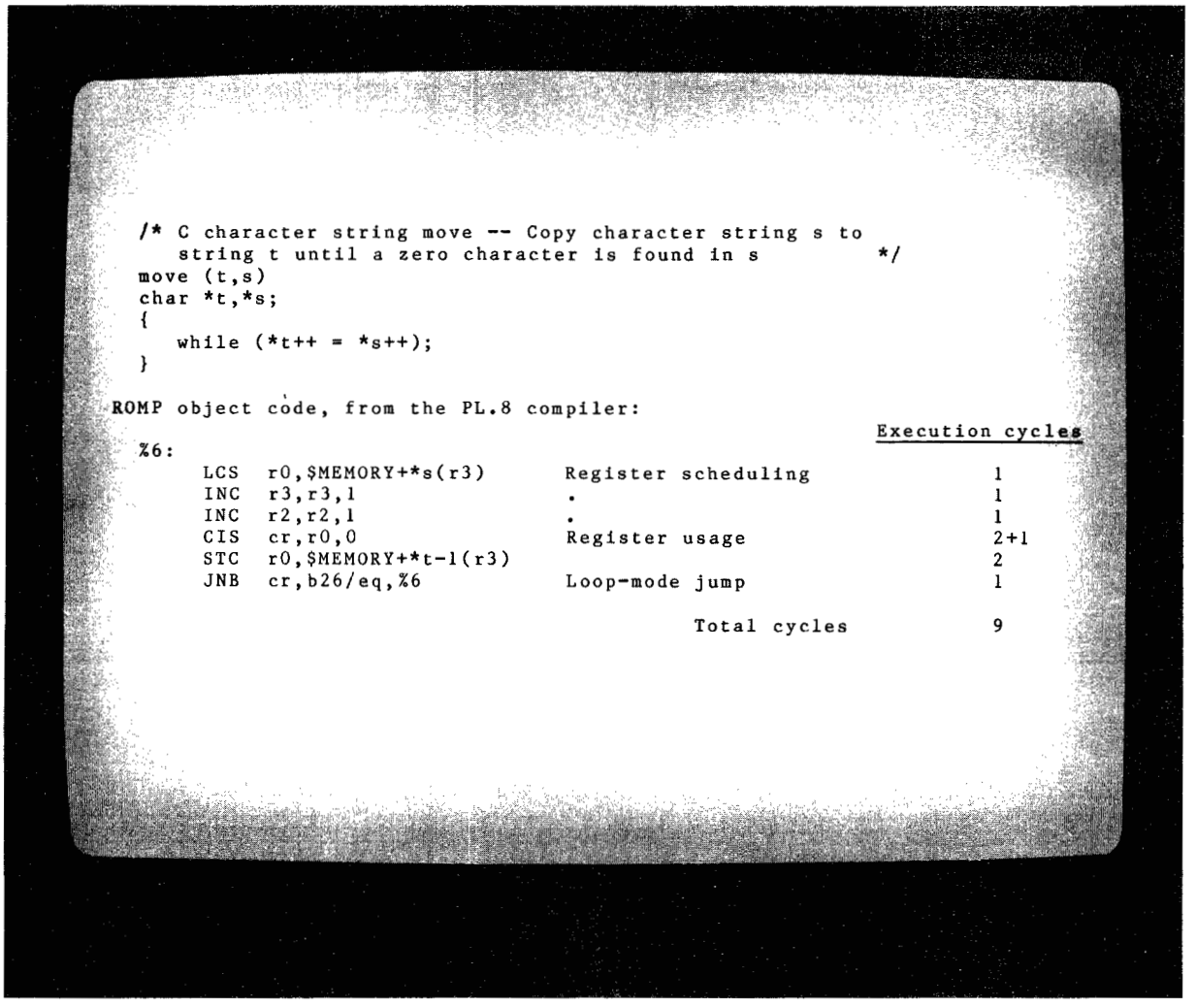
None of the 801's instruction set philosophy was changed, however. All storage accesses are still through "load" and "store" instructions, and all computation is done on operands in the general registers. That the ROMP instruction set is a good target for a compiler is demonstrated by the fact that the PL.8 compiler generates ROMP object code that is generally smaller than 801 object code for the same program.

The elimination of the 801's caches means that ROMP has a longer latency on memory accesses. The compiler "pipelines" the "load" operations by separating them from the use of the loaded data as far as possible. If several instructions can be placed between a load and the use of the data, the storage access is done in parallel with useful work in the CPU, as shown in Figure 1. The short loop illustrated is executed entirely from the ROMP instruction pre-fetch buffer, so the loop-closing jump instruction takes only one cycle rather than five. The compiler schedules the load operation two instructions prior to the use of the data loaded. Five total cycles are required for the load data to become available, and the CIS (compare) must wait two cycles for the load to complete. However, this has reduced the effective load time from five to three cycles. The combination of loop-mode execution from the pre-fetch buffer and load scheduling has reduced the total number of execution cycles for this loop from 15 to 9, a 40 percent performance improvement. This loop executes at approximately 6.7 MIPS in an RT with an Advanced Processor Card, 100 nanoseconds cycle time. This pipelining was also done on the 801, but the compiler tries to move the load and use instructions farther apart on ROMP. Note that the pipelining is possible because the storage access (load) and computation instructions are separate. If an instruction such as "add storage to register" were implemented, there would be no opportunity to fetch the operand from storage in advance of its use.

ROMP does not have an instruction cache such as the 801 had. Rather, instructions are fetched ahead a word at a time into a 16-byte instruction queue (called a pre-fetch buffer on ROMP). This queue is filled asynchronously as instructions are executed; the processor does not normally have to wait for an instruction to be returned from memory before beginning execution except for branches taken. Two features are provided which utilize the time that would otherwise be wasted waiting for a branch target to be fetched.

