7. Application Binary Interface

NII51016-8.1.0

This chapter describes the Application Binary Interface (ABI) for the Nios® II processor. The ABI describes:

- How data is arranged in memory
- Behavior and structure of the stack
- Function calling conventions

This chapter contains the following sections:

- "Data Types" on page 7–1
- "Memory Alignment" on page 7–1
- "Register Usage" on page 7–2
- "Stacks" on page 7–3
- "Arguments and Return Values" on page 7–6
- "Relocation" on page 7–8

Data Types

Table 7–1 shows the size and representation of the C/C++ data types for the Nios II processor.

Table 7–1. Representation of Data Types

Type	Size (Bytes)	Representation
char, signed char	1	two's complement (ASCII)
unsigned char	1	binary (ASCII)
short, signed short	2	two's complement
unsigned short	2	binary
int, signed int	4	two's complement
unsigned int	4	binary
long, signed long	4	two's complement
unsigned long	4	binary
float	4	IEEE
double	8	IEEE
pointer	4	binary
long long	8	two's complement
unsigned long long	8	binary

Memory Alignment

Contents in memory are aligned as follows:

- A function must be aligned to a minimum of 32-bit boundary.
- The minimum alignment of a data element is its natural size. A data element larger than 32 bits need only be aligned to a 32-bit boundary.
- Structures, unions, and strings must be aligned to a minimum of 32 bits.
- Bit fields inside structures are always 32-bit aligned.

Register Usage

The ABI adds additional usage conventions to the Nios II register file defined in the *Programming Model* chapter of the *Nios II Processor Reference Handbook*. The ABI uses the registers as shown in Table 7–2.

Table 7–2. Nios II ABI Register Usage (Part 1 of 2)

Register	Name	Used by Compiler	Callee Saved *(1)*	Normal Usage
r0	zero	✓		0x00000000
r1	at			Assembler temporary
r2		✓		Return value (least-significant 32 bits)
r3		✓		Return value (most-significant 32 bits)
r4		✓		Register arguments (first 32 bits)
r5		✓		Register arguments (second 32 bits)
r6		✓		Register arguments (third 32 bits)
r7		✓		Register arguments (fourth 32 bits)
r8		✓		Caller-saved general-purpose registers
r9		✓		
r10		✓		
r11		✓		
r12		✓		
r13		✓		
r14		✓		
r15		✓		
r16		✓	✓	Callee-saved general-purpose registers
r17		✓	✓	
r18		✓	✓	
r19		✓	✓	
r20		✓	✓	
r21		✓	✓	
r22		✓	✓	
r23		✓	✓	
r24	et			Exception temporary
r25	bt			Break temporary
r26	gp	✓		Global pointer
r27	sp	✓		Stack pointer

Table 7–2. Nios II ABI Register Usage (Part 2 of 2)

Register	Name	Used by Compiler	Callee Saved *(1)*	Normal Usage
`r28`	`fp`	✓		Frame pointer *(2)*
`r29`	`ea`			Exception return address
`r30`	`ba`			Break return address
`r31`	`ra`	✓		Return address

Notes to Table 7–2:

(1) A function can use one of these registers if it saves it first. The function must restore the register's original value before exiting.

(2) If the frame pointer is not used, the register is available as a temporary register. Refer to "Frame Pointer Elimination" on page 7–4.

