C6000 Host Intrinsics

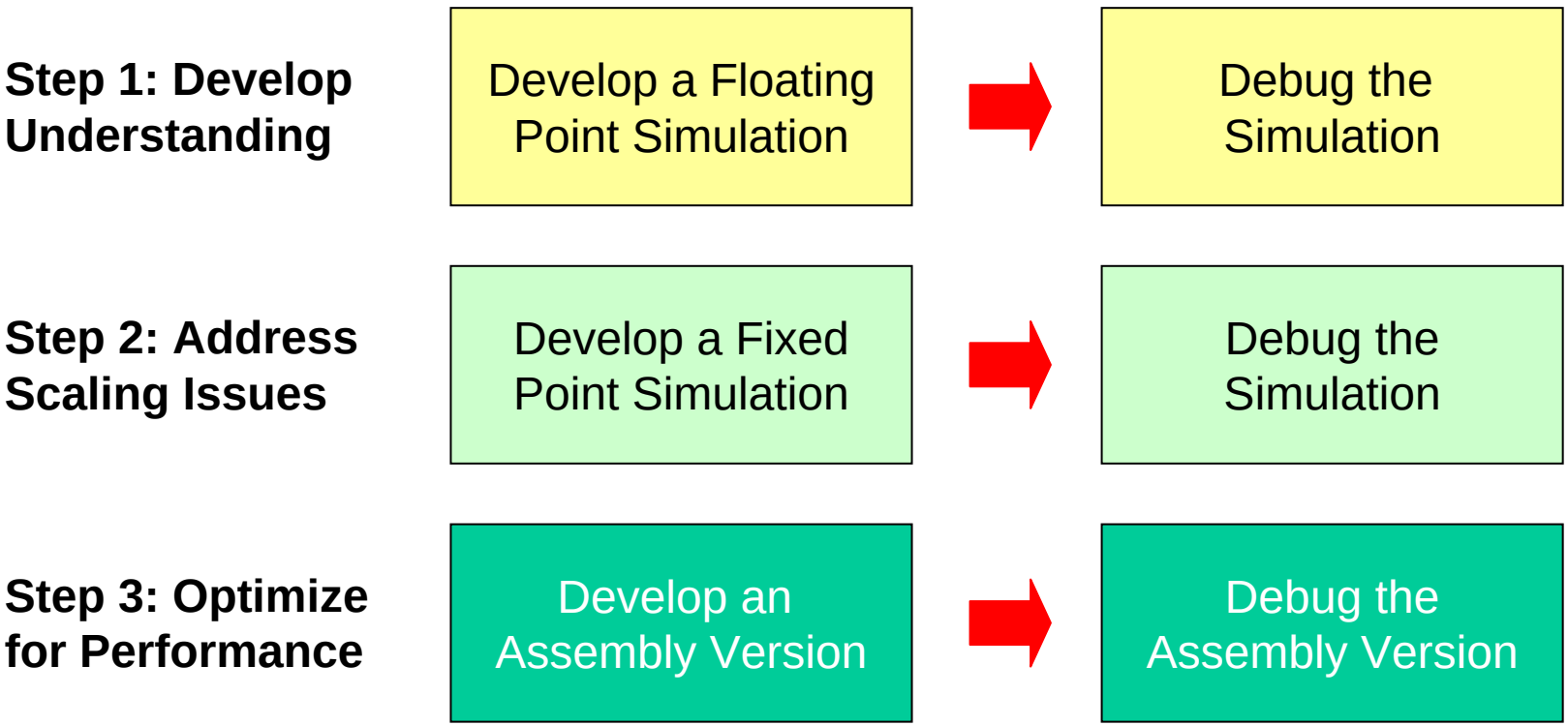
Texas Instruments

Outline

- DSP Software Development cycle
- Introduction to C Intrinsics
- Host Intrinsics implementation
- C64x IIR Example – build, run, test on DSP & host
- Conclusions

DSP Software Development Cycle

Common steps taken to develop DSP software



Issues with the 3 Step Approach

- Each step takes time and resources
- Algorithm testing at each stage
- Multiple versions of the algorithm – version control headaches
- Evaluation of processor instruction set compatibility and MIPS requirements often occurs late in the software development cycle
- Debugging algorithms on a pipelined and/or parallel processor can be very difficult (the problem is getting more difficult as processors get more complicated)

Can the Development Cycle be shortened ? Yes !

