7. Application Binary Interface

NII51016-11.0.0

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

Table 7–1 lists 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 listed 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		✓	*(2)*	
r23		✓	*(3)*	
r24	et			Exception temporary

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

Register	Name	Used by Compiler	Callee Saved *(1)*	Normal Usage
r25	bt			Break temporary
r26	gp	✓		Global pointer
r27	sp	✓		Stack pointer
r28	fp	✓	*(4)*	Frame pointer
r29	ea			Exception return address
r30	ba			■ Normal register set: Break return address ■ Shadow register sets: SSTATUS register
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) In the GNU Linux operating system, r22 points to the global offset table (GOT). Otherwise, it is available as a callee-saved general-purpose register.

(3) In the GNU Linux operating system, r23 is used as the thread pointer. Otherwise, it is available as a callee-saved general-purpose register.

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

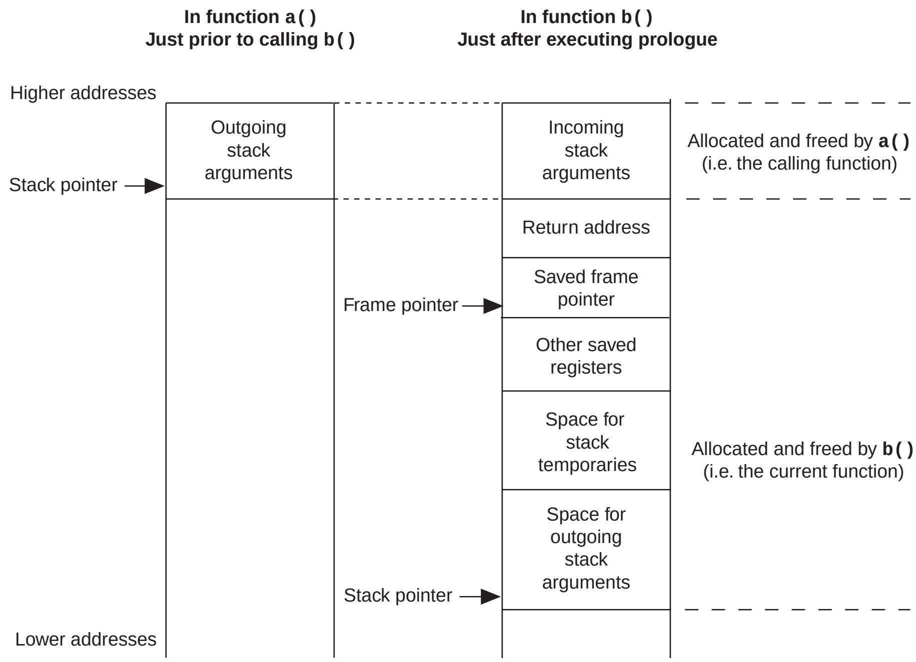
The endianness of values greater than 8 bits is little endian. The upper 8 bits of a value are stored at the higher byte address.

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

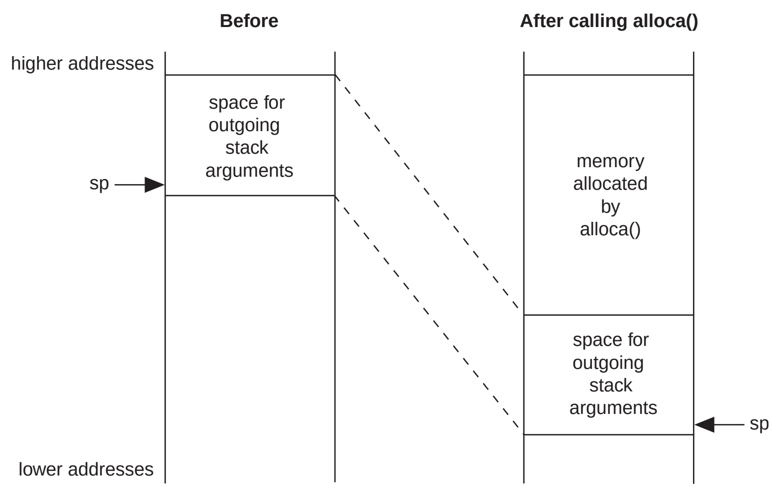
There are a number of special cases for stack layout, which are described in this section.