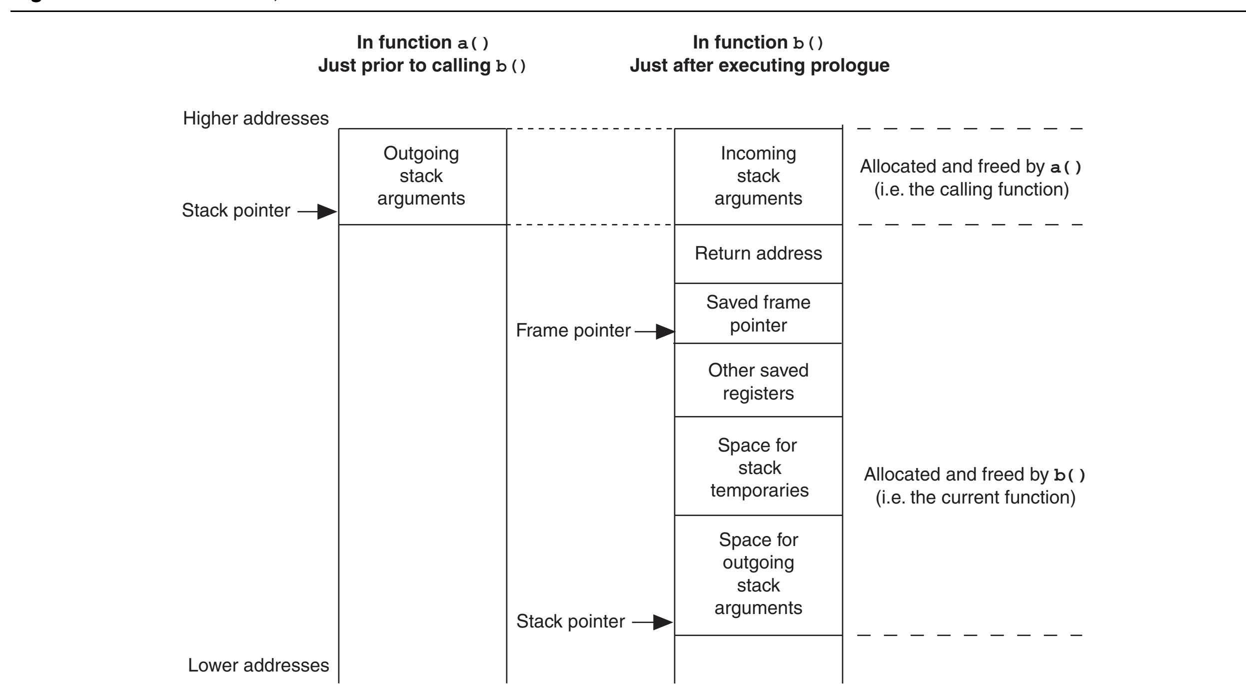
The endianness of values greater than 8 bits is little endian. The upper 8 bits of a value are stored at the higher byte address.

Stacks

The stack grows downward (i.e. towards lower addresses). The stack pointer points to the last used slot. The frame pointer points to the saved frame pointer near the top of the stack frame.

Figure 7–1 shows an example of the structure of a current frame. In this case, function `a()` calls function `b()`, and the stack is shown before the call and after the prologue in the called function has completed.

Figure 7–1. Stack Pointer, Frame Pointer and the Current Frame



Each section of the current frame is aligned to a 32-bit boundary. The ABI requires the stack pointer be 32-bit aligned at all times.

Frame Pointer Elimination

The frame pointer is provided for debugger support. If you are not using a debugger, you can optimize your code by eliminating the frame pointer, using the `-fomit-frame-pointer` compiler option. When the frame pointer is eliminated, register fp is available as a temporary register.

Call Saved Registers

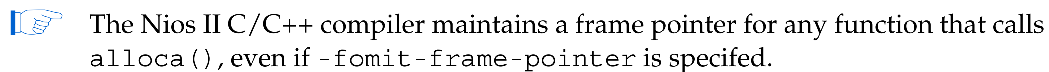
The compiler is responsible for saving registers that need to be saved in a function. If there are any such registers, they are saved on the stack, from high to low addresses, in the following order: `ra`, `fp`, `r2`, `r3`, `r4`, `r5`, `r6`, `r7`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`, `r16`, `r17`, `r18`, `r19`, `r20`, `r21`, `r22`, `r23`, `r24`, `r25`, `gp`, and `sp`. Stack space is not allocated for registers that are not saved.

Further Examples of Stacks

There are a number of special cases for stack layout, which are described in this section.

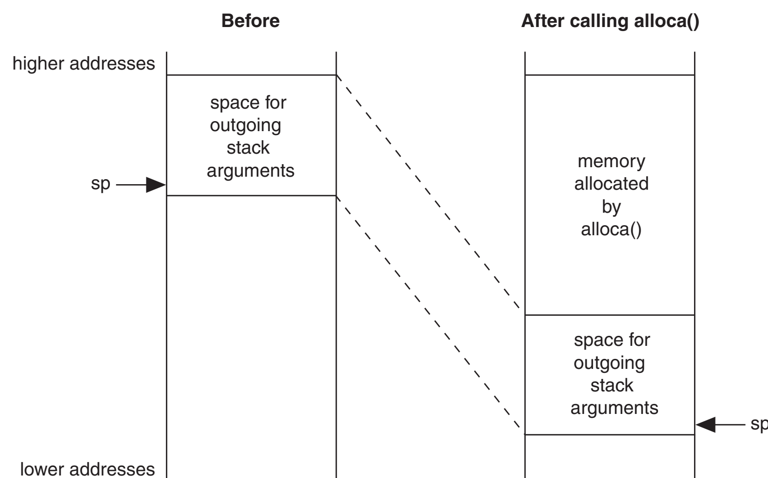
Stack Frame for a Function With alloca()

The Nios II stack frame implementation provides support for the `alloca()` function, defined in the Berkeley Software Distribution (BSD) extension to C, and implemented by the gcc compiler. Figure 7–2 depicts what the frame looks like after `alloca()` is called. The space allocated by `alloca()` replaces the outgoing arguments and the outgoing arguments get new space allocated at the bottom of the frame.



