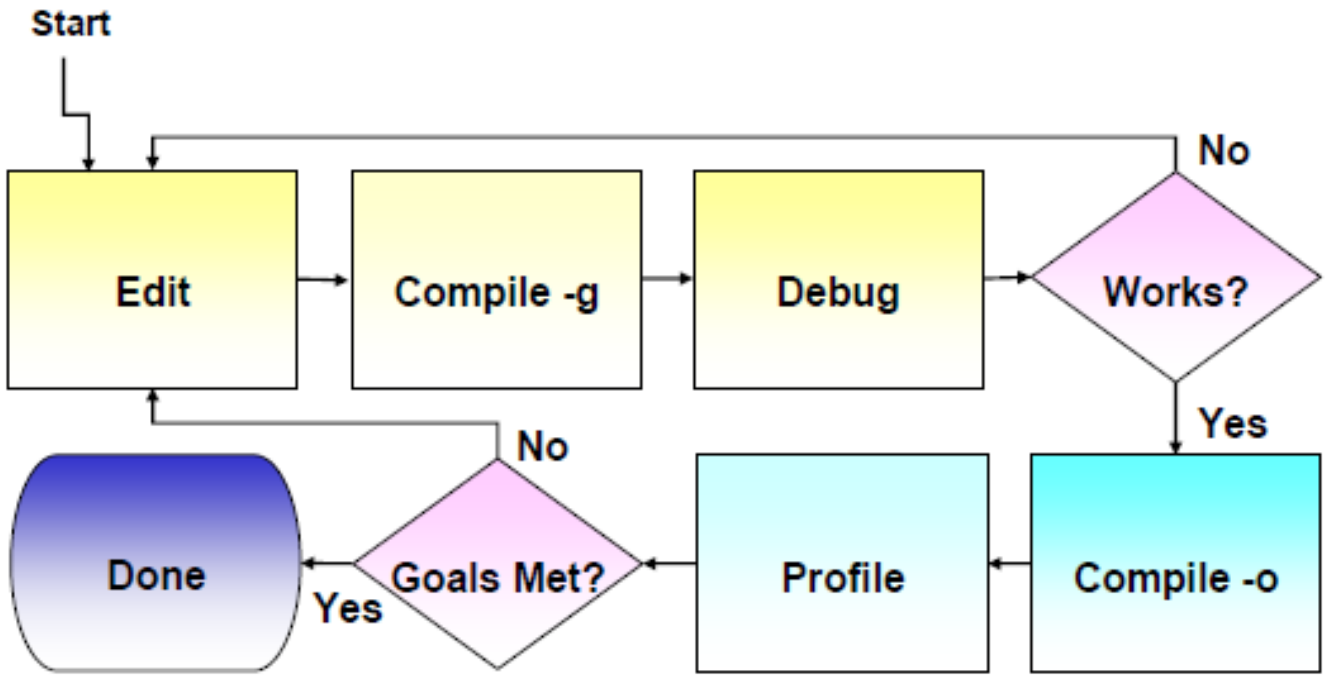
Optimization on TMS320C64x+

T. Grandpierre
(Based on TI Web training material
& TI Developper conference material)

C64x+ C code Optimization

- Optimizing with the C Compiler
- Writing Intrinsic C code
- Loop transformations
- Fundamental Rules of DSP Programming

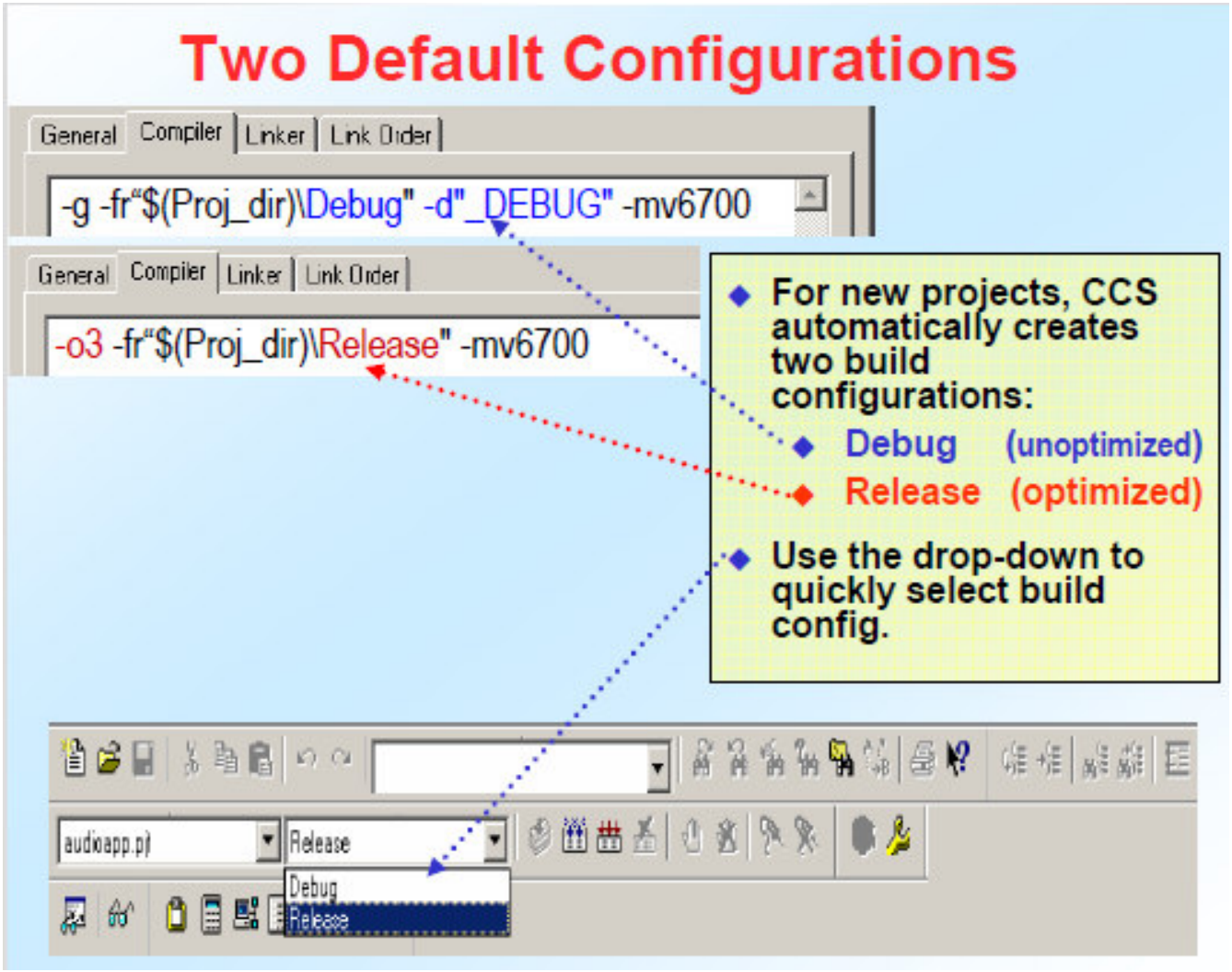
Recommended Development Flow



- Debug first. Tune second.
- After making sure you program is logically correct, We can use compiler optimizer to optimize.

Debug & Optimize options conflict with each other, therefore they should not be used together

Two Default Configurations

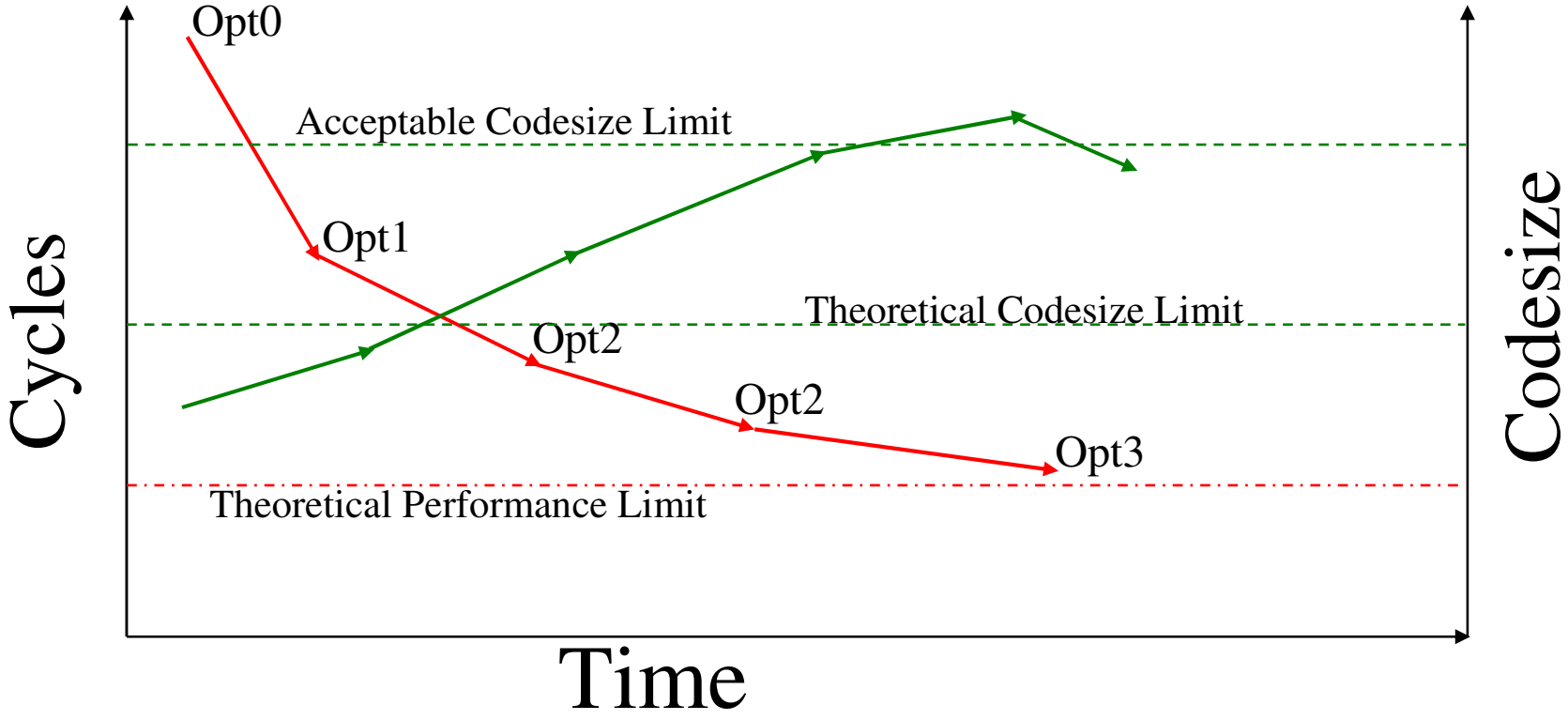


Optimization Goals

- Knowledge of function => optimization
- Before Writing any Code
  - must know what the goal is, performance, codesize, multiplies/cycle, bytes/cycle, etc.
  - Need to be able to estimate performance based
    - Divide Resources on C64x+ into Function Resource Needs
    - C64x+: 8 16x16 Multiplies, 4 32 bit ADDs, 2 64 bit loads
  - Ideally know alignment, loop iteration limits and loop multiples

