The Spectrum Instruction Set, A 3000 Hacker's View

By Robert M. Green
(c) 1988, Robelle Consulting Ltd.
8648 Armstrong Road, R.R. #6
Langley, B.C. V3A 4P9 Canada
(604) 888-3666

I have been writing and debugging code for the Classic 3000 instruction set for over 15 years. Like most of you, I have been impatiently awaiting Hewlett-Packard's new line of computers, code-named "Spectrum". Now that actual Spectrum machines are spewing forth from Hewlett-Packard factories in ever-increasing numbers and going into production in my customers' computer rooms, it is time to start learning about these CPUs.

By the way, Hewlett-Packard prefers us to refer to their new computer family as the **HPPA** (*HP Precision Architecture*), not as the **Spectrum**.

I called HP's Direct Marketing group at (800) 538-8787 and used my VISA card to order a manual on the HPPA instruction set:

Precision Architecture and Instruction Reference Manual,
HP part number 09749-90014.

Like everyone else, I had read numerous articles from Hewlett-Packard about the objectives of the Precision Architecture, but I found clean, solid facts about the MPE XL machines hard to pin down. I needed some basic information to build on and the hardware instructions themselves seemed like a good base. Whatever software we eventually run on the HPPA machines will all be coded in the HPPA machine instructions.

Starting from the HP manual, I set out to compare the HPPA instructions with the Classic 3000 instructions and see what interesting differences I could uncover. Whether you ever personally write machine code for the HPPA or not, it can't hurt you to know something about these basic building blocks.

Goals of the Spectrum Project

A computer is a tool to execute programs built from sequences of simple "instructions". A typical instruction is something like "Add these two numbers together". The Classic HP 3000, designed in 1970, has a "complex instruction set", meaning that the instructions which programmers use are not the <u>real</u> hardware instructions. Each complex instruction is implemented by a hidden microprogram written in the real instructions.

The HPPA is a RISC machine, a "Reduced Instruction Set Computer", meaning that the microprogrammed instructions were removed. The programmers use the machine's real hardware instructions. Any task too complex for the RISC hardware is done by executing a series of the basic machine instructions, either as in-line code or by calling a subroutine.

The HP manual describes the observations that led to the idea of RISC computers:

> "Extensive research into patterns of computer usage reveals that general-purpose computers spend up to 80% of their time executing simple instructions such as load, store, and branch. The more complex instructions are used infrequently. On architectures with large, complex instruction sets, the simple, often-executed instructions incur a performance penalty caused by the overhead of additional instruction decoding, the use of microcode, and longer cycle time resulting from increased functionality..."
>
> "The RISC features implemented with the HP Precision Architecture include:
>
> Direct hardware implementation; no microcode.
> Fixed instruction size, one word in length.
> Small number of instruction types.
> Small number of addressing modes.
> Reduced memory access -- only load and store."

A primary goal of the HPPA is to complete the execution of one instruction in each machine cycle, and to keep that cycle time as short as possible.

General Structure of Spectrum Machines

According to the HP manual, the HPPA machines have 32 general registers available to the programmer, each with 32 bits. They are referred to as GR 0 to GR 31. Only GR 0, GR 1, and GR 31 have a hardware-defined special purpose, although other registers may be reserved by software convention. GR 0 is the bit bucket; when used as a source of data, it always provides zeroes. When used as a target, it throws away the result. GR 1 is used as the target in the Add Immediate Left instruction and GR 31 is used in the Branch and Link External instruction.

This general-register organization contrasts strongly with the stack organization of the older 3000. In the Classic 3000, the programmer has access to a push-down stack and to 16-bit registers whose hardware function is highly specialized (i.e., Q points to local variables, DB to global variables, etc.). The closest thing to a general register is the Index Register, and it is used for special functions in many instructions.

All HPPA instructions are 32 bits long, with the first 6 bits reserved for the Major Opcode. This allows 64 major opcodes, allocated as follows:

23 opcodes are currently illegal (allowing HP lots of room for expansion),
27 opcodes are single instructions (e.g., 1 is Load Byte, LDB),
12 opcodes are instruction groups (e.g., 0 is the System Operation group),
1 opcode for up to 4 tightly-coupled Special Function Assist processors,
1 opcode for Co-processors, including 15 floating-point instructions.

The 27 major opcodes that invoke a single instruction provide load, store, branch, and other functions (e.g., LDB, STB, LDW, STW, etc.).

The 12 instruction groups expand into 108 unique instructions, as shown below. I show the opcodes for the 12 groups as Hexadecimal (base 16) values with a $ prefix. You should brush up on your Hex arithmetic, because the Spectrums are definitely Hex machines.

