bear them in mind and seek the advice of counsel when your enterprise enters new
distribution schemes.

Technology Export

The federal regulations governing the export of technology are in a constant state of flux,
due to the recent thawing of cold-war tensions between the United States and the nations of
Eastern Europe. Again, you should consult your attorney for the latest rules and regulations
governing the export of technology.

Bankruptcy

When a technology vendor liquidates its assets in a bankruptcy proceeding, its customer's
may not be able to obtain support for the vendor's software. Many buyers will insist that the
vendor place the product source code and any other documentation in an escrow, so that if
the vendor goes out of business, the buyers will have access to the necessary support
documentation.

Federal Contracts

Contracts with the federal government are controlled by the Federal Acquisition Regulations
(FAR), which appear in Chapter 48 of the Code of Federal Regulations. FAR contracts differ
from private sector contracts. In particular, FAR defines the rights in software that the
federal government normally expects to acquire. Also, the remedies available for various
breach situations are different under FAR than under private sector contracts. A federal
contracts specialist should be consulted if you plan on doing federal contract work.

Application Tuning for Speed and/or Size

Peter Gilmour
Motorola
Austin, Texas

Peter S. Gilmour is a manager/project leader/senior systems analyst with over seventeen
years of embedded systems experience and is currently employed by the
Motorola Advanced Microcontroller Group in Austin, Texas, where he works on the
JewelBox line of real time emulators. He has a BSEE from the Case Institute of
Technology and an MSEE from Arizona State University. Gilmour is the author of
numerous papers and articles.
Application Tuning for Speed and/or Size
by
Peter S. Gilmour

Embedded Systems Conference
Santa Clara Convention Center
September 23, 1992

Application tuning for performance is a two sided coin
EXECUTION SPEED versus CODE SIZE
Can only tune for one or the other, not both!
SPEED for real time or other performance criteria
CODE SIZE for very little ROM space remaining
This presentation focuses mainly on assembly
language, although some references to the C
language are also included

Fastest way to initialize 256 byte memory area?
- use 128 store word Instructions!

<table>
<thead>
<tr>
<th>68HC11 In Line Code</th>
<th>68HC11 Loop Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 STD LOCO00</td>
<td>100 LDX #LOCO00</td>
</tr>
<tr>
<td>101 STD LOC002</td>
<td>101 LOOP STD 0,</td>
</tr>
<tr>
<td>102 STD LOC004</td>
<td>103 INX</td>
</tr>
<tr>
<td></td>
<td>104 INX</td>
</tr>
<tr>
<td>127 STD LOC126</td>
<td>105 CPX #LOCO00+256</td>
</tr>
<tr>
<td></td>
<td>106 BNE LOOP</td>
</tr>
</tbody>
</table>

Speed: 5u X 128 = 640u
Size: 3 X 128 = 384 bytes

Fastest way to Initialize 256 byte memory area?

Use most efficient addressing mode possible
- consult RED entries on card
- direct addressing is most efficient
- extended addressing is next most efficient
- indexed addressing is least efficient
Some assemblers optimize for speed/size
- DIRECT addressing mode used if ASM knows
during pass 1 that address will fit in one byte
- 68332 "MOVE.L #0,00" optimized to "MOVEQ.L #0,00" (6 bytes optimized to 2 bytes)

Loop Control Optimization (applies to ASM and C)
- pre-load all constants outside of loops
- do all possible calculations outside of loops
- construct most efficient loop condition testing
  (decrement to zero)
- process elements in reverse order?
- 1u saved in a loop is multiplied by loop count N
- avoid special condition testing where possible
- use index register as a "switch": compute
  address of loop code to execute outside the loop
to avoid recalculation during each loop iteration
- better optimizing C compilers will automatically
do these types of loop optimizations

Loop Control Optimization (applies to ASM and C)
- pre-load all constants outside of loops
- do all possible calculations outside of loops
- construct most efficient loop condition testing
  (decrement to zero)
- process elements in reverse order?
- 1u saved in a loop is multiplied by loop count N
- avoid special condition testing where possible
- use index register as a "switch": compute
  address of loop code to execute outside the loop
to avoid recalculation during each loop iteration
- better optimizing C compilers will automatically
do these types of loop optimizations

Study MPU/MCU User's Manual (registers &
instruction set)
Study MPU/MCU reference card
- mark fastest execution Instructions in RED
- mark fewest byte Instructions in YELLOW
- mark a spot with the two colors and label as KEY

HOW TO START TUNING YOUR APPLICATION

APPLICATION TUNING FOR SPEED
Faster speed usually means fewer Instruction bytes
- multiply (MUL) instruction is big exception!
- seek Instructions with fewest cycles (RED)
Fastest code is "In line" code
- no loops
- no subroutines
- each Instruction does a job, i.e., no overhead!

