C55x Code Generation Tools

Tips and Tricks

Topics

- Performance
  - Compilation Options
  - Data Types
  - Efficient Loop Control & Indexing
  - Efficient Multiply
  - Efficient Loops
  - Enabling Dual MAC
  - Intrinsics
  - C Idioms
  - Circular Addressing
- Code size
  - Compilation Options
  - Unused function elimination
  - Overlays
- Convenience/Information
  - Diagnostic control pragmas and options
  - Optimization information
  - Function code alignment

Compile Options for Performance

- -o3: highest optimization level; enables automatic inlining
  - Inlining trades space for speed and saves on branch latencies
- -oi*size* (with –o3): set auto-inline threshold
  - Higher sizes allow inlining of larger functions (default is 1)
- -pm: program mode compiles multiple files together
  - Gives compiler a full program view (e.g. cross-file inlining)
  - May not have access to source files in other subsystems
  - Increases compile time
- -op2 (with –pm): assert no other uses/calls of C funcs & vars
  - This is the default but if only compiling a subsystem, may need to consider other values of -op
- -v*device:rev*: select target hardware device(s)
  - Default will compile code for all C55x revisions (not P2)
  - Can use one or more –v options to assure minimum silicon bug workaround code and full use of available instructions

Optimization Levels

Statement (-o0)	Basic Block (-o1)
•Simplifies control flow •Allocates variables to registers •Eliminates unused code •Simplifies expressions and statements •Expands calls to inline functions	•Performs local copy/constant propagation •Removes unused assignments •Eliminates local common expressions
Function (-o2)	**File (-o3)**
•Performs loop optimizations •Eliminates global common sub-expressions •Eliminates global unused assignments •Performs loop unrolling	•Removes functions that are never called •Simplifies functions with return values that are never used •Inlines calls to small functions (regardless of declaration) •Reorders functions so that attributes of called function are known when caller is optimized •Identifies file-level variable characteristics

-op Options

Indicates if other modules can call this module's global functions or modify the module's global variables

	Use this option
functions are called + variables are modified	-op0
functions are not called + variables are modified	-op1
functions are not called + variables are not modified	-op2
functions are called + variables are not modified	-op3

PRAGMAs for Performance

Ways to tell the optimizer more about function behavior

PRAGMA	Description
FUNC_CANNOT_INLINE (func)	function cannot be inlined
FUNC_EXT_CALLED (func)	function may be called by external function
FUNC_IS_PURE (func)	function has no side effects
FUNC_IS_SYSTEM (func)	functions has same behavior as ANSI function with same name
FUNC_NEVER_RETURNS (func)	function never returns
FUNC_NO_GLOBAL_ASG (func)	functions makes no assignments to global variables and contains no asm statements
FUNC_NO_IND_ASG (func)	function makes no assignments through pointers and contains no asm statements

Compile Options for Performance

And avoid…

- -g: debugging
- -s, -ss: source interlisting
- -ms: optimize for space rather than speed
- -mr: disable generation of hardware loop instructions

But if you must, then use…

- -mn: allow code motion optimizations even with –g etc.

Best…

cl55 –o3 –oi50 –pm –op2 –vcpu:2.2 *.c

Typical…

cl55 –o3 –oi50 –vcpu:2.2 –g –mn file.c

C55x is a 16-bit DSP

- Complete instruction set for 16-bit data
- Loop control registers are 16 bits
- Indexed addressing uses 16-bit values
- Multiplies are 16x16 (actually 17x17)
  - But does have 32-/40-bit accumulator registers

So…

- Use 16-bit C types for: loop control variables, indexing expressions, basic data
- Accumulate into 32- or 40-bit variables
  - Can maintain accuracy

C Data Types

For 16-bit data use…

- int: 16 bits on C55x, but not on other targets
- short: also 16 bits on C55x, and will probably have better portability
- int16_t (from stdint.h): for best portability

Signed is **much** better than unsigned for loop control and indexing

- Unless you need the defined behavior on overflow
- C has looser rules for signed and thus allows greater flexibility for optimization

Efficient Loop Control/Indexing

```
void clear(int *a, int n) {
    int i;
    for (i=0; i<n; i++) a[i] = 0
}
```

int

- Zero-overhead hardware loop
- Auto-increment address arithmetic
- Loop body: 1 cycle

```
repeat (CSR)
    *AR0+ = #0
```

IDEAL

NOT Efficient Loop Control/Indexing

```
void clear(int *a, unsigned int n) {
     unsigned int i;
     for (i=0; i<n; i++) a[i] = 0
}
```

unsigned int (& not Small Mem Model)

