ALPHA Code-Generation Notes

Instruction Scheduling

A compiler's instruction scheduler reorders instructions to optimize execution performance while making sure that:

- Correct results are maintained. That is, values are still computed before they are used, and write operations to the same location remain in order.
- The operands are issued before the instruction using them is issued.
- Ideally, a value finishes being computed before the instruction that uses the value gets issued.

For example, consider the following instruction sequence:

```
I1    Defines R1 (3-cyle instruction)
I2
I3
I4
I5    Uses R1
...
In
```

The first rule above states that I1 cannot be moved beyond I5. That would yield incorrect results.

The second rule exists because modern architectures do not always complete an instruction in the same cycle that the instruction is issued. (Multiple-cyle instructions include floating-point operations, loads, integer multiplies.)

So, in our example, an Alpha 21164 (EV5) processor must wait for the 3-cycle instruction I1 to complete before issuing I5.

However, if I5 *is* issued before the 3 cycles complete, on Alpha processors there is no problem. In this case, the processor stops issuing instructions and waits. This is an example of a resource not being available. An instruction can only be issued if the operands for the instruction are available.

A notable characteristic of the 21164 processor is that it issues instructions in order. So if one instruction stops because its operands are not available, even if the next instruction *has* its operands available, it cannot be issued until the previous instruction is issued. (Instructions do not necessarily *complete* in order, but they must *begin* in order.)

Multiple Issues -- Pipelines

The 21164 processor can issue 4 instructions at a time. It does this by viewing a 16-byte chunk of instructions (each instruction is 4 bytes). The address of the chunk must be aligned on a 16-byte boundary. The address of the first instruction in a loop, for example, should be on a 16-byte boundary because you want to get as much efficiency out of the chunk as you can if you are doing something repetitively.

Although there are four instructions in the chunck, execution appears sequential because the hardware goes through these instructions in order using the following architectural units:

The 21164 processor contains an Integer Execution Unit (Ebox) and a Floating-Point Execution Unit (Fbox). Each of these execution units contain two execution pipelines that perform the actual arithmetic operations:

- Ebox-- two integer execution pipelines:
  - E0
  - E1
- Fbox -- two floating-point execution pipelines:
  - FA
  - FM

The best-case scenario in the four-instruction chunk is one instruction using E0, one using E1, one using FA, and one using FM. All instructions will issue simultaneously, assuming all their operands are available.

Pipeline Stages and Instruction Issue

The 21164 pipeline divides instruction processing into four static and a number of dynamic stages of execution.

The first four stages are executed in the Instruction Fetch/Decode Unit (Ibox) and consist of:

1. Instruction fetch
2. Buffer and decode
3. Slotting
4. Issue-check logic

These stages are static in that instructions may remain valid in the same pipeline stage for multiple cycles while waiting for a resource or stalling for other reasons. A pipeline stall can occur while zero instructions issue, or while some instructions of a set of four issue and the others are held at the issue stage. A pipleine stall implies that a valid instruction is presented to be issued but cannot proceed. Dynamic stages (executed in the Ebox and Fbox) always advance the state and are unaffected by any stall in the pipeline.

Upon satisfying all issue requirements, instructions are issued into their slotted pipeline. After issuing, instructions cannot stall in a subsequent pipeline stage. The issue stage is responsible for ensuring that all resource conflicts are resolved before an instruction is allowed to continue. The only means of stopping instructions after the issue stage is an abort condition. See the *21164 Hardware Reference Manual* for a description of aborts and exceptions.

Instruction-Class Definition

The Alpha instructions are divided into classes. The instructions in each class can be executed only in certain execution pipelines. For example, the LD (load) class is all load instructions except those having to do with locking; LD instructions execute in either E0 or E1. Similarly, the ST (store) class is all store instructions except those having to do with locking; ST instructions execute in E0.

The following table provides the 21164 scheduling and multiple-issue rules. These rules are performance-related only; that is, there are no functional dependencies related to scheduling or multiple issuing. The rules are defined in terms of instruction classes. The table specifies all of the instruction classes and the pipeline that executes each class. With a few additional rules, the table provides the information necessary to determine the functional resource conflicts that determine which instructions can issue in a given cycle.