APPLICATION TUNING FOR SIZE
Use most efficient addressing mode possible
- consult RED entries on card
- direct addressing is most efficient
- extended addressing is next most efficient
- indexed addressing is least efficient
Some assemblers optimize for speed/size
- DIRECT addressing mode used if ASM knows
during pass 1 that address will fit in one byte
- 68332 "MOVE.L #0,00" optimized to "MOVEQ.L #0,00" (6 bytes optimized to 2 bytes)

Keep operands in registers wherever possible
- memory accessing is slower than register accessing

APPLICATION TUNING FOR SPEED AND/OR SIZE
Seek best algorithm for the task and MCU
- dramatic speed Improvements can be obtained
  by simply using a better algorithm
  for example, character conversion can best be
done by constructing a table of the conversion
  values and simply using the entry value as an
  Index to retrieve the converted value

APPLICATION TUNING FOR SIZE
Use most efficient addressing mode possible
- consult RED entries on card
- direct addressing is most efficient
- extended addressing is next most efficient
- indexed addressing is least efficient
Some assemblers optimize for speed/size
- DIRECT addressing mode used if ASM knows
during pass 1 that address will fit in one byte
- 68332 "MOVE.L #0,00" optimized to "MOVEQ.L #0,00" (6 bytes optimized to 2 bytes)
Application Tuning for Speed and/or Size
by
Peter S. Gilmour

Embedded Systems Conference
Santa Clara Convention Center
September 23, 1992

Application tuning for performance is a two sided coin
EXECUTION SPEED versus CODE SIZE
Can only tune for one or the other, not both!
SPEED for real time or other performance criteria
CODE SIZE for very little ROM space remaining

Application tuning for performance is a two sided coin
EXECUTION SPEED versus CODE SIZE
Can only tune for one or the other, not both!
SPEED for real time or other performance criteria
CODE SIZE for very little ROM space remaining

This presentation focuses mainly on assembly
language, although some references to the C
language are also included

Fastest way toinitialize 256 byte memory area?
- use 128 store word Instructions!

<table>
<thead>
<tr>
<th>16H</th>
<th>134 124 140 154 132 151 147 126</th>
</tr>
</thead>
<tbody>
<tr>
<td>16A</td>
<td>136 125 141 155 133 152 148 127</td>
</tr>
<tr>
<td>16B</td>
<td>137 126 142 156 134 153 149 128</td>
</tr>
</tbody>
</table>

How many instructions to execute all 128 memory locations?
64 instructions

64 instructions

Loop Control Optimization (applies to ASM and C)
- pre-load all constants outside of loops
- do all possible calculations outside of loops
- construct most efficient loop condition testing (decrement to zero)
- process elements in reverse order?
-1u ... by loop count N
- avoid special condition testing where possible
- use Index register as a "switch": compute
address of loop code to execute outside the loop
to avoid recalculation during each loop iteration
- better optimizing C compilers will automatically
do these types of loop optimizations

Seek best algorithm for the task and MCU
- dramatic speed improvements can be obtained
by simply using a better algorithm
- for example, character conversion can best be
done by constructing a table of the conversion
values and simply using the entry value as an
index to retrieve the converted value

Keep operands in registers wherever possible
- memory accessing is slower than register accessing

Use most efficient addressing mode possible
- consult RED entries on card
- direct addressing is most efficient
- extended addressing is next most efficient
- indexed addressing is least efficient

Some assemblers optimize for speed/size
- DIRECT addressing mode used if ASM knows
during pass 1 that address will fit in one byte
- 68332 "MOVE.L #0,0" optimized to "MOVEQ.L
#0,0" (6 bytes optimized to 2 bytes)

Faster speed usually means fewer Instruction bytes
- multiply (MUL) Instruction is big exception!
- seek Instructions with fewest cycles (RED)

Fastest code is "In line" code
- no loops
- no subroutines
- each Instruction does a job, i.e., no overhead!

