

第三章

程式設計

3.1 AndeStar™ 指令集介紹

AndeStar™ ISA 是 AndesCore 的指令集名稱 (ISA : Instruction Set Architecture)。目前已發佈到 V3 及 V3m 版，本書使用的系統核心 N903 及 N801-S 分別使用 V2j 及 V3m 的指令集。AndeStar™ ISA 支援 32/16 位元混合使用的指令集，其特性概述如下：

- 32 位元與 16 位元指令在程式中可任意混合使用
- 大多數的 16 位元指令是常用的 32 位元的指令重新編碼而成
- 32 位元／16 位元指令切換時不會造成性能低落
- 大端 (big-endian) 的儲存格式
- 32 位元指令以 5 個位元定址暫存器，16 位元指令有 5/4/3 個位元的暫存器定址方式
- 大多數指令與一般精簡指令集的指令一樣，只有少數例外

AndeStar™ ISA 處理的資料格式：

- 整數
 - 位元 (bit)
 - 位元組 (byte)
 - 半字組 (HalfWord : 16 位元)
 - 字組 (Word : 32 位元)
- 浮點數
 - 單精準浮點數 (32bit, single-precision)
 - 倍精準浮點數 (64bit, double-precision)

AndesCore 的暫存器概分成三類：1. GPR (general purpose register) 通用暫存器；2. USR (user special register) 使用者特殊暫存器；3. SR (system Register) 系統暫存器。原則上，AndesCore 內部有 32 個通用暫存器 (表 3.1-1) 提供一般權限的使用者使用，其中 R0 ~ R5 可用來傳遞參數給被呼叫的副程式，R15 保留給組譯器使用，R26 ~ 27 保留給作業系統使用 (特

權模式使用者)使用,因此一般會建議程式設計者不要使用。當 AndesCore 規劃成精簡暫存器集的核心時,能使用的暫存器只剩 16 個,如表 3.1-2 所示。本書使用的兩個 CPU 皆只能使用 16 個暫存器。表 3.1-1 與表 3.1-2 中第 2 欄是暫存器的別名,32 位元的指令碼編成機器碼時,使用 5 個位元指定暫存器的號碼,當使用 16 位元指令時,機器碼指定暫存器的方式有 5 個位元、4 個位元以及 3 個位元,暫存器定址使用不同位元數的編碼方式使用不同的暫存器名稱。R28 ~ R31 具有特殊功能任務,R28 (framepointer) 通常用在副程式中存取堆疊區資料的指標暫存器,R29 (global pointer) 通常用來存取整體變數及常數用,R30 (link pointer) 存放回返主程式的位址,R31 (stack pointer) 擔任存取堆疊區資料的指標暫存器。表 3.1-3 是具有累積器功能的使用者特殊暫存器。只有 32 位元指令才可以存取,d0.hi 與 d0.lo 組合成 D0,d1.hi 與 d1.lo 的組合成 D1,d0.hi ~ d1.lo 暫存器可各別分開使用,也可各自組合成 2 個 64 位元的暫存器。至於系統暫存器種類甚多,計有系統規劃 (configuration)、中斷 (interruption)、記憶體管理 (memory management)、性能監測 (performance monitoring)、本地記憶體直接存取 (local memory DMA)、系統資源存取控制暫存器 (Resource Access Control)、嵌入式除錯模組 (embedded debug module) 與製造相關的暫存器 (implementation-dependent) 等。

表 3.1-1 AndeStar™ 通用暫存器

Register	32/16-bit (5)	16-bit (4)	16-bit (3)	Comments
r0	a0	h0	o0	Saved by caller
r1	a1	h1	o1	Saved by caller
r2	a2	h2	o2	Saved by caller
r3	a3	h3	o3	Saved by caller
r4	a4	h4	o4	Saved by caller
r5	a5	h5	o5	Implied register for beg38 and bnes38
r6	s0	h6	o6	Saved by callee
r7	s1	h7	o7	Saved by callee
r8	s2	h8		Saved by callee

Register	32/16-bit (5)	16-bit (4)	16-bit (3)	Comments
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]z s8
r16		h12		Saved by caller
r17		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

表 3.1-2 精簡暫存器集 (16 Registers)

Register	32/16-bit (5)	16-bit (4)	16-bit (3)	Comments
r0	a0	h0	o0	Saved by caller
r1		h1	o1	Saved by caller
r2		h2	o2	Saved by caller
r3		h3	o3	Saved by caller
r4	a4	h4	o4	Saved by caller
r5	a5	h5	o5	Implied register for beqs38 and bnes38

r6	s0	h6	o6	Saved by callee
r7	s1		o7	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	s9/fp			Frame pointer /Saved by callee
r29	gp			Global pointer
r30	lp			Link pointer
r31	sp			Stack pointer

表 3.1-3 AndeStar™ 使用者特殊暫存器 (USR)

Register	32-bit Instr.	16-bit Instr.	Comments
D0	d0.{hi, lo}	N/A	與乘法與除法指令相關的暫存器 (64-bit)
D1	d1.{hi, lo}	N/A	與乘法與除法指令相關的暫存器 related instructions (64-bit)

3.1.1 指令編碼

32 位元指令集與 16 位元指令集的辨識方式取決於指令集的編碼方式，32 位元指令集的機器碼最高次位元一定是「0」，但 16 位元指令集的機器碼最高次位元一定是「1」，以上所述簡化如下：

Bit [31] = 0 表示是 32 位元指令集

Bit [15] = 1 表示是 16 位元指令集

典型的 32 位元指令編碼格式如下：

0	30:25 (6)	24:0 (25)
---	-----------	-----------

Bit [31] = 0 表示是 32 位元指令

Bit [30:25] 表示是運算碼 (6 個位元)

Bit [24:0] 表示是運算元／立即運算元／子運算碼

»» 微處理器應用與實作：C 語言與 Andes MCU 系列

0	opc_6	{sub_1, imm_24}				
0	opc_6	rt_5	imm_20			
0	opc_6	rt_5	sub_4	imm_16		
0	opc_6	rt_5	ra_5	rb_5	rd_5	sub_5
0	opc_6	rt_5	ra_5	imm1_5	imm2_5	sub_5
0	opc_6	rt_5	ra_5	rb_5	sub_10	
0	opc_6	rt_5	ra_5	imm_5	sub_10	
0	opc_6	rt_5	ra_5	rb_5	rd_5	sub_5
0	opc_6	rt_5	ra_5	imm1_5	imm2_5	sub_5
0	opc_6	rt_5	ra_5	rb_5	rd_5	sub_5
0	opc_6	rt_5	ra_5	imm1_5	imm2_5	sub_5

3.1.2 資料運算處理指令 (Data Processing)

在指令助記憶碼 (Mnemonic) 中，W：word (字元組)，H：halfword (半字組)，B：byte (位元組)，I：immediate (立即運算元)。

表 3.1.2-1 帶立即運算元的算術邏輯運算

Mnemonic	Instruction	Operation
ADDI rt5, ra5, imm15s	Add immediate	$rt5 = ra5 + SE(imm15s)$
SUBRI rt5, ra5, imm15s	Subtract Reverse immediate	$rt5 = SE(imm15s) - ra5$
ANDI rt5, ra5, imm15u	And Immediate	$rt5 = ra5 \&\& ZE(imm15u)$
ORI rt5, ra5, imm15u	Or Immediate	$rt5 = ra5 \parallel ZE(imm15u)$
XORI rt5, ra5, imm15u	Exclusive Or Immediate	$rt5 = ra5 \wedge ZE(imm15u)$
SLTI rt5, ra5, imm15s	Set on Less Than Immediate	$rt5 = (ra5 \text{ (unsigned)} < SE(imm15s)) ? 1 : 0$
SLTSI rt5, ra5, imm15s	Set on Less Than Signed Immediate	$rt5 = (ra5 \text{ (signed)} < SE(imm15s)) ? 1 : 0$
MOVI rt5, imm20s	Move Immediate	$rt5 = SE(imm20s)$
SETHI rt5, imm20u	Set High Immediate	$rt5 = \{imm20u, 12'b0\}$ 。
u：unsigned 無號數，s：signed 有號數。SE：有號數擴展成 32 位元，也就將最高次位元值填滿到位元 31。 $\{imm20u, 12'b0\}$ ，表示 bit31~12 填入立即運算元 imm20u，bit11~bit0 填入 0。		

表 3.1.2-2 算術邏輯運算 V2

Mnemonic	Instruction	Operation
ADDI gp rt5, imm19s	GP-implied Add Immediate	rt5 = gp + SE(imm19s)

表 3.1.2-3 未帶立即運算元的算術邏輯運算

Mnemonic	Instruction	Operation
ADD rt5, ra5, rb5	Add	rt5 = ra5 + rb5
SUB rt5, ra5, rb5	Subtract	rt5 = ra5 - rb5
AND rt5, ra5, rb5	And	rt5 = ra5 && rb5
NOR rt5, ra5, rb5	Nor	rt5 = ~(ra5 rb5)
OR rt5, ra5, rb5	Or	rt5 = ra5 rb5
XOR rt5, ra5, rb5	Exclusive Or	rt5 = ra5 ^ rb5
SLT rt5, ra5, rb5	Set on Less Than	rt5 = (ra5 (unsigned) < rb5) ? 1 : 0
SLTS rt5, ra5, rb5	Set on Less Than Signed	rt5 = (ra5 (signed) < rb5) ? 1 : 0
SVA rt5, ra5, rb5	Set on Overflow Add	rt5 = ((ra5 + rb5) overflow) ? 1 : 0
SVS rt5, ra5, rb5	Subtract	rt5 = ((ra5 - rb5) overflow) ? 1 : 0
SEB rt5, ra5	Sign Extend Byte	rt5 = SE(ra5[7:0])
SHE rt5, ra5	Sign Extend Halfword	rt5 = SE(ra5[15:0])
ZEB rt5, ra5 (alias of ANDI rt5, ra5, 0xFF)	Zero Extend Byte	rt5 = ZE(ra5[7:0])
ZEH rt5, ra5	Zero Extend Halfword	rt5 = ZE(ra5[15:0])
WSBH rt5, ra5	Word Swap Byte within Halfword	rt5 = {ra5[23:16], ra5[31:24], ra5[7:0], ra5[15:8]}

