About this Module

This Module Describes the Basic Software Build Process of VisualDSP++, Specific Blackfin Programming "Gotchas", and Basic Code Optimization Strategies.

Users Will See an Example Demonstrating "Zero Effort" Optimization by Using Built-In Optimizer and Cache.

Agenda

- Software Build Process in VisualDSP++
- Explanation of Linker Description File (LDF)
- Programming Blackfin Processors In C
- Creating Efficient C Code
- Ogg Vorbis Tremor Example

Software Build Process in VisualDSP++

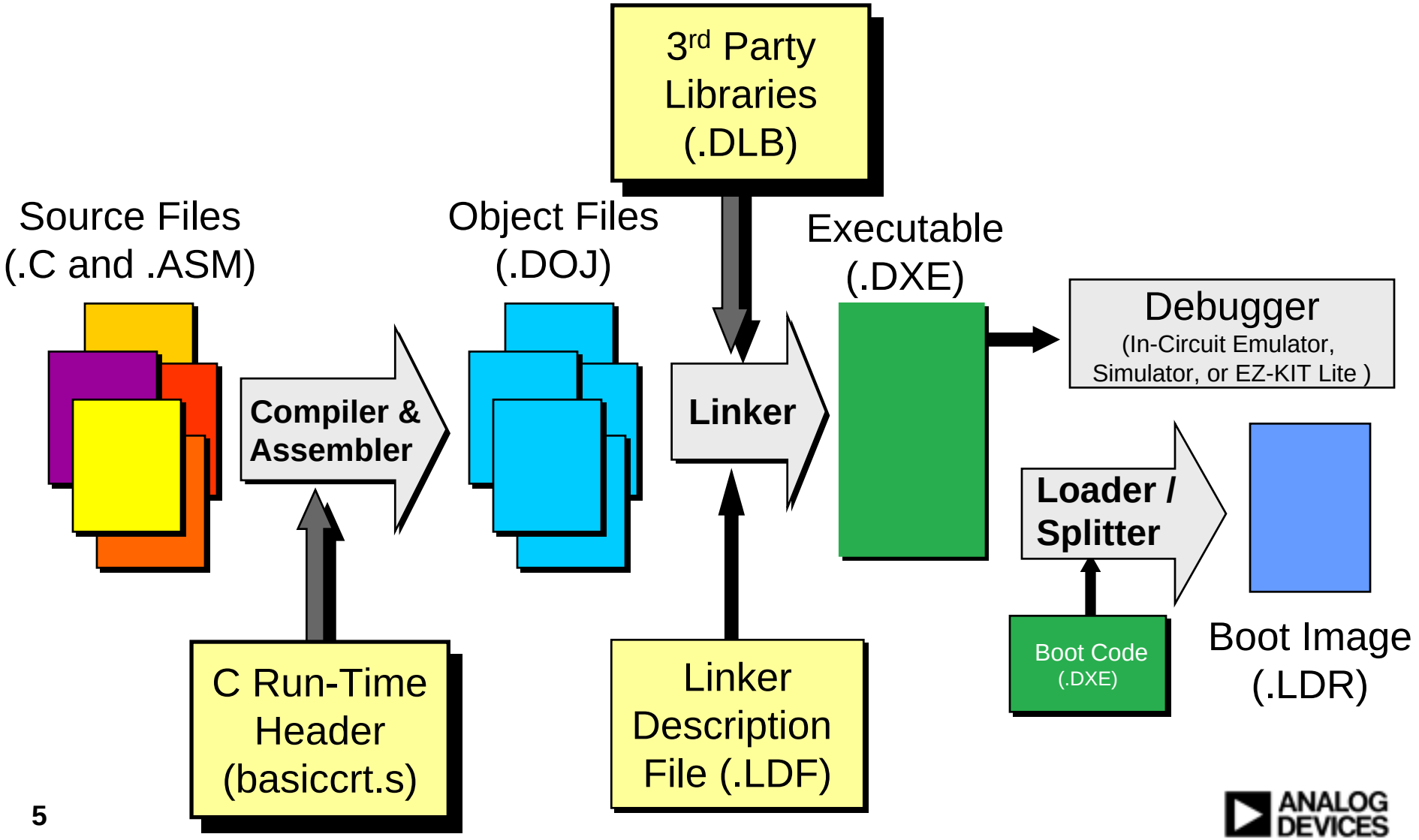
Porting C Code to Blackfin

Ported C Code Will Build and Execute Out-of-Box

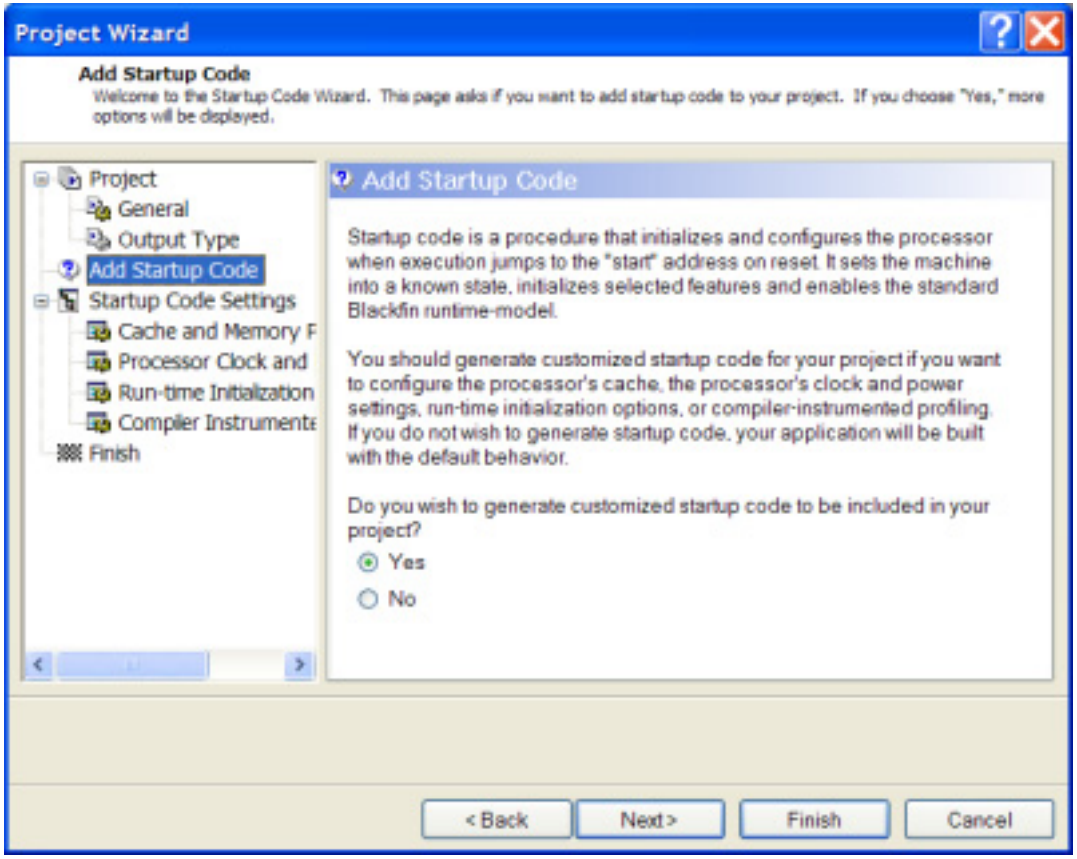
- **Code Can Be Large and Slow**
