digital

Technical Support

Association of Software & Application Partners

Code-Generation Notes

Code-Generation Notes

The following notes suggest how compilers targetted for Alpha processors can generate instruction sequences that best implement typical language constructs. The notes show how to take full advantage of the design features of the Alpha architecture.

The reader is assumed to be familiar with the Alpha instruction set and assembler formats.

List of Notes

- How to Load Constants
- Code Generation for LOAD Operations
- Code Generation for Unsigned Arithmetic Operations
- Code Generation for STORE Operations
- Loading and Storing Fields
- IEEE Floating-Point Implementation
- How to do Integer Multiply
- How to do Integer Divide
- Setting Up Stack Frames
- Instruction Scheduling
- Heuristics for Inlining
- Heuristics for Loop Unrolling
- Valuable Peephole Optimizations
- Speculative Scheduling Facilities

About the Examples in These Notes

The examples in these notes were obtained using the Digital UNIX DEC C compiler with the following command line:

```
cc -migrate -O4 -source_listing -machine_code -c filename.c
```

The `dis` utility was then run against the object module to produce a more readable .lst file:

```
dis -S filename.o > filename.lst
```

How to Load Constants

The Alpha chip has 64-bit registers, but there is no direct way (no instruction) to load a 64-bit or 32-bit constant. So how is this best achieved?

Let's take a look at loading constants into a register, starting with 8-bit constants and working up to 64-bit constants:

8-bit Constants

If the constant is small enough (value < 256), the *immediate* instructions can be used to perform the load. These instructions have an 8-bit immediate field as their second operand.

Immediate instructions have the format:

```
{name} Ra,L,Rc
```

Where L is the immediate operand -- a 1-byte literal number.

All immediate instructions handle their immediate operand as a 64-bit, zero-extended number between 0 and 255, inclusive.

The contents of Ra are operated on by L, and the sign-extended result is placed in target register Rc.

Example:

```
SLL $17,32,$17   # Shift contents of Register 17 32 bits left
```

The complete set of immediate instructions in the Alpha architecture follow:

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

Mnemonic	Operation
ADDL	Longword Add
SxADDL	Scaled Longword Add
ADDQ	Quadword Add
SxADDQ	Scaled Quadword Add
CMPxx	Integer Signed Compare
CMPUxx	Integer Unsigned Compare
MULL	Longword Multiply
MULQ	Quadword Multiply
UMULH	Unsigned Quadword Multiply High
SUBL	Longword Subtract
SxSUBL	Scaled Longword Subtract
SUBQ	Quadword Subtract
SxSUBQ	Scaled Quadword Subtract
AND	Logical Product
BIC	Logical Product with Complement
BIS	Logical Sum (OR)
EQV	Logical Equivalence (XORNOT)
ORNOT	Logical Sum with Complement
XOR	Logical Difference
CMOVxx	Conditional Move Integer
SLL	Shift Left Logical
SRA	Shift Right Arithmetic
SRL	Shift Right Logical
CMPBGE	Compare Byte
EXTxx	Extract Byte
INSxx	Byte Insert
MSKxx	Byte Mask
ZAPx	Zero Bytes

16-bit Constants

Signed 16-bit constants can be loaded using the LDA (Load Address) instruction.

This instruction has the format:

```
LDA    Ra,C(Rb)
```

The LDA instruction adds Rb to the sign-extended 16-bit displacement (constant C), and writes the 64-bit arithmetic result to Ra:

```
Ra = Rb + C
```

A signed 16-bit constant C ($-2^{15} <= C <= 2^{15} - 1$) can be loaded directly with an LDA instruction using Register 31 (which is always 0) as Rb, and the signed 16-bit constant as the displacement.
For example, to load the constant 32767, generate the following:

```
LDA Ra,32767($31)
```

The assembler accepts LDA $Ra,C as shorthand for LDA $Ra,C($31).

32-bit Constants

Signed 32-bit constants can be loaded using the LDA (Load Address) and LDAH (Load Address High) instructions.

The LDAH instruction has the format:

```
LDAH   Ra,C(Rb)
```

Rb is added to 2^{16} times the sign-extended 16-bit displacement (constant C), and written to the 64-bit arithmetic result to Ra:

```
Ra = Rb + (2^16 * C)
```

Consider a 32-bit constant C:

```
C:

|<----------------H----------------->|<----------------L---------------->|

+ - + - - - - - - - - - - - - - - - + - + - - - - - - - - - - - - - - - +
| 0 |                               | 0 |                               |
+ - + - - - - - - - - - - - - - - - + - + - - - - - - - - - - - - - - - +
 31                        . . .   16  15  14              . . .   2 1 0
```

This 32-bit constant can be loaded by generating an LDA to load the low-order 16 bits (L), followed by an LDAH to load the high-order 16 bits (H); register 31 ($31) always contains 0:

```
LDA  rx,L($31)
LDAH rx,H(rx)
```

The above solution works only if the high-order bit of L is 0.

Special Cases:

If the high-order bit of L (Bit 15) is 1, the LDA will sign-extend and thereby subtract 1 from the high-order 16 bits (H). To compensate for this, adjust H by adding +1:

```
LDA  rx,L($31)
LDAH rx,H+1(rx)
```

This method must be modified if the value of H is 0x7FFF, since 0x7FFF + 1 yields 0x8000, which will sign-extend and yield a negative number. So, in this case, perform the compensating addition as a separate instruction; this solution will add positive quantities to positive quantities and yield a positive number:

```
LDA  rx,L($31)
LDAH rx,H(rx)
LDAH rx,1(rx)
```

In fact, there is a whole class of 32-bit constants that require two LDAH instructions, of which the previous example is the first member.

Consider the class of 32-bit constants with the high-order 16 bits (H) in either of the following ranges:

```
2**15 - 1 < H < 2**16 - 2

        or

-2**15 > H > 2**16
```

Because LDAH uses a signed displacement, it can only load displacements up to 32767 (or -32768).

But what if, for example, you need to perform the following operation:

```
LDAH $1,40000($2)
```

This can happen if you are loading an unsigned 32-bit quantity or a generalized bit pattern, or if you are loading a 64-bit constant by chopping it in half and then loading the 32-bit parts (and shifting one 32-bit pattern so that it becomes the high part of a 64-bit pattern).

You can load such a constant using two LDAH instructions:

```
LDAH rx,D-Md ($31)
LDAH rx,Md(rx)
```

Where:

- *D* is the displacement you want to load.
- *Md* is the maximum displacement that LDAH supports: 32767 or -32768, depending on the sign of D.

For example, to load a constant that has an H value of 40000, generate the following:

```
LDAH $1,7233($31)
LDAH $1,32767($1)
```

The displacements 65535 and 65536 are special cases. Because LDAH has a maximum displacement of +32767, and 2*32767 = 65534, a third LDAH is required to load these values:

```
LDAH $1,32767($31)
LDAH $1,32767($1)
LDAH $1,1($1)       # or LDAH $1,2($1) to get 65536
```

This looks similar to the case of 0x7FFF (which is 2**15-1, or 32767) + 1. But 0x7FFF+1 is nothing more than the first example of this class of constant. Reverse the order of the LDAH instructions, and substitute 32767 for 0x7FFF, and this becomes more evident:

```
LDAH $1,1($31)
LDAH $1,32767($1)
```

Is equivalent to:

```
LDAH $1,0x7FFF($31)
LDAH $1,1($1)
```

Summary:

In general, solutions to the problem of loading 32-bit constants fall into three categories:

- Generate more complicated instruction streams (as shown above).
- Use specialized instructions, such as ZAP and ZAPNOT, and byte insert instructions (INSxx).
- Create a static variable initialized to the constant value you want, and use a LDQ or LDT instruction to load its value into the target register. This method is recommended if more than 6 instructions would be required to form the constant using other means.

Examples -- Code Generation for 32-bit Constants:

1. *Source:*

   ```
   int result = 0;

   void load1600()
   {
       result = 1600;
   }
   ```

   Generated Code:

   ```
   LDA    $0, 1600($31)
   STL    $0, result
   ```

2. *Source:*

   ```
   int result = 0;

   void load33000()
   {
       result = 33000;
   }
   ```

   Generated Code:

   ```
   LDAH    $0, 1($31)
   LDA     $0, -32536($0)
   ```

```
STL     $0, result
```

3. *Source:*

```
int result = 0;

void load67000()
{
    result = 67000;
}
```

Generated Code:

```
LDAH    $0, 1($31)
LDA     $0, 1464($0)
STL     $0, result
```

Listing of generated code for above examples.

64-bit Constants

Use the following technique for loading 64-bit constants if it can be accomplished in no more than three instructions; otherwise, load from memory (LDQ instruction).

Signed 64-bit constants can be loaded in much the same way that 32-bit constants are loaded.

Consider the 64-bit constant to be comprised of two 32-bit constants. Then:

1. Use the LDA (Load Address) and LDAH (Load Address High) instructions, as required, for each 32-bit part.

2. Use shift and add (or shift and LDA) instructions to construct the final 64-bit value.

Examples -- Code Generation for 64-bit Constants:

[Note: The l at the end of the number 32768l is a lowercase l for `long`.]

