Andes Programming Guide

Driving Innovations™

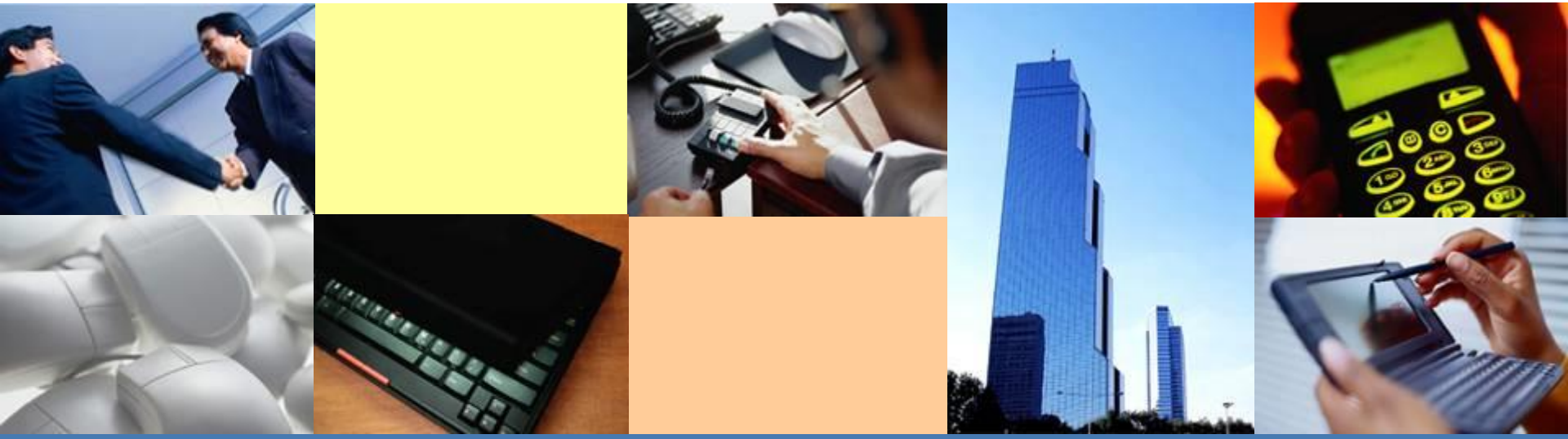
Overview

- Andes Toolchains
- NDS32 Assembly Language
- Pseudo-instructions
- Application Binary Interface (ABI)
- Andes Intrinsic Function Programming
- Inline Assembly Programming
- Advanced Programming Optimization

Documents for Reference

- Andes Programming Guide
- AndeStar™ ISA (Instruction Set Architecture)
- AndeStar™ SPA (System Privilege Architecture)
- These documents can be found in AndeSight™ or BSP.
  Folder: C:\Andestech\AndeSight212MCU\doc
  or C:\Andestech\AndeSight212STD\doc

Andes Toolchains



Toolchains

- Andes toolchains are built from GNU, including gcc, as, and ld.
- Latest toolchains are in
  - BSP v4.1.0 (Windows and Linux version)
  - AndeSight v2.1.2 STD/MCU (IDE)
- Andes library support includes glibc, uClibc, Newlib and MCUlib.
  - For non-OS/RTOS application – MCUlib and Newlib
  - For Linux application – glibc and uClibc
- Document:
  - 《Andes_Programming_Guide_for_ISA_V3_PG010_V1.2》

Andes Toolchains

- 32/16-bit mixed-length instructions
- Andes defines three versions of baseline instruction set – v1, v2, and v3
- V3m (N7, N8) is subset of v3 and is reduced registers
- For HW configuration related of V3
  - V3j is for reduced registers (16 GPRs)
  - V3f is for double precision FPU
  - V3s is for single precision FPU

Andes Toolchains

- Naming format:
  - EX: nds32le-elf-mculib-v3
  - nds32[endian]-[os]-[lib]-[baseline ver.]
    - [endian] – le | be.
    - [os] – elf | linux.
    - [lib] –
      - For elf – mculib | newlib
      - For linux – glibc | uclibc
    - [baseline ver.] – v1, v2, v3, v3m
      - v[1|2|3][m|j|f|s] – EX: v3m, v3j, v3f, ….

Andes Toolchains

- MCUlib (based on Newlib) is a library with Andes optimization enhancement for MCU applications and small code size.

- N705, N801, E801, and S801 – V3m
  - nds32le-elf-mculib-v3m

- N968A – V3
  - nds32le-elf-mculib-v3
  - nds32le-elf-mculib-v3j (16 GPR)

Andes Toolchains

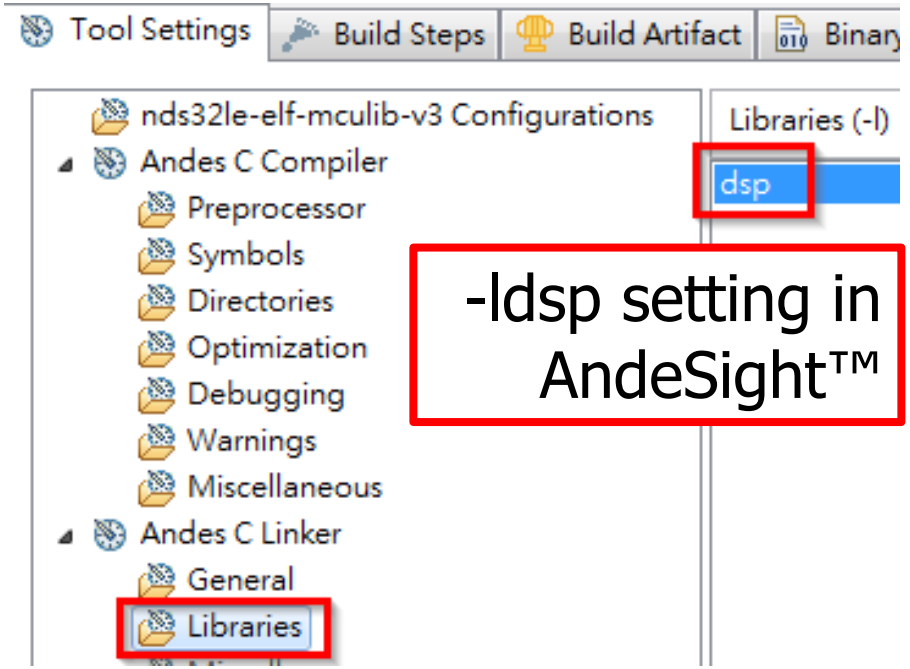
- N1068A – V3
  - nds32le-elf-mculib-v3
  - nds32le-elf-newlib-v3s
  - nds32le-elf-mculib-v3j (16 GPR)
  - nds32le-linux-glibc-v3 (Linux)
  - nds32le-linux-uclibc-v3 (Linux)
- N1337 – V3
  - nds32le-elf-mculib-v3
  - nds32le-linux-glibc-v3 (Linux)
  - nds32le-linux-uclibc-v3 (Linux)

Andes DSP Library (1)

- Basic math: vector mathematics
- Fast math: sin, cos, atan, atan2, sqrt, etc.
- Complex math
- Statistics: max, min, RMS, etc.
- Filtering: IIR, FIR, LMS, etc.
- Transforms: FFT, DCT/DCT4
- Matrix functions
- PID controller, Clark and Park transforms
- MISC: copy/fill arrays, data type conversions, etc.

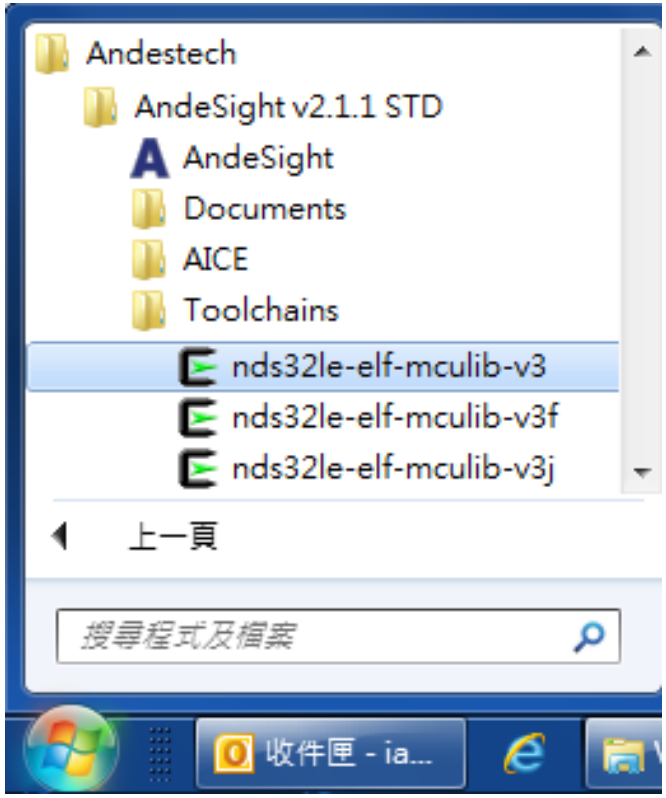
Andes DSP Library (2)

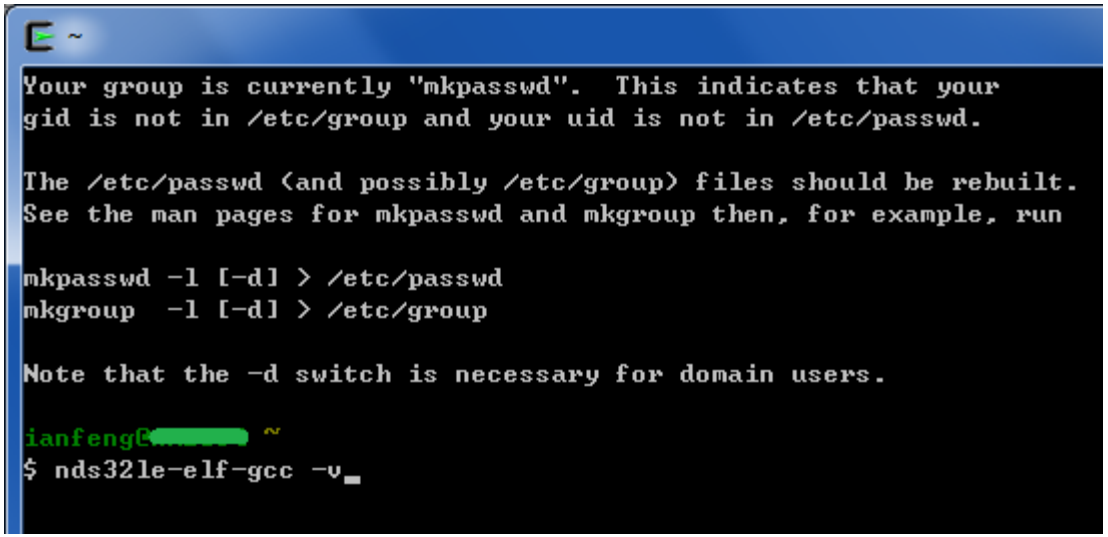
- Linking with -ldsp (in the nds32le-elf/lib/libdsp.a of the toolchains)
- Including header files provided by Andes DSP library. They are named according to function categories:
  - nds32_basic_math.h
  - nds32_complex_math.h
  - nds32_controller_math.h
  - nds32_filtering_math.h
  - nds32_matrix_math.h
  - nds32_statistics_math.h
  - nds32_transform_math.h
  - nds32_utils_math.h



Start a Toolchain Console

- Start → All Programs → Andestech → AndeSight v2.1.2 STD/MCU → Toolchains
- Running "Cygwin".