  - If Code Is Too Large to Fit in Internal Memory…
    - It Spills Into External Memory
    - Fetching from External Memory Is Slower
  - Optimization Switches Are Off
    - Generates Unoptimized Functional Code
    - Includes Debug Information
- **Default Clock Settings Used**
- **Cache Is Off**

Software Development Flow

What Files Are Involved?



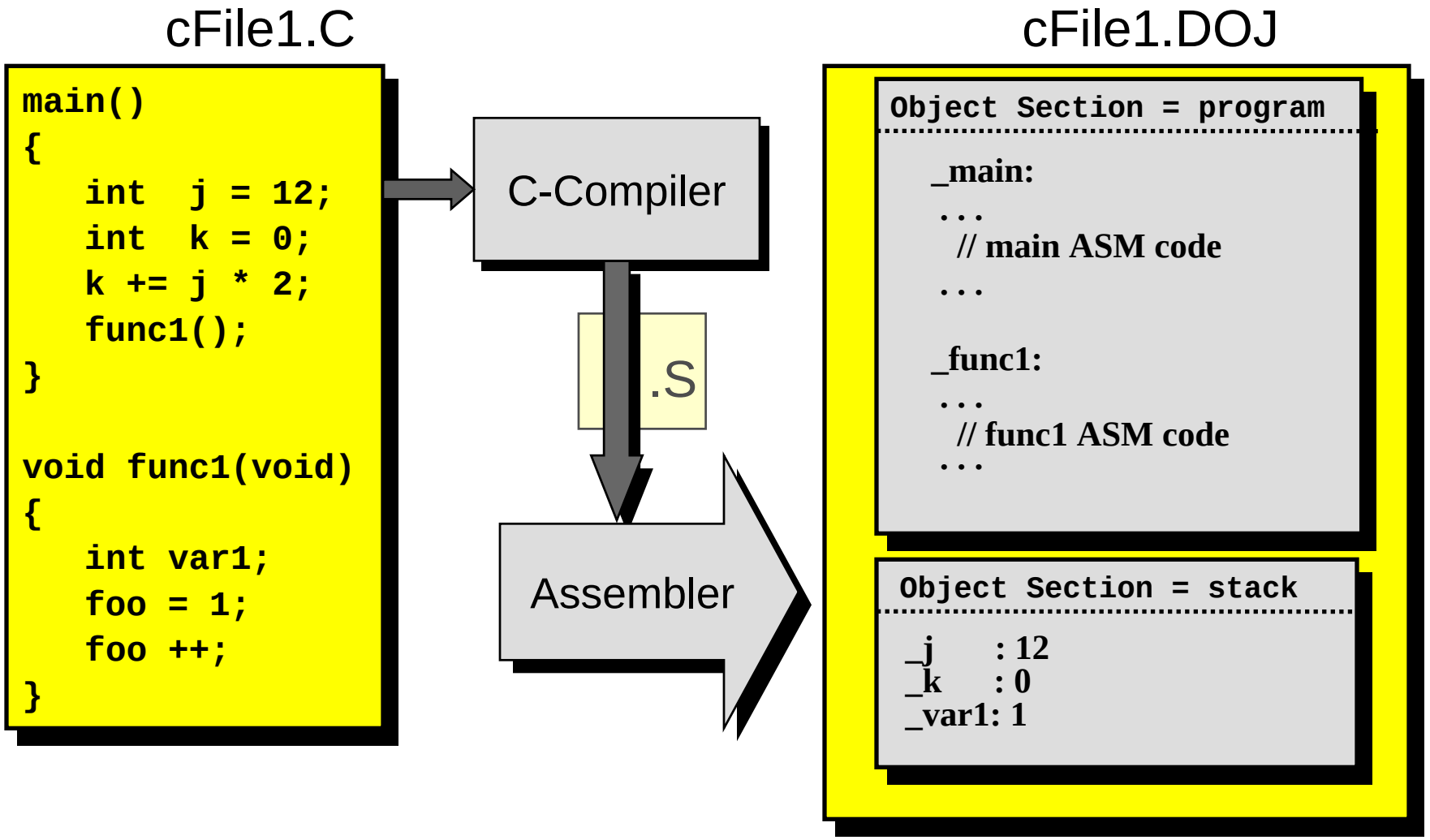
C Run-Time Header (basiccrt.s)



- **Sets Up C Run-Time (CRT) Environment**
  - Sets Up Stack Pointer
  - Enables Cycle Counters for Benchmarking Code
  - Configures Cache, If Requested
  - Change Clock/Voltage Settings, If Requested
- **Can Be Modified Through Project Options Window**