Opcode	Instruction Count / Type	
$00	11	system control instructions
$01	19	memory management instructions
$02	31	arithmetic/logical instructions
$03	15	indexed and memory instructions
$3A	4	unconditional branch instructions
$25	6	immediate arithmetic instructions
$2C		"
$2D		"
$34	4	extract/deposit instructions
$35		"
$09	8	co-processor load and store instructions
$0B		"

Floating Point

The floating-point co-processor follows the IEEE standard. It provides 15 functions, including square root, on three sizes of number: 32-bit, 64-bit, and 128-bit. Please note, however, that MPE XL versions of HPPA also support the Classic 3000 format for floating-point, via software emulation. The two formats are not compatible. For maximum performance I assume that you must convert your data files to the IEEE standard and also convert all of your application programs and third-party or contributed tools.

Another source of HPPA information, the book **Beyond RISC!** published by SRN, describes the floating-point problem this way:

> "The HP 3000 uses 9 bits for the exponent and 22 bits for the mantissa... The IEEE format defines a single precision number to contain 8 bits for the exponent and 23 bits for the mantissa. While the difference is small, it affects the size of the numbers that can be stored... The floating point format can be a problem in migration if the format is used extensively in disc files and databases."

The Fortran/XL and Pascal/XL compilers have options to force use of the old Classic 3000 floating point, but remember: this uses software instead of hardware for arithmetic. There is no good way that I am aware of to tell which format is being used for the floating point numbers in a particular database or file.

Total Instruction Count

If my arithmetic is correct, that makes a total of 155 instructions for the HPPA. Many of these are minor variations or are of interest only to low-level systems programmers.

How Does the Spectrum Multiply?

When comparing the Classic 3000 architecture with the HPPA, two of the most obvious deletions are the Integer Multiply and Integer Divide instructions. Since these instructions are used less than 1% of the time, neither the HPPA nor the Classic 3000 has the expensive hardware needed to do fast Integer Multiply and Divide. On the Classic 3000, the designers use a complex microprogram that is hidden from the customer and may vary from model to model. The HPPA designers wanted to multiply with reasonable performance without the microprogram or special hardware.

There is an interesting article about this problem in the Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), October 1987, published by The Computer Society of the IEEE. Whew! The paper is:

Integer Multiplication and Division on the HP Precision Architecture
Daniel J. Magenheimer, Liz Peters, Karl Pettis, Dan Zuras.

In their search for a fast method to multiply and divide, the HPPA team started out with general algorithms for subroutines to multiply and divide two integers. The usual algorithm for integer multiply, called "Booth encoding", involves looping through the multiplier two bits at a time replacing strings of zeroes or ones with a constant.

The HPPA stores numbers as 32-bit words, each bit or binary digit being capable of storing a one or a zero. If two bits are processed per loop, the Booth algorithm requires 16 such loops. To support this clever and tricky algorithm, machine designers often include a Multiply Step (and Divide Step) instruction to perform the two-bit processing. However, the HPPA designers found that the special Multiply Step and Divide Step instructions would have made their CPU more complex and increased the basic cycle time. This was not acceptable because it conflicted with the goals of a RISC architecture.

The HPPA designers attacked this problem in the same way that we often attack performance problems with database systems. They did a frequency analysis of actual multiplications in real user programs (and divisions -- the strategies in the IEEE article apply to divide as well as to multiply). They found that 91% of multiply tasks include a constant operand known at compile time, that the constant operands tend to be small, that non-constant operands also tend to be small, and that standard results occur more frequently than extended results (i.e., 32-bit answer versus 64-bit).

> "In the spirit of the philosophy espoused by RISC architects we should atttempt to optimize the most frequent cases, even at the cost of decreasing performance of the less frequent cases... By recognizing the inherent non-uniformity and special cases of operands (and results), we may be able to increase the overall performance."

Rather than implement a costly Multiply Step instruction that would slow down the basic cycle time for all instructions, they decided to take advantage of some hardware that the HPPA needed for another purpose. Remember, the HPPA is a byte machine, meaning that the byte (8 bits) is the basic unit that it can manipulate. All of memory is referenced via consecutive byte addresses, such as byte 1, byte 23, byte 4,567,890. However, most numbers are stored as words (4 bytes, 32 bits), half-words (2 bytes, 16 bits), or double words (8 bytes, 64 bits).

If you have a table of words starting at byte address 4000 and you want the 30th word in the table, you must compute the byte address of that word. The calculation is simple: 4 times 30 + 4000 = 4120. The position of the 30th word after byte 4000 is "4 times 30" because there are 4 bytes per word.

The HPPA has basic instructions to access elements in tables of words, half-words, and double-words. To ensure that these instructions execute in one CPU cycle, the HPPA includes hardware called a "pre-shifter"; it can multiply any register by 2, 4, or 8 prior to adding it with another register. It is called a shifter because it shifts bits to the left; in binary arithmetic, shifting one bit to the left is the same as multiplying by 2, shifting two bits is multiplying by 4, and shifting 3 bits is the same as multiplying by 8. (Computers like powers of 2, such as 4, 8, 16, 32...)