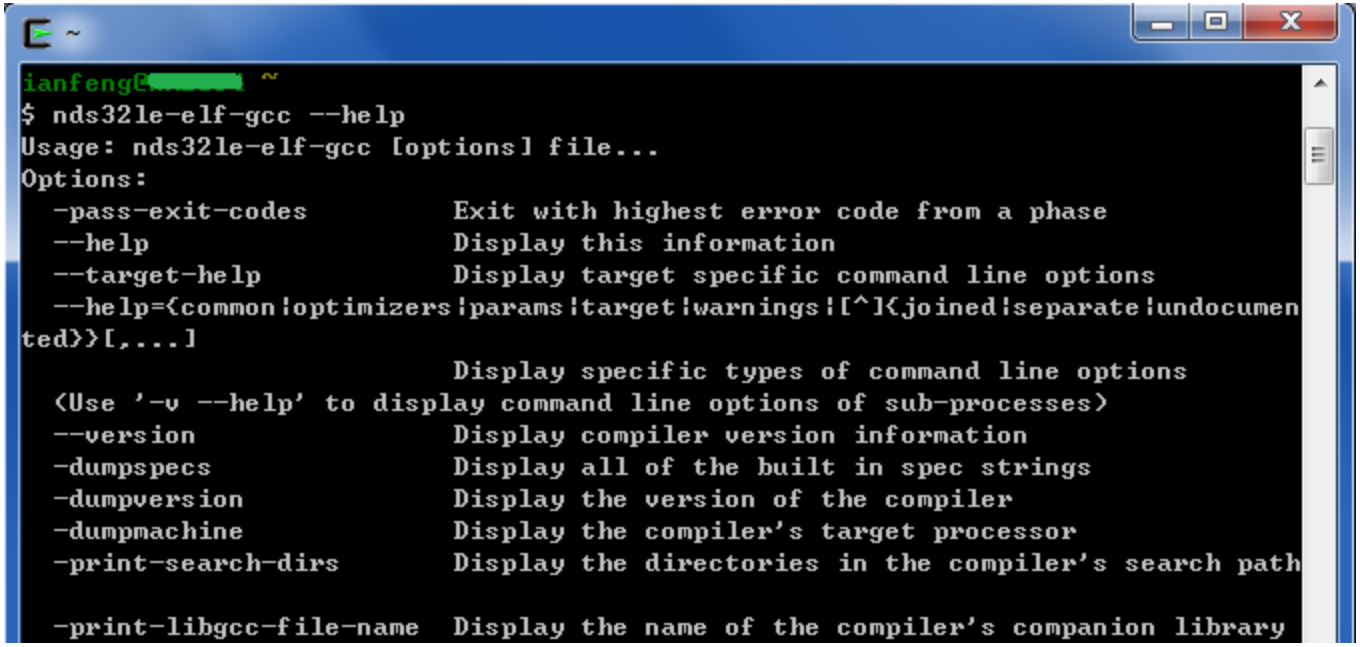




Compiler Options

- Get a list of all supported options, use command:

  mypc> nds32le-elf-gcc --help



- For target specific options, enter:
  - mypc> nds32le-elf-gcc --target-help

Binutils

- nds32le-elf-gcc
  - Could do compile, assemble, link…
- nds32le-elf-as
  - Assembler
- nds32le-elf-ld
  - Link object files
- nds32le-elf-ar
  - For creating, modifying and extracting from archives

Binutils (cont)

- nds32le-elf-nm
  - List symbols from object files
- nds32le-elf-size
  - List the section size of an object or archive file
- nds32le-elf-readelf
  - Display information from any ELF format object file
- nds32le-elf-objdump
  - Display code information in object files
- nds32le-elf-objcopy
  - Copy and translates object files

Toolchains Default Definitions

- To check if a feature is enabled as default, issue the following command:
  - nds32le-elf-gcc -E -dM - < /dev/null | grep NDS32

```
$ nds32le-elf-gcc -E -dM - < /dev/null |grep NDS32
#define NDS32_EXT_PERF2 1
#define NDS32_ABI_2 1
#define __NDS32__ 1
#define __NDS32_BASELINE_V3__ 1
#define __NDS32_EXT_STRING__ 1
#define __NDS32_BASELINE_V3 1
#define __NDS32_GP_DIRECT__ 1
#define NDS32_EL 1
#define __NDS32_EL 1
#define __NDS32_EXT_PERF2__ 1
#define __NDS32_ABI_2__ 1
#define __NDS32_16_BIT__ 1
#define __NDS32_ABI_2 1
#define __NDS32_EXT_STRING 1
#define __NDS32_EXT_EX9__ 1
#define NDS32_BASELINE_V3 1
#define __NDS32_EXT_IFC__ 1
#define __NDS32_EXT_PERF__ 1
#define __NDS32_EXT_PERF 1
#define __NDS32_ISR_VECTOR_SIZE_4__ 1
#define NDS32_EXT_EX9 1
#define NDS32_EXT_IFC 1
#define __NDS32_EXT_PERF2 1
```

  - Useful to determine which toolchains are used.

Nds32-elf-gcc Options Introduction

- --help : Display gcc help information
- --target-help
  - Display target specific options
- -g
  - Produce debugging information
  - The default level is 2
  - -g3 includes macro definitions
- -S
  - Compile only; do not assemble or link
- -mvh
  - Enable Virtual Hosting support.

Nds32-elf-ld Options Introduction

- -nostartfiles
  - Do not use the standard system startup files when linking.

Sections

- GNU Default Pseudo-ops Supporting Sections:

.text	code and constants
.data	Initialized data
.bss	un-initialized data

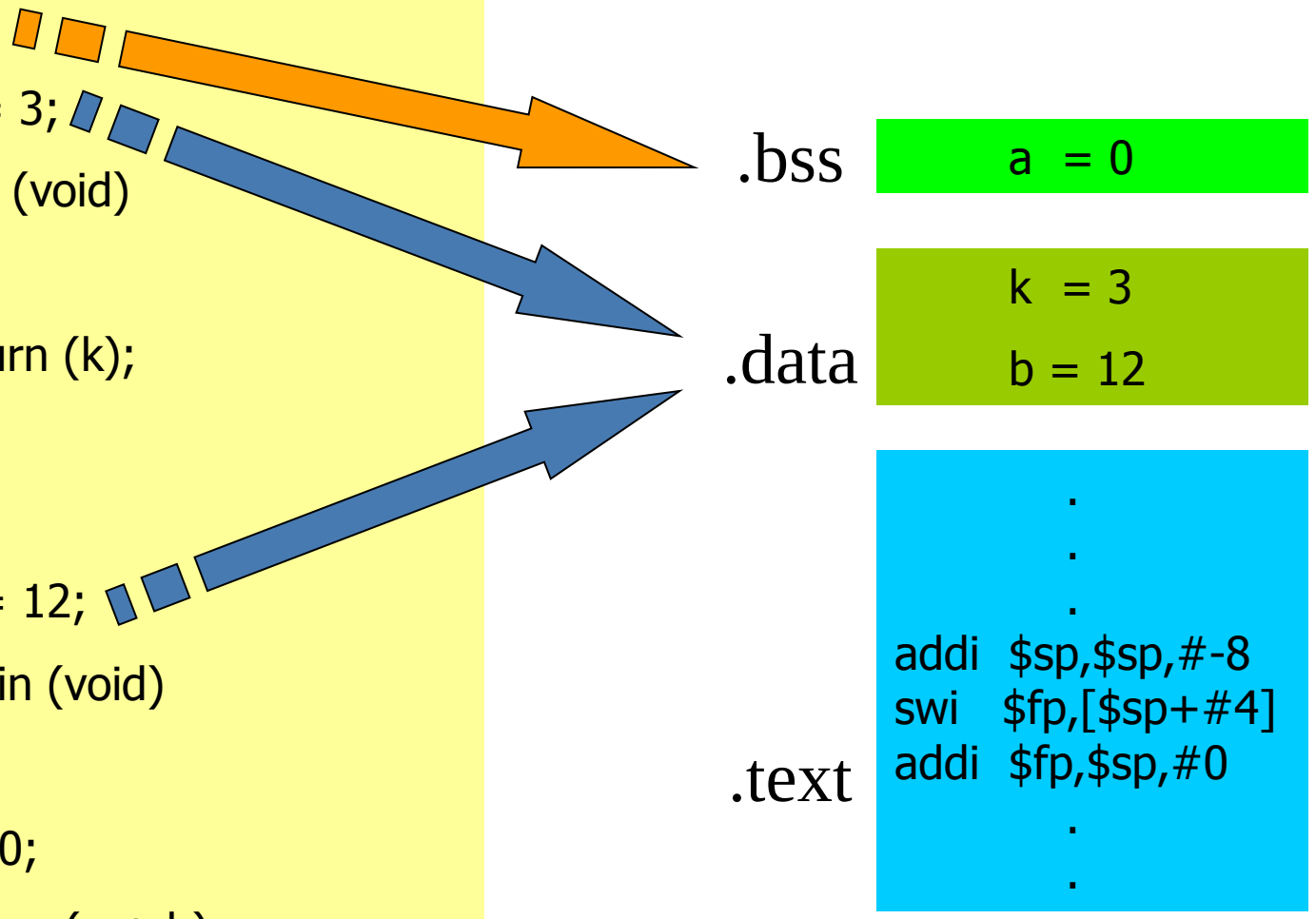
- Andes Pseudo-ops Supporting Sections:

.sdata_d	for double-word sized (8-byte) small data items.
.sdata_w	for word sized (4-byte) small data items.
.sdata_h	for half-word sized (2-byte) small data items.
.sdata_b	for byte sized small data items.
.sbss_d	for double-word sized (8-byte) small data items.
.sbss_w	for word sized (4-byte) small data items.
.sbss_h	for half-word sized (2-byte) small data items.
.sbss_b	for byte sized small data items.

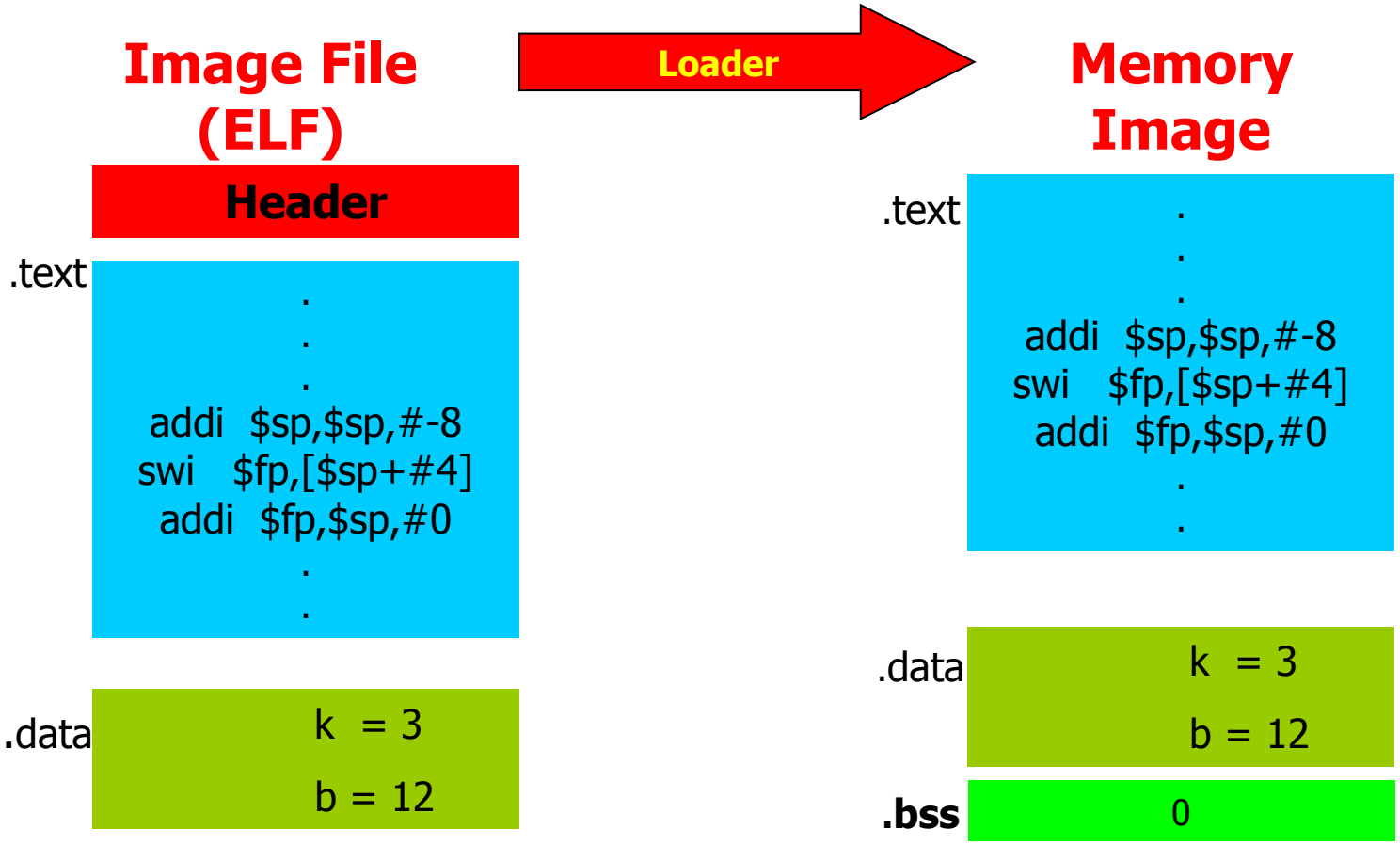
Linker Analysis (1)

```
int a;
int k = 3;
int foo (void)
{
    return (k);
}

int b = 12;
int main (void)
{
    a =0;
    return (a + b);
}
```