Optimization Path

- Need to know

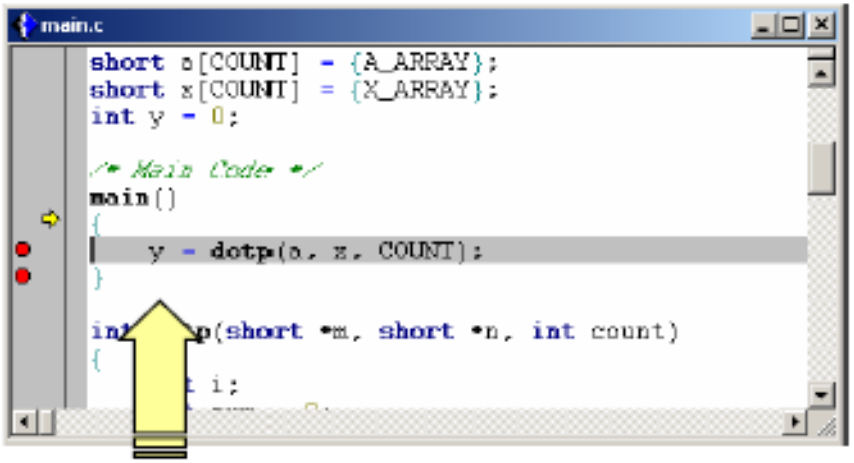


Profiling

Profiling methods

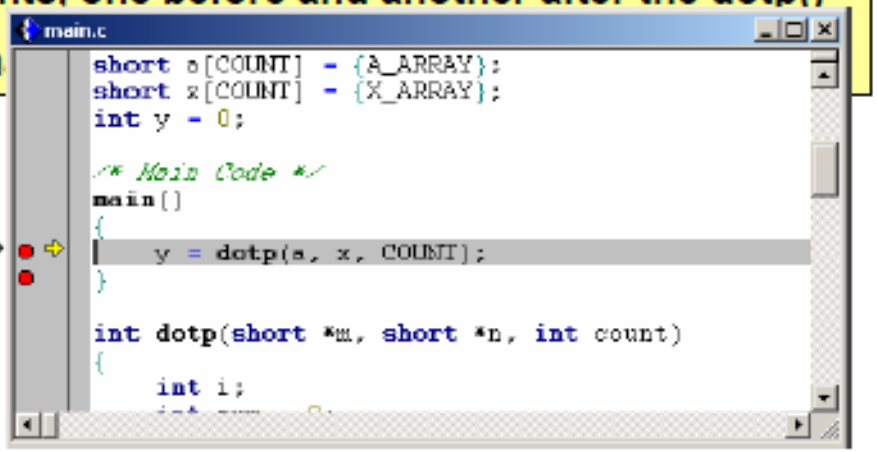
- Code Composer Studio (CCS) profiler:
  - Function level profiling with inclusive/exclusive counts
  - Does not work on multi-threaded applications
- Code instrumentation:
  - The C64x+ CPU contains a free running 64-bit counter that advances each CPU clock under normal operation (TSCH/TSCHL registers).
  - CSL_tscStart() and CSL_tscRead() APIs (6 cycles for tscRead)
- *Simulator Analysis Tool Kit (ATK):*
  - *Function level profiling with only exclusive counts*
  - *Works on multi-threaded applications*

Benchmark Code Performance



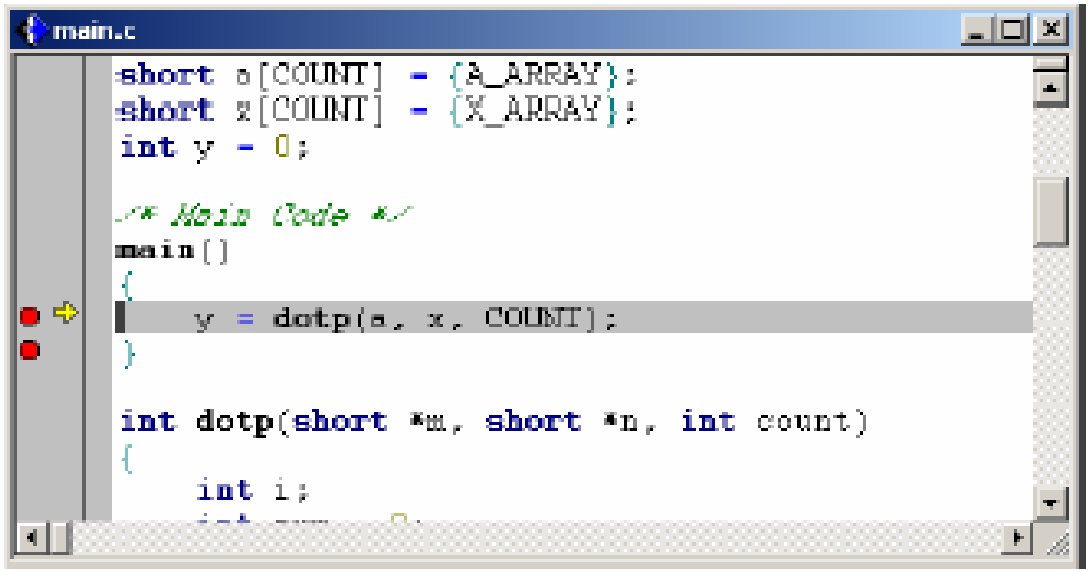
- We set two breakpoints, one before and another after the dotp()
- Now we can benchm

Run to the first breakpoint



Benchmark Code Performance

Run to the first breakpoint



Reset the Clock by double click the item

Benchmark Code Performance

main.c

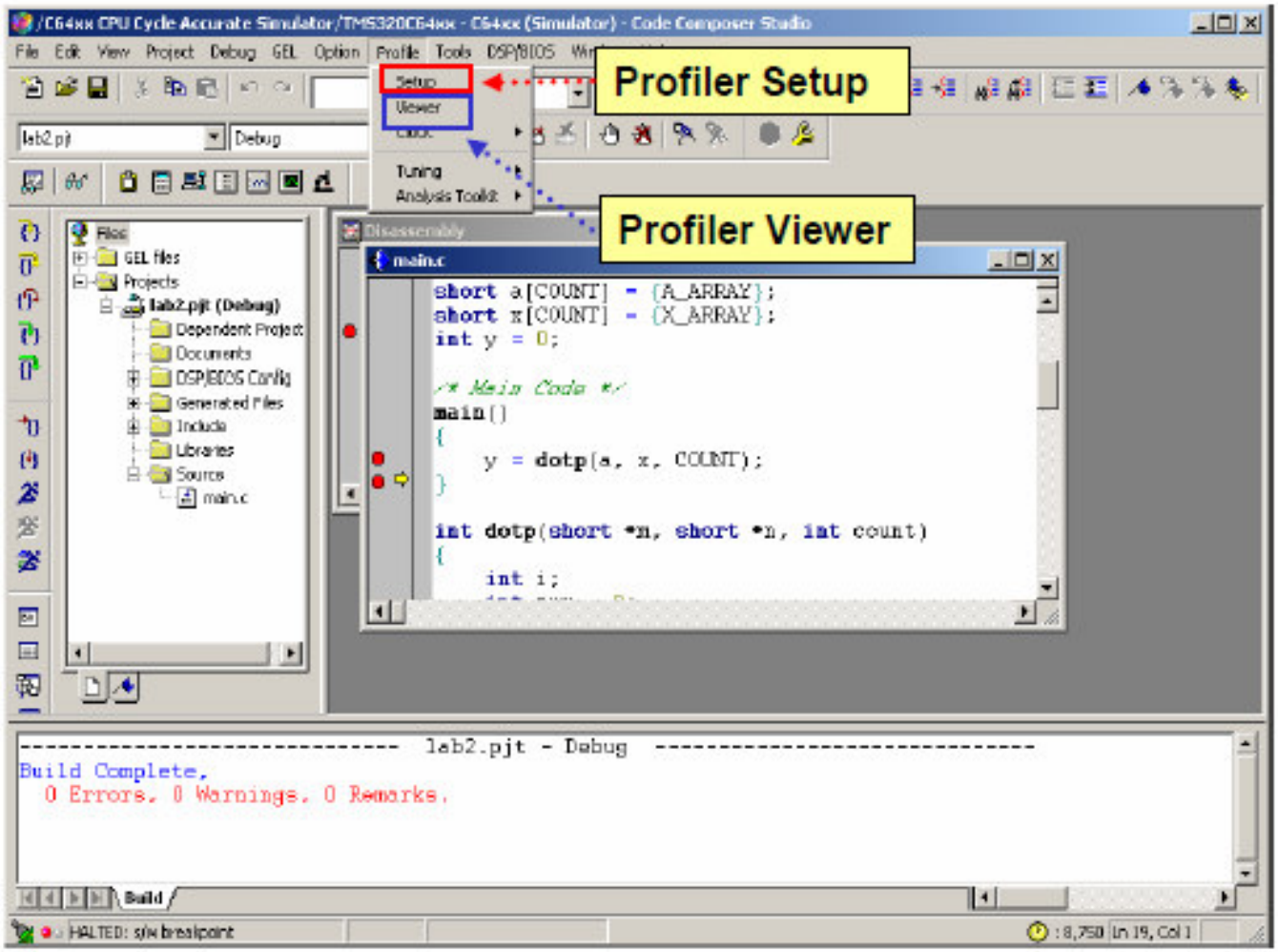
```
short a[COUNT] = {A_ARRAY};
short x[COUNT] = {X_ARRAY};
int y = 0;

/* Main Code */
main()
{
    y = dotp(a, x, COUNT);
}

int dotp(short *n, short *n, int count)
{
    int i;
```

Run to the next breakpoint and get the result of clock cycles to performing this function

Benchmark with Profiler



Setup the Profiler

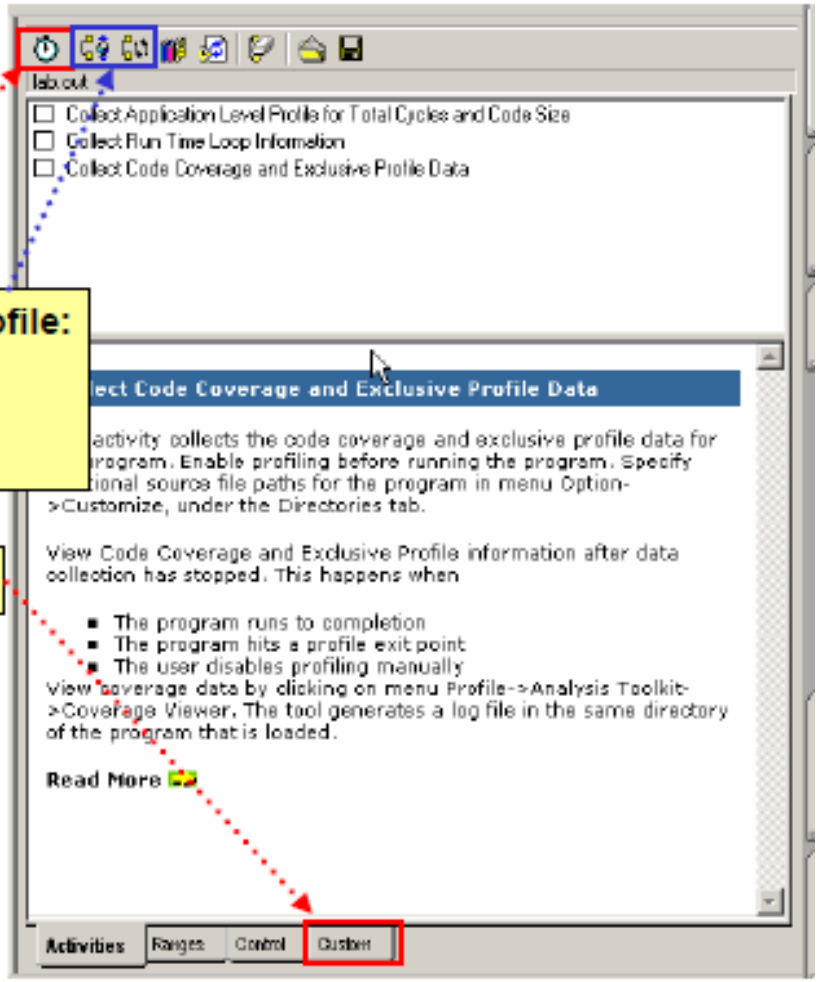
This is a three step process:

Click "Enable Profiling"