Study MPU/MCU reference card
- mark fastest execution Instructions in RED
- mark fewest byte Instructions in YELLOW
- mark a spot with the two colors and label as KEY

This presentation focuses mainly on assembly
language, although some references to the C
language are also included

Use most efficient addressing mode possible
- consult RED entries on card
- direct addressing is most efficient
- extended addressing is next most efficient
- indexed addressing is least efficient

Some assemblers optimize for speed/size
- DIRECT addressing mode used if ASM knows
during pass 1 that address will fit in one byte
- 68332 "MOVE.L #0,0" optimized to "MOVEQ.L
#0,0" (6 bytes optimized to 2 bytes)

Faster speed usually means fewer Instruction bytes
- multiply (MUL) Instruction is big exception!
- seek Instructions with fewest cycles (RED)

Fastest code is "In line" code
- no loops
- no subroutines
- each Instruction does a job, i.e., no overhead!

Study MPU/MCU reference card
- mark fastest execution Instructions in RED
- mark fewest byte Instructions in YELLOW
- mark a spot with the two colors and label as KEY

This presentation focuses mainly on assembly
language, although some references to the C
language are also included

Use most efficient addressing mode possible
- consult RED entries on card
- direct addressing is most efficient
- extended addressing is next most efficient
- indexed addressing is least efficient

Some assemblers optimize for speed/size
- DIRECT addressing mode used if ASM knows
during pass 1 that address will fit in one byte
- 68332 "MOVE.L #0,0" optimized to "MOVEQ.L
#0,0" (6 bytes optimized to 2 bytes)
APPLICATION TUNING FOR SPEED AND/OR SIZE

Instruction Modification
- changes instruction in RAM and then executes it
- very risky
- if used in ROM based application, code must be moved to RAM before execution
- document very carefully; use "0-0" unique string in operand field plus text comments
- only use when absolutely necessary; sometimes it is the only way to accomplish a task

68HC05 Instruction Modification Example
- accumulator (A) and Index (X) register are both only 8-bits

- get byte specified at A,X address subroutine:
  - entry: A,X:: address 01 byte to get (A:: MS, X = LS)
  - exit: A:: byte value at specified address
  - X:: preserved

- set X= SOURCE addr
- set Y = DEST addr
- set B = DEST addr
- get SOURCE byte and move to DEST addr
- count 1 byte moved
- continue until all bytes moved

Macros generate "in line" code
- macro is a named sequence that is substituted at assembly time when the name is invoked in the opcode field
- parameters can be passed to the macro
- can be used with assembler and C
- gives the benefits of a subroutine without the overhead
- faster code execution
- increased productivity
- lower maintenance
- ease of porting to new MCU

APPLICATION TUNING FOR SIZE

Lowest code size always achieved by using the fewest and smallest instructions possible
- seek instructions with fewest bytes (YELLOW)
- maximize use of loops and subroutines
- minimize use of macros
- eliminate non-functional instructions, i.e., branch always (BRA) and jump (JMP)
- seek longest instructions (3+ bytes) and replace with shorter ones where possible

Code size reduction techniques are bad programming
- very complex to document properly
- hard to maintain
- more prone to error
- introduction
- very limited in older ROM designs where modifications or fixes are required and insufficient reserve memory space exists

APPLICATION TUNING FOR SIZE

Instruction and data caches can dramatically improve execution speed, especially when a loop fits entirely within the instruction cache
- do not manipulate caches after enabling; they operate best by leaving them as designed by the manufacturer
- cache hits reduce external bus activity which allows better access (bandwidth) to other bus masters

APPLICATION TUNING FOR SIZE

Macros generate "in line" code
- macro is a named sequence of that is substituted at assembly time when the name is invoked in the opcode field
- parameters can be passed to the macro
- can be used with assembler and C
- gives the benefits of a subroutine without the overhead
  - faster code execution
  - increased productivity
  - lower maintenance
  - ease of porting to new MCU

APPLICATION TUNING FOR SIZE

Lowest code size always achieved by using the fewest and smallest instructions possible
- seek instructions with fewest bytes (YELLOW)
- maximize use of loops and subroutines
- minimize use of macros
- eliminate non-functional instructions, i.e., branch always (BRA) and jump (JMP)
- seek longest instructions (3+ bytes) and replace with shorter ones where possible