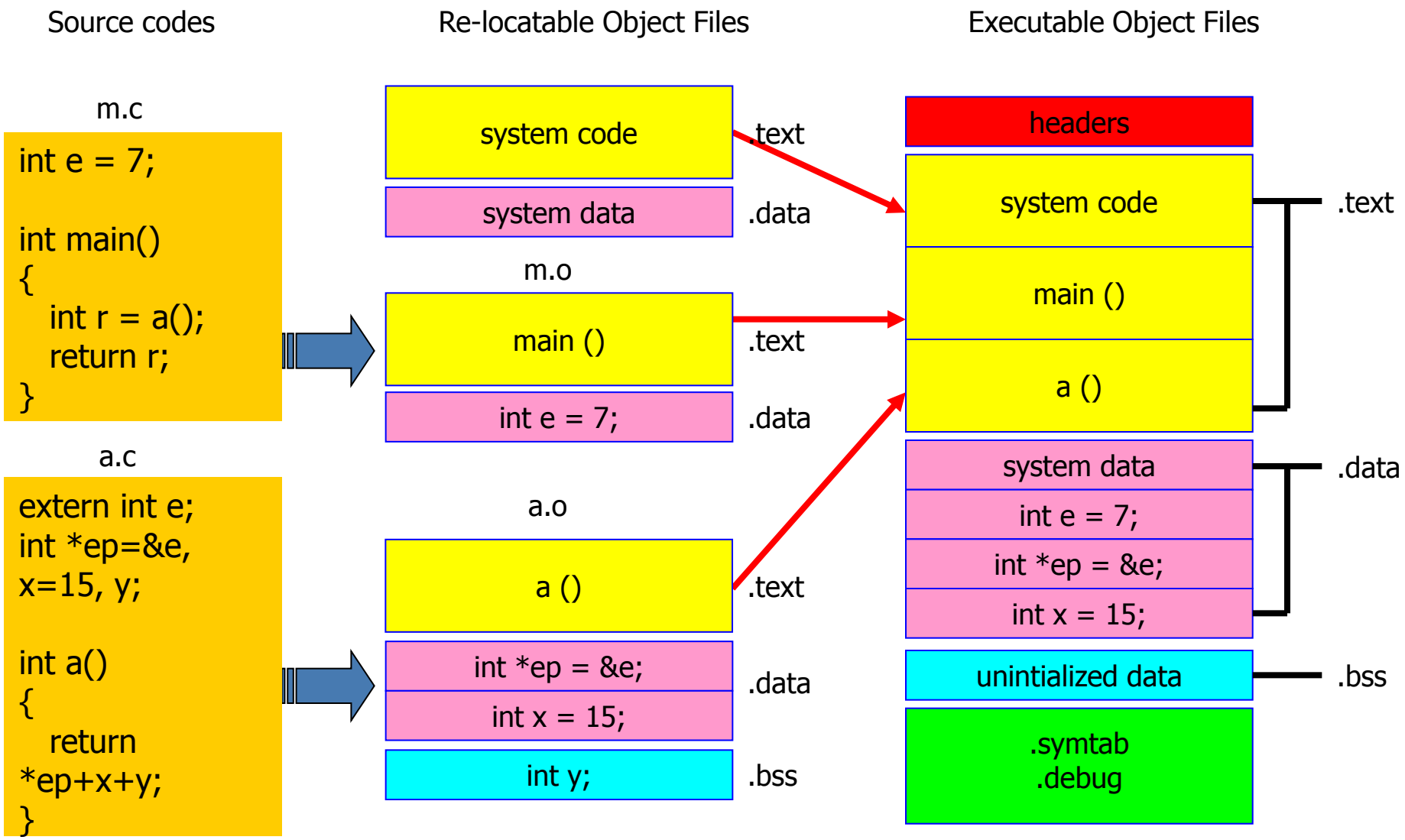


Linker Analysis (2)

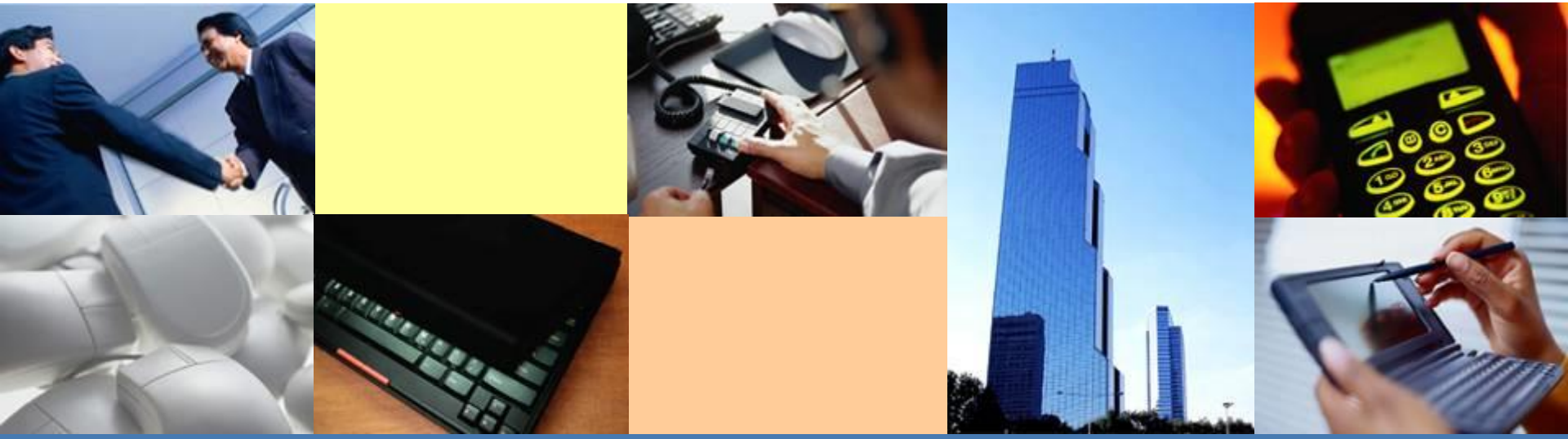


- Image file has header. It has specific meaning for "Loader".
- Memory image has extra ".bss" section.

Linker Analysis (3)



NDS32 Assembly Language



General Syntax

- Comment : '#' at column 1 and '!' anywhere in the line except inside quotes.
- Multiple instructions per line are allowed though not recommended and should be separated by ";".
- An integer can use specified in decimal, octal (prefixed with 0), hexadecimal (prefixed with 0x), and binary (prefixed with 0b).
  - For example, 128, #128, 0200, #0200, 0x80, #0x80, 0b10000000, and #0b10000000 are all identical.
  - The leading '#' is optional.
- Assembler is not case-sensitive in general except user defined label.
  - For example, "jral F1" is different from "jral f1" while it is the same as "JRAL F1".

GPR (General Purpose Register)

Table 1. AndeStar General Purpose Registers

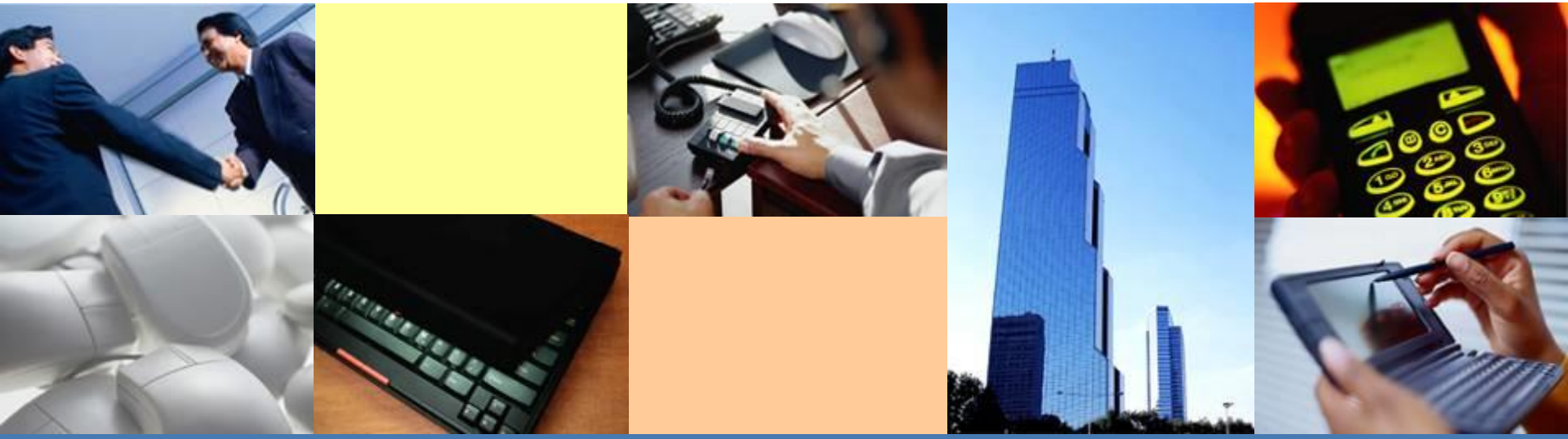
Register	32/16-bit (5)	16-bit (4)	16-bit (3)	Comments
r0	a0	h0	o0	
r1	a1	h1	o1	
r2	a2	h2	o2	
r3	a3	h3	o3	
r4	a4	h4	o4	
r5	a5	h5	o5	Implied register for beqs38 and bnes38
r6	s0	h6	o6	Saved by callee
r7	s1	h7	o7	Saved by callee
r8	s2	h8		Saved by callee
r9	s3	h9		Saved by callee
r10	s4	h10		Saved by callee
r11	s5	h11		Saved by callee
r12	s6			Saved by callee
r13	s7			Saved by callee
r14	s8			Saved by callee
r15	ta			Temporary register for assembler Implied register for slt(s|i)45, b[eq|ne]zs8
r16	t0	h12		Saved by caller
r17	t1	h13		Saved by caller
r18	t2	h14		Saved by caller
r19	t3	h15		Saved by caller
r20	t4			Saved by caller
r21	t5			Saved by caller
r22	t6			Saved by caller
r23	t7			Saved by caller
r24	t8			Saved by caller
r25	t9			Saved by caller
r26	p0			Reserved for Privileged-mode use.
r27	p1			Reserved for Privileged-mode use
r28	s9/fp			Frame pointer / Saved by callee
r29	gp			Global pointer
r30	lp			Link pointer
r31	sp			Stack pointer

GPR of Reduced Register

Table– 4 Registers for Reduced Register Configuration

Register	32/16-bit (5)	16-bit (4)	16-bit (3)	Comments
r0	a0	h0	00	
r1	a1	h1	01	
r2	a2	h2	02	
r3	a3	h3	03	
r4	a4	h4	04	
r5	a5	h5	05	Implied register for beqs38 and bnes38
r6	s0	h6	06	Saved by callee
r7	s1	h7	07	Saved by callee
r8	s2	h8		Saved by callee
r9	s3	h9		Saved by callee
r10	s4	h10		Saved by callee
r15	ta			Temporary register for assembler Implied register for slt(s|i)45, b[eq|ne]zs8
r28	fp			Frame pointer / Saved by callee
r29	gp			Global pointer
r30	lp			Link pointer
r31	sp			Stack pointer

Pseudo-instructions



Pseudo-instructions

- There are many software instructions defined to make assembly programming much easier. These are pseudo-instructions.