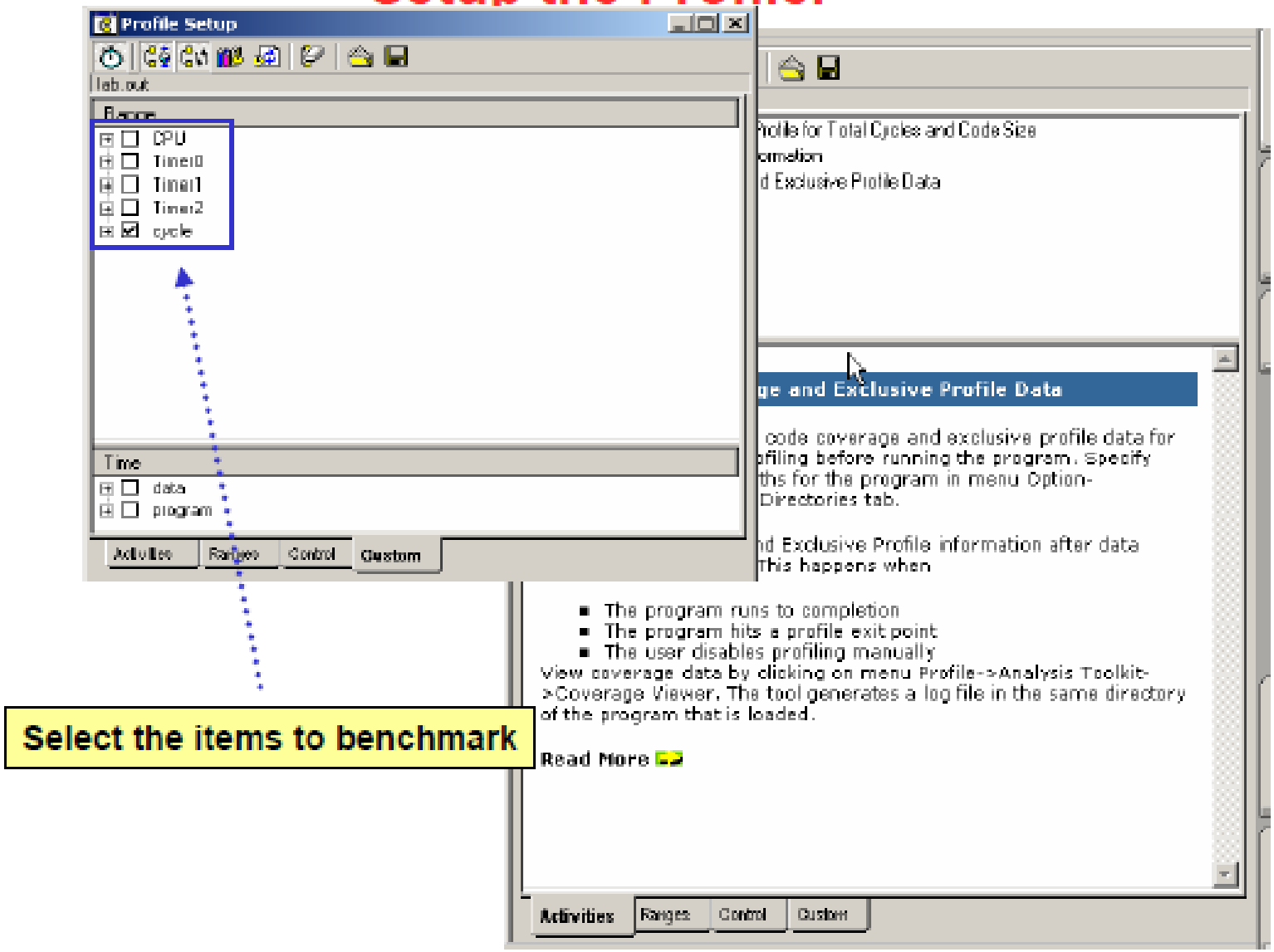
Select the code we want to profile:

- **Functions**
- **Loops**

Select the "Custom" Button

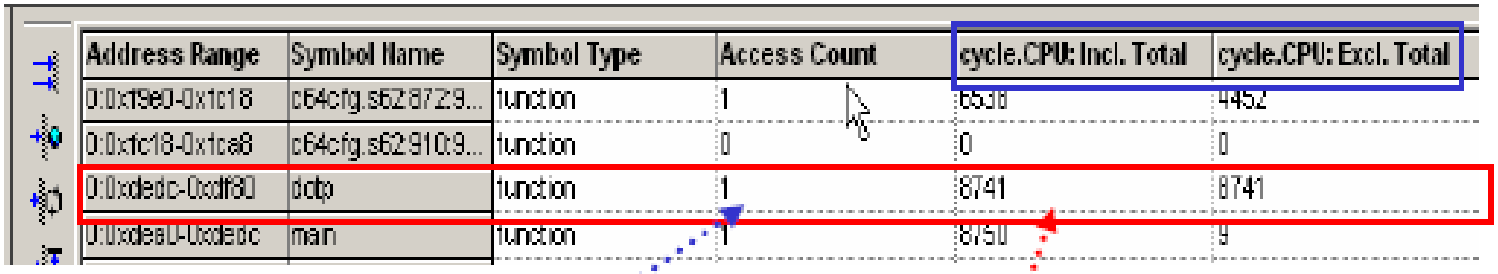


Setup the Profiler



Select the items to benchmark

Open up the Viewer



Address Range	Symbol Name	Symbol Type	Access Count	cycle.CPU: Incl. Total	cycle.CPU: Excl. Total
0:0x19e0-0x1c18	c64cfg.s62 87Z9...	function	1	6538	4452
0:0x1c18-0x1ca8	c64cfg.s62 910 9...	function	0	0	0
0:0xdedc-0xdf80	dobp	function	1	8741	8741
0:0xdea0-0xdedc	main	function	1	8750	9

