C66 Efficient Code

C66 efficient code

Agenda:

- **Understanding Compiler Feedback**
- Overview of few C66x Instructions
- Restrict keyword
- Control loop/code optimizations
- Variables and Types

Software Pipelining feedback (.asm file)

```
;*----------------------------------------------------------

;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count         : 2
;*      Known Maximum Trip Count         : 2
;*      Known Max Trip Count Factor      : 2
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound     : 4
;*      Partitioned Resource Bound(*)    : 5
;*      Resource Partition:
;*        A-side   B-side
;*      .L units        2        3
;*      .S units        4        4
;*      .D units        1        0
;*      .M units        0        0
;*      .X cross paths  1        3
;*      .T address paths 1       0
;*      Long read paths  0       0
;*      Long write paths 0       0
;*      Logical  ops (.LS)          0        1      (.L or .
;*      Addition ops (.LSD)         6        3      (.L or .
;*      Bound(.L .S .LS) 3       4
;*      Bound(.L .S .D .LS .LSD)    5*       4
;*

;*      Searching for software pipeline schedule at ...
;*         ii = 5  Register is live too long
;*         ii = 6  Did not find schedule
;*         ii = 7  Schedule found with 3 iterations in paral
;*      done
```

- **Loop unroll factor.** The number of times the loop was unrolled specifically to increase performance based on the resource bound constraint in a software pipelined loop.
- **Known minimum trip count.** The minimum number of times the loop will be executed.
- **Known maximum trip count.** The maximum number of times the loop will be executed.
- **Known max trip count factor.** Factor that would always evenly divide the loops trip count. This information can be used to possibly unroll the loop.
- **Loop label.** The label you specified for the loop in the linear assembly input file. This field is not present for C/C++ code.
- **Loop carried dependency bound.** The distance of the largest loop carry path. A loop carry path occurs when one iteration of a loop writes a value that must be read in a future iteration. Instructions that are part of the loop carry bound are marked with the ^ symbol.
- **Iteration interval** (ii). The number of cycles between the initiation of successive iterations of the loop. The smaller the iteration interval, the fewer cycles it takes to execute a loop.
- **Resource bound.** The most used resource constrains the minimum iteration interval. For example, if four instructions require a .D unit, they require at least two cycles to execute (4 instructions/2 parallel .D units).

TEXAS INSTRUMENTS

Software Pipelining feedback (.asm file)

- **Unpartitioned resource bound.** The best possible resource bound values before the instructions in the loop are partitioned to a particular side.
- **Partitioned resource bound (*).** The resource bound values after the instructions are partitioned.
- **Resource partition.** This table summarizes how the instructions have been partitioned. This information can be used to help assign functional units when writing linear assembly. Each table entry has values for the A-side and B-side registers. An asterisk is used to mark those entries that determine the resource bound value. The table entries represent the following terms:
  - **.L units** is the total number of instructions that require .L units.
  - **.S units** is the total number of instructions that require .S units.
  - **.D units** is the total number of instructions that require .D units.
  - **.M units** is the total number of instructions that require .M units.
  - **.X cross paths** is the total number of .X cross paths.
  - **.T address paths** is the total number of address paths.
  - **Long read path** is the total number of long read port paths.
  - **Long write path** is the total number of long write port paths.
  - **Logical ops (.LS)** is the total number of instructions that can use either the .L or .S unit.
  - **Addition ops (.LSD)** is the total number of instructions that can use either the .L or .S or .D unit
- **Bound(.L .S .LS)** is the resource bound value as determined by the number of instructions that use the .L and .S units. It is calculated with the following formula:

  Bound(.L .S .LS) = ceil((.L + .S + .LS) / 2)
- **Bound(.L .S .D .LS .LSD)** is the resource bound value as determined by the number of instructions that use the .D, .L and .S unit. It is calculated with the following formula:

  Bound(.L .S .D .LS .LSD) = ceil((.L + .S + .D + .LS + .LSD) / 3)
- **Minimum required memory pad.** The number of bytes that are read if speculative execution is enabled. See section 3.2.3, *Collapsing Prologs and Epilogs for Improved Performance and Code Size*, on page 3-13, for more information.