(If your browser does not support tables, click here.)

Class Name	Pipeline	Instruction List
LD	E0 or E1	All loads except LDx_L
ST	E0	All stores except St_C
MBX	E0	LDx_L, MB, WMB, STx_C, HW_LD-lock, HW_STcond, FETCH
RX	E0	RS, RC
MXPR	E0 or E1 (depends on the IPR)	HW_MFPR, HW_MTPR
IBR	E1	Integer conditional branches
FBR	FA	Floating-point conditional branches
JSR	E1	Jump-to-subroutine instructions: JMP, JSR, RET, or JSR_COROUTINE, BSR, BR, HW_REI, CALLPAL
IADD	E0 or E1	ADDL, ADDL/V, ADDQ, ADDQ/V, SUBL, SUBQ, SUBQ/V, S4ADDL, S4ADDQ, S8ADDL, S8ADDQ, S4SUBL, S4SUBQ, S8SUBL, S8SUBQ, LDA, LDAH
ILOG	E0 or E1	AND, BIS, XOR, BIC, ORNOT, EQV
SHIFT	E0	SLL, SRL, SRA, EXTQL, EXTLL, EXTWL, EXTBL, EXTQH, EXTLH, EXTWH, MSKQL, MSKLL, MSKWL, MSKBL, MSKQH, MSKLH, MSKWH, INSQL, INSLL, INSWL, INSBL, INSQH, INSLH, INSWH, ZAP, ZAPNOT
CMOV	E0 or E1	CMOVEQ, CMOVNE, CMOVLT, CMOVLE, CMOVGT, CMOVGE, CMOVLBS, CMOVLBC
ICMP	E0 or E1	CMPEQ, CMPLT, CMPLE, CMPULT, CMPULE, CMPBGE
IMULL	E0	MULL, MULL/V
IMULQ	E0	MULQ, MULQ/V
IMULH	E0	UMULH
FADD	FA	Floating-point operations, including CPYSN and CPYSE, except multiply, divide, and CPYS
FDIV	FA	Floating-point divide
FMUL	FM	Floating-point multiply
FCPYS	FM or FA	CPYS, not including CPYSN or CPYSE
MISC	E0	RPCC, TRAPB
UNOP	none	UNOP

E0 = Ebox pipeline 0, E1 = Ebox pipeline 1, FA = Fbox add pipeline, FM = Fbox multiply pipeline, UNOP is LDQ_U R31,0(Rx)

All instructions use one of the pipelines. The scheduler would like to order the instructions so that the four-instruction chunk contains, ideally, two integer operations and two floating-point operations.

However, that is rarely the case.

Slotting

How does the hardware take the instructions out of the chunk and feed them to the function unit? The compiler developer needs to know something about this.

The slotting function in the Instruction Fetch/Decode Unit (Ibox) determines which instructions will be sent forward to attempt to issue. The slotting function detects and removes all static functional resource conflicts. The set of instructions output by the slotting function will issue if no register or other dynamic resource conflict is detected in stage 3 of the pipeline.

The slotting algorithm follows:

1. Starting from the first (lowest addressed) valid instruction in the 16-byte chunk in stage 2 of the 21164 Ibox pipeline, attempt to assign that instruction to one of the four pipelines (E0, E1, FA, FM). If it is an instruction that can issue either in E0 or E1, assign it to E0. However, if one of the following is true, assign it to E1:
    - E0 is not free and E1 is free.
    - The next integer instruction in this 16-byte chunk can issue only in E0. (The next integer instruction is one that can issue in one or both of E0 or E1, but not FA or FM)
2. If the current instruction is one that can issue in either FA or FM, assign it to FA unless FA is not free. If it is an FA-only instruction, it must be assigned to FA. If it is an FM-only instruction, it must be assigned to FM. Mark the pipeline selected by this process as taken and resume with the next sequential instruction. Stop when an instruction cannot be allocated in an execution pipeline because any pipeline it can use is already taken.