- load 32-bit value/address
  - li rt5,imm_32
    - loads 32-bit integer into register rt5.
    - sethi rt5,hi20(imm_32) and then
      ori rt5,rt5,lo12(imm_32)
  - **la** rt5,var
    - loads 32-bit **address** of var into register rt5.
    - sethi rt5,hi20(var) and then
      ori rt5,rt5,lo12(var)

Pseudo Instructions (cont)

- load/store variables
  - l.{bhw} rt5,var
    - loads **value** of var into register rt5.
    - sethi $ta,hi20(var) and then
      l{bhw}i rt5,[$ta+lo12(var)]
  - s.{bhw} rt5, var
    - stores register rt5 to var.
    - sethi $ta,hi20(var) and then
      s{bhw}i rt5,[$ta+lo12(var)]

Pseudo Instructions (cont)

- negation
  - not rt5,ra5 — alias of nor rt5,ra5,ra5
  - neg rt5,ra5 — alias of subri rt5,ra5,0

- branch to label/branch and link to function name
  - br rb5 — alias of jr rb5
  - b label — branch to label
  - bral rb5 — alias of jral rb5
  - bal fname — It is translated into "jal fname" or "la $ta,fname; bral $ta"
  - call fname — call function name, same as "jal fname"

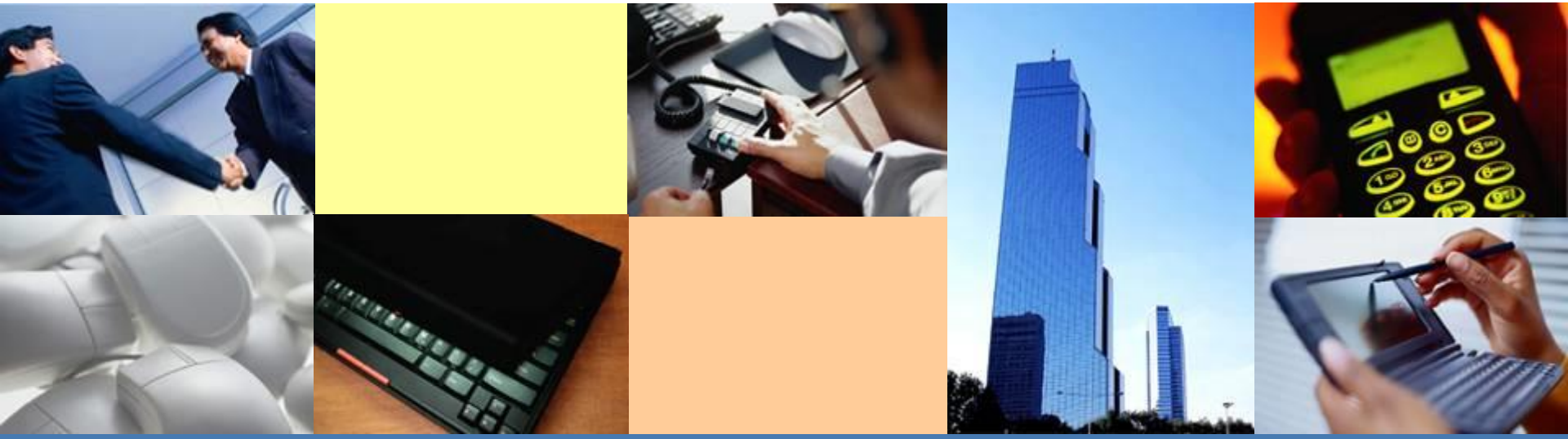
Pseudo Instructions (cont)

- move
  - move rt5,ra5 — for 16-bits, this is mov55 rt5,ra5; for 32-bits, this is ori rt5,ra5,0
  - move rt5,var — this is the same as l.w rt5,var
  - move rt5,imm_32 — this is the same as li rt5, imm_32

Pseudo Instructions (cont)

- push/pop
  - pushm ra5,rb5 push content from ra5 to rb5 into stack.
  - popm ra5,rb5 pop stack values into ra5 to rb5.
  - push ra5 push content of register ra5 into stack.
    same as pushm ra5,ra5
  - pop rt5 pop stack value into register rt5.
    same as popm rt5,rt5

Application Binary Interface (ABI)



Introduction to ABI

- ABI - Application Binary Interface
  - The Andes architecture ABI defines the interface for compiled programs and assembled programs running on Andes architecture to work jointly.
  - The purpose of Andes architecture ABI is to deliver high performance and binary compatibility.

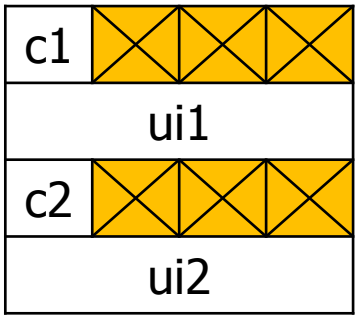
- Our ABI
  - ABI2 (for v3, v3j and v3m Toolchains)
  - ABI2FP+ (for v3s and v3f Toolchains)

Primitive Data Types

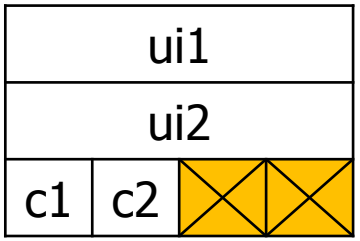
Table 4. Size and Byte Alignment of Primitive Data Types

Class	Machine Type	Size (in Byte)	Alignment (in Byte)
Integer	Unsigned byte	1	1
	Signed byte	1	1
	Unsigned half word	2	2
	Signed half word	2	2
	Unsigned word	4	4
	Signed word	4	4
	Unsigned double word	8	8
	Signed double word	8	8
Floating Point	Single precision (IEEE 754)	4	4
	Double precision (IEEE 754)	8	8
Pointer	Instruction Pointer	4	4
	Data Pointer	4	4


```
struct Test2{
    char c1;
    unsigned int ui1;
    char c2;
    unsigned int ui2;
};
```




```
struct Test1 {
    unsigned int ui1;
    unsigned int ui2;
    char c1;
    char c2;
};
```



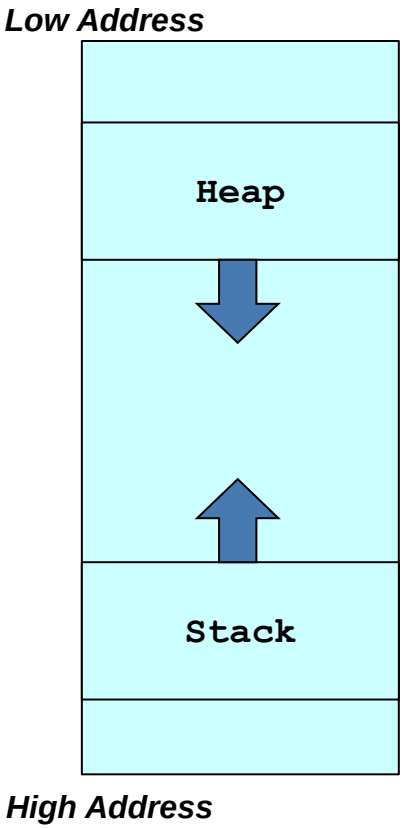
C Language Mapping of Andes Platform

Table 5. Mapping of C Primitive Data Types

C/C++ Type	Machine Type	Size (in Byte)
[singed] char	Signed byte	1
unsigned char	Unsigned byte	1
[signed] short	Signed half word	2
unsigned short	Unsigned half word	2
[signed] int	Signed word	4
unsigned int	Unsigned word	4
[signed] long	Signed word	4
unsigned long	Unsigned word	4
[signed] long long	Signed double word	8
unsigned long long	Unsigned double word	8
size_t	Unsigned word	4
float	Single precision (IEEE 754)	4
double	Double precision (IEEE 754)	8
long double	Double precision (IEEE 754)	8
float _Complex	Two Single precision (IEEE 754)	8
double _Complex	Two Double precision (IEEE 754)	16
long double _Complex	Two Double precision (IEEE 754)	16

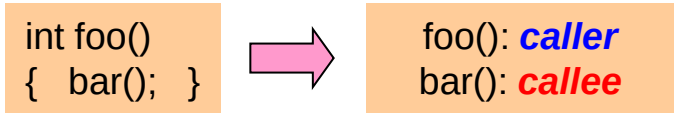
Memory Layout

- Heap
  - global variables
  - static variables
  - constant data
  - …
- Stack
  - local variables
  - context switch data
  - …



Calling Convention – Registers

- "caller" invokes "callee".



- caller save registers
  - $r0 ~ $r5, $r16 ~ $r27
- callee save registers
  - $r6 ~ $r10, $r11 ~ $r14, $r28, $r29, $r30

reduced register configuration

full-set register configuration

- Argument Passing: $r0 ~ $r5
- Return Values: $r0 ~ $r1
- other registers
  - $r15 : Temporary Register (reserved for assembler instruction expansion)
  - $r16 : Trampoline Register (as static chain register for nested function)
  - $r28 (fp), $r29 (gp), $r30 (lp), $r31 (sp)

callee-save registers

Calling Convention – Stack Frame

- Stack frame
  - $sp : Stack Pointer
    - Top of stack frame
  - $fp : Frame Pointer
    - Record the original $sp position
    - ***Omitted by default.***

```
int foo(int, int);
int bar();

int main()
{
  int r;
  r = foo(77, 88);
  return r;
}

int foo(int x, int y)
{
  int z;
  z = bar();
  z = x + y + z;
  return z;
}

int bar()
{
  return 99;
}
```

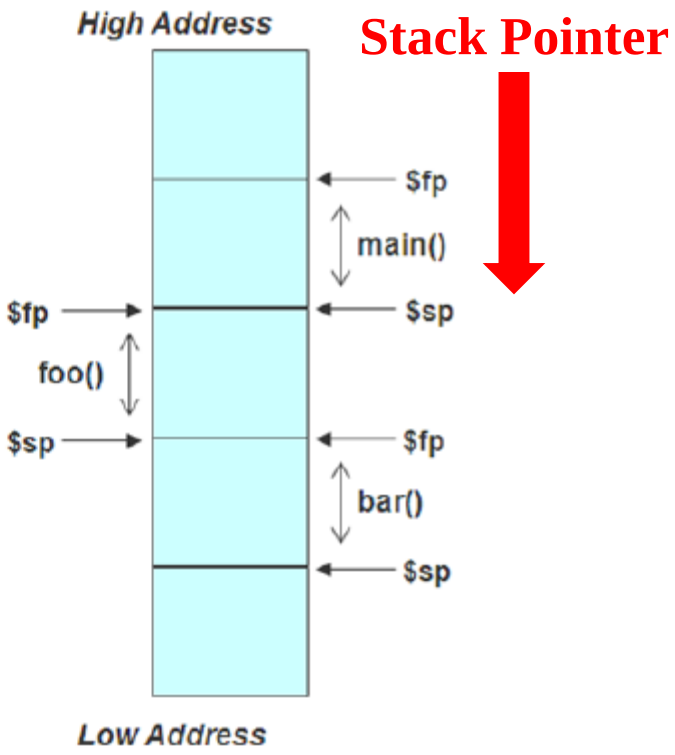
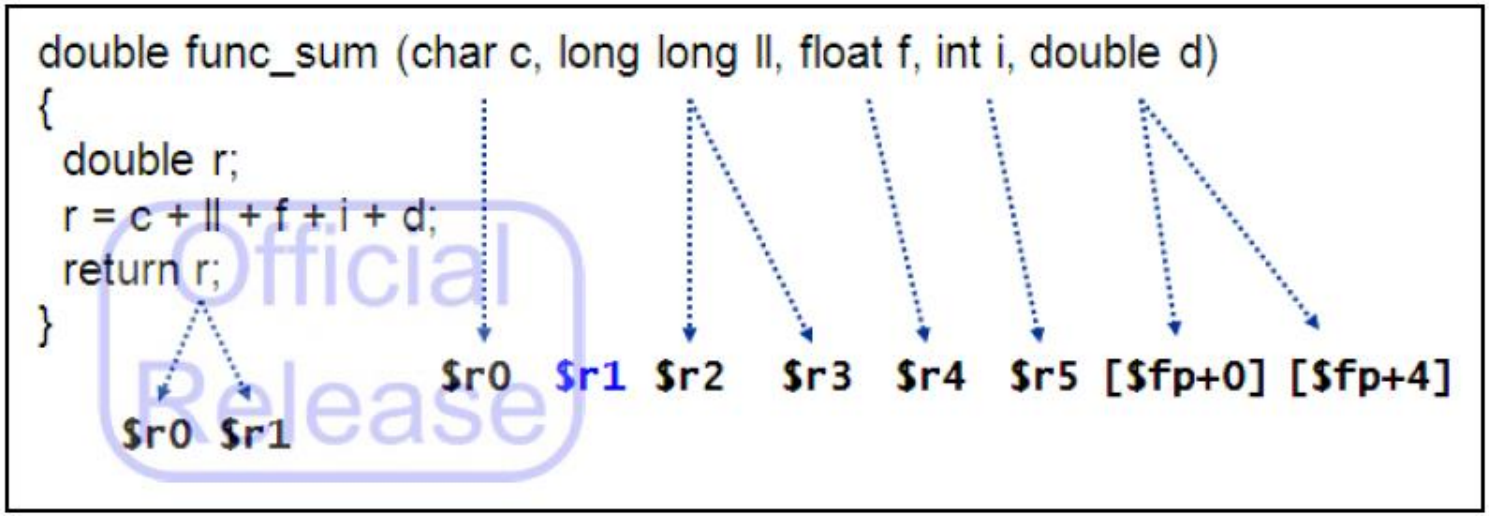