表 3.1.2-4 移位指令

Mnemonic	Instruction	Operation
SLLI rt5, ra5, imm5u	Shift Left Logical Immediate	rt5 = ra5 << imm5u
SRLI rt5, ra5, imm5u	Shift Right Logical Immediate	rt5 = ra5 (logic) >> imm5u
SRAI rt5, ra5, imm5u	Shift Right Arithmetic Immediate	rt5 = ra5 (arith) >> imm5u
ROTRI rt5, ra5, imm5u	Rotate Right Immediate	rt5 = ra5 >> imm5u
SLL rt5, ra5, rb5	Shift Left Logical	rt5 = ra5 << rb5(4,0)
SRL rt5, ra5, rb5	Shift Right Logical	rt5 = ra5 (logic) >> rb5(4,0)
SRA rt5, ra5, rb5	Shift Right Arithmetic	rt5 = ra5 (arith) >> rb5(4,0)
ROTR rt5, ra5, rb5	Rotate Right	rt5 = ra5 >> rb5(4,0)
Rb5(4,0) : 只取 bit4~0 的值		

表 3.1.2-5 乘法指令

Mnemonic	Instruction	Operation
MUL rt5, ra5, rb5	Multiply Word to Register	$rt5 = ra5 * rb5$
MULTS64 d1, ra5, rb5	Multiply Word Signed	$d1 = ra5 \text{ (signed)} * rb5$
MULT64 d1, ra5, rb5	Word	$d1 = ra5 \text{ (unsigned)} * rb5$
MADDS64 d1, ra5, rb5	and Add Signed	$d1 = d1 + ra5 \text{ (signed)} * rb5$
MADD64 d1, ra5, rb5	and Add	$d1 = d1 + ra5 \text{ (unsigned)} * rb5$
MSUBS64 d1, ra5, rb5	Multiply and Subtract Signed	$d1 = d1 - ra5 \text{ (signed)} * rb5$
MSUB64 d1, ra5, rb5	Multiply and Subtract	$d1 = d1 - ra5 \text{ (unsigned)} * rb5$
MULT32 d1, ra5, rb5	Multiply Word	$d1.LO = ra5 * rb5$
MADD32 d1, ra5, rb5	Multiply and Add	$d1.LO = d1.LO + ra5 * rb5$
MSUB32 d1, ra5, rb5	Multiply and Subtract	$d1.LO = d1.LO - ra5 * rb5$
MFUSR rt5, USR	Move From User Special Register	$rt5 = USReg[USR]$
MTUSR rt5, USR	Move To User Special Register	$USReg[USR] = rt5$

表 3.1.2-6 除法指令

Mnemonic	Instruction	Operation
DIV Dt, ra5, rb5	Unsigned Integer Divide	$Dt.L = ra5 \text{ (unsigned)} / rb5$; $Dt.H = ra5 \text{ (unsigned)} \bmod rb5$;
DIVS Dt, ra5, rb5	Signed Integer Divide	$Dt.L = ra5 \text{ (signed)} / rb5$; $Dt.H = ra5 \text{ (signed)} \bmod rb5$;

表 3.1.2-7 乘法與除法指令 V2

Mnemonic	Instruction	Operation
MULR64 rt5, ra5, rb5	Multiply unsigned word to registers	$res = ra5 \text{ (unsigned)} * rb5$; if (PSW.BE == 1) { $(rt5_even, rt5_odd) = res$; } else { $(rt5_odd, rt5_even) = res$; }
MULSR64 rt5, ra5, rb5	Multiply signed word to registers	$res = ra5 \text{ (signed)} * rb5$; if (PSW.BE == 1) { $(rt5_even, rt5_odd) = res$; } else { $(rt5_odd, rt5_even) = res$; }
MADDR32 rt5, ra5, rb5	Multiply and add to 32-bit register	$res = ra5 * rb5$; $rt5 = rt5 + res(31,0)$;
MSUBR32 rt5, ra5, rb5	Multiply and subtract from 32-bit register	$res = ra5 * rb5$; $rt5 = rt5 - res(31,0)$;

Mnemonic	Instruction	Operation
DIVR rt5, rs5, ra5, rb5	Unsigned integer divide to registers	rt5 = Floor(ra5 (unsigned) / rb5); rs5 = ra5 (unsigned) mod rb5;
DIVSR rt5, rs5, ra5, rb5	Signed integer divide to registers	rt5 = Floor(ra5 (signed) / rb5); rs5 = ra5 (signed) mod rb5;

3.1.3 載入與儲存指令 (load and store)

Load/Store 屬於記憶體存取指令，負責 CPU 與屬於記憶體空間 (memory space) 元件之間資料移轉的指令，由於 AndesCore™ 將所有外部 I/O 元件皆規劃成記憶體應對式 I/O (memory mapped I/O)，因此 I/O 元件亦被視為記憶體一般存取。記憶體空間元件定址採用暫存器間接定址，也就是運算元的位址計算後置於暫存器。共有四種模式：

表 3.1.3-1 載入 / 儲存指令運算元定址模式

Mode	Operand Type	Index Left Shift (0-3 bits)	Before Increment with Base Update
1	Base Register + Immediate	No	No
2	Base Register + Immediate	No	Yes
3	Base Register + Register	Yes	No
4	Base Register + Register	Yes	Yes

在載入／儲存指令中 L：load (載入)，S：store (儲存)，W：word (字組)，H：halfword (辦字元組)，B：byte (位元組)，LWI rt5, [ra5 +(imm15s << 2)] 其中 (imm15s << 2) 代表編碼的方式，實際上寫程式碼時直接寫數值 (索引值) 即可，此一指令會先計算出來源運算元的記憶體位址 ra5 +(imm15s << 2)，在從屬於此位址的記憶體載入 4bytes 的資料到暫存器 rt5 中。表 3.1.3-2 與表 3.1.3-3 的差異在於運算元定址及該指令運算後 ra5 是否更新，表 3.1.3-2 的運算元位址在執行載入前先計算位址，執行後 ra5 保持不變，相反地表 3.1.3-3 中的指令，運算元的位址就在 ra5 中無須計算運算元位址直接執行載入，但執行載入後 ra5 的內容必須根據指令計算更新。表 3.1.3-4 與表 3.1.3-5 的差異如同表 3.1.3-2 與表 3.1.3-3 的差異，只是表 3.1.3-4 與表 3.1.3-5 中來源運算元的位址計算由 ra5、rb5 計算而得。

表 3.1.3-2 載入 / 儲存指令 (I)

Mnemonic	Instruction	Operaion
LWI rt5, [ra5 +(imm15s << 2)]	Load Word Immediate	address = ra5 + SE(imm15s << 2) rt5 = Word-memory(address)
LHI rt5, [ra5 + (imm15s << 1)]	Load Halfword Immediate	address = ra5 + SE(imm15s << 1) rt5 = ZE(Halfword-memory(address))
LHSI rt5, [ra5 +(imm15s << 1)]	Load Halfword Signed Immediate	address = ra5 + SE(imm15s << 1) rt5 = SE(Halfword-memory(address))
LBI rt5, [ra5 + imm15s]	Load Byte Immediate	address = ra5 + SE(imm15s) rt5 = ZE(Byte-memory(address))
LBSI rt5, [ra5 + imm15s]	Load Byte Signed	address = ra5 + SE(imm15s << 2) Word-memory(address) = rt5
SWI rt5, [ra5 + imm15s]	Store Word Immediate	address = ra5 + SE(imm15s << 2) Word-memory(address) = rt5
SHI rt5, [ra5 +(imm15s << 1)]	Store Halfword Immediate	address = ra5 + SE(imm15s << 1) Halfword-memory(address) = rt5[15:0]
SBI rt5, [ra5 +imm15s]	Store Byte Immediate	address = ra5 + SE(imm15s) Byte-memory(address) = rt5[7:0]

表 3.1.3-3 載入 / 儲存指令 (II)

Mnemonic	Instruction	Operaion
LWI.bi rt5, [ra5], (imm15s << 2)	Load Word Immediate with Post Increment	rt5 = Word-memory(ra5) ra5 = ra5 + SE(imm15s << 2)
LHI.bi rt5, [ra5], (imm15s << 1)	Load Halfword Immediate with Post Increment	rt5 = ZE(Halfword-memory(ra5)) ra5 = ra5 + SE(imm15s << 1)
LHSI.bi rt5, [ra5], (imm15s << 1)	Load Halfword Signed Immediate with Post Increment	rt5 = SE(Halfword-memory(ra5)) ra5 = ra5 + SE(imm15s << 1)
LBI.bi rt5, [ra5], imm15s	Load Byte Immediate with Post Increment	rt5 = ZE(Byte-memory(ra5)) ra5 = ra5 + SE(imm15s)
LBSI.bi rt5, [ra5], imm15s	Load Byte Signed Immediate with Post Increment	rt5 = SE(Byte-memory(ra5)) ra5 = ra5 + SE(imm15s)
SWI.bi rt5, [ra5], (imm15s << 2)	Store Word Immediate with Post Increment	Word-memory(ra5) = rt5 ra5 = ra5 + SE(imm15s << 2)
SHI.bi rt5, [ra5], (imm15s << 1)	Store Halfword Immediate with Post Increment	Halfword-memory(ra5) = rt5[15:0] ra5 = ra5 + SE(imm15s << 1)
SBI.bi rt5, [ra5], imm15s	Store Byte Immediate with Post Increment	Byte-memory(ra5) = rt5[7:0] ra5 = ra5 + SE(imm15s)

表 3.1.3-4 載入 / 儲存指令 (III)

Mnemonic	Instruction	Operation
LW rt5, [ra5 + (rb5 << sv)]	Load Word	address = ra5 + (rb5 << sv) rt5 = Word-memory(address)
LH rt5, [ra5 + (rb5 << sv)]	Load Halfword	address = ra5 + (rb5 << sv) rt5 = ZE(Halfword-memory(address))
LHS rt5, [ra5 + (rb5 << sv)]	Load Halfword Signed	address = ra5 + (rb5 << sv) rt5 = SE(Halfword-memory(address))
LB rt5, [ra5 + (rb5 << sv)]	Load Byte	address = ra5 + (rb5 << sv) rt5 = ZE(Byte-memory(address))
LBS rt5, [ra5 + (rb5 << sv)]	Load Byte Signed	address = ra5 + (rb5 << sv) rt5 = SE(Byte-memory(address))
SW rt5, [ra5 + (rb5 << sv)]	Store Word	address = ra5 + (rb5 << sv) Word-memory(address) = rt5
SH rt5, [ra5 + (rb5 << sv)]	Store Halfword	address = ra5 + (rb5 << sv) Halfword-memory(address) = rt5[15:0]
SB rt5, [ra5 + (rb5 << sv)]	Store Byte	address = ra5 + (rb5 << sv) Byte-memory(address) = rt5[7:0]

表 3.1.3-5 載入 / 儲存指令 (IV)

