Keystone Architecture
Code Optimization

Outline

- Software Architecture Considerations
- Development Flow
- Build Options
- Reducing Loop Overhead
- The *restrict* Keyword
- Optimizing Structure References
- *MUST_ITERATE* and *_nassert* pragmas
- Optimizing *if* Statements
- Benchmarking

Outline

- Software Architecture Considerations
- Development Flow
- Build Options
- Reducing Loop Overhead
- The *restrict* Keyword
- Optimizing Structure References
- *MUST_ITERATE* and *_nassert* pragmas
- Optimizing *if* Statements
- Benchmarking

Software Architecture Considerations

- Follow appropriate Multicore design guidelines
- Use Peripherals to offload CPU Tasks
  - EDMA
  - Multicore Navigator
- Cache Behavior
  - Avoid Conflict Misses by ensuring that parent/child functions don't share cache lines
  - Avoid Capacity Misses by ensuring that the cache is large enough
  - Ensure that parent/child functions don't share cache lines (Conflict Miss)
  - Ensure that Cache is large enough (Capacity Miss)
- Some Assembly Required? Use Linear Assembly!
- DON'T USE PRINTF

Outline

- Software Architecture Considerations
- Development Flow
- Build Options
- Reducing Loop Overhead
- The *restrict* Keyword
- Optimizing Structure References
- *MUST_ITERATE* and *_nassert* pragmas
- Optimizing *if* Statements
- Benchmarking

Development Flow

1. Always compile with: –s, –mw
   - Adds extra information to the resulting assembly file
     - -s: show source code after high level optimization
     - -mw: provide extra information on software pipelined loops
     - Safe for production code – **No performance impact**
2. Select the “best” build options
   - More than just “turn on –o3”!
3. Make sure the trip counters are signed integers
4. Provide as much information as possible to the compiler
   - Restrict keywords, MUST_ITERATE pragmas, nasserts
5. DO NOT use –g
6. Analyze the information in the generated assembly file. Identify bottlenecks.

Outline

- Software Architecture Considerations
- Development Flow
- Build Options
- Reducing Loop Overhead
- The *restrict* Keyword
- Optimizing Structure References
- *MUST_ITERATE* and *_nassert* pragmas
- Optimizing *if* Statements
- Benchmarking

Choosing the “Right” build options

- -mv6600 enables 6600 ISA
  - Enables 64+ instruction selection
- –o[2|3]. Optimization level. Critical!
  - -o2/-o3 enables SPLOOP (c66 hardware loop buffer). –o3, file-level optimization is performed. –o2, function-level optimization is performed. –o1, high-level optimization is minimal
- –ms[0-3]. If codesize is a concern…
  - Use in conjunction with –o2 or –o3. Try –ms0 or –ms1 with performance critical code. Consider –ms2 or –ms3 for seldom executed code
  - Note that improved codesize may mean better cache performance
- –mi[N]
  - -mi100 tells the compiler it cannot generate code that turns interrupts off for more than (approximately) 100 cycles.
  - For loops that do *not* SPLOOP, choose ‘balanced’ N (i.e. large enough to get best performance, small enough to keep system latency low)

The -mt Compiler Option

- –mt. Assume no pointer-based parameter writes to a memory location that is read by any other pointer-based parameter to the same function.
  - **generally safe except for *in place* transforms**
  - **E.g. consider the following function:**

```
selective_copy(int *input, int *output, int n)
{
   int i;
   for (i=0; i<n; i++)
      if (myglobal[i]) output[i] = input[i];
}
```

- –mt is safe when memory ranges pointed to by “input” and “output” don’t overlap.
- *limitations of –mt:* applies *only* to pointer-based function parameters. It says nothing about:
  - **relationship between parameters and other pointers (for example, “myglobal” and “output”).**
  - **non-parameter pointers used in the function.**
  - **pointers that are members of structures, even when the structures are parameters.**
  - **pointers dereferenced via multiple levels of indirection.**
- NOTE: -mt is **not** a substitute for restrict-qualifiers which are key to achieving good performance

The –mh Compiler Option

–mh<num>. Speculative loads. Permit compiler to fetch (but not store) array elements beyond either end of an array by <num> bytes. Can lead to:

- better performance, especially for “while” loops.
- smaller code size for both “while” loops and “for” loops.