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 spec if ed

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

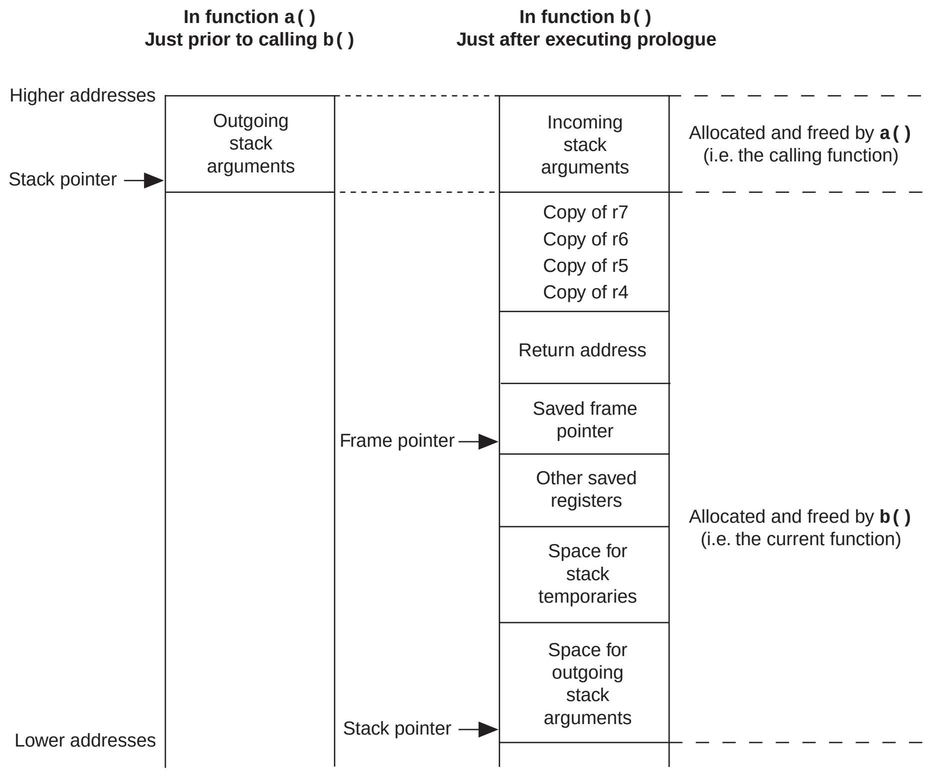


Stack Frame for a Function with Variable Arguments

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 DWARF-2 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 7–5.