1. *Source:*

```
#define BIG (32768l*32768l*32768l)

long result = 0;

void foo()
{
 result = BIG;
}
```

Generated Code:

```
LDAH    $0, 1($31)
SLL     $0, 45, $0
STQ     $0, result
```

2. *Source:*

```
#define BIG (32768l*32768l*32768l)

long result = 0;

void foo1()
{
result = (3*BIG);
}
```

Generated Code:

```
LDAH    $0, 3($31)
SLL     $0, 45, $0
STQ     $0, result
```

3. *Source:*

```
#define BIG (32768l*32768l*32768l + 1)

long result = 0;

void foo1()
{
result = (3*BIG);
}
```

Generated Code:

```
LDAH    $0, 3($31)
SLL     $0, 45, $0
LDA     $0, 3, $0
STQ     $0, result
```

[Beginning of Note | Top of Document | Cookbook Home]

Code Generation for LOAD Operations

[Previous Note | Next Note | Top of Document | Cookbook Home]

This memo describes how to generate instruction sequences for loading values of different sizes from memory into registers.

Except where noted, the instruction sequences shown are those generated by the DEC C compiler for Digital UNIX (formerly DEC OSF/1).

Other compilers mentioned are:

- ACC -- the C compiler initially provided with Digital UNIX as the default system compiler. It is a port and retarget of the MIPS U-code compilers

- GCC -- the Alpha port of the GNU C compiler developed by Richard Kenner for the Free Software Foundation.

When looking at the problem of loading values from memory into registers, assume that the address of the location is in register *RA* and that the result is to be placed in an integer register *RT*.

The following two tables (the first for signed loads, and the second for unsigned loads) describe the code sequences the DEC C compiler generates for each size of data. These tables work for both volatile and non-volatile memory locations because there is only one load operation in each pattern. So there can be no corruption of the data in the middle of the load operation. However, the optimizer must handle loads of volatile and non-volatile data differently since it must record that the load is equivalent to a load and a modification of data.

Table 1: Signed Load Sequences for Digital Product Compilers
Address in Register RA
Result in Register RT

Data Size	DEC C Generated Code Sequence	
64-bit	LDQ	RT,0(RA) *
32-bit	LDL	RT,0(RA)
16-bit	BIC	RA,0x6,R1
	AND	RA,0x6,R2
	LDQ	RT,0(R1)
	LDA	R2,2(R2)
	EXTQH	RT,R2,RT
	SRA	RT,0x30,RT
8-bit	LDQ_U	RT,0(RA)
	LDA	R1,1(RA)
	EXTQH	RT,R1,RT
	SRA	RT,0x38,RT

*The GNU compiler (GCC) uses a LDT RT,0(RA) followed by STT RT,... to move one long int to another. This operation uses a floating-point register, and thereby saves on integer registers when there are not enough of them.

Table 2: Unsigned Load Sequences for Digital Product Compilers
Address in Register RA
Result in Register RT

Data Size	DEC C Generated Code Sequence	
64-bit	LDQ	RT,0(RA) *
32-bit	LDL	RT,0(RA)
	ZAPNOT	RT,0xf,RT
16-bit	BIC	RA,0x2,R1
	AND	RA,0x2,R2
	LDQ	RT,0(R1)
	EXTWH	RT,R2,RT
8-bit	LDQ_U	RT,0(RA)
	EXTBL	RT,RA,RT

*Again, the GNU compiler (GCC) uses a LDT RT,0(RA) followed by STT RT,... to move long values.

The preceding instruction sequences assume that the data is properly aligned. If the data is not properly aligned, the code generated by the DEC C compiler always generates a trap. For example, the ldq instruction used for loading a 64-bit quantity generates a trap if the bottom 3 bits of the address are not zero. The same is true for 32-bit quantities.

Other compilers have different assumptions about what should happen if the data is unaligned. While DEC C generates a trap indicating the error to the user, the ACC compiler for example, gives incorrect results.

In some situations, the compiler knows the low 3 bits of the address. This can occur with external or static variables that are allocated in memory or with fields of a structure. The following two tables show the code sequences generated by the DEC C compiler for loading 16-bit quantities with the low 3 address bits equal to 0, 2, 4, and 6.

Table 3: Load of Signed 16-bit Quantities with known low 3 address bits
Address of quadword in RA
Result value in RT

Low-order 3 address bits	DEC C Generated Code Sequence		GCC Generated Code Sequence	
0	LDQ	RT,0(RA)		
	SLL	RT,0x30,RT		
	SRA	RT,0x30,RT		
2	LDQ	RT,0(RA)	ldl	RT,0(RA)
	SLL	RT,0x20,RT	sra	RT,0x10,RT
	SRA	RT,0x30,RT		
4	LDQ	RT,0(RA)		
	SLL	RT,0x10,RT		
	SRA	RT,0x30,RT		
6	LDQ	RT,0(RA)		
	SRA	RT,0x30,RT		

```
Table 4: Load of Unsigned 16-bit Quanitities with known low 3 address bits
                        Address of quadword in RA
                          Result Value in RT


Low-order 3              DEC C
address bits     Generated Code Sequence

  0              LDQ      RT,0(RA)
                 ZAPNOT   RT,0x3,RT

  2              LDQ      RT,0(RA)
                 EXTWL    RT,0x2,RT

  4              LDQ      RT,0(RA)
                 EXTWL    RT,0x4,RT

  6              LDQ      RT,0(RA)
                 SRL      RT,0x30,RT
```

The DEC C and GCC compilers generate similar code patterns for these loads with the exception of the better pattern that GCC generates for the signed case when the address low-order bits are 2. In this case, GCC takes advantage of the fact that an ldl instruction can be used to load the value into the upper quarter of the lower half of the register with sign-extended bits in the upper half of the register. Therefore, a simple arithmetic shift can be used to justify the integer, eliminating one instruction.

[Beginning of Note | Top of Document | Cookbook Home]

Code Generation for Unsigned Arithmetic Operations

[Previous Note | Next Note | Top of Document | Cookbook Home]

The issue here is how to perform arithmetic operations on unsigned data objects that are not a full 64 bits. Consider, for example, loading a 32-bit value from memory into a register:

```
LDL rx, disp(ry)
```

The LDL instruction performs a signed load; the source operand is fetched from memory, sign-extended, and written to rx. For such signed numbers, arithmetic is not problematic.

But for unsigned numbers, this operation is less than optimal. To set up a 32-bit unsigned data object as a 64-bit unsigned data object, two instructions are needed:

```
LDL rx, disp(ry)
ZAPNOT rx,0xF,rx
```

In this example, the 32-bit number at disp(ry) is loaded sign-extended into rx. The ZAPNOT instruction then zeros out the high-order 32 bits of rx.

But if you do a lot of unsigned arithmetic, you don't want to do all these ZAPNOTS or ZAPs.

The optimal solution, then, is for your compiler to:

1. Remember what the sign of the data object is.
2. Delay dealing with the sign extension until it has to.

If. for example, you just need to move an unsigned data quantity from one memory location to another, you can ignore the sign extension and just generate:

```
LDL rx, disp1(ry)
STL rx, disp2(rz)
```

The general principal to apply, then, is to delay generating instructions to convert smaller data into 64-bit data. Just keep track if what's in the upper bits. There are three possibilities for how the data is represented in a register:

- Sign-extended (SE)
- Zero-extended (ZE)
- Unknown (UNK)

Then when you have to do an arithmetic operation, you could have 9 possible code sequences to generate, where the possibility matrix is:

Operand2-> / Operand1 \	SE	ZE	UNK
SE			
ZE			
UNK			

Many arithmetic operations require no special handling of the high-order 32 bits of their operands. For these instructions, the nine possible code sequences are the same.

Adding 32-bit Numbers

Let's consider, for example, adding two 32-bit numbers. The ADDL instruction adds rx to rb, ignoring the high-order 32 bits, and writes the sign-extended 32-bit sum to rz:

```
ADDL rx,ry,rz
```

The result is always a sign-extended 64-bit number. For this operation, the compiler does not need to do

any special handling of the high-order 32 bits of the operands, regardless of whether the original 32-bit numbers are signed or unsigned.

Shifting 32-bit Numbers

If you need to shift one register by another (assume 32-bit integers), the resulting high-order 32 bits in the register are now unknown. Flag them as such.

Then if you need to do a store of that register, just generate an STL instruction. However, if you need to do a compare operation, for example, you must first zero-extend or sign-extend the value in the low-order 32 bits throughout the 64-bit register.

Comparing 32-bit Numbers

Consider comparing two 32-bit unsigned numbers:

- Case 1: If both unsigned numbers are zero-extended: Use CMPULT and CMPULE instructions
- Case 2: If both unsigned numbers are stored in a sign-extended fashion: Again, use CMPULT and CMPULE instructions
- Case 3: If one unsigned number is stored in a register with unknown high-order 32 bits: Convert it to whatever form (ZE or SE) the other one is in.
- Case 4:If both numbers are stored in registers with unknown high-order 32 bits, convert both to either ZE or SE. See the following table for conversion code.

From \ To->	SE	ZE	UNK
SE	---	ZAPNOT rx,15,rx	
ZE	ADDL rx,0,rx	___	
UNK	ADDL rx,0,rx	ZAPNOT rx,15,rx	---

Subroutine Calls

When calling or returning from a subroutine, you don't know what's in the high-order bits of a register.

For signed quantities, this is not a problem.

For unsigned quantities, the register must be put into a consistent form. Choices for doing this are:

- The caller could ZAP or sign-extend the register
- The subroutine could ZAP sign-extend the register
- Do no conversion: assume the high-order 32 bits of passed registers are unknown, and generate the worst possible code all the time.

The convention for Alpha compilers is for the caller to sign-extend the high-order 32 bits of longwords, byte integers, and word integers; and for the caller to ZAP the high-order 32 bits of byte logicals and word logicals. (See the Data Manipulation chapter of the *Digital UNIX Calling Standards for Alpha Systems*.)