Improved Development Cycle

- Merge Steps 2 and 3

Step 1: Develop Understanding

Develop a Floating Point Simulation → Debug the Simulation

Step 2: Address Scaling Issues and Optimize for Performance

Simultaneously Develop a Fixed Point Simulation and an Assembly Version → Simultaneously Debug the Simulation and the Assembly

Question: How can these steps be combined ?

Introduction to C Intrinsics

Answer: C Intrinsics

What are C Intrinsics ?

- Special C function calls that the TI compiler recognizes and then maps directly into inline assembly instructions

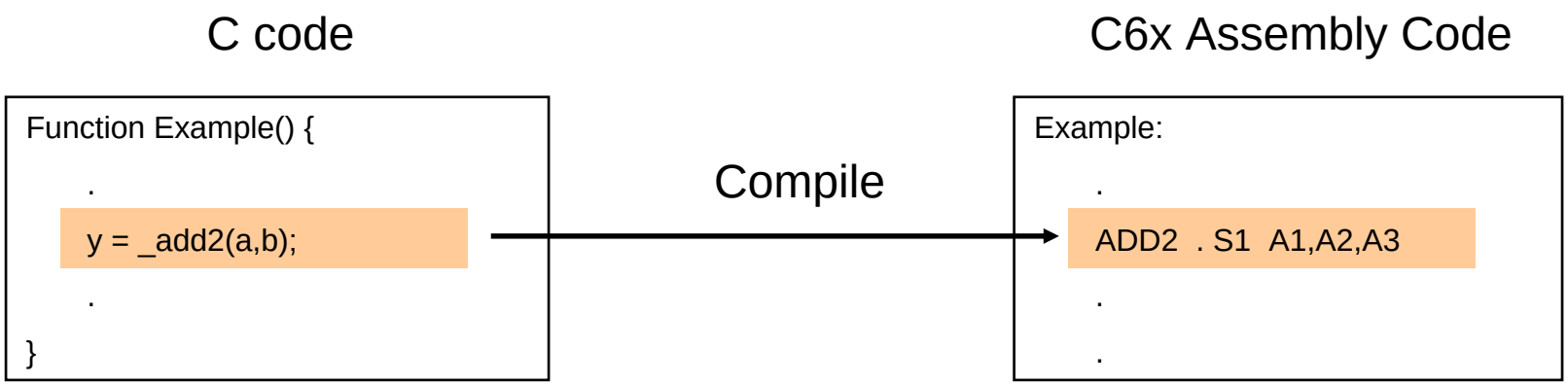
Example: ADD2 – adds the upper and lower 16-bit portions of a 32 bit register

Assembly instruction: ADD2 (.unit) src1,src2,dst

C Intrinsic: dst = _add2(src1,src2)

C Intrinsics Basics

- Example: C code with _add2() intrinsic and the resulting assembly code after compiling with TI compiler



Host Intrinsics

How can C Intrinsics be leveraged to shorten
The Software Development Cycle ?

- Create a Library with a function for each C Intrinsic. The function simulates the mathematical operations of the corresponding DSP assembly instruction.

This library is called Host Intrinsics

Host Intrinsics

- Example: ADD2

C code

```
example() {
        .
   y = _add2(a,b);
        .
}
```

```
// C6xSimulatorTypes.h
union reg32
{
  int32    x1;
  int32x2  x2;
  int32x4  x4;

  uint32   x1u;
  int32x2u x2u;
  int32x4u x4u;
};

// C6xSimulator.c
int32 _add2(int32 a,int32 b)
{
  union reg32 a32,b32,y32;

  a32.x1 = a;
  b32.x1 = b;

  y32.x2.lo = a32.x2.lo + b32.x2.lo;
  y32.x2.hi = a32.x2.hi + b32.x2.hi;

  return(y32.x1);
} /* end of _add2() function */
```

Host Intrinsics

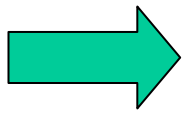
How many Intrinsics exist for each DSP family ?