APPLICATION TUNING FOR SIZE

Code size reduction techniques are bad programming
- very complex to document properly
- hard to maintain
- more prone to error introduction
- very costly in terms of programming time
- very non-structured
- may cause slower execution

Memory is inexpensive today!
Code size reduction should only be used in older ROM designs where modifications or fixes are required and insufficient reserve memory space exists
APPLICATION TUNING FOR SPEED AND/OR SIZE

Instruction Modification
- changes instruction in RAM and then executes it
- very risky
- if used in ROM based application, code must be moved to RAM before execution
- document very carefully; use "0-0" unique string in operand field plus text comments
- only use when absolutely necessary; sometimes it is the only way to accomplish a task

--- APPLICATION TUNING FOR SPEED AND/OR SIZE ---

Macros generate "in line" code
- macro is a named sequence that is substituted at assembly time when the name is invoked in the opcode field
- parameters can be passed to the macro
- can be used with assembler and C
- gives the benefits of a subroutine without the overhead
  + faster code execution
  + increased productivity
  + lower maintenance
  + ease of porting to new MCU

--- APPLICATION TUNING FOR SPEED AND/OR SIZE ---

Instruction Modification Example
- accumulator (A) and Index (X) register are both only 8-bits

```
100 * Get byte specified at A,X address subroutine:
101 * Entry: A,X:: address 01 byte to get (A :: MS, X = LS)
102 * Exit: A:: byte value at specified address
103 * X = preserved
104 *
105 GETAX STA MSADR Store MS address byte
106 STX LSADR Store LS address byte
107 LDA EFFFF+0 O get byte at address (Instr. mod.)
108 * Note: "EFFF" used above to force Extended Addressing:
109 MSADR EQU *·2 MS address + 2nd byte of instr.
110 LSADR EQU *·1 LS address + 3rd byte of instr.
111 RTS
```

--- APPLICATION TUNING FOR SPEED AND/OR SIZE ---

Move Memory macro (68HC11 assembler):
```
100 * Calling Sequence: MOVE SOURCE,DEST,CNT
101 * where: SOURCE = "from" location
102 * DEST = "to" location
103 * CNT = nbr of bytes to move 1-255 (0 ==> 256)
104 * Destroys A,B,X,Y registers
105 MOVE LDX #1 Set X SOURCE addr
106 LDY #3 Set Y DEST addr
107 LDA #0 Set SOURCE byte and
108 STA #0 move to DEST addr
109 DECY Count 1 byte moved
110 SNE Continue until all bytes moved
111 ENDM
```

--- APPLICATION TUNING FOR SIZE ---

Lowest code size always achieved by using the fewest and smallest instructions possible
- seek Instructions with fewest bytes (YELLOW)
- maximize use of loops and subroutines
- minimize use of macros
- eliminate non-functional Instructions, i.e., branch always (BRA) and jump (JMP)
- seek longest Instructions (3+ bytes) and replace with shorter ones where possible

--- APPLICATION TUNING FOR SIZE ---

Code size reduction techniques are bad programming
- very complex to document properly
- hard to maintain
- more prone to error
- very costly in terms of programming time
- very non-structured
- may cause slower execution

Memory is inexpensive today!
Code size reduction should only be used in older ROM designs where modifications or fixes are required and insufficient reserve memory space exists
APPLICATION TUNING FOR SPEED AND/OR SIZE

SUBROUTINE OPTIMIZATION FOR SIZE
Scan program for common code segments that can be combined into a subroutine
- be careful the added bytes for the subroutine calls (JSR SUBR) don't negate the bytes saved; total bytes must be less!
All return from subroutine (RTS) opcodes are equal
- you can branch to any RTS to return!
All good subroutines start in the middle
- might be more efficient to conditionally branch back before the entry point (eliminates unconditional branch/jump at the end)
Reorganize to eliminate RTS by having subroutines cascade into one another

BEFORE CASCADING SUBROUTINES

AFTER CASCADING SUBROUTINES

APPLICATION TUNING FOR SPEED AND/OR SIZE

Search for more efficient coding
- following 68HC11 hardware initialization example saves one byte by combining the two LDA instructions into one LDD instruction (D register a concatenated A and B) with a STB instruction