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);
}
```

DWARF-2 Definition

Registers `r0` through `r31` are assigned numbers 0 through 31 in all DWARF-2 debugging sections.

Object Files

Nios II object file headers contain Nios II-specific values as listed in Table 7–3.

Table 7–3. Nios II-Specific ELF Header Values

Member	Value
`e_ident[EI_CLASS]`	`ELFCLASS32`
`e_ident[EI_DATA]`	`ELFDATA2LSB`
`e_machine`	`EM_ALTERA_NIOS2` == 113

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 7–4 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 7–4. Nios II Relocation Calculation (Part 1 of 3)

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

Table 7–4. Nios II Relocation Calculation (Part 2 of 3)

Name	Value	Overflow check *(1)*	Relocated Address R *(2)*	Bit Mask M	Bit Shift B
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	Adj(S+A)	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
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_GOT16 *(3)*	22	Yes	G	0x003FFFC0	6
R_NIOS2_CALL16 *(3)*	23	Yes	G	0x003FFFC0	6
R_NIOS2_GOTOFF_LO *(3)*	24	No	(S + A – GOT) & 0xFFFF	0x003FFFC0	6
R_NIOS2_GOTOFF_HA *(3)*	25	No	Adj (S + A – GOT)	0x003FFFC0	6
R_NIOS2_PCREL_LO *(3)*	26	No	(S + A – PC) & 0xFFFF	0x003FFFC0	6
R_NIOS2_PCREL_HA *(3)*	27	No	Adj (S + A – PC)	0x003FFFC0	6
R_NIOS2_TLS_GD16 *(3)*	28	Yes	Refer to "Thread-Local Storage" on page 7–13	0x003FFFC0	6
R_NIOS2_TLS_LDM16 *(3)*	29	Yes	Refer to "Thread-Local Storage" on page 7–13	0x003FFFC0	6
R_NIOS2_TLS_LDO16 *(3)*	30	Yes	Refer to "Thread-Local Storage" on page 7–13	0x003FFFC0	6
R_NIOS2_TLS_IE16 *(3)*	31	Yes	Refer to "Thread-Local Storage" on page 7–13	0x003FFFC0	6
R_NIOS2_TLS_LE16 *(3)*	32	Yes	Refer to "Thread-Local Storage" on page 7–13	0x003FFFC0	6
R_NIOS2_TLS_DTPMOD *(3)*	33	No	Refer to "Thread-Local Storage" on page 7–13	0xFFFFFFFF	0
R_NIOS2_TLS_DTPREL *(3)*	34	No	Refer to "Thread-Local Storage" on page 7–13	0xFFFFFFFF	0

Table 7–4. Nios II Relocation Calculation (Part 3 of 3)

Name	Value	Overflow check *(1)*	Relocated Address R *(2)*	Bit Mask M	Bit Shift B
R_NIOS2_TLS_TPREL *(3)*	35	No	Refer to "Thread-Local Storage" on page 7–13	0xFFFFFFFF	0
R_NIOS2_COPY *(3)*	36	No	Refer to "Copy Relocation" on page 7–13	n/a	n/a
R_NIOS2_GLOB_DAT *(3)*	37	No	S	0xFFFFFFFF	0
R_NIOS2_JUMP_SLOT *(3)*	38	No	Refer to "Jump Slot Relocation" on page 7–13	0xFFFFFFFF	0
R_NIOS2_RELATIVE *(3)*	39	No	BA+A	0xFFFFFFFF	0
R_NIOS2_GOTOFF *(3)*	40	No	S+A	0xFFFFFFFF	0

Notes to Table 7–4:

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

(2) Expressions in this column use the following conventions:

- S: Symbol address
- A: Addend
- PC: Program counter
- GP: Global pointer
- Adj(X): (((X >> 16) & 0xFFFF) + ((X >> 15) & 0x1)) & 0xFFFF
- BA: The base address at which a shared library is loaded
- GOT: The value of the Global Offset Table (GOT) pointer (Linux only)
- G: The offset into the GOT for the GOT slot for symbol S (Linux only)

(3) Relocation support is provided for Linux systems.

With the information in Table 7–4, 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 listed in Table 7–4
- B is the bit shift listed in Table 7–4
- M is the bit mask listed in Table 7–4
- X is the original instruction
- Xr is the relocated instruction

ABI for Linux Systems

This section describes details specific to Linux systems beyond the Linux-specific information in Table 7–2 on page 7–2 and Table 7–4 on page 7–9.

Linux Toolchain Relocation Information

Dynamic relocations can appear in the runtime relocation sections of executables and shared objects, but never appear in object files (with the exception of R_NIOS2_TLS_DTPREL, which is used for debug information). No other relocations are dynamic. The dynamic relocations are listed in Table 7–5.

Table 7–5. Dynamic Relocations

R_NIOS2_TLS_DTPMOD
R_NIOS2_TLS_DTPREL
R_NIOS2_TLS_TPREL
R_NIOS2_COPY
R_NIOS2_GLOB_DAT
R_NIOS2_JUMP_SLOT
R_NIOS2_RELATIVE

A global offset table (GOT) entry referenced using R_NIOS2_GOT16 must be resolved at load time. A GOT entry referenced only using R_NIOS2_CALL16 can initially refer to a procedure linkage table (PLT) entry and then be resolved lazily.

Because the GOT-relative and TP-relative relocations are 16-bit relocations, no single object file can require more than 64 KB of GOT and no dynamic object using local dynamic or local executable thread-local storage (TLS) can have more than 64 KB of TLS data. New relocations might be added to support this in the future.

Several new assembler operators are defined to generate the Linux-specific relocations, as listed in Table 7–6.

Table 7–6.

Relocation	Operator
R_NIOS2_GOT16	`%got`
R_NIOS2_CALL16	`%call`
R_NIOS2_GOTOFF_LO	`%gotoff_hiadj`
R_NIOS2_GOTOFF_HA	`%gotoff_lo`
R_NIOS2_PCREL_LO	`%hiadj`
R_NIOS2_PCREL_HA	`%lo`
R_NIOS2_TLS_GD16	`%tls_gd`
R_NIOS2_TLS_LDM16	`%tls_ldm`
R_NIOS2_TLS_LDO16	`%tls_ldo`
R_NIOS2_TLS_IE16	`%tls_ie`
R_NIOS2_TLS_LE16	`%tls_le`
R_NIOS2_TLS_DTPREL	`%tls_ldo`
R_NIOS2_GOTOFF	`%gotoff`

The %hiadj and %lo operators generate PC-relative or non-PC-relative relocations, depending whether the expression being relocated is PC-relative. For instance, %hiadj(_gp_got - .) generates R_NIOS2_PCREL_HA. %tls_ldo generates R_NIOS2_TLS_LDO16 when used as an immediate operand, and R_NIOS2_TLS_DTPREL when used with the .word directive.

Copy Relocation

The R_NIOS2_COPY relocation is used to mark variables allocated in the executable that are defined in a shared library. The variable's initial value is copied from the shared library to the relocated location.

Jump Slot Relocation

Jump slot relocations are used for the PLT. For information about the PLT, refer to "Procedure Linkage Table" on page 7–19.

Thread-Local Storage

The Nios II processor uses the Variant I model for thread-local storage. The end of the thread control block (TCB) is located 0x7000 bytes before the thread pointer. The TCB is eight bytes long. The first word is the dynamic thread pointer (DTV) pointer and the second word is reserved. Each module's dynamic thread pointer is biased by 0x8000 (when retrieved using __tls_get_addr). The thread library can store additional private information before the TCB.

In the GNU Linux toolchain, the GOT pointer (_gp_got) is always kept in r22, and the thread pointer is always kept in r23.

In the following examples, any registers can be used, except that the argument to __tls_get_addr is always passed in r4 and its return value is always returned in r2. Calls to __tls_get_addr must use the normal position-independent code (PIC) calling convention in PIC code; these sequences are for example only, and the compiler might generate different sequences. No linker relaxations are defined.

Example 7–4 shows the general dynamic model.

Example 7–4. General Dynamic Model