Software-pipelined loop information in the compiler-generated assembly file suggests the value of <num>

```
;* Minimum required memory pad : 0 bytes
;*
;* For further improvement on this loop, try option -mh56
```

Indicates compiler is fetching 0 bytes beyond the end of an array.

- If loop is rebuilt with –mh56 (or greater), there might be better performance and/or smaller code size.
- NOTE : need to pad buffer of <num> bytes on both ends of sections that contain array data

```
MEMORY {
   /* pad (reserved): origin = 1000, length = 56 */
   myregion: origin = 1056, length = 3888
   /* pad (reserved): origin = 3944, length = 56 */
}
```

Alternatively, can use other memory areas (code or independent data) as pad regions

Build options to avoid

- –g. full symbolic debug. Great for debugging. Do not use in production code.
  - inhibits code reordering across source line boundaries
  - limits optimizations around function boundaries.
  - Can cause a 30-50% performance degradation for control code
  - basic function-level profiling support now provided by default.

- –ss. Interlist source code into assembly file.
  - As with –g, this option can negatively impact performance.

Outline

- Software Architecture Considerations
- Development Flow
- Build Options
- Reducing Loop Overhead
- The *restrict* Keyword
- Optimizing Structure References
- *MUST_ITERATE* and *_nassert* pragmas
- Optimizing *if* Statements
- Benchmarking

Reducing Loop Overhead

- If the compiler does not know that a loop will execute at least once, it will need to:
  1. insert code to check if the trip count is <= zero
  2. conditionally branch around the loop.
- This adds overhead to loops.
- If loop is guaranteed to execute at least once, insert pragma immediately before loop to tell the compiler this:

#pragma MUST_ITERATE(1,,);

or, more generally,
#pragma MUST_ITERATE(min, max, mult);

```
myfunc:

        compute trip count
        if (trip count <= 0)
                branch to postloop

        for (…)
        {
                load input
                compute
                store output
        }

postloop:
```

If trip count not known to be less than zero, compiler inserts code In yellow.

Detecting Loop Overhead

myfunc.c:

```
myfunc(int *input1, int *input2, int *output,
   int n)
{
    int i;
    for (i=0; i<n; i++)
       output[i] = input1[i] - input2[i];
}
```

Extracted from myfunc.asm (generated using –o –mv6600 –s –mw):

```
;** 4   -----------------------    if ( n <= 0 ) goto g4;
;**      -----------------------    U$11 = input1;
;**      -----------------------    U$13 = input2;
;**      -----------------------    U$16 = output;
;**      -----------------------    L$1 = n;
;**      -----------------------    #pragma MUST_ITERATE(1,…)
;**      -----------------------g3:
;** 5   -----------------------    *U$16++ = *U$11++-*U$13++;
;** 4   -----------------------    if ( --L$1 ) goto g3;
;**      -----------------------g4:
```

Outline

- Software Architecture Considerations
- Development Flow
- Build Options
- Reducing Loop Overhead
- The *restrict* Keyword
- Optimizing Structure References
- *MUST_ITERATE* and *_nassert* pragmas
- Optimizing *if* Statements
- Benchmarking