This convention is not always followed, however. If you have control of boths sides of the calling interface, you could implement a different convention.

Generally, when in doubt, a compiler should sign-extend, but in cases where the compiler knows something about the interface (either because of a prototype, or because the caller and callee are in the same compilation unit), relaxing the calling standard for optimization purposes is all right.

[Beginning of Note | Top of Document | Cookbook Home]

Code Generation for STORE Operations

[Previous Note | Next Note | Top of Document | Cookbook Home]

This memo describes how to generate instruction sequences for storing values of different sizes into memory.

The instruction sequences shown are those generated by the DEC C compiler for Digital UNIX.

When looking at the problem of storing values into memory, assume that the value to be stored is in register *t0*, and the address of the memory location in which to store the value is in register *a0*

The following table describes the code sequences the DEC C compiler generates for each size of data. This table applies to both signed and unsigned values.

Table 1: Store Sequences for Digital Product Compilers
Value to be stored - in register t0
Address of memory location - in register a0

Data Size	DEC C Generated Code Sequence

```
64-bit      STQ   t0,0(a0)  ;Store value in t0 at quadword location in a0.

32-bit      STL   t0,0(a0)  ;Store value in t0 at longword location in a0.

16-bit      BIC   a0,0x2,t3 ;t3 gets address of 32-bit word containing the
                            ;   memory addr. in a0.
            LDL   t5, 0(t3) ;Load that word into t5.
            AND   a0,0x2,t4 ;Put 0 or 2 into low-order 3 bits. Result in t4.
            MSKWL t5,t4,t5  ;Clear 16-bit area.
            INSWL t0,t4,t6  ;Shift value in t0 into 16-bit area. Result in t6.
            BIS   t5,t6,t5  ;Combine (OR) t5 and t6. Result in t5.
            STL   t5,0(t3)  ;Store t5 at word containing memory address in a0.

8-bit       LDQ_U t3,0(a0)  ;Load quadword containing the memory address in a0.
            MSKBL t3,a0,t3  ;Put 0 in the byte in t3 pointed to by a0.
            INSBL t0,a0,t4  ;Shift value to store into the right place.
            BIS   t3,t4,t3  ;Combine (OR) t3 and t4. Result in t3.
            STQ_U t3,0(a0)  ;Store t3 at quadword containing memory address in a0.
```

[Beginning of Note | Top of Document | Cookbook Home]

Loading and Storing Fields

[Previous Note | Next Note | Top of Document | Cookbook Home]

This note addresses code generation for structure member fields that are not aligned on 8-bit, 16-bit, 32-bit, or 64-bit boundaries.

Consider the following structure containing three bit-field members:

```
typedef struct foo{
  int  x:3;
  int  y:2;
  int  z:3;
} bar;
```

Loading Fields

The following example shows generated code for loading these bit-field structure members:

Source:

```
typedef struct foo{
  int  x:3;
  int  y:2;
  int  z:3;
} bar;

int loadx (bar *p)
{
 return p->x;
}

int loady (bar *p)
{
 return p->y;
}

int loadz (bar *p)
{
 return p->z;
}
```

Generated Code:

```
loadx:

LDL   a0, 0(a0)      # Load the longword containing x.
SLL   a0, 0x3d, v0   # Shift left 64 bits - #bits in x (64-3=61).
SRA   v0, 0x3d, v0   # Shift right 64 bits - #bits in x (64-3=61).
RET   zero, (ra), 1  # Return.

loady:

LDL   a0, 0(a0)      # Load the longword containing y.
SLL   a0, 0x3b, v0   # Shift left 64 bits - (#bits in y + x) (64-5=59).
SRA   v0, 0x3e, v0   # Shift right 64 bits - #bits in y (64-2=62).
RET   zero, (ra), 1  # Return.

loadz:

LDL   a0, 0(a0)      # Load the longword containing z.
SLL   a0, 0x38, v0   # Shift left 64 bits - (#bits in z + y + x) (64-5=59).
SRA   v0, 0x3d, v0   # Shift right 64 bits - #bits in z (64-3=61).
RET   zero, (ra), 1  # Return.
```

Listing (.lst file) for above example

In general, to load a signed quantity that is not 16-bit-, 32-bit-, or 64-bit-aligned:

1. Load the 32-bit or 64-bit word containing the quantity.
2. Sign-extend the quantity.

To sign-extend such a quantity:

1. Shift left 64 bits minus the sum of the number of bits comprising the quantity plus its displacement from the beginning of the word containing it.
2. Shift right 64 bits minus the number of bits comprising the quantity.

For example, to sign-extend the 2-bit field y above:

```
SLL   R1, 0x3b, R2  # Shift left 59 bits.
SRA   R2, 0x3e, R2  # Shift right 62 bits.
```

An optimization is achieved if the field being referenced happens to have its high-order bit in bit 31 of the longword or quadword loaded. In this case an instruction is saved (the SLL is not needed):

```
LDL   a0, 0(a0)              # Load the longword containing the field.
```

```
SRA   v0, 32-fieldsize, v0   # Shift right 32 bits - #bits in the field.
```

Storing Fields

The following example shows generated code for storing these bit-field structure members:

Source:

```
typedef struct foo{
  int  x:3;
  int  y:2;
  int  z:3;
} bar;

void storex (bar *p, int v)
{
 p->x = v;
}

void storey (bar *p, int v)
{
 p->y = v;
}

void storez (bar *p, int v)
{
 p->z = v;
}
```

Generated Code:

```
storex:

LDL   t1, 0(a0)     # Load the longword containing the 3-bit field x.
AND   a1, 0x7, a1   # Clear every bit of a1 (which contains v) that is not
                    # part of the 3-bit quantity to be stored.
BIC   t1, 0x7, t2   # Clear the 3-bit field x, leaving rest of register intact.
BIS   t2, a1, t2    # OR x and v together, placing v into x.
STL   t2, 0(a0)     # Store result into memory.

storey:

LDL   t1, 0(a0)     # Load the longword containing the 2-bit field y.
AND   a1, 0x3, a1   # Clear every bit of a1 (which contains v) that is not
                    # part of the 2-bit quantity to be stored.
SLL   a1, 0x3, a1   # Shift left 3 bits to get v into y's position.
BIC   t1, 0x18, t2  # Clear the 2-bit field y, leaving rest of register intact.
BIS   t2, a1, t2    # OR y and v together, placing v into y.
STL   t2, 0(a0)     # Store result into memory.

storez:

LDL   t1, 0(a0)     # Load the longword containing the 3-bit field z.
AND   a1, 0x7, a1   # Clear every bit of a1 (which contains v) that is not
                    # part of the 3-bit quantity to be stored.
SLL   a1, 0x5, a1   # Shift left 5 bits to get v into z's position.
BIC   t1, 0xe0, t2  # Clear the 3-bit field z, leaving rest of register intact.
BIS   t2, a1, t2    # OR z and v together, placing v into z.
STL   t2, 0(a0)     # Store result into memory.
```

Listing (.lst file) for above example

[Beginning of Note | Top of Document | Cookbook Home]

IEEE Floating-Point Implementation

[Previous Note | Next Note | Top of Document | Cookbook Home]

The Alpha architecture implements IEEE floating-point format for normalized floating-point numbers in its instruction set.

There are six exceptions that can be generated by floating-point operate instructions, all signaled by an arithmetic exception trap. These exceptions are:

- Invalid operation
- Division by zero
- Overflow
- Underflow; may be disabled
- Inexact result; may be disabled
- Integer overflow (conversion to integer only); may be disabled

Floating-Point Trap Modes

The following trap modes can be used for IEEE floating-point processing:

- Fast mode -- provided by the Alpha instructions without qualifiers. For example:

  ```
  MULT  F1,F2,F1
  ```

  This mode traps on the following exceptions, which are considered to be fatal:

  - Overflow
  - Divide by zero
  - Invalid operation

  Underflows are not trapped and produce zeros. Inexact results are not trapped.

  In most cases, Alpha application developers will want to use this mode or the fast mode with underflow, which are ideal for operations on finite operands that give finite results.

- Fast mode with underflow -- provided by the Alpha instructions with the /U qualifier. For example:

```
MULT/U  F1,F2,F1
```

This mode traps on the same exceptions as does fast mode, but also traps on underflow.

- Portable IEEE floating-point -- provided by the Alpha instructions with the /SU qualifiers. For example:

```
MULT/SU  F1,F2,F3
TRAPB
```

Note that unlike the previous two modes, the destination register must be different than the source registers.

This mode provides trap *and recovery* for the following exceptions:

  - Overflow
  - Divide by zero
  - Invalid operation
  - Underflow

None of these exceptions are fatal. Inexact results are not trapped.

This mode involves using a software arithmetic trap handler, and building a valid trap shadow (using the TRAPB instruction and following register reuse rules) for all floating-point instructions that can cause an exception.

This mode is rarely used. However, use it if you wish to generate the IEEE exceptional results (infinities, NaNs, denormals). For an overview of this mode, see the next section of this note. For more details about this mode, see the *Floating-Point Instructions* section (Section 4.7) of the *Alpha Architecture Reference Manual*).

- Inexact IEEE floating-point -- provided by the Alpha instructions with the /SUI qualifiers. For example:

```
MULT/SUI  F1,F2,F3
TRAPB
```

Same as "fast with underflow" mode, but also traps on inexact rounding results.

Portable IEEE Floating-Point Trap Mode

[This section applies only if you need to develop programs that deliberately operate outside the overflow/underflow range, or that use IEEE NaNs and denormals. In such cases, software assistance can be provided by a software arithmetic trap handler, plus constraints on the instructions surrounding the trap.]