C66 efficient code

Agenda:

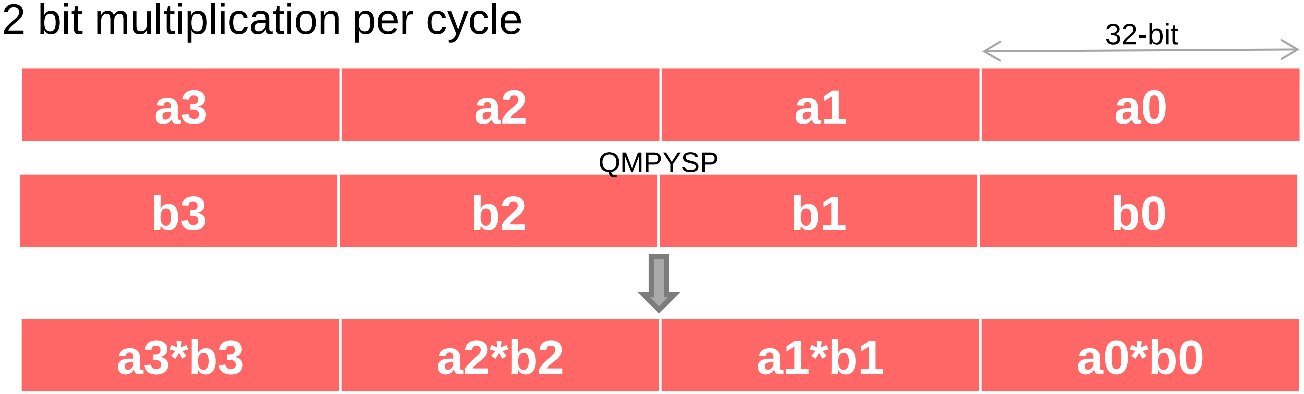
- Understanding Compiler Feedback
- **Overview of few C66x Instructions**
- Restrict keyword
- Control loop/code optimizations
- Variables and Types

Load - Store Instructions

- LDDW
  - Will load aligned 64 bit data from memory to register pair.
  - Occupies one .D, and one .T unit.
- STDW
  - Will store aligned 64 bit data from memory to register pair.
  - Occupies one .D, one .T unit.
- LDNDW
  - Will load non-aligned 64 bit data from memory to register pair.
  - Occupies one .D, and two .T unit.
- STNDW
  - Will store non-aligned 64 bit data from memory to register pair.
  - Occupies one .D, and two .T unit.
- Address increment/offset, if fits within 5 bits then it can happen in free. Otherwise extra instructions are needed to increment/decrement the source or destination pointer.

Multiply Instructions

- 8-Bit operands
  - DMPYU4 (.M)
    - 16 8-bit multiplication per cycle
- 16-Bit operands
  - DMPY2 (.M)
    - 8 16-bit multiplication per cycle. For complex matrix multiply it is 32 multiplication per cycle
- 32-Bit operands
  - QMPY, QMPYSP (.M)
    - 8 32 bit multiplication per cycle



Addition/Subtraction Instructions

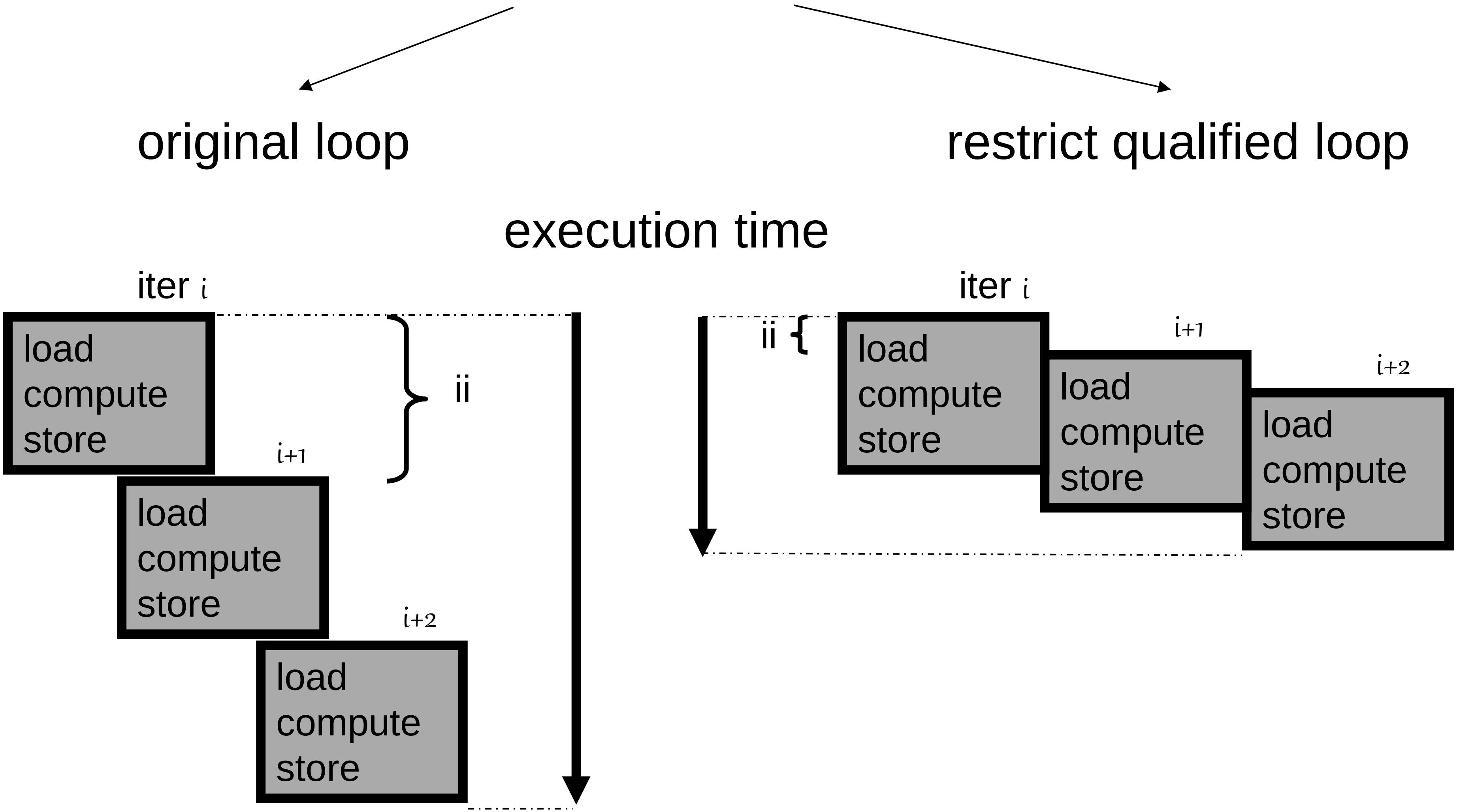
- 8-bit operands
  - ADD4, SUB4 (**.L**)
    - 8 8-bit addition/subtraction per cycle
- 16-bit operands
  - DADD2, DSUB2 (**.L,.S**)
  - ADD2, SUB2 (.L,.S,**.D**)
    - 20 16-bit addition/subtraction per cycle
- 32-bit operands
  - DADD, DSUB, DADDSP, DSUBSP (**.L,.S**)
  - ADD, SUB (.L,.S,**.D**)
    - 10 32-bit addition/subtraction per cycle