TMS320C62x: 49

TMS320C64x: 107

TMS320C64x+ : 27

TMS320C67x: 6



Most algorithms written in assembly can now be expressed in C code with Intrinsic function calls

Note : above numbers reflect unique intrinsics per ISA. All c62 & c64 intrinsics can naturally be used on c64+ devices.

Using the Host Intrinsics

- Develop C/C++ code with C Intrinsic function calls.
- Compile the C/C++ code with the Host Intrinsics
- Debug and evaluate the performance of the algorithms in a C/C++ host programming environment
- Rely on TI tools to generate an optimized assembly version of the C/C++ code for the DSP

Benefit: One version of C/C++ code that runs in both the C/C++ host programming environment and the DSP

Using the Host Intrinsics

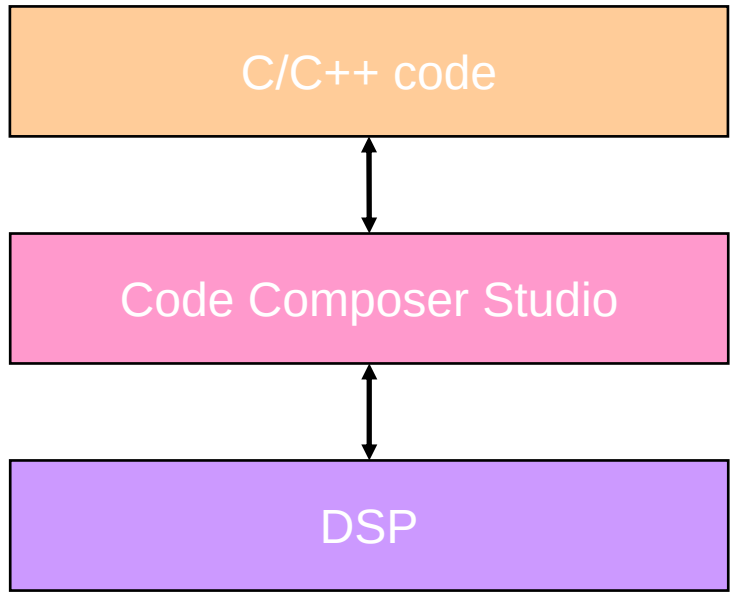
Given the Host Intrinsics, what C/C++ programming and host platforms can be used to develop and debug DSP code ?

- Many options exist
  - Third party IDEs (Microsoft Visual C/C++, Borland C/C++)
  - Matlab and Simulink by MathWorks
  - GNU tools
  - Others

Migrating C/C++ Code to the DSP

The steps required to execute the fixed point C/C++ code designed with Host Intrinsics

- Link the same C/C++ code into your CCS environment
- Compile the C/C++ code using TI tools
- Download the executable file into the DSP
- Run the code



- The compiler replaces the Intrinsic function calls with inline assembly instructions
- The level of optimization for the assembly instructions is set using compiler options

How Does it Work ?

- C/C++ code can directly access DSP assembly instructions without actually writing assembly code
- Eliminate headaches associated with assembly programming
  - Pipeline scheduling
  - Register allocation
  - Unit allocation
  - Stack manipulation
  - Parallel instruction debug

- Conclusion: Make the compiler do the hard work !

C6x Host Intrinsics - Implementation

- Core consists of three files
  - C6xSimulator.c - implementation
  - C6xSimulator.h - prototypes
  - C6xSimulatorTypes.h - typedefs
- Contains C functions for representing the numerical operations of DSP assembly instructions

- Examples / Test-suite
  - Unit_test : tests each intrinsic individually
    - C6xSimulator_test, C6xSimulator_main
  - System test(s) : exercises multiple intrinsics together in real functions
    - K_iir, K_scaling1to4, K_median3x3, K_filt_long, K_fdct_8x8, K_idct_8x8 : c64 (Kelvin) examples
    - J_cnv_dec : c64+ (Joule) example