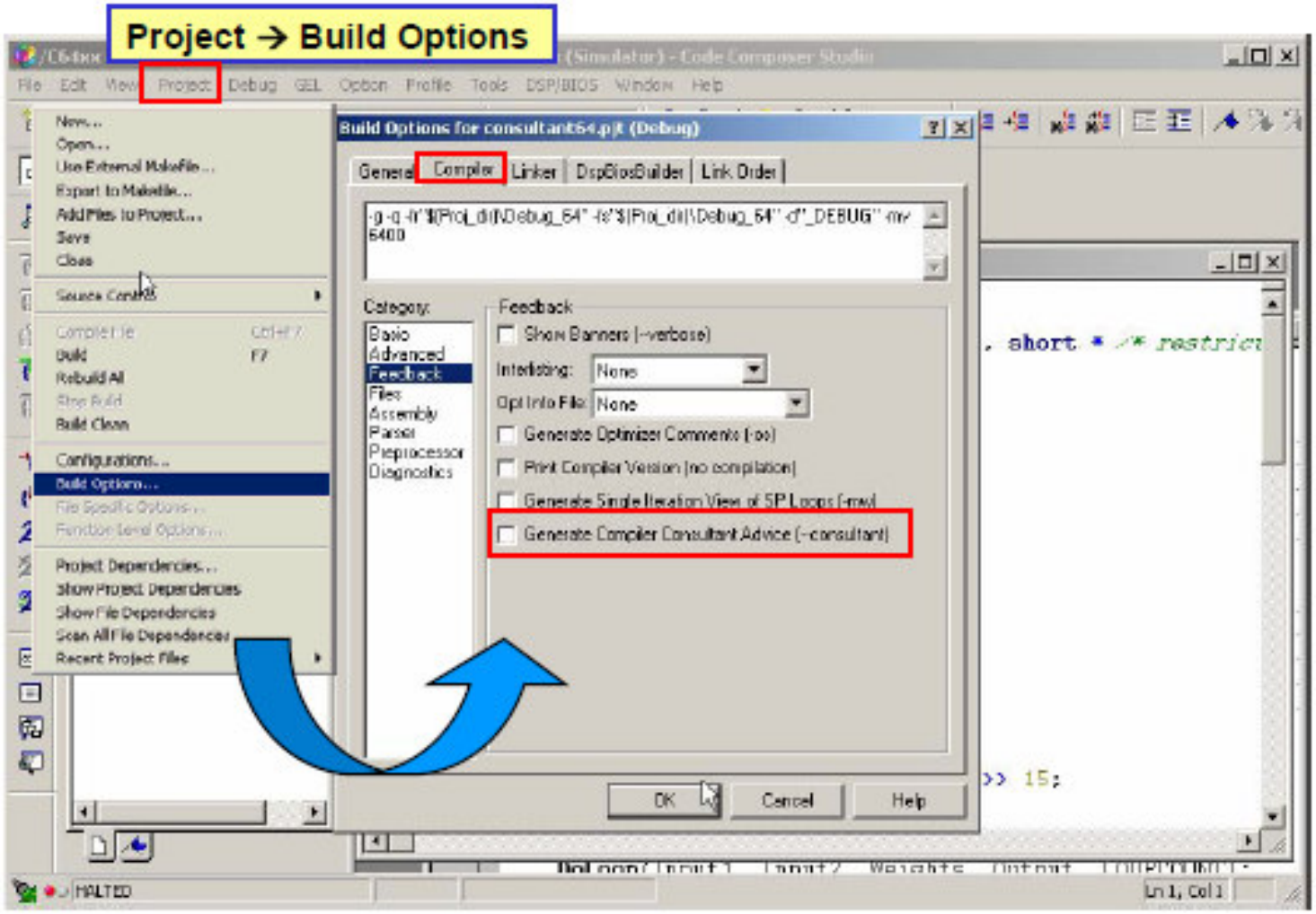
Count

How many times we run into this function

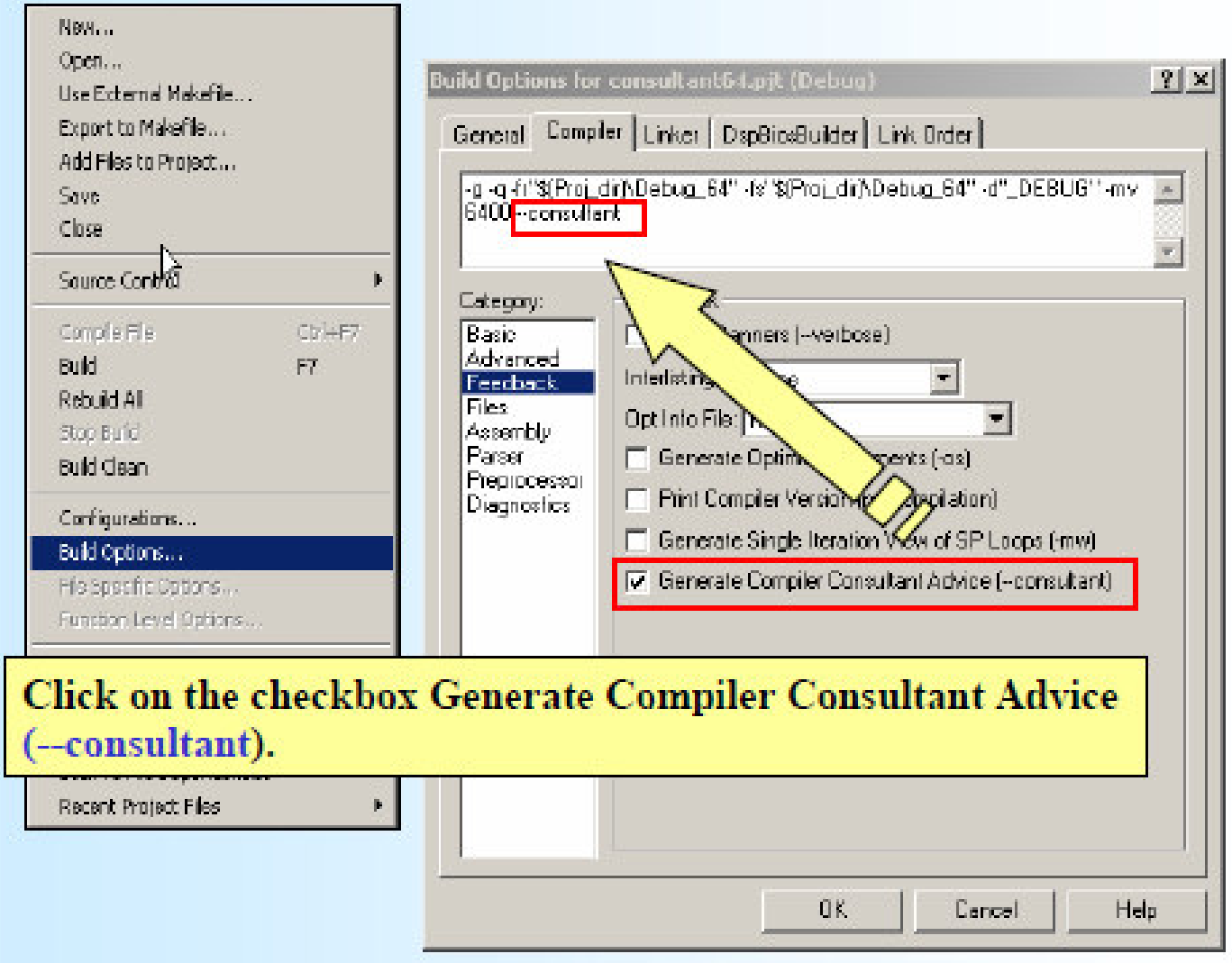
Total

Total number of clock cycles to execute this function

Compiler Consultant

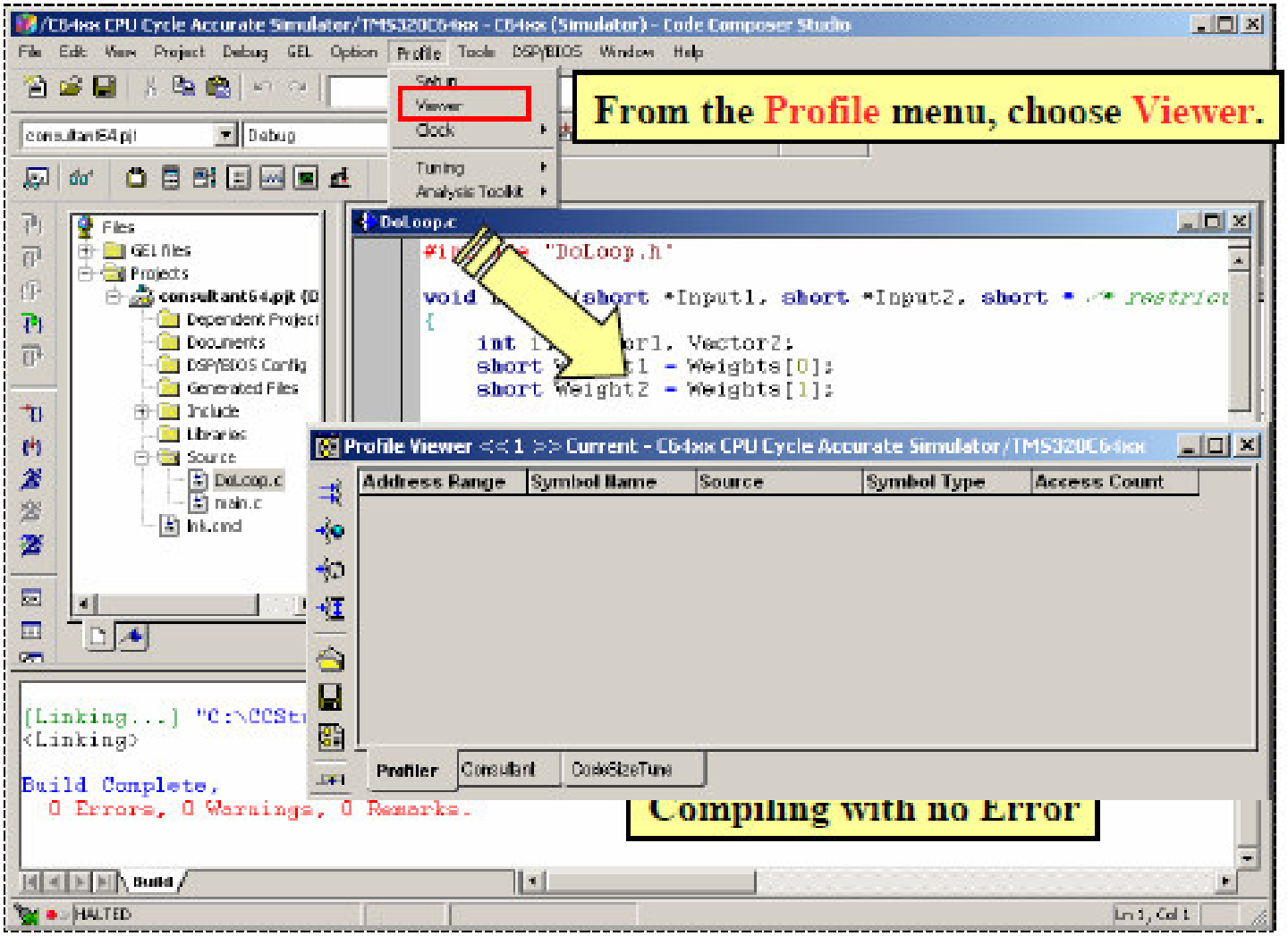


Compiler Consultant

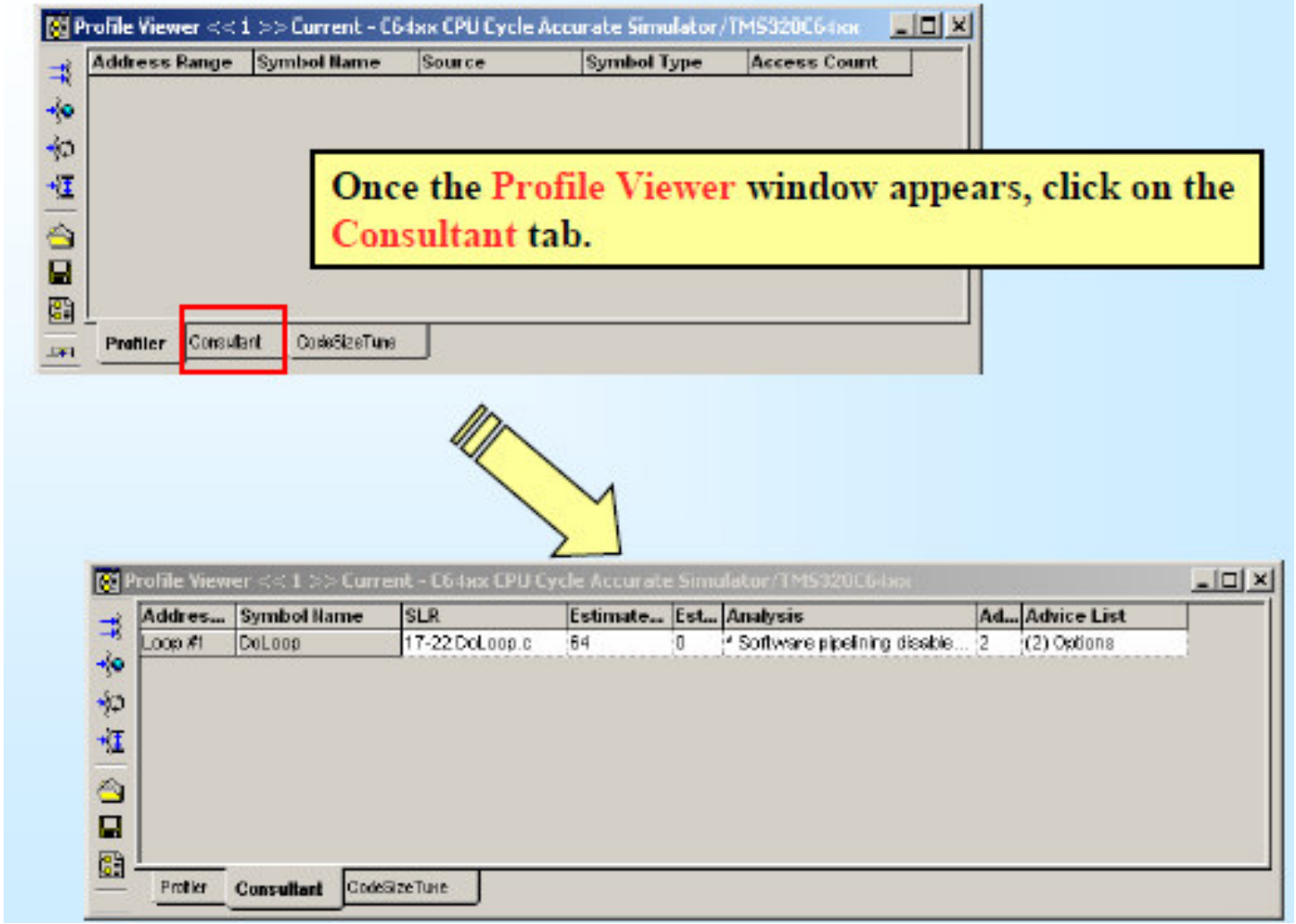


Click on the checkbox Generate Compiler Consultant Advice (--consultant).