The slotting logic does not send instructions forward out of logical instruction order because the 21164 always issues instructions in order. The slotting logic also enforces the special rules in the following list, stopping the slotting process when a rule would be violated by allocating the next instruction an execution pipeline:

- An instruction of class LD cannot be simultaneously issued with an instruction of class ST.
- All instructions are discarded at the slotting stage after a predicted-taken IBR or FBR class instruction, or a JSR class instruction.
- After a predicted not-taken IBR or FBR, no other IBR, FBR, or JSR class can be slotted together.
- The following cases are detected by the slotting logic:
    - From lowest address to highest within the 16-byte chunk, with the following arrangement:

      ```
      I-instruction, F-instruction, I-instruction, I-instruction
      ```

      I-instruction is any instruction that can issue in one or both of E0 or E1. F-instruction is any instruction that can issue in one or both of FA or FM.
    - From lowest address to highest within the 16-byte chunk, with the following arrangement:

      ```
      F-instruction, I-instruction, I-instruction, I-instruction
      ```

  When this type of case is detected, the first two instructions are forwarded to the issue point in one cycle. The second two are sent only when the first two have both issued, provided no other slotting rule would prevent the second two from being slotted in the same cycle.

Coding Guidelines

Code should be scheduled according to latency and function unit availability. This is good practice in most RISC architectures. Code-alignment and the effects of split-issue should be considered. (Split issue is the situation in which not all instructions sent from the slotting stage to the issue stage issue. One or more stalls result.)

The slotting and issue stages do not shift new instructions into empty slots at the end of each cycle. Not including these shifts eliminates key critical paths and enables a considerably faster CPU clock speed. Compilers and assembly language programmers can insert no-ops to minimize, and in most cases to eliminate, any negative performance effect. To facilitate this process, the 21164 handles three different kinds of no-ops:

- NOP (integer no-op)
- FNOP (floating-point no-op)
- UNOP (universal no-op)

NOP (BIS R31,R31,R31) can issue in either integer pipeline (E0 or E1).
FNOP (CPYS F31,F31,F31) can issue in either floating-point pipeline (FA or FM).
Both NOP and FNOP allow the computer to delay the issue of an instruction, creating more opportunities for dual issue by preventing the case where a pair of instructions in which the second is data-dependent on the first are slotted together even though they cannot issue in the same cycle.

UNOPs (LDQ_U R31,...) are detected by the slot logic. They are invalidated in the slot stage so that they do not require an issue slot in either pipeline. UNOP allows compilers to align instructions without using pipeline issue slots unnecessarily.

For example, align the start of a loop so that its address is a multiple of 16.

Instruction Latencies

After slotting, instruction issue is governed by the availability of registers for read or write operations, and the availability of the floating divide unit and the integer multiply unit. There are producer-consumer dependencies (also know as write-after-write conflicts), and dynamic function unit availability dependencies (integer multiply and floating divide).

The latency to produce a valid result for most instructions is fixed. The exceptions are loads where the data is not in the cache, floating-point divides, and integer multiplies. See the *21164 Hardware Reference Manual* for a table showing latencies for each instruction class.

Issue Rules

All instructions sent to the issue stage (stage 3) by the slotting stage (stage 2) are issued subject to the set of rules specified in the *21164 Hardware Reference Manual*. If issue is prevented for a given instruction at the issue stage, all logically subsequent instructions at that stage are prevented from issuing automatically. The 21164 only issues instructions in order.

Replay Traps

To make the hardware really fast, you do not always wait for every possible condition to occur before continuing processing. The hardware is optimistic; it lets some things go on because if it stopped to wait and see if something could go on or not, processing would slow down. Of course, at some point, it has to figure out if something it did was legitimate or not. If it is not, the processor has to stop and back up. It does so as follows:

The processor keeps track of all the instructions it might have issued since that point where it made the optimistic decision and keeps track of the status of registers used. If an instruction cannot be executed because of insufficient resources (or some other reason), these instructions trap and the Ibox restarts their execution from the beginning of the pipeline. This is called a *replay trap*.

If the compiler's instruction scheduler determines that one of these replay traps can happen, and keeps it from happening, the program will run faster.

The compiler needs to control the hardware sufficiently accurately so that the hardware will run as fast as possible, which means replay traps should be avoided. This is not always possible. The compiler might know that a replay trap might happen, but have no way to keep it from happening. (You cannot avoid it, for example, if you are copying from an integer to floating-point register.)