Figure 1. ABI2 Stack Frame Scenario

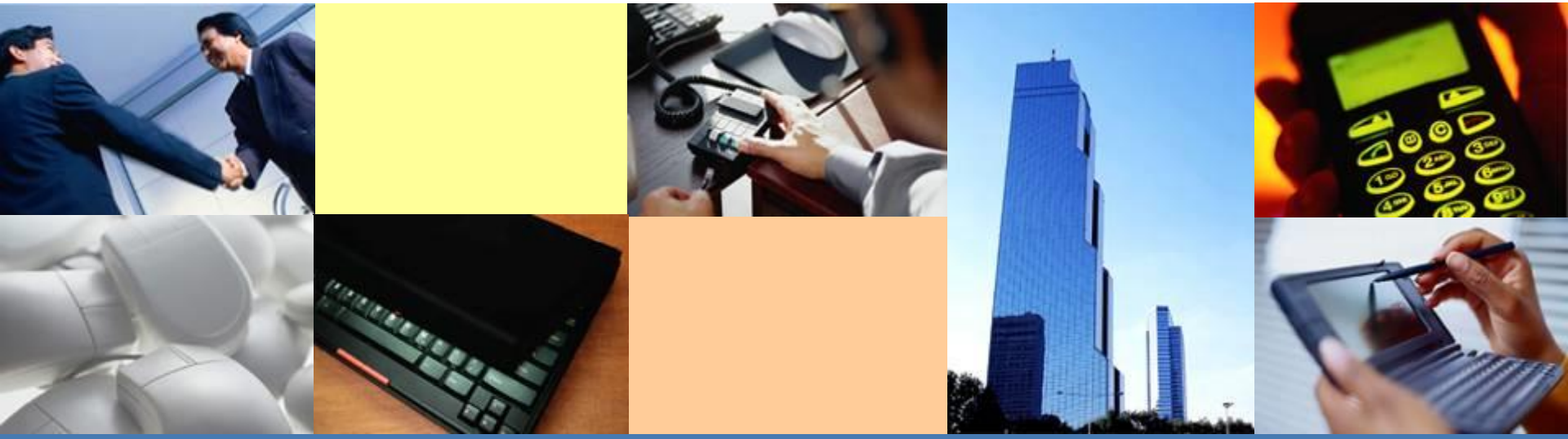
- Without omit $fp
  - -fno-omit-frame-pointer

Calling Convention- Argument Passing and Return

- GPRs $r0~$r5 are used to pass arguments.
- If GPRs $r0~$r5 are not sufficient to hold all arguments, the remaining ones will be passed in the outgoing arguments block of caller's stack frame.
- For 4-byte primitive type, the return value is returned in $r0. For 8-byte primitive type, the return value is returned in $r0 and $r1.



Andes Intrinsic Function Programming



Intrinsic Function

- These functions available in a given language whose implementation is handled specially by the compiler.
- Intrinsic function is usually inserted inline.
- Avoid the overhead of a function call.

Most Commonly Used

- __nds32__mtsr(0x0, NDS32_SR_INT_MASK);
  - Set $int_mask=0
- tmp = __nds32__mfsr(NDS32_SR_PSW);
  - Put the vaule of $psw to tmp
- __nds32__gie_dis()
  - Disable global interrupt (will take effect immediately)
- __nds32__gie_en()
  - Enable global interrupt (will take effect immediately)

Standby

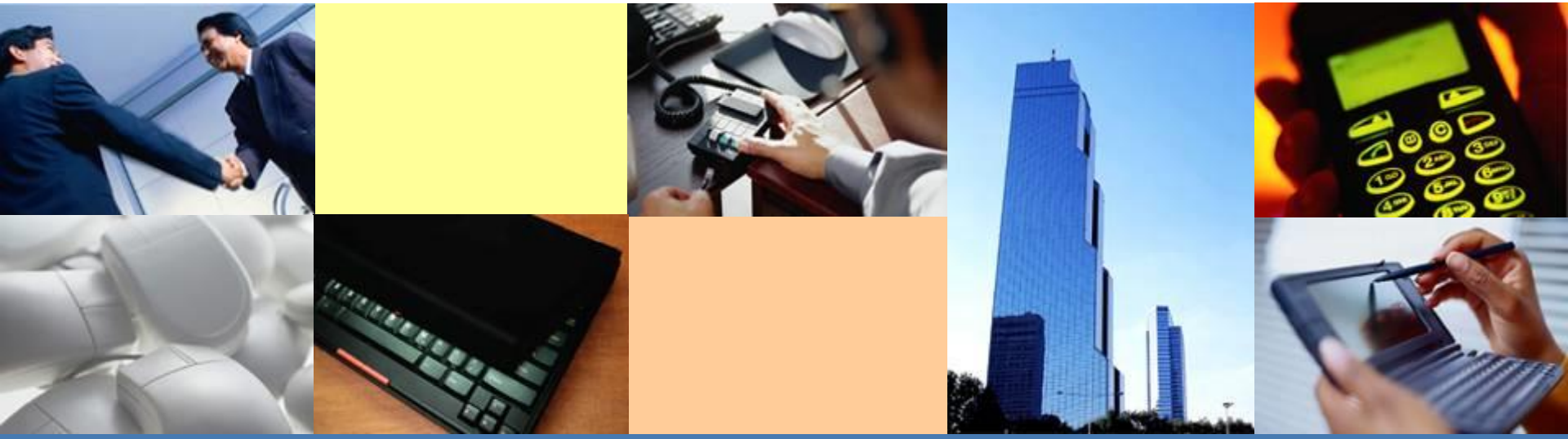
- __nds32__standby_no_wake_grant()
  - STANDBY no_wake_grant
- __nds32__standby_wake_grant()
  - STANDBY wake_grant
- __nds32__standby_wait_done()
  - STANDBY wait_done

Other Useful Intrinsic Functions (1)

- __nds32__dsb()
- __nds32__isb()
- __nds32__nop()
- __nds32__enable_int(enum nds32_intrinsic int id)
- __nds32__disable_int(enum nds32_intrinsic int id)
- __nds32__set_pending_swint ()
  - Sets pending software interrupt
- __nds32__clr_pending_swint ()
  - Clears pending software interrupt

ANDES TECHNOLOGY

Inline Assembly Programming



Inline Assembly

- Inline assembly programming is a way GCC provides to write assembly code embedded in C program.
- To declare inline assembly functions, we use __asm__ (...) or asm (...)
- Example:

```
        sprintf(buf, "Syscall handler #%d\n", cnt);
        uart_puts(buf);

        if (++cnt < 5)
            asm("syscall 0x5000\n\t":::"$r0");
}
```

Format of Inline Assembly

- Basic form of inline assembly programming:
  - __asm__ ("an assembly code template"
    : a list of output operands
    : a list of input operands
    : a list of clobber registers);
  - In a clobber list, registers or memory are listed to inform GCC that these items have been modified.
  - Register used in an assembly code template have to be specified in the clobber list **so that** GCC will assume the content of the registers are invalid after the inline assembly statement and generate extra instructions to maintain correct register status.

Example of Inline Assembly

```
int func_asm_1 (int i, int j)
{
    int ret;

    __asm__ ("add\t%0, %1, %2\n\t"
             "movi\t$r6, 123\n\t"
             "add\t%0, %0, $r6"
             : "=r" (ret)
             : "r" (i), "r" (j)
             : "$r6");
    return ret;
}
```

➔ ret = i + j + 123;

- "%0", "%1", and "%2" represent three operands and GCC will replace them from the output operand list to the input operand list.
- A constraint of an operand is used to indicate the addressing mode. Constraint "r" means operands should be placed in general registers and constraint modifier "=" is used for output operands, indicating the operands are write-only.
- "\t" to separate an instruction from its first operand in an assembly code template
- "\n\t" is due to gcc sends each instruction as a string to AS.

Volatile

- GCC may move or delete statements in optimization.
- If our assembly statement must execute where we put it, (must not moved out when optimization).
- If it is just for doing some calculations and no side effects, it's better to use volatile to avoid optimization.

```
__asm__ __volatile__ ("setend.b");
asm volatile ( ... : ... : ... : ...);
```

Proper Use of Volatile in C (1)

- A variable should be declared volatile whenever its value could change unexpectedly.
  1. Memory-mapped peripheral registers
  2. Global variables modified by an interrupt service routine
  3. Global variables accessed by multiple tasks within a multi-threaded application
- Reference link:
  - http://www.barrgroup.com/Embedded-Systems/How-To/C-Volatile-Keyword

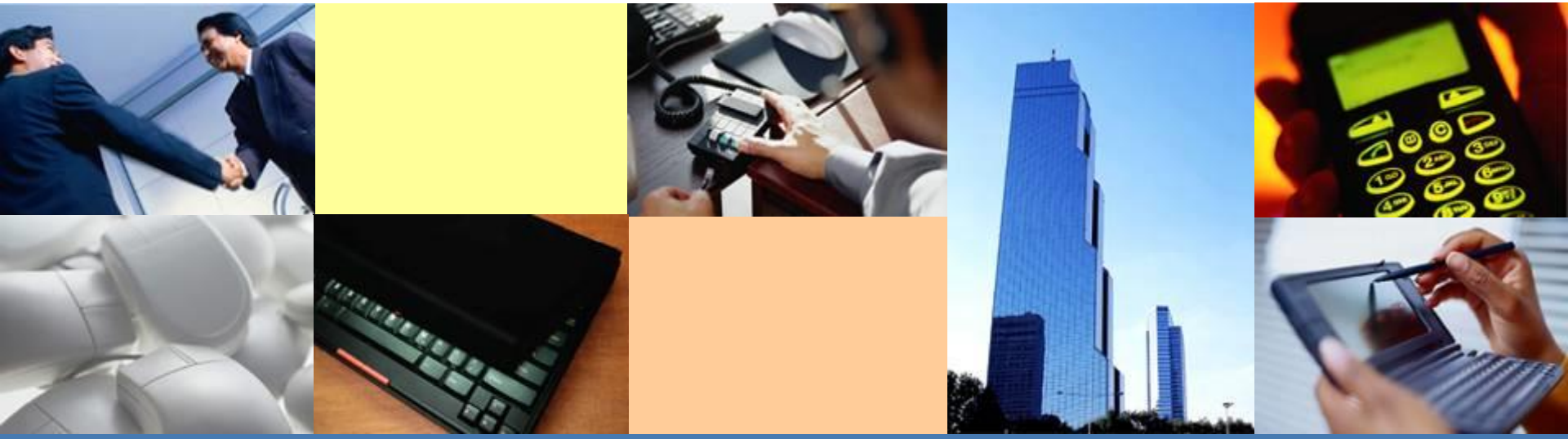
Proper Use of Volatile in C (2)

- Example: Global variables modified by an interrupt service routine

```
void xxx(void)
{
    wait=1;
    while (wait!=0);
    .....
}
void timer0_isr(void)
{
  wait=0;
  .......
}
```

ANDES TECHNOLOGY

Advanced Programming Optimization



Options for Code Size Optimization

- Compiler Options
  - -Os

- Sometimes the code size optimizations may degrade the performance.
- Three code size optimization levels.

