Series 32000® Assembly Language Optimizations

National Semiconductor
Application Note 636
October 1989

1.0 INTRODUCTION

In this applications note, we discuss various optimization strategies that apply to the NS32008, NS32016, NS32CG16 and NS32032 when writing assembly language programs. Most optimizations will also apply to the NS32332, NS32GX32 and NS32532. The emphasis will be on the NS32016 and NS32CG16.

2.0 DESCRIPTION

All Series 32000 processors have a common, rich instruction set. Chooosing the optimal instructions is often difficult. As a general rule when writing Series 32000 assembly language, try to minimize the number of instructions to accomplish a given task. In general, minimizing the size of code (or number of instructions) will also minimize the execution time, and maximize execution speed. For example:

```
     addr      scrl,r0   # point to source
     movb      0(r0),rl  # get the byte at (srcl)
     addr      destl,r0  # point to destination
     movb      rl,0(r0)  # store the byte at (destl)
```

This code simply moves a byte from one address to another. A better way is to use the Series 32000 memory to memory architecture.

```
     movb      srcl,destl   # get the byte at (srcl), place in (destl)
```

While this may appear obvious, note that since this code does not use registers, it may be used in an interrupt service routine without saving or restoring registers. This technique can also be used when a routine is ''out'' of registers for storing temporary values. Series 32000 can use any legal addressing mode for either the source or destination of most instructions.

Another technique to optimize programs is to make the fall through case of a branch instruction the most common case. This is an optimization because Series 32000 is pipelined, and a branch instruction breaks the pipeline (flushes the instruction execution queue). An example of this would be:

```
     cmpd      $l00,r0   # is the argument out of range?
     bgt       ok        # no, it is ok
     addr      err,tos   # place error message on tos
     br        err_hand  # handle the error
ok:  . . .               # code continues
```

Here, we are checking for an error condition (r0 $\geq$ 100). A better way to write this code is to branch TO the error condition:

```
     cmpd      $l00,r0   # is the argument out of range?
     ble       bad       # yes, branch out
     . . .               # code continues
bad: addr      err,tos   # place error message on tos
     br        err_hand  # handle the error
```

This technique can be further expanded as well, by common tail end expansion. If you have a section of code that reads as follows:

```
     if (cond) {
          process;
          process;
     } else {
          processl;
          processl;
     }
     com_proc;
     com_proc;
     com_proc; . . .
```

This can be optimized by unrolling the common code (com__proc) into each of the if cases. This eliminates at least one branch instruction. When code like this appears within loops, the savings in execution time can be considerable, at the expense of code space.

RRD-B30M75/Printed in U. S. A.

A simple optimization that can show significant benefits is the aligning of the target of branch instructions. For the NS32008, no alignment is required. For the NS32016, NS32GX32 and NS32CG16, .align 2 should be used. For the NS32032, .align 4 should be used. For the NS32332 and NS32532, .align 16 is best, but .align 4 can also be used. The .align 16 allows the NS32332, NS32GX32 and NS32532 to use the burst option, if the hardware supports this mode. The most important time to align instructions is the ''top'' of loops, as shown:

```
     addr     20,r0          # iterate 20 times
     movqd    0,rl           # zero accumulator
     .align   4              # align loop
lp:  addd     r0,rl          # add to accumulator
     addd     rl,rl          # again
     subd     r0,rl          # subtract
     acbd     $-l,r0,lp      # and loop
```

The .align 4 in the above example will fill the space between the movqd and the addd with a single, fast instruction (4 clocks or less) of length 1, 2, or 3 bytes, depending on alignment. The instruction inserted will, effectively, be a NOP in that it will affect no registers, memory or flags.

When loading 32-bit constants, the obvious technique is to use the MOVD instruction. On the NS32008, 016, CG16, 032 and 332, a better way may be to use the ADDR instruction. The reason this may be faster is that the ADDR instruction is shorter due to the encoding of the immediate value. If the MOVD instruction is not fully in the instruction queue, the ADDR will be faster. Below is a table suggesting the range that each instruction is best suited for. On the NS32532 and NS32GX32 the MOVD instruction should be used on all but the quick immediate range (−8 to +7).

Range	Instruction
−8 to +7	MOVQD $imm,dest
+8 to +8191	ADDR imm,dest
+8192 to 2^32	MOVD $imm,dest

Another useful technique is to make use of the pipeline overlap possible after instructions which reference memory in a Read-Modify-Write fashion. For example, on the NS32016 and NS32CG16, two register/register instructions (8 clock cycles) may be executed following a RMW instruction that affects 32 bits of memory, as shown:

```
     ord      r2,0(r0)       # place pattern in memory (RMW)
     addd     r6,r0          # add source warp
     addd     r7,rl          # add destination warp
```

If an instruction which references memory is executed following a RMW-type instruction, the new instruction will not be started until the write completes. If only register/register instructions are used, they will be executed DURING the write cycle of the RMW. In the above example, on the NS32016 and NS32CG16, both the addd instructions will be executed in parallel with the write cycle. This can only occur if the instructions are in the instruction execution queue. This is one more reason to minimize the use of the branch instruction, which flushes the queue.