```
addi   r4, r22, %tls_gd(x)       # R_NIOS2_TLS_GD16 x
call   __tls_get_addr            # R_NIOS2_CALL26 __tls_get_addr
# Address of x in r2
```

In the general dynamic model, a two-word GOT slot is allocated for x, as shown in Example 7–5.

Example 7–5. GOT Slot for General Dynamic Model

```
GOT[n]                            R_NIOS2_TLS_DTPMOD x
GOT[n+1]                          R_NIOS2_TLS_DTPREL x
```

Example 7–6 shows the local dynamic model.

Example 7–6. Local Dynamic Model

```
addi   r4, r22, %tls_ldm(x)     # R_NIOS2_TLS_LDM16 x
call   __tls_get_addr           # R_NIOS2_CALL26 __tls_get_addr
addi   r5, r2, %tls_ldo(x)      # R_NIOS2_TLS_LDO16 x
# Address of x in r5
ldw    r6, %tls_ldo(x2)(r2)     # R_NIOS2_TLS_LDO16 x2
# Value of x2 in r6
```

One 2-word GOT slot is allocated for all R_NIOS2_TLS_LDM16 operations in the linked object. Any thread-local symbol in this object can be used, as shown in Example 7–7.

Example 7–7. GOT Slot with Thread-Local Storage

```
GOT[n]                             R_NIOS2_TLS_DTPMOD x
GOT[n+1]                           0
```

Example 7–8 shows the initial exec model.

Example 7–8. Initial Exec Model