1. 801-style delayed branches (called branch-with-execute) allow ROMP to get a head start on fetching the branch target. The instruction which physically follows the branch in memory is executed regardless of the outcome of the conditional branch test, as if it had been *before* the branch. While this instruction (the *subject* instruction) is being executed, the branch target is being fetched. The compiler is often able to find an instruction which can be moved from just before to just after the branch in this way; such instructions execute "for free," as shown in Figure 2. The final store operation of this loop places the computed value of **y** into array element **x[i]**. The execution of the STS (store) instruction is completely overlapped with the fetching of the branch target, the first

Figure 1 A typical memory-to-memory move in C programming language



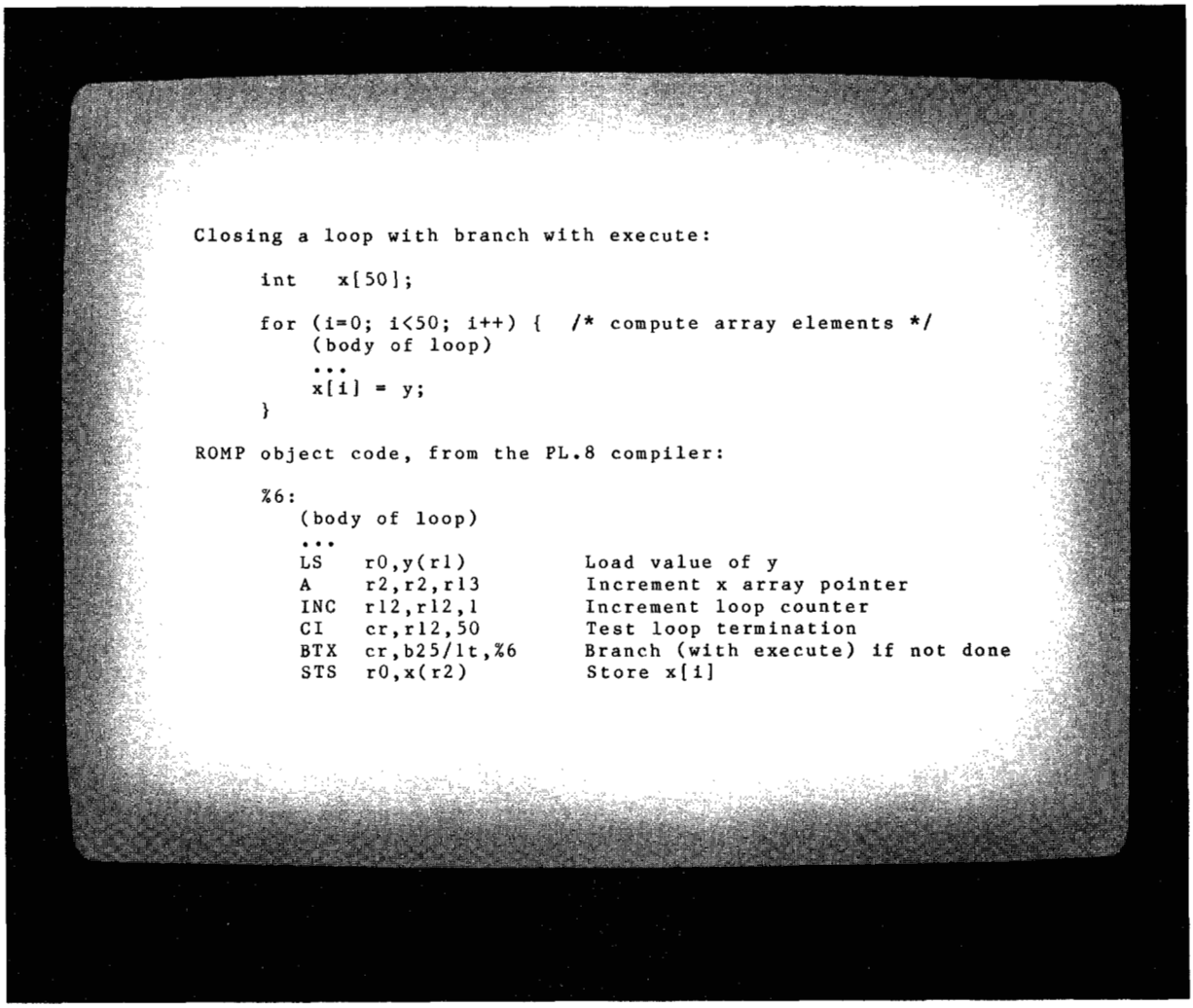
```
/* C character string move -- Copy character string s to
   string t until a zero character is found in s          */
move (t,s)
char *t,*s;
{
   while (*t++ = *s++);
}

ROMP object code, from the PL.8 compiler:

                                                    Execution cycles
%6:
     LCS   r0,$MEMORY+*s(r3)      Register scheduling          1
     INC   r3,r3,1                .                            1
     INC   r2,r2,1                .                            1
     CIS   cr,r0,0                Register usage               2+1
     STC   r0,$MEMORY+*t-1(r3)                                 2
     JNB   cr,b26/eq,%6           Loop-mode jump               1

                                  Total cycles                 9
```

instruction of the loop body. Thus, the store takes zero cycles. This code fragment also shows an example of reduction in strength: Rather than multiply the array index **i** by 4 (or even shift it left 2), an auxiliary variable is used as a pointer into the array and is bumped by 4 with a one-cycle ADD instruction each time through the loop.

2. Loops within the pre-fetch buffer are recognized by ROMP, and subsequent iterations of the loop are not re-fetched from memory. The branch which closes such a loop is thus reduced to one cycle and the entire memory bandwidth becomes available for data traffic. Real application programs have been observed in which the compiler has generated these "tight" inner loops, and such programs approach the theoretical maximum rate of one instruction per cycle (10 MIPS at a 100-ns cycle time). System subroutines such as character string move are hand-coded to take advantage of high-speed looping within the pre-fetch buffer.

Although most ROMP instructions execute in only one cycle, additional cycles are taken when it is necessary to wait for data to be returned from memory for loads and branches. As a result, ROMP takes about 2.3 cycles on the average for each instruction. At the cycle time of 100 nanoseconds used in the RT PC, ROMP runs at about 4.3 MIPS.

Figure 2 Closing a loop with branch and execute



```
Closing a loop with branch with execute:

     int   x[50];

     for (i=0; i<50; i++) {  /* compute array elements */
         (body of loop)
         ...
         x[i] = y;
     }

ROMP object code, from the PL.8 compiler:

     %6:
        (body of loop)
        ...
        LS    r0,y(r1)          Load value of y
        A     r2,r2,r13         Increment x array pointer
        INC   r12,r12,1         Increment loop counter
        CI    cr,r12,50         Test loop termination
        BTX   cr,b25/lt,%6      Branch (with execute) if not done
        STS   r0,x(r2)          Store x[i]
```

The ROMP programming model

Here we describe the major features of the registers, instruction set, and addressing model of ROMP. More detail can be found in the references[7,8] and in the reference manuals for the RT PC.[9,10]

General-Purpose Registers. ROMP has sixteen 32-bit General-Purpose Registers (GPRs), as shown in Figure 3. The registers can be used for computation or as base registers, as with System/370. Register 0 has special meaning when used as a base register: The value 0 is used as the base rather than the contents of register 0. Thus, absolute addressing is provided by a convention on the base register name rather than by a separate addressing mode.

