

この資料は英語版を翻訳したもので、内容に相違が生じる場合には原文を優先します。こちらの日本語版は参考用としてご利用ください。設計の際には、最新の英語版で内容をご確認ください。

NII51016-5.1.0

この章では、Nios[®] II プロセッサ用のアプリケーション・バイナリ・インタフェース (ABI) について説明します。ABI は以下を表します。

- メモリ内でのデータ・アラインメント方法
- スタックの動作と構造
- 関数呼び出し規則

データ型

表 7-1 に、Nios II プロセッサのための C/C++ におけるデータ型のサイズと表現を示します。

表 7-1. データ型の表現		
型	サイズ (バイト)	表現
char, signed char	1	2 の補数 (ASCII)
unsigned char	1	バイナリ (ASCII)
short, signed short	2	2 の補数
unsigned short	2	バイナリ
int, signed int	4	2 の補数
unsigned int	4	バイナリ
long, signed long	4	2 の補数
unsigned long	4	バイナリ
float	4	IEEE
double	8	IEEE
pointer	4	バイナリ
long long	8	2 の補数
unsigned long long	8	バイナリ

メモリ・アラインメント

メモリ内容は以下のようにアラインメントされます。

- 関数は最小の 32 ビット境界にアラインメントしなければなりません。
- データ・エレメントの最小アラインメントは、エレメントの自然サイズです。32 ビットを超えるデータ・エレメントは、32 ビット境界にのみアラインメントする必要があります。

- 構造体、共用体、および文字列は、最小の 32 ビットにアラインメントしなければなりません。
- 構造体内部のビット・フィールドは常に 32 ビットでアラインメントされます。

レジスタの使用法

ABI は、「Nios II プロセッサ・リファレンス・ハンドブック」の「**プログラミング・モデル**」の章で定義される Nios II レジスタ・ファイルに、使用規則を追加します。ABI は、**表 7-2** に示すレジスタを使用します。

表 7-2. Nios II ABI レジスタの使用法 (1 / 2)

レジスタ	名称	コンパイラによる使用	呼び出し先での保存 (1)	一般的な使用法
r0	zero	√		0x00000000
r1	at			アセンブラ用テンポラリ
r2		√		戻り値 (最下位 32 ビット)
r3		√		戻り値 (最上位 32 ビット)
r4		√		レジスタ引数 (最初の 32 ビット)
r5		√		レジスタ引数 (2 番目の 32 ビット)
r6		√		レジスタ引数 (3 番目の 32 ビット)
r7		√		レジスタ引数 (4 番目の 32 ビット)
r8		√		呼び出し先で保存される汎用レジスタ
r9		√		
r10		√		
r11		√		
r12		√		
r13		√		
r14		√		
r15		√		
r16		√	√	呼び出し先で保存される汎用レジスタ
r17		√	√	
r18		√	√	
r19		√	√	
r20		√	√	
r21		√	√	
r22		√	√	
r23		√	√	

表 7-2. Nios II ABI レジスタの使用法 (2 / 2)

レジスタ	名称	コンパイラによる使用	呼び出し先での保存 (1)	一般的な使用法
r24	et			例外用テンポラリ
r25	bt			ブレーク用テンポラリ
r26	gp	√		グローバル・ポインタ
r27	sp	√		スタック・ポインタ
r28	fp	√		フレーム・ポインタ (2)
r29	ea			例外戻りアドレス
r30	ba			ブレーク戻りアドレス
r31	ra	√		戻りアドレス

表 7-2 の注：

- (1) 関数は最初に保存する場合に、これらのレジスタのうち1つを使用できます。関数は終了前にレジスタの元の値を復元する必要があります。
- (2) フレーム・ポインタを使用しない場合、レジスタをテンポラリ・レジスタとして使用できます。
7-4 ページの「フレーム・ポインタの削除」を参照してください。