Software Build Process

Step-1 Example: C Source



Compiler-Generated Object Section Names

- **Compiler Uses Default Section Names that Will Be Used By the Linker**

Name	Contents
program	Program Instructions
data1	Global and “static” Data
constdata	Data Declared as “const”
ctor	C++ Constructor Initializations
cplb_code	Instruction Cache Config Tables
cplb_data	Data Cache Config Tables

C Source with Alternate Sections

foo.C

```
section (“sdram0”) int array[256];

section (“L1_code”) void bar(void)
{
   int foovar;
   foovar = 1;
   foovar ++;
}
```

C-Compiler

Assembler

foo.DOJ

```
Object Section = sdram0
.....
_array [00]
_array [01]
  …
_array [255]
```

```
Object Section = L1_code
.....
 _bar :
  . . .
    // bar ASM code
```

```
Object Section = stack
.....
 _foovar: 1
```

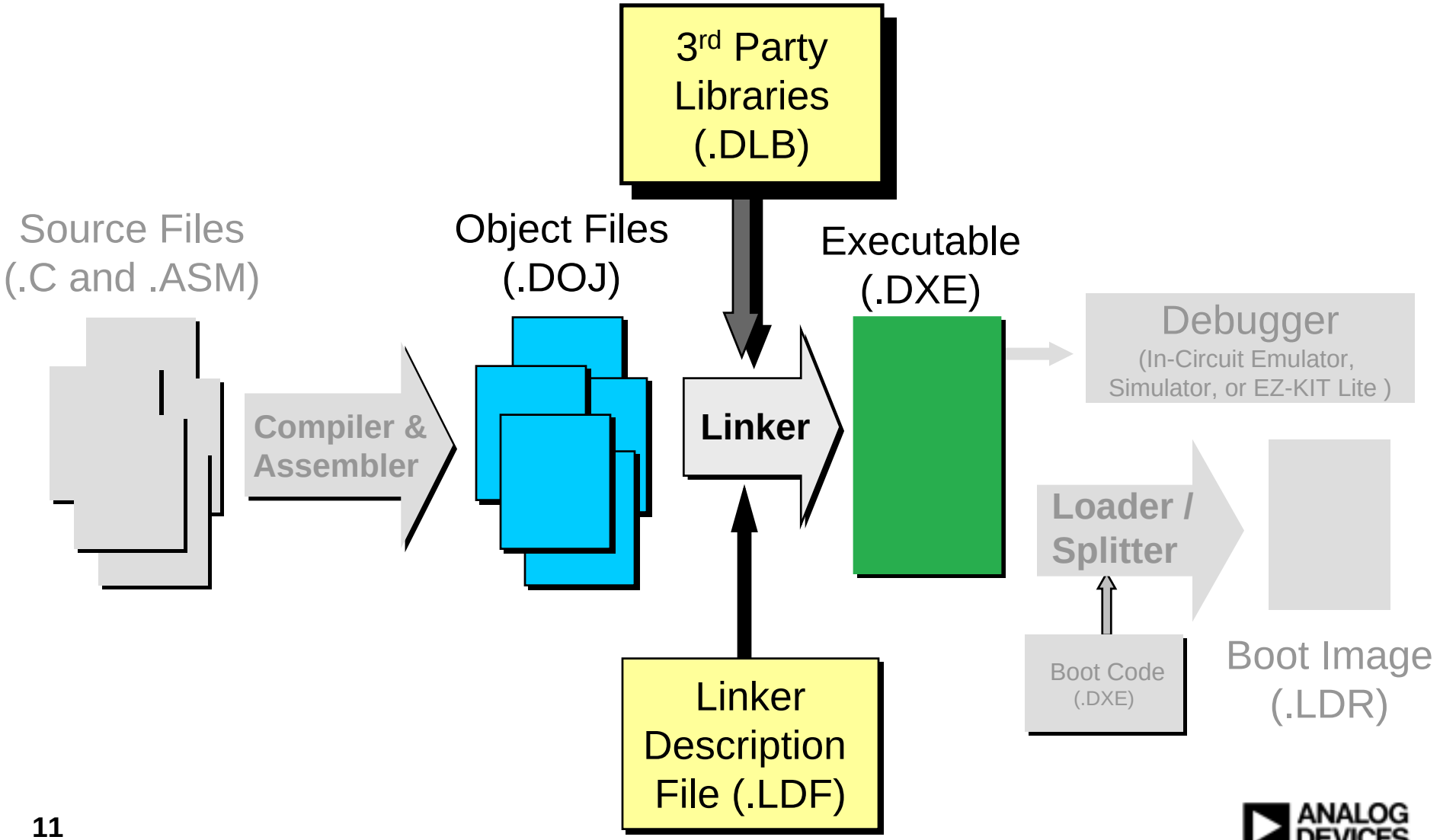
Note: The section( ) directive is used to place data or code into a section other than the default section used by the compiler.

Other Convenient Input Section Names

- **sdram0**
  - **Code/Data to be Explicitly Placed in External Memory**
- **L1_code**
  - **Code to be Mapped Directly into On-Chip Program Memory**
- **L1_data_a; L1_data_b**
  - **Data to be Mapped Directly into On-Chip Data Memory (Banks A and B, Respectively)**