Mnemonic	Instruction	Operation
LW.bi rt5, [ra5], rb5<<sv	Load Word with Post Increment	rt5 = Word-memory(ra5) ra5 = ra5 + (rb5 << sv)
LH.bi rt5, [ra5], rb5<<sv	Load Halfword with Post Increment	rt5 = ZE(Halfword-memory(ra5)) ra5 = ra5 + (rb5 << sv)
LHS.bi rt5, [ra5], rb5<<sv	Load Halfword Signed with Post Increment	rt5 = SE(Halfword-memory(ra5)) ra5 = ra5 + (rb5 << sv)
LB.bi rt5, [ra5], rb5<<sv	Load Byte with Post Increment	rt5 = ZE(Byte-memory(ra5)) ra5 = ra5 + (rb5 << sv)
LBS.bi rt5, [ra5], rb5<<sv	Load Byte Signed with Post Increment	rt5 = SE(Byte-memory(ra5)) ra5 = ra5 + (rb5 << sv)
SW.bi rt5, [ra5], rb5<<sv	Store Word with Post Increment	Word-memory(ra5) = rt5 ra5 = ra5 + (rb5 << sv)
SH.bi rt5, [ra5], rb5<<sv	Store Halfword with Post Increment	Halfword-memory(ra5) = rt5[15:0] ra5 = ra5 + (rb5 << sv)
SB.bi rt5, [ra5], rb5<<sv	Store Byte with Post Increment	Byte-memory(ra5) = rt5[7:0] ra5 = ra5 + (rb5 << sv)

表 3.1.3-6 載入 / 儲存指令 (V2)

Mnemonic	Instruction	32-Bit Operation
LBi.gp rt5, [+ imm19s]	GP-implied Load unsigned Byte Immediate	address = gp + SE(imm19s) rt5 = ZE(Byte-memory(address))

LBSI.gp rt5, [+ imm19s]	GP-implied Load signed Byte Immediate	address = gp + SE(imm19s) rt5 = SE(Byte-memory(address))
LHI.gp rt5, [+ (imm18s << 1)]	GP-implied Load unsigned Halfword Immediate	address = gp + SE(imm18s << 1) rt5 = ZE(Halfword-memory(address))
LHSI.gp rt5, [+ (imm18s << 1)]	GP-implied Load signed Halfword Immediate	address = gp + SE(imm18s << 1) rt5 = SE(Halfword-memory(address))
LWI.gp rt5, [+ (imm17s << 2)]	GP-implied Load Word Immediate	address = gp + SE(imm17s << 2) rt5 = Word-memory(address)
SBI.gp rt5, [+ imm19s]	GP-implied Store Byte Immediate	address = gp + SE(imm19s) Byte-memory(address) = rt5[7:0]
SHI.gp rt5, [+ (imm18s << 1)]	GP-implied Store Halfword Immediate	address = gp + SE(imm18s << 1) Halfword-memory(address) = rt5[15:0]
SWI.gp rt5, [+ (imm17s << 2)]	GP-implied Store Word Immediate	address = gp + SE(imm17s << 2) Word-memory(address) = rt5
LMWA.{b a}{i d}{m?} rb5, [ra5], re5, Enable4	Load multiple word with alignment check	
SMWA.{b a}{i d}{m?} rb5, [ra5], re5, Enable4	Store multiple word with alignment check	
LBUP Rt, [Ra + (Rb << sv)]	Load Byte with User Privilege	Equivalent to LB instruction but with the user mode privilege address translation
SBUP Rt, [Ra + (Rb << sv)]	Store Byte with User Privilege	Equivalent to SB instruction but with the user mode privilege address translation. See page 346 for details

表 3.1.3-7 多重載入 / 儲存

Mnemonic	Instruction	Operation
LMW.{b a}{i d}{m?} Rb5, [Ra5], Re5, Enable4	Load Multiple Word (before/after; in/decrement; update/no-update base)	多重載入
SMW.{b a}{i d}{m?} Rb5, [Ra5], Re5, Enable4	Store Multiple Word (before/after; in/decrement; update/no-update base)	多重儲存

表 3.1.3-7 中 LMW 與 SMW 分別是把連續位置的記憶體上多筆資料載入通用暫存器或是將通用暫存器存入連續位置的記憶體中，其中使用的通用暫存器是由 Rb5、Re5 間連續編號的暫存器擔任來源暫存器 (SMW) 或目的暫存器 (LMW)，通用暫存器與記憶體間的映對原則是編號小的暫存器對應到低位址的記憶體，編號大的暫存器對應到高位址的記憶體，例如要將 R1 ~ R5 的暫存器內容存入位址 0x100000 ~ 0x100013 的記憶體中，就

是將 R1 放入 0x100000 ~ 0x100003 中，R2 置於 0x100004 ~ 0x100007……，R5 置於 0x100010 ~ 0x100013 中。記憶體位址由 [Ra5] 與 {b|a}{id} 組合計算得到第一個記憶體位址之後第二個記憶體位址由 {id} 決定，{b|a}{id} 組合得到的記憶體運算元關係如表 3.1.3-8。至於 {m?} 決定在完成指令執行後是否更新基底暫存器 Ra 的內容，如果未出現 m 則 Ra 的內容在指令執行前後不變，出現 m 則 Ra 的內容依 {b|a}{id} 決定內容，也就是說最後一個搬移記憶體或相鄰的記憶體位址會置入 Ra 中。表 3.1.3-7 中 Enable4 是一個 4 位元的數值，代表分別 R28 ~ R31 是否參與此一多重載入／儲存指令的運算。由表 3.1.3-9 可知若 Enable4=10=0b1010 表示 R28、R30 參與運算，以此類推。

表 3.1.3-8 多重載入 / 儲存記憶體位址計算

	i(increment)	d(decrement)
b(before)	[Ra], [Ra+4]....	[Ra], [Ra-4]....
a(after)	[Ra+4], [Ra+8].....	[Ra-4], [Ra-8].....

表 3.1.3-9 Enable4 值與暫存器的關係

Enable4	0b1000	0b0100	0b0010	0b0001
GPR	R28(fp)	R29(gp)	R30(lp)	R31(sp)

3.1.4 分叉指令 (Jump and Branch)

表 3.1.4-1 跳躍及回返指令

Mnemonic	Instruction	Operation
J imm24s	Jump	PC = PC + SE(imm24s << 1)
JAL imm24s	Jump and Link	LP = next sequential PC (PC + 4); PC = PC + SE(imm24s << 1)
JR rb5	Jump Register	PC = rb5
RET rb5	Return from Register	PC = rb5
JRAL rb5 JRAL rt5, rb5	Jump Register and Link	jaddr = rb5; LP = PC + 4; or rt5 = PC + 4; PC = jaddr;
請注意：imm24s 表示前後跳躍的範圍為±16M 的位址空間		

表 3.1.4-2 條件分叉指令

Mnemonic	Instruction	Operation
BEQ rt5, ra5, imm14s	Branch on Equal (2 Register)	$PC = (rt5 == ra5)? (PC + SE(imm14s \ll 1)) : (PC + 4)$
BNE rt5, ra5, imm14s	Branch on Not Equal (2 Register)	$PC = (rt5 \neq ra5)? (PC + SE(imm14s \ll 1)) : (PC + 4)$
BEQZ rt5, imm16s	Branch on Equal Zero	$PC = (rt5 == 0)? (PC + SE(imm16s \ll 1)) : (PC + 4)$
BNEZ rt5, imm16s	Branch on Not Equal Zero	$PC = (rt5 \neq 0)? (PC + SE(imm16s \ll 1)) : (PC + 4)$
BGEZ rt5, imm16s	Branch on Greater than or Equal to Zero	$PC = (rt5 \text{ (signed)} \geq 0)? (PC + SE(imm16s \ll 1)) : (PC + 4)$
BLTZ rt5, imm16s	Branch on Less than Zero	$PC = (rt5 \text{ (signed)} < 0)? (PC + \text{sign-ext}(imm16s \ll 1)) : (PC + 4)$
BGTZ rt5, imm16s	Branch on Greater than Zero	$PC = (rt5 \text{ (signed)} > 0)? (PC + SE(imm16s \ll 1)) : (PC + 4)$
BLEZ rt5, imm16s	Branch on Less than or Equal to Zero	$PC = (rt5 \text{ (signed)} \leq 0)? (PC + SE(imm16s \ll 1)) : (PC + 4)$
imm14s 表示前後跳躍的範圍為±16K 的位址空間，imm16s 表示前後跳躍的範圍為±64K 的位址空間		

表 3.1.4-3 條件分叉及回返指令

Mnemonic	Instruction	Operation
BGEZAL rt5, imm16s	Branch on Greater than or Equal to Zero and Link	LP = next sequential PC (PC + 4); $PC = (rt5 \text{ (signed)} \geq 0)? (PC + SE(imm16s \ll 1)), (PC + 4);$
BLTZAL rt5, imm16s	Branch on Less than Zero and Link	LP = next sequential PC (PC + 4); $PC = (rt5 \text{ (signed)} < 0)? (PC + SE(imm16s \ll 1)), (PC + 4);$

3.1.5 系統資源存取特權指令 (Privilege Resource Access Instructions)

存取系統資源暫存器前必須先將 CPU 的操作模式由使用者模式改成超級使用者模式，也就是將 PSW.POM 設為 0x1。

表 3.1.5-1 系統暫存器存取指令

Mnemonic	Instruction	Operation
MFSR rt5, SRIDX	Move from System Register	rt5 = SR[SRIDX]
MTSR rt5, SRIDX	Move to System Register	SR[SRIDX] = rt5

表 3.1.5-2 附帶更新系統暫存器的跳躍指令

Mnemonic	Instruction	Operation
JR.ITOFF rb5	Jump Register and Instruction Translation OFF	PC = rb5; PSW.IT = 0;
JR.TOFF rb5	Jump Register and Translation OFF	PC = rb5; PSW.IT = 0, PSW.DT = 0;
JRAL.ITON rb5 JRAL.ITON rt5, rb5	Jump Register and Link and Instruction Translation ON	jaddr = rb5; LP = PC+4 or rt5 = PC+4; PC = jaddr; PSW.IT = 1;
JRAL.TON rb5 JRAL.TON rt5, rb5	Jump Register and Link and Translation ON	jaddr = rb5;

3.1.6 其它 (Miscellaneous Instructions)

表 3.1.6-1 條件搬移指令

Mnemonic	Instruction	Operation
CMOVZ rt5, ra5, rb5	Conditional Move on Zero	rt5 = ra5 if (rb5 == 0)
CMOVN rt5, ra5, rb5	Conditional Move on Not Zero	rt5 = ra5 if (rb5 != 0)