データのエンディアン

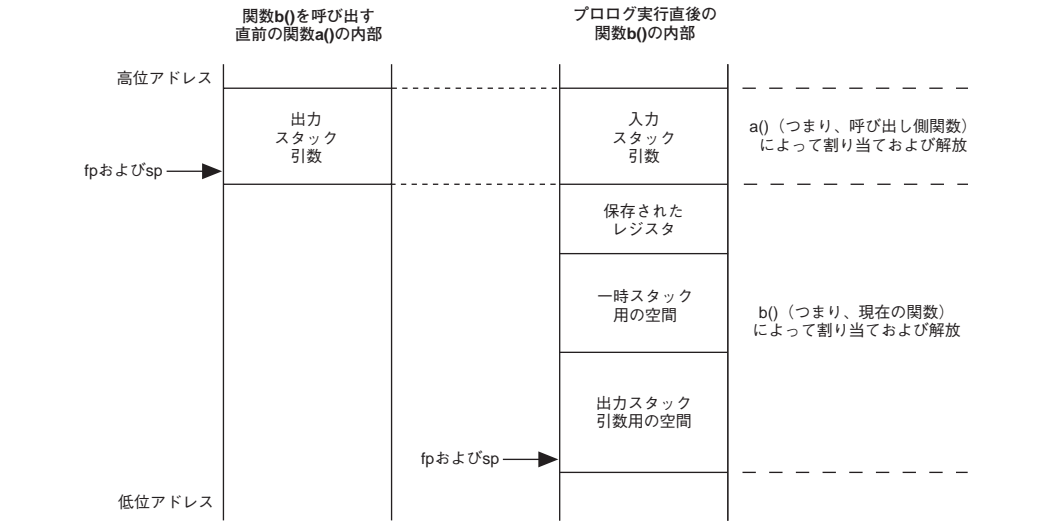
9 ビット以上の値のエンディアンは、リトル・エンディアンです。値の上位 8 ビットは、高位バイト・アドレスに格納されます。

スタック

スタックは下方（つまり、低位アドレス方向に）に成長します。スタック・ポインタは最後に使用されたスロットを指します。フレームは上方に成長しますが、これはフレーム・ポインタがフレームのボトムを指すことを意味します。

図 7-1 に、カレント・フレームの構造例を示します。この場合、関数 a() は関数 b() を呼び出し、スタックは呼び出された関数のプロログが完了した後の呼び出し前に表示されます。

図 7-1. スタック・ポインタ、フレーム・ポインタ、およびカレント・フレーム



カレント・フレームの各セクションは、32 ビット境界にアラインメントされます。ABI では、スタック・ポインタは常に 32 ビットでアラインメントする必要があります。

フレーム・ポインタの削除

通常のケースでは、フレーム・ポインタはスタック・ポインタと同じなので、フレーム・ポインタ内の情報が冗長となります。したがって、最適なコードを得るには、フレーム・ポインタを削除することが望ましいといえます。ただし、GDB にはスタックを正しく配置する上で問題があるため、フレーム・ポインタを削除した場合はフレーム・ポインタなしのデバッグが困難になります。フレーム・ポインタを削除すると、レジスタ fp はテンポラリ・レジスタとして利用できるようになります。

保存されたレジスタの呼び出し

実装に関する注：関数で保存する必要があるレジスタは、コンパイラが保存しなければなりません。そのようなレジスタが存在する場合、上位アドレスから、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、および sp の順にスタックに保存されます。保存されないレジスタにはスタック空間は割り当てられません。

その他のスタックの例

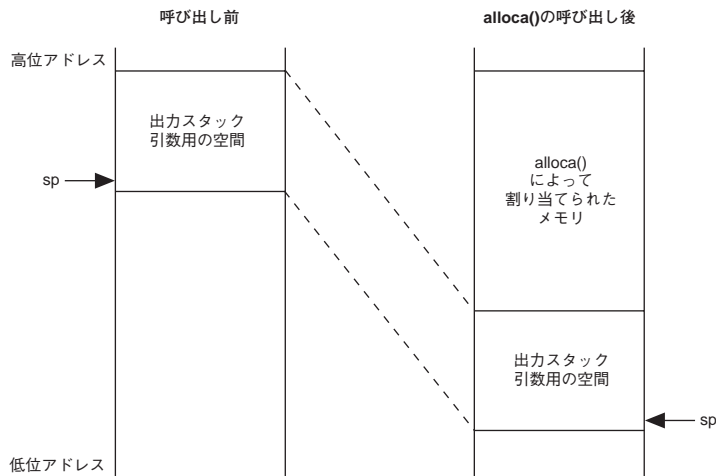
スタック・レイアウトには特殊なケースが多数あり、ここでそのいくつかを説明します。