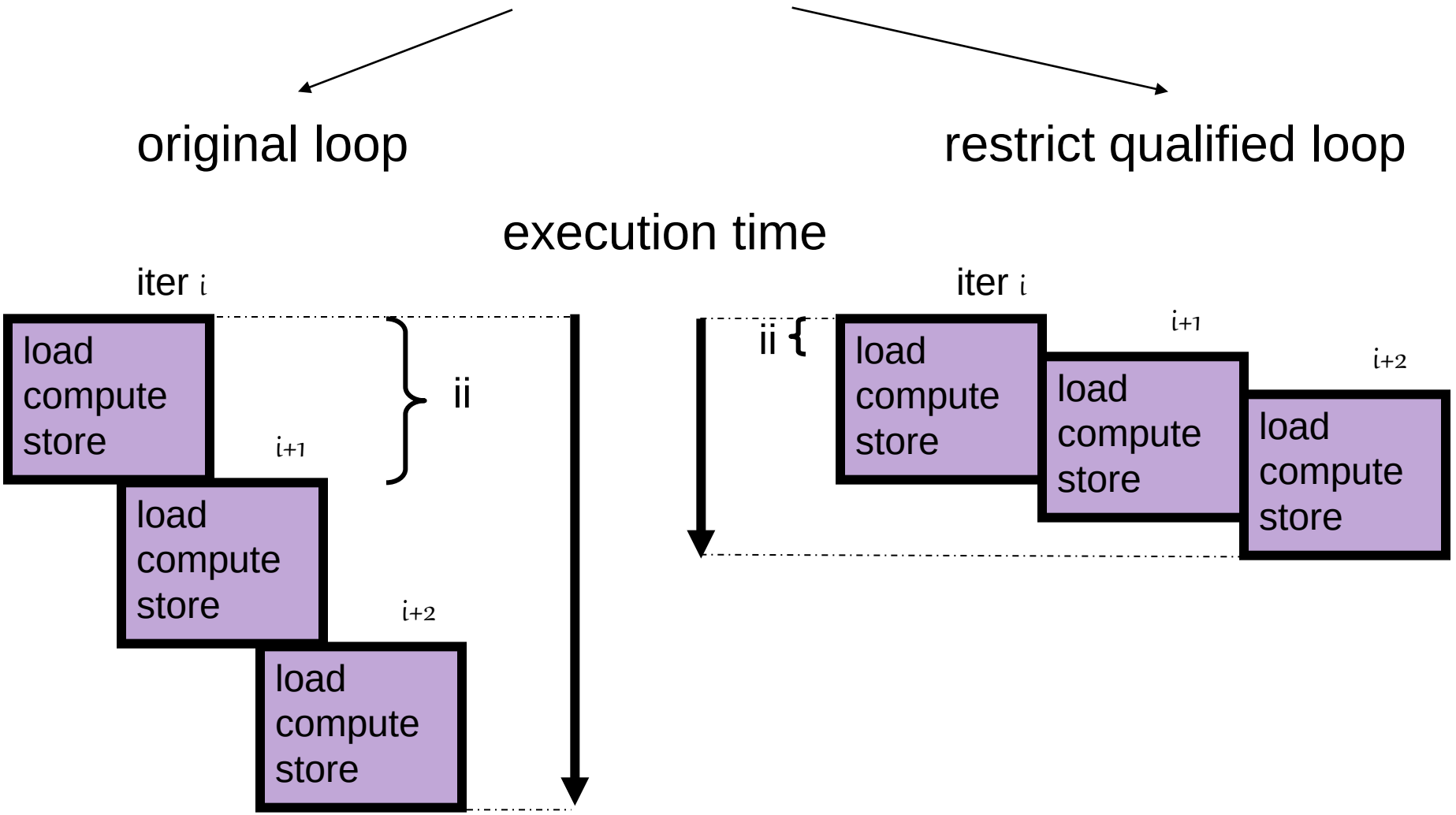
Restrict Qualifiers

```
myfunc(type1 input[ ],
            type2 *output)
{
    for (…)
    {
                load from
    input
        compute
        store to output
    }
}
```

- C6000 depends on overlapping loop iterations for good (software pipelining) performance.
- Loop iterations cannot be overlapped unless input and output are *independent* (do not reference the same memory locations).
- Most users write their loops so that loads and stores do not overlap.
- Compiler does not know this unless the compiler sees all callers or user tells compiler.
- Use restrict qualifiers to tell compiler:

```
myfunc(type1 input[restrict],
                  type2 *restrict output)
```

Restrict Qualifiers (cont.) myfunc



Outline

- Software Architecture Considerations
- Development Flow
- Build Options
- Reducing Loop Overhead
- The *restrict* Keyword
- Optimizing Structure References
- *MUST_ITERATE* and *_nassert* pragmas
- Optimizing *if* Statements
- Benchmarking

Restrict Qualifying Pointers in Structures

- At present, pointers that are structure elements *cannot* be *directly* restrict-qualified neither with –mt nor by using the restrict keyword.
  - Fixed in CGT 6.1.0
- Instead, create local pointers *at top-level of function* and restrict qualify pointers instead.
- Use local pointers in function instead of original pointers.

```
myfunc(_str *s)
{
        _str *t
    // declare local pointers at
    // top-level of function
    int * restrict p
    int * restrict v
    …
    // assign to sp and tp
    p = s->q->p
    v = t->u->v
    // use sp and tp instead
    // of s->q->p and t->u->v
    *p = …
    *v = …
       = *p
       = *v
}
```

Writing Efficient Code with Structure References

General Tips:

- Avoid dereferencing structure elements in loop control and loops.
- Instead create/use local copies of pointers and variables when possible.
- Non-restrict-qualified locals do not need to be declared at top-level of function.