Option	Code Size Optimization Level
-Os1	Minimum code size optimizations. Performance is still concerned.
-Os2	Partial code size optimizations. Little performance concern.
-Os3 (-Os)	Maximum code size optimizations. Performance may seriously drop.

Options for Code Speed Optimization

- Compiler Options
  - -O3 (-O0, -Og, -O1, -O2, -O3) (-Og: better debuggability than -O1)
  - -funroll-loops / -funroll-all-loops
  - -ftree-switch-shortcut – Experimental for switch statement.
  - -malign-functions – aligns function entries to 4-byte boundaries.
  - -malways-align – enforces 4-byte alignment on jump targets, return addresses and function entries.
- "-O3" implies "-finline-functions". Avoid it with "-fno-inline-functions".

Option	Description
-funroll-loops	Number of iterations can be determined at compile time or upon entry to the loop. Compiler has a set of heuristics to estimate whether to unroll loop or not.
-funroll-all-loops	Even if their number of iterations is uncertain when the loop is entered. This option probably makes programs run more slowly if it loses locality after unrolling.

Options to Remove Unused Sections

- Following compiler and linker options have to be enabled at the same time.
- Compiler Options
  - -ffunction-sections
  - -fdata-sections
- Linker Options
  - (gcc as linker) -Wl,--gc-sections
  - (ld as linker) --gc-sections
- Easily see what sections are discarded by linker
  - (gcc as linker) -Wl,--print-gc-sections
  - (ld as linker) --print-gc-sections

EX9 Optimization

- Compiler Option
  - -mex9
- Linker Options
  - (gcc as linker) -Wl,--mex9
  - (ld as linker) --mex9
- Enable by default in "-Os".
  - To disable it
    - (gcc as linker) -Wl,--mno-ex9
    - (ld as linker) --mno-ex9
- Compiler uses16-bit "ex9.it <INDEX>" with <INDEX> pointing to the corresponding 32-bit instruction.

Original

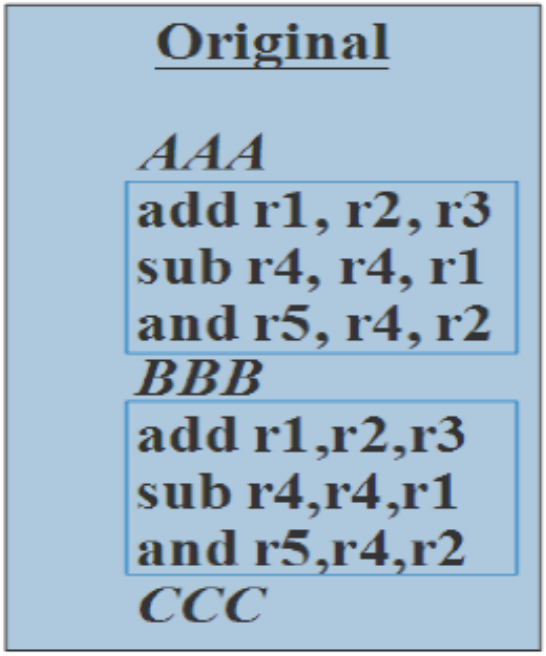
```
...
lbsi $r0,[$r14+#0x0]
...
```

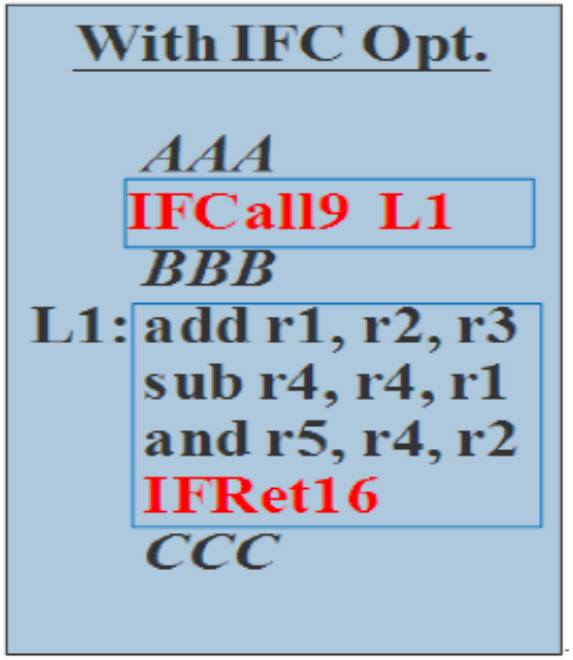
With EX9 Opt.

```
...
ex9.it #1  ! lbsi $r0,[$r14+#0x0]
...
.ex9.itable:
   sb $r0,[$r7+($r6<<#0x0)]
   lbsi $r0,[$r14+#0x0]
   ...
```

IFC (Inline Function Call) Optimization

- IFCall9 (16b), IFCall (32b) and IFRet16 (16b) instructions are used to share the common code.
- Compiler Option
  - -mifc
- Linker Options
  - (gcc as linker) -Wl,--mifc
  - (ld as linker) --mifc





Default Applied Options of Optimization

Table 26. Default Applied GCC Options at Each Optimization Level

Mnemonic	-O0	-Og	-O1	-O2	-O3	-Os1	-Os2	-Os/-Os3
-fomit-frame-pointer	√	√	√	√	√	√	√	√
-fno-delete-null-pointer-checks	√	√	√	√	√	√	√	√
-finline-functions					√			
-mrelax	√	√	√	√	√	√	√	√
-malign-functions		√	√	√	√			
-malways-align		√	√	√	√			
-minnermost-loop							√	
-mex9								√
-mifc								√

- Using -fno-omit-frame-pointer, -fdelete-null-pointer-checks, -fno-inline-functions, -mno-relax, -mno-align-functions, -mno-always-align, -mno-innermost-loop, -mno-ex9, and -mno-ifc can avoid the options.

Set Different Optimization Levels (1)

- Set optimization level for a function.
- Syntax:

```
void __attribute__ ((optimize("Os"))) __cpu_init()
{
    .....
 }
```

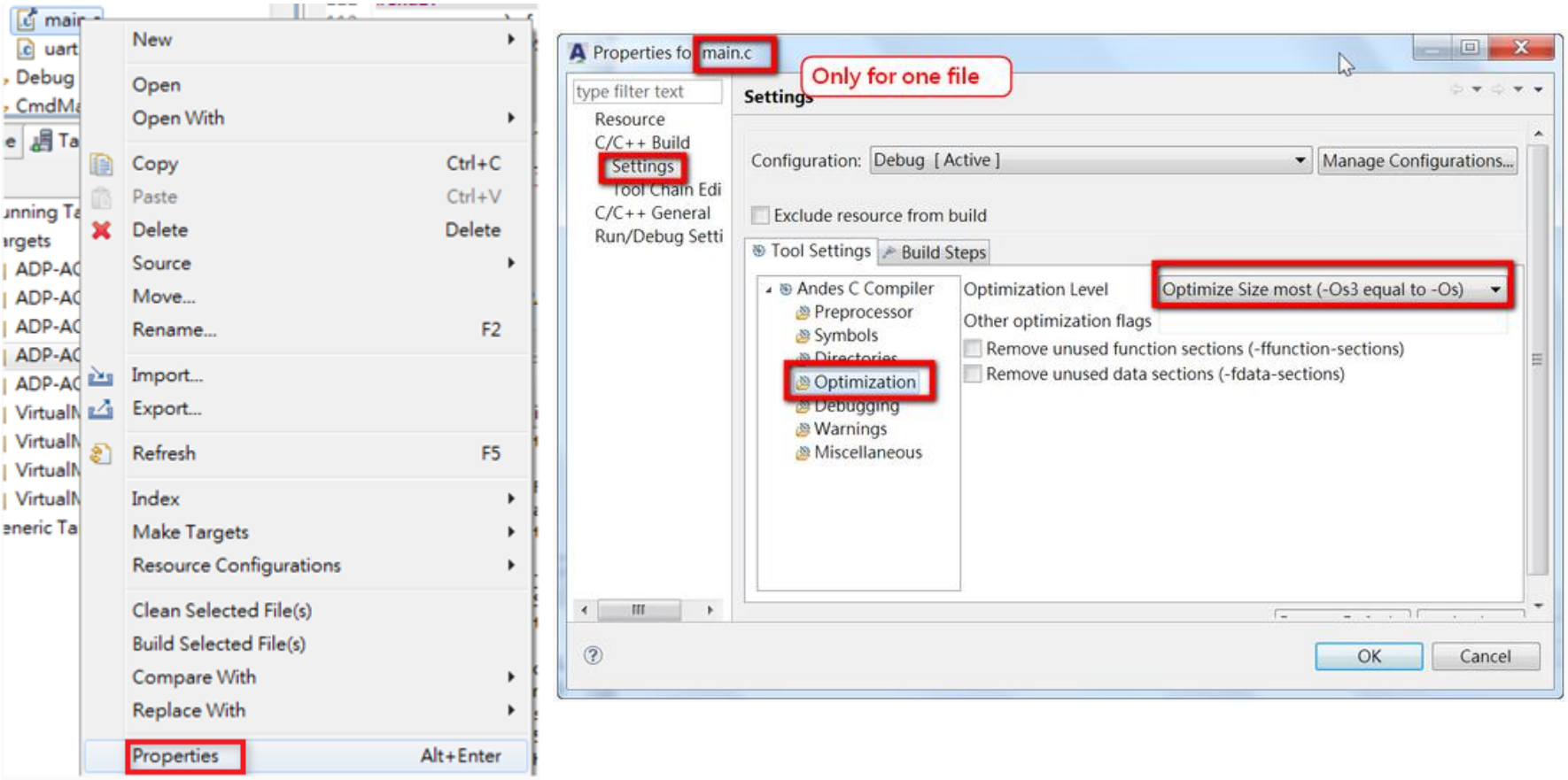
Set Different Optimization Levels (2)

- Set optimization level for a block codes in a C file.
- Syntax

```
#pragma GCC push_options
#pragma GCC optimize ("Os")
  /* code to be set to -Os */
#pragma GCC pop_options
 /* restore to original optimization level */
```

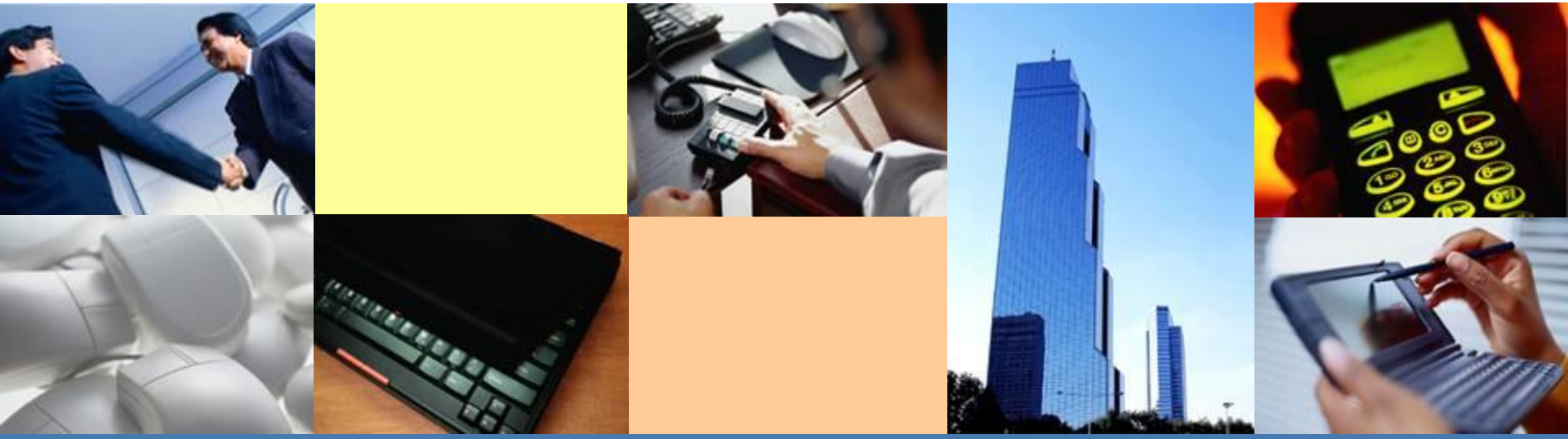
Set Different Optimization Levels (3)

- Set optimization level for a C file.
- Right-Click on the file, then select "Properties".