Another technique is to make full use of the Complex Instruction Set nature of Series 32000. For example, take the following code fragment:

```
     cmpd     $l000,r7       # Is r7 > l000?
     ble      yes            # yes, it is
     movqd    $0,rl          # set rl to 0
     br       cont           # continue the code
yes: movqd    $l,rl          # set rl to l
cont: . . .
```

A better way to write this code is to use the Scond instruction, as follows:

```
     cmpd     $l000,r7       # Is r7 > l000?
     sled     rl             # rl = (r7 > l000) ? l : 0
cont: . . .
```

To do a three-operand add, of the form:

c = b + a + constant

the Series 32000 addr instruction can be used. For example:

```
     addr     326(r0)[rl:b],_C   # c = a + b + 326
```

An extension of this form can be used for adds that involve multiplies, when the multiplier is 1, 2, 4 or 8. By replacing the ''b'' in the above example with a '':w'', the r1 register can be multiplied by two before being added with r0 and 326. '':d'' will effect a multiply by 4, and '':q'' will multiply by eight. Note that the destination of the addr instruction need not be a register.

Another variation of this form may be used when it is required to add a large constant to a register, producing a result to a different variable, as follows:

```
     addr     320*4(r0),_start   # start = r0 + l280
```

The scaled index addressing mode may also be used for generation or checking of parity on characters. A 128-byte table containing the correct (even or odd) parity information in the most significant bit of each byte is created, then a simple OR can be used to set the parity:

```
     orb       partab[r0:b],r0    # set parity on byte in r0
```

An extension of this technique can be used on strings of characters, with the MOVST instruction. This can be used with variable length, null terminated strings, or fixed length strings. A 256-byte table containing the complete character set, with parity information (even, odd or none) is created. The MOVST instruction then copies the source string to a new destination buffer (optionally, it may replace the source string). During the copy, each source character is indexed into the table, and the corresponding character placed in the output buffer.

```
     movqd     -1,r0          # unlimited (very large) number of chars
     addr      source,r1      # source pointer
     addr      dest,r2        # destination pointer
     addr      cpartab,r3     # point to character parity table
     movqd     0,r4           # copy until a zero source character
     MOVST     U              # move string, with translation, until 0
     movqb     0,0(r2)        # place a null terminator as the last char
```

This technique should be used for variable length, null terminated strings.

A common optimization when multiplying by powers of two is to use a shift instruction. On the NS32008, NS32016, NS32CG16, NS32032 multiple add instructions should be used instead of a single shift, up to a shift of 6 (multiply by 64). On the NS32332, NS32GX32 and NS32532, use add instructions only for a multiply by two. When the quantity to be multiplied is in memory, however, a shift instruction should always be used, as it only does a single RMW access to memory.

```
# On the NS320xx processors, the following code should be used instead
# of MULD $16,r0 or LSHD $4,r0
     addd      r0,r0      multiply by 2
     addd      r0,r0      multiply by 4
     addd      r0,r0      multiply by 8
     addd      r0,r0      multiply by 16
```

A study of the Series 32000 Programmer's Reference Manual will show many other opportunities for optimizations using the full Series 32000 instruction set.

Lit. # 100636

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

National Semiconductor Corporation
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-8090
Tel: 1(800) 272-9959
TWX: (910) 339-9240

National Semiconductor GmbH
Livry-Gargan-Str. 10
D-82256 Fürstenfeldbruck
Germany
Tel: (81-41) 35-0
Telex: 527649
Fax: (81-41) 35-1

National Semiconductor Japan Ltd.
Sumitomo Chemical
Engineering Center
Bldg. 7F
1-7-1, Nakase, Mihama-Ku
Chiba-City,
Ciba Prefecture 261
Tel: (043) 299-2300
Fax: (043) 299-2500

National Semiconductor Hong Kong Ltd.
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductores Do Brazil Ltda.
Rue Deputado Lacorda Franco
120-3A
Sao Paulo-SP
Brazil 05418-000
Tel: (55-11) 212-5066
Telex: 391-1131931 NSBR BR
Fax: (55-11) 212-1181

National Semiconductor (Australia) Pty, Ltd.
Building 16
Business Park Drive
Monash Business Park
Nottinghill, Melbourne
Victoria 3168 Australia
Tel: (3) 558-9999
Fax: (3) 558-9998