Software Development Flow

Step 2 - Linking



VisualDSP++ Linker

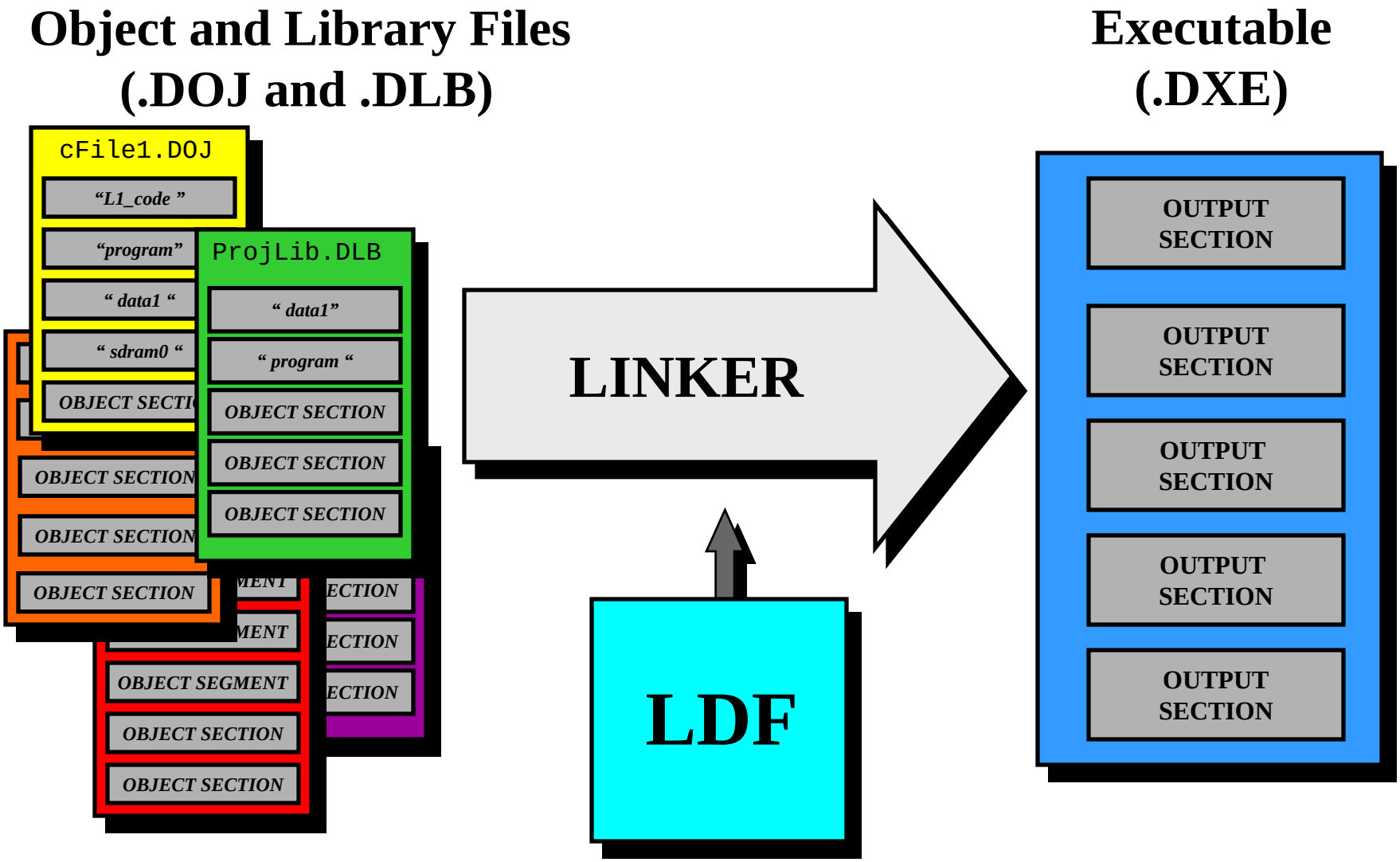
- **Inputs**
  - **Compiled/Assembled Code Object Files (DOJ)**
  - **Linker Description File (LDF)**
  - **Compiled Libraries (3rd Party or Other)**
- **Outputs A Fully Resolved Blackfin Executable File (DXE)**
  - **Can Be Run In Simulator**
  - **Can Be Loaded Onto Target Via Emulator or Debug Monitor**
- **LDF Defines the Hardware System for VisualDSP++**
  - **Specifies Processor Being Used**
  - **Specifies All Available Internal/External System Memory**
    - If It's Not Specified, the Linker Cannot Use It!!
    - User Assigns Code/Data to Available Memory

Blackfin Memory

- **Blackfin Processors Have Highly Efficient Memory Architecture**
- **Memory Supported by A Hierarchical Memory Scheme**
  - **On-Chip Level 1 Memory – L1**
  - **On-Chip Level 2 Memory – L2 (BF535 and BF561)**
  - **Off-Chip Memory – SDRAM/SRAM**
- **Each Type of Memory Has Its Own Address Range**
- **Memory Accesses Automated by Processor**

Explanation of Linker Description File (LDF)

Link Process



LDF Expresses How Program Is Mapped (1)

- **Linker Description File (LDF) Is the Way the User Describes Where in Memory to Place Parts of a Program**
  - **Every Project *MUST* Include an LDF**
  - **Default LDF Chosen By Linker if None Specified**
- **User Sets the Target:**

  `ARCHITECTURE()`
- **User Defines the Memory Layout:**

  `MEMORY{}`

Setting Target & Defining Memory Layout

- **User Must Declare All Available System Memory**
- **Refer To Blackfin System Memory Map When Defining Segments**

```
ARCHITECTURE (ADSP-BF537)
SEARCH_DIR ($ADI_DSP\Blackfin\lib)
$OBJECTS = $COMMAND_LINE_OBJECTS;

MEMORY
{
    MEM_L1_DATA_A     { TYPE(RAM) START(0xFF800000)  END(0xFF803FFF) WIDTH(8) }
    MEM_L1_DATA_B     { TYPE(RAM) START(0xFF902000)  END(0xFF907FFF) WIDTH(8) }
    MEM_L1_CODE       { TYPE(RAM) START(0xFFA00000) END(0xFFA0BFFF) WIDTH(8) }
    MEM_SDRAM0        { TYPE(RAM) START(0x00000004) END(0x1FFFFFFF) WIDTH(8) }
}
```

Global Commands

Memory Segment (User-Assigned Name)

Start Address (User-Specified)

End Address (User-Specified)

LDF Expresses How Program Is Mapped (2)