ANDES TECHNOLOGY

Link Time Optimization in GCC



Link Time Optimization in GCC

- Link Time Optimization (LTO) – A very aggressive optimization implemented by GCC.
- Should use GCC to complete all the works of building a program, including compilation and linking.
- Option – "-flto"
- Notice
  - Avoid defining the same module name as it's presented in the library.
  - If a module that may be called from the MCU standard library (e.g. the weak function nds_write() redirected from libc.a), it is suggested to use __attribute__((used)) to prevent it from being optimized out by LTO.

Optimization for Speed – Console Mode

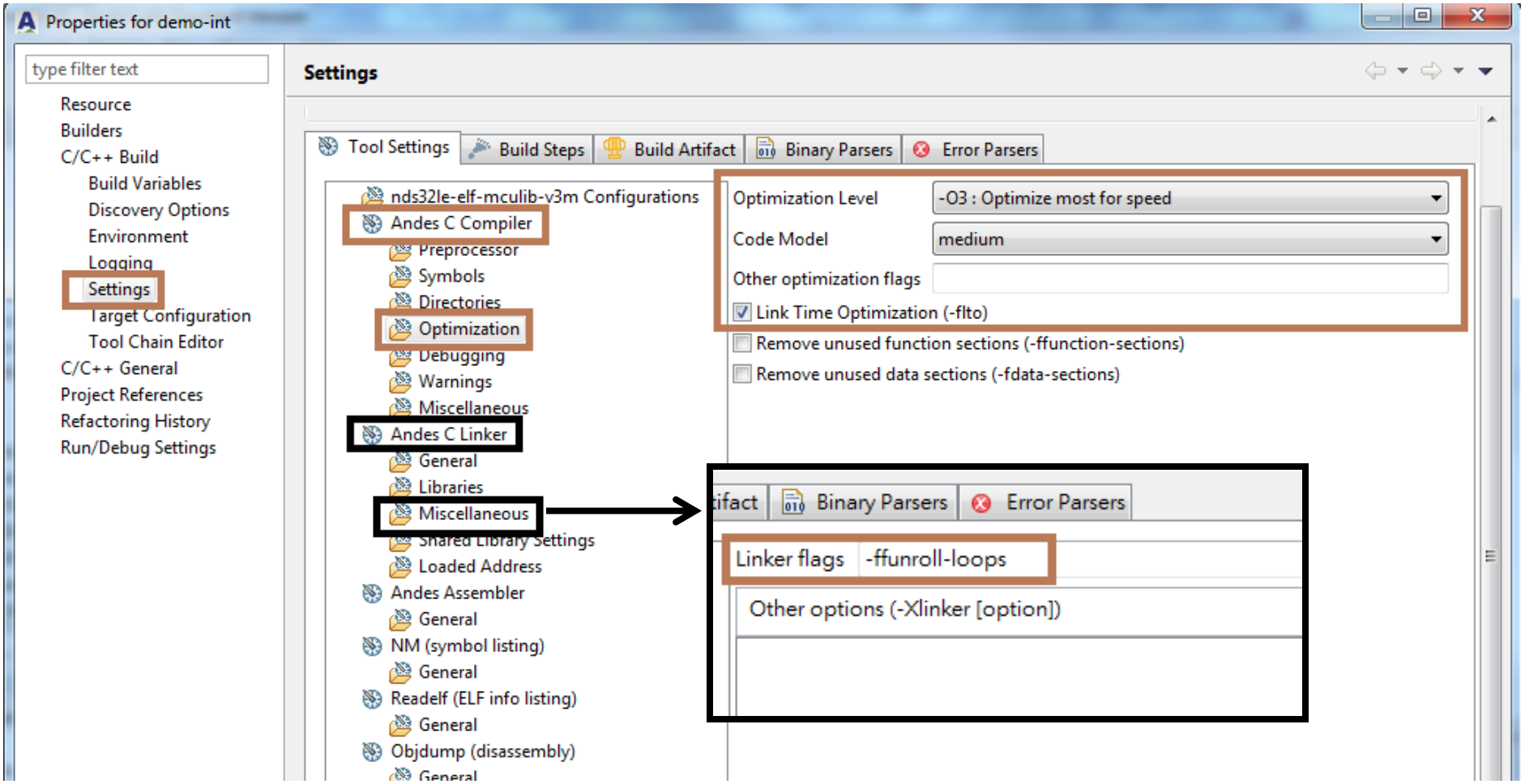
- Setting C compiler options "-O3 -flto -funroll-loops"

```
Building file: ../src/init-default.c
Invoking: Andes C Compiler
nds32le-elf-gcc -O3 -mcmodel=medium -funroll-loops -flto -g3 -Wall -mcpu=n968a -c -fmessage-length=0 -MMD -MP
-MF"src/init-default.d" -MT"src/init-default.d" -o "src/init-default.o" "../src/init-default.c"
Finished building: ../src/init-default.c
```

- Setting C linker options "-O3 -flto -funroll-loops"

```
Invoking: Andes C Linker
nds32le-elf-gcc -O3 -nostartfiles -static -funroll-loops -T"../nds32-xip.ld" -flto -o "demo-int.adx"  ./src/crt0.o
./src/crtbegin.o ./src/crtend.o ./src/init-default.o ./src/init-soc.o ./src/interrupt.o ./src/main.o ./src/uart.o   -ldsp
Finished building target: demo-int.adx
```

Optimization for Speed – AndeSight



Optimization for Code Size – Console Mode

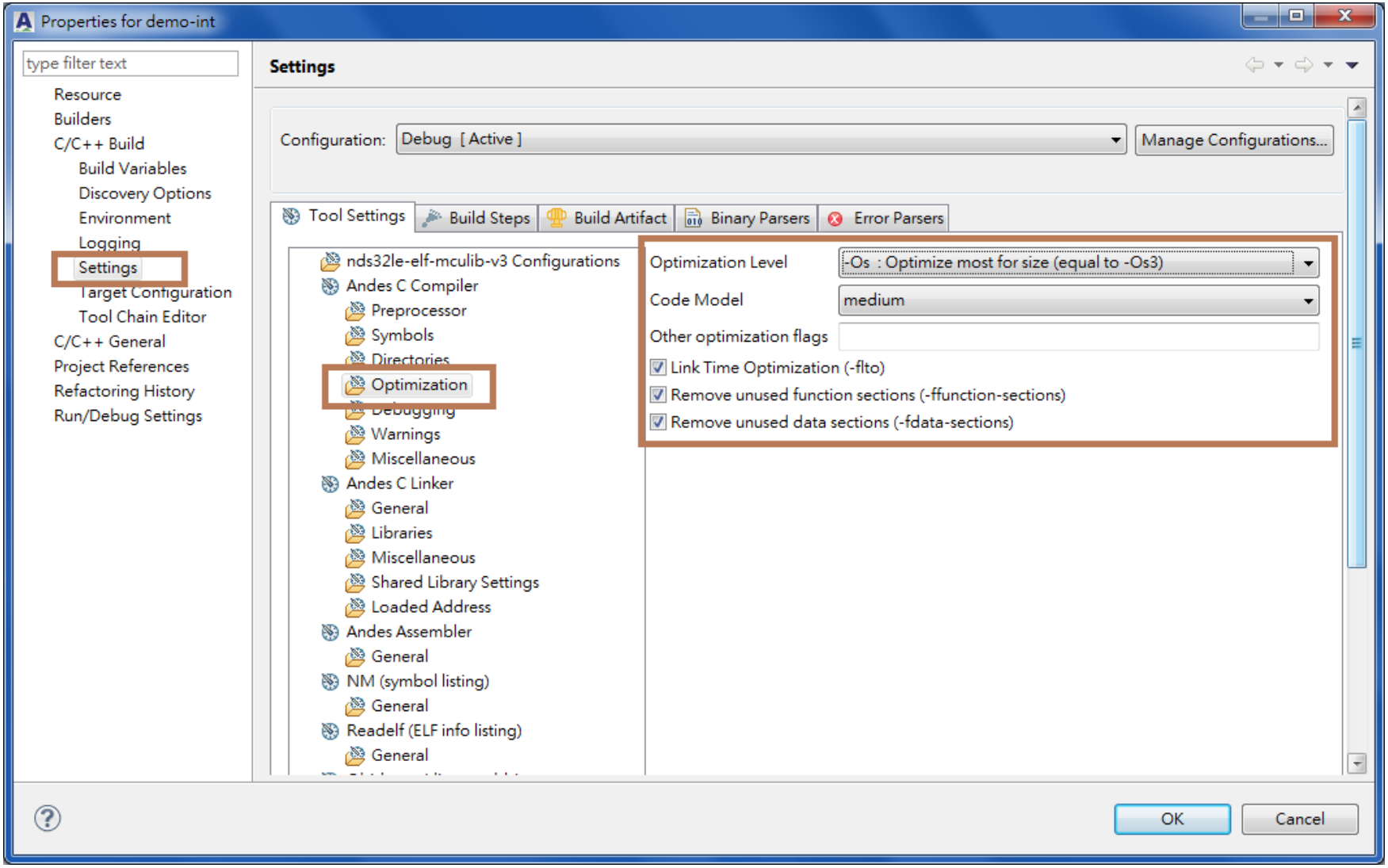
- Setting C compiler options "-Os -flto -ffunction-sections -fdata-sections"

```
Building file: ../src/interrupt.c
Invoking: Andes C Compiler
nds32le-elf-gcc -Os -mcmodel=medium -flto -g3 -Wall -mcpu=n968a -c -fmessage-length=0 -ffunction-sections -fdata-sections -MMD
-MP -MF"src/interrupt.d" -MT"src/interrupt.d" -o "src/interrupt.o" "../src/interrupt.c"
Finished building: ../src/interrupt.c
```