Since this "pre-shifter" was already needed, the HPPA designers could inexpensively include nine instructions to make the shifter accessible to the programmer. Each instruction shifts any of the 32 registers, adds it to any register, and stores the result in any register. The shift count can be one, two, or three bits and the add can be INTEGER, LOGICAL, or INTEGER with TRAP on OVERFLOW.

SH1ADD	Shift One and Add
SH1ADDL	Shift One and Add Logical
SH1ADDO	Shift One, Add and Trap on Overflow
SH2ADD	Shift Two and Add
SH2ADDL	Shift Two and Add Logical
SH2ADDO	Shift Two, Add and Trap on Overflow
SH3ADD	Shift Three and Add
SH3ADDL	Shift Three and Add Logical
SH3ADDO	Shift Three, Add and Trap on Overflow

Overflow - An Aside.

When the result of a multiplication won't fit in the 32-bit word that the HPPA uses to hold numbers, an "overflow" occurs. Some programming languages insist that overflow errors be detected and handled in a special way. Other languages, such as C, have no special requirements. Most computation instructions on the HPPA have a form that traps for overflow and another form that does not trap.

Now they have very fast instructions that allow both a shift (i.e., a multiply by two, four or eight) and an add. This is a powerful building block for fast multiplication.

Multiplying By Constants

Let's look at multiplying by a constant first, since that happens so often in practice. Multiplying a number by 10 is the same as adding the number together 10 times, or multiplying by 5 twice. The key to implementing constant multiplication is that any multiply operation can be converted to a series of additions and smaller multiplies:

```
120 = 10 times 12
120 = (5 times 12) + (5 times 12)
120 = (4 times 12 + 12) + (4 times 12 + 12)
```

The only tricky step above is the last one. We converted to this format because computers like to multiply by powers of 2 (e.g., 2, 4, 8). Remember, our *Shift-and-Add* instructions give us the ability to multiply any register by 2, 4 or 8 in a single instruction. The compiler can convert any multiplication by a constant into a series of specific Add and Shift-and-Add instructions. Generally, the HPPA can do such a multiply in four or fewer instructions, often in only one or two.

To multiply register R by 10, we would use the following two instructions:

```
SH2ADD R,R,R ; shift R two (multiply by 4),  add to original R, store in R.
ADD R,R,R ; add R to R (double R) and store back in R.
```

The first instruction leaves us with (5 times R) in register R and the second instruction doubles R, leaving us the (10 times R) in register R.

In our example, R would have the value 12:

```
SH2ADD    12 * 4 = 48 (shift two), 48 + 12 = 60 (add to original).
ADD       60 + 60 = 120
```

Multiplication by small constants occurs often in programs because of the need to index into tables. In order to compute the byte address of the table element, the program must multiply the element number by the element size in bytes (usually small!).

Multiplying By Variables

Examine the following statement from a Pascal program:

```
ExtendedPrice := Quantity * UnitPrice;
```

It multiplies "quantity" by "price" to give the extended price. Neither the quantity nor the price is known when the program is written, since this statement will be executed for many different parts on many different orders. In this case, the HPPA needs a multiply method that can handle any possible values for the operands.

When neither of the operands is known at compile-time, we have "multiply-by-variable". You can construct a very simple, but slow, method of multiplying by an arbitrary variable using just three basic computer functions: addition, divide by two (arithmetic right shift), and bit test. This "naive" algorithm loops 32 times, processing one of the 32 bits each time. A simple optimization is to exit the loop as soon as the shifted multiplier is zero. Since most operands have many leading zero bits, this improves the "average" multiply by at least 40 percent. For example, the number 40 is represented in the computer as 26 leading zero digits, followed by 101000.

Here is the algorithm to multiply A times B giving Result, represented in a high-level language:

```
Result := 0
while A not zero do
   if A odd then Result:=Result+B  {add if last bit = 1}
   B := B + B    {double B; multiply by 2; shift left 1 bit}
   A := A / 2    {shift right 1 bit, get next bit to test}
```

Each time through the loop, we add the current value of B to the Result if A has a one in the last bit (i.e., is odd), then we double the value of B and shift A right one bit to bring up the next bit to be examined. Although this may seem obscure, it is actually the same process you apply when you do decimal multiplication by hand. The only difference is that we are using Base 2 instead of Base 10.

The Spectrum's Shift-and-Add instructions allow us to examine several bits at a time from both the multiplier and multiplicand, reducing the worst case and the average case. But this is just the start of optimizing based on frequency analysis:

> "It is rarely the case that both operands are large, say larger than 16 bits, because the result will be an overflow... If the multiplication does not result in an overflow, at least one of the operands must be representable in less than 16 bits. With a simple test and possible swap, we have reduced the maximum number of times through the loop to four and the average to two."

The HPPA team then noticed that each time through the loop multiplies the multiplicand by a number between 0 and 15. Because the HPPA can do any small constant multiply in three instructions or less as described above, they introduced a CASE statement to include in the loop the explicit code for each possible constant multiply. They also observed that one of the operands is less than 16 over half the time, so they optimized for single loop multiply. Finally, they added a quick exit for values of zero and one, and special checks for positive operands (which occur 90% of the time).