Original Loop:

```
while (g->q->y < 25)
{
    g->p->a[i++] = …
}
```

Hand-optimized Loop:

```
int    y  = g->q->y;
short *a  = g->p->a;

while (y < 25)
{
    a[i++] = …
}
```

Example: Restrict and Structures

```
myfunc(_str *restrict s)
{
   int i;
   #pragma MUST_ITERATE(2,,2);
   for (i=0; i<s->data->sz; i++)
       s->data->q[i] = s->data->p[i];
}
```

restrict does not help! Only applies to s, not to s→ data→ p or s→ data→ q

-mt does not help! Only applies to s, not to s→data→p or s→data→q

cl6x –o –mw –s –mt –mv6600

Extracted from .asm file:

```
;** - g2:
;** -   *(i*4+(*V$0).q) = *(i*4+(*V$0).p);
;** -   if ( (*V$0).sz > (++i) ) goto g2;
…
;*----------------------------------------
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 17
;*      Loop opening brace source line   : 18
;*      Loop closing brace source line   : 18
;*      Known Minimum Trip Count         : 2
;*      Known Max Trip Count Factor      : 2
;*      Loop Carried Dependency Bound(^) : 11
…
;*      ii = 12 Schedule found with 2 iterat…
```

Note: Addresses of p, q, and sz are calculated during **every** loop iteration.

Bottom line: 12 cycles/result, 72 bytes

Example: Restrict and Structures (cont.)

```
myfunc(_str *s)
{
   int *restrict p, *restrict q;
   int sz;
   int i;
   ...
   p  = s->data->p;
   q  = s->data->q;
   sz = s->data->sz;

   #pragma MUST_ITERATE(2,,2);
   for (i=0; i < sz; i++)
       q[i] = p[i];
}
```

Hand-optimized source file

cl6x –o –s –mw –mv6600

Extracted from .asm file:

```
;** -   // LOOP BELOW UNROLLED BY FACTOR(2)
…
;** - g2:
;** -   _memd8((void *)U$17) =
                                  _memd8((void *)U$14);
;** -   U$14 += 2;
;** -   U$17 += 2;
;** -   if ( L$1 = L$1-1) ) goto g2;

;----------------------------------------
; SOFTWARE PIPELINE INFORMATION


;*      Loop Unroll Multiple                  : 2x
;*      Known Minimum Trip Count              : 1
;*      Known Max Trip Count Factor           : 1
;*      Loop Carried Dependency Bound(^)      : 0
…
;*      ii = 2 Schedule found with 3 iterati…
```

Observe: Now the compiler automatically unrolls loop and SIMDs memory accesses.

Bottom line:
1 cycle/result, 44 bytes

Outline

- Software Architecture Considerations
- Development Flow
- Build Options
- Reducing Loop Overhead
- The *restrict* Keyword
- Optimizing Structure References
- *MUST_ITERATE* and *_nassert* pragmas
- Optimizing *if* Statements
- Benchmarking

Example: MUST_ITERATE, nassert and SIMD

```
myfunc(int * restrict input1,
       int * restrict input2,
       int * restrict output,
       int n)
{
 int i;

 #pragma MUST_ITERATE(1,,);
 for (i=0; i < n; i++)
   output[i] =
     input1[i] – input2[i];
}
```

cl6x –o –s –mw –mv6600

-mw comments (from .asm file):

```
;----------------------------------------
;* SOFTWARE PIPELINE INFORMATION
;*
;*    Known Max Trip Count Factor      : 1
;*    Loop Carried Dependency Bound(^) : 0
;*    Unpartitioned Resource Bound     : 2
;*    Partitioned Resource Bound(*)    : 2
;*    Resource Partition:
;*                                  A-side   B-side
;*    .D units                        2*       1
;*    .T address paths                2*       1
;*
;*      ii = 2  Schedule found with 4 iter...
;*
;*      SINGLE SCHEDULED ITERATION
;*
;*      $C$C24:
;*   0              LDW     .D1T1   *A5++,A4
;*   1              LDW     .D2T2   *B4++,B5
;*   2              NOP             4
;*   6              SUB     .L1X    B5,A4,A3
;*   7              STW     .D1T1   A3,*A6++
;*     ||           SPBR            $C$C24
;*   8              ; BRANCHCC OCCURS {$C$C24}
```