The compiler's instruction scheduler cannot see replay traps, but it can do a few things to help avoid them. For example, the instruction scheduler should avoid storing into a particular location and then reloading immediately from that same location. To increase efficiency, the compiler can put a 3-cycle delay between the store and the load.

More Information

For more information on instruction scheduling, see the *21164 Hardware Reference Manual* or the *Alpha Implementations and Architecture* reference and guide.

A basic block instruction scheduler is available for the 21064-AA (EV4) processor. It is the file `instr-sched.c` on the Alpha Freeware CDROM. The Digital UNIX FAQ has information on how to obtain this CDROM. From the FAQ site, search on "freeware".

Heuristics for Inlining

This note describes a new heuristic algorithm that Digital product compilers use to inline functions.

Primary Factors

The primary factors for inlining decisions are:

- How many bytes of code would be added by inlining this call?

  This increase in size is calculated by subtracting the estimated size of the call sequence from the estimated size of the called routine.

- How often is this call site executed?

  This can be judged fairly accurately when run-time feedback is used. Without run-time feedback, only an estimate of the loop nesting depth of the call site is available. Therefore, when feedback is available, the frequency of execution heavily influences the inlining decisions. When it is not available, frequency is a minor factor.

Overall, the new algorithm first filters out all routines and call sites that either cannot be inlined or are obviously not profitable to inline (for example, they use varargs, are too large, and so on).

Next, it assigns each call site worth considering a priority based on the code size and frequency factors described above. The inline candidates are then kept in a sorted list, with the best candidates first.

The inlining is performed on a walk of the sorted list. The first call is removed from the list. Some additional heuristic factors are evaluated for that call, and a final decision is made to inline it or not. If the call is inlined, then the priority of any calls to the caller are re-evaluated, and they are possibly moved to a new location in the sorted list. The inline expansion can also contain calls. If so, these calls are prioritized and added to the list.

Secondary Factors

In addition to the size and frequency factors described above, these secondary factors also affect the inlining decision:

- How many times does the called routine refer to a parameter whose corresponding argument at the call site is a constant?

- Estimate the number of registers in use across the call site plus the peak number of registers needed by the inline code (separately for each register class). How much does the total number of registers for a register class exceed the number of available registers in that class?

The weighting of the various factors is determined by the mode of inlining requested by the front end in the arguments to GEM_CO_COMPILE_MODULE. The simplest way to understand the new heuristics is to view the inlining mode as selecting an approximate number of instructions by which any call can be expanded by inlining. The factors other than size are adjustments that raise or lower that number.

Two Levels Of Inlining

There are now two levels of automatic inlining:

- The lower level of inlining limits the increase to a small number of instructions per call (about 2 to 8). If the call is not in any loop, the called routine uses too many registers, or the call is in a LANDC, then the increase is even more limited -- perhaps even requiring a space savings before inlining. If the call uses constant parameters, then a slightly larger increase is allowed.

- The higher level of inlining allows a larger number of instructions to be added per call (about 20 to 25). The adjustments for register usage, constant parameters, and so on, are also given different weights.

Measured Results

The effects of the new heuristics on SPEC benchmarks have been measured.

Compilers with the old heuristics were compared to compilers with the conservative setting of the new heuristics:

- The lower level of inlining reduces SPECfp92 by a negligible fraction of a percent, and reduces SPECint92 by a little over 1%, mostly because of LI. LI uses feedback, but the conservative-mode heuristic still has limits on code-size growth that prevent some useful inlining; therefore, LI lost 6.6%.

Compilers with the old heuristics were also compared to compilers with the aggressive setting of the new heuristics:

- The new aggressive mode increases SPECfp92 by a negligible fraction of a percent, and increases SPECint92 by almost 1%, mostly because of GCC, which was about 4% faster with 2.3% smaller code.

A comparison of the generated code size for these compilers is not yet available as of this writing.

Front-End Interface

For front-end control over inlining, Digital product compilers provide an inlining mode argument to GEM_CO_COMPILE_MODULE, two flags in the routine block node, and a flag on INITCALL tuples. They are:

(If your browser does not support tables, click here.)