- **User Describes Where in Memory to Place Input Sections:**

```
PROCESSOR{
  SECTIONS{
        … INPUT_SECTIONS()…
  }
}
```

  - **INPUT_SECTIONS() Describes What Contents from Input Files Are to be Mapped into the Current Output Section**

```
INPUT_SECTIONS{ file_source ( input_labels ) }
```

    - file_source - List of Files or LDF Macro that Expands into a File List
    - input_labels - List of Section Names
    - Example:

```
INPUT_SECTIONS{ main.doj (program) }
```

      Maps All Code in “program” Section from main.doj File

Example LDF (Continued)

Link Commands – Heart of LDF

```
PROCESSOR p0
{
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
    SECTIONS
    {
        sec_data_a
        { INPUT_SECTIONS( $OBJECTS(data1) ) } > MEM_L1_DATA_A

        sec_data_b
        { INPUT_SECTIONS( $OBJECTS(L1_data_b) ) } > MEM_L1_DATA_B

        sec_prog
        { INPUT_SECTIONS( $OBJECTS(program) ) } > MEM_L1_CODE

        sec_sdram
        { INPUT_SECTIONS( $OBJECTS(data1) )
          INPUT_SECTIONS( $OBJECTS(program) ) } > MEM_SDRAM0
    }
}
```

OBJECT SECTIONS from .ASM/.S files

DXE SECTION NAMES Used in .map file

MEMORY SEGMENTS Declared in the LDF

Using Macros

- **$OBJECTS = CRT, $COMMAND_LINE_OBJECTS**
  - **CRT Is the C Run-Time Object File basiccrt.doj**
  - **$COMMAND_LINE_OBJECTS**
    - Contains All Object (.doj) Files that Are Included in Your Project
    - Ordering of DOJ Files Is Exactly the Order in Which Source Files Appear in Project

```
sec_prog
{
        INPUT_SECTIONS( $OBJECTS(program) )
} > MEM_L1_CODE
```

How Does The Linker Work?

- **Object Sections Can Be Mapped to Many Memory Segments**
- **SECTIONS Parsed Left-to-Right and Top-to-Bottom**
  - **If Object Section Fits, It Goes Into the Memory Segment**
  - **If It Doesn't Fit, Linker Proceeds to…**
    - Put It Off To Side Until Next Executable Section Is Processed
    - Check Next Item(s) On Line For Object Sections That WILL Fit
    - Check Next Line(s) For Object Sections That WILL Fit
  - **When All Executable Sections Have Been Processed, A Linker Error Is Generated If Any Object Sections Have Gone Unmapped**

Example LDF (Elaborated)

Input Files: main.c and file2.c

```
sec_data_a
{ INPUT_SECTIONS( $OBJECTS(data1) ) } > MEM_L1_DATA_A
sec_prog
{ INPUT_SECTIONS( $OBJECTS(program) ) } > MEM_L1_CODE
sec_sdram
{ INPUT_SECTIONS( $OBJECTS(data1) )
  INPUT_SECTIONS( $OBJECTS(program) ) } > MEM_SDRAM0
```

Expand Macro:

```
sec_data_a
{ INPUT_SECTIONS( main.doj(data1)   file2.doj(data1) ) }     > MEM_L1_DATA_A
sec_prog
{ INPUT_SECTIONS( main.doj(program)  file2.doj(program) ) } > MEM_L1_CODE
sec_sdram
{ INPUT_SECTIONS( main.doj(data1)   file2.doj(data1) )
  INPUT_SECTIONS( main.doj(program)  file2.doj(program) ) } > MEM_SDRAM0
```

Optimizing Using the Linker/LDF

- **For Size, Use “Eliminate Unused Objects” Feature**
- **For Performance**
  - **Move Frequently Used Program/Data into Internal Memory**

    section(“L1_code”) void myFunction (void);
    section(“L1_data_a”) char myArray[256];
  - **Place Data into Different Banks to Avoid Access Conflicts**

    section(“L1_data_a”) short vector_mult_1[256];
    section(“L1_data_b”) short vector_mult_2[256];
  - **Create Special Section Name and Map It Directly**

    sec_prog
    { INPUT_SECTIONS(main.doj(Map1st)) } > MEM_L1_CODE
  - **Create Special Macro List to Prioritize Input Files**

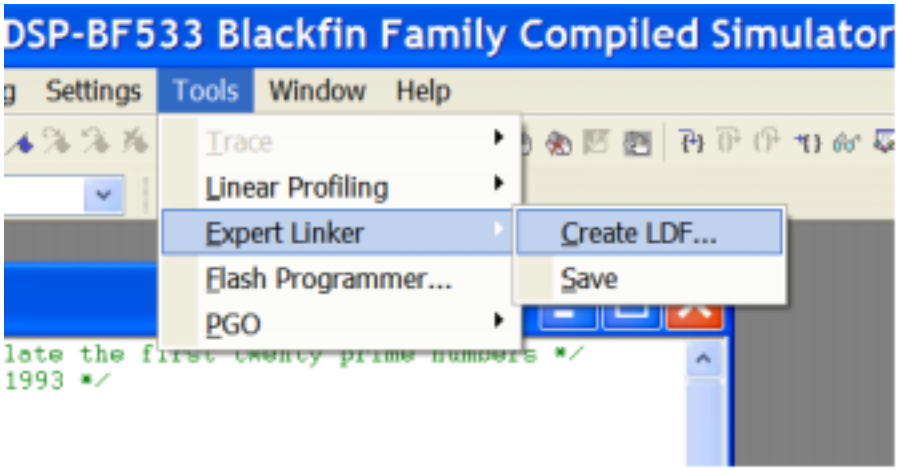
    $MY_L1_OBJECTS = file3.doj, file5.doj, file106.doj

Expert Linker

Expert Linker

- **Presents Graphical Interface for Manipulating LDF Files**
- **Resize Memory by Grab/Stretch**
  - **Enlarge One Memory Segment at Expense of Another**
- **Map Sections with Drag/Drop**
- **Displays Run-Time Information**
  - **Watermarks for Stack/Heap**
- **Simply Changing a Text File; Makes Changes Locally, Not Globally**
  - **Changes Made with Text Editor Will Be Reflected in Expert Linker**
  - **Changes Made with Expert Linker Will Show Up in Text Editor**