2 cycles / result

resources unbalanced

-s comments (from .asm file):

```
;** -   U$12 = input1;
;** -   U$14 = input2;
;** -   U$17 = output;
;** -   L$1  = n;
…
;** - g2:
;** -   *U$17++ = *U$12++ - *U$14++;
;** -   if ( --L$1 ) goto g2;
```

Example: MUST_ITERATE, nassert and SIMD (cont)

Suppose we know that the trip count is a multiple of 4…

```
myfunc(int * restrict input1,
       int * restrict input2,
       int * restrict output,
       int n)
{
   int i;
   #pragma MUST_ITERATE(1,,4);
   for (i=0; i < n; i++)
     output[i] = input1[i] – input2[i];
}
```

Example: MUST_ITERATE, nassert and SIMD (cont)

cl6x –o –s –mw –mv6600

-s comments (from .asm file):

```
;**   // LOOP BELOW UNROLLED BY FACTOR(2)
;**   U$12 = input1;
;**   U$14 = input2;
;**   U$23 = output;
;**   L$1  = n >> 1;
…
;** g2:
;**   _memd8((void *)U$23) =
      _itod(*U$12[1]-*U$14[1],*U$12-*U$14);
;**   U$12 += 2;
;**   U$14 += 2;
;**   U$23 += 2;
;**    if ( --L$1 ) goto g2;
```

-mw comments (from .asm file):

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*   Loop Unroll Multiple                  : 2x
;*   Loop Carried Dependency Bound(^)      : 0
;*   Unpartitioned Resource Bound          : 3
;*   Partitioned Resource Bound(*)         : 3
;*   Resource Partition:
;*                                A-side   B-side
;*   .D units                      3*       2
;*   .T address paths              3*       3*
;*
;*   ii = 3 Schedule found with 3 iter...
;*
;*   SINGLE SCHEDULED ITERATION
;*   $C$C24:
;*        0        LDW     .D1T1   *A6++(8),A3
;*          ||     LDW     .D2T2   *B6++(8),B4
;*        1        LDW     .D1T1   *A8++(8),A3
;*          ||     LDW     .D2T2   *B5++(8),B4
;*        2        NOP             3
;*        5        SUB     .L1X    B4,A3,A4
;*        6        NOP             1
;*        7        SUB     .L1X    B4,A3,A5
;*        8        STNDW   .D1T1   A5:A4,*A7++(8)
```

1.5 cycles / result (resource balance better but not great)

Example: MUST_ITERATE, _nassert, SIMD (cont)

Suppose we tell the compiler that input1, input2 and output are aligned on double-word boundaries…

**** Note – must*** **`_nassert(x)`** ***before*** **`x`** ***is used***

```
myfunc(int * restrict input1,
       int * restrict input2,
       int * restrict output,
       int n)
{
   int i;
   _nassert((int) input1 % 8 == 0);
   _nassert((int) input2 % 8 == 0);
   _nassert((int) output % 8 == 0);
   #pragma MUST_ITERATE(1,,4);
   for (i=0; i < n; i++)
     output[i] = input1[i] – input2[i];
}
```

Example: MUST_ITERATE, nassert and SIMD (cont)

cl6x –o –s –mw –mv64+

-s comments (from .asm file):