Here is how they describe their optimization results:

> "Multiplication by compile-time constants can generally be performed in four or fewer (single-cycle) instructions; multiplications by variables, including full overflow checking, can be obtained in an average of 20 (single-cycle) instructions... on the Precision Architecture the average multiply requires about six cycles."

The HPPA designers chose to concentrate on speeding up the most-often requested multiplication tasks at the expense of the least-often used tasks. They used a shift-and-add algorithm to accomplish this, because the hardware for this algorithm was "free". When the desired multiply operation includes large values, loops similar to those on a Classic 3000 are executed. When the operands include a constant or a small variable value, the multiply is executed in a very few of the HPPA's fast instructions. The result is better overall performance, with the proviso that the operations will be harder to spot during program debugging. The corollary is that frequency analysis is the first step in optimization.

What does this mean to you? If the compiler writers use these ideas fully, the code generated should give excellent performance. However, when you are debugging programs with the System Debugger, don't expect to see a simple MPY instruction, nor even an obvious subroutine call. You may see one of many methods of doing an integer multiply, depending on the operand values, on the compiler's intelligence, and perhaps on the level of optimization you requested!

Spectrum Instruction Format

The basic HPPA instruction format is 6 bits for the major opcode and 10 bits to specify the source registers (if any).

```
-------------------------------------------------------------
|  Opcode  |  Source2  |  Source1  |  Rest of Instruction|
-------------------------------------------------------------
   6 bits     5 bits      5 bits           16 bits
```

Source2 and Source1 select input registers for the instruction. The position of the target register (if any) can vary, but is often at the end of the instruction.

The only memory reference instructions are Load and Store; there are no instructions to add memory, as in the Classic 3000. Let's look at a typical memory reference instruction, Store Word (STW):

```
                     STW - Store Word
-------------------------------------------------------------
|  $1A      | Base-Reg | Data-Reg | S |    Offset          |
-------------------------------------------------------------
   6 bits     5 bits      5 bits    2 bits   14 bits
```

The function of STW is to store 32 bits from a specified General Register (i.e., Data-Reg) into memory. The effective byte address is computed by adding the 14-bit Offset value in the instruction to the 32-bit Base address in another General Register (i.e., Base-Reg). The computed address for a Word load or store must be at a multiple of 4 bytes, just as the address for a Half-word load or store must be at a multiple of 2 bytes.

Storage Boundaries on the Spectrum

The HPPA is a byte-address machine, but as the manual says...

> "All addressable units must be stored on their naturally aligned boundaries. A byte may appear at any address, halfwords must begin at even addresses, and words begin at addresses that are multiples of 4."

The Classic 3000 usually stores 16-bit values at even-byte addresses, like the HPPA, but does not force 32-bit values to be on 32-bit boundaries. In this matter, the two machines are incompatible. If you restore a Classic-3000 file or database onto the HPPA, the machine will not be able to use the Load and Store Word instructions to reference mis-aligned fields. Instead, it will have to use pairs of Load Halfword and Store Halfword instructions. You will use fewer instructions if you convert your files to be HPPA-aligned and re-compile your programs, but we can't say whether this improvement will even be measurable until we can run some performance tests.

If you have an MPE V file (i.e., aligned on 16-bit words) and you access it via a native-mode MPE XL program, you must be certain that your compiler is aligning fields in data buffers on 16-bit boundaries. Otherwise, your program will read and update the wrong fields. The Native-Mode compilers have options to cope with alignment issues and the details vary from compiler to compiler. This issue is well explained in the **Beyond RISC!** book from SRN.

Delayed Branches

We have seen that one of the primary goals of the HPPA is to complete the execution of a useful instruction in each machine cycle: 80 ns on the 950. Note: the "official" machine cycle of the 950 is subdivided into four sub-cycles, in order to implement the different phases of an instruction. This allows HPPA to work on several instructions at once. Thus it actually takes a minimum of three machine cycles to execute an instruction, but the effective throughput is one instruction completed every cycle. We will tend to ignore these details, except where necessary.

The branch operation is a serious stumbling block to these HPPA goals. Branch instructions are difficult to implement in one cycle because they must first compute the branch location, then actually retrieve the new instruction at that location.

The HPPA is a pipelined computer, meaning that it has a pipeline of instructions that it is preparing to execute while it is actually executing the current instruction. As long as the instructions execute in sequence, the hardware to pre-retrieve the following instruction is fairly straightforward. When we branch, we cannot have the instruction at the branch destination in our pipeline, because we don't know in advance which instruction it is.

The choice seems to be between using two cycles to execute the branch or stretching the length of the basic cycle time to allow for retrieving the branch location from memory. Neither choice seems attractive.