In IEEE format, there are several kinds of nonstandard floating-point numbers: NaNs, Infinities, denormals (see the discussion of IEEE floating-point formats in the *Alpha Architecture Reference Manual*).

The Alpha built-in floating-point instructions might not deal with these special cases. For example, in the event of an underflow during program execution, the standard floating-point instructions simply turn the underflow to zero. They do not usually deal with NaNs and infinities directly.

This causes a problem when you need to provide all the features of the IEEE floating-point standard. The hardware does not provide a solution because all the special checks required would cause the hardware to be bigger and run slower. So rather than implementing it in hardware, the Alpha architecture provides the /SU instruction qualifier as an option to the built-in instructions (see Appendix C of the *Alpha Architecture Reference Manual*).

Instructions using the /SU qualifier do check for and trap on the special cases (NaNs, infinities, denormals). They cause a trap to the software trap handler. The trap handler does whatever is necessary to simulate the IEEE floating-point standard. That means that you run at a reasonable speed for normalized numbers, but as soon as one of the previously described special cases is encountered, you might be running a thousand times slower because of the software simulation.

So, to get true IEEE floating-point processing, use the /SU qualifier on floating-point instructions. This executes a modified floating-point instruction opcode, which traps on the special cases.

The trap handler then software simulates what the IEEE standard requires. Each operating system has a trap handler that simulates these instructions.

Imprecise Trapping

The problem with the trap-handling process is that traps do not occur instantly on a RISC processor. The term for this is *imprecise trapping*. That is, when the trap occurs, the trap handler has to determine which instruction caused the trap, for the handler to start simulating it.

This instruction is not the one that the program counter (pc) points to, because the pc can be at the end of a floating-point add operation and realize that an infinity needs to be generated. Therefore, you need to make it possible for the trap handler to back up to that instruction, simulate it, and continue execution. That means you must make sure that instructions immediately following the floating-point instruction that caused the exception (the *trigger* instruction) do not change any registers that had meaning (were live) at the trigger instruction. If such a register were changed, you cannot back up, reexecute that instruction, and go on. There is, therefore, the idea of the *trap shadow*, which places constraints on the instructions surrounding the trap.

The Trap Shadow

The trap shadow is the set of instructions potentially executed after an instruction that signals an arithmetic trap but before the trap is actually taken.

That is, the trap shadow consists of all instructions from and including the floating-point operation to an instruction that guarantees that all exceptions that might have occurred have in fact occurred. There are such instructions. One of them, the TRAPB (trap barrier) instruction, is used by compilers to end a trap shadow.

The TRAPB instruction ensures that the program does not continue executing until all previously

executing instructions that might cause a trap have at least gotten past the point where they could cause a trap. A trap shadow goes from that floating-point operation to a point where the exception is trapped or to a TRAPB instruction. If you have a branch point, you need to place a TRAPB before it to finish the trap shadow.

Another Consideration -- Register Reuse

When you have a floating-point operation and you are dealing with IEEE conformance, you cannot put a value into a register that had a value that was used previously in the trap shadow. That means you can run out of registers quickly.

In the trap shadow, you cannot redefine a register that is live at the floating-point instruction that can cause a trap. If you get to a point where you need to reuse a register because you have no registers left, you must insert a TRAPB to finish the trap shadow.

This slows your code down a bit, and if you have a lot of NaNs or infinities, your code slows down to a crawl. NaNs or infinities are rare, but TRAPB instructions still slow you down because the TRAPB usually stalls waiting for the floating-point pipeline to report error.

Assume that a floating-point operation takes four cycles and you have six floating-point operations in a row. Also assume you can issue about six instructions before you run out of registers. You have three adds and three multiplies -- that's three cycles. But it takes four cycles for the last floating-point operation to finish. Therefore, the total is seven cycles. So if you have these six instructions followed by a TRAPB, rather than taking three cycles, it takes seven cycles, which slows execution down by a factor of about two.

Implementing IEEE floating-point will slow you down. You have to put TRAPBs in where absolutely needed, but you want to optimize them away as much as possible.

The basic rule is: you cannot change a register in the trap shadow that has a value that has been used earlier in the trap shadow. You cannot change it in the trap shadow because of the possibility of having to reexecute that floating-point operation and all the instructions after it.

You can change a register that is *not* in use on entry to the trap shadow. And if you use a register that was not in use, you can use it again.

The trap handler has to be able to back up to that point in the program and continue execution from the floating-point instruction that can cause a trap, using what is in the registers at that point.

Another condition that must hold is: within the trap shadow, no register may be used more than once as a destination register.

For a more complete discussion of trap shadow constraints, see the *Alpha Architecture Reference Manual*. For the most current information on trap shadows, see the calling standard documents.

IEEE Rounding Modes

For IEEE floating-point operations, four rounding modes are provided:

- Normal rounding (unbiased round to nearest)
- Rounding toward minus infinity
- Rounding toward zero
- Rounding toward plus infinity

The first three can be specified statically in the instruction. Rounding toward plus infinity can be obtained by setting the Floating-point Control Register (FPCR) to select it and then specifying dynamic rounding mode in the instruction (using the /D qualifier).

For more information on IEEE Rounding modes and a general discussion of IEEE floating-point support, see the *Alpha Architecture Reference Manual*.

Summary for Implementing IEEE Floating-Point:

1. Compile using the /SU qualifier on instructions that can cause a floating-point exception.

2. Compute the trap shadow in the compiler. The trap shadow is all instructions from (and including) the floating-point instruction that can cause a trap, to a TRAPB placed before one of the following:
   - A branch point
   - The point where you do not have enough registers to do what you need to do

3. Insert a TRAPB instruction just before the branch, or at the point where you have run out of registers and need to begin reusing them. The TRAPB forces an end to the trap shadow.

4. When you are doing register allocation in the compiler, in the trap shadow do not modify a register that is *live* (that is, in use) at the floating-point instruction that begins the trap shadow (including operands of the floating-point instruction itself). In the trap shadow, do not modify the result register of the trigger floating-point instruction.

5. Select the IEEE floating-point rounding mode you want to use.

[Beginning of Note | Top of Document | Cookbook Home]

How to Do Integer Multiply

[Previous Note | Next Note | Top of Document | Cookbook Home]

The Alpha integer multiply instructions consume many processor cycles. So, when multiplying an integer register by an integer constant, you don't want to generate integer multiply instructions. You can instead generate a small sequence of fast (single-cycle) instructions to accomplish integer multiply.

Combinations of the following instructions can be used to multiply an integer register by any integer constant. In most cases, this method is more efficient than using a multiply instruction:

```
ADDQ rx,rx,ry     # Multiply by 2

S4SUBQ rx,rx,ry   # Multiply by 3
```

```
S4ADDQ rx,R31,ry  # Multiply by 4

S4ADDQ rx,rx,ry   # Multiply by 5

S8SUBQ rx,rx,ry   # Multiply by 7

S8ADDQ rx,R31,ry  # Multiply by 8

S8ADDQ rx,rx,ry   # Multiply by 9

SLL    rx,#n,ry   # Multiply by 2^n
```

There's no straightforward algorithm for determining the most efficient sequence of instructions for doing a particular integer multiply, so try different sequences to see which ones are best.

The following examples show the code sequences generated for multiplying an integer register by an integer constant.

Examples:

1. To multiply an integer register rx by 2^n-1 [for n > 3]:

   ```
   SLL rx,#n,ry   # Multiply by 2^n.
   SUBQ ry,rx,ry  # Subtract rx.
   ```

2. To multiply an integer register rx by 2^n+1 [for n > 3]:

   ```
   SLL rx,#n,ry   # Multiply by 2^n.
   ADDQ ry,rx,ry  # Add rx.
   ```

3. To multiply an integer register rx by 57:

   Factor 57:

   57 = 3 * 19
   And 19 = 4*5 - 1

   So,

   rx*57 = rx*(3*(4*5 - 1)) = rx*3*4*5 - rx*3

   Generated Code:

   ```
   S4SUBQ rx,rx,ry    # Multiply by 3 (ry = 3*rx).
   S4ADDQ ry,ry,rz    # Multiply result by 5 (rz = 15*rx).
   S4ADDQ rz,$31,rz   # Multiply result by 4 (rz = 60*rx).
   SUBQ   rz,ry,rz    # Subtract ry, which is 3*rx (rz = 60*rx - 3*rx = 57*rx).
   ```

   The above sequence can be further collapsed by combining the last two instructions into one:

   ```
   S4SUBQ rx,rx,ry    # Multiply by 3 (ry = 3*rx).
   S4ADDQ ry,ry,rz    # Multiply result by 5 (rz = 15*rx).
   S4SUBQ rz,ry,rz    # Multiply result by 4 and subtract 3*rx
                      #             (rz = 60*rx - 3*rx = 57*rx).
   ```

   Collapsing multiple instructions into a single instruction is one of the peephole optimations compilers can perform to reduce cycle time.

   So, you can do the previous multiply in only three instruction cycles.

