PROGRAMMING THE 80960

Tony Baker
Intel Corporation
Hillsboro, OR

Tony Baker is a technical marketing manager for the 80960 product family. He has been employed at Intel for the past ten years in technical marketing and sales functions. He received his BSEE degree from the University of Portland and has done graduate studies at Portland State University.

Programming the 80960

Tony Baker, Intel Corporation, Hillsboro OR

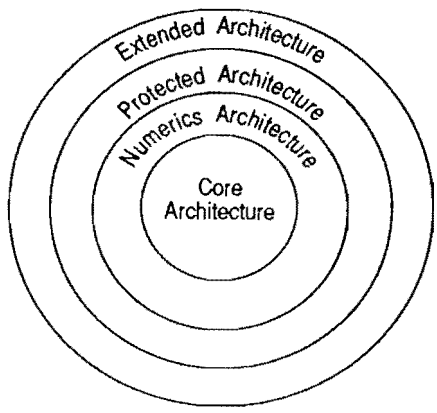
1. ABSTRACT

Intel Corporation's i960[1] processor family combines architectural features normally found in RISC processors with others contained in more traditional processors. The resulting family of processors yields high performance, yet provides a robust and easy to use architecture model for compiler designers and application programmers. This paper discusses many ways in which software designers can use the features of the i960 architecture.

2. INTRODUCTION

Intel's i960 architecture was developed during the mid-1980's with three equally important goals: First, it should have applicability to a wide range of areas, from single chip controllers to large-scale, high performance devices. Second, it should have high staying power. And finally, it should be optimized for high-performance implementations, especially when used with low cost memory systems.

To facilitate the wide range of applications envisioned by the architects of the i960, several architectural *layers* were defined from the onset (see Figure).



Each layer is a complete superset of the layer it encompasses. The first layer, known as the "core", defines the fundamental instructions, registers, and operation of all i960 components. For applications requiring numerics support, a second layer is added which provides full IEEE 754 floating point support, *long-double* (80-bit) floating point registers, and a robust set of floating point instructions. Applications requiring memory management and hardware support for multitasking (such as those written in ADA) are provided support through implementations of the protected layer of the architecture. The final layer, the

[1] i960 is a registered trademark of Intel Corporation

extended architecture, provides hardware enforced objects, useful for secure and object oriented language applications.

The i960 architecture is implemented in a wide range of components, ranging in performance from 5 MIPS to over 50 MIPS sustained operation. Intel introduced the first i960 components in 1988 with the 80960KA (core architecture), 80960KB (numerics architecture), and 80960MC (protected architecture). These components are low cost integrated 32-bit devices which provide performance in the range of 7-10 VUPS[2]. For systems with more demanding cost requirements, the 80960SA (core architecture) and 80960SB (numeric architecture) components were introduced in 1990, providing approximately 5 VUPS through a low-cost 16-bit system data bus. For designs requiring higher performance, Intel introduced the high performance 80960CA. The 80960CA demonstrates the flexibility of the architecture; it is capable of executing up to three instructions per clock cycle while retaining object code compatibility with previous components. In the spring of 1991, Intel introduced new high performace parts: the 80960MM and 80960MX military components. These components further extend the superscalar capabilities introduced in the 80960CA through the addition of execution units and a second high speed system bus.

3. A PROGRAMMER'S VIEW OF THE ARCHITECTURE

Advances in RISC research occurred simultaneously with much of the i960 architecture definition. Many of the results of this research were applied to the i960 to meet the criteria of a high performance foundation. Some attributes (such as dependence on high speed memory subsystems) did not match well to the stated goals of the architecture and therefore were avoided. The resulting architecture has been called everything from a CRISP (complex reduced instruction set processor) to a 'CISCy RISC' to a 'RISCy CISC'.

3.1 Flat Address Space

The address space of all i960 processors spans four gigabytes (2^{32}). Addressing is linear, i.e. there is no segmentation. All input/output is memory mapped. Except for a small boot area and on-chip I/O functions, there are no reserved locations in the address space. Interrupt tables, fault (error condition) tables, and system call tables can be located anywhere in the address space. For processors such as the 80960MC and 80960MX which implement virtual memory, an MMU provides 4K-byte mapping of virtual memory to physical memory.