What the HPPA does is ingenious and profoundly disturbing: the HPPA delays the execution of the branch for one cycle. This strategy is frequently used in microcode (see the HP 3000 6x and 7x machines, for example), but this is the first time I have seen it in an official instruction set.

Here is how the Precision Architecture manual describes the concept of Delayed Branching:

> "All branch instructions exhibit the delayed branch feature. This implies that the major effect of the branch instruction, the actual transfer of control, occurs one instruction after the execution of the branch. As a result, the instruction following the branch (located in the 'delay slot' of the branch instruction) is executed before control passes to the branch destination."

When you look at assembler listings of MPE XL native-mode programs, you will sometimes see instruction sequences like this:

```
BL  opencarton      ;branch and link
NOP                 ;delay slot
```

The compilers and/or optimizers could not find anything useful to do in the cycle after the branch ("the delay slot"), so they inserted a NOP (no operation). Effectively, this sequence is a two-instruction branch.

Or, you may see an instruction sequence like this:

```
BL closecarton      ; branch
```

```
        LDW 26 ...             ; load word in the delay slot
```

The BL branch instruction is executed before the LDW instruction, **but does not take effect until one cycle later.** The LDW instruction that comes after the BL instruction is actually executed while the BL completes. The LDW in this case is probably loading one of the parameters needed by the 'closecarton' subroutine.

Let me see if I can make this a little more clear with an analogy. If you are taking an airplane trip to Minneapolis, you follow a program with instructions like these:

```
1.      book flight
2.      reserve hotel
3.      pay for ticket
4.      reserve rental car
5.      pack bags
6.      fly to Minneapolis
7.              (wasted time, read a book perhaps)
8.                        collect baggage
9.                        get rental car
10.                       check into hotel
```

Everything takes about the same time ("one machine cycle"), except for the actual flight. This takes more than one machine cycle, leaving a wasted time period when you catch up on reading or napping.

Now, imagine that you are a HPPA computer, determined not to waste a single machine cycle. How would your trip be programmed?

```
1.      book flight
2.      reserve hotel
3.      pay for ticket
4.      reserve rental car
5.      fly to Minneapolis
6.              (pack baggage during the delay slot)
7.                        collect baggage
8.                        get rental car
9.                        check into hotel
```

Using the HPPA airplane, our trip took only 9 cycles instead of 10. How did we shorten the time? By packing our bags while we are flying to Minneapolis. If this sounds like "being in two places at the same time" to you, you're right! The HPPA can work on several tasks at the same time due to the power of pipelining. The HPPA compilers try to take the last step in your program before the branch and move it to after the branch. In this way, it can be executed during the time that the branch is delayed.

Of course, there are a few catches.

If the branch is a conditional one, such as "branch if register two equals zero", then we couldn't move any instruction that might change the value of register two. We would have to find an instruction that could not have any impact on whether the branch would be taken.

Another wild and unbelievable implication of the delayed branch is spelled out very dryly in the instruction set manual, as follows:

"Consider the situation in Figure 4-2.

Figure 4-2. Branch instruction in the delay slot

```
Loc.  Instruction                                  Reference#
100   STW
104   BV r0(r7)    branch vectored to location 200        I1
108   BL r4 r0     IA relative branch to location 400     I2
10C   ADD r2,r6,r9 next instruction in linear code sequence

   ....

200   LDW   0(r3),r11   target of BV instruction I1       I3
204   ADD               next instruction, never executed!

   ....

400   LDW               target of BL instruction I2       I4
404   STW                                                 I5
```

"A taken branch instruction, I2, is executed in the delay slot of a preceding taken branch, I1. When this occurs, the first branch I1 schedules its target instruction, I3, to execute after I2, and the second branch, I2, schedules its target instruction, I4, to execute after I3. The net effect is the out-of-line execution of I3, followed by the execution of I4. Also, if I3 were to be a taken branch, its target, I5, would execute after I4, and I4 would also have been executed out of its spatial context."

How can we translate this strange computer situtation into a real life analogy? It would be as if, on our flight to Minneapolis, we were high-jacked to New York, but the high-jackers were unable to keep us from stopping in Minneapolis just long enough to lose our bags (i.e., one cycle).

When I first saw this example in the manual, I was at a loss to think of any practical use for it. But it is never wise to assume that anything in the HPPA is by accident. On further reflection, I thought I could see a way to use two branch instructions in a row to good advantage. Classic 3000 hackers are aware of the XEQ instruction, which allows you to execute another instruction, one that is not known until execution time. The HPPA does not have an XEQ instruction. But, by combining the above example with the fact that the HPPA does not distinguish between code and data as strongly as the Classic 3000 does, you might be able to produce a reasonable facsimile of the XEQ instruction.

The HPPA delayed branch is not just an intellectual curiosity that you can ignore. It has may practical implicatons. When you are using the system debugger to set a breakpoint near a branch instruction, it is very difficult to remember that the instruction after the branch will be executed before the target of the branch. Deciding where to set the breakpoint can sometimes seriously strain your imagination.