The Nios II C/C++ compiler maintains a frame pointer for any function that calls `alloca()`, even if `-fomit-frame-pointer` is specifed.

Figure 7–2. Stack Frame after Calling alloca()

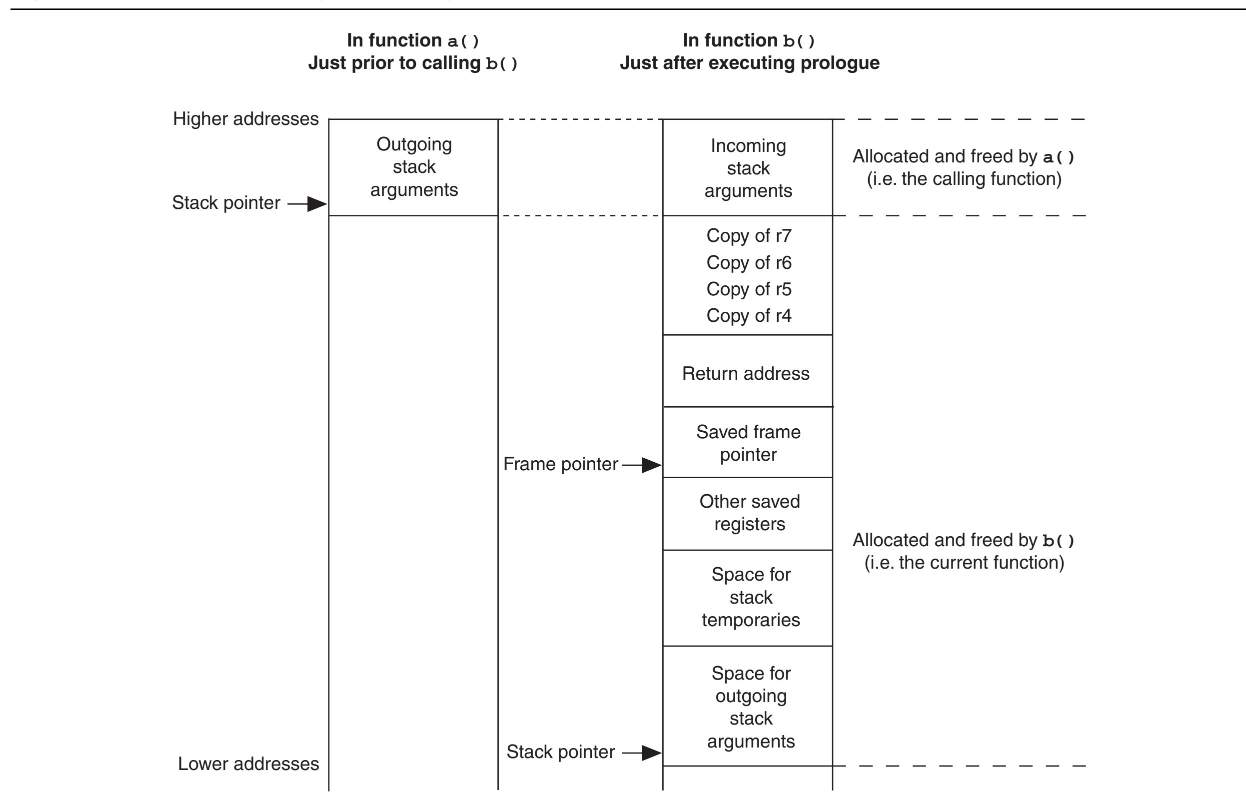


Stack Frame for a Function with Variable Arguments

Functions that take variable arguments (`varargs`) still have their first 16 bytes of arguments arriving in registers `r4` through `r7`, just like other functions.

In order for `varargs` to work, functions that take variable arguments allocate 16 extra bytes of storage on the stack. They copy to the stack the first 16 bytes of their arguments from registers `r4` through `r7` as shown in Figure 7–3.

Figure 7–3. Stack Frame Using Variable Arguments



Stack Frame for a Function with Structures Passed By Value

Functions that take `struct` value arguments still have their first 16 bytes of arguments arriving in registers `r4` through `r7`, just like other functions.