4. To multiply an integer register rx by 111:

   Factor 111:

   111 = 3 * 37
   And 37 = 4*9 + 1

   So,

   rx*111 = rx*(3*(4*9 + 1)) = rx*3*4*9 + rx*3

   Generated Code:

   ```
   S4SUBQ rx,rx,ry    # Multiply by 3 (ry = 3*rx).
   S8ADDQ ry,ry,rz    # Multiply result by 9  (rz = 27*rx).
   S4ADDQ rz,$31,rz   # Multiply result by 4  (rz = 108*rx).
   ADDQ   rz,ry,rz    # Add ry, which is 3*rx (rz = 108*rx + 3*rx = 111*rx).
   ```

   And again, the above sequence can be further collapsed by combining the last two instructions:

   ```
   S4SUBQ rx,rx,ry    # Multiply by 3 (ry = 3*rx).
   S8ADDQ ry,ry,rz    # Multiply result by 9  (rz = 27*rx).
   S4ADDQ rz,ry,rz    # Multiply result by 4 and add 3*rx
                      #             (rz = 108*rx + 3*rx = 111*rx).
   ```

- To multiply an integer register rx by 741:

Factor 741:

741 = 3 * 247
And 247 = 8*31 - 1
And 31 = 2^5 - 1

So,

rx*741 = rx*(3*(8*(2^5 - 1) - 1))

Generated Code:

```
S4SUBQ rx,rx,ry    # Multiply by 3 (ry = 3*rx).
SLL    ry,#5,rz    # Multiply result by 32 (rz = 96*rx)
SUBQ   rz,ry,rz    # Subtract 3 from result (rz = 93*rx)
S8SUBQ rz,$31,rz   # Multiply result by 8 (rz = 744rx)
SUBQ   rz,ry,rz    # Subtract 3 from result (rz = 741)
```

The above sequence can be further collapsed by combining the last two instructions:

```
S4SUBQ rx,rx,ry     # Multiply by 3 (ry = 3*rx).
SLL    ry,#5,rz     # Multiply result by 32 (rz = 96*rx)
SUBQ   rz,ry,rz     # Subtract 3 from result (rz = 93*rx)
S8SUBQ rz,ry,rz     # Multiply result by 8 and subtract 3
                    #  (rz = 744rx-3rx = 741rx)
```

Your compiler can be designed to generate code with even greater efficiencies on the Alpha 21164 (EV5) processor by taking advantage of simultaneous execution of instructions that use different registers. For example, the DEC C compiler does the integer multiply in the first example above as follows:

```
SLL    rx,0x4,ry    # Multiply rx by 16. Result in ry (ry = 16*rx).
S4SUBQ rx,rx,rx     # Multiply rx by 3. Result in rx.
ADDL   ry,rx,rx     # Add rx and ry (rx=rx+ry= 19*rx).
```

Listing of generated code for all integer multiply examples above.

[Beginning of Note | Top of Document | Cookbook Home]

How to Do Integer Divide

[Previous Note | Next Note | Top of Document | Cookbook Home]

Alpha processors have no integer divide instruction. So, how are integer divides accomplished?

For integers less than 2^{52}, the floating-point registers could be used for divide operations, but that is not an efficient method because of the many memory operations involved.

For divide operations where the divisor is not a constant, use the div and ldiv library routines.

Constant Divisor

For divide operations where the divisor is a constant, the compiler can do the divide as follows. To divide i by c to obtain quotient q and remainder r:

1. Compute the reciprocal of the divisor (1/c) in fixed-point format.

   To do this, see the algorithm and accompanying discussion in the next section.

2. Multiply the dividend i by this reciprocal.

   Then to compute the quotient q:

   - For unsigned short int, shift right 16 bits.
   - For unsigned int, shift right 32 bits.
   - For unsigned long int, use the UMULH instruction, which does the unsigned multiply but keeps only the high-order 64 bits of the 128-bit multiply result, effectively performing a shift right 64.

3. Compute the remainder: r = c - iq

Computing the Reciprocal of a Constant Divisor

Consider a divide operation of two unsigned 16-bit integers i/c, where c is a constant:

$$i/c = q$$

To compute the reciprocal of an unsigned 16-bit constant divisor in fixed-point format, you want to find a number *m* such that *m* multiplied by the dividend i and shifted right 16 bits (for 16-bit unsigned numbers) yields the correct quotient:

$$m * i >> 16 = quotient$$

Such an *m*, when multiplied by the divisor c, must be greater than or equal to 2^{16}:

$$m * c >= < 2^{16}$$

But *m* must not be too big. So, the upper bound for *m* should be such that:

$$m * c < 2^{16} + 2^{n}$$

where *n* is the smallest power of 2 such that 2^n is greater than or equal to the divisor c.

Combining these upper and lower bounds gives us the following criteria for finding a multiplier *m*:

$$2^{16} + 2^{n} > m * c >= 2^{16}$$

Example

To divide i/100:

1. Compute the reciprocal of 100 by finding a number m such that:

   $$2^{16} + 2^{7} > 100*m >= 2^{16}$$

   That is:

   $$65764 > 100 * m >= 65536$$

2. Make a first estimate for m by dividing 65536 by 100: m = 655.
3. Test: 655 * 100 = 65500, which is too small (must be >= 65536), so...
4. Add 1 to 655: m = 656.

5. Test: 656 * 100 = 65600, which is greater than or equal to 65536, and less than 65764 so...

   The multiplier m is 656.

6. Multiply i * m.

7. Shift right 16 bits to obtain quotient q.

   For example, if i = 100, then i/c is 1. Let's check it:

   a. 100 * 656 = 65600

   b. 65600 >> 16 = 1

Further Optimization

In our example above, notice that 656 = 16 * 14.

The 16 represents a shift left of 4 bits, which when subtracted from our shift right of 16, yields a shift right of 12. So, if you remove the factor of 16 above, and use a shift right of 12, the proper quotient can be obtained by:

```
(i * 41) >> 12
```

32-Bit and 64-bit Integers

The same algorithms described in this section can be used for 32-bit and 64-bit unsigned integers, except that you shift right 32 bits and 64 bits, respectively.

For 32-bit integers, the criteria for calculating the reciprocal multiplier *m* is:

$2^{32} + 2^{n} > m * c >= 2^{32}$

Then the MULQ instruction can be used, followed by a shift right of 32 bits:

```
MULQ of m * i >> 32 = quotient
```

There is some controvery over 32-bit divide as to whether to use a multiply instruction or the shift, add, and subtract instructions.

For 64-bit integers, the criteria for calculating the reciprocal multiplier *m* is:

$2^{64} + 2^{n} > m * c >= 2^{64}$

Then the UMULH instruction can be used, eliminating the need for a separate shift instruction:

```
UMULH of m * i = quotient
```

Examples

Setting Up Stack Frames

This note describes how to direct the assembler to lay out a Digital UNIX stack frame for a C procedure.

The Assembly Language Programmer's Manual, Section 6.3.3, has an excellent discussion of the issues the compiler writer must consider when creating a stack frame. Here we present an example of a program, the compiler code that sets up its stack frames, and the generated assembly code.

Consider the following procedure:

```
int ten_args ( int arg1, int arg2, int arg3, int arg4, int arg5,
               int arg6, int arg7, int arg8, int arg9, int arg10 );

int main ( int argc, char* argv[], char * env[] )
{

    printf (" ten_args() = %d\n",
              ten_args ( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ) );

    return 0;
}

int ten_args ( int arg1, int arg2, int arg3, int arg4, int arg5,
               int arg6, int arg7, int arg8, int arg9, int arg10 )
{
     printf ( " arg1 = %d\n", arg1 );
     return arg10;
}
```

Like most C code, these procedures can use a fixed stack-frame layout. Consider a null-frame layout for very simple procedures that do not call other procedures or use any variables not passed as arguments.

The stack layout we will use is a *homed-argument* stack; we place a save area for the ten_args register arguments in memory adjacent to the arguments passed on the stack.

The stack frame layout after ten_args has been called looks like this:

```
$sp ->  -----           # ten_arg's stack base
RSA ->  Saved $26       # ten_arg's register save area.  Only $26 is saved.
        -----
        Local
        Variable
        Storage
        -----
        Saved $16-$21   #  Argument save area
VFP ->  -----           #  Virtual Frame Pointer: end of ten_args' stack
        -----           #  main's stack base, same address as ten_arg's VFP
        Args 7-10       #  Args saved into memory by caller
        -----
```

```
RSA'->  Saved $26       #  main's register save area
        Other           #  Return register must be saved at offset zero
        Saved           #  within the RSA.
        Registers
        -----
        Local
        Variable
        Storage
        -----
        Saved $16-$18   #  Argument save area [note that since main does
                        #  not use its arguments, this could be omitted].
VFP'->  -----           #  Virtual Frame pointer; end of main's stack.
```

The most important components of each procedure's stack are its *stack base*, *Virtual Frame Pointer (VFP)*, and *RSA offset*:

- *Stack base*:

  Register 30 (also called $sp) points to the procedure's stack base. Digital UNIX stacks grow downward from high virtual addresses.

  To allocate a stack frame of 64 bytes, for example, compilers generate the instruction:

  ```
  lda $sp,-64($sp)
  ```

  And to restore the caller's stack frame, compilers generate the complementary instruction:

  ```
  lda $sp,64($sp)
  ```

- *VFP*:

  A procedure's Virtual Frame Pointer (VFP) is defined as:

  ```
      The procedure's stack base + its frame size
  ```

  Therefore, the VFP is the address of the high end of the procedure's stack space. Local offsets are computed relative to the VFP. Becuase the VFP is the address of the high end of the procedure's stack space, such offsets are negative. A commonly generated assembly language idiom for accessing a stack variable (for example, a variable offset of -48 bytes within a 64-byte stack frame) is:

  ```
  -48+64($sp)
  ```

  A Digital UNIX procedure can have a frame-pointer register, which by convention is $15. A frame-pointer register is necessary if the procedure must modify its stack pointer for any reason. The frame pointer register points to the low end of the stack frame, as does $sp for a fixed frame procedure.