Inline Mode	Meaning
zero (default)	Select an inlining mode based on the optimization level as follows: • O0 = NONE • O1 - O3 = MANUAL • O4 - O5 = CONSERVATIVE
NONE	Do no inlining.
MANUAL	Inline calls to routines marked INLINE_PREFERRED unless those routines are very large.
CONSERVATIVE	Perform manual inlining, and also inline other calls that are likely to improve performance without significantly increasing code size. For backward compatibility the old AUTOMATIC mode is now defined as a synonym for CONSERVATIVE.
AGGRESSIVE	Perform manual inlining, and also inline any other calls that are likely to improve performance.
ALL	Inline every call that can be inlined.

In addition, the DO_NOT_INLINE flag on the routine and the MUST_NOT_INLINE flag on the INITCALL tuple turn off inlining of that routine or call regardless of the inline mode.

New Command-Line Features

To select the lower or higher level of automatic inlining, users require a new command-line syntax.

On Digital UNIX (Alpha) the suggested syntax is:

```
-inline size        To select the CONSERVATIVE inlining mode
-inline speed       To select the AGGRESSIVE inlining mode
```

On OpenVMS systems, the suggested syntax is:

```
/OPTIMIZE=INLINE=SIZE
/OPTIMIZE=INLINE=SPEED
```

The CONSERVATIVE mode is recommended as the default level of automatic inlining on all platforms.

Heuristics for Loop Unrolling

For optimization level 3 or higher, the Digital product compilers attempt to unroll certain loops to minimize the number of branches and group more instructions together to allow efficient overlapped instruction execution (instruction pipelining). The best candidates for loop unrolling are innermost loops with limited control flow.

As more loops are unrolled, the average size of basic blocks increases. Loop unrolling generates multiple loop code iterations in a manner that allows efficient instruction pipelining.

The loop body is replicated a certain number of times, substituting index expressions. An initialization loop may be created to align the first reference with the main series of loops. A remainder loop may be created for leftover work.

The number of times a loop is unrolled can be determined by the optimizer or the user can specify the limit for loop unrolling using a command line switch (-unroll n for the Digital UNIX DEC C compiler). Unless the user specifies a value, the optimizer unrolls a loop 4 times for most loops or 2 times for certain loops (large estimated code size or branches out the loop).

Software Pipelining

The Digital product compilers enable software pipelining and additional software dependence analysis when optimization level 5 is specified, which in certain cases improves run-time performance.

Loop unrolling is constrained in that it cannot schedule across iterations of a loop. Because software pipelining can schedule across loop iterations, it can perform more efficient scheduling that eliminates instruction stalls within loops by rearranging instructions between different unrolled loop iterations to improve performance.

For example, if software dependence analysis of data flow reveals that certain calculations can be done before or after that iteration of the unrolled loop, software pipelining reschedules those instructions ahead of or behind that loop iteration, at places where their execution can prevent instruction stalls or otherwise improve performance.

Loops chosen for software pipelining:

- Are always innermost loops (those executed the most)
- Do not contain branches or procedure calls

By modifying the unrolled loop and inserting instructions as needed before and/or after the unrolled loop, software pipelining generally improves run-time performance, except for cases where the loops contain a large number of instructions with many existing overlapped operations. In this case, software pipelining may not have enough registers available to effectively improve execution performance, and run-time performance using optimization level 5 may not improve compared to using level 4.

To determine whether using level 5 benefits a particular program, time program execution for the same program compiled at levels 4 and 5. For programs that contain loops that exhaust available registers, longer execution times may result with level 5.

Loop Unrolling Example

[Note: Cycle times in this example are for the 21064 processor. 21164 processors are faster.]

To understand the issues in compiling efficient code for Alpha processors, consider the following C language code fragment:

```
float variance(float a[100]) {
   int i;
   float var = 0.0;
   for  (i = 0; i < 100; i++)
         var = var+a[i]*a[i];
         return var/100;
   }
```

This code computes the variance of a distribution given a mean of 0. The code is a simple loop that involves:

1. One load operation from data in memory
2. An add
3. A multiply

It also contains integer bookkeeping code that supports the repetition of the loop and the computation of the address to load.