3.2 Data Types

The i960 can access memory as 8, 16, 32, 64, 96, or 128-bit quantities. For 8, 16, and 32-bit accesses, *integer* load/store instructions will sign extend the most significant bit as appropriate. *Ordinal* instruction are provided for unsigned data. The larger accesses (64, 96,

---

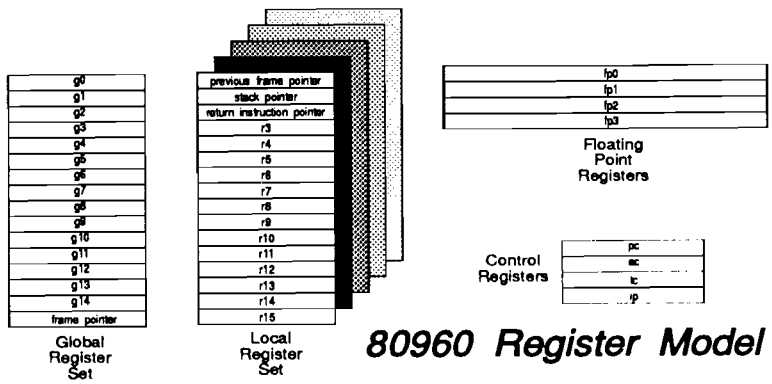
[2] 30 Vax [equivalent] Units of Processing, or 30 times the speed of the same program executing on a VAX 11/780 processor. VAX is a trademark of Digital Equipment Corp.

and 128-bit) are useful for manipulation of floating point values, data structures, and provide an excellent match to the hardware memory burst capability of the i960. The i960CA exploits this data movement by allowing access to an on-chip SRAM through a 128-bit data bus, allowing up to four registers to be accessed in a single cycle.

3.3 Register Model

The i960 provides thirty two 32-bit general purpose registers that are visible to the programmer at any given time. Of these thirty two registers, four provide linkage information for the i960 call/return mechanism, and twenty eight are freely available to the application. For versions of the i960 that implement the numerics architecture, the processors provides an additional four 80-bit floating point registers. Floating point is not restricted to these registers; any general purpose register (for *float* operations) or pair of registers (for *double* operations) can be used.

The general purpose registers are divided into two types: *global* (**g0..g15**) and *local* (**r0..r15**). Global registers are like those of virtually any existing processor, in that they are always accessible and change only through their explicit use as operands. The local registers are accessible in the same way as the global registers. Their difference lies in the fact that the processor allocates a new set each time a call instruction executes. A ret instruction restores the previous register set for the calling procedure.



At first inspection, such a model would appear to create tremendous overhead for allocation of local registers. The i960 avoids this overhead by implementing a special cache (64 registers on the 80960KA/KB/MC, up to 240 on the 80960CA) in which the processor stores previous sets of local registers. The cache behaves like a FIFO; only when it fills is the oldest set of registers flushed to a previously allocated place on the stack. The i960 restores these registers only when the call depth decreases to the level corresponding to the flushed registers.

The local registers allocated to a procedure are used for storage of variables restricted

to that procedure. Architecture modeling during design of the i960 showed that for the Unix[3] System V kernel, about 97% of the functions required no local storage beyond the local registers. Additionally, stack pointers maintain "place holders" for stack frames at all times. When the i960 allocates a new frame, space is reserved on the procedure stack (pointed to by a register called the frame pointer). The processor only updates these reserved locations on the stack when it spills from its internal frame cache, or under software control. By pre-allocating these frames, two benefits are derived: 1) each procedure is guaranteed to have resources available to it to spill a frame if necessary, and 2) If the procedure requires local storage beyond that of the local registers, additional space can be allocated on the stack contiguous with the local registers.

3.4 Call/Return

When a call instruction executes, the i960 allocates a new frame, and (if necessary), spills a frame cache entry to the stack. Also, the processor records links to the calling subroutine, stores the instruction pointer of the calling subroutine, and establishes a new stack pointer. On execution of a return instruction, the opposite sequence occurs.

In some instances, the called procedure requires little to no register resources of its own. For these cases, the i960 architecture provides a branch-and-link (bal) instruction. This instruction records the "calling" address and transfers control to the new procedure without allocating a new set of registers. By providing both the call and the bal methods of procedure invocation, high level language compilers can choose the optimal method for procedure invocation.