  Note that the frame-pointer register and the VFP point to opposite ends of the procedure's stack space.

- *RSA*:

  The Register Save Area is specified as an offset from the VFP in the .mask directive. Because the VFP points to the high end of the procedure's stack, the RSA offset given in the .mask directive is negative.

  In our example program, ten_args' RSA offset = -(stacksize), because ten_args' register save area is at the base of the procedure's stack frame.

Notes on the .frame directive:

- The assembler .frame directive specifies the procedure's stack frame.
- For a stack frame based on $sp, with a size of 64 bytes, and with the caller's return address passed in $26, the .frame directive is ".frame $sp,64,$26".
- The .frame directive accepts an additional parameter, which is the offset of a procedure's local variables from its VFP. This is often defaulted to zero, but can be omitted unless the procedure has local variables.

Assuming ten_args' stack frame is 64 bytes in length, the following sequence of assembler instructions and directives sets up the frame:

```
ldgp $gp, ($27)
lda $sp,-64($sp)          #  Push the frame onto the stack
stq $26,0($sp)            #  Save the return register
.mask 0x4000000,-64       #  RSA is 64 bytes from VFP (VFP is at the
                          #  end of the frame, RSA is at the base).
.frame $sp,64,$26         #  Specify frame is based on $sp, 64 bytes
                          #  long, return address in $26
.prologue 1               #  End of procedure prologue, "1" indicates to
                          #  the assembler that we have modified $gp
```

The stack frame for main is slightly different, since main allocates a 32-byte area to pass arguments to ten_args. Note the relationship of the "magic" numbers 64 and 96 in this example: the lcc compiler allocates 96 bytes of storage for main, and reserves 32 bytes at the stack base for argument passing. And 96 - 32 = 64, so the RSA, which the compiler allocates next, is 32 bytes from the stack base, or 64 bytes from the VFP. This compiler forms addresses as offsets from the VFP, so the offsets it generates are of the form <negative offset from VFP> + <VFP's offset from $sp>.

The following sequence of assembler instructions and directives sets up the frame for main:

```
ldgp $gp, ($27)
lda $sp,-96($sp)          #  Push the frame onto the stack;
                          #  first 32 bytes of the stack frame will be
                          #  used to pass arguments to ten_args
.mask 0x4000000,-64       #  RSA is 64 bytes from the VFP
.frame $sp,96,$26         #  Frame is based on $sp, 96 bytes long,
                          #  return address in $26
stq $26,-64+96($sp)       #  Save the return register
.prologue 1               #  End of procedure prologue, "1" indicates to
                          #  the assembler that we have modified $gp
```

Such pre-allocation of argument-passing areas is commonly done. This is because a fixed-frame procedure can only modify $sp once; that is the convention between a procedure and the exception

handlers that support it. The RSA and VFP offsets are relative to the base of the stack, and the exception handlers interpret the contents of RSA($sp) as a return address, and will pop <frame-sized> bytes off the stack to unwind the procedure. If $sp or the frame pointer register is not pointing to the stack base when an exception handler gets control, then GIGO takes over.

A procedure that can safely move its stack pointer is a *variable-size stack-frame format*, which uses a frame-pointer register. Variable stack-frame format is explained in the *Digital UNIX Calling Standard for Alpha Systems*.

There are some useful optimizations that a compiler can do to this example code. In particular, arguments passed in registers can be left in their original registers or cached in scratch registers, and no stack space need be allocated for them. However, varargs processing requires that args 1-6 be saved contiguously with the arguments passed in memory.

If a C language procedure does not access any statically allocated data (such as string literals), or addresses (such as a jump table for a switch statement), then it is not required to issue the ldgp macro to load its global offset table pointer.

There is also a type of procedure, called a *null-frame procedure*, that has no local variables, makes no calls to other procedures, and can leave its arguments in registers. Null-frame procedures do not need to allocate stack storage at all. See the *Digital UNIX Calling Standard for Alpha Systems* for more information on null-frame procedures. An example null-frame procedure and generated code follows:

```
int foo ( int bar, int baz )
{
    return bar + baz;
}

.ent foo
foo:
addl $16,$17,$0
trapb
ret
.end foo
```

Since foo does not reference statically allocated data or addresses, no ldgp macro is necessary to set up the procedure.

A procedure that self-recurses need not repeat a ldgp:

```
int foo ( int bar )
{
    static int static_int = 1;

    if ( bar <= static_int )
        return bar;
    else
        return bar + foo ( bar - 1 );
}


foo:
ldgp $gp,0($27)
L.foo.ugen:
lda  $sp,-32($sp)
stq  $26,0($sp)
...
bsr $26,L.foo.ugen
```

[Beginning of Note | Top of Document | Cookbook Home]

Instruction Scheduling

[Previous Note | Next Note | Top of Document | Cookbook Home]

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.

[Beginning of Note | Top of Document | Cookbook Home]

Heuristics for Loop Unrolling

[Previous Note | Next Note | Top of Document | Cookbook Home]

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.

[Beginning of Note | Top of Document | Cookbook Home]

Valuable Peephole Optimizations

[Previous Note | Next Note | Top of Document | Cookbook Home]

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.

[Beginning of Note | Top of Document | Cookbook Home]

Speculative Scheduling Facilities

[Previous Note | Top of Document | Cookbook Home]

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.

[Beginning of Note | Top of Document | Cookbook Home]

Appendix: Examples and Text-Formatted Tables

This section contains the examples and text-formatted tables accessed from various hot links in the preceding notes.

Immediate Instructions in the Alpha Architecture

[Return]

```
Mnemonic     Operation
--------     ---------
ADDL         Longword Add
SxADDL       Scaled Longword Add
ADDQ         Quadword Add
SxADDQ       Scaled Quadword Add

CMPxx        Integer Signed Compare
CMPUxx       Integer Unsigned Compare

MULL         Longword Multiply
MULQ         Quadword Multiply
UMULH        Unsigned Quadword Multiply High

SUBL         Longword Subtract
SxSUBL       Scaled Longword Subtract
SUBQ         Quadword Subtract
SxSUBQ       Scaled Quadword Subtract

AND          Logical Product
BIC          Logical Product with Complement
BIS          Logical Sum (OR)
EQV          Logical Equivalence (XORNOT)
ORNOT        Logical Sum with Complement
XOR          Logical Difference

CMOVxx       Conditional Move Integer

SLL          Shift Left Logical
SRA          Shift Right Arithmetic
SRL          Shift Right Logical

CMPBGE       Compare Byte
EXTxx        Extract Byte
INSxx        Byte Insert
MSKxx        Byte Mask
ZAPx         Zero Bytes
```

[Return]

Digital Product Compiler Inlining Modes

[Return]

```
  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.
```

[Return]