Example – K_iir (c64 IIR filter) (1)

Walk-thru of K_iir

- Code
- Directory layout
- Build & run under GCC
- Build & run in CCS

```
//  Iterate over the biquads, two per iteration.
for (i = j = k = 0; i < nCoefs; i += 8, j += 2, k++)
{
    p0 = _dotp2(_hi(CoefAddr[j+0]), StateAddr_i[j+0]);
    p1 = _dotp2(_hi(CoefAddr[j+1]), StateAddr_i[j+1]);
    p2 = _dotp2(_lo(CoefAddr[j+0]), StateAddr_i[j+0]);
    p3 = _dotp2(_lo(CoefAddr[j+1]), StateAddr_i[j+1]);

    t0 = x + ( p0       >> 14);
    x  = x + ((p0 + p2) >> 14);
    t1 = x + ( p1       >> 14);
    x  = x + ((p1 + p3) >> 14);

    StateAddr_i[j+0] = _pack2(StateAddr_i[j+0], t0);
    newState_j_1     = _pack2(StateAddr_i[j+1], t1);
    if (i + 4 < nCoefs)
        StateAddr_i[j+1] = newState_j_1;
}
```

iir_i.c

Code

- Exercises multiple intrinsics
- iir_i.c : Intrinsics C version

Example – K_iir (c64 IIR filter) (2)

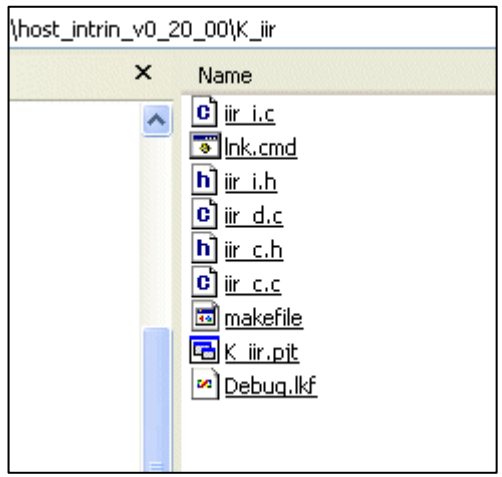
K_iir – Directory layout

- iir_i.[ch] : intrinsics C version
- Iir_c.[ch] : pure C version
- iir_d.c : test-harness comparing results of C & intrinsics-C versions

- lnk.cmd : TI DSP linker command file
- K_iir.pjt : CCS project file for TI DSP
- Debug.lkf : linker options, libraries for TI DSP CCS project

- makefile : host makefile for GCC build environment

This pattern followed by all examples



Example – K_iir (c64 IIR filter) (3)

K_iir – Build & run under gcc

- Build via 'make'
- Warnings explained in Caveats slide
- -D flags (see readme for more detail)
  - -DSIMULATION : required – brings in C intrinsics prototypes
  - -DLITTLE_ENDIAN_HOST or -DBIG_ENDIAN_HOST : required. Little endian for e.g. Linux, PC. Big-endian for Sparc.
  - -DTMS320C62X or -DTMS320C64X or -DTMS320C64PX : required : constrain to c62 or c64 or c64+ intrinsics

- All tests return fail = 0 or 1.
- 0 indicates success, 1 is failure. Enables scripting regression tests