表 3.1.6-2 空白運算指令

Mnemonic	Instruction	Operation
NOP (alias of SRLI R0, R0, 0)	No Operation	No Operation

表 3.1.6-3 的使用條件可經由表 3.1.6-4、表 3.1.6-5 的輔助得知是否需要使用。換句話說，使用表 3.1.6-4 第二欄的指令 (writer) 時，必須在第三欄指令使用前插入 DSB，以確保第一欄所列的暫存器或暫存器位元被讀取時，該暫存器已被第二欄指令更新。

表 3.1.6-3 依序執行指令

Mnemonic	Instruction	Operation
DSB	Data Serialization Barrier	
ISB	Instruction Serialization Barrier	

表 3.1.6-4 需要插入 DSB 指令的指令列表

State	Writer	Reader	Value
System register	MTSR	MFSR	New SR value
PSW.BE	SETEND	load/store	New endian state
PSW.DT	MTSR	load/store	New translation behavior
PSW.GIE/ SIM/ H5IM-H0IM	MTSR	following instructions	Interrupt behavior
PSW.GIE	SETGIE	following instructions	New global interrupt enable state
PSW.INTL	MTSR	following instructions	Interruption stack behavior
D\$ line valid	CCTL L1D (sub=invalid, WB&invalid, WR tag)	load/store	D\$ hit/miss
		CCTL L1D (sub=RD tag)	Tag valid
DTLB	TLBOP RWR/ RWLK/ TWR	load/store	Data PA
	TLBOP FLUA/INV	load/store	TLB miss
	TLBOP TWR	TLBOP TRD	TLB entry data
CACHE_CTL (DC_ENA)	MTSR	load/store	D\$ enable/disable behavior
DLMB (EN)	MTSR	load/store	Data local memory access behavior
L1_PPTB	MTSR	load/store	HPTWK behavior

表 3.1.6-5 需要插入 ISB 的指令表

State	Writer	Reader	Value
PSW.IT	MTSR	Instruction fetch	Instruction translation behavior
ITLB	TLBOP RWR/ RWLK/ TWR	Instruction fetch	Instruction PA
	TLBOP FLUA/ INV	Instruction fetch	TLB miss
I\$ line valid/data	ISYNC	Instruction fetch	New instruction data
I\$ line valid	CCTL L1I (sub=invalid)	Instruction fetch	I\$ miss
I\$ line valid/tag	CCTL L1I (sub=WR tag)	CCTL L1I (sub=Rd tag)	I\$ hit
I\$ line data	CCTL L1I (sub=WR word)	CCTL L1I (sub=RD word)	Instruction data
Memory	MSYNC	Instruction fetch	New instruction data
CACHE_CTL (IC_ENA)	MTSR	Instruction fetch	I\$ enable/disable behavior

State	Writer	Reader	Value
ILMB (EN)	MTSR	Instruction fetch	Instruction local memory access behavior
LI_PPTB	MTSR	Instruction fetch	HPTWK behavior
IVB	MTSR	Instruction fetch	Interruption behavior

中斷服務副程式結束時須以 IRET 指令進行回返中斷點，因為 IRET 指令的執行，才可將存放回返位址的 IPC 暫存器的內容放置於 PC 以及 IPSW 回存 PSW 暫存器，此外可視當時 INTL 的值決定回存哪些暫存器。

表 3.1.6-6 中斷回返指令

Mnemonic	Instruction	Operation
IRET	Interruption Return	Return from Interruption (exception or interrupt)

表 3.1.6-7 快取記憶體存取控制指令

Mnemonic	Instruction	Operation
CCTL	Cache Control	Read, write, and control cache states.

表 3.1.6-8 主要是設定與清除 PSW 暫存器的內容，第一列指令設定資料的儲存方式是屬於大端 (big endian) 或小端 (little endian) 模式；第二列指令設定整體終端致能位元是否允許中斷，也就是要取消中斷功能就把 GIE 位元清除，除了不可遮斷的中斷除外。

表 3.1.6-8 其它

Mnemonic	Instruction	Operation
SETEND.B SETEND.L	Atomic set or clear of PSW.BE bit	PSW.BE = 1; // SETEND.B PSW.BE = 0; // SETEND.L
SETGIE.E SETGIE.D	Atomic set or clear of PSW.GIE bit	PSW.GIE = 1; // SETGIE.E PSW.GIE = 0; // SETGIE.D
STANDBY	Wait for External Event	Enter standby state and wait for external event.

3.1.7 內建函式運算子 (function operator)

以下 3 個內建函式可用於所有組合語言指令：

1. hi20 (var) : 擷取變數 var 的位址中最高次 20 位元。
2. lo12 (var) : 擷取變數 var 的位址中最低次 12 位元。
3. sda (var) : 擷取變數 var 在小型資料節區 (small data area) 相對位移位址 (15 位元)。

3.1.8 虛擬指令 (pseudo instruction)

表 3.1.8-1 虛擬指令

指令	意義	運算
li rt5,imm_32	load 32-bit integer into register rt5.	sethi rt5,hi20(imm_32) and then ori rt5,reg,lo12(imm_32)
la rt5,var	load 32-bit address of var into register rt5.	sethi rt5,hi20(var) and then ori reg,rt5,lo12(var)
l.{bhw} rt5,var	load value of var into register rt5.	sethi \$ta,hi20(var) and then l{bhw}i rt5,[\$ta+lo12(var)]
l.{bh}s rt5,var	load value of var into register rt5.	sethi \$ta,hi20(var) and then l{bh}si rt5,[\$ta+lo12(var)]
l.{bhw}p rt5,var,inc	load value of var into register rt5 and increment \$ta by amount inc.	la \$ta,var and then l{bhw}i.bi rt5,[\$ta],inc
l.{bhw}pc rt5,inc	continue loading value of var into register rt5 and increment \$ta by amount inc.	l{bhw}i.bi rt5,[\$ta],inc.
l.{bh}sp rt5,var,inc	load value of var into register rt5 and increment \$ta by amount inc..	la \$ta,var and then l{bh}si.bi rt5,[\$ta],inc
l.{bh}spc rt5,inc	continue loading value of var into register rt5 and increment \$ta by amount inc.	l{bh}si.bi rt5,[\$ta],inc.
s.{bhw} rt5,var	store register rt5 to var.	sethi \$ta,hi20(var) and then s{bhw}i rt5,[\$ta+lo12(var)]
s.{bhw}p rt5,var,inc	store register rt5 to var and increment \$ta by amount inc.	la \$ta,var and then s{bhw}i.bi rt5,[\$ta],inc
s.{bhw}pc rt5,inc	continue storing register rt5 to var and increment \$ta by amount inc.	s{bhw}i.bi rt5,[\$ta],inc.
not rt5,ra5	1's complement of ra5	alias of nor rt5,ra5,ra5
neg rt5,ra5	2's complement of rt5	alias of subri rt5,ra5,0
br rb5	branch	alias of jr rb5
b label	branch to label	
bral rb5		alias of jral br5
bal fname		alias of jal fname
call fname	call function fname	same as "jal fname".
bgezal rt5,fname	if rt5>=0, call function fname	
bltzal rt5,fname	if rt5<0, call function fname	
move rt5,ra5		for 16-bit, this is mov55 rt5,ra5 , for no 16-bit, this is ori rt5,ra5,0
move rt5,var		same as l.w rt5,var
move rt5,imm_32		same as li rt5,imm_32
pushm ra5,rb5	push contents of registers from ra5 to rb5 into stack.	

指令	意義	運算
push ra5	push content of register ra5 into stack.	same as pushm ra5,ra5
push.d var	push value of double-word variable var into stack.	
push.w var	push value of word variable var into stack.	
push.h var	push value of half-word variable var into stack.	
push.b var	push value of byte variable var into stack.	
pusha var	push 32-bit address of variable var into stack.	
pushi imm_32	push 32-bit immediate value into stack.	
popm ra5,rb5	pop top of stack values into registers ra5 to rb5.	
pop rt5	pop top of stack value into register.	same as popm rt5,rt5
pop.d var,ra5	pop value of double-word variable var from stack using register ra5 as 2nd scratch register. (1st is \$ta)	
pop.w var,ra5	pop value of word variable var from stack using register ra5.	
pop.h var, ra5	pop value of half-word variable var from stack using register	
pop.b var,ra5	pop value of byte variable var from stack using register ra5.	

3.1.9 指引指令 (directive instruction)

在組合語言程式用以引導組譯器 (assembler) 組譯程式碼的指令稱之為指引指令或稱虛擬指令 (pseudo instruction)，在 AndeStar™ 分為 GNU 預設與 Andes 支援的指引指令，以下將列出部分常用指引指令：

表 3.1.9-1 指引指令

指引指令	意義
.data subsec	資料節區，subsec 為該節區識別碼或名稱 Default of subsec is 0, which is created automatically.
.text subsec	指令碼節區 Default of subsec is 0, which is created automatically.
.bss subsec	bss 節區 Default of subsec is 0, which is created automatically.
.section	使用自行定義的節區
.sdata_d	存放雙字組 (8 bytes) 型態的小資料節區 (small data area)

指引指令	意義
.sdata_w	存放單字組型態 (4 bytes) 的小資料節區
.sdata_h	存放半字組型態 (2 bytes) 的小資料節區
.sdata_b	存放位元組型態 (1 byte) 的小資料節區
.sbss_d	存放雙字組 (8 bytes) 型態的 bss 節區
.end	程式結束
.sbss_w	存放字元組 (4 bytes) 型態的 bss 節區
.sbss_h	存放半字組 (2 bytes) 型態的 bss 節區
.sbss_b	存放位元組 (1 byte) 型態的 bss 節區
.align type, fill, max	程式及資料存放對齊的方式，也就是存放的容量必然是 (2^{type}) 的倍數，當程式碼或資料的記憶體容量不是 (2^{type}) 的倍數時，將填入「 fill 」直到滿足 (2^{type}) 的倍數。
.ascii	定義字串
.asciz	定義字串但最後自動補上「 \0 」
.string string	同上
.byte	定義以位元組 (byte) 為單位的資料
.2byte	定義以半字元組 (2bytes) 為單位的資料
.half	定義以半字元組 (2bytes) 為單位的資料 (2bytes 對齊)
.short	定義以半字元組 (2bytes) 為單位的資料 (2bytes 對齊)
.4byte	定義以字元組 (4bytes) 為單位的資料
.word	定義以字元組 (4-bytes) 為單位的資料 (4bytes 對齊)
.int	定義以字元組 (4-bytes) 為單位的資料 (4bytes 對齊)
.long	定義以字元組 (4-bytes) 為單位的資料 (4bytes 對齊)
.single	定義以字元組 (4-bytes) 為單位的單精準浮點數資料 (4bytes 對齊)
.float	定義以字元組 (4-bytes) 為單位的單精準浮點數資料 (4bytes 對齊)
.8byte	定義以雙字元組 (8-bytes) 為單位的資料
.dword	定義以 2 字元組 (8-bytes) 為單位的資料 (8bytes 對齊)
.double	定義以 2 字元組 (8-bytes) 為單位的倍精準浮點數 (8bytes 對齊)
.qword	定義以 4 字元組 (16-bytes) 為單位的資料 (16bytes 對齊)
.octa	定義以 4 字元組 (16-bytes) 為單位的資料
.skip size,fill	「 fill 」代表要填入的值，將填入「 size 」bytes 的值「 fill 」
.space size,fill	同「 skip 」
.equ symbol,expr	定義「 symbol 」的值或內容為「 expr 」
.equiv symbol,expr	同上
.set symbol,expr	定義「 symbol 」的值或內容為「 expr 」
.little	資料以小端格式儲存
.big	資料以大端格式儲存