Create an LDF with Expert Linker



- **Proceed Through Dialog, Selecting:**
  - **LDF Name**
  - **Language (C, C++, or ASM)**
- **Template LDF Files Are in ldf Installation Directory**

  C:\Program Files\VisualDSP++ 4.0\Blackfin\ldf
  - **Named with _ASM, _C, _CPP Suffixes**

    i.e., ADSP-BF537_C.ldf

Programming Blackfin Processors In C

Making Use of Peripherals

- **Blackfin Peripherals Are Programmed/Used Via Sets of Memory-Mapped Registers (MMRs)**
- **System Services and Device Drivers Automatically Handle Programming of These MMRs**

Blackfin Header Files (defBF537.h)

```
#include <def_LPBlackfin.h>                    /* Include all Core registers and bit definitions      */
#include <defBF534.h>                          /* Include all MMR and bit defines common to BF534     */

/* Define EMAC Section Unique to BF536/BF537                                                          */
/* 10/100 Ethernet Controller                  (0xFFC03000 - 0xFFC031FF)                              */
#define     EMAC_STAADD          0xFFC03014          /* Station Management Address                    */

/* EMAC_STAADD Masks                                                                                  */
#define     STABUSY              0x00000001  /* Initiate Station Mgt Reg Access / STA Busy Stat       */
#define     STAOP                0x00000002  /* Station Management Operation Code (Write/Read*)       */
#define     STADISPRE            0x00000004  /* Disable Preamble Generation                           */
#define     STAIE                0x00000008  /* Station Mgt. Transfer Done Interrupt Enable           */
#define     REGAD                0x000007C0  /* STA Register Address                                  */
#define     PHYAD                0x0000F800  /* PHY Device Address                                    */

#define     SET_REGAD(x)         (((x)&0x1F)<<  6 )      /* Set STA Register Address                  */
#define     SET_PHYAD(x)         (((x)&0x1F)<< 11 )      /* Set PHY Device Address                    */
```

- Associates MMR Names to Their Physical Addresses in Memory
- Defines All Bits in MMRs by Name to Be Used in More Legible Code
- Sets Up Useful Macros to Perform Common Tasks
- Located in C:\Program Files\VisualDSP++ 4.0\Blackfin\include

C Header Files (cdefBF537.h)

```
#include <cdefBF534.h>          /* Include MMRs Common to BF534                    */
#include <defBF537.h>           /* Include all Core registers and bit definitions  */

/* Include Macro "Defines" For EMAC (Unique to BF536/BF537)                        */
/* 10/100 Ethernet Controller      (0xFFC03000 - 0xFFC031FF)                       */

#define pEMAC_STAADD            ((volatile unsigned long  *)EMAC_STAADD)
```

```
#include <cdefBF537.h>
main ( )
{
   *pEMAC_STAADD |= SET_PHYAD(0x15);     /* Set PHY Device Address to 0x15 */
}
```

- The *cdef* Headers Define Convenient Macros for MMR Access
- Using *p Pointer Notation, Each MMR Will Be Properly Accessed
- System Services and Device Drivers Care for This Automatically

One Special Gotcha (cdefBF537.h)

```
/* SPORT0 Controller                (0xFFC00800 - 0xFFC008FF)       */
#define pSPORT0_TX                  ((volatile unsigned long  *)SPORT0_TX)
#define pSPORT0_RX                  ((volatile unsigned long  *)SPORT0_RX)
#define pSPORT0_TX32                ((volatile unsigned long  *)SPORT0_TX)
#define pSPORT0_RX32                ((volatile unsigned long  *)SPORT0_RX)
#define pSPORT0_TX16                ((volatile unsigned short *)SPORT0_TX)
#define pSPORT0_RX16                ((volatile unsigned short *)SPORT0_RX)

/* SPORT1 Controller                (0xFFC00900 - 0xFFC009FF)       */
#define pSPORT1_TX                  ((volatile unsigned long  *)SPORT1_TX)
#define pSPORT1_RX                  ((volatile unsigned long  *)SPORT1_RX)
#define pSPORT1_TX32                ((volatile unsigned long  *)SPORT1_TX)
#define pSPORT1_RX32                ((volatile unsigned long  *)SPORT1_RX)
#define pSPORT1_TX16                ((volatile unsigned short *)SPORT1_TX)
#define pSPORT1_RX16                ((volatile unsigned short *)SPORT1_RX)
```

- **SPORTx Data Registers Are the Only MMRs that Can Be Both 16- and 32-bit, Depending on Hardware Configuration**
- **Using the Wrong Macro Results in Non-Functional SPORT Code**
- **Again, Device Drivers Care for This Automatically**

Special Case Code

```
//-------------------------------------------------------------------------------------------- //
// Function:            Init_SPORT                                                            //
//                                                                                            //
// Description:         This function initializes SPORT0 for 16-bit data                      //
//-------------------------------------------------------------------------------------------- //

#include<cdefBF537.h>

void Init_SPORT(void)
{
        *pSPORT0_TCR2     = SLEN(15);          // Configure SPORT0 for 16-bit data
        *pSPORT0_TX16     = 0xAA55;            // Use 16-bit Access Macro to Write Register
        *pSPORT0_TCR1     |= TSPEN;            // Enable SPORT0 Transmitter

        while(1)
        {
            while(!(*pSPORT0_STAT & TXHRE));   // wait for hold register to empty

            *pSPORT0_TX32   = 0xAA55;          // INCORRECT: Using 32-bit Access Macro
            *pSPORT0_TX     = 0xAA55;          // INCORRECT: Using 32-bit Access Macro

            *pSPORT0_TX16   = 0xAA55;          // CORRECT: Using 16-bit Access Macro
        }
}
```