If part of a structure is passed using registers, the function might need to copy the register contents back to the stack. This operation is similar to that required in the variable arguments case as shown in Figure 7–3.

Function Prologues

The Nios II C/C++ compiler generates function prologues that allocate the stack frame of a function for storage of stack temporaries and outgoing arguments. In addition, each prologue is responsible for saving the state of the calling function. This entails saving certain registers on the stack. These registers, the callee-saved registers, are listed in Table 7–2 on page 7–2. A function prologue is required to save a callee-saved register only if the function uses the register.

Given the function prologue algorithm, when doing a back trace, a debugger can disassemble instructions and reconstruct the processor state of the calling function.

An even better way to find out what the prologue has done is to use information stored in the DWARF2 debugging fields of the executable and linkable format (**.elf**) file.

The instructions found in a Nios II function prologue perform the following tasks:

- Adjust the stack pointer (to allocate the frame)
- Store registers to the frame
- Set the frame pointer to the location of the saved frame pointer

Example 7–1 shows a function prologue.

Example 7–1. A function prologue

```
/* Adjust the stack pointer */
addi    sp, sp, -16   /* make a 16-byte frame */

/* Store registers to the frame */
stw     ra, 12(sp)    /* store the return address */
stw     fp, 8(sp)     /* store the frame pointer*/
stw     r16, 4(sp)    /* store callee-saved register */
stw     r17, 0(sp)    /* store callee-saved register */

/* Set the new frame pointer */
addi    fp, sp, 8
```

Prologue Variations

The following variations can occur in a prologue:

- If the function's frame size is greater than 32,767 bytes, extra temporary registers are used in the calculation of the new stack pointer as well as for the offsets of where to store callee-saved registers. The extra registers are needed because of the maximum size of immediate values allowed by the Nios II processor.
- If the frame pointer is not in use, the final instruction, recalculating the frame pointer, is not generated.
- If variable arguments are used, extra instructions store the argument registers on the stack.
- If the compiler designates the function as a leaf function, the return address is not saved.
- If optimizations are on, especially instruction scheduling, the order of the instructions might change and become interlaced with instructions located after the prologue.

Arguments and Return Values

This section discusses the details of passing arguments to functions and returning values from functions.

Arguments

The first 16 bytes to a function are passed in registers `r4` through `r7`. The arguments are passed as if a structure containing the types of the arguments were constructed, and the first 16 bytes of the structure are located in `r4` through `r7`.

A simple example:

```
int function (int a, int b);
```

The equivalent structure representing the arguments is:

```
struct { int a; int b; };
```

The first 16 bytes of the `struct` are assigned to `r4` through `r7`. Therefore `r4` is assigned the value of *a* and `r5` the value of *b*.

The first 16 bytes to a function taking variable arguments are passed the same way as a function not taking variable arguments. The called function must clean up the stack as necessary to support the variable arguments. Refer to "Stack Frame for a Function with Variable Arguments" on page 7–4.

Return Values

Return values of types up to 8 bytes are returned in `r2` and `r3`. For return values greater than 8 bytes, the caller must allocate memory for the result and must pass the address of the result memory as a hidden zero argument.

The hidden zero argument is best explained through an example.

Example 7–2. Returned struct

```
/* b() computes a structure-type result and returns it */
STRUCT b(int i, int j)
{
    ...
    return result;
}

void a(...)
{
    ...
    value = b(i, j);
}
```

In Example 7–2, if the result type is no larger than 8 bytes, `b()` returns its result in `r2` and `r3`.

If the return type is larger than 8 bytes, the Nios II C/C++ compiler treats this program as if `a()` had passed a pointer to `b()`. Example 7–3 shows how the Nios II C/C++ compiler sees the code in Example 7–2.

Example 7–3. Returned struct is Larger than 8 Bytes

```
void b(STRUCT *p_result, int i, int j)
{
    ...
    *p_result = result;
}

void a(...)
{
    STRUCT value;
    ...
    b(*value, i, j);
}
```

Relocation

In a Nios II object file, each relocatable address reference possesses a relocation type. The relocation type specifies how to calculate the relocated address.Table 7–3 lists the calculation for address relocation for each Nios II relocation type. The bit mask specifies where the address is found in the instruction.