```
ldw    r4, %tls_ie(x)(r22)     # R_NIOS2_TLS_IE16 x
add    r4, r23, r4
# Address of x in r4
```

A single GOT slot is allocated to hold the offset of x from the thread pointer, as shown in Example 7–9.

Example 7–9. GOT Slot for Initial Exec Model

```
GOT[n]                             R_NIOS2_TLS_TPREL x
```

Example 7–10 shows the local exec model.

Example 7–10. Local Exec Model

```
addi   r4, r23, %tls_le(x)      # R_NIOS2_TLS_LE16 x
# Address of x in r4
```

There is no GOT slot associated with the local exec model.

Debug information uses the GNU extension DW_OP_GNU_push_tls_address, as shown in Example 7–11.

Example 7–11. Debug Information

```
.byte 0x03                      # DW_OP_addr
.word %tls_ldo(x)               # R_NIOS2_TLS_DTPREL x
.byte 0xe0                      # DW_OP_GNU_push_tls_address
```

Linux Function Calls

Register `r23` is reserved for the thread pointer on GNU Linux systems. It is initialized by the C library and it may be used directly for TLS access, but not modified. On non-Linux systems `r23` is a general-purpose, callee-saved register.

The global pointer, `r26` or `gp`, is globally fixed. It is initialized in startup code and always valid on entry to a function. This method does not allow for multiple `gp` values, so `gp`-relative data references are only possible in the main application (that is, from position dependent code). `gp` is only used for small data access, not GOT access, because code compiled as PIC may be used from shared libraries. The linker may take advantage of `gp` for shorter PLT sequences when the addresses are in range. The compiler needs an option to disable use of `gprel`; the option is necessary for applications with excessive amounts of small data. For comparison, XUL (Mozilla display engine, 16 MB code, 2 MB data) has only 27 KB of small data and the limit is 64 KB. This option is separate from -G 0, because -G 0 creates ABI incompatibility. A file compiled with -G 0 puts global `int` variables into `.data` but files compiled with -G 8 expect such `int` variables to be in `.sdata`.

PIC code which needs a GOT pointer needs to initialize the pointer locally using `nextpc`; the GOT pointer is not passed during function calls. This approach is compatible with both static relocatable binaries and System V style shared objects. A separate ABI is needed for shared objects with independently relocatable text and data.

Stack alignment is 32-bit. The frame pointer points at the top of the stack when it is in use, to simplify backtracing. Insert `alloca` between the local variables and the outgoing arguments. The stack pointer points to the bottom of the outgoing argument area.

A large `struct` return value is handled by passing a pointer in the first argument register (not the disjoint return value register).

Linux Operating System Call Interface

Unhandled instruction-related exceptions in user programs are mapped to the signals listed in Table 7–7.

Table 7–7. Signals for Unhandled Instruction-Related Exceptions

Exception	Signal
Supervisor-only instruction address	`SIGSEGV`
TLB permission violation (execute)	`SIGSEGV`
Supervisor-only instruction	`SIGILL`
Unimplemented instruction	`SIGILL`
Illegal instruction	`SIGILL`
Break instruction	`SIGTRAP`
Supervisor-only data address	`SIGSEGV`
Misaligned data address	`SIGBUS`
Misaligned destination address	`SIGBUS`
Division error	`SIGFPE`

Table 7–7. Signals for Unhandled Instruction-Related Exceptions

Exception	Signal
TLB Permission Violation (read)	SIGSEGV
TLB Permission Violation (write)	SIGSEGV

There are no floating-point exceptions. The optional floating point unit (FPU) does not support exceptions and any process wanting exact IEEE conformance needs to use a soft-float library (possibly accelerated by use of the attached FPU).

The `break` instruction in a user process might generate a `SIGTRAP` signal for that process, but is not required to. Userspace programs should not use the `break` instruction and userspace debuggers should not insert one. If no hardware debugger is connected, the OS should assure that the `break` instruction does not cause the system to stop responding. For information about userspace debugging, refer to "Userspace Breakpoints" on page 7–21.

The page size is 4 KB. Virtual addresses in user mode are all below 2 GB due to the MMU design. The NULL page is not mapped.

Linux Process Initialization

The stack pointer, `sp`, points to the argument count on the stack. Table 7–8 lists the initial state of the stack when a userspace process starts.

Table 7–8. Stack Initial State at User Process Start

Purpose	Start Address	Length
Unspecified	High addresses	
Referenced strings		Varies
Unspecified		
Null auxilliary vector entry		4 bytes
Auxilliary vector entries		8 bytes each
NULL terminator for envp		4 bytes
Environment pointers	sp + 8 + 4 × argc	4 bytes each
NULL terminator for argv	sp + 4 + 4 × argc	4 bytes
Argument pointers	sp + 4	4 bytes each
Argument count	sp	4 bytes
Unspecified	Low addresses	

If the application should register a destructor function with `atexit`, the pointer is placed in `r4`. Otherwise `r4` is zero.

The contents of all other registers are unspecified. User code should set `fp` to zero to mark the end of the frame chain.

The auxiliary vector is a series of pairs of 32-bit tag and 32-bit value, terminated by an `AT_NULL` tag.

Linux Position-Independent Code

Every position-independent code (PIC) function which uses global data or global functions must load the value of the GOT pointer into a register. Any available register may be used. If a caller-saved register is used the function must save and restore it around calls. If a callee-saved register is used it must be saved and restored around the current function. Examples in this document use `r22` for the GOT pointer.

The GOT pointer is loaded using a PC-relative offset to the `_gp_got` symbol, as shown in Example 7–12.

Example 7–12. Loading the GOT Pointer