Creating Efficient C Code for Blackfin

Optimizing C Code

- **Optimization Can Decrease Code Size or Lead to Faster Execution**
  - **Controlled Globally by Optimization Switch or Compiler Tab of Project Options**

no switch	optimization disabled
-O	optimization for speed enabled
-Os	optimization for size enabled
-Ov num	enable speed vs size optimization (sliding scale)

  - **Controlled Dynamically In C Source Code Using Pragmas**

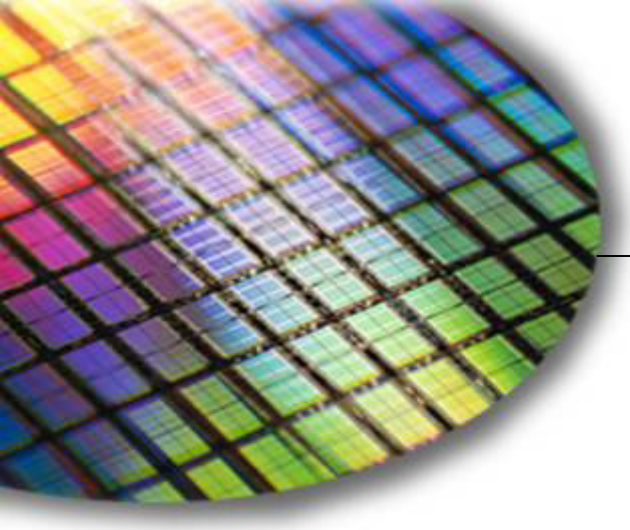
#pragma optimize_off	- Disables Optimizer
#pragma optimize_for_space	- Decreases Code Size
#pragma optimize_for_speed	- Increases Performance
#pragma optimize_as_cmd_line	- Restore optimization per command line options

    Many More Pragmas Available, These Are the Most Commonly Used

  - **Controlled Locally On Per-File Basis**

General Optimization Guidelines

- **Native Data Types Are Optimal**
- **Try to Avoid Division**
- **Take Advantage of Memory Architecture**
  - **Internal vs External**
  - **Cache**
  - **DMA**
- **C-Libraries Are Already Optimized!!**



The World Leader in High Performance Signal Processing Solutions

Ogg Vorbis Tremor Example

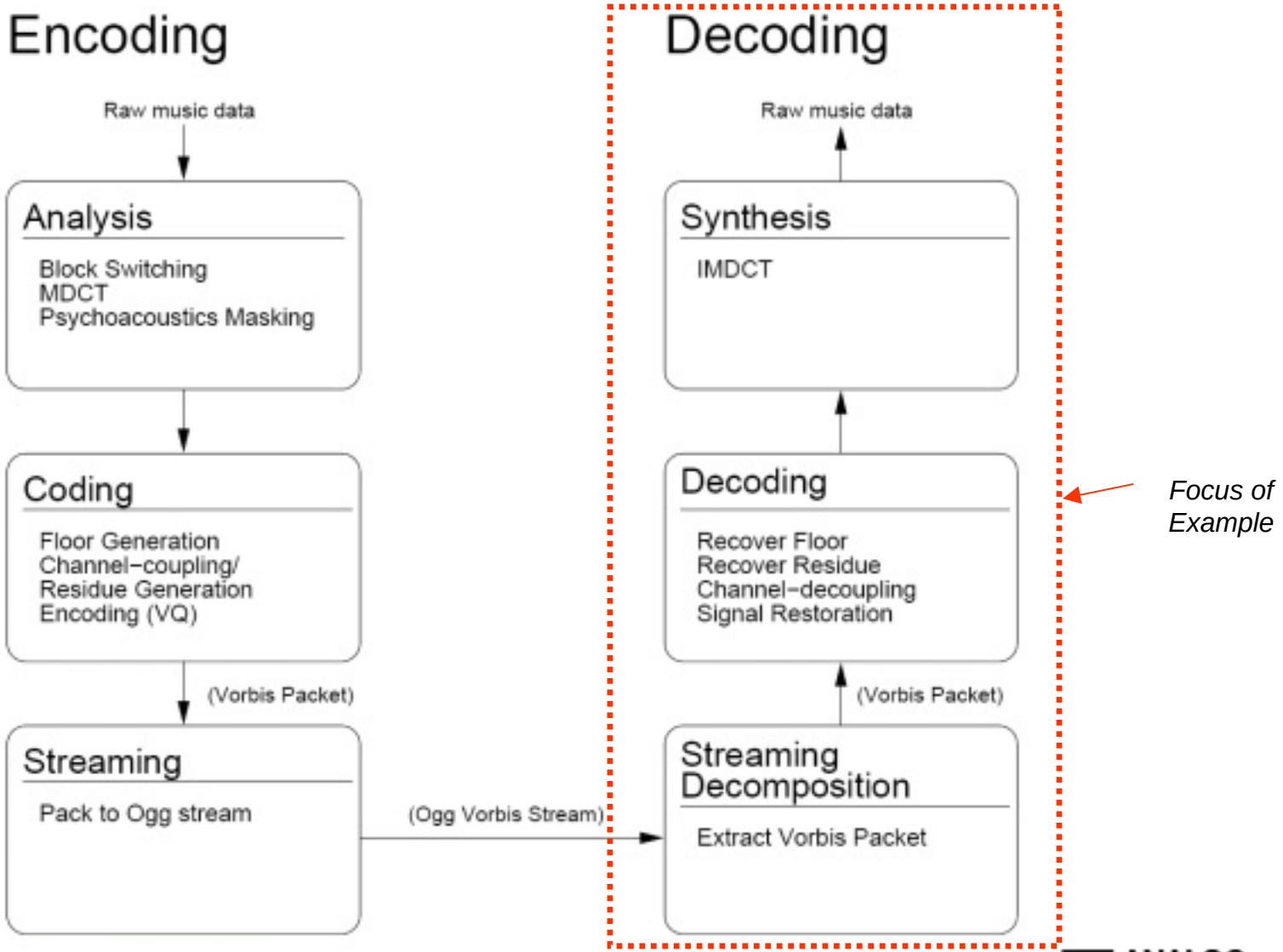
ANALOG DEVICES

What Is Ogg Vorbis Tremor?