Memory Addressing

One of the goals of the HPPA project was to allow programs to address enormous amounts of memory, many times more than the current hardware technology can deliver. The memory Load and Store functions compute an effective byte address by adding an Offset value to a 32-bit Base address held in one of the 32 registers.

The effective address, however, is not a real memory address, nor is it an address in the stack Data Segment (as in the Classic 3000). It is an address within a Space. The HPPA can have thousands of spaces active and the program can access eight of them at any instant using eight hardware Space Registers. Within the memory reference instructions, the S field tells how to select the Space Register. If S is 1, 2, or 3, the instruction uses the space defined by Space Register 1, 2, or 3. If S is 0, it uses the space defined by 4 plus the upper two bits of the address in the Base-Reg value (i.e., 4, 5, 6, or 7).

Spaces are similar to Segments in the Classic 3000 architecture, but can be <u>much</u> larger. Only SR 0 has a hardware-defined purpose; it is used for the return address of interspace calls. Software conventions define SR 4 as the code space, SR 5 as the stack space, SR 6 as a space for shared data, and SR 7 for public operating system code, literals, and data. SR 1, SR 2, and SR 3 are available to the programmer as temporaries for the construction of 64-bit long pointers (i.e., pointers into any space).

Quadrants

When the Space field in the instruction is 0, the target Space Register is selected by adding 4 to the upper two bits of the 32-bit address. Two bits allows four possible values: 0 selects SR 4, 1 selects SR 5, 2 selects SR 6 and 3 selects SR 7. However, to keep the hardware simple, the entire 32-bit word is the byte address within that space, not just the 30 bits after the space selector. This is called short pointer addressing since only 32 bits are used to select both the space and the offset within the space. With long pointers, a full 32-bit word is used for the space number and another 32 bits for the offset within the space.

Because the upper two bits that select the Space Register are included as part of the address, you can only address a quarter of the potential addresses in a space using a short pointer. HP describes the restriction as follows:

> "Only one fourth of the space is directly addressable by the base register with short pointers and the region corresponds to the quadrant selected by the upper two bits. For example, if a base register contains the hex value $40020000, space register 5 is used as the space identifier and the second quadrant of the space is directly addressable."

In 32 bits, we can address the first quadrant of the space pointed to by SR4, the second quadrant of SR5, the third quadrant of SR6 and the fourth quadrant of SR7. This makes for some very large memory addresses, and can be quite surprising to an old-time 3000 programmer. To get at the other 3 quadrants of those spaces we must use a long pointer where the desired space register is not encoded in the 32-bit offset value.

Bounds checking of subroutine parameter addresses is a more complex issue on the HPPA than on the Classic 3000. On the Classic, you check a parameter address to see if it is between the DL and S registers. On the HPPA, a legitimate parameter address may not even point to the user's data space, and read/write security within a space can theoretically vary from page to page (a page is 4096 bytes). Anyone who understands this issue is invited to correspond with the author.

Pipelining and Register Interlock

In order for the HPPA to complete the execution of an instruction in each machine cycle and to have that cycle be as fast as possible, it has pipelining. It has a pipeline of instructions that it is preparing to execute while, at the same time, it is actually executing the current instruction. A major obstacle to completing an instruction in every cycle is the Memory Load operation.

What happens with memory loads on the HPPA is similar to what can happen with your checking account. You deposit a check to your account on Monday, but if you try to withdraw that amount on Tuesday, you may be blocked by the bank. They claim it takes two cycles for the check to clear.

The same thing happens with memory load instructions. It takes one cycle just to compute where in memory you want to load data from. Then it takes another cycle to retrieve that data and put it in the desired register.

The machine designers faced a choice: they could either make the basic cycle time longer to allow for both the computation of the effective address and the load from main memory. Or, they could go on to the next instruction before they had finished with the load instruction.

This second choice is what the HPPA does. When you load from memory, the pipeline has already prepared the next instruction for execution. The HPPA goes ahead and starts executing that instruction, even though it has not completed the preceding load operation.

Question: What happens if the next instruction requires the data that is being loaded?

Answer: Register Interlock.

You should not refer to the target of a Load instruction in the instruction that follows a Load instruction. If you do, you get a **Register Interlock**. This pauses the program for one cycle, allowing the load from memory into the register to complete.

The compilers and optimizers on the HPPA attempt to re-order the machine instructions to avoid Register Interlock. For example, a sequence that does Load-Store, Load-Store and causes two Register Interlocks, can be converted to Load-Load, Store-Store with no interlock delays.

```
A := B;     LDW 31 load B      LDW 31 load B
C := D;     STW 31 store A     LDW 30 load D
            LDW 30 load D      STW 31 store A
            STW 30 store C     STW 30 store C
             six cycles         four cycles
```

The first code sequence with alternating LDW and STW instructions takes two cycles longer than the second code sequence, which does both LDWs first, then both STWs.