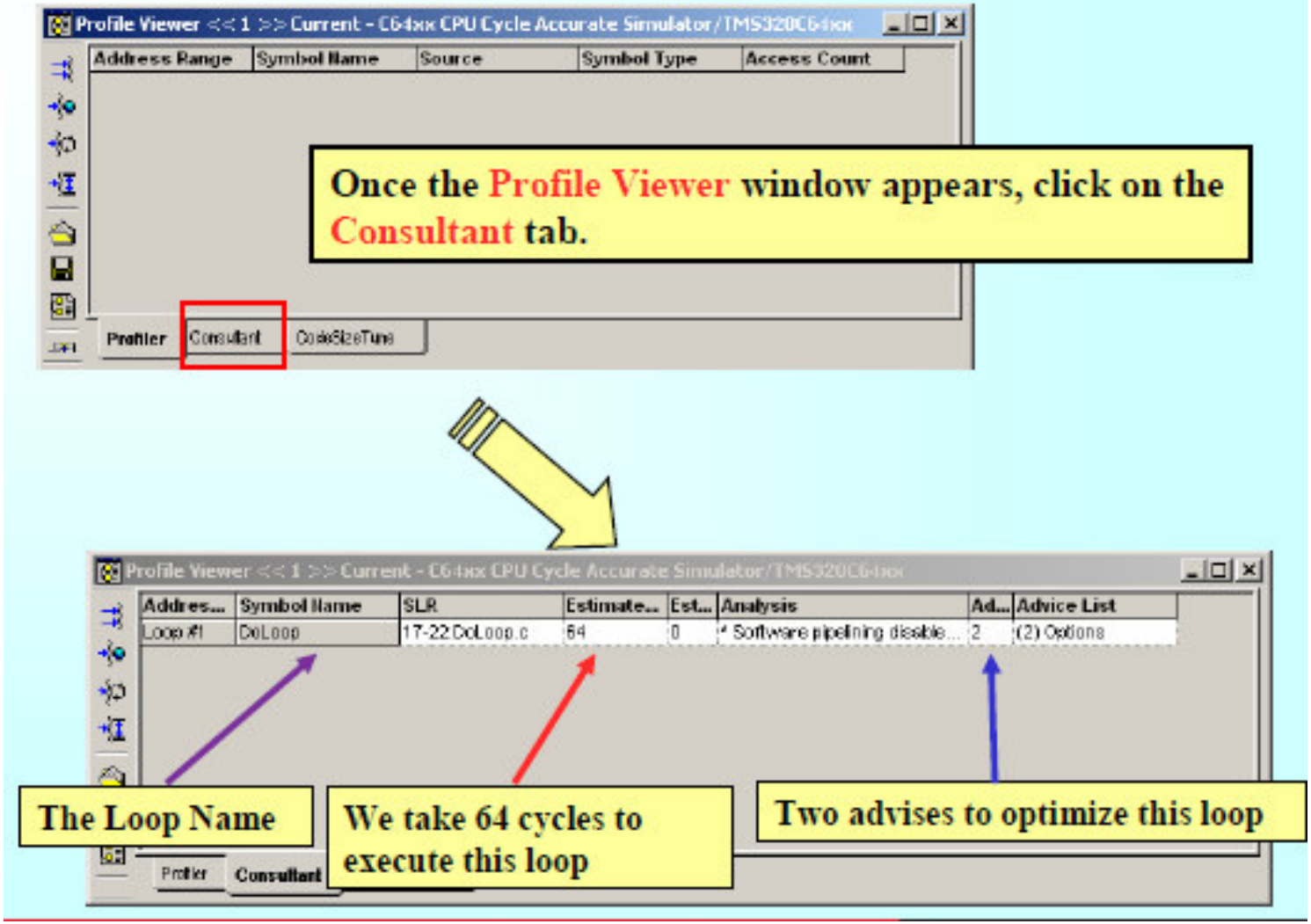
Compiler Consultant



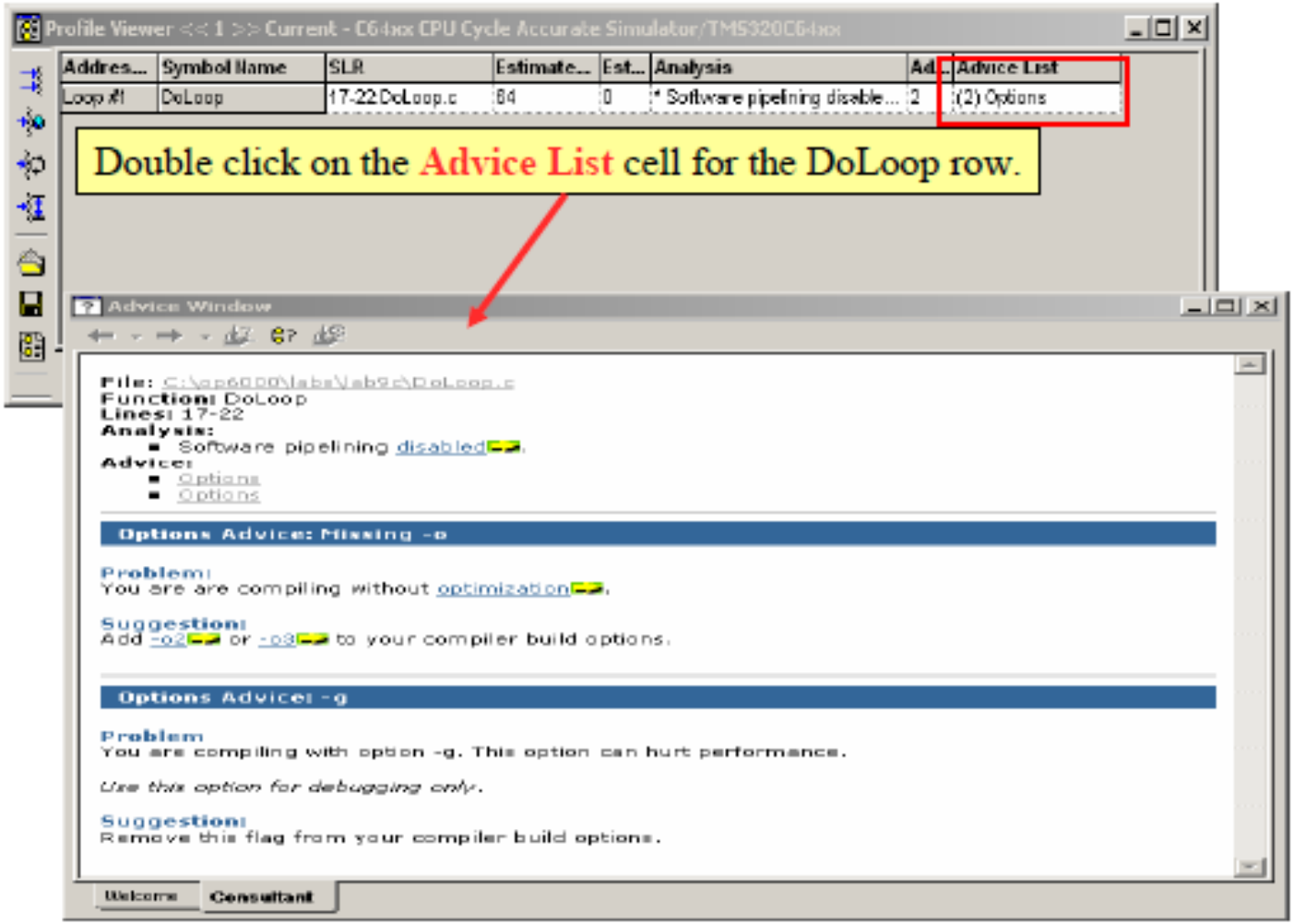
Compiler Consultant



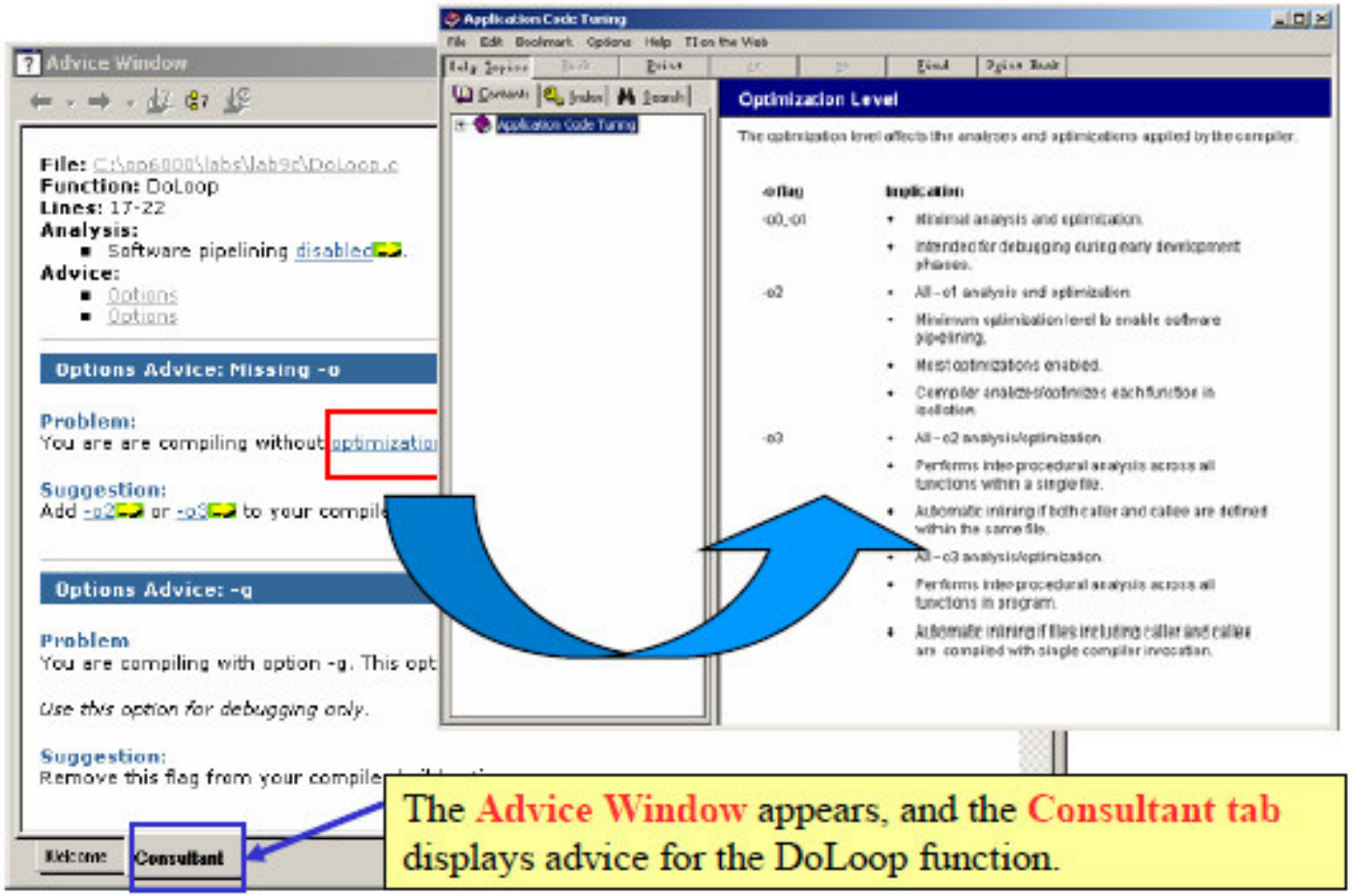
Compiler Consultant



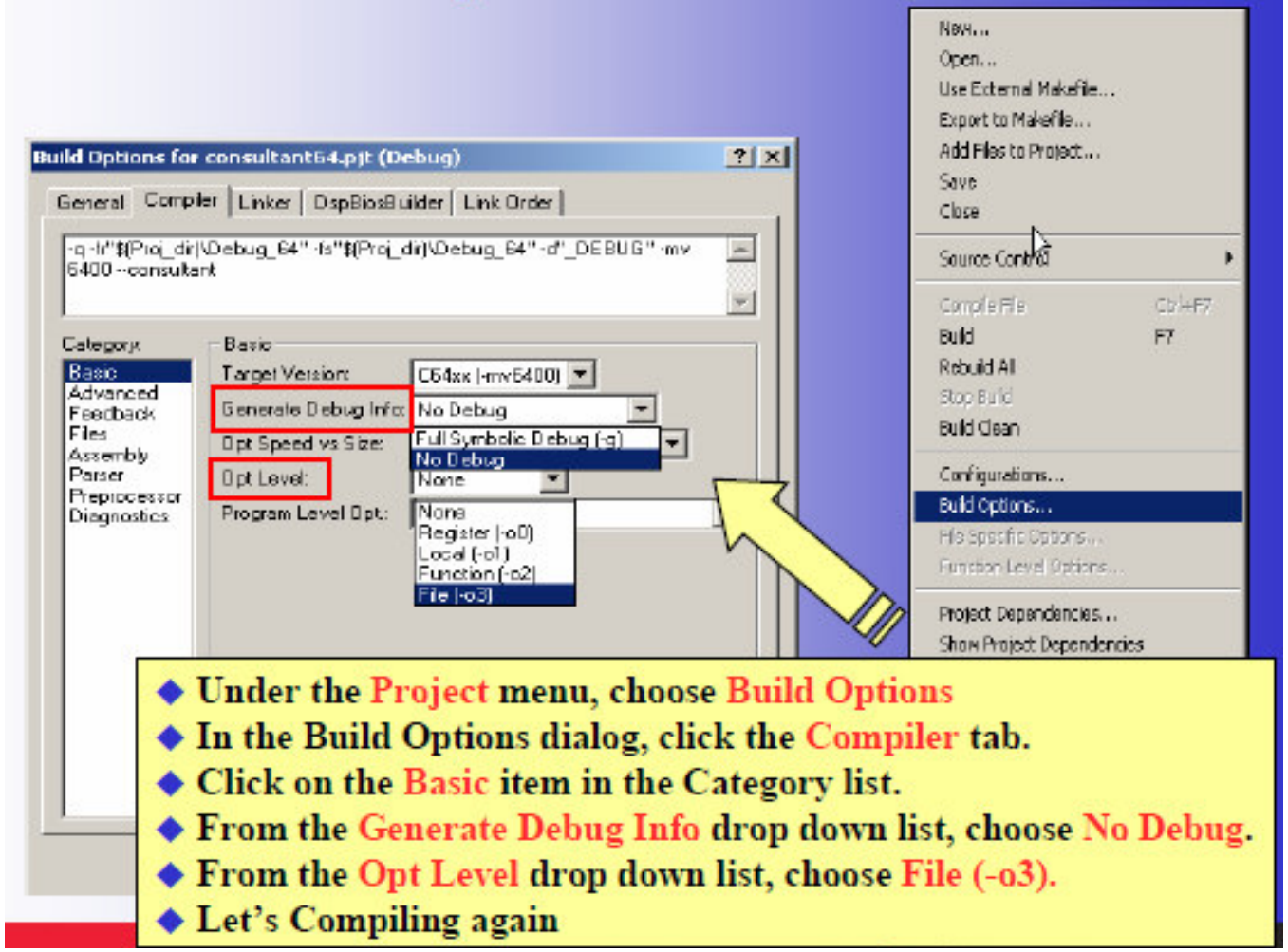
Compiler Consultant



Compiler Consultant

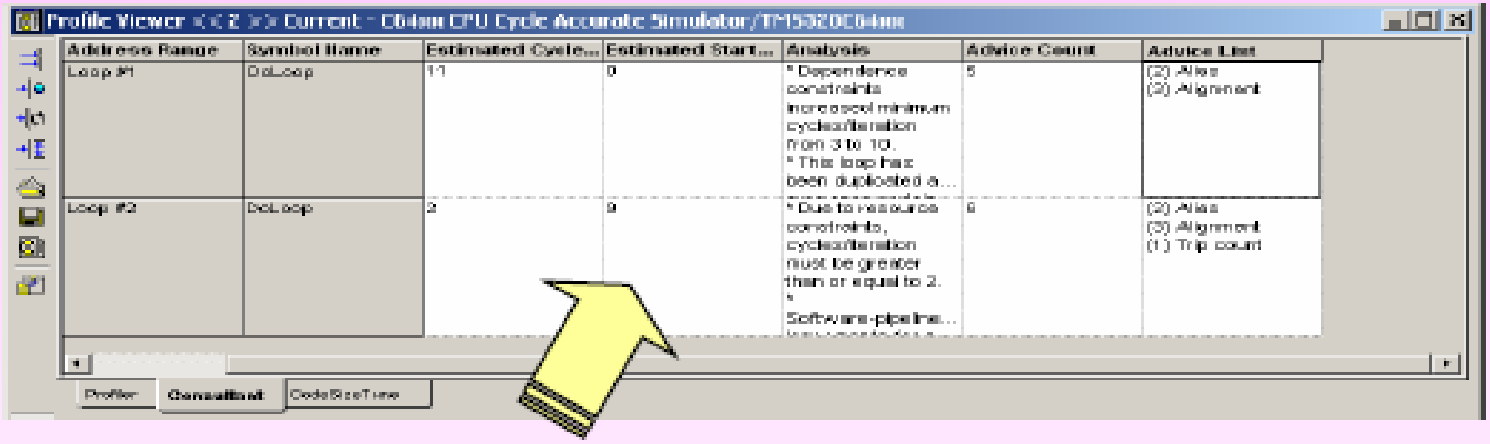


Compiler Consultant



- Under the Project menu, choose Build Options
- In the Build Options dialog, click the Compiler tab.
- Click on the Basic item in the Category list.
- From the Generate Debug Info drop down list, choose No Debug.
- From the Opt Level drop down list, choose File (-o3).
- Let's Compiling again

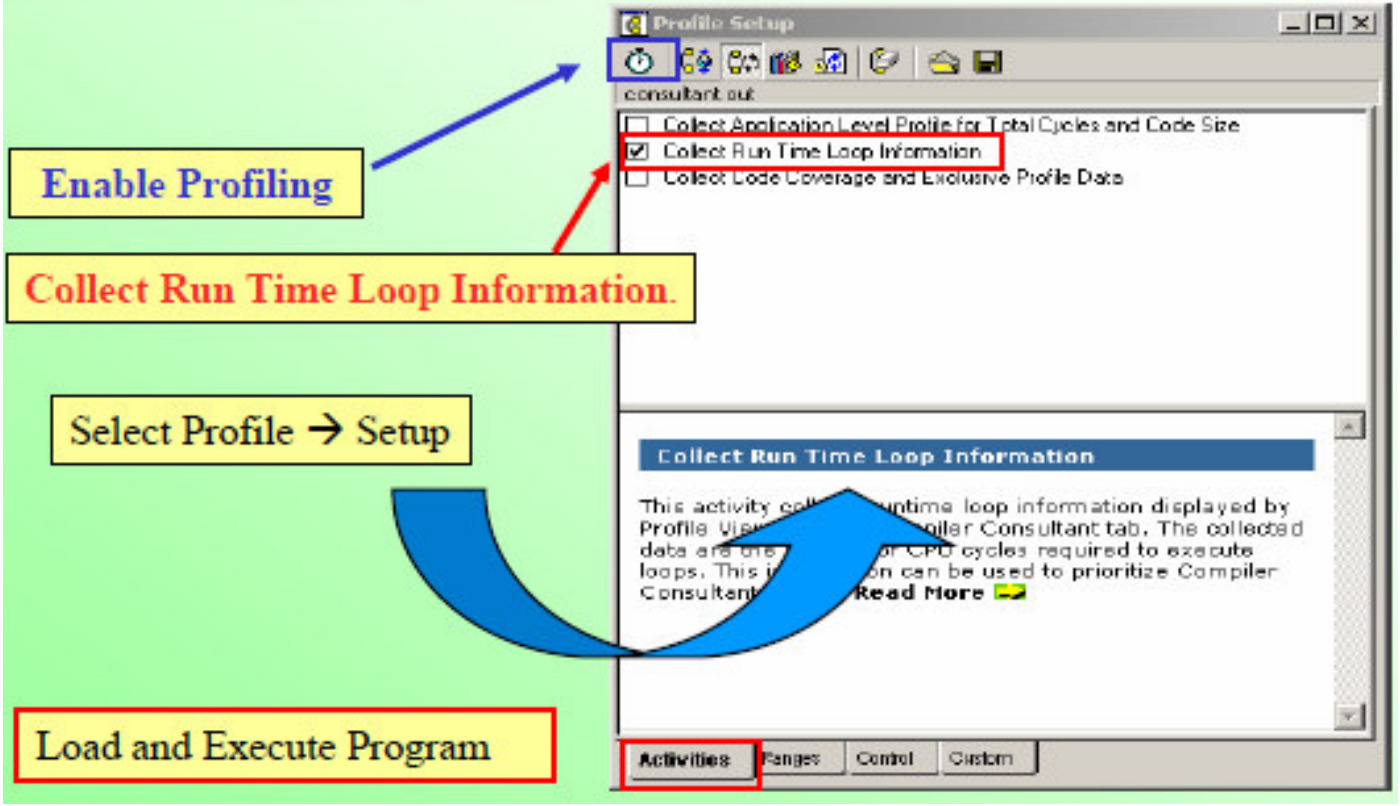
Compiler Consultant



- Loop Duplicated.
- Compiler is unable to determine if one or more pointers are pointing at the same memory address as some other variable. Such pointers are called *aliases*.
- One version of the loop presumes the presence of aliases, the other version of the loop does not.
- The compiler generates code that checks, at runtime, whether certain aliases are present.
- Compiler executes the appropriate version of the loop based on the check

Compiler Consultant

We can profile the code to see which loop versions get selected when the code is executed. From the **File** menu, choose **Load Program** to start the program load.



optimization

Optimizing with the C Compiler
Loop transformations
Writing Intrinsic C code
Fundamental Rules of DSP Programming

Compiler Feedback

- Compiler Gives several important feedback metrics:
  - ii = 2 - loop size has this many cycles in it
  - recurrence bound - minimum number of cycles before beginning next loop iteration
  - X - number of cross paths used in A/B file
  - .L, .S, .D, .M - number of units used of each type
  - .T1, .T2 - load datapaths used in each A/B file
  - Resource bound - minimum number cycles based only on resources
  - …and loop disqualified messages

Software Pipelining details

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

;*   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 parallel
;*      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).

Software Pipelining details

- **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.

Compiler Flags

Read SPRU187 (www.ti.com) TMS320C6000 Optimizing Compiler and also “cl6x –h” on cmd line

- -mv6400+ : chooses the C64x+ ISA
- -mv6700 : generate ‘C67x code (‘C62x is default)
- -mw : generates verbose soft. pipelining info
- -k : keep assembly output around
- -oi0 : disables automatic inlining
- -o2/-o3 : function/file level optimization