- Zero-overhead hardware loop
- Address arithmetic in ACs
  - Or for Rev 3, in XARs
- Must implement modular arithmetic
  - C language has defined semantics for unsigned "overflow"
- Loop body: 6 + 2 = 8 cycles
  - Or for Rev 3: 7 cycles

```
AR1 = #0                 ; AR1 holds 'i'
localrepeat {
    AC1 = AR1 & 0xffff
    AC0 = XAR0           ; XAR0 holds 'a'
    AC0 = AC0 + AC1
    XAR3 = AC0
    AR1 = AR1 + #1
    *AR3 = #0 }
```

NOT Efficient Loop Control/Indexing

```
void clear(int *a, long n) {
    long i;
    for (i=0; i<n; i++) a[i] = 0
}
```

long (& not Small Mem Model)

- if…goto loop control
  - 16-bit hardware loop counters
- Address arithmetic in ACs
  - Or for Rev 3, in XARs
- Loop control in AC
- Loop body: lots of cycles

```
    AC1 = 0             ; AC1 holds 'i'
L:  AC2 = XAR0          ; XAR0 holds 'a'
    AC2 = AC2 + AC1
    XAR3 = AC2
    AC1 = AC1 + #1
    TC1 = (AC1 < AC0); AC0 holds 'n'
    *AR3 = #0
    if (TC1) goto L
```

Efficient Multiply

- Usual intention is 16 x 16 → 32 multiply
- C gives you these choices
  - int x int → int 16 x 16 → 16
  - long x long → long 32 x 32 → 32 (done in RTS on C55x!)
- Do NOT write

```
long_var = short_var1 * short_var2;                // WRONG!
long_var = (long)(short_var1 * short_var2);        // Also WRONG!
```

- DO write

```
long_var = (long)short_var1 * (long)short_var2; // OK
long_var = (long)short_var1 * short_var2;       // Also OK
```

  - Accurate representation of what you want
  - Efficient implementation

Efficient Loops

Goal: Generate hardware loops (block/local/repeat)

- Use 16-bit signed loop control variable (see above)
- Call in body (usually) means no hardware loop
- At most three, normally only two, levels of hardware loops
- Smaller body gives better chance at localrepeat or repeat single
- For hardware loops, must be able to compute "trip count" (# of times loop executes) before entering loop
- The more the compiler knows about the trip count the more it can do
  - Use pragma MUST_ITERATE(min, max, mod)
  - Assert things about the bounds

Efficient Loops

MUST_ITERATE(min, max, mod)

- min > 0 ➔ can ignore zero-iteration case
- mod == n ➔ can unroll loop n times

```
#pragma MUST_ITERATE(1,,2); // iterates at least once and
for (i = 0; i < n; i++)     // an even number of times
```

assert <predicate>

- Standard C
- Run-time check of the predicate
- Can be deleted (-DNDEBUG)
- Can be converted to _nassert (-DNASSERT)

_nassert <predicate>

- Tells compiler that predicate holds (no check)

Enabling Dual MAC

- Do the right multiply (see above)
- Meet hardware requirements
  - Two consecutive MACs (MASs or MPYs) producing distinct results
  - All multiplicands in memory
  - Share one operand of the multiply
  - Shared operand must be in "on chip" memory
    - Use –mb or onchip keyword to tell compiler
  - Allocate operands in memory such that all three accesses may be done simultaneously
    - Use pragma DATA_SECTION to aid data placement

```
long_t1 += ((long)*a * *coef)
long_t2 += ((long)*b * *coef)
```

Enabling DualMAC - example

```
void fir(short *x, short *h,
         short *y,
         int m, int n)
{
   int i, j;
   long y0;

   for (i = 0; i < m; i++)
   {
      y0 = 0;

      for (j = 0; j < n; j++)
         y0 += (long)x[i + j] * h[j];

      y[i] = (short)(y0 >> 16);
   }
}
```

Enabling DualMAC - example

```
void fir(short *x, onchip short *h,
         short *y,
         int m, int n)
{
   int i, j;
   long y0, y1;
   for (i = 0; i < m; i+=2)
   {
      y0 = 0;
      y1 = 0;
      for (j = 0; j < n; j++)
      {
         y0 += (long)x[i + j]     * h[j];
         y1 += (long)x[i + 1 + j] * h[j];
      }
      y[i]   = (short) (y0 >> 16);
      y[i+1] = (short) (y1 >> 16);
   }
}
```