指引指令	意義
.if expr	條件組譯，「expr」為真則執行
.else	條件組譯
.elseif	條件組譯，「expr」為真則執行
.endif	結束條件組譯
.func symbol,label	把程式碼當成函式看待產生除錯相關資訊
.endfunc	for terminating a function.
.include file	引入檔案
.macro name,params	定義巨集指令，name 代表巨集名稱，params 代表
.endm	for terminating macro expansion.
.exitm	for exiting macro expansion.
.fill rept,size,value	for filling data chunk.
.float expr	for single precision floating data.
.func symbol,label	for issuing debugging information.
.global symbol	告訴組譯器「symbol」在此模組中定義
.globl symbol	same as .global.
.extern symbol	宣告「symbol」在外部程式模組中定義

3.2 AndesCore MCU 系列與相關工具鏈介紹

表 3.1.9-1 Andes MCU 系列 SoC 與使用的工具鏈

Toolchain Name	Andes Cores			
	N968A-S	N903-S(for 32GPRs)	N801-S	N705-S
nds32le-elf-[newlib mcuilib]-v2		◎		
nds32le-elf-[newlib mcuilib]-v2j		◎		
nds32le-elf-[newlib mcuilib]-v3	◎			
nds32le-elf-[newlib mcuilib]-v3m			◎	◎

表 3.1.9-1 中每列皆列出兩種版本工具鏈：newlib 與 mcuilib。這兩種工具鏈皆適用於無作業系統的微處理機系統中，但 mcuilib 版本可以產生更精簡的程式碼。表 3.1.9-1 中顯示 N801-S 使用 nds32le-elf-[newlib|mcuilib]-v3m 工具鏈，N903-S 可使用 nds32le-elf-[newlib|mcuilib]-v2、nds32le-elf-[newlib|mcuilib]-v2j 等工具鏈，但這是針對 32 GPR 的核心，至於 WT59F064 則只具備了 16GPR，因此只能夠使用 nds32le-elf-[newlib|mcuilib]-v2j 的工具鏈。

3.3 基本程式設計、編譯執行 (VEP) 與除錯

3.3.1 基本組合語言程式設計、編譯執行與除錯

基本上一個完整獨立的組合語言程式至少會有一個 `.text` 節區 (section)，常數部分可以直接置於 `.text` 區或者置於 `.data` 區，未具初值的變數會被置於 `.bss` 區。一個組合語言程式模組至少要有一個節區，「#」與「！」之後的文字符號都被視為註解，但「#」必須放在每一列的最前面第 1 行的位置上，「！」則不受此限制。程式中整數可用十進制、八進制 (0)、十六進制 (0x)、二進制 (0b) 等方式書寫，例如 128, #128, 0200, #0200, 0x80, #0x80, 0b10000000, #0b10000000 皆代表相同的整數。浮點數中「e」或「E」之後的數字代表指數部分 (exponent)，「f」或「F」代表單精準浮點數，「d」或「D」代表倍精準浮點數，例如 0f123.456 或 0d1.23456e2 代表相同的數值只是單精準浮點數與倍精準浮點數的差別罷了。

在程式碼中除了使用者自訂的標題符號有大、小寫的分別外，其餘不管是運算碼或是運算元大、小寫都視為相同的符號，比如說 JAL FUNC 與 jal FUNC 呼叫完全相同的副程式 FUNC，但與 JAL func 則呼叫不同的副程式「func」。以下將提供一些基本組合語言程式讓讀者熟悉 Andes 組合語言程式寫法。

■ ASM_HELLO 專案

此專案利用組合語言呼叫程式庫中「printf()」的函式，在「console」顯示字串「Andes Assembly Programming」，圖 3.3.1-1 是此專案的唯一程式碼且是組合語言程式，此一程式中用到了兩個節區：`.rodata` 與 `.text`，其中 `.rodata` 只存放未來顯示的字串常數，程式碼將不會寫入資料，所以使用只讀節區存放字串常數。在 `.rodata` 區中使用指引指令定義字串常數「Andes Assembly Programming」，並使用 `.LC0` 標籤變數 (label) 記錄字串常數的

起始位址，在第 16 行中透過「la \$r0, .LC0」虛擬指令將字串起始位址存入 GPR r0 中，當主程式利用第 17 行「bal printf」呼叫程式庫中程式「printf()」時，將字串常數的起始位址當成參數置於 r0 傳遞給「printf()」，順利將字串「Andes Assembly Programming」顯示於 console。

```

1  .section    .rodata
2  .align 2
3  .LC0:
4  .string "Andes Assembly Programming\n"
5  .text
6  .align 2
7  .globl main
8 main:
9  .LFB2:
10 movi    $r6, 0      !LOOP COUNTER
11
12 .L2:
13 move    $r7, $r6
14 sltsi   $r7, $r7, 5
15 beqz    $r7, .L3
16 la      $r0, .LC0    !the address of the shown string is placed in R0
17 bal     printf        !呼叫"printf()"
18 addi    $r6, $r6, 1  ! increase LOOP COUNTER
19 b       .L2
20 .L3:
21 ret

```

Annotations in the image:

- Line 1: `.section .rodata` is annotated with "因只定義字串常數所以放置於只讀資料節區".
- Line 4: `.string "Andes Assembly Programming\n"` is annotated with "LC0是代表字串起始位址的標題變數，將其位址置入r0，當成參數傳給'printf()'".
- Line 16: `la $r0, .LC0` is annotated with "the address of the shown string is placed in R0".
- Line 17: `bal printf` is annotated with "呼叫'printf()'".

圖 3.3.1-1 hello.S 程式碼

整個專案的建置步驟如下：

- Step 1.** 在 project explorer 視窗空白處按下右鍵出現選單，將滑鼠游標移到「New」(圖 3.3.1-2 ①)，自動出現另一選單，接著以滑鼠左鍵點擊「Project」(圖 3.3.1-2 ②)出現圖 3.3.1-3 新專案視窗，以滑鼠左鍵點選「Andes C Project」(圖 3.3.1-3 ①)再點擊「Next」(圖 3.3.1-3 ②)。繼續在圖 3.3.1-4 中依序輸入專案名稱 (①)，選擇以模擬器方式執行程式 (②)，選擇工作平台 (③)，選擇以空的專案進行開發 (④)，選擇適合的工具鏈 (⑤) 最後按「Next」繼續。

»» 微處理器應用與實作：C 語言與 Andes MCU 系列

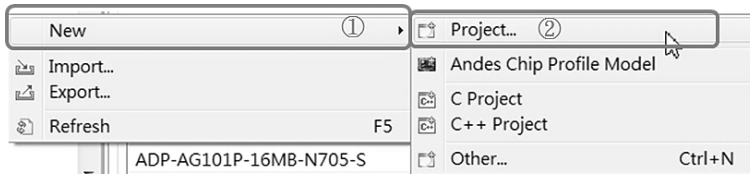


圖 3.3.1-2 開啟新專案

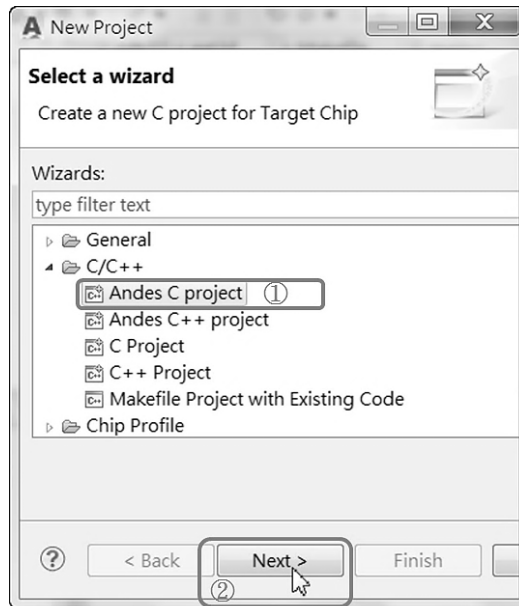




圖 3.3.1-3 建立新專案視窗

Step 2. 編輯程式碼。以滑鼠右鍵點擊新建專案名稱 (圖 3.3.1-5 ①)，出現下拉式選單後將滑鼠游標移到「New」(圖 3.3.1-5 ②)，自動出現另一選單，以滑鼠左鍵點擊「Source File」(圖 3.3.1-2 ③)，出現圖 3.3.1-6 新程式碼視窗，在圖 3.3.1-6 ① 輸入程式碼名稱「hello.S」，並繼續按「Finish」(圖 3.3.1-6 ②) 完成檔案建立，並於「Code Editor」中編輯程式碼 (圖 3.3.1-1)。

Step 3. 完成程式碼編輯後，以滑鼠左鍵點擊「」進行編譯連結建置完成執行檔「ASM_HELLO.adx」，再點擊「」執行專案，執行後於「Console」視窗中可得如圖 3.3.1-7 所示結果。