- -mu : disables software pipelining
- -mh : allows speculative loads
- -pm –op2 -o3 : program Level Optimization

As we have seen, the optimizer can work miracles, but…

What else can we do with C after turning on the optimizer?

Provide Compiler with More Insight

- **Why you program still not optimized even use –o3 option to turn on the optimizer?**
  1. **Restrict Memory Dependencies (Aliasing)**
  2. **Program Level Optimization: -pm –op2 -o3**
  3. **#pragma UNROLL(*# of times to unroll*);**
     #pragma MUST_ITERATE(*min, max, %factor*);
  5. **#pragma DATA_ALIGN(*variable, 2n alignment*);**

  Like –pm, #pragmas are an easy way to pass more information to the compiler

  The compiler uses this information to create "better" code

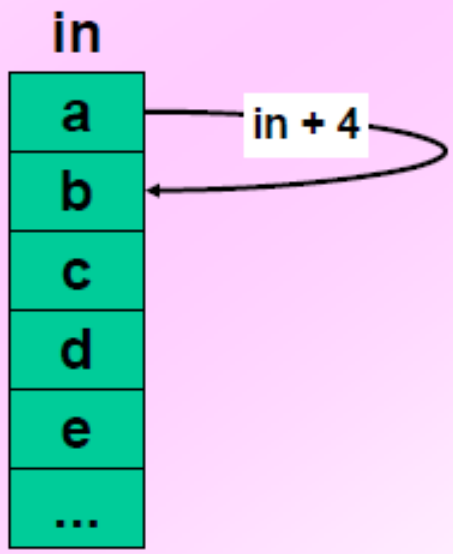
  #pragmas are ignored by other C compilers if they are not supported

1 - Memory Alias Disambiguation

What happens if the function is called like this?

```
fcn(*myVector, *myVector+1)
```

```
void fcn(*in, *out)
{
     LDW    *in++, A0
     ADD    A0, 4, A1
     STW    A1, *out++
}
```



- Can reorder memory references only when they are not aliases
- Aliases access same memory location
- Sometimes Compiler cannot figure out two pointers point to independent addresses or not
- Keyword `restrict` says pointer gives sole access to underlying memory

void fcn(short *restrict in, short *out)

2 – Program Level Optimization (-pm)

- -pm is *critical* in compiling for maximum performance
- -pm creates a temp.c file which includes all C source files, thus giving the optimizer a program-level optimization context
- -op$_n$ describes a program's external references
- -pm requires the use -o3
- Cannot be used as file or function specific option
- Without knowing which -opn option to use, TI couldn't use -pm in default *Release* config
- Unfortunately, -pm cannot provide optimizer with visibility into object code libraries
- External References:
  - For example, if your program modifies a global variable from another code module, -op2 cannot be used
  - Similarly, if your code calls a function in an external module (who's source isn't visible to the optimizer), -op2 cannot be used (and will be overriden)

3 - Pragmas and Assertions

- These are entered in the c code itself
  - tells the compiler what to do, overrides flags
    - `e.g #pragma MUST_ITERATE(2, ,5);`
  - two types of pragma:
    - **static**: DATA_ALIGN, DATA_SECTION, CODE_SECTION, DATA_MEM_BANK Etc.
    - **dynamic**: MUST_ITERATE, UNROLL
      - key to achieving **SIMD** optimized code.