C66 efficient code

Agenda:

- Understanding Compiler Feedback
- Overview of few C66x Instructions
- **Restrict keyword**
- Control loop/code optimizations
- Variables and Types

Restrict Qualifiers (cont.)



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 caller or user tells compiler.
- Use restrict qualifiers to tell compiler:

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

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.

- Instead, create local pointers and restrict qualify pointers instead.

- Use local pointers in function/loop instead of original pointers.

- Local pointers can be declared within any local scope, not just the top-level of the function

```
typedef struct {int *p} _str;

myfunc(_str *s)
{
   _str *t;

      // create local pointers
      int * restrict sp = s->p;
      int * restrict tp = t->p;
      …
      // use sp and tp instead
      // of s->p and t->p
      *tp = …
      *sp = …
          = *sp
          = *tp
}
```

Writing Efficient Code
Structure References

General Tips:

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

Original Loop:

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

Hand-optimized Loop:

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

while (y < 25)
{
     a[i++] = …
}
```

Example

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

```
void BasicLoop(int *restrict output,
               int *restrict input1,
               int *restrict input2,
               int n)
{
    int i;
    _nassert((int) input1 % 8 == 0); // input1 is 8-byte aligned
    _nassert((int) input2 % 8 == 0); // input2 is 8-byte aligned
    _nassert((int) output % 8 == 0); // output is 8-byte aligned
    #pragma MUST_ITERATE(4,,4) // n >= 4, n % 4 = 0
    for (i=0; i<n; i++) {
      output[i]   = input1[i] + input2[i];
    }
}
```

Example Contd..