Some shift instructions consider the registers to be "paired" (e.g., 0 and 1, 14 and 15), with input from one register and output to the other register of the pair (an implicit operand). This deviation from true RISC style was made in order to be able to specify nondestructive shift instructions in the short (16-bit) format.

Other registers. The other registers which can be seen by the programmer are the System Control Registers (SCRs) shown in Figure 4. Most of these deal with hardware control of timers and interrupts or record exception conditions (page faults, program checks). Two are directly usable in application programs.

- The *Condition Status* (CS, Control Register 15) is the equivalent of the System/370 Condition Code. Results of comparisons and arithmetic operations are recorded here; the conditional branch instructions specify a bit in the Condition Status which determines whether the branch is taken. One bit is permanently zero, providing for unconditional branches (and no-ops). Another, the "test bit," can be loaded with any selected bit in any general register and then used as a branch condition.
- The *Multiplier/Quotient* (MQ, Control Register 10) register holds half of the 64-bit product or dividend for the Multiply Step and Divide Step instructions. It is an implicit operand of those instructions, and is the result of a trade-off between architectural elegance and hardware efficiency: As an implicit operand, the MQ need not be named in the instruction (thus the Multiply Step and Divide Step instructions can be 16 bits long) and it is implemented as a high-speed register separate from the general register file.

Figure 3 ROMP General-Purpose Registers

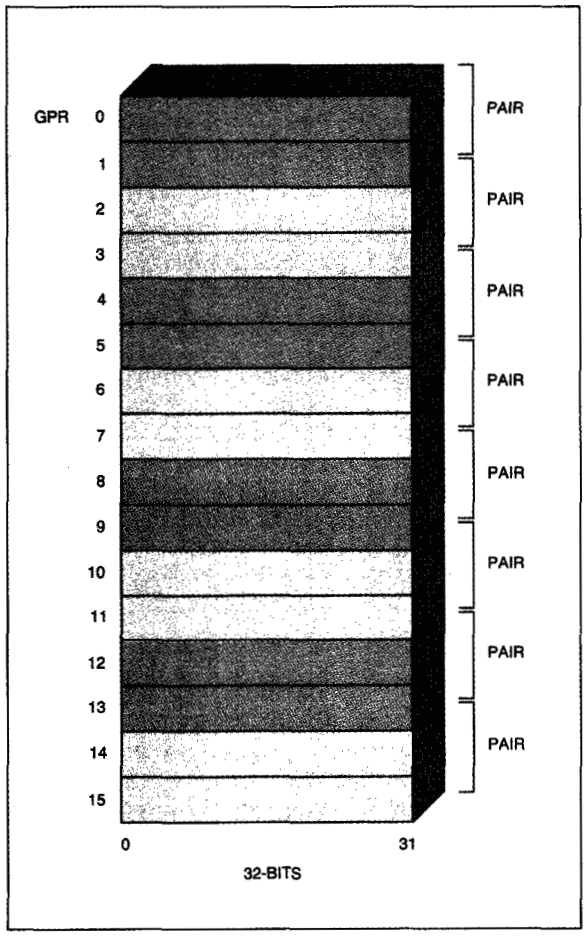
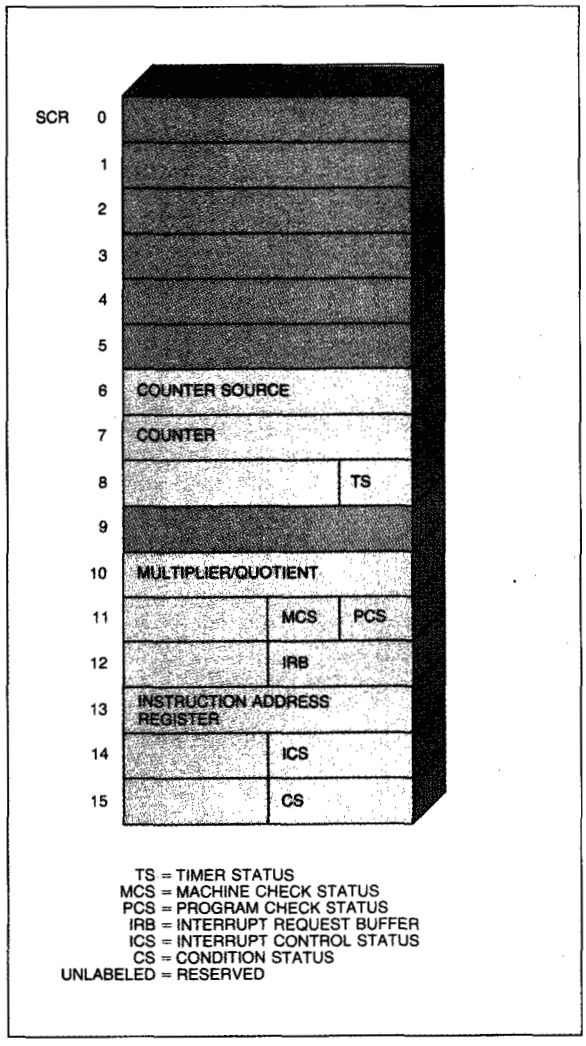


Figure 4 ROMP System Control Registers



The Instruction Address Register (IAR) is contained in Control Register 13, but it is not needed directly by the programmer. Most branch instructions are implicitly relative to the contents of the IAR, and the value in the IAR can be loaded into a general register by the Branch and Link instruction when necessary.

Instruction set. The ROMP provides a total of 118 instructions in the following ten classes:

Instruction Class	Number of Instructions
1. Memory Access	17
2. Address Computation	8
3. Branch and Jump	16
4. Trap	3
5. Move and Insert	13
6. Arithmetic	21
7. Logical	16
8. Shift	15
9. System Control	7
10. Input and Output	2
Total	118

The *Memory Access* instructions permit loading and storing data between the 16 GPRs and main memory. These instructions support four types of data:

- 8-bit (character) quantities
- 16-bit (halfword) quantities
- 16-bit algebraic (sign-extended halfword) quantities
- 32-bit (fullword) quantities.

Load Multiple and Store Multiple instructions are also included in this class. They permit loading or storing from one to 16 of the GPRs. A test and set instruction is provided for multiprocessor synchronization.