```
nextpc r22
1:
  orhi   r1, %hiadj(_gp_got - 1b)   # R_NIOS2_PCREL_HA _gp_got
  addi   r1, r1, %lo(_gp_got - 1b)  # R_NIOS2_PCREL_LO _gp_got - 4
  add    r22, r22, r1
  # GOT pointer in r22
```

Data may be accessed by loading its location from the GOT. A single word GOT entry is generated for each referenced symbol. For global symbols, the entry is as shown in Example 7–13.

Example 7–13. GOT Entry for Global Symbols

```
addi   r3, r22, %got(x)           # R_NIOS2_GOT16

GOT[n]                              R_NIOS2_GLOB_DAT x
```

For local symbols, the symbolic reference to *x* is replaced by a relative relocation against symbol zero, with the link time address of *x* as an addend, as shown in Example 7–14.

Example 7–14. Local Symbols

```
addi   r3, r22, %got(x)           # R_NIOS2_GOT16

GOT[n]                              R_NIOS2_RELATIVE +x
```

The `call` and `jmpi` instructions are not available in position-independent code. Instead, all calls are made through the GOT. Function addresses may be loaded with `%call`, which allows lazy binding. To initialize a function pointer, load the address of the function with `%got` instead. If no input object requires the address of the function its GOT entry is placed in the PLT GOT for lazy binding, as shown in Example 7–15. For information about the PLT, refer to “Procedure Linkage Table” on page 7–19.

Example 7–15. GOT entry in PLT GOT

```
ldw    r3, %call(fun)(r22)        # R_NIOS2_CALL16 fun
callr  r3

PLTGOT[n]                           R_NIOS_JUMP_SLOT fun
```

When a function or variable resides in the current shared object at compile time, it can be accessed via a PC-relative or GOT-relative offset, as shown in Example 7–16.

Example 7–16. Accessing Function or Variable in Current Shared Object

```
orhi   r3, %gotoff_hiadj(x)       # R_NIOS2_GOTOFF_HA x
addi   r3, r3, %gotoff_lo(x)      # R_NIOS2_GOTOFF_LO x
add    r3, r22, r3
# Address of x in r3
```

Multiway branches such as switch statements can be implemented with a table of GOT-relative offsets, as shown in Example 7–17.

Example 7–17. Switch Statement Implemented with Table