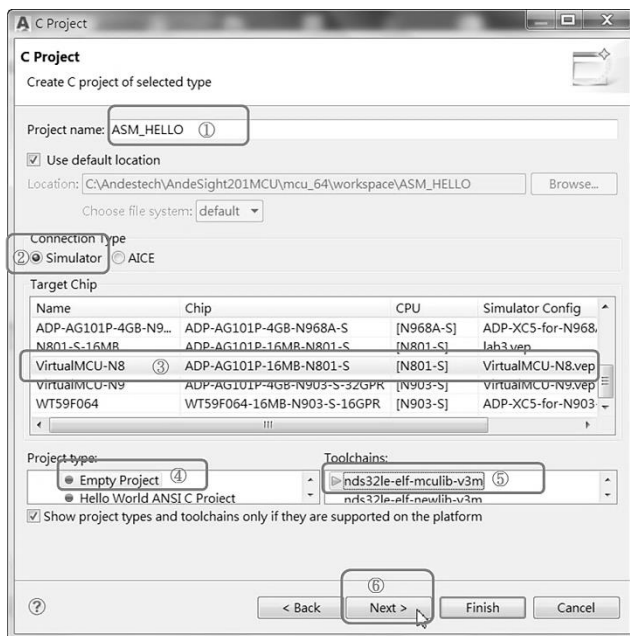


圖 3.3.1-4 輸入專案名稱選擇工作平台與工具鏈

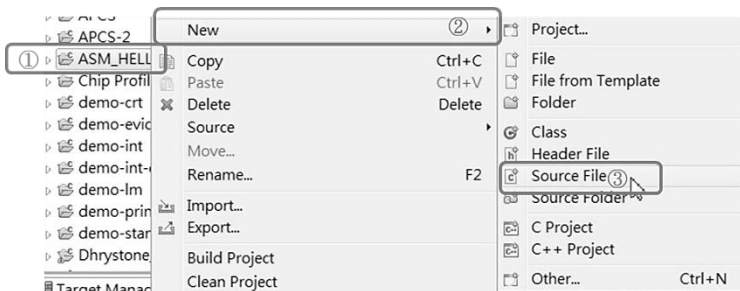


圖 3.3.1-5 新增程式

»» 微處理器應用與實作：C 語言與 Andes MCU 系列

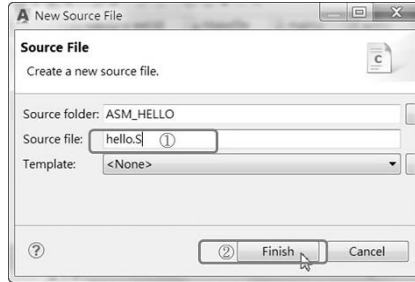


圖 3.3.1-6 輸入組合語言程式名稱

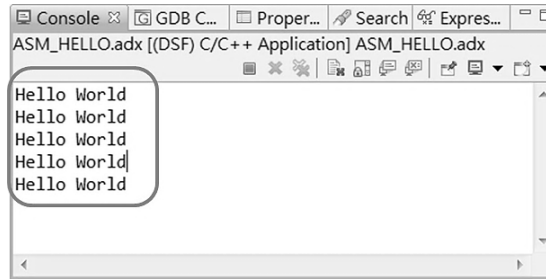



圖 3.3.1-7 「ASM_HELLO」專案執行結果

如第二章 2.2 節 Step 6. 的除錯步驟對此專案進行除錯，一般預設除錯開始第一個停止點在「main()」(圖 3.3.1-8)，接著在第 17 行與第 21 行設中斷點，然後點擊「」繼續執行程式，可以在「register」視窗中看到 r0 已填入常數字串的起始位址 0x300ef0 (圖 3.3.1-9 ①)，在「Memory Browser」將此數值填入圖 3.3.1-9 ③，可觀察到字串常數「Andes Assembly Programming」(圖 3.3.1-9 ④)。此時以單步執行方式執行「bal print」，完成第一次字串顯示(圖 3.3.1-10)，接著取消第 17 行的中斷點並點擊回復執行(resume)完成程式執行。

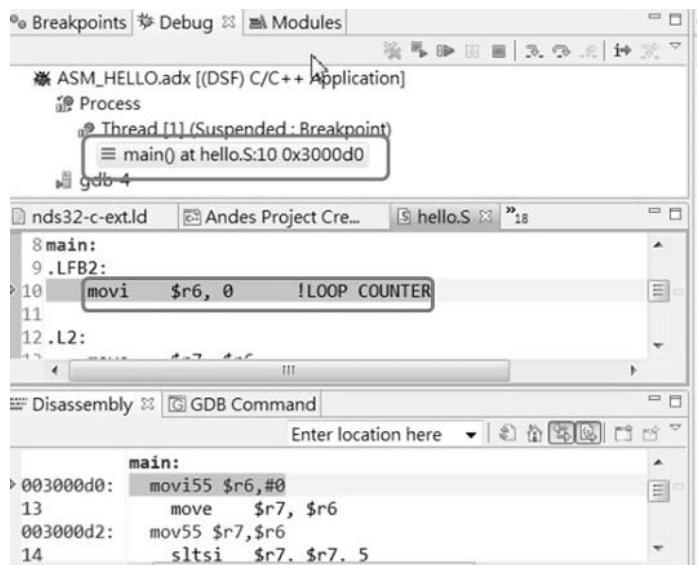


圖 3.3.1-8 除錯後第一個停止執行的指令

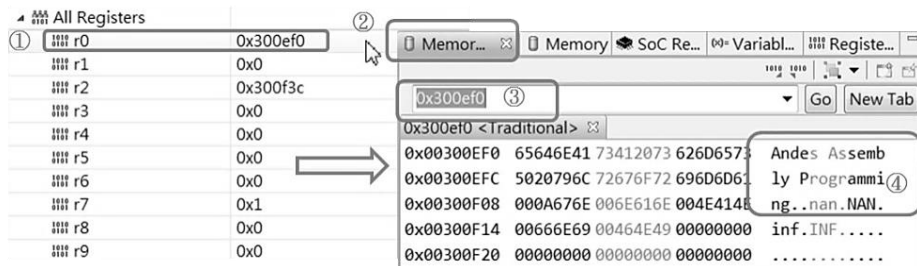


圖 3.3.1-9 觀察記憶體中的字串常數

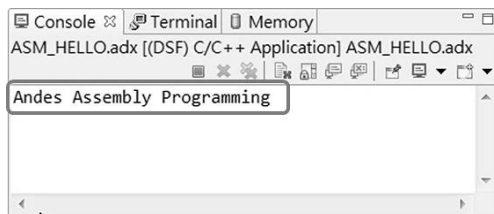


圖 3.3.1-10 第一次顯示字串

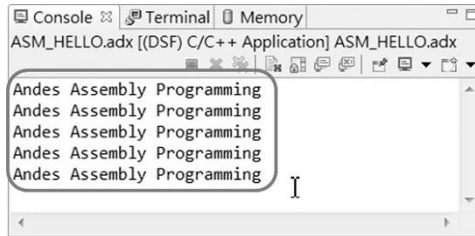


圖 3.3.1-11 專案執行完的結果

■ ASM_PSEUDO 專案

此專案主要目的在於展示虛擬指令「nop」、「li」、「la」、「move」、「b」、「br」、「bal」、「bral」、「call」、「l.w」、「neg」、「push.w」、「pop」等用途，此一程式執行後只出現一行「Error happens」，此字串的出現是因主程式呼叫副程式 sub2() 而產生的，此副程式執行時因定義變數 num1 時並未使用邊界對齊的指引指令導致該變數存放的起始位址並非 4 的倍數（圖 3.3.1-14），因此 r6-r7 不會等於 0，因而列印錯誤信息。如果加上圖 3.3.1-14 第 66 行指令（去除註解符號「！」）則此錯誤訊息將不會產生。

```
1 .section .rodata
2 .align 2
3 err_msg:
4 .string "Error Happens\n"
5 ! .data
6 ! .align 2
7 var1:
8 .word -42
9 var2:
10 .word 42
11 numlist:
12 .byte 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88
13
14 .text
15 .global main    主程式main共呼叫sub0、
16                sub1、sub2、與sub3等
17 main:          副程式
18     nop ! Do nothing: actually "mov r0,r0"
19     bal sub0    ! Call the first subroutine: "move"
20     bal sub1    ! Call the second subroutine: "li, la"
21     call sub2   ! Third subroutine: "l.w"
22     la $r6, sub3
23     bral $r6    ! Third subroutine: "ldr" and "adr"
24 exit: ret      ! Terminate the program
```

表示err_msg與var1的起始位址一定是4的倍數

圖 3.3.1-12 ASM_PSEUDO 專案 (I)

```

26 sub0:
27     move    $r6, 0x55aa          !r6=0x55aa
28     slti    $r6, $r6, 0x3fff     !r6=0, since 0x55aa>0x3fff
29     blez    $r6, 1f              !跳到本地標籤"1"的地方
30     la      $r0, err_msg
31     push    $lp                  !必須將回到main的位址存起來
32     bal     printf
33     pop     $lp
34 1:
35     ret
36
37 !Subroutine 1: Demonstrate the "li, la" pseudo-instruction
38 sub1:
39     li      $r6, #42             ! r6=42
40     la      $r7, var1            ! r7=0x300fa0
41     lw      $r8, [$r7]           ! r8=-42
42     add     $r8, $r8, $r6        ! r8=0
43     beqz    $r8, 1f              ! 往前跳到本地標籤"1"
44     la      $r0, err_msg
45     push    $lp
46     bal     printf
47     pop     $lp
48 1:
49     ret

```

驗證“li”、“la”的作用

圖 3.3.1-13 ASM_PSEUDO 專案 (II)

展示“l.w”、“neg”的作用

```

52 sub2:
53     move    $r6, #0x12345678    !r6=0x12345678
54     !load the value stored in num1 into r7
55     l.w     $r7, num1           !r7=0x56782078
56     neg     $r7, $r7            !r7=-1450713208
57     add     $r7, $r7, $r6
58     beqz    $r7, 1f
59 !load the address of the memory conatining the err_msg label
60     la      $r0, err_msg        !r0=0x300f90
61     push    $lp
62     bal     printf
63     pop     $lp
64 1:
65     ret
66     !.align 2
67 num1:
68     .word   0x12345678
69

```

0x300158
0x300158 <Traditional> 32
0x00300158 56782078 F03C1234 FF3AF4FF
0x00300164 6F3A3CBC 7D3C0498 7340F5FF
0x30015a
0x30015a <Traditional> 32
0x0030015a 12345678 F4FFFD3C 3CBCFF3A
0x00300166 04986F3A F5FF7D3C 00987340

圖 3.3.1-14 ASM_PSEUDO 專案 (III)

»» 微處理器應用與實作：C 語言與 Andes MCU 系列

```
70 ! Subroutine 3: Demonstrate "push.w" and "pop"
71 sub3:
72     push.w    var1      ! push var1 into stack
73     pop       $r6       ! r6 = -42
74     l.w       $r7, var2 ! r7 = 42
75     add       $r7, $r7, $r6
76     beqz      $r7, 123f !往前跳到本地標籤"123"
77     la        $r0, err_msg
78     push      $lp
79     bal       printf    展示"push.w"、"pop"的作用
80     pop       $lp
81 123:
82     ret
```

圖 3.3.1-15 ASM_PSEUDO 專案 (IV)