```
;*       Loop opening brace source line    : 13
;*       Loop closing brace source line    : 15
;*       Loop Unroll Multiple              : 4x
;*       Known Minimum Trip Count          : 1
;*       Known Max Trip Count Factor       : 1
;*       Loop Carried Dependency Bound(^)  : 0
;*       Unpartitioned Resource Bound      : 3
;*       Partitioned Resource Bound(*)     : 3
;*       Resource Partition:
;*                                 A-side   B-side
;*       .L units                     0        0
;*       .S units                     1        0
;*       .D units                     3*       3*
;*       .M units                     0        0
;*       .X cross paths               2        2
;*       .T address paths             3*       3*
;*       Long read paths              0        0
;*       Long write paths             0        0
;*       Logical  ops (.LS)           0        0     (.L or .S unit)
;*       Addition ops (.LSD)          2        2     (.L or .S or .D unit)
;*       Bound(.L .S .LS)             1        0
;*       Bound(.L .S .D .LS .LSD)     2        2
;*
;*       Searching for software pipeline schedule at ...
;*          ii = 3  Schedule found with 3 iterations in parallel
...
;*----------------------------------------------------------------------------*
;*         SINGLE SCHEDULED ITERATION
;*
;*         C26:
;*     0                LDDW     .D2T2    *B5++(16),B9:B8    ; |14|
;*       ||             LDDW     .D1T1    *A16++(16),A7:A6   ; |14|
;*     1                LDDW     .D2T2    *-B5(8),B7:B6      ; |14|
;*       ||             LDDW     .D1T1    *-A16(8),A9:A8     ; |14|
;*     2                NOP               1
;*     3         [ A0]  BDEC     .S1      C26,A0             ; |13|
;*     4                NOP               1
;*     5                ADD      .S1X     B9,A7,A5           ; |14|
;*     6                ADD      .L1X     B8,A6,A4           ; |14|
;*       ||             ADD      .L2X     B6,A8,B6           ; |14|
;*     7                ADD      .L2X     B7,A9,B7           ; |14|
;*     8                STDW     .D1T1    A5:A4,*A3++(16)    ; |14|
;*       ||             STDW     .D2T2    B7:B6,*++B4(16)    ; |14|
;*     9                ; BRANCHCC OCCURS {C26}              ; |13|
```

C66 efficient code

Agenda:

- Understanding Compiler Feedback
- Overview of few C66x Instructions
- Restrict keyword
- **Control loop/code optimizations**
- Variables and Types

What is Control Code?

- Control Code compilation often results in:
  - Lots of software branches
    - Unfilled delay slots
  - Irregular loops
    - Gets in the way of software pipelining
  - Serial software
    - Little use of the up to 8 instructions/CPU cycle
- Typical symptom:

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

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

- Compiler may not if-convert large 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 may have to transform large if statements because compiler may think it is unprofitable or it may be too complex.

Expressing conditions in loops

- Certain control structures are problematic to compiler
  - static conditionals should sometimes be moved out of loops
  - heavy use of structures and more so unions can prevent software pipelining

```
for (i=0; i < N; i++){
   if (x->z[j] == TRUE && v[k]->w == FALSE && i&0x3)
   {
     y += a_st[i];
   }
}
```

```
cond = (x->z[j] == TRUE &&
          v[k]->w == FALSE);
for (i=0; i < N; i++)
{
   if (cond && i&0x3)
          y += a_st[i];
}
```

Logical vs bitwise operators

- For logical operators (a || b), where a and b are expressions, the expression “a” must be evaluated first and “b” will not be evaluated unless “a” is evaluated to false.
- Bitwise operators (a | b) can avoid the control flow (branches) that is required when using logical operators, and improve control loop efficiency
- Changing from logical to bitwise operators can make some control loops pipeline

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_trip, max_trip, multiple);

```
myfunc:

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

      for (…)
      {
            load input
            compute
            store output
      }
```

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 –os):

```
;** 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;
```

Example1:
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.

Example2:
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

Example4:
Function Calls in Loops

Function calls within a loop prevent software pipelining.
Overhead by function call is about 30 CPU cycles.

Original function:

```
funcs(int *x, int *p, int n){
#pragma MUST_ITERATE(1,200,)
  for (i=0; i<n; i++){
        if (*x++)
        {
          // large block
          do_something(*p);
        }
        p++;
  }
}
void do_something(int v);
```

Hand-optimized function:

```
funcs (int *x, int *p, int n){
   int vect[200];
   int j=0, k=-1, cnt=1;
#pragma MUST_ITERATE(1,200,)
  for (i=0; i<n; i++){
        if (*x++) {
            vect[j++] = *p;
        }
  }
  do_something_vect(vect, j);
}
void do_something_vect(int * vect, int
  size) {
  int i;
  for (i=0; i<size; i++){
        /* Do something */
  }
```

register items for later processing

loop will pipeline

C66 efficient code

Agenda:

- Understanding Compiler Feedback
- Overview of few C66x Instructions
- Restrict keyword
- Control loop/code optimizations
- **Variables and Types**

“Variable type” optimizations

```
Int8  count;
count = count + 1;
   becomes:
.asg _count A20
ADD _count, 1, _count
EXTU _count, 24, 24, _count
```

```
Int32  count;
count = count + 1;
   becomes:
.asg _count A20
ADD _count, 1, _count
```

- The type of variable used will affect performance
  - use 32 bit precision whenever possible for control variables
- Need to have the correct precision for computations
  - don’t declare Int32 when expecting a 16 x 16 bit multiply
  - use casting for intermediate multiplications
  - try to make all accumulators maximum precision of 32 bits
- Compilers give you exactly what you ask for!

The End!