Before

LDA #$30
STA CTRLREG
LDA #$10
STA CTRLREG

After

LDA #$30
LDD #$3010

Use "SKIP" opcode instead of "branch always" to skip one or two instruction bytes
- non-destructive opcode combines with the skipped byte(s) as an address to make a "skip" instruction (see 68HC11 example below)

Before

100 LDA #$30
101 STA CTRLREG
102 LDA #$10
103 STA CTRLREG

After

100 LDA #$30
101 LDD #$3010
102 SUBR2 LDA #2
103 COM EQU '
104 'other code'
105 RTS

Optimize end-of-text and end-of-table codes
- these markers serve no useful purpose other than to mark the end of some message text or of a table
- you can pick any value you want for these markers; 500 or SFF are popular
- why not pick any negative two's complement value, since half the opcodes for a given processor are negative? then the marker becomes the next opcode!
- must document thoroughly
- watch out for porting problems to another processor

Before

100 JSR SUBR3
103 'other code'
110 JSR SUBR3
113 'other code'
120 JSR SUBR3
123 'other code'

After

100 JSR SUBR3
110 JSR SUBR3
120 JSR SUBR3

APPLICATION TUNING FOR SPEED AND/OR SIZE

Avoid unnecessary instructions
- short versus long branches or jumps
- shorter addressing modes
Replace "extended" or "absolute" addressing instructions with direct or Index addressing wherever possible
Eliminate all unnecessary instructions

Before

100 JSR SUBR1
101 STA CTRLREG
102 LOA #$10
103 STA CTRLREG

After

100 LOA #$10
102 STA CTRLREG
101 STA CTRLREG

APPLICATION TUNING FOR SPEED AND/OR SIZE

Optimize code flow so smaller instructions can be used
- watch out for porting problems to another
- must document thoroughly

Before

500 SUBR3 EQU ' 501 'other code'
502 RTS

After

500 SUBR3 EQU ' 501 'other code'
502 'Fast load SUBR'

APPLICATION TUNING FOR SPEED AND/OR SIZE

Condition Code (CC) Register Techniques
Use condition code register as temporary flag
- set carry (C) bit as flag, then can carefully use any other non-carry affecting instruction until you need to check the flag
Eliminate specific CC setting/clearing instructions by carefully examining all instructions that set or clear the bit(s) you need and chose one of these instead of another instruction already in use
APPLICATION TUNING FOR SPEED AND/OR SIZE

SUBROUTINE OPTIMIZATION FOR SIZE
Scan program for common code segments that can be combined into a subroutine
- be careful the added bytes for the subroutine calls (JSR SUBR) don’t negate the bytes saved; total bytes must be less!
All return from subroutine (RTS) opcodes are equal
you can branch to any RTS to return!
All good subroutines start in the middle
might be more efficient to conditionally branch back before the entry point (eliminates unconditional branch/jump at the end)
Reorganize to eliminate RTS by having subroutines cascade into one another

BEFORE CASCADING SUBROUTINES

AFTER CASCADING SUBROUTINES

Search for more efficient coding
- following 68HC11 hardware initialization example saves one byte by combining the two LDA instructions into one LDD instruction (D register a concatenated A and B) with a STB instruction

Use "SKIP" opcode instead of "branch always" to skip one or two instruction bytes
- non-destructive opcode combines with the skipped byte(s) as an address to make a "skip" instruction (see 68HC11 example below)

Optimize end-of-text and end-of-table codes
- these markers serve no useful purpose other than to mark the end of some message text or of a table
you can pick any value you want for these markers; $00 or $FF are popular
- why not pick any negative two’s complement value, since half the opcodes for a given processor are negative? then the marker becomes the next opcode!
- must document thoroughly
- watch out for porting problems to another processor

CONDITION CODE (CC) REGISTER TECHNIQUES
Use condition code register as temporary flag
- set carry (C) bit as flag, then can carefully use any other non-carry affecting instruction until you need to check the flag
Eliminate specific CC setting/clearing instructions by carefully examining all instructions that set or clear the bit(s) you need and chose one of these instead of another instruction already in use
Use "unassigned" address bits to your advantage
- for example, 68332 uses 24-bit addresses
which normally require four bytes for storage,
but it also has absolute short addressing (16-bit
sign extended address)

$FFFFFAOO

MOVE.W MCR,DO

Do not use any undocumented instructions
(unassigned opcodes) for application tuning
- manufacturer can alter these at will, even
between different mask sets of the same
processor

THE END