3-Pragmas and Assertions

- #pragma MUST_ITERATE(a, b, c);
  - a - min number of times
  - b - max number of times
  - c - multiple of c times
  - #pragma PROB_ITERATE(a,b); - may iterate
- #pragma UNROLL(n)
  - Forces the compile to unroll the next immediate loop by n
- Assertions give information to compiler (and user)
  - _nassert(<rule>); e.g. short *x; _nassert((int) x % 8 == 0);

Inner Loop Unrolling

- Simple Loop

```
for (j = 0; j < 2*nh; j += 2)
{
    real += h[j+0] * x[i-j+0] - h[j+1] * x[i-j+1];
    imag += h[j+1] * x[i-j+0] + h[j+0] * x[i-j+1];
}
```

- Unrolled Loop
  - runs 1/2 as many times, performs twice as much

```
for (j = 0; j < 2*nh; j += 4)
{
   real += h[j+0] * x[i-j+0] - h[j+1] * x[i-j+1] +
           h[j+2] * x[i-j-2] - h[j+3] * x[i-j-1];
   imag += h[j+1] * x[i-j+0] + h[j+0] * x[i-j+1] +
           h[j+3] * x[i-j-2] + h[j+2] * x[i-j-1];
}
```

Outer Loop Unrolling

- Simple 2d Loop

```
for (i = 0; i < 2*nr; i += 2) {
    imag = 0; real = 0;
    for (j = 0; j < 2*nh; j += 2) {
        real += h[j+0] * x[i-j+0] - h[j+1] * x[i-j+1];
        imag += h[j+1] * x[i-j+0] + h[j+0] * x[i-j+1];
    }
    r[i  ] = (real >> 15);  r[i+1] = (imag >> 15);
}
```

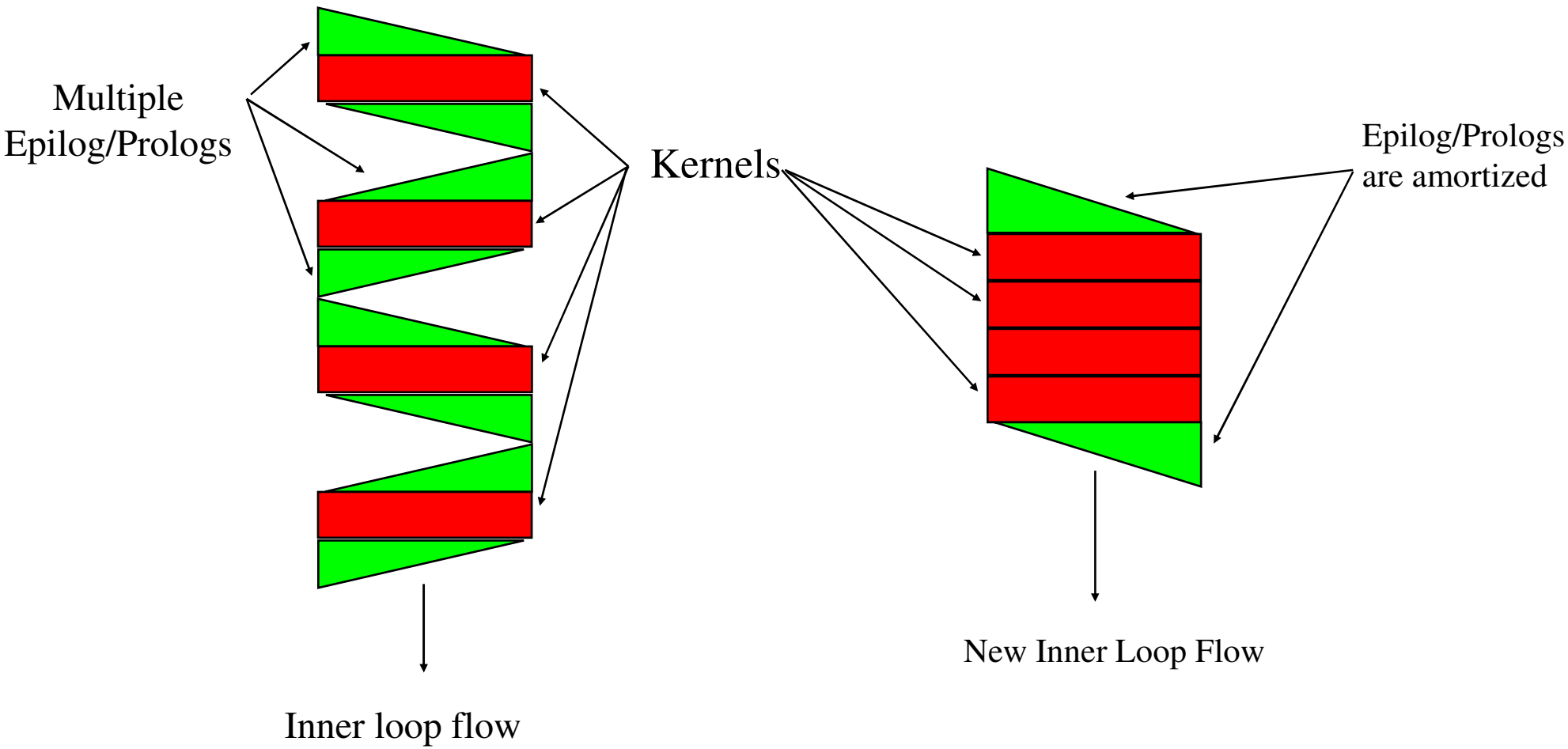
- Unrolled Loop
  - runs 1/2 as many times, performs twice as much

```
for (i = 0; i < 2*nr; i += 4) {
    imag0 = 0; real0 = 0; imag1 = 0; real1 = 0;
    for (j = 0; j < 2*nh; j += 2) {
        real0 += h[j+0] * x[i-j+0] - h[j+1] * x[i-j+1];
        imag0 += h[j+1] * x[i-j+0] + h[j+0] * x[i-j+1];
        real1 += h[j+0] * x[i-j+2] - h[j+1] * x[i-j+3];
        imag1 += h[j+1] * x[i-j+2] + h[j+0] * x[i-j+3];
    }
    r[i  ] = (real0 >> 15); r[i+1] = (imag0 >> 15);
    r[i+2] = (real1 >> 15); r[i+3] = (imag1 >> 15);
}
```

Effect of Outer Loop Unrolling

- Easy to see effect of **inner** loop unrolling
- **Outer** loop unrolling is more subtle
  - provides more parallelism to inner loop
  - also reduces epilog/prolog overhead
  - significant for inner loops with **smaller 'trip' counts**

Outer Loop Unrolling



UNROLL(*# of times to unroll*)

Example :

```
#pragma UNROLL(2);
for(i = 0; i < count ; i++) {
      sum += a[i] * x[i];
}
```

- **Tells the compiler to unroll the for() loop twice**
- **The compiler will generate extra code to handle the case that count is odd**
- **The #pragma must come right before the for() loop**
- **UNROLL(1) tells the compiler not to unroll a loop**

MUST_ITERATE(*min, max, %factor*)

Example :

```
#pragma UNROLL(2);
#pragma MUST_ITERATE(10, 100, 2);
for(i = 0; i < count ; i++) {
      sum += a[i] * x[i];
}
```

- **Gives the compiler information about the trip (loop) count : In the code above, we are *promising* that: count >= 10, count <= 100, and count % 2 == 0**
- **If you break your promise, you might break your code**
- **Allows the compiler to remove unnecessary code**
- **Modulus (%) factor allows for efficient loop unrolling**
- **The #pragma must come right before the for() loop**

SPLOOP buffer and loop unrolling

- Unrolling increases ii and the dynamic length of a single iteration
- Loop buffer not used
  - ii (initiation interval) > 14 cycles
  - Dynamic length (of a single iteration) > 48 cycles
  - The optimizer completely unrolls the loop

fir_cmpl_cn + tweaks

```
void fir_cplx_cn (
    const short *restrict x, const short *restrict h,
    short       *restrict r,
    short nh, short nr )
{
    int i, j, imag, real;
    _nassert((int) h % 8 == 0);
    _nassert((int) x % 8 == 0);
    _nassert((int) r % 8 == 0);

    #pragma MUST_ITERATE(4, ,4);
    #pragma UNROLL(4);
    for (i = 0; i < 2*nr; i += 2)
    {
        imag = 0; real = 0;
        #pragma MUST_ITERATE(2, ,2);
        #pragma UNROLL(2);
        for (j = 0; j < 2*nh; j += 2)
        {
            real += h[j+0] * x[i-j+0] - h[j+1] * x[i-j+1];
            imag += h[j+1] * x[i-j+0] + h[j+0] * x[i-j+1];
        }
        r[i  ] = (real >> 15); r[i+1] = (imag >> 15);
    }
}
```

- c with pragmas and assertions

fir_cplx_co.asm: loop kernel

```
L3:    ; PIPED LOOP KERNEL

   [!A0]   SUB     .D2     B18,B21,B18    ; |108|
|| [!A0]   ADD     .S1     A5,A9,A9       ; |109|
||         DOTP2   .M2X    B4,A23,B21     ; |109|
||         DOTP2   .M1X    B4,A7,A17      ; |109|
||         LDW     .D1T1   *-A8(4),A17    ; @|108|

           MV      .D1     A4,A19  ;split long life
|| [!A0]   SUB     .D2     B7,B21,B7      ;|108|
||         DOTP2   .M2X    B5,A25,B21     ; |109|
||         DOTPN2  .M1X    B4,A6,A19      ; |108|
|| [!A0]   SUB     .S1     A18,A5,A18     ; |108|

   [ B0]   BDEC    .S2     L3,B0          ;
|| [!A0]   SUB     .D2     B16,B21,B16    ; |108|
||         DOTP2   .M2X    B5,A7,B21      ; |109|
||         DOTPN2  .M1X    B5,A19,A7      ; |108|
|| [!A0]   ADD     .D1     A6,A21,A21     ; |109|

           DOTPN2  .M1X    B4,A19,A5      ; |108|
||         DOTP2   .M2X    B5,A23,B21     ; |109|
|| [!A0]   SUB     .D2     B20,B21,B20    ; |108|
|| [!A0]   ADD     .D1     A17,A22,A22    ; |109|
||         PACKLH2 .S1     A7,A7,A19      ; @|109|
||         PACKLH2 .L1     A6,A6,A23      ; @|109|

   [!A0]   ADD     .D2     B21,B6,B6      ; |109|
|| [!A0]   ADD     .D1     A17,A24,A24    ; |109|
||         DOTPN2  .M2X    B4,A5,B21      ; @|108|
||         DOTP2   .M1X    B4,A19,A5      ; @|109|

   [!A0]   ADD     .D2     B21,B8,B8      ; |109|
||         DOTPN2  .M2X    B5,A5,B21      ; @|108|
||         DOTPN2  .M1X    B4,A7,A5       ; @|108|
||         PACKLH2 .S1     A5,A5,A25      ; @|109|
||         PACKLH2 .L1     A17,A17,A5     ; @|109|
||         LDDW    .D1T1   *--A8,A5:A4    ; @@|108|

   [!A0]   ADD     .D2     B21,B9,B9      ; |109|
|| [!A0]   SUB     .S1     A3,A19,A3      ; |108|
|| [!A0]   SUB     .L1     A16,A7,A16     ; |108|
||         DOTPN2  .M2X    B5,A6,B21      ; @|108|
||         DOTP2   .M1X    B5,A5,A6       ; @|109|
||         LDDW    .D1T1   *+A8(8),A7:A6  ; @@|108|

   [ A0]   SUB     .D1     A0,1,A0        ;
|| [!A0]   SUB     .L1     A20,A5,A20     ; |108|
|| [!A0]   ADD     .S2     B21,B17,B17    ; |109|
||         DOTPN2  .M2X    B5,A17,B21     ; @|108|
||         PACKLH2 .S1     A4,A4,A7       ; @|109|
||         DOTP2   .M1X    B4,A25,A17     ; @|109|
||         LDDW    .D2T2   *B19++,B5:B4   ; @@|108|
```

- Assembly code output from compiler –k with –disable:sploop

C64x+ C code Optimization

- Optimizing with the C Compiler
- Loop transformations
- Writing Intrinsic C code
- Fundamental Rules of DSP Programming

Writing Intrinsic C code

- C can use instructions directly using intrinsics
  - these are like function calls
  - directly instantiates an instruction
  - can have direct control over instruction selection
    - Not leaving it to chance
- Examples are:
  - INST_NAME src1, src2, dst **<=>**
  - dst =_<instr. name>(src1, src2)
- Almost every instruction can be used in this way
- 1 step up from linear assembly code
- #include <c6x.h> required

Intrinsic Examples

- `c = _dotp2(a0a1, b0b1);`
- `y = _max2(a0a1, b0b1);`
- `Im_re = _cmpy(a1a0, b1b0);`
- `z = _add2(a0a1, b0b1);`
- `long long h3h2h1h0 = _amem8( &h[4]);`
- `int h1h0 = _loll(h3h2h1h0);`
- `int h3h2 = _hill(h3h2h1h0);`
- `long long h3h2h1h0 = _itoll(h3h2, h1h0);`
- `int h0h1 = _packlh2(h1h0, h1h0);`