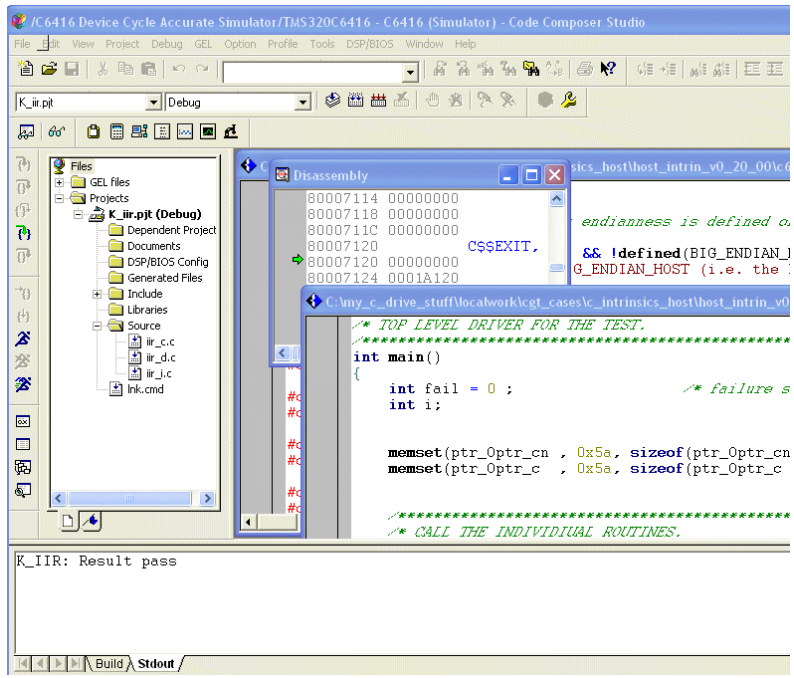
```
$ [K_iir > ] make
-------- Building debug application --------
gcc -I. -I../c6xsim -Wall -g -DSIMULATION -
DLITTLE_ENDIAN_HOST -DTMS320C64X -o d
bg/K_iir iir_i.c iir_c.c iir_d.c
../c6xsim/C6xSimulator.c \
-lc
iir_i.c:77: warning: ignoring #pragma CODE_SECTION
iir_c.c:36: warning: ignoring #pragma CODE_SECTION
iir_c.c:37: warning: ignoring #pragma CODE_SECTION
iir_d.c:32: warning: ignoring #pragma DATA_ALIGN
iir_d.c:33: warning: ignoring #pragma DATA_ALIGN
iir_d.c:34: warning: ignoring #pragma DATA_ALIGN
iir_d.c:36: warning: ignoring #pragma DATA_ALIGN
iir_d.c:37: warning: ignoring #pragma DATA_ALIGN
The build was successful


$ [K_iir > ] ./dbg/K_iir.exe
K_IIR: Result pass
```

Example – K_iir (c64 IIR filter) (4)

K_iir – Build & run in CCS

- Build via CCS project file (pjt)
- -D flags
  - none
- Same pass/fail as host (same code!)

- TI DSP version naturally uses native TI DSP intrinsics – not host intrinsics!



Caveats / Notes

1. Warnings building under gcc
   - These are expected. No DATA_ALIGN pragma on gcc [instead gcc supports `attribute (aligned)`]
   - Warnings left in place for user to resolve as desired e.g. ignore or #ifdef for your host

2. amemdN() memory intrinsics not fully modeled
   - E.g. _amemd8, _amemd8_const intrinsics tell the compiler to read eg an array of shorts with double–word accesses. This causes LDDW and STDW instructions to be issued for array accesses on TI C64x/c64x+ DSP.
   - No easy way to model this via an intrinsic on host side

3. Be careful with code using ‘double’, ‘long’ etc
   - Long is 40 bits on c6000, but 32 bits on most hosts
   - advice is to grep your code for 'long', 'double' etc and change to typedefs e.g. int40 from C6xSimulatorTypes.h

4. #include “C6xSimulator.h”
   – Include this header file in your code so you can use the typedefs to run on both TI C6000 DSP & your host platform. eg int64_d, int40 typedefs etc.
   – On TI C6000 DSP the only effect will be inclusion of the base typedefs.
   – On a host platform the intrinsic prototypes will be included etc

Conclusions

- Complete Host port of c62, 64, c64+ intrinsics
  - Enables run/prototype code on host (eg PC, Sparc...) where the debug environment is often richer (purify, faster I/O etc)
- Tested in wide variety of environments
  - cygwin with gcc3.3.3
  - Linux hosted on a PC
  - Sparc Solaris workstation (for tests that support big-endian)
  - MSVC 6.0
  - CCS (for native C6000 intrinsics usage)
- Many customers have 2 versions of algorithms : ‘golden C’ and optimized version. Golden C runs on host or DSP.
  - Now can have just 1 version : optimized C
  - Vastly reduced maintenance.