As mentioned previously, the local and global registers are completely general purpose in the i960. To simplify programming and interface between tool sets, Intel has defined a calling sequence for i960 compilers. Tools produced by Intel, as well as third party designers, support this calling sequence.

The global registers are commonly used to pass parameters to and from procedures. Registers **g0** through **g7** are used to pass up to eight words of parameters to a procedure. The calling procedure places values into the registers in increasing register numbers going from left to right through the parameter list. Parameters with size shorter than or equal to one word are placed in a single register, two word parameters (such as type *double*) are placed in an even numbered register and the subsequent registers. For example, the procedure call

```
int xyz, abc;
...
...
abc = test(xyz, 1, 'x');
```

---

[3]Unix is a registered trademark of AT & T

produces the following:

```
mov       r7, g0      # local variable xyz first parameter
ldconst   1, g1       # constant 1 second parameter
ldconst   120, g2     # 'x'  third parameter
call      _test       # subroutine call
mov       g0, r3      # result returned and stored in 'abc'
```

The above example also illustrates the return mechanism for this calling convention. The processor returns values shorter than four words in length in registers **g0...g3**, allowing integer, single, double, or extended precision floating point results to be returned.

In this calling convention, the calling subroutine must assume that the values in registers **g0... g7** are lost during the system call. Registers **g8... g12** are preserved across calls. Register **g13** is used as a pointer to the results of a procedure call when the returned value is too large to fit within the registers.

During definition of the i960 architecture, our research showed that over 98% of all procedure calls use six or fewer parameters . For the cases in which this size is exceeded, global registers through **g12** can be used for up to twelve parameters. For those procedures that require more than 12 parameters, register **g14** is used to point at an block of memory containing the remaining arguments. If no block exists, **g14** is assumed to be zero. Since the use of many parameters occurs infrequently, this register is initialized at startup and seldom changed. Typically, existing compilers and applications use this register as a constant zero value, and therefore programmers using this register are advised to clear it when done.

3.5 Instructions

The i960 is a load/store architecture, which implies that memory operations are disjoint from ALU operations. The i960 exploits this model by allowing concurrent execution of both types of instructions in the machine. Instructions in the i960 typically are three operand, with two sources and one destination. For instructions involving the ALU (called **REG** instructions), source operands can be any general purpose local or global register, floating point register (for floating point instructions) or literal in the range of 0..31. For memory reference instructions (called **MEM** instructions), memory addresses can be one of several addressing modes. The i960 was designed to support the simple, high performance addressing modes (direct and register-indirect) normally found in RISC processors, *and* more complex modes which are not traditionally associated with RISC. The complex addressing modes were added to allow further exploitation of parallelism in the i960, and to improve code density in systems.

A third class of instructions, called **CTRL** (for control), provide all flow control and call/return. CTRL instructions also may execute independently and concurrently with other

instructions in the processor. Unlike other RISC architectures, the i960 does not utilize delay slots after branches. Because the i960 architecture supports superscalar implementations (and hence the ability for several instructions to be in the same pipeline slot), the number of delay slots could not easily be fixed across all parts. The architects chose an alternative method, called *branch prediction.* Branch prediction takes the form of a bit in the CTRL opcode which provides the processor a hint of the *usual* path through the branch (i.e. taken or not taken). The control logic of the processor loads the pipeline from this path, and if the hint was correct, the processor incurs no penalty. If the hint was wrong, a pipeline stall occurs.

3.6 Superscalar

The i960CA, i960MM, and i960MX components all are implemented around a superscalar processor core. A superscalar processor is one in which the processor can fetch, decode, and execute more than one instruction simultaneously from an otherwise conventional instruction stream. In the case of the i960CA, i960MM, and i960MX, these processors can dispatch up to three instructions per clock cycle: one REG, one MEM, and one CTRL. The i960MM and i960MX also can use the MEM instruction processor as a second ALU, effectively allowing up to two REG instructions to execute in the same clock cycle.

The nature of software dictates that the components cannot alter the flow of the program, therefore any dependencies that exist between instructions that would otherwise be dispatched simultaneously will force the processor to dispatch the instructions sequentially. Additionally, a *scoreboarding* mechanism exists in the component which effectively prevents dispatch of instructions involving any resources that are busy. These resources typically include registers targeted by a pending load operation or functional units, such as the integer multiplier. Since such dependencies reduce the performance from a peak of two to three instructions per cycle to one instruction per cycle, performance can easily be gained by avoiding such dependencies.