- Adjacent MACs
- Shared “onchip” operand

Enabling Dual MAC

- Compiler can do loop transformation to create a dual MAC situation with help from programmer
  - Nested efficient loops (see above)
    - outer loop guaranteed to have even trip count
    - inner loop guaranteed to execute
  - Single MAC in inner loop with memory multiplicands
  - Output does not overlap with inputs
    - use restrict keyword to tell compiler
      - restrict says pointer is only access path to underlying memory
    - most useful to restrict pointers used in memory writes
  - One multiplicand depends at most on inner loop control variable
    - Use –mb or 'onchip' to indicate allocation in on-chip memory
  - No control-flow in outer loop
  - Inner loop bounds constant wrt outer loop

DSP™ TEXAS INSTRUMENTS TECHNOLOGY

Enabling DualMAC - example

```
void fir(short *x, short *h,
         short * y,
         int m, int n)
{
   int i, j;
   long y0;

   for (i = 0; i < m; i++)
   {
      y0 = 0;

      for (j = 0; j < n; j++)
         y0 += (long)x[i + j] * h[j];

      y[i] = (short) (y0 >> 16);
   }
}
```

DSP™ TEXAS INSTRUMENTS TECHNOLOGY

Enabling DualMAC - example

```
void fir(short *x, onchip short *h,
         short * restrict y,
         int m, int n)
{
   int i, j;
   long y0;

   for (i = 0; i < m; i++)
   {
      y0 = 0;

      for (j = 0; j < n; j++)
         y0 += (long)x[i + j] * h[j];

      y[i] = (short) (y0 >> 16);
   }
}
```

Enabling DualMAC - example

```
void fir(short *x, onchip short *h,
         short * restrict y,
         int m, int n)
{
   int i, j;
   long y0;
   #pragma MUST_ITERATE(1,,2)
   for (i = 0; i < m; i++)
   {
      y0 = 0;
      #pragma MUST_ITERATE(1)
      for (j = 0; j < n; j++)
         y0 += (long)x[i + j] * h[j];

      y[i] = (short) (y0 >> 16);
   }
}
```

Enabling DualMAC - example

```
void fir(short *x, onchip short *h,
         short * restrict y,
         int m, int n)
{
   int i, j;
   long y0;
   _nassert((m >= 1) && ((m % 2) == 0));
   for (i = 0; i < m; i++)
   {
      y0 = 0;
      _nassert(n >= 1);
      for (j = 0; j < n; j++)
         y0 += (long)x[i + j] * h[j];

      y[i] = (short) (y0 >> 16);
   }
}
```

Enabling DualMAC - example

```
void fir(short *x, onchip short *h,
         short * restrict y,
         int m, int n)
{
   int i, j;
   long y0;
   assert((m >= 1) && ((m % 2) == 0));
   for (i = 0; i < m; i++)
   {
      y0 = 0;
      assert(n >= 1);
      for (j = 0; j < n; j++)
         y0 += (long)x[i + j] * h[j];

      y[i] = (short) (y0 >> 16);
   }
}
```

Compile with –DNASSERT

assert ➔ _nassert

Intrinsics

- Functional notation; maps to single instruction
- Use instead of asm(…)
- Intrinsics are the "right" way to access DSP features C/C++ does not support
  - Saturation
  - Rounding
  - Fractional
  - Complex, powerful C55x instructions
    - firs, absdst, sqdst, lms, …
- <u>Intrinsics do not disrupt optimization</u>

C Idiom Recognition

- Standard C expression resulting in extremely efficient C55x code
- Examples (complete list in documentation)
  - Fractional Multiply

```
long l; int  i,j;           HI(AC1)=T0 || bit(ST1,ST1_FRCT)=#1
l = ((long)i * j) << 1;   AC0 = AC1 * T1
```

  - Bi-directional Shift

```
long v; int sh;
(sh > 0) ? v << sh : v >> -sh   AC1 = AC0 << T1
```

  - Min/Max

```
(a > b) ? a : b                  AR1 = max(T0, T1)
```

  - Abs

```
(a < 0) ? -a : a)                AR1 = |T0|
```

Circular Addressing: Example

C Code:

```
int a[10], i = 0, j;
for (j = 0; j < 20; j ++)   (1) start of lifetime
{
   ... a[i] ...
   i = (i + 1) % 10;
}                           (2) end of lifetime
```

Assembly Code:

```
     BSA01 = <initial address of "a">
     AR0 = #0
     BK03 = #10
  || bit(ST2, #ST2_AR0LC) = 1  (1) start
     repeat(#19)
        ... *AR0+ ...
     bit(ST2, #ST2_AR0LC) = 0  (2) end
```

Circular Addressing: Fine Print

In a region *including a loop* the compiler can recognize references to an array A indexed by a variable I as a circular buffer of size S if the following hold:

- *I* is initialized with a positive constant.
- All modifications of I in the region are "modulo S" increments (always followed by '% S') for a constant S.
- Increments of *I* are always by positive constants.
- Index expressions of a reference are of the form 'a*I+b' where a and b are positive constants.