```
# Scaled table offset in r4
 orhi   r3, %gotoff_hiadj(Ltable)  # R_NIOS2_GOTOFF_HA Ltable
 addi   r3, r3, %gotoff_lo(Ltable) # R_NIOS2_GOTOFF_LO Ltable
 add    r3, r22, r3                # r3 == &Ltable
 add    r3, r3, r4
 ldw    r4, 0(r3)                  # r3 == Ltable[index]
 add    r4, r4, r22                # Convert offset into destination
 jmp    r4
 ...
Ltable:
 .word  %gotoff(Label1)
 .word  %gotoff(Label2)
 .word  %gotoff(Label3)
```

Linux Program Loading and Dynamic Linking

Global Offset Table

Because shared libraries are position-independent, they can not contain absolute addresses for symbols. Instead, addresses are loaded from the GOT.

The first word of the GOT is filled in by the link editor with the unrelocated address of the `_DYNAMIC`, which is at the start of the dynamic section. The second and third words are reserved for the dynamic linker. For information about the dynamic linker, refer to "Procedure Linkage Table" on page 7–19.

The linker-defined symbol `_GLOBAL_OFFSET_TABLE_` points to the reserved entries at the beginning of the GOT. The linker-defined symbol `_gp_got` points to the base address used for GOT-relative relocations. The value of `_gp_got` might vary between object files if the linker creates multiple GOT sections.

Function Addresses

Function addresses use the same `SHN_UNDEF` and `st_value` convention for PLT entries as in other architectures, such as x86_64.

Procedure Linkage Table

Function calls in a position-dependent executable may use the `call` and `jmpi` instructions, which address the contents of a 256-MB segment. They may also use the `%lo`, `%hi`, and `%hiadj` operators to take the address of a function. If the function is in another shared object, the link editor creates a callable stub in the executable called a PLT entry. The PLT entry loads the address of the called function from the PLT GOT (a region at the start of the GOT) and transfers control to it.

The PLT GOT entry needs a relocation referring to the final symbol, of type R_NIOS2_JUMP_SLOT. The dynamic linker may immediately resolve it, or may leave it unmodified for lazy binding. The link editor fills in an initial value pointing to the lazy binding stubs at the start of the PLT section.

Each PLT entry appears as shown in Example 7–18.

Example 7–18. PLT Entry

```
.PLTn:
  orhi   r15, r0, %hiadj(plt_got_slot_address)
  ldw    r15, %lo(plt_got_slot_address)(r15)
  jmp    r15
```

Example 7–19 shows the PLT entry when the PLT GOT is close enough to the small data area for a relative jump.

Example 7–19. PLT Entry Near Small Data Area

```
.PLTn:
  ldw    r15, %gprel(plt_got_slot_address)(gp)
  jmp    r15
```

Example 7–20 shows the initial PLT entry.

Example 7–20. Initial PLT Entry

```
res_0:
  br     .PLTresolve
  ...
.PLTresolve:
  orhi   r14, r0, %hiadj(res_0)
  addi   r14, r14, %lo(res_0)
  sub    r15, r15, r14
  orhi   r13, %hiadj(_GLOBAL_OFFSET_TABLE_)
  ldw    r14, %lo(_GLOBAL_OFFSET_TABLE_+4)(r13)
  ldw    r13, %lo(_GLOBAL_OFFSET_TABLE_+8)(r13)
  jmp    r13
```

In front of the initial PLT entry, a series of branches start of the initial entry (the `nextpc` instruction). There is one branch for each PLT entry, labelled `res_0` through `res_N`. The last several branches may be replaced by `nop` instructions to improve performance. The link editor arranges for the *Nth* PLT entry to point to the *Nth* branch; `res_N` – `res_0` is four times the index into the `.rela.plt` section for the corresponding R_JUMP_SLOT relocation.

The dynamic linker initializes GOT[1] to a unique identifier for each library and GOT[2] to the address of the runtime resolver routine. In order for the two loads in `.PLTresolve` to share the same `%hiadj`, `_GLOBAL_OFFSET_TABLE_` must be aligned to a 16-byte boundary.

The runtime resolver receives the original function arguments in `r4` through `r7`, the shared library identifier from GOT[1] in `r14`, and the relocation index times four in `r15`. The resolver updates the corresponding PLT GOT entry so that the PLT entry transfers control directly to the target in the future, and then transfers control to the target.

In shared objects, the `call` and `jmpi` instructions can not be used because the library load address is not known at link time. Calls to functions outside the current shared object must pass through the GOT. The program loads function addresses using `%call`, and the link editor may arrange for such entries to be lazily bound. Because PLT entries are only used for lazy binding, shared object PLTs are smaller, as shown in Example 7–21.

Example 7–21. Shared Object PLT