Table 7–3. Nios II Relocation Calculation (Part 1 of 2)

Name	Value	Overflow check *(1)*	Relocated Address R *(2)*	Bit Mask M	Bit Shift B
R_NIOS2_NONE	0	n/a	None	n/a	n/a
R_NIOS2_S16	1	Yes	S + A	0x003FFFC0	6
R_NIOS2_U16	2	Yes	S + A	0x003FFFC0	6
R_NIOS2_PCREL16	3	Yes	((S + A) – 4) - PC	0x003FFFC0	6
R_NIOS2_CALL26	4	No	(S + A) >> 2	0xFFFFFFC0	6
R_NIOS2_IMM5	5	Yes	(S + A) & 0x1F	0x000007C0	6
R_NIOS2_CACHE_OPX	6	Yes	(S + A) & 0x1F	0x07C00000	22
R_NIOS2_IMM6	7	Yes	(S + A) & 0x3F	0x00000FC0	6
R_NIOS2_IMM8	8	Yes	(S + A) & 0xFF	0x00003FC0	6
R_NIOS2_HI16	9	No	((S + A) >> 16) & 0xFFFF	0x003FFFC0	6
R_NIOS2_LO16	10	No	(S + A) & 0xFFFF	0x003FFFC0	6
R_NIOS2_HIADJ16	11	No	((((S+A) >> 16) & 0xFFFF) + (((S+A) >> 15) & 0x1)) & 0xFFFF	0x003FFFC0	6
R_NIOS2_BFD_RELOC_32	12	No	S + A	0xFFFFFFFF	0
R_NIOS2_BFD_RELOC_16	13	Yes	(S + A) & 0xFFFF	0x0000FFFF	0
R_NIOS2_BFD_RELOC_8	14	Yes	(S + A) & 0xFF	0x000000FF	0
R_NIOS2_GPREL	15	No	(S + A - GP) & 0xFFFF	0x003FFFC0	6
R_NIOS2_GNU_VTINHERIT	16	n/a	None	n/a	n/a
R_NIOS2_GNU_VTENTRY	17	n/a	None	n/a	n/a
R_NIOS2_UJMP	18	No	((S + A) >> 16) & 0xFFFF, (S + A + 4) & 0xFFFF	0x003FFFC0	6

Table 7–3. Nios II Relocation Calculation (Part 2 of 2)

Name	Value	Overflow check (1)	Relocated Address R (2)	Bit Mask M	Bit Shift B
R_NIOS2_CJMP	19	No	((S + A) >> 16) & 0xFFFF, (S + A + 4) & 0xFFFF	0x003FFFC0	6
R_NIOS2_CALLR	20	No	((S + A) >> 16) & 0xFFFF) (S + A + 4) & 0xFFFF	0x003FFFC0	6
R_NIOS2_ALIGN	21	n/a	None	n/a	n/a
R_NIOS2_ILLEGAL	22	n/a	None	n/a	n/a

Notes to Table 7–3:

(1) For relocation types where no overflow check is performed, the relocated address is truncated to fit the instruction.

(2) S: Symbol address, A: Addend, PC: Program counter, GP: Global pointer

With the information in Table 7–3, any Nios II instruction can be relocated by manipulating it as an unsigned 32-bit integer, as follows:

```
Xr = (( R << B ) & M | ( X & ~M ));
```

where:

- R is the relocated address, calculated as shown in Table 7–3
- B is the bit shift shown in Table 7–3
- M is the bit mask shown in Table 7–3
- X is the original instruction
- Xr is the relocated instruction

Validated Relocation Types

The Nios II C/C++ compiler generates and uses a subset of the available relocation types. The following five types are used frequently and have been thoroughly validated:

- R_NIOS2_HIADJ16
- R_NIOS2_LO16
- R_NIOS2_CALL26
- R_NIOS2_GPREL
- R_NIOS2_BFD_RELOC32

Other relocation types are not supported.

Referenced Documents

This chapter references the following documents:

- *Programming Model* chapter of the *Nios II Processor Reference Handook*

Document Revision History

Table 7–4 shows the revision history for this document.

Table 7–4. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2008 v8.1.0	Maintenance release.	—
May 2008 v8.0.0	Frame pointer description updated Relocation table added	Frame pointer implementation redefined
October 2007 v7.2.0	Maintenance release.	—
May 2007 v7.1.0	■ Added table of contents to Introduction section. ■ Added Referenced Documents section.	—
March 2007 v7.0.0	Maintenance release.	—
November 2006 v6.1.0	Maintenance release.	—
May 2006 v6.0.0	Maintenance release.	—
October 2005 v5.1.0	Maintenance release.	—
May 2005 v5.0.0	Maintenance release.	—
May 2004 v1.0	Initial release.	—

 © November 2008 Altera Corporation