```
;**  // LOOP BELOW UNROLLED BY FACTOR(4)
;**  U$12 = (double * restrict)input1;
;**  U$16 = (double * restrict)input2;
;**  U$27 = (double * restrict)output;
;**  L$1  = n >> 2;
…
;** g2:
;**   C$5   = *U$16;
;**   C$4   = *U$12;
;**   *U$27 = _itod((int)_hi(C$4)-
                   (int)_hi(C$5),
                   (int)_lo(C$4)-
                   (int)_lo(C$5));
;**   C$3   = *U$16[1];
;**   C$2   = *U$12[1];
;**   *U$27 = _itod((int)_hi(C$2)-
                   (int)_hi(C$3),
                   (int)_lo(C$2)-
                   (int)_lo(C$3));
;**   U$12 += 2;
;**   U$16 += 2;
;**   U$27 += 2;
;**    if ( --L$1) ) goto g2;
```

0.75 cycles / result (resources balanced)

-mw comments (from .asm file):

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop Unroll Multiple                  : 4x
;*      Loop Carried Dependency Bound(^)      : 0
;*      Unpartitioned Resource Bound          : 3
;*      Partitioned Resource Bound(*)         : 3
;*      Resource Partition:
;*                                A-side   B-side
;*      .D units                    3*       3*
;*      .T address paths            3*       3*
;*
;*      ii = 3 Schedule found with 3 iter...
;*
;*      SINGLE SCHEDULED ITERATION
;*      $C$C24:
;*   0     LDDW    .D2T2   *B18++(16,B9:B8
;*       ||LDDW    .D1T1   *A9++(16),A7:A6
;*   1     LDDW    .D1T1   *A3++(16),A5:A4
;*       ||LDDW    .D2T2   *B5++(16),B17:B16
;*   2     NOP             3
;*   5     SUB     .L2X    A7,B9,B7
;*   6     SUB     .L2X    A6,B8,B6
;*       ||SUB     .L1X    B16,A4,A4
;*   7     SUB     .L1X    B17,A5,A5
;*   8     STDW    .D2T2   B7:B6,*B4++(16)
;*       ||STDW    .D1T1   A5:A4,*A8++(16)
```

Outline

- Software Architecture Considerations
- Development Flow
- Build Options
- Reducing Loop Overhead
- The *restrict* Keyword
- Optimizing Structure References
- *MUST_ITERATE* and *_nassert* pragmas
- Optimizing *if* Statements
- Benchmarking

If Statements

- Compiler will if-convert short if statements:

Original C code:

```
if (p) then x = 5 else x = 7
```

Before if conversion:

```
        [p] branch thenlabel
            x = 7
            goto postif
thenlabel:  x = 5
postif:
```

After if conversion:

```
        [p] x = 5 || [!p] x = 7
```

If Statements (cont.)

- Compiler will not if convert long if statements.
- Compiler will not software pipeline loops with if statements that are not if-converted.

```
;*----------------------------------------------------
;*   SOFTWARE PIPELINE INFORMATION
;*      Disqualified loop: Loop contains control code
;*----------------------------------------------------
```

- For software “pipelinability”, user must transform long if statements because compiler does not know if this is profitable.

Example of If Statement Reduction When No Else Block Exists

Original function:

```
largeif1(int *x, int *y)
{
   for (…)
   {
       if (*x++)
       {
           i1
           i2
           …
           *y = …
       }
       y++
   }
}
```

Hand-optimized function:

```
largeif1(int *x, int *y)
{
   for (…)
   {
       i1
       i2
       …
       if (*x++)
           *y = …
       y++
   }
}
```

pulled out of if stmt

Note: Only assignment to y must be guarded for correctness.
Profitability of if reduction depends on sparsity of x.

Or If Statement Can Be Eliminated Entirely

Original function:

```
large_if1(int *x, int *y)
{
   for (…)
   {
        if (*x++)
        {
            i1
            i2
            …
            *y += …
        }
        y++
   }
}
```

Hand-optimized function:

```
large_if1(int *x, int *y)
{
   for (…)
   {
        i1
        i2
        …
        p   = (*x++ != 0)
        *y += p * (…)
        y++
   }
}
```

Sometimes this works better….