```
.PLTn:
  orhi   r15, r0, %hiadj(index * 4)
  addi   r15, r15, %lo(index * 4)
  br     .PLTresolve
```

Example 7–22 shows the initial PLT entry.

Example 7–22. Initial PLT Entry

```
.PLTresolve:
  nextpc r14
  orhi   r13, r0, %hiadj(_GLOBAL_OFFSET_TABLE_)
  add    r13, r13, r14
  ldw    r14, %lo(_GLOBAL_OFFSET_TABLE_+4)(r13)
  ldw    r13, %lo(_GLOBAL_OFFSET_TABLE_+8)(r13)
  jmp    r13
```

If the initial PLT entry is out of range, the resolver can be inline, because it is only one instruction longer than a long branch, as shown in Example 7–23.

Example 7–23. Initial PLT Entry Out of Range

```
.PLTn:
  orhi   r15, r0, %hiadj(index * 4)
  addi   r15, r15, %lo(index * 4)
  nextpc r14
  orhi   r13, r0, %hiadj(_GLOBAL_OFFSET_TABLE_)
  add    r13, r13, r14
  ldw    r14, %lo(_GLOBAL_OFFSET_TABLE_+4)(r13)
  ldw    r13, %lo(_GLOBAL_OFFSET_TABLE_+8)(r13)
  jmp    r13
```

Linux Program Interpreter

The program interpreter is **/lib/ld.so.1**.

Linux Initialization and Termination Functions

The implementation is responsible for calling `DT_INIT()`, `DT_INIT_ARRAY()`, `DT_PREINIT_ARRAY()`, `DT_FINI()`, and `DT_FINI_ARRAY()`.

Linux Conventions

System Calls

The Linux system call interface relies on the `trap` instruction with immediate argument zero. The system call number is passed in register `r2`. The arguments are passed in `r4`, `r5`, `r6`, `r7`, `r8`, and `r9` as necessary. The return value is written in `r2` on success, or a positive error number is written to `r2` on failure. A flag indicating successful completion, to distinguish error values from valid results, is written to `r7`; 0 indicates `syscall` success and 1 indicates `r2` contains a positive `errno` value.

Userspace Breakpoints

Userspace breakpoints are accomplished using the `trap` instruction with immediate operand 31 (all ones). The OS must distinguish this instruction from a `trap 0` system call and generate a `trap` signal.

Atomic Operations

The Nios II architecture does not have atomic operations (such as load linked and store conditional). Atomic operations are emulated using a kernel system call via the `trap` instruction. The toolchain provides intrinsic functions which perform the system call. Applications must use those functions rather than the system call directly. Atomic operations may be added in a future processor extension.

Processor Requirements

Linux requires that a hardware multiplier be present. The full 64-bit multiplier (`mulx` instructions) is not required.

Development Environment

The following symbols are defined:

- `__nios2`
- `__nios2__`
- `__NIOS2`
- `__NIOS2__`

Document Revision History

Table 7–9 lists the revision history for this document.

Table 7–9. Document Revision History

Date	Version	Changes
May 2011	11.0.0	Maintenance release.
December 2010	10.1.0	Added Linux ABI section.
July 2010	10.0.0	■ DWARF-2 register assignments ■ ELF header values ■ `r23` used as thread pointer for Linux ■ Linux toolchain relocation information ■ Symbol definitions for development environment
November 2009	9.1.0	Maintenance release.
March 2009	9.0.0	Backwards-compatible change to the `eret` instruction B field encoding.
November 2008	8.1.0	Maintenance release.
May 2008	8.0.0	■ Frame pointer description updated. ■ Relocation table added.
October 2007	7.2.0	Maintenance release.
May 2007	7.1.0	■ Added table of contents to Introduction section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	Maintenance release.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	Maintenance release.
May 2005	5.0.0	Maintenance release.
September 2004	1.1	Maintenance release.
May 2004	1.0	Initial release.