The objective of the compiler is to generate code that executes as fast as possible. This can be done in two ways:

- The compiler can eliminate redundant operations, such as loading values from memory. In this case, it loads the value of a[i] only once, even though it is used twice in the expression.

- The compiler attempts to keep each of several function units busy with useful work at all times. This objective is not possible because there is one load for each iteration of the loop, while there are two operations (an add and a multiply) that use the floating-ponit function unit. Still the compiler tries to keep each of the function units as busy as possible.

The application of typical compiler optimization techniques to our sample code above might produce the following generated Alpha instructions (The calling-sequence code has been eliminated to simplify the description):

```
L$2:    FCLR    var                 # var = 0
        CLR     i                   # i = 0

L$3:    S4ADDQ  i,a,r2              # address(a[i]) = address(a) + 4*i
        ADDQ    i,-100,r1           # Check loop exit; compute i-100
        BGE     r1,L$5              # Exit loop if i-100 >= 0
        LDS     f1,(r2)             # Load a[i]
        ADDL    i,1,i               # Increment i
        MULS    f1,f1,f1            # Compute a[i]*a[i]
        ADDS    var,f1,var          # var = var + a[i]*a[i]
        BR      L$3                 # Repeat loop

L$5:    LDS     f1, literal 100     # Load literal 100
        DIVS    var,f1,f0           # var/100.0
        RET     r26                 # Return result
```

Special-case instuctions such as S4ADDQ are generated to replace both a Shift instruction and an Add instruction for calculating data addresses.

At a glance, this looks like well-optimized code for the Alpha processor. However, this naively optimized code is not the fastest possible code for the sample loop.

Current implementations of the Alpha processor can start two instructions in each clock cycle. Unlike previous processors, these instructions do not complete before the next instructions are begun. When the processor needs data computed by an instruction that has not yet completed, the processor waits. The program will execute faster if the code can be reorganized so the instruction that computes a value is separated from the instruction that uses the value.

So, although optimized, the generated code above still wastes processing cycles. It does not use the Alpha processor effectively since the instructions can process only one thing at a time:

- The loop is represented as a direct translation of the C *for* loop, where the test for loop termination occurs at the beginning of the loop, occupying at least three cycles of execution.
- The load instruction takes three cycles to execute; however, there is only one instruction between it and the multiply that uses the loaded value. This wastes one cycle.
- The multiply takes six cycles and is followed directly by the floating add instruction, wasting five cycles. Fortunately, the floating add value is not used at the end of any iteration of the loop except the last, so most of the time there is no delay.
- In total, 6 of the 13 cycles are wasted; therefore, about half the machine is wasted.

To overcome the limitation of processing only one thing at a time and to use the Alpha processor more effectively, a compiler would rewrite the original sample loop to be equivalent to the following C code fragment:

```
float variance(float a[100]) {
    int i;
    float var = 0.0;
    for (i=0; i<100; i=i+4) {
        var = var + a[i]*a[i];
        var = var + a[i+1]*a[i+1];
        var = var + a[i+2]*a[i+2];
        var = var + a[i+3]*a[i+3];
        }
        return var/100;
}
```

And the following highly optimized code would be generated:

```
L$2:    FCLR    var                 # var = 0.0
        CLR     i                   # i = 0

L$3:    LDS     f1,(r16)            # Load a[i]
        ADDL    i,4,i               # Increment for 4 iterations
        LDS     f10, 4(r16)         # Load a[i+1]
        ADDQ    i,-100,r2           # Prepare for check for end of loop
        LDS     f11, 8(r16)         # Load a[i+2]
        MULS    f1,f1,f1            # Compute a[i]*a[i]
        MULS    f10,f10,f10         # Compute a[i+1]*a[i+1]
        MULS    f11,f11,f11         # Compute a[i+2]*a[i+2]
        ADDS    var,f1,f0           # var + a[i]*a[i]
        LDS     f1, 12(r16)         # Load a[i+3]
        LDA     r16, 16(r16)        # Increment pointer to a
        MULS    f1,f1,f1            # Compute a[i+3]*a[i+3]
        ADDS    f0,f10,f0           # var + a[i]*a[i] + a[i+1]*a[i+1]
        LDS     f1, literal 100     # Load literal 100 for division
        ADDS    f0,f11,f0           # var + ... + a[i+2]*a[i+2]
        ADDS    f0,f1,var           # var = var + ... + a[i+3]*a[i+3]
        BLT     r2,L$3              # Repeat loop if i < 100
        DIVS    var,f10,f0          # var/100.0
        RET     r26                 # Return result
```