The ability of the compilers to reorder instruction sequences to avoid register interlocks is important to attaining the theoretical speed of the HPPA.

The 930 Register Interlock is not as smart, nor as expensive, as the 950. Rather than check the type of the next instruction to see if it could contain a reference to a register, the 930 just looks for the 5-bit pattern of the register number in the position in the instruction where register numbers usually appear. Of course, if you are executing an instruction like LDIL (load immediate left), with its 21-bit constant operand, the 5-bit pattern of the register may occur in the constant operand by chance. Unfortunately, you get a register interlock anyway. The 950 avoids this trap with extra AND and OR logic.

The only reference to Register Interlock in the HP manual is on page 5-16:

> "Execution is faster if software avoids dependence on register interlocks. Instruction scheduling to avoid the need for interlocking is recommended. This does not restrict the delay a load instruction may incur in a particular system to a single execution cycle; in fact, the delay will be much longer for a cache miss, a TLB miss, or a page fault."

Register Arithmetic Vs. Stack Arithmetic

The Classic 3000 has 19 **arithmetic** instructions that operate on the top of stack values. The functions provided are Add, Subtract, Compare, Zero, Multiply, Divide, And, Not, Or and Xor.

The Classic 3000 provides Add, Subtract and Compare for 16-bit and 32-bit integers and for 16-bit unsigned integers. Zero pushes a 16-bit or 32-bit zero value onto the stack. And, Not, Or and Xor (exclusive OR) are provided for 16-bit values only. In all cases, the operands are taken off the top of the stack and the result, if any, is pushed onto the stack. All instructions compute and set Overflow, Carry and Condition Code fields in the status register, whether you need them or not.

The HPPA has similar arithmetic functions, but the operands are always 32-bit values. If you want to Add two 16-bit values together, you actually do a 32-bit Add and then manually check the result for overflow of 16 bits using another instruction. The arithmetic functions take two of the 32 general-purpose registers as input and one of the registers as output. The Multiply and Divide functions are not included in the HPPA instructions set and were discussed in detail earlier in this article.

Addition

There are numerous variations on each of the basic arithmetic functions. For example, here are the ways you can Add two registers and put the result in a third:

ADD	Add (32-bit signed addition)
ADDO	Add and Trap On Overflow
ADDC	Add with Carry
ADDCO	Add with Carry and Trap on Overflow
ADDL	Add Logical (32-bit unsigned addition)

The regular ADD instruction does not trap on integer overflow; there are variations on the instruction for trapping. Languages differ in their treatment of overflow conditions. Pascal has very precise requirements for overflow traps, while the C language explicitly says that it does not trap on overflow - the result is undefined. "With Carry" means that the carry bit from a previous add function is included in computing the answer. ADDL is an unsigned 32-bit addition, where all numbers are treated as positive values.

Nullification

On the Classic 3000, arithmetic functions adjust bits in the status register which you can then test in a branch instruction. For example, you might subtract one number from another and branch if the result is zero. On the HPPA, arithmetic instructions allow you to "nullify" the next instruction if a certain condition, such as zero result, occurs or does not occur. Rather than have numerous conditional branch instructions, the HPPA puts the conditional logic in the arithmetic instructions. This is similar to many other machines that have "skip" conditions in instructions (if something happens, skip the next instruction). Experienced programmers who have worked with the HPPA at the machine-language level report that having Nullify in so many instructions is a big aid to writing tight code.

Subtract

There are even more variations on Subtract than there are on Add:

```
SUB    Subtract      (32-bit signed subtraction)
SUBO   Subtract and Trap On Overflow
SUBB   Subtract with Borrow
SUBBO  Subtract with Borrow and Trap On Overflow
SUBT   Subtract and Trap on Condition
SUBTO  Subtract and Trap on Condition or Overflow
```

Subtract "with Borrow" is similar to Add "with Carry": it means that the borrow bit from a previous subtract operation is to be included in computing the answer. As with Add, there are versions of Subtract to trap or not on overflow. The "trap on condition" instructions give you the ability to trap to an error routine if a certain condition results after doing the subtraction. Having a trap instead of a skip on the condition means that you can test the condition in one instruction instead of two, but the usefulness of this is limited to tests that will likely abort the program when the condition occurs. These are probably included for doing efficient bounds checking of array indices.

Compare Function

COMCLR is the Compare and Clear function. The two input registers are compared and the next instruction can be conditionally skipped, based on the result. In addition, you have the ability to clear another register in the same cycle. Some day you may appreciate being able to compare and clear in the a single instruction. For example, using the Nullify field in the COMCLR instruction with a following Add Immediate Instruction, you could compare two registers and leave a zero or one in another register that reflects the result of the compare.