Loading 32-Bit Constants -- Listing File of Examples

[Return]

```
        increment0:
File 'mylconst2.c':
1: /* Load constant Tests */
2:
3: int result = 0;
4:
5: void increment0()
  [mylconst2.c:   5] 0x0:       27bb0001        ldah    gp, 1(t12)
  [mylconst2.c:   5] 0x4:       23bd8100        lda     gp, -32512(gp)
6: {
7:     result = result + 0;
  [mylconst2.c:   7] 0x8:       a43d8010        ldq     t0, -32752(gp)
  [mylconst2.c:   7] 0xc:       a0410000        ldl     t1, 0(t0)
  [mylconst2.c:   7] 0x10:      b0410000        stl     t1, 0(t0)
8: }
  [mylconst2.c:   8] 0x14:      6bfa8001        ret     zero, (ra), 1
        increment7:
9:
10: void increment7()
  [mylconst2.c:  10] 0x18:      27bb0001        ldah    gp, 1(t12)
  [mylconst2.c:  10] 0x1c:      23bd80e8        lda     gp, -32536(gp)
11: {
12:     result = result + 7;
  [mylconst2.c:  12] 0x20:      a43d8010        ldq     t0, -32752(gp)
  [mylconst2.c:  12] 0x24:      a0410000        ldl     t1, 0(t0)
  [mylconst2.c:  12] 0x28:      4040f002        addl    t1, 0x7, t1
  [mylconst2.c:  12] 0x2c:      b0410000        stl     t1, 0(t0)
13: }
  [mylconst2.c:  13] 0x30:      6bfa8001        ret     zero, (ra), 1
  [mylconst2.c:  13] 0x34:      47ff041f        bis     zero, zero, zero
        increment254:
14:
15: void increment254()
  [mylconst2.c:  15] 0x38:      27bb0001        ldah    gp, 1(t12)
  [mylconst2.c:  15] 0x3c:      23bd80c8        lda     gp, -32568(gp)
16: {
17:     result = result + 254;
  [mylconst2.c:  17] 0x40:      a43d8010        ldq     t0, -32752(gp)
  [mylconst2.c:  17] 0x44:      a0410000        ldl     t1, 0(t0)
  [mylconst2.c:  17] 0x48:      405fd002        addl    t1, 0xfe, t1
  [mylconst2.c:  17] 0x4c:      b0410000        stl     t1, 0(t0)
18: }
  [mylconst2.c:  18] 0x50:      6bfa8001        ret     zero, (ra), 1
  [mylconst2.c:  18] 0x54:      47ff041f        bis     zero, zero, zero
        increment16000:
19:
20: void increment16000()
  [mylconst2.c:  20] 0x58:      27bb0001        ldah    gp, 1(t12)
  [mylconst2.c:  20] 0x5c:      23bd80a8        lda     gp, -32600(gp)
21: {
22:     result = result + 16000;
  [mylconst2.c:  22] 0x60:      a43d8010        ldq     t0, -32752(gp)
  [mylconst2.c:  22] 0x64:      a0410000        ldl     t1, 0(t0)
  [mylconst2.c:  22] 0x68:      20423e80        lda     t1, 16000(t1)
  [mylconst2.c:  22] 0x6c:      b0410000        stl     t1, 0(t0)
23: }
  [mylconst2.c:  23] 0x70:      6bfa8001        ret     zero, (ra), 1
  [mylconst2.c:  23] 0x74:      47ff041f        bis     zero, zero, zero
        increment33000:
24:
25: void increment33000()
  [mylconst2.c:  25] 0x78:      27bb0001        ldah    gp, 1(t12)
  [mylconst2.c:  25] 0x7c:      23bd8088        lda     gp, -32632(gp)
26: {
27:     result = result + 33000;
  [mylconst2.c:  27] 0x80:      247f0001        ldah    t2, 1(zero)
  [mylconst2.c:  27] 0x84:      a43d8010        ldq     t0, -32752(gp)
  [mylconst2.c:  27] 0x88:      206380e8        lda     t2, -32536(t2)
  [mylconst2.c:  27] 0x8c:      a0410000        ldl     t1, 0(t0)
  [mylconst2.c:  27] 0x90:      40430002        addl    t1, t2, t1
  [mylconst2.c:  27] 0x94:      b0410000        stl     t1, 0(t0)
28: }
  [mylconst2.c:  28] 0x98:      6bfa8001        ret     zero, (ra), 1
  [mylconst2.c:  28] 0x9c:      47ff041f        bis     zero, zero, zero
        increment67000:
29:
30: void increment67000()
  [mylconst2.c:  30] 0xa0:      27bb0001        ldah    gp, 1(t12)
  [mylconst2.c:  30] 0xa4:      23bd8060        lda     gp, -32672(gp)
31: {
32:     result = result + 67000;
  [mylconst2.c:  32] 0xa8:      247f0001        ldah    t2, 1(zero)
  [mylconst2.c:  32] 0xac:      a43d8010        ldq     t0, -32752(gp)
  [mylconst2.c:  32] 0xb0:      206305b8        lda     t2, 1464(t2)
  [mylconst2.c:  32] 0xb4:      a0410000        ldl     t1, 0(t0)
  [mylconst2.c:  32] 0xb8:      40430002        addl    t1, t2, t1
  [mylconst2.c:  32] 0xbc:      b0410000        stl     t1, 0(t0)
33: }
  [mylconst2.c:  33] 0xc0:      6bfa8001        ret     zero, (ra), 1
  [mylconst2.c:  33] 0xc4:      00000000        call_pal        halt
  [mylconst2.c:  33] 0xc8:      00000000        call_pal        halt
  [mylconst2.c:  33] 0xcc:      00000000        call_pal        halt
```

[Return]

Loading Bit Fields -- Listing File of Examples

[Return]

```
        loadx:
File 'loadfield.c':
Skipping source listing to line 1...
2: typedef struct foo {
3:
4: int x:3;
5: int y:2;
6: int z:3;
7: } bar;
8:
9: int loadx (bar *p)
```

```
10: {
11:  return p->x;
  [loadfield.c:  11] 0x0:       a2100000        ldl     a0, 0(a0)
  [loadfield.c:  11] 0x4:       4a07b720        sll     a0, 0x3d, v0
  [loadfield.c:  11] 0x8:       4807b780        sra     v0, 0x3d, v0
  [loadfield.c:  11] 0xc:       6bfa8001        ret     zero, (ra), 1
       loady:
12: }
13:
14: int loady (bar *p)
15: {
16:  return p->y;
  [loadfield.c:  16] 0x10:      a2100000        ldl     a0, 0(a0)
  [loadfield.c:  16] 0x14:      4a077720        sll     a0, 0x3b, v0
  [loadfield.c:  16] 0x18:      4807d780        sra     v0, 0x3e, v0
  [loadfield.c:  16] 0x1c:      6bfa8001        ret     zero, (ra), 1
       loadz:
17: }
18:
19: int loadz (bar *p)
20: {
21:  return p->z;
  [loadfield.c:  21] 0x20:      a2100000        ldl     a0, 0(a0)
  [loadfield.c:  21] 0x24:      4a071720        sll     a0, 0x38, v0
  [loadfield.c:  21] 0x28:      4807b780        sra     v0, 0x3d, v0
  [loadfield.c:  21] 0x2c:      6bfa8001        ret     zero, (ra), 1
```

[Return]

Storing Bit Fields -- Listing File of Examples

[Return]

```
       storex:
File 'storefield.c':
Skipping source listing to line 1...
2: typedef struct foo {
3:
4: int x:3;
5: int y:2;
6: int z:3;
7: } bar;
8:
9: void storex(bar *p, int v)
10: {
11:  p->x = v;
  [storefield.c:  11] 0x0:      a0500000        ldl     t1, 0(a0)
  [storefield.c:  11] 0x4:      4620f011        and     a1, 0x7, a1
  [storefield.c:  11] 0x8:      4440f103        bic     t1, 0x7, t2
  [storefield.c:  11] 0xc:      44710403        bis     t2, a1, t2
  [storefield.c:  11] 0x10:     b0700000        stl     t2, 0(a0)
12: }
  [storefield.c:  12] 0x14:     6bfa8001        ret     zero, (ra), 1
       storey:
13:
14: void storey(bar *p, int v)
15: {
16:  p->y = v;
  [storefield.c:  16] 0x18:     a0500000        ldl     t1, 0(a0)
  [storefield.c:  16] 0x1c:     46207011        and     a1, 0x3, a1
  [storefield.c:  16] 0x20:     4a207731        sll     a1, 0x3, a1
  [storefield.c:  16] 0x24:     44431103        bic     t1, 0x18, t2
  [storefield.c:  16] 0x28:     44710403        bis     t2, a1, t2
  [storefield.c:  16] 0x2c:     b0700000        stl     t2, 0(a0)
17: }
  [storefield.c:  17] 0x30:     6bfa8001        ret     zero, (ra), 1
  [storefield.c:  17] 0x34:     47ff041f        bis     zero, zero, zero
       storez:
18:
19: void storez(bar *p, int v)
20: {
21:  p->z = v;
  [storefield.c:  21] 0x38:     a0500000        ldl     t1, 0(a0)
  [storefield.c:  21] 0x3c:     4620f011        and     a1, 0x7, a1
  [storefield.c:  21] 0x40:     4a20b731        sll     a1, 0x5, a1
  [storefield.c:  21] 0x44:     445c1103        bic     t1, 0xe0, t2
  [storefield.c:  21] 0x48:     44710403        bis     t2, a1, t2
  [storefield.c:  21] 0x4c:     b0700000        stl     t2, 0(a0)
22: }
  [storefield.c:  22] 0x50:     6bfa8001        ret     zero, (ra), 1
  [storefield.c:  22] 0x54:     00000000        call_pal        halt
  [storefield.c:  22] 0x58:     00000000        call_pal        halt
  [storefield.c:  22] 0x5c:     00000000        call_pal        halt
```

[Return]

Integer Multiply -- Listing File of Examples

[Return]

```
File 'mult_long.c':
1:
2: /* Test code for DEC C integer multiply:                          */
3: /*                                                                */
4: /* cc -migrate -O4 -source_listing -machine_code -c mult_long.c */
5: /*                                                                */
6: long n = 1;
7:
8: void load19()
  [mult_long.c:   8] 0x0:       27bb0001        ldah    gp, 1(t12)
  [mult_long.c:   8] 0x4:       23bd80a0        lda     gp, -32608(gp)
9: {
10: long m = 19;
11:     n=n*m;
  [mult_long.c:  11] 0x8:       a43d8010        ldq     t0, -32752(gp)
  [mult_long.c:  11] 0xc:       a4410000        ldq     t1, 0(t0)
  [mult_long.c:  11] 0x10:      48409723        sll     t1, 0x4, t2
  [mult_long.c:  11] 0x14:      40420562        s4subq  t1, t1, t1
  [mult_long.c:  11] 0x18:      40620402        addq    t2, t1, t1
  [mult_long.c:  11] 0x1c:      b4410000        stq     t1, 0(t0)
12: }
  [mult_long.c:  12] 0x20:      6bfa8001        ret     zero, (ra), 1
  [mult_long.c:  12] 0x24:      47ff041f        bis     zero, zero, zero
       load111:
13:
14:
```