Compilers available from Intel and third parties follow some simple guidelines for avoiding dependencies:

1) Move load operations as early in the instruction stream as possible. In this way, the bus latency incurred from the load operation can be masked.

2) Begin multi-clock instructions as early as possible. While these instructions are executing, other instructions can be dispatched and executed.

3) Set the branch prediction bit for the 'usual' case. For software loops, assume the branch is taken.

4) Where possible, reorder the instructions to allow dispatch of REG, MEM, and CTRL instructions simultaneously.

4. COMPILER OPTIMIZATIONS

All the available i960 compilers are optimized to support the feature set of the processor. In turn, the richness of the architecture allow easy mapping of many high level language functions to the processor.

4.1 SRAM

The i960CA component contains 1K-byte of on-chip SRAM. This RAM connects to the processor core via a 128-bit data bus, allowing quad word accesses in a single cycle with no external bus cycles. When this RAM is used as scratch space and to store commonly accessed global variables, dramatic performance gains[4] are achievable. *Gcc960* provides support, through a two pass compiler, for identification of commonly referenced global variables (through static means). This data is then utilized by a second compiler pass to allocate these variables to on-chip SRAM. Since this allocation is done statically (through data gathered via data flow analysis at compile time), application of knowledge of run-time behavior of global variables may further improve the results.

4.2 Complex Addressing

The addressing modes supported by the i960 map well to high level languages. The following example declarations (although somewhat contrived) illustrate the code generated by *gcc960*:

```
int global; int g_array[20];
test(){
          int local; int i; int *ptr = &g_array;
          struct{
                char a,b,c,d;
                short e,f;
                int g;
                int var_2;
          } s, as[5];

/* below declarations here */

}
```

[4]Performance gains of approximately 10-20% have been seen on page description language applications. Gains of over 30% were seen on Dhrystone 1.1

HLL Code	Assembler Code	Addressing Mode
x=global;	ld _global, g0	12 or 32-bit address
x=*ptr;	ld (r6), g0	register-indirect
x=local;	ld 80(fp), g0	register-indirect + offset
x=s->var_2	ld 12(r8), g0	register-indirect + offset
x=ptr[i]	ld (r6)[r4*4], g0	indexed indirect
x=as[i]->var_2;	ld 12(r9)[r4*16], g0	indexed indirect + offset

Complex addressing in the i960 produces an average[5] of about 20% less code than simple register indirect addressing, which results in reduced bus traffic for operand fetching.

4.3 Leaf Procedures

The i960 provides a branch and link instruction for procedures which require little additional register resources. The branch and link instruction simply records the return address and transfers control to the target address, with no change in local registers. *Gcc960* recognizes candidate procedures and generates a special form of subroutine, which is reachable by either branch and link or the traditional call instruction. Such a procedure is called a *leaf procedure*. For example, the procedure

```
int product(int a)
{
return (a*a);
}
```

compiles to the code

```
.leafproc       _product, product.lf

_product:               #CALL entry point
     lda   LR1, g14     #load address of ret instruction in g14
product.lf:             #Branch and Link entry (return in g14)
     mov   g14, g1      #copy g14 into g1
     mov   0, g14       #clear g14
     muli  g0, g0, g0   #produce product
     bx    (g1)         #branch and link return
LR1: ret                #return from call
```

Note that the multiple entry points to the procedure provide for each entry mechanism, a return for entry by a call instruction and a branch for return from a branch and link.

4.4 Cache Optimizations

---

[5]Measurements on *gcc960* were done with complex addressing and with register indirect only across several applications.

During the first compile pass on *gcc960*, data flow analysis provides information on how procedures interface with one another. This data is used for two purposes: to optimize code organization for better instruction cache hit ratios, and to recognize the best opportunities for inlining procedures.

4.5 Branch Prediction

The run time profile can optionally provide heuristic data back to a branch prediction post processor. This data provides results indicating which path was usually taken for each CTRL instruction encountered during execution. This data is then used to set the appropriate bit in the CTRL instruction opcode. This optimization provides an average of 3-5% performance improvement to applications.

5. SUMMARY

The i960 processor family has several members covering a wide range of performance. The i960 offers an easy to use, robust architecture, from which both application designers and tool designer can create powerful applications.