Logical Functions: And, Not, Or, Xor

The HPPA has basically the same logical functions as the Classic 3000, except that the operands are any of the 32-bit registers rather than the 16-bit values currently at the top of the stack. There is one unusual instruction:

```
ANDCM   And with Complement
```

ANDCM complements the value of one register, then ANDs it with another register, leaving the result in a third register. The ANDCM instruction can produce the same result as the Classic 3000's NOT instruction; ANDCM `r,r,r` flips the bits in register r, ANDs them with the register r and stores the result back in register r, effectively a NOT of register r.

Immediate Functions

The HPPA has 19 machine instructions with constant or "immediate" operands:

Opcode	Operand-Size	Function
LDO	14 bits	Load Offset
LDIL	21 bits	Load Immediate Left
ADDIL	21 bits	Add Immediate Left
ADDI	11 bits	Add to Immediate
ADDIT	11 bits	Add to Immediate and Trap On Condition
ADDIO	11 bits	Add to Immediate and Trap on Overflow
ADDITO	11 bits	Add to Immediate and Trap on Cond/Ovfl
SUBI	11 bits	Subtract from Immediate
SUBIO	11 bits	Subtract from Immediate and Trap on Overflow
COMICLR	11 bits	Compare Immediate and Clear
COMIBT	5 bits	Compare Immediate and Branch if True
COMIBF	5 bits	Compare Immediate and Branch if False
MOVIB	5 bits	Move Immediate and Branch
ADDIBT	5 bits	Add Immediate and Branch if True
ADDIBF	5 bits	Add Immediate and Branch if False
DEPI	5 bits	Deposit Immediate
VDEPI	5 bits	Variable Deposit Immediate
ZDEPI	5 bits	Zero and Deposit Immediate
ZVDEPI	5 bits	Zero and Variable Deposit Immediate

The Classic 3000 has only 10 immediate instructions and they all have an 8-bit constant operand. The Classic 3000 has MPYI and DIVI, but lacks the Deposit Immediate instructions. The HPPA has more immediate instructions, but they are all "basic" one-cycle instructions (i.e., no multiply immediate instruction).

The size of the immediate operand on the HPPA varies: 5 bits (-15 to +15) for branches and bit deposits, 11 bits (-1K to +1K) for most immediate functions, 14 bits (+8K to -8K) for Load Offset which can also add, plus Load and Add Immediate Left (they take a 21-bit operand that is shifted left to have 11 zeroes after it). Using LDIL and LDO together, you can load any 32-bit constant into any register in two machine cycles. The LDIL instruction has another interesting use: to embed statement numbers in the object program. When the compilers want to leave road markers in the code, they use LDIL to register 0, which is a null operation. The constant parameter refers back to the source code statement number or line number!

The Classic 3000 has one Add Immediate Instruction (ADDI). The HPPA has at least 6. The HPPA can add an 11-bit signed constant to a register and store the result in another register, with four varations: no trap (ADDI), trap on overflow (ADDIO), trap on one of 16 conditions such as less than zero (ADDIT), or trap on overflow or a condition (ADDITO). ADDI and ADDIO can also specify one of 16 conditions for nullifying the next instruction (i.e., never, <, =, odd, etc.). Two other instructions, ADDIBT and ADDIBF, allow you to add a 5-bit signed constant to a register, test a condition, and branch if the condition is true or false. The conditions that can be tested in the Add Immediate instructions (and in most HPPA arithmetic instructions) replace eleven separate branch instructions on the Classic 3000 (e.g., BRO, BRE, BCC, and so on).

Conclusions

The design of the Classic 3000 instruction set was heavily influenced by programmers. Their desire was to avoid the programming problems they had encountered writing systems software in Assembler on HP's earlier 2100 computer line. As such, the instruction set represented their practical feel for what would be nice to program in.

The HPPA instruction set is striking in the degree to which it is based on measurements, not feelings. Everywhere you look, it shows signs of hard engineering tradeoffs. If measurements showed that few real programs used a particular function, that function didn't have much chance of making it into HPPA. Conversely, if you find a particular function in the instruction set, it probably has a valuable use in some large class of programs (no matter how obscure it looks to you).

The Classic 3000 code is more compact and easier for people to read and write, but the HPPA code is more powerful. Not only can the HPPA instructions access vastly more memory, but they are at least a hundred times more flexible than their Classic 3000 equivalent. The HPPA instructions resemble the internal microcode of the Classic 3000 much more than they resemble the official "machine" instructions of the Classic. Because of the many options available, the HPPA depends on adroit compilers to select the best instruction for each situation.

When Hewlett-Packard announced their intention to build a RISC machine, the theoretical papers on RISC led me to expect a machine with 30 to 50 instructions. When I first looked at the HPPA, I was surprised to find over 150 instructions. As I studied HPPA, however, I started to get a feel for it. The instructions have incredible complexity and power, but they also have "reduced complexity" from a computer engineer's point of view. They are easy to implement in fast, elegant hardware. Although this strategy has shifted many complex problems from the hardware to the software, the HPPA designers appear to have provided the programmers with the power they need to solve their problems.