■ ASM_BLOCKCOPY 專案

此專案主要是練習以 LMW 及 SMW 的指令進行多重字組載入的動作，其中常數「num」代表要複製的字組個數，「src」、「dst」分別代表資料來源與目的地，於是使用「la」的虛擬指令將其位址置入「r1」與「r0」(圖 3.3.1-16 行 7、8)，r2 則放入要複製的個數「num」。 「lmw.bim \$r5, [\$r1], \$r8, 0」是從記憶體連續取四個字組 (word) 放入 \$r5 ~ \$r8，記憶體的位址是 r1 中的內容，且每取一個字組 r1 就加 4，換言之取出的資料是從 [r1]、~[r1+12] 連續 16 個 bytes 的資料，取出的資料依記憶體位址置入 r5 ~ r8，最高位址的資料是從 [r1+12] 取出因此置入 r8；最低位址的資料來自 [r1] 因此只入 r5，圖 3.3.1-16 第 11 行指令執行完 r1 的值會更新 (r1=r1+12)，複製到 r5 ~ r8 的字組則於圖 3.3.1-16 第 12 行指令中存放到記憶體中，該行指令使用「smw.bim \$r5, [\$r1], \$r8, 0」完成複製的動作。

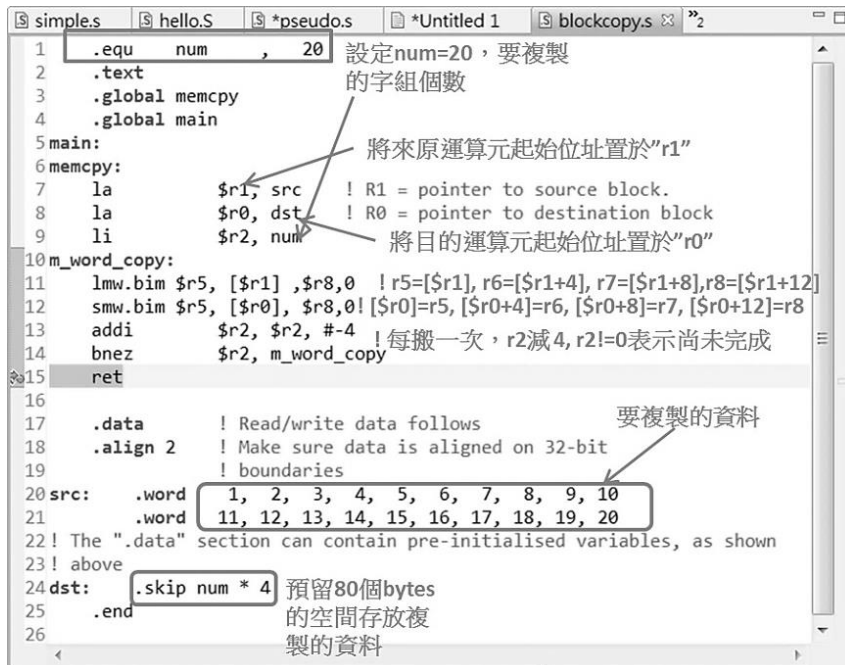


圖 3.3.1-16 ASM_BLOCKCOPY 專案

■ ASM_MACRO 專案

此專案旨於練習巨集的建置及使用，此外在此專案可以見到「.if」、「.else」、「.esleif」、「.endif」等指引指令的用法。事實上在 AndeStar™ 中的虛擬指令大都是以巨集完成的指令，圖 3.3.1-17 中第 1、11 行以 .macro 及 .endm 指引指令定義出 loadi_imm32 的巨集指令，「rt5」、「imm32」是該巨集的參數，該巨集在第 3 行中首先確認 imm32 的大小是否落於 -0x800000 ~ 0x7ffff 之間，若符合此一範圍則以「movi rt5, imm32」即可將 imm32 放置於 rt5，若 imm32 的值不在 -0x800000 ~ 0x7ffff 的範圍，則借助 lo12() 及 hi20() 函式運算子完成運算（圖 3.3.1-17 第 6、8、9 行）。

```

1  .macro load_imm32 rt5,imm32
2  !Copyright c 2006-2007 Andes Technology Corporation 33
3  .if ((imm32 <= 0x7ffff) && (imm32 >= -0x80000))
4      movi rt5,imm32
5  .elseif (imm32 & 0x00000fff == 0x0)
6      sethi rt5,hi20(imm32)
7  .else
8      sethi rt5,hi20(imm32)
9      ori rt5,rt5,lo12(imm32)
10 .endif
11 .endm
12 .global main
13 .text
14 main:
15 ! both sethi and ori
16     load_imm32 $r20,#0x12345678 ! ok
17 ! sethi only
18     load_imm32 $r22,-0x12345000 ! ok
19 ! movi only
20     load_imm32 $r0,#0xffff ! ok
21 end:
22     ret
23 .end
    
```

巨集的名稱

巨集的參數

取出imm32的bit31~12置於rt5的bit31~12，rt5中bit11~0被清除為0

取出imm32的bit11~0與rt5進行or運算，也就是將imm32的bit11~0置入rt5的bit11~0

圖 3.3.1-17 ASM_MACRO 專案

3.3.2 基本 C 語言程式設計、編譯執行與除錯

AndeStar™ 程式中使用的基本資料格式有整數、浮點數與指標 (pointer)，所占用的位元組容量如表 3.3.2-1 所示。表 3.3.2-2 列出在 C 語言程式中慣用的資料型態對應於 AndeStar™ 中使用的資料格式，寫程式時必須參照表 3.3.2-2 以了解使用的變數或常數在 Andes 平台所占用的記憶體容量。

表 3.3.2-1 AndeStar™ 基本資料格式的容量與邊界對齊容量

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

Class	Machine Type	Size(in Byte)	Alignment (in Byte)
Floating Point	Unsigned double word	8	8
	Signed double word	8	8
	Single precision (IEEE754)	4	4
	Double precision (IEEE754)	8	8
Pointer	Instruction Pointer	4	4
	Data Pointer	4	4

表 3.3.2-2 C/C++ 程式使用的基本資料格式與 AndeStart™ 基本資料格式對照

C/C++ Type	Machine Type
[signed] char	Signed byte
unsigned char	Unsigned byte
[signed] short	Signed half word
unsigned short	Unsigned half word
[signed] int	Signed word
unsigned int	Unsigned word
[signed] long	Signed word
unsigned long	Unsigned word
[signed] long long (C99)	Signed double word
unsigned long long (C99)	Unsigned double word
size_t (C99)	Unsigned word
float	Single precision (IEEE 754)
double	Double precision (IEEE 754)
long double	Double precision (IEEE 754)
float _Complex (C99)	Two Single precision (IEEE 754)
double _Complex (C99)	Two Double precision (IEEE 754)
long double _Complex (C99)	Two Double precision (IEEE 754)
float	Single precision (IEEE 754)
double	Double precision (IEEE 754)

本書第四章到第九章中使用的範例中除了少數硬體初始化的部分動用組合語言外絕大多數使用 C 語言完成，對多數 C 語言程式設計的使用者而言不會有困難，只需要注意的部分是定義輸入變數時要注意使用「violate」的關鍵字加以限制，要求編譯不要以優化的方式處理，以免造成錯誤。例如以下程式片段中 PAD_PD 代表某一 GPIO 接腳，程式必須處理等待輸入信號變為高電位時才得以繼續執行，編譯器編譯程式時發現 PAD_PD 未曾改變，因此經過優化（optimization）後編譯如圖 3.3.2-1 右方，明顯地程式已悖離原意，因此必須在變數前加上「violate」以避免程式優化造成錯誤。



圖 3.3.2-1 程式優化

3.3.3 Andes 程式呼叫介面 (APCS：Andes Procedure Call Standard)

Andes ABI (Application Binary Interface) 定義應用與組合語言程式介面的架構，由原來傳統的 ABI (又稱 ABI 1) 演進到 ABI 2 與 ABI 2fp，以下將以 ABI 1 與 ABI 2 兩種架構為中心簡易說明之。首先以較簡易的副程式執行完畢傳回執行結果說明：

- 回傳執行結果：
 - 傳回值如果是一般基本型態 (如表 3.1-1) 且小於或等於 4bytes，回傳的資料置於「\$r0」。
 - 傳回值如果是一般基本型態 (如表 3.1-1) 但大於 4bytes，回傳的資料將置於「\$r0」與「\$r1」。
 - 傳回值如果不是一般基本型態 (如表 3.1-1) 且小於或等於 8bytes，回傳的資料將如上述兩種方式處理。
 - 傳回值如果不是一般基本型態 (如表 3.1-1) 但大於 8bytes，回傳的資料將以額外的參數回傳該回傳值的記憶體位址 (以傳址方式處理)。如果將回傳值置於記憶體時，「\$r0」會置入該記憶體位址，並將傳回值的第一筆資料置於「\$r1」。

圖 3.3.3-1 是呼叫副程式時主程式除將參數透過 r0 ~ r5 傳遞給副程式外，也可將其他參數置於堆疊區 (Caller's Frame incoming arguments) 中傳送給副程式，在副程式中除了將存放回返位址的 \$lp 存放於堆疊區 (Current Frame locals and temporaries) 外，必要時也會將主程式的 \$fp 存放於此，此外有些需要存放的 GPR 也存放於 register save area。

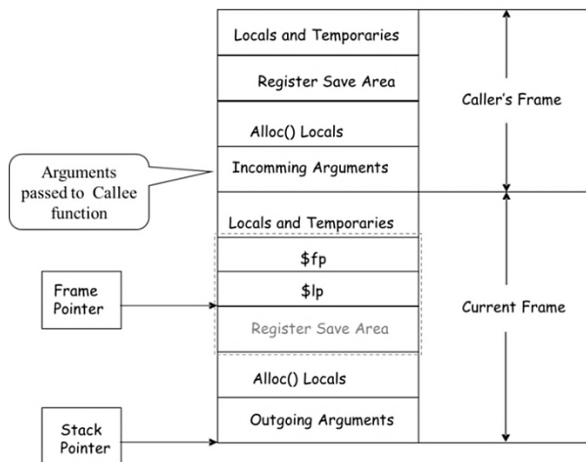


圖 3.3.3-1 Andes 呼叫副程式傳統參數傳遞

- 參數傳遞 (ABI 1)