All Memory Access instructions compute the effective memory address as the sum of a GPR contents plus an immediate field specified in the instruction (base + displacement addressing). Two-byte memory access instructions provide a 4-bit immediate field, with 4-byte instructions providing a 16-bit immediate field.

The Memory Access instructions operate on data between memory and one or more GPRs. No memory-to-memory operations are provided. The architecture allows instruction execution to continue beyond a load instruction before the load is complete, if subsequent instructions do not use the loaded data. This increases system performance by overlapping memory access with subsequent instruction execution.

The *Address Computation* instructions compute memory addresses without changing the status of the condition codes. These instructions include a three-address add instruction (Compute Address Short), increment, decrement, and 2- and 4-byte instructions which permit loading a GPR with a 4-bit or 16-bit immediate value, respectively. Separate Compute Address Lower and Compute Address Upper instructions are provided to load a 16-bit immediate value into either the lower half or upper half of a GPR. Two Address Computation instructions are provided specifically to aid in the emulation of 16-bit architectures. They allow the computation of a 16-bit quantity to replace the low-order 16 bits of a GPR without altering the upper 16 bits.

The *Branch* and *Jump* instructions are provided for decision making. Jumps are 2 bytes long, and provide a relative range of plus or minus 254 bytes. Branches are 4 bytes long and provide a range of up to plus or minus 1 megabyte. A group of Branch and Link (BAL) instructions are also provided for subroutine linkage.

Many branch and branch-and-link instructions have a delayed branch form (called "Branch with Execute") which allows overlap of the branch target fetch with execution of one instruction following the branch (called the subject instruction). Execution of the subject occurs in parallel with fetching of the target instruction, thereby eliminating dead cycles that would normally occur during fetching of the target instruction.

Three *Trap* instructions are provided for run-time address checking. These instructions compare a register quantity against a limit, and cause a program check interrupt if the limit is exceeded.

The *Move* and *Insert* instructions support testing the value of any bit in a GPR, and the movement of any of the four 1-byte fields in a GPR. A Move instruction is provided that allows moving any one of the 32 bits in a GPR to a test bit in the condition status register, with a corresponding instruction that moves the test bit value to any of the 32 bits in a GPR. A series of Move Character instructions are included that move any of the four 1-byte fields in a GPR to another 1-byte field in a GPR.

The *Arithmetic* class supports standard Add and Subtract operations in both single- and extended-precision modes. Other instructions in this class include absolute value, ones- and twos-complement, compare, and sign-extend. Also, Multiply Step and Divide Step instructions are provided. The Multiply Step instruction produces a 2-bit result per step, and can be used to construct variable-length multiply operations. The Divide Step instruction produces a

single-bit result per step, and can be used to construct variable-length divide operations.

The *Logical* class provides AND, OR, XOR, and negation operations using two register quantities or one register and an immediate value. Also included in this class is a group of set and clear bit instructions that allow any bit in any GPR to be set to one or to zero.

The *Shift* class provides algebraic shift right, shift right, shift left, and left and right paired shifts. Shift amounts from 0 to 31 bits can be specified either as an immediate quantity in the instruction or as an indirect amount using the value in a GPR. The paired shifts provide nondestructive shifts that shift a specified GPR a given amount, and place the result in a different register (the other register of a register pair) without altering the source register.

Instructions in the *System Control* class are generally privileged instructions that are valid only in supervisor state. Included in this class are instructions that move GPRs to and from SCRs, set and clear SCR bits, Load Program Status, and Wait for interrupt. Also included is a nonprivileged Supervisor Call instruction.

Two instructions that load and store GPRs to I/O devices are included in the *Input and Output* class. These instructions are normally used to access control registers in the MMU or other system elements.

Memory addressing model

The basic concepts of memory addressing in ROMP are shown in Figure 5 and are similar to those of System/370. The smallest addressable storage unit is the 8-bit byte. Two bytes make a halfword, four bytes make a word (or fullword). Halfword and fullword quantities must be properly aligned on 2-byte and 4-byte boundaries in storage in order to be loaded and stored by the ROMP storage access instructions (in this area ROMP's memory model is that of System/360 rather than System/370). Strings of bytes (character strings) can begin on any byte address, but they are manipulated by subroutines rather than low-level ROMP instructions.

Memory addresses begin at 0 and increase "to the right," as in System/370. All quantities in storage are addressed by their leftmost (high-order) byte, without exception. This is true of the operands of load and store instructions and branch targets. The bytes of arithmetic quantities are never reversed as they are moved between memory and registers.

Figure 5 ROMP memory addressing with integral boundaries

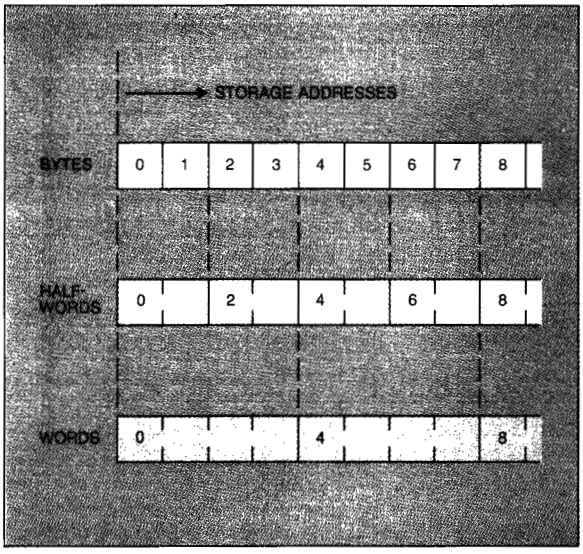
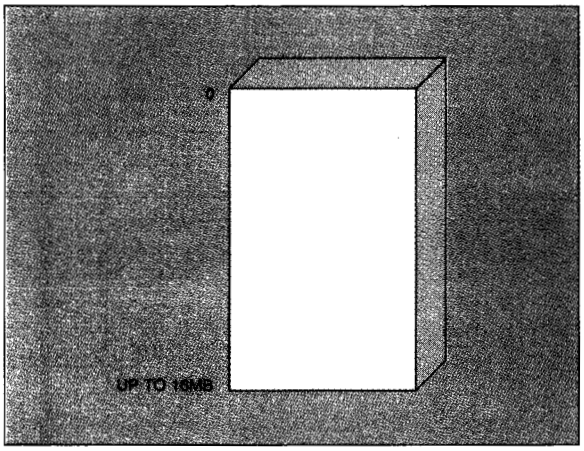