  - Notice we use "**helpful**" **variable names**

Intrinsic C code: memory accesses

- The memory intrinsic functions are written using
  - _amem8_const() and _amem8(), _amem4() and _mem4();
    - aligned and non aligned data to constant and non-constant data
    - _const form allows constant arrays to be accessed
    - e.g `long long a = _amem8( &b[4] );`
  - Preserves memory dependencies in c code
  - **Do not use pointer recasting** to accomplish this
    - can destroy memory tracking
    - not ansii c compliant anymore

Memory Dependencies

```
short *x;
long long *y;

x[0] = 0;

asm(“  NOP  1”);

for (i=0; i < N; i++)
{
 y[i] = *((long long *) &x[i]);
}
x[0] = 0;
```

Bad code

```
short *x;
long long *y;

x[0] = 0;
for (i=0; i < N; i++)
{
  y[i] = _mem8(&x[i]);
}
```

Good code

- The memory dependency is preserved
  - _memxyz() calls prevent the compiler from losing track

Helpful Tricks

- You can use DOTP2 for 16-bit self-terminating counters in loops that are bottlenecked on L, S, and D. This works when your count is <= 32768.

```
      MV        count, i       ; outside the loop
           ; ...
[i]   SUB      i, 1, i ; inside the loop
```

is equivalent to

```
      MVKL   0xFFFF0001,   incr
      MVKH   0xFFFF0001,   incr
      NEG    count, i    ; outside the loop
           ; ...
      DOTP2  incr,  i, i ; inside the loop
```

- Instead of:

```
a = b;
```

; you can use the M unit:

```
a = _mvd(b);
```

Helpful Tricks: Smart Macros

- Right most bit detection

```
#define _rmbd(a,b) _lmbd(a,_bitr(b))
```

- Bit count over 32 bits

```
#define _bitc32(a) _dotpu4(_bitc4(a),0x01010101)
```

- Bit reverse of 3 LSBs

```
#define _bitr3(value) _extu(_bitr(value),0,29)
```

C64x+ C code Optimization

- Optimizing with the C Compiler
- Loop transformations
- Writing Intrinsic C code
- Fundamental Rules of DSP Programming

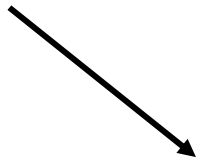
Some rules for DSP applications

- Avoid use of global variables in application code
  - Ultimately hurts system reliability / Maintainability
  - Performance / Compiler optimizations are often blocked by global-ness
- Try to keep structure depths as small as possible
  - x->y->z[i]->w is very inefficient to access
    - sometimes algorithms are still more efficient - e.g. linked lists, trees etc.
  - keep array dimensions to no more than 2 x[][] is enough
- Provide as much info. as possible to the compiler.
  - Restrict keywords, MUST_ITERATE pragmas, nasserts.

Structural Optimizations

- Some data structures are better
  - structs of arrays are better than arrays of structs
  - one exposes more parallelism with more probability to use SIMD instructions

```
for (i=0; i < N; i++)
{
   y += a_st[I]->a * st_b[I]->b;
}
```



```
for (i=0; i < N; i++)
{
   y += st_a->a[I]*st_b->b[I];
}
```

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:

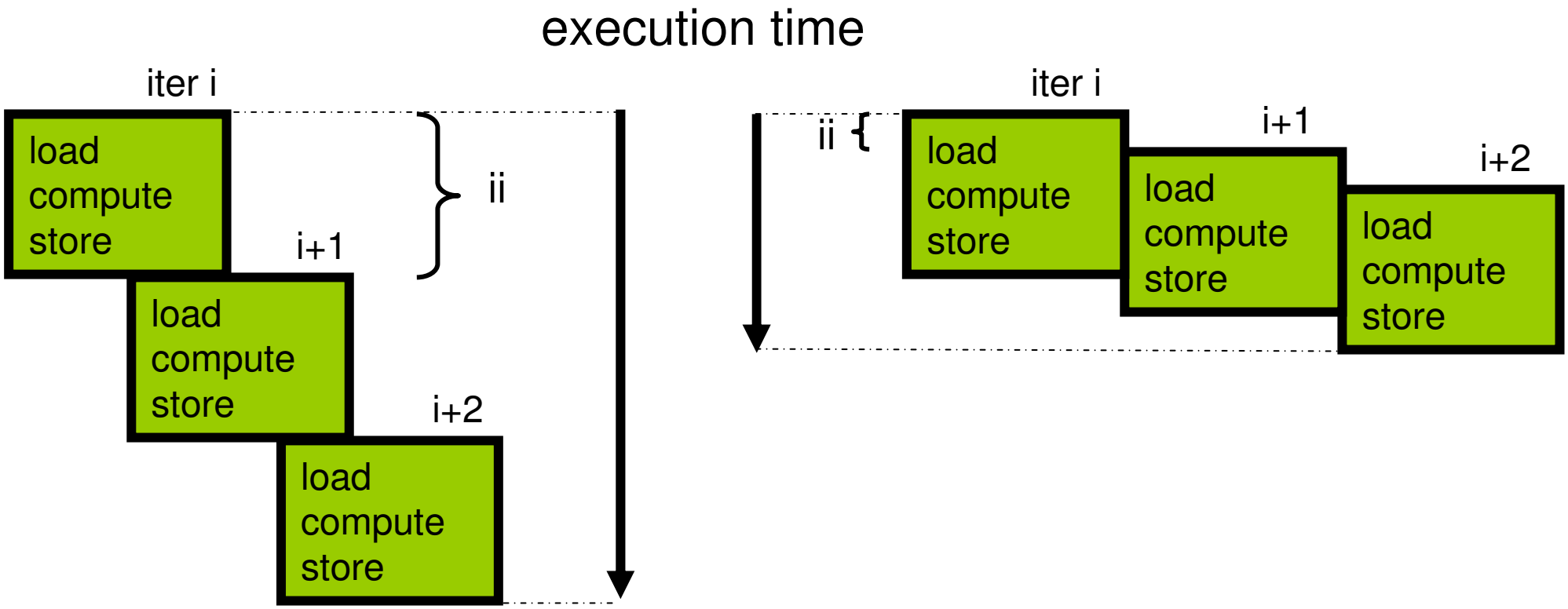
```
myfunc(type1 input[restrict],
       type2 *restrict output)
```

Restrict Qualifiers (cont.)

myfunc

original loop

restrict qualified loop



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 *at top-level of function* and restrict qualify pointers instead.
- Use local pointers in function/loop instead of original pointers.

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

myfunc(_str *s)
{
    _str *t;

    // create local pointers at
    // top-level of function
    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 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->y < 25)
{
    g->p->a[i++] = …
}
```

Hand-optimized Loop:

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

while (y < 25)
{
    a[i++] = …
}
```

Example: Restrict and Structures

Original struct.c:

```
struct_refs(_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 –os –mt –mv64+ struct.c

Extracted from struct.asm:

```
;** -  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
…
;*       i = 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.)

```
struct_refs(_str *s)
{
    int *restrict p = s->data->p;
    int *restrict q = s->data->q;
    int sz          = s->data->sz;

    int i;

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

cl6x –o –os –mw –mv64+ struct.c

Extracted from struct.asm:

```
;** -   // 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         : 2
;*      Loop Carried Dependency Bound(^)    : 0
;*
;*      ii = 2 Schedule found with 3 iterati…
```

Hand-optimized struct.c:

Bottom line:
1 cycle/result,
44 bytes

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

If Statements

- Compiler will if-convert loops with small if statements:

Original C code:

```
f (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 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 must transform large if statements because compiler does not know if this is profitable.

Structural Improvements

- Some program and data structures are better than others
  - if-for is better than for-if
  - one software pipelines, one doesn't

```
for (i=0; i < N; i++)
{
   if (case0)
     do_exec0[i];
   else if (case1)
     do_exec1[i];
   else
     do_exec2[i];
}
```

```
if (case0)
   for (i=0; i < N; i++)
     do_exec0[i];
else if (case1)
   for (i=0; i < N; i++)
     do_exec1[i];
else
   for (i=0; i < N; i++)
     do_exec2[i];
```

  - Complex control blocks or compromises pipelining

Structural Improvements - *fast* Case statements

- Complex conditional statements can be optimized
  - Sometimes branching is too costly
  - conditional code can be thought of as a computation

```
if (case1(x, y, z))
  y = comp_res1(x,y,z);
else if (case2(x,y,z))
  y = comp_res2(x,y,z);
else
  y = comp_res3(x,y,z);
```

```
y_t = comp_res3(x,y,z);

if (case2(x,y,z))
  y_t = comp_res2(x,y,z);

if (case1(x, y, z))
  y_t = comp_res1(x,y,z);

y = y_t;
```

  - priority is achieved by conditional replacement

More Structural Optimizations

- Certain control structures are problematic to compiler
  - static conditionals should be ‘hoisted’ out of loops
  - heavy use of structures and more so unions can prevent this

```
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];
}
```

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, gcd);

```
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 –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;
;**      -----------------------g4:
```

“Variable type” optimizations

- The type of variable used will affect performance
  - use 32 bit precision whenever possible for control variables

```
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
```

- 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!

Some more examples of variable optimizations

- Using obvious type definitions can affect the code a lot

```
Int32 i;
Int16 sum, p0;
for (i=0; i < N; i++) {
   p0 = 1 + x[i];
   sum += (p0 * y[i])>>16;
   z[i] = (Int16) (sum * table[i])>>12;
}
```

- This code will use a lot of instructions to make data Int16
- Redefining types and using intrinsics, the unneeded instructions and the >>16 can be removed. Using local variables within a loop can free up register resources at function level

```
Int32 i;
for (i=0; i < N; i++) {
 Int32 sum, p0;
   p0 = 1 + x[i];
   sum += _mpyhl(p0, y[i]);
   z[i] = _mpy(sum * table[i])>>12;
}
```

Complex Calculations - a+jb

- Complex data is ideal for c64x+ architecture
  - inherent parallelism
  - but must be organized correctly
  - Data structures are of the form

```
Int16 dataRe[SIZE], dataIm[SIZE];
```

  - Most complex computations are linked, even if not (more later) it is better to do this:

```
Int16 data[2*SIZE];
```

  - Basically only 50% of memory accesses are needed as real and imaginary parts are loaded together

Complex calculation examples - summing up

```
void add_up(Int16 *xRe, Int16 *xIm,
            Int16 *sumRe, Int16 *sumIm)
{
   Int32 i;
   sumRe[0] = 0;
   sumIm[0] = 0;
   for (i=0; i < N; i++){
     sRe += xRe[i];
     sIm += xIm[i];
   }
}
```

```
void add_up(Int16 *x, Int16 *sum)
{
   Int32 i, s;
   for (i=0; i < N; i++){
     s = _add2(s, _amem4(x[2*i]));
   }
}
```

- Performance can be at least increase by 100%
- Parallelism is 2x, memory accesses are 32-bit wide

Complex calculation examples - multiplication

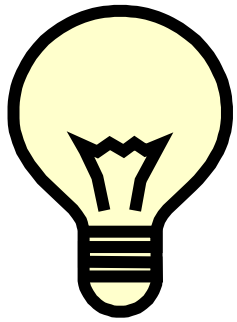
```
Int16 * xRe, * xIm, *yRe, *yIm;
prodRe = xRe[i] * yRe[i] – xIm[i] * yIm[i];
prodIm = xRe[i] * yIm[i] + xIm[i] * yRe[i];
```

```
Int32 * x, *y;
Long long Im_Re;
Im_Re  = _cmpy(x[i], y[i]);
prod = _pack2(_loll(Im_Re), _hill(Im_Re));
```

- Performance increased by 200%
- Code size also reduces

Summary
Tips for Developing Efficient Code

- *Understand/exploit .asm file comments generated when compiling with –os and –mw.*
- *Use MUST_ITERATE pragmas and restrict qualifiers.*
  --- Remember, -mt does not cover structure-field pointers.
- *Pull structure references out of loops and especially loop control.*
- *Reduce complexity/length of if statements.*
- *Split large loops.*



Reference Documents

- SPRU198 -- TMS320C6000 Programmer's Guide
- SPRU187 -- TMS320C6000 Optimizing Compiler v 6.1 User's Guide
- SPRA666 -- Hand-Tuning Loops and Control Code on the TMS320C6000