The compiler notices that the loop is being executed a fixed number of times, so the test for loop termination can be executed at the end of the loop. Furthermore, the compiler *unrolls* the loop by making four copies of the loop body available to be executed on each iteration of the loop. This yields multiple operations that allow overlapped execution by choosing an advantageous instruction order.

In this case, the number of times the loop is unrolled evenly divides the number of iterations. Loop unrolling is also applicable when this is not the case or the number of iterations is not known; however, the generated code is more complex. Loop unrolling is not performed if the loop is so large that the code size would be excessive.

The compiler also noticed that the multiplications can be overlapped using the pipelined nature of the Alpha processor. In fact, some of the multiplies are overlapped with additions. The compiler places the integer operations for updating the loop counter and computing addresses into positions within the code that allow execution at the same time as the floating-point operations. Therefore, the cost of this loop after optimization is about 29 cycles for four iterations of the original loop (or about seven cycles per original iteration). The code is nearly twice as fast as the naively optimized code.

As a footnote, you might have noticed an odd instruction in the highly optimized generated code above:

```
LDS     f1, literal 100   # Load literal 100 for division
```

This instruction to load the constant 100 is within the loop, even though the constant is used after the loop. This is an example of speculative execution.

Valuable Peephole Optimizations

Compilers can perform peephole optimizations to reduce cycle time. Peephole optimizations scan generated code for instruction patterns that can be collapsed into more efficient sequences.

Here are some peephole optimizations performed by the DEC C compiler or the lcc Alpha port:

- Scans for the following sequences in add and subtract instructions (ADDL, ADDQ, S4ADDL, S4ADDQ, S8ADDL, S8ADDQ SUBL, SUBQ, S4SUBL, S4SUBQ, S8SUBL, S8SUBQ):

  ```
  ADDL   R1, $31, R2
  ADDL   R2, R3,  R4
  ```

The optimization collapses these two instructions into one:

```
ADDL   R1, R3,  R4
```

The peephole optimizer checks that R2 in the example is used only within the block containing the code sequence.

- Scans for the following sequence, with no intervening instructions:

```
STL R1, 0(t1)
LDL R2, 0(t1)
```

This sequence is optimized to:

```
STL R1, 0(t1)
BIS R1, R31, R2
```

And if R1 and R2 are the same, the sequence is optimized to:

```
STL R1, 0(t1)
```

- Scans for the following sequence, with no intervening instructions that use R2:

```
bis   R1,R1,R2
cmpeq R2,R3,R2
```

R3 can also be an 8-bit constant.

This sequence is optimized to:

```
cmpeq R1,R3,R2
```

- Scans for:

```
addl $9,1,$18
subl $9,$18,$19
```

This sequence is optimized to:

```
addl $9,1,$18
lda  $19,-1($31)
```

This optimization is extensible to other arithmetic operations that use common subexpressions that yield identities:

($9 - $18 = $9 - ($9 + 1) = $9 - $9 - 1 = 0 - 1)

Peephole optimizations are done before and after instruction scheduling.

Speculative Scheduling Facilities

The Loop-Unrolling Note contains an unusual instruction placement in the highly-optimized generated code: the instruction to load the constant 100 is *within* the loop, even though the constant is used *after* the loop. The constant is loaded 100 times even though it used only once.

Can't we improve the program by moving the load out of the loop? No! The compiler noticed that it had a free place to put an instruction, so it put the load there. The load can be executed at virtually no cost. Placing the load after the loop would add another three cycles to the execution of the subroutine.

This is a simple example of *speculative execution* -- executing instructions with values that might not be used, to improve the performance when the values are used.