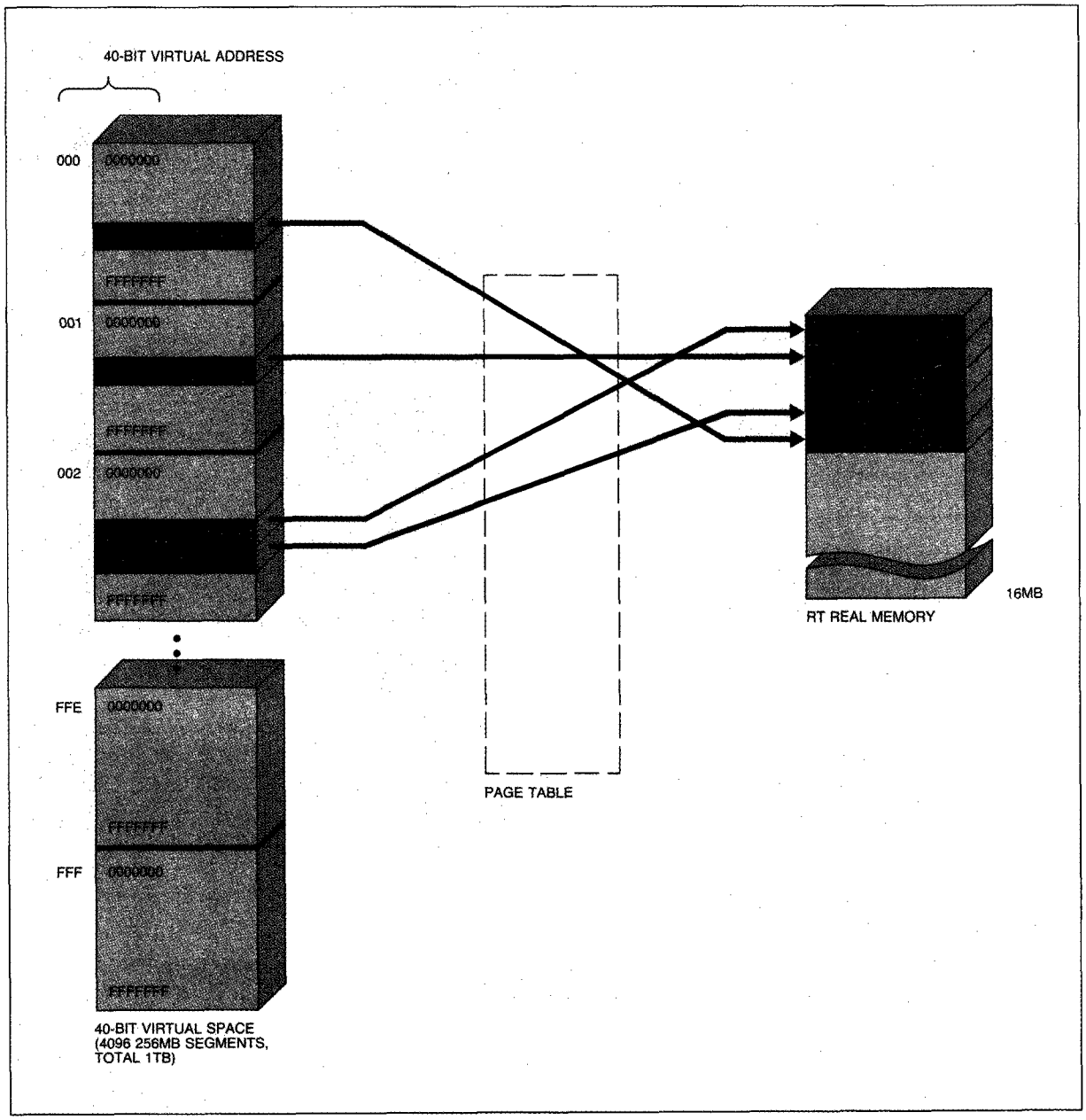
Figure 6 RT System real memory



When the virtual addressing mode of the ROMP MMU is enabled, there are three conceptual levels of addressing to be considered.

Real addressing. This is the lowest level, as depicted in Figure 6, the level that the programmer sees when running with virtual addressing turned off. Memory is a linear array of bytes starting at 0. Except for the recording of reference and change information, there

Figure 7 Mapping of 40-bit virtual addresses to real addresses



is no concept of "pages" at this level. Architecturally, real addresses are 32-bit quantities, but in the RT PC the maximum amount of memory that can be installed is 16 megabytes.

Long-form virtual addressing. This intermediate level, illustrated in Figure 7, becomes active when the MMU is enabled for virtual address translation. Virtual memory can be viewed as a collection of *segments* of 256 megabytes each. There are 4096 possible segments, each named by a 12-bit *segment id.* Each segment is made up of 2K-byte pages. Virtual addresses in this level are 40 bits: 12 bits for the segment id, 17 bits for the page number within

the segment, and 11 bits for the byte offset within the page. The total virtual space is thus 2^{40} bytes, or 1 terabyte.

Of course, this is virtual space. In reality only a tiny fraction of the 2^{40}-byte space is actually in use at once—only a few of the 4096 segments, and a maximum of only a few megabytes within each of those segments. The mapping of 40-bit virtual addresses to real addresses is done by the MMU, which signals a page fault to ROMP when an access to an unmapped page is attempted.

The programmer does not deal directly with 40-bit addresses. As is seen below, the ROMP hardware generates 32-bit virtual addresses, and the MMU constructs the 40-bit virtual address internally before translating it to real.

Short-form virtual addressing. This is the highest level, the level at which the programmer deals with virtual addressing. ROMP's address generation process is the same whether or not virtual address translation is enabled: Address computations result in 32-bit values, and 32-bit addresses are transmitted from ROMP to the MMU for loads, stores, and instruction fetches.

Inside the MMU, the first step of virtual address translation is the expansion of the 32-bit virtual address to a long-form (40-bit) address, as shown in Figure 8. To do this, the high-order four bits of the 32-bit short-form address are removed and used to select a 12-bit segment id from one of 16 *segment registers*. The segment registers are contained within the MMU and are loaded under control of the operating system; they are protected from modification by application programs. The 12-bit segment id is concatenated with the remaining 28 bits of the short-form virtual address to make a 40-bit virtual address.

The effect of this is that a program has a 32-bit window on the 40-bit world, in the form of a set of 16 of the possible 4096 256-megabyte segments. The segment to be used is selected by the top four bits of the 32-bit address, while the remaining bits are an offset within that segment.