Compile Options for Code Size

- -ms favors size over speed optimization
  - Some changes in instruction selection
  - No loop unrolling
  - Less code hoisted out of loops
  - Fewer predicated instructions
- -mo puts each function, f, in object subsection, .text:_f, and marks it as conditional
  - Linker will not include the subsection unless it is referenced
  - Beware: now all calls are "long"
    - Many intra-file calls & few unused functions → size grows!
    - Solution (future): linker-generated trampolines
      - Calls left as short and fixed by linker as necessary

Support for Overlays

- Overlay Management

```
UNION {
   os1: { task1.obj(.text) } load > LDMEM
   os2: { task2.obj(.text) } load > LDMEM
      } run = RNMEM
```

  - Single run address; separate load addresses
  - Code must be copied before execution
  - Copy routine needs: run/load addresses and sizes
  - Can use –mo or pragma CODE_SECTION(func, “sect”) to aid in placement
- Use linker-generated “copy tables” to describe allocation
  - Supports auto-split of both load and run allocations
- The directive creates the table entry

```
os1:{ task1.obj(.text) } load > LDMEM, table(os1_ctbl)
os1:{ task2.obj(.text) } load > LDMEM, table(os2_ctbl)
```

- General purpose copy routine in RTS

```
copy_in(&os1_ctbl); /* copy from load to run address */
```

DSP™ TEXAS INSTRUMENTS TECHNOLOGY

Diagnostic Control

- Want less? Use -pdw to turn off warnings
- Want more? Use -pdr to turn on remarks
- Want more context? Use -pdv to display source
- Want something else? Use -pden to get diagnostic id's then use these …

Pragma	Option	Meaning
`pragma diag_suppress id`	`-pds=id`	Suppress id
`pragma diag_remark id`	`-pdsr=id`	Treat id as remark
`pragma diag_warning id`	`-pdsw=id`	Treat id as warning
`pragma diag_error id`	`-pdse=id`	Treat id as error
`pragma diag_default id`	N/A	Reset id to default

Diagnostic Control

```
/* Little program */
/* Big problems   */

hope(int m) {
   int a = v;
   int b = f(a);
   return b;
 }
```

Compile in "strict ANSI" mode

```
cl55 diag.c -ps
line 5: error: identifier "v" is undefined
```

Is that all?

Well, technically, yes.

Diagnostic Control

```
/* Little program */
/* Big problems   */

hope(int m) {
   int a = v;
   int b = f(a);
   return b;
 }
```

Turn on remarks

```
cl55 diag.c -pdr
line 4: remark: explicit type is missing ("int" assumed)
line 5: error: identifier "v" is undefined
line 6: remark: function declared implicitly
line 4: remark: parameter "m" was never referenced
```

Diagnostic Control

```
#pragma diag_error 225 /* Require explicit function decls   */
#pragma diag_error 262 /* Require explicit func return type */

hope(int m) {
   int a = v;
   int b = f(a);
   return b;
 }
```

No remarks, just errors

```
cl55 diag.c
line 4: error: explicit type is missing ("int" assumed)
line 5: error: identifier "v" is undefined
line 6: error: function declared implicitly
```

Getting Info on Your Program

- -k: Keeps generated assembly
  - Contains info on each function: Register usage, stack usage, frame size, …
- -os: Adds optimizer comments to generated asm code
  - Comments are C-source-like description of the program after re-arrangement by the optimizer
- -on1, -on2: .nfo file of optimizer decisions (e.g. to inline or not)
  - -on2 produces a more verbose file
  - Some information in this file for each function …
    - "size" of function in terms the units used in –oi to set the inlining threshold
    - Known callers
    - Called functions
    - Et c.
- -s, -ss: C source interleaved in generated asm code
  - Can reduce amount of optimization

Function Code Alignment

- Some code sequences (esp in hardware loops) can have timing vary depending on alignment of the code in memory
  - Due to mechanics of the Instruction Buffer Queue
- Thus, timing for an *unchanged* function can change due to change in location of the code

- Use --align_functions to force code for all functions to be aligned on longword (32-bit) boundary
  - Takes extra space, gains consistency in profiling

- OLD: alignment done unless optimizing for space (-ms)
- NEW: separate from space/speed option setting