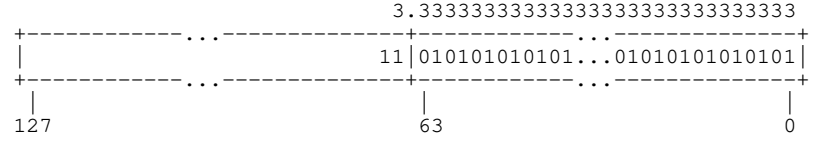
```
15: void load111()
  [mult_long.c:  15] 0x28:      27bb0001        ldah    gp, 1(t12)
  [mult_long.c:  15] 0x2c:      23bd8078        lda     gp, -32648(gp)
16: {
17: long m = 111;
18:     n=n*m;
  [mult_long.c:  18] 0x30:      a43d8010        ldq     t0, -32752(gp)
  [mult_long.c:  18] 0x34:      a4410000        ldq     t1, 0(t0)
  [mult_long.c:  18] 0x38:      48409723        sll     t1, 0x4, t2
  [mult_long.c:  18] 0x3c:      4840f724        sll     t1, 0x7, t3
  [mult_long.c:  18] 0x40:      40430402        addq    t1, t2, t1
  [mult_long.c:  18] 0x44:      40820522        subq    t3, t1, t1
  [mult_long.c:  18] 0x48:      b4410000        stq     t1, 0(t0)
19: }
  [mult_long.c:  19] 0x4c:      6bfa8001        ret     zero, (ra), 1
       load741:
20:
21: void load741()
  [mult_long.c:  21] 0x50:      27bb0001        ldah    gp, 1(t12)
  [mult_long.c:  21] 0x54:      23bd8050        lda     gp, -32688(gp)
22: {
23: long m = 741;
24:     n=n*m;
  [mult_long.c:  24] 0x58:      a43d8010        ldq     t0, -32752(gp)
  [mult_long.c:  24] 0x5c:      a4410000        ldq     t1, 0(t0)
  [mult_long.c:  24] 0x60:      48413723        sll     t1, 0x9, t2
  [mult_long.c:  24] 0x64:      4840b724        sll     t1, 0x5, t3
  [mult_long.c:  24] 0x68:      40420442        s4addq  t1, t1, t1
  [mult_long.c:  24] 0x6c:      40620402        addq    t2, t1, t1
  [mult_long.c:  24] 0x70:      40840764        s8subq  t3, t3, t3
  [mult_long.c:  24] 0x74:      40820402        addq    t3, t1, t1
  [mult_long.c:  24] 0x78:      b4410000        stq     t1, 0(t0)
25: }
  [mult_long.c:  25] 0x7c:      6bfa8001        ret     zero, (ra), 1
```

[Return]

UMULH Example

[Return]

For example, to divide 10 by 3 (q=3, r=1), use UMULH to obtain the quotient in the high=order 64 bits:



The remainder is then computed as r = 10 - 3 * 3 = 1.

Integer Divide -- Listing File of Examples

[Return]

```
       divide1:
File 'divide_int.c':
Skipping source listing to line 2...
3: /*                                                  */
4: /* cc -migrate -O4 -source_listing -machine_code -c divide_long.c */
5: /*                                                  */
6: unsigned long n1 = 10;
7: unsigned long n2 = 9;
8: unsigned long n3 = 200;
9: unsigned long q = 0;
10: unsigned long r = 0;
11:
12: void divide1()
  [divide_int.c:  12] 0x0:      27bb0001        ldah    gp, 1(t12)
  [divide_int.c:  12] 0x4:      23bd8180        lda     gp, -32384(gp)
13: {
14:     unsigned long m = 3;
15:
16:     q = n1/m;
  [divide_int.c:  16] 0x8:      a79d8010        ldq     at, -32752(gp)
  [divide_int.c:  16] 0xc:      a43d8030        ldq     t0, -32720(gp)
17:     r = n1-m*q;
  [divide_int.c:  17] 0x10:     a4bd8018        ldq     t4, -32744(gp)
  [divide_int.c:  16] 0x14:     a45c0008        ldq     t1, 8(at)
  [divide_int.c:  16] 0x18:     a4210000        ldq     t0, 0(t0)
  [divide_int.c:  16] 0x1c:     a49d8020        ldq     t3, -32736(gp)
  [divide_int.c:  16] 0x20:     4c220602        umulh   t0, t1, t1
  [divide_int.c:  16] 0x24:     48403682        srl     t1, 0x1, t1
  [divide_int.c:  16] 0x28:     b4440000        stq     t1, 0(t3)
  [divide_int.c:  17] 0x2c:     40420563        s4subq  t1, t1, t2
  [divide_int.c:  17] 0x30:     40230521        subq    t0, t2, t0
  [divide_int.c:  17] 0x34:     b4250000        stq     t0, 0(t4)
18: }
  [divide_int.c:  18] 0x38:     6bfa8001        ret     zero, (ra), 1
  [divide_int.c:  18] 0x3c:     47ff041f        bis     zero, zero, zero
       divide2:
19:
20:
21: void divide2()
  [divide_int.c:  21] 0x40:     27bb0001        ldah    gp, 1(t12)
  [divide_int.c:  21] 0x44:     23bd8140        lda     gp, -32448(gp)
22: {
23:     q = n1/3;
  [divide_int.c:  23] 0x48:     a79d8010        ldq     at, -32752(gp)
  [divide_int.c:  23] 0x4c:     a43d8030        ldq     t0, -32720(gp)
24:     r = n1-3*q;
  [divide_int.c:  24] 0x50:     a4bd8018        ldq     t4, -32744(gp)
  [divide_int.c:  23] 0x54:     a45c0008        ldq     t1, 8(at)
  [divide_int.c:  23] 0x58:     a4210000        ldq     t0, 0(t0)
  [divide_int.c:  23] 0x5c:     a49d8020        ldq     t3, -32736(gp)
  [divide_int.c:  23] 0x60:     4c220602        umulh   t0, t1, t1
  [divide_int.c:  23] 0x64:     48403682        srl     t1, 0x1, t1
  [divide_int.c:  23] 0x68:     b4440000        stq     t1, 0(t3)
  [divide_int.c:  24] 0x6c:     40420563        s4subq  t1, t1, t2
  [divide_int.c:  24] 0x70:     40230521        subq    t0, t2, t0
  [divide_int.c:  24] 0x74:     b4250000        stq     t0, 0(t4)
25: }
  [divide_int.c:  25] 0x78:     6bfa8001        ret     zero, (ra), 1
  [divide_int.c:  25] 0x7c:     47ff041f        bis     zero, zero, zero
```

```
       divide3:
26:
27:
28: void divide3()
  [divide_int.c:  28] 0x80:     27bb0001        ldah    gp, 1(t12)
  [divide_int.c:  28] 0x84:     23bd8100        lda     gp, -32512(gp)
29: {
30:     q = n2/3;
  [divide_int.c:  30] 0x88:     a79d8010        ldq     at, -32752(gp)
  [divide_int.c:  30] 0x8c:     a43d8028        ldq     t0, -32728(gp)
31:     r = n2-3*q;
  [divide_int.c:  31] 0x90:     a4bd8018        ldq     t4, -32744(gp)
  [divide_int.c:  30] 0x94:     a45c0008        ldq     t1, 8(at)
  [divide_int.c:  30] 0x98:     a4210000        ldq     t0, 0(t0)
  [divide_int.c:  30] 0x9c:     a49d8020        ldq     t3, -32736(gp)
  [divide_int.c:  30] 0xa0:     4c220602        umulh   t0, t1, t1
  [divide_int.c:  30] 0xa4:     48403682        srl     t1, 0x1, t1
  [divide_int.c:  30] 0xa8:     b4440000        stq     t1, 0(t3)
  [divide_int.c:  31] 0xac:     40420563        s4subq  t1, t1, t2
  [divide_int.c:  31] 0xb0:     40230521        subq    t0, t2, t0
  [divide_int.c:  31] 0xb4:     b4250000        stq     t0, 0(t4)
32: }
  [divide_int.c:  32] 0xb8:     6bfa8001        ret     zero, (ra), 1
  [divide_int.c:  32] 0xbc:     47ff041f        bis     zero, zero, zero
       divide4:
33:
34: void divide4()
  [divide_int.c:  34] 0xc0:     27bb0001        ldah    gp, 1(t12)
  [divide_int.c:  34] 0xc4:     23bd80c0        lda     gp, -32576(gp)
35: {
36:     q = n2/100;
  [divide_int.c:  36] 0xc8:     a43d8028        ldq     t0, -32728(gp)
  [divide_int.c:  36] 0xcc:     a79d8010        ldq     at, -32752(gp)
37:     r = n2-100*q;
  [divide_int.c:  37] 0xd0:     a4fd8018        ldq     t6, -32744(gp)
  [divide_int.c:  36] 0xd4:     a4210000        ldq     t0, 0(t0)
  [divide_int.c:  36] 0xd8:     a47c0000        ldq     t2, 0(at)
  [divide_int.c:  36] 0xdc:     a4dd8020        ldq     t5, -32736(gp)
  [divide_int.c:  36] 0xe0:     44203102        bic     t0, 0x1, t1
  [divide_int.c:  36] 0xe4:     4c430602        umulh   t1, t2, t1
  [divide_int.c:  36] 0xe8:     4840d682        srl     t1, 0x6, t1
  [divide_int.c:  36] 0xec:     b4460000        stq     t1, 0(t5)
  [divide_int.c:  37] 0xf0:     405f0444        s4addq  t1, zero, t3
  [divide_int.c:  37] 0xf4:     4840f725        sll     t1, 0x7, t4
  [divide_int.c:  37] 0xf8:     40840764        s8subq  t3, t3, t3
  [divide_int.c:  37] 0xfc:     40a40524        subq    t4, t3, t3
  [divide_int.c:  37] 0x100:    40240521        subq    t0, t3, t0
  [divide_int.c:  37] 0x104:    b4270000        stq     t0, 0(t6)
38: }
  [divide_int.c:  38] 0x108:    6bfa8001        ret     zero, (ra), 1
  [divide_int.c:  38] 0x10c:    00000000        call_pal        halt
```

[Return]

Instruction Classes and Pipelines

[Return]

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)

[Return]

[Top of Document | Cookbook Home]

Local home | Digital home | Browse | Search | Feedback | Help

Legal

© Digital Equipment Corporation, 1996. The DIGITAL logo is a trademark of Digital Equipment Corporation. All other trademarks are the property of their respective owners.

Powered by Digital Alpha

Page last updated: "Aug 26 1996"