For most purposes, the program is not aware of the presence of one or more segments in its 32-bit virtual space. With the AIX operating system, addresses of items on the stack lie in the range 30000000 through 3FFFFFFF (hex), while addresses of programs are in the range 20000000 through 2FFFFFFF. However, it is possible to make direct use of the segments by requesting that the operating system "map" a file to a segment, share one or more segments with other tasks, or use one of the segments to access the RT PC's memory-mapped I/O. A discussion of these uses is given below.

Table 1 Page protection keys

Page Key	Type of Page	Access Key	Load	Store
00	System read/write	0	Yes	Yes
	User no access	1	No	No
01	System read/write	0	Yes	Yes
	User read-only	1	Yes	No
10	Public read/write	0	Yes	Yes
		1	Yes	Yes
11	Public read-only	0	Yes	No
		1	Yes	No

Memory protection. In real addressing mode, no memory protection is given. This mode is intended for use by low-level system programs such as interrupt handlers. Almost all application code runs in virtual addressing mode.

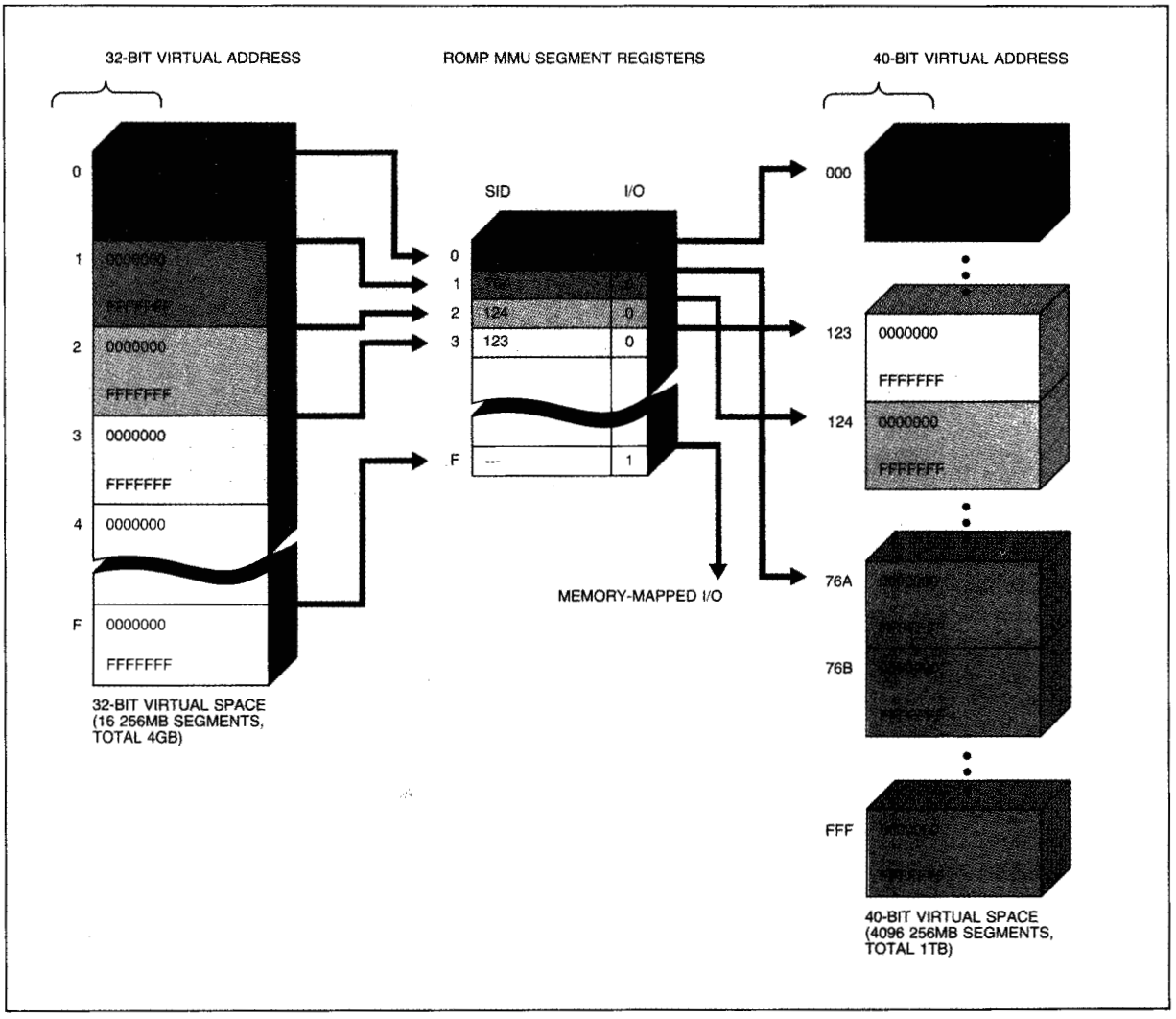
When virtual translation is enabled, one of two protection modes is selected. The most commonly used mode (for "normal" segments) is similar to that of System/370, in which a one-bit access key in the segment register is tested against a two-bit page protection key for the selected page, as shown in Table 1.

For "special" segments, a finer granularity of protection is provided. This mode is intended for use by database programs and others who need to distinguish between read/write and read-only access on items smaller than a page. Each page is divided into 16 "lines" of 128 bytes each, with read and write access granted on the individual line level. Details of this "line locking" function are discussed in References 7, 8, and 11.

Programming the ROMP

Almost all programming on the RT is done in high-level languages which hide low-level details such as the instruction set and the number of registers. These details are still important, for they affect the efficiency of the code that the compilers generate and thus the performance of RT programs. Some architectural features are pervasive, and show through the

Figure 8 Conversion of 32-bit virtual addresses to 40-bit virtual addresses



languages. They affect the methodology used by the programmer in designing and coding an application.

32 bits. On the RT, the natural unit of data is the 32-bit word. The general registers are 32 bits wide; hence, arithmetic on 32-bit quantities and 32-bit addresses is very efficient. The default integer size and pointer size for the various high-level languages on the RT is 32 bits.

Large linear memory space. There are no 64-kilobyte boundaries in the RT's addressing space. The smallest boundary is 256 megabytes, the size of a virtual storage segment. Almost all programs can be written with the presumption that any data item can be addressed with a standard 32-bit pointer; there are no distinctions between "near" and "far" pointers. There are no array size limitations such as 64K total entries or 64 kilobytes total space. Programs and data can easily be several megabytes in size.