alloca() を使用する関数用スタック・フレーム

図 7-2 は、`alloca()` を呼び出した後のフレームの状態を示します。`alloca()` で割り当てられた空間は出力引数と置き換わり、出力引数はフレームのボトムに割り当てられた新しい空間を取得します。

実装に関する注：Nios II C/C++ コンパイラは、媒介フレーム・ポインタが指定された場合でも、`alloca()` を呼び出す関数に対するフレーム・ポインタを維持します。

図 7-2. `alloca()` 呼び出し後のスタック・フレーム

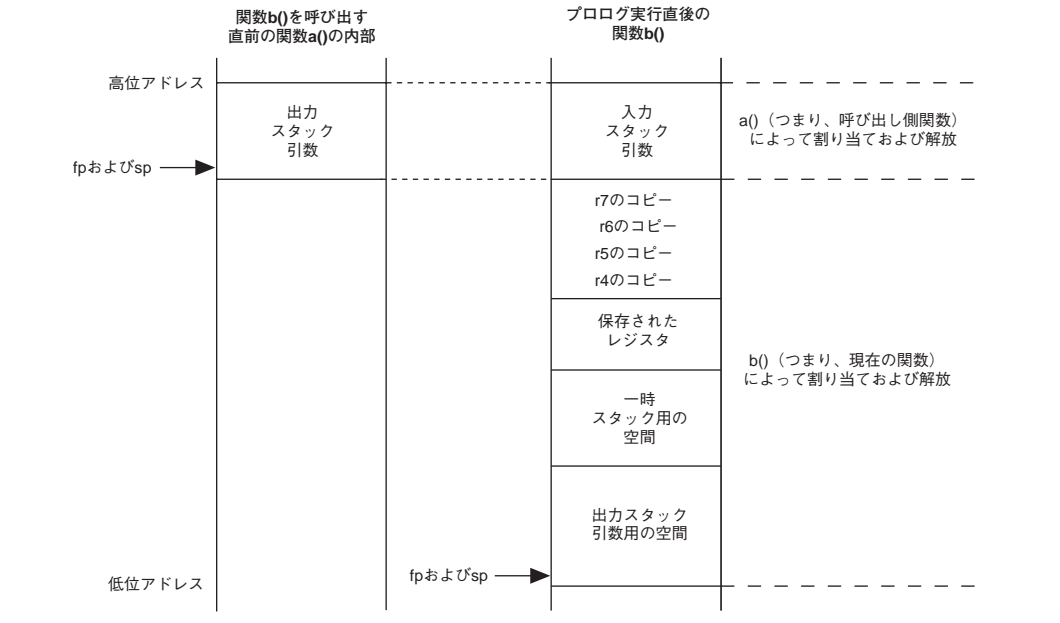


可変引数をもつ関数用スタック・フレーム

可変引数を受け取る関数も他の関数と同様に、レジスタ `r4 ~ r7` に到着する引数の最初の 16 バイトを維持します。

実装に関する注：可変引数を受け取る関数は、可変引数で使用するためにスタック上に 16 バイト余分にストレージを割り当てます。これらの関数は、図 7-3 に示すように、レジスタ `r4 ~ r7` から引数の最初の 16 バイトをスタックにコピーします。

図 7-3. 可変引数を使用するスタック・フレーム



値渡し構造体をもつ関数用スタック・フレーム

struct 型の値引数を受け取る関数も他の関数と同様に、レジスタ r4 ~ r7 に到着する引数の最初の 16 バイトを維持します。

実装に関する注：構造体の一部がレジスタを介して渡される場合、関数はレジスタの内容を再びスタックにコピーしなければならないことがあります。これは、図 7-3 に示す可変引数の場合と類似しています。