- **Flash-Based Ogg Vorbis Decoder with an Audio DAC Interface**
- ***Ogg Vorbis*[1] is an Audio Compression Format**
  - **Fully Open**
  - **Patent- and Royalty-Free**
- ***Tremor* Is A Free Ogg Vorbis Decoder Implementation**
  - **Fixed-Point (Integer Arithmetic Only)**

[1] www.xiph.org

Ogg Vorbis Overview



source: http://oggonachip.sourceforge.net

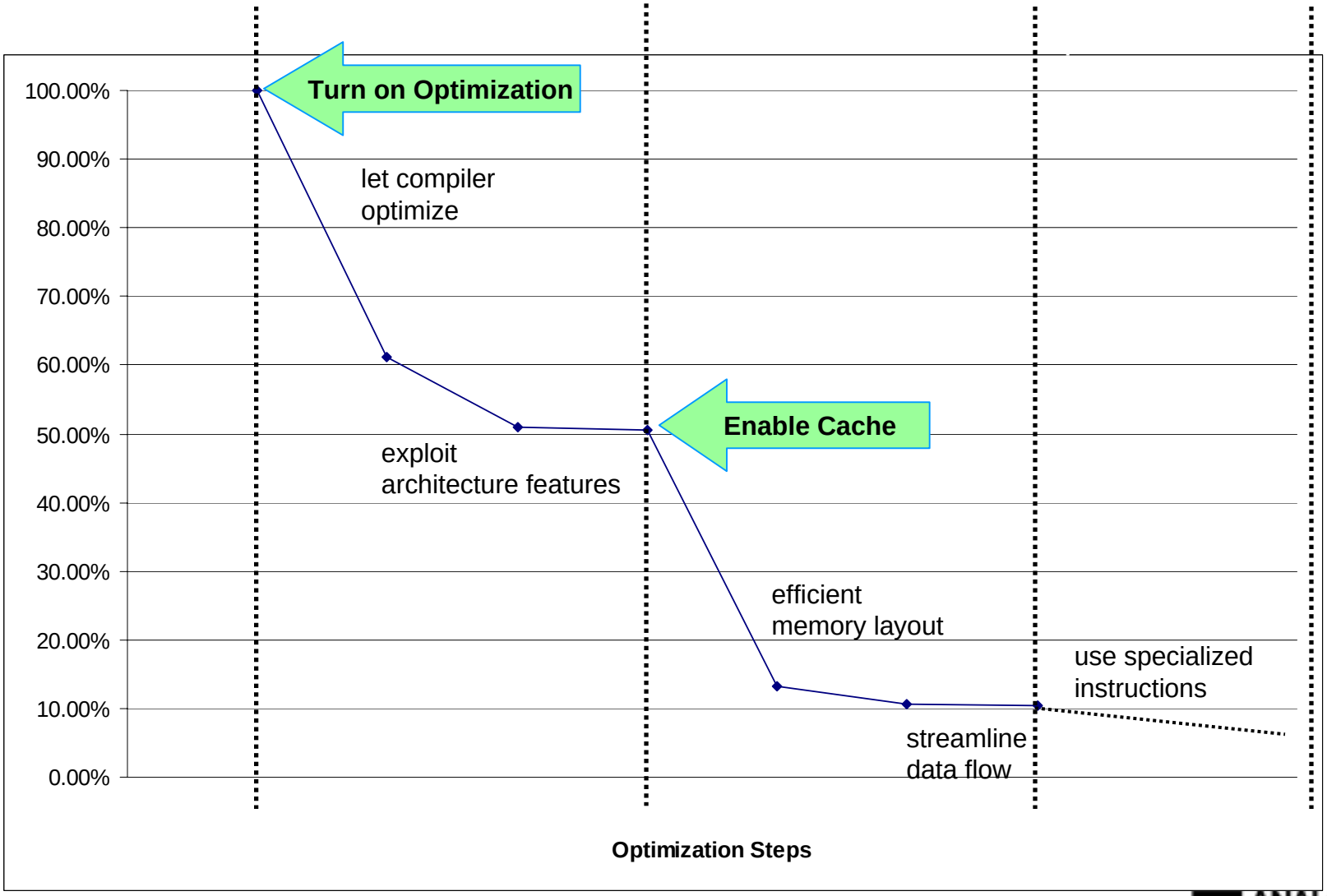
Simple Steps To Increase Efficiency

- **Optimizer Enables Compiler to Generate More Efficient Code**
  - **Performs Basic C Optimization Analysis**
  - **Produces ASM Code that Takes Advantage of Blackfin Architecture**
    - Multi-Issue Instructions
    - Hardware Loops
    - Pipeline Considerations
- **Instruction Cache Considerably Increases Performance**

Compiler Optimization Strategies

- **Reference Code Based On Host File I/O**
  - Compiles Out-of-the-Box
  - Replace File I/O With Flash I/O
- **Benchmark On Simulator and/or BF537 EZ-KIT Lite**
- **Computation Cycles vs. Data Access Cycles**
  - Decrease Computation Cycles Now
  - Optimize Memory Later
- **Profile for Hotspots**
  - 80/20 Rule
  - Loops Important
- **Global Compiler Settings**
  - Optimize for Speed
  - Automatic Inlining

Optimizing for Execution Speed



ANALOG DEVICES

Conclusion

- **Walked Through Software Development Process**
- **Described How To Use The Linker to Help With Optimization**
- **Demonstrated Simple 2-Step Process for Significant Performance Increase**

For Additional Information

- **For VisualDSP++ :**
  http://www.analog.com/processors/resources/crosscore
  or Analog Devices : http://www.analog.com

- **Embedding Ogg Vorbis: http://oggonachip.sourceforge.net**
  or Ogg Vorbis Itself : http://www.xiph.com

Or Click the “Ask A Question” Button