Segments. A *segment* in the RT is much larger than the segments on most other machines, and it is used for different purposes. Each segment can be up to 256 megabytes and begins on a 256-megabyte boundary in the 32-bit virtual space; segments do

not overlap. Up to 16 segments can be mapped at once, and the program can change the mapping through calls to the operating system.

RT segments are used to provide

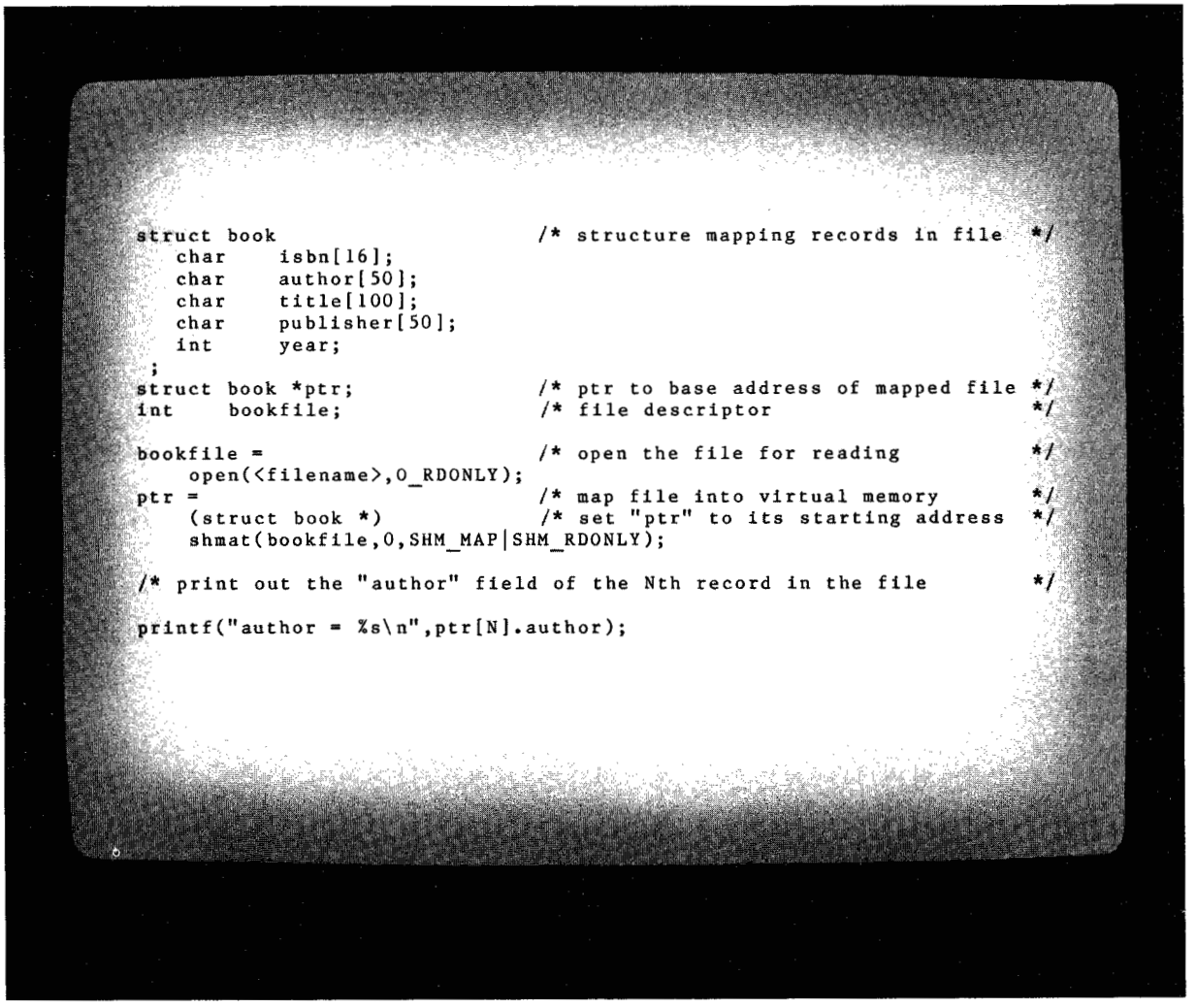
- Data isolation, through multiple address spaces. The address space of a task is the collection of segments to which it has access. In the AIX operating system, for example, one segment will contain the private code for the process, another its private data, and another its stack. These segments are not mapped into the address spaces of other processes, and thus are completely isolated from the other processes.
- Data sharing, by mapping the same segment into several different processes. Private data remain isolated as described above.
- Different levels of protection, by assigning different hardware protection keys to different segments. Thus, a segment containing code (even private code) is normally read-only, while a data segment is read-write.
- Access to input/output, by assigning a segment to memory-mapped I/O.
- Mapped files, by mapping the image of a file on disk into a virtual storage segment.

The other traditional use for segments, which is to extend addressability beyond the natural address

Figure 9 Writing data to bit-mapped display using RT System's memory-mapped I/O

```
int             bus;            /* file descriptor for I/O bus            */
struct hwdbase busbase;         /* returned values of bus addresses       */
short          *regen_buf;      /* memory-mapped addr of display buffer   */
short          *p;              /* ptr to a location in display buffer    */

bus = open("/dev/bus",O_RDWR);  /* get read/write access to I/O space     */
ioctl(bus,HWDBASE,&busbase);    /* get starting addr of memory-mapped I/O */
display_buf =                   /* compute starting addr of display buffer */
    (short *) (busbase.hwdmem + 0xD80000);

p = display_buf +               /* point to proper location in display buf */
    <offset of desired location>;
*p = <value>;                   /* store data into display buffer         */
```

Figure 10 Random access to a mapped file on the RT System



```
struct book                         /* structure mapping records in file  */
   char     isbn[16];
   char     author[50];
   char     title[100];
   char     publisher[50];
   int      year;
 ;
struct book *ptr;                   /* ptr to base address of mapped file */
int     bookfile;                   /* file descriptor                    */

bookfile =                          /* open the file for reading          */
    open(<filename>,O_RDONLY);
ptr =                               /* map file into virtual memory       */
    (struct book *)                 /* set "ptr" to its starting address  */
    shmat(bookfile,0,SHM_MAP|SHM_RDONLY);

/* print out the "author" field of the Nth record in the file             */

printf("author = %s\n",ptr[N].author);
```

width of the machine, is also possible in the RT but is not used nearly as often as on machines with smaller address widths. A 32-bit address spans 4 gigabytes, which is enough for most purposes. On the RT, this can be extended to 1 terabyte by manipulating the contents of the segment registers.