関数のプロログ

Nios II C/C++ コンパイラは、スタック・テンポラリと出力引数を格納するために、関数のスタック・フレームを割り当てる関数プロログを生成します。さらに、各プロログは ABI が呼び出し先保存とマークした変数について、呼び出し関数のすべての状態を保存しなければなりません。7-2 ページの表 7-2 に呼び出し先保存レジスタを示します。関数プロログは、関数がレジスタを使用する場合に限り、呼び出し先保存レジスタを保存する必要があります。

デバグは、関数プロログによる命令分解方法に関する情報を使用して、バック・トレース実行時に状態を再構築することができます。デバグは DWARF2 デバグ情報に格納された情報を使用して、プロログが実行した内容を検出できるのが理想的です。

Nios II 関数プロログ内で検出された命令は、以下のタスクを実行します。

- SP の調節（フレームを割り当てるため）
- フレームへのレジスタの格納
- FP への SP の割り当て

図 7-4 は、関数プロログの一例を示します。

図 7-4. 関数プロログ

```

/* Adjust the stack pointer */
addisp, sp, -120/* make a 120 byte frame */

/* Store registers to the frame */
stw ra, 116(sp)/* store the return address */
stw fp, 112(sp)/* store the frame pointer*/
stw r16, 108(sp)/* store callee-saved register */
stw r17, 104(sp) /* store callee-saved register */

/* Set the new frame pointer */
mov fp, sp

```

プロログの変化

プロログで以下の変化が発生することがあります。

- 関数のフレーム・サイズが 32,767 バイトより大きい場合、新しい SP および呼び出し先保存レジスタの格納場所のオフセットを計算するには、他のテンポラリ・レジスタが使用されます。これは Nios II プロセッサで許される即値の最大サイズのためです。
- フレーム・ポインタが使用されていない場合、SP を FP に移動することはありません。
- 可変引数を使用されている場合、引数レジスタをスタックに格納するための追加命令が存在します。
- 関数がリーフ関数の場合、戻りアドレスは保存されません。
- 特に命令のスケジューリングなど、最適化が指定されている場合、命令の順序が変わり、プロログの後に配置されている命令と混在する可能性があります。

引数と戻り値

ここでは、関数に引数を渡す方法、および関数から値を返す方法を詳細に説明します。

引数

最初の 16 バイト・データはレジスタ `r4 ~ r7` で関数に渡されます。引数は、引数の型を含む構造体が構築されたのと同様に渡され、構造体の最初の 16 バイトは、`r4 ~ r7` に置かれます。

簡単な例

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

引数を表す同等な構造体は、以下のとおりです。

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

構造体の最初の 16 バイトは、`r4 ~ r7` に割り当てられます。したがって、`r4` には `a` の値、`r5` には `b` の値が割り当てられます。

可変引数

可変引数を受け取る関数の最初の 16 バイトは、可変引数を受け取らない関数の場合と同様に渡されます。可変引数をサポートするために、必要に応じてスタックをクリーンアップするのは呼び出された関数の役割です。7-5 ページの「[可変引数をもつ関数用スタック・フレーム](#)」を参照してください。

戻り値

8 バイトまでの型の戻り値は、`r2` と `r3` に返されます。戻り値が 8 バイトを超える場合、呼び出し側は結果にメモリを割り当て、さらに結果メモリのアドレスを隠れゼロ引数として渡す必要があります。

隠れゼロ引数は、例を使えばわかりやすく説明できます。

例: 関数 `a()` は、構造体を返す関数 `b()` を呼び出します。

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

void a(...)
{
```

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

この例では、**result** 型が 8 バイト以下である限り、**b()** は **r2** と **r3** に **result** を返します。

return 型が 8 バイトよりも大きい場合、Nios II C/C++ コンパイラは、**a()** がポインタを **b()** に渡したのと同様に、このプログラムを処理します。以