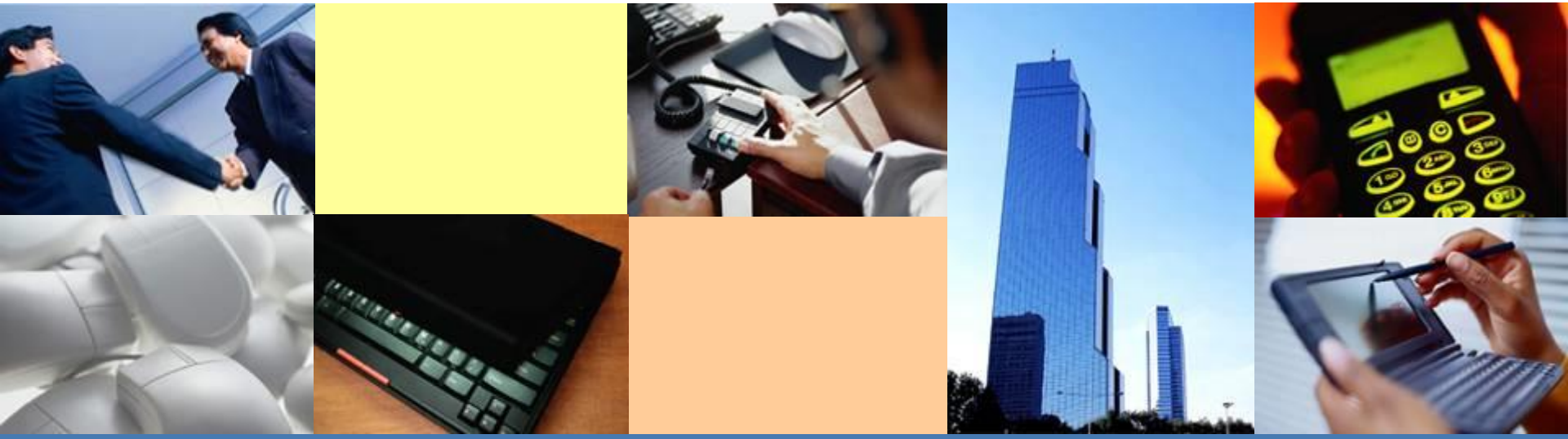
- Setting C linker options "-Os -flto -Wl,--gc-sections"

```
Building target: demo-int.adx
Invoking: Andes C Linker
nds32le-elf-gcc -Os -nostartfiles -static -T"../nds32-xip.ld" -Wl,--gc-sections -flto -o "demo-int.adx"  ./src/crt0.o
./src/crtbegin.o ./src/crtend.o ./src/init-default.o ./src/init-soc.o ./src/interrupt.o ./src/main.o ./src/uart.o   -ldsp
Finished building target: demo-int.adx
```

Optimization for Code Size – AndeSight



Addressing Space for Programs



Small Data Area

- Most programs do not require complete 32-bit addressing space because of limited resources (e.g. ROM size) in practice.
- Improve the overall performance and code size simply with the concept of small data area or using different code models in compiler option.

- Andes SDA (Small Data Area) can be addressed by an offset plus register $gp. (limited to +/- 256KB)
  - .sdata_{b|h|w|d}: Section for initialized global variables.
  - .sbss_{b|h|w|d}: Section for uninitialized global variables.

Code Models

- Tell compiler which scale your programs and data are
  - Option: -mcmodel=[small | medium | large]
- -mcmodel=small (code model: 16M text, 512K data+rodata)
  - Compiler assumes that all the data is in the small data area and generates addressing with offset plus $gp.
- -mcmodel=medium (code model: 16M text, 512K data, 4G rodata)
  - **Default setting.** RO-data beyond 512K of small data area full 32-bit address (constant variables). Other global variables are within 512K range of small data area and accessible with $gp relative instruction.
- -mcmodel=large (code model: 4G text, 4G data + rodata)
  - All the text and data are all over complete 32-bit addressing space. Compiler leaves all the relaxation works to linker.

Others



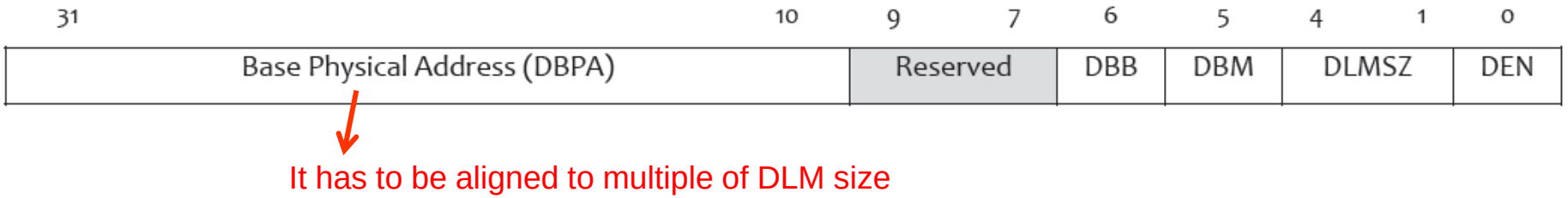




Local Memory

- Please refer to SPA document 10.4.8 DLM Base Register

DCM_CFG.BSAV == 1



- If we want to set DLM base in 0x100000

gdb command: set $dlmb=0x100001

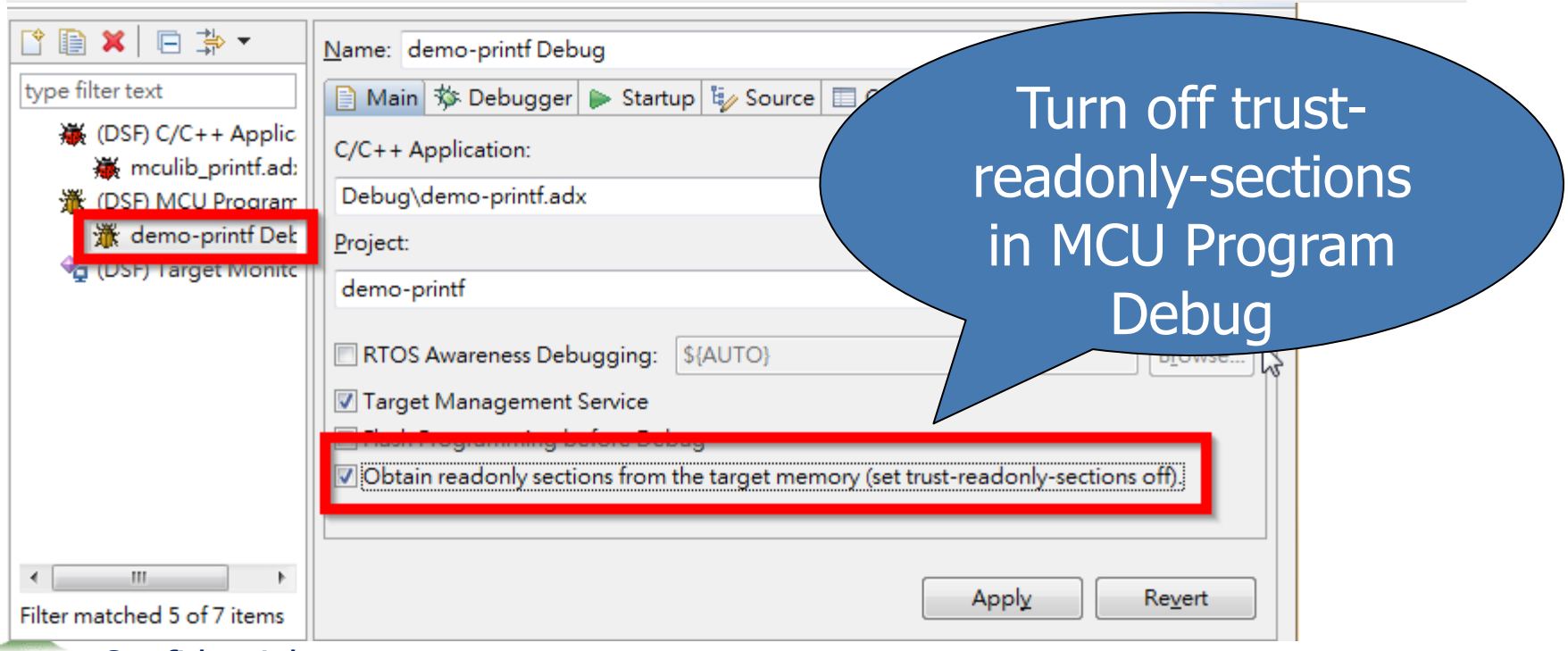
C Code:

```
__nds32__mtsr(0x100001, NDS32_SR_DLMB);
__nds32__dsb();
```

GDB Returns Unexpected Value from Memory

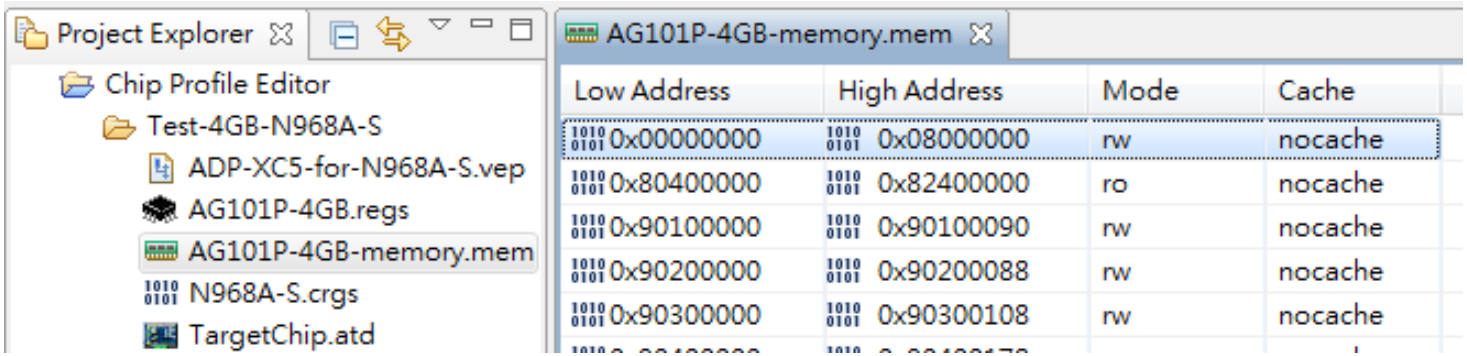
If trust-readonly-sections is set, GDB will fetch values from read-only sections out of local files, rather than from the target.

```
(gdb) show trust-readonly-sections
Mode for reading from readonly sections is on.
(gdb) set trust-readonly-sections off
```



Memory Map Setting

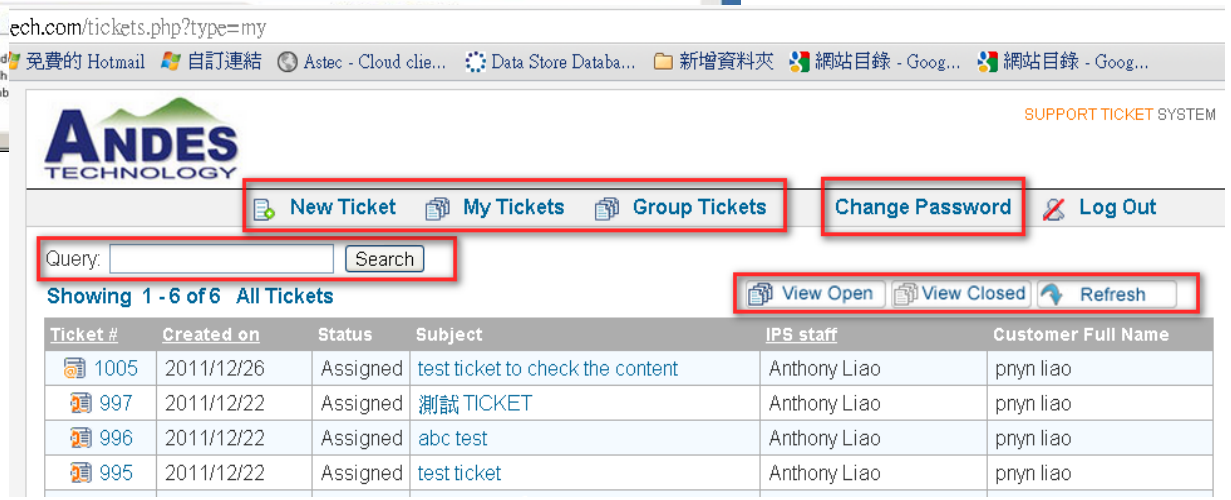
- Internal breakpoints is set by gdb commands like next and finish.
- GDB can automatically decide using HW or SW breakpoints base on the address is RO or RW.
  - With AndeSight target platform, please check "AndeSight*\MemoryMap*.mem".
  - With user's target platform, please check Memory Map file.



- Andes e-service is a web-based support ticket system. It is a convenient way to get quick response of your question.
- How to apply for a new account?
  - Please send your information to es.admin@andestech.com
    - Including your name, e-mail, company name and telephone number

How to Use E-service?

- E-service website: http://es.andestech.com/ilogin.php



You can track your ticket in e-service system