Memory-mapped I/O. On the RT, a single segment register is normally assigned to map the I/O address space rather than virtual memory. Access to this segment is controlled by protection hardware in the RT PC's I/O channel controller in a manner similar to the method used by the MMU to control access to pages of virtual storage. Thus, an application which has not been explicitly granted access to the I/O memory map cannot accidentally trigger an I/O operation.

For applications which need to deal directly with I/O, it is simple to do this in high-level languages by assigning the appropriate values to pointers and then using assignment statements to read from and write to the I/O space. As an example, it is easy to code a C program which writes a screen full of graphics data to a bit-mapped display, as illustrated in Figure 9.

Mapped files. RT segments can be used to "map" files into virtual storage, as depicted in Figure 10.

After a file is opened, an operating system call converts the file identifier into an address in virtual storage (a pointer) at which the first byte of the file appears. From then on, the file can be treated as a large array or structure in virtual storage, and is read and written using assignment statements (i.e., load and store instructions). Actual I/O to the file is implicit and is done as needed by the paging supervisor. When the file is closed, modified pages are flushed out to their proper places in the file system.

This treatment of files can greatly simplify an application, especially those that must read and write files at random. Assuming that the file will fit into a 256-megabyte segment, it is simpler and more efficient to access elements by moving a pointer or adjusting an array index than it is to do explicit reads and writes through an I/O buffer.

Conclusions

The IBM RT PC's ROMP processor represents an effective adaptation of the RISC approach begun in the 801 to the real world of a small but powerful computer system product. Many cost-driven trade-offs had to be made, but even so the ROMP executes at a sustained rate of 4.3 MIPS, just under half that of an 801 with the same cycle time. This is done without the benefit of the 801's two caches.

ROMP is a good architectural base for future growth. All the size limits in the current implementation (12-bit segment ids, 16-megabyte real memory) can be increased without major architectural modifications. It is possible to add selected functions which are not normally considered part of the RISC domain as long as they are carefully chosen and known from measurements of actual code to pay back more in performance than they cost. Examples that come to mind are floating-point and character string operations.

The RT PC is truly a high-level language machine in that almost everything written for it has been in C, FORTRAN, or some other high-level language. The programming interface has been moved above the assembly language level. Because of this, it is even possible to change the ROMP's instruction set and still maintain the parts of the programming model that show through in high-level languages. Only the lowest-level routines in the operating system (interrupt handlers, for example) would need to be changed; applications would need only to be recompiled. This raising of the level of the programming interface may prove to be one of the major benefits of the RT PC in the long run.

Cited references

1. M. E. Hopkins, "A perspective on the 801/Reduced Instruction Set Computer," *IBM Systems Journal* **26,** No. 1, 107–121 (1987).
2. G. Radin, "The 801 minicomputer," *SIGARCH Computer Architecture News* **10,** No. 2, 39–47 (March 1982). Revised version published in *IBM Journal of Research and Development* **27,** No. 3, 237–246 (May 1983).
3. M. Auslander and M. E. Hopkins, "An overview of the PL.8 compiler," *ACM SIGPLAN Notices* **17,** No. 26, 22–31 (June 1982); *Proceedings of the SIGPLAN '82 Symposium on Compiler Writing,* Boston, MA, June 23–25, 1982; published by the Association for Computing Machinery, 11 West 42nd Street, New York, NY 10036.
4. G. J. Chaitin, "Register allocation and spilling via graph coloring," *ACM SIGPLAN Notices* **17,** No. 26, 98–105 (June 1982); *Proceedings of the SIGPLAN '82 Symposium on Compiler Writing,* Boston, MA, June 23–25, 1982; published by the Association for Computing Machinery, 11 West 42nd Street, New York, NY 10036.
5. M. E. Hopkins, "Compiling for the RT PC ROMP," *RT Personal Computer Technology* (pp. 76–82), SA23-1057 (1986); available through IBM branch offices.
6. V. Markstein, J. Cocke, and P. Markstein, "Optimization of range checking," *ACM SIGPLAN Notices* **17,** No. 26, 114–119 (June 1982); *Proceedings of the SIGPLAN '82 Symposium on Compiler Writing,* Boston, MA, June 23–25, 1982; published by the Association for Computing Machinery, 11 West 42nd Street, New York, NY 10036. See also "The IBM RT PC Subroutine Linkage Conventions," *RT Personal Computer Technology* (pp. 131–133), SA23-1057 (1986); available through IBM branch offices.
7. P. D. Hester, R. O. Simpson, and A. Chang, "The IBM RT PC ROMP and Memory Management Unit Architecture," *RT Personal Computer Technology* (pp. 48–56), SA23-1057 (1986); available through IBM branch offices.
8. R. O. Simpson, "The IBM RT Personal Computer," *BYTE* **11,** No. 11, 43–78 (November 1986).
9. IBM Corporation, *IBM RT PC Advanced Interactive Executive Operating System Assembler Language Reference,* 59X7994 (1986); available through IBM branch offices.
10. IBM Corporation, *IBM RT PC Hardware Technical Reference,* Volume I, 75X0232 (1986); available through IBM branch offices.
11. A. Chang and M. F. Mergen, "801 Storage architecture and programming," *Proceedings of the 11th ACM Symposium on Operating Systems Principles,* 1987, to appear.

Richard O. Simpson *IBM Advanced Engineering Systems, Austin, Texas 78758.* Mr. Simpson has been involved with the architecture of the ROMP processor and memory management unit since 1981. Previously, he was part of the development groups for the IBM 5520 Administrative System, the Network Job Entry Facility for JES2, and a Federal Systems Division project to automate the U.S. Army's Pentagon Telecommunication Center.

Mr. Simpson joined IBM in 1969. He holds B.A. and M.E.E. degrees from Rice University, and is a Ph.D. candidate in computer science at the University of Texas at Austin.

Phillip D. Hester *IBM Advanced Engineering Systems, Austin, Texas 78758.* Mr. Hester is Engineering Center Manager for Advanced Engineering Systems in Austin, Texas. Previously, he was responsible for architecture and performance of the RT PC. He was also involved in the architecture, design, and implementation of the RT PC/RISC microprocessor and memory management unit. He has numerous patents and technical publications on RISC processors. Mr. Hester joined IBM Austin in 1976 after receiving a B.S. degree in electrical engineering from the University of Texas at Austin. While in IBM, he received an M.S. degree in engineering from the University of Texas in 1981. He is a member of Eta Kappa Nu and Tau Beta Pi.

Reprint Order No. G321-5301.