If Reduction Via Common Code Consolidation

Original function:

```
large_if2(int *x, int *y, int *z)
{
    for (…)
    {
          if (*x++)
          {
              int t = *z++
              *w++  = t
              *y++  = t
          }
          else
          {
              int t = *z++
              *y++  = t
          }
    }
}
```

Hand-optimized function:

```
large_if2(int *x, int *y, int *z)
{
    for (…)
    {
          int t = *z++
          if (*x++)
          {
              *w++ = t
          }
          *y++  = t
    }
}
```

Note: Makes loop body smaller. Eliminates 2nd copy of:

t = *z++

***y++ = t**

Eliminating Nested If Statements

- *Compiler will software pipeline nested if statements less efficiently, if at all.*

Original function:

```
complex_if(int *x, int *y,
           int *z)
{
    for (…)
    {
        // nested if stmt
         if (*z++)
             i1
         else
             if (*x)
                 *y = c
         y++
         x++
    }
}
```

Hand-optimized function:

```
complex_if(int *x, int *y,
           int *z)
{
    for (…)
    {
        // nested if stmt removed
         if (*z++)
             i1
         else
         {
             p = (*x != 0)
            *y = !p * *y + p * c
         }
         y++
         x++
    }
}
```

Outline

- Software Architecture Considerations
- Development Flow
- Build Options
- Reducing Loop Overhead
- The *restrict* Keyword
- Optimizing Structure References
- *MUST_ITERATE* and *_nassert* pragmas
- Optimizing *if* Statements
- Benchmarking

Benchmarking

• C66x corepac has a 64-bit timer (Time Stamp Counter) incremented at the CPU speed.
• Simplest benchmarking approach is to use lower 32 bits (TSC**L**)
  • sufficient for most benchmarking needs

```
#include <c6x.h>              // bring in references to TSCL, TSCH

void main() {
   ...
   TSCL = 0;      // Initiate CPU timer by writing any val to TSCL
   ...
   t1 = TSCL;     // benchmark snapshot of free-running ctr
   my_code_to_benchmark();
   t2 = TSCL;     // benchmark snapshot of free-running ctr

   printf("# cycles == %d\n", (t2-t1));
}
```

Advantages

• no need to worry about interrupts (as opposed to when reading *both* TSCL & TSCH)
• no assembly code
• no need for Chip Support Library (CSL) or other APIs
• fast

Benchmarking (2)

- If you need more than 32 bits for benchmarking (rare) …

```
#include <c6x.h>              // bring in references to TSCL, TSCH
#include <stdint.h>           // get C99 data types such as uint64_t

uint64_t t1, t2;

t1 = _itoll(TSCH, TSCL);      // get full 64-bit time snapshot
my_code_to_benchmark();
t2  = _itoll(TSCH, TSCL);     // get full 64-bit time snapshot

printf("# cycles == %lld\n", (t2-t1));
```

- Beware!
  - Not protected from interrupts between reading of TSCL and TSCH!
  - Fix by adding **`_disable_interrupts(), _restore_interrupts()`** intrinsics

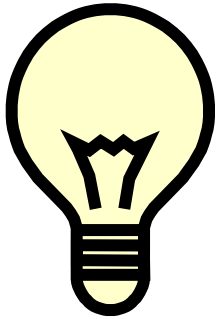
- Similar code exists in many CSL implementations
  - it *does* provide interrupt protection (via assembly code branch delay slots)

Misc C66x User Advice

- Do not let loops get too large
  - SPLOOP limits:
    - single iteration (dynamic length) must be ≤ 48 cycles
    - # of cycles in loop body (ii) must be ≤ 14
  - Beware of unroll pragmas with respect to SPLOOP limits
- Leverage New C66x Intrinsics (Examples Below)
  - **_dadd2 -** Four-way SIMD addition of signed 16-bit values producing four signed 32-bit results.
  - **_ddotp4h** - Performs two dot-products between four sets of packed 16-bit values.
  - **_qmpy32** - Four-way SIMD multiply of signed 32-bit values producing four 32-bit results.

Summary : Tips for Developing Efficient Code

- Understand/exploit .asm file comments generated when compiling with –s and –mw.
- Get your CGT build options right
- Use restrict qualifiers, MUST_ITERATE pragmas and _nasserts.
  --- Remember, -mt does not cover pointers embedded in structures.
- Pull structure references out of loops and especially loop control.
- Reduce complexity/length of if statements.
- Don’t let loops get too large



References

- spra666, "*Hand-Tuning Loops and Control Code on the TMS320C6000*" [link]
- spraa46, "*Advanced Linker Techniques for Convenient and Efficient Memory Usage*" [link]
- spru187, "*TMS320C6000 Optimizing Compiler User's Guide*" [link]