- 參數傳遞以 $r0 \sim r5$ 為主，若超過 6 個參數，其餘的就透過堆疊區傳送。
- 若參數小於 4bytes 仍以 4bytes 為單位傳送參數，也就是資料位址邊界必須是 4 的倍數；如果參數是以 8 個 bytes 為邊界對齊 (alignment) 則資料位址邊界必須是 8 的倍數。
- 主程式必須幫副程式準備 24bytes 的堆疊空間讓副程式儲存放置在 $\$r0 \sim \$r5$ 的參數。
- 如果傳遞的參數是不定長度的參數則以堆疊區傳遞，否則如傳統的傳遞方式以 $\$r0 \sim \$r5$ 為主，不足的部分以堆疊區傳遞。
- 一旦以堆疊區傳遞參數，則後續參數皆以堆疊區傳遞。

- 參數傳遞 (ABI 2)

- 如果傳遞的參數是不定長度的參數則以傳址的方式 (call by reference) 傳遞參數。
- 副程式自行準備複製 $\$r0 \sim \$r5$ 的空間。
- 預留 $\$r24$ 、 $\$r25$ 不使用。

3.4 進階 C 語言程式設計

3.4.1 進階中斷服務副程式 (interrupt service routine)

(for V3m, V3f, V3j and V3 ISA Only, Excluding SN801)

一般中斷服務副程式必須以組合語言程式完成，至少在前半段的中斷服務副程式必須使用到系統暫存器以及建置中斷向量表 (interrupt vector table) 等，而一般高階程式語言是無法直接使用系統暫存器及建置中斷向量表的，但是自從 V3m、V3f、V3j 和 V3 等指令集開發後，其編譯器已經支援了無需使用組合語言也能完成中斷服務副程式及中斷向量表的建置，因此往後 Andes 系統的開發者在系統開發上可更有效率地完成任務或在維護上更加簡易。上述指令集的編譯器會根據 NDS32ATTR_RESET() 的參數宣告及系統暫存器主動設置中斷向量表，經由 NDS32ATTR_RESET() 宣告的函式就是系統開機後第一個要執行的程式碼，也可以稱之為系統重置服務程式。

系統重置服務函式宣告方法如下 (斜體字表示使用者自行命名或依系統決定):

- void NDS32ATTR_RESET("*<option_list>*") *reset_hdlr*(void) 或
- void *reset_hdlr*(void) __attribute__((reset("*<option_list>*"))))

參數選項有三項 (可有可無):

1. vectors=XXX，所有中斷數目 (含 exception、interrupt、software interrupt)。
2. nmi_func=YYY，不可遮罩中斷服務副程式名稱 (NMI)。
3. warm_func=ZZZ，一般重置中斷服務副程式名稱 (warm reset)。

無論是 cold reset (電源重置)、一般重置 (warm reset) 或不可遮罩中斷 (NMI) 重置對中斷而言都是使用 vecor_0 進入點 (entry point) (詳見第九章) 因此當發生重置時系統必須根據產生中斷的原因決定是否執行 nmi_func 或 warm_func。此二中斷服務副程式稍後說明如何宣告編輯。Andes

的範例目錄 (Andestech\AndeSight201MCU\demo\startup) 中提供一個以 C 語言編輯的中斷服務副程式的範例「 demo-int-c-ext.tgz 」, 其中的系統重置服務函式「 reset_handler(void) 」置於「 init-default.c 」, 該程式定義如下：

```
void NNDS32ATTR_RESET("vectors=16;i_func=nmi_handler;warm_func=nmi_handler")
reset_handler(void)
{
    __cpu_init(); //初始化 CPU 參數，設定快取、系統暫存器等參數
    __c_init(); //複製資料到.data 區及初始化.bss 區等
    __soc_init();//初始化周邊元件暫存器，此時可開始使用整體與靜態變數。
    main(); 開始執行主程式
}
```

實際上在「 reset_handler() 」完全使用 C 語言指令編輯程式碼，要注意的只是在完成 .data 區與 .bss 區的資料設定後才可開始使用整體與靜態變數，以上是系統重置服務程式的編輯，nmi_func 與 warm_func 的宣告定義為：int nmi_func(int *reg_ptr)及 int warm_func(int *reg_ptr)，當執行 NMI 或一般重置副程式時系統會主動將 GPR 儲存於緩衝區並將起始位址當成參數放置於指標變數 reg_ptr。

中斷除系統重置中斷外還需為各種例外中斷 (exception) 與中斷 (interrupt) 設置中斷服務副程式，以下針對中斷與例外處理服務副程式各別說明：

- 中斷服務副程式宣告與定義

根據系統進入執行中斷服務副程式前儲存的暫存器個數分成兩種格式：

1. 只存 Caller_Regs；2. 儲存所有 GPR。

1. 只存 Caller_Regs (詳見表 3.1-1)

```
void intr_hdlr(int vid)
```

```
NDS32ATTR_ISR(“id=xxx[/save_caller_regs;<is_nested>]”)
```

「 id 」代表中斷編號，例如編輯軟體中斷服務副程式就必須使用「 6 」。

2. 儲存所有 GPR

```
void intr_hdlr(int vid, NDS32_CONTEXT *ptr) NDS32ATTR_ISR
```

(“id=xxx [;save_all_regs;<is_nested>]”)

save_all_regs 代表系統在進入中斷副程式前會先將所有的 GPR 存放到堆疊區，堆疊區記憶體佈建如圖 3.4.1-1 所示，其中 NDS32_CONTEXT 是一結構體資料。

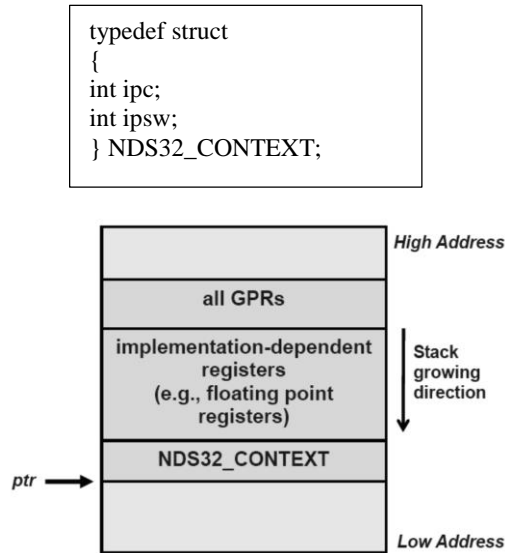


圖 3.4.1-1 中斷服務副程式堆疊區資料佈建

上述中斷服務副程式宣告格式中的參數意義如下說明：

- intr_hdlr：由使用者命名的中斷服務副程式名稱
- vid：中斷編號
- ptr：指向 NDS32_CONTEXT 的指標變數
- id=中斷編號(0 ~ 63)，多個中斷可共用一個副程式，如 vid=0,1,2
- <is_nested>：可被中斷，形成巢串中斷
- <not_nested>：不可被中斷
- <ready_nested>：在中斷服務副程式可以自行設定 PSWGIE 成可被中斷
- <critical>：不可被中斷

以下幾個宣告設定皆屬相同的中斷服務副程式的宣告，三個中斷編號 0、1、2 的服務副程式皆使用相同的中斷服務副程式 `timer_hdlr()`。

```
void timer_hdlr(int vid) NDS32ATTR_ISR("id=0,1,2;save_caller_regs;nested");
void timer_hdlr(int vid) NDS32ATTR_ISR("id=0,1,2;nested;save_caller_regs");
void timer_hdlr(int vid) NDS32ATTR_ISR("id=0,1,2")。
```

- 例外處理服務副程式宣告與定義

如中斷服務副程式的定義與宣告一樣分成兩類：

1. 只存 `Caller_Regs` (詳見表 3.1-1)

```
void except_hdlr(int vid)
NDS32ATTR_EXCEPT("id=xxx[;save_caller_regs;<is_nested>]")
```

「id」代表例外 (exception) 編號，例如編輯系統中斷 (system call) 服務副程式就必須使用「8」。

2. 儲存所有 GPR

```
void except_hdlr(int vid NDS32_CONTEXT *ptr) NDS32ATTR_EXCEPT
("id=xxx[;save_all_regs;<is_nested>]")；
```

讀者在使用系統呼叫 (syscall) 時，請於例外處理副程式結束回反前將 `NDS_CONTEXT->ipc+4`。

3.4.2 Andes 本質 (intrinsic function) 函式

雖說以函式命名並以函式格式呼叫，但本質函式並非以一般函式方式執行，而是透過編譯器直接將指令置入本質函式所在位置，如此一來不僅可節省一般函式執行因 overhead 導致一般效能的差異外，也可讓使用者免去寫組合語言程式的不便。在此列出常用的本質函式與其對應使用的指令，其餘請讀者自行參閱「Andes Programming Guide」。

表 3.4.2-1 載入 / 儲存本質函式

Intrinsic Function Syntax	Mapped ANDES Instruction
unsigned int _nds32_llw (unsigned int *a)	LLW
char _nds32_lbup(unsigned char *a)	LBUP
unsigned int _nds32_lwup (unsigned int *a)	LWUP
unsigned int _nds32_scw (unsigned int *a, unsigned int b)	SCW
void _nds32_sbup (unsigned char *a, char b)	SBUP
void _nds32_swup (unsigned int *a, unsigned int b)	SWUP

表 3.4.2-2 讀寫系統 / 使用者特殊暫存器讀寫本質函式

Intrinsic Function Syntax	Mapped ANDES Instruction
unsigned int __nds32_mfsr (const enum nds32_sr sname)	mfsr
unsigned int __nds32_mfsr (const enum nds32_usr usname)	mfusr
Void __nds32_mtsr(unsigned int val, const enum nds32_sname)	mtsr
void nds32_mtusr (unsigned int val, const enum nds32_usname)	mtusr

表 3.4.2-3 中斷本質函式

Intrinsic Function Syntax	Mapped ANDES Instruction
void nds32_enable_int(enum nds32_intrinsic int id)	
void __nds32_disable_int(enum nds32_intrinsic int id)	
void nds32_gie_dis()	
void nds32_gie_en()	
void nds32_set_pending_swint()	
void nds32_clr_pending_swint()	
unsigned int nds32_get_all_pending_int()	MFSR
unsigned int nds32_get_pending_int(enum nds32_intrinsic int id)	
void nds32_set_int_priority(enum nds32_intrinsic int_id, unsigned int prio)	
unsigned int nds32_get_int_priority(enum nds32_intrinsic int id)	
void nds32_setgie_en()	SETGIE
void nds32_setgie_dis()	SETGIE

表 3.4.2-4 其他

Intrinsic Function Syntax	Mapped ANDES Instruction
void __nds32__isb()	ISB
void __nds32__dsb()	DSB
void __nds32__nop()	NOP
void __nds32__trap(const unsigned int swid)	TRAP
void __nds32__setend_little()	SETEND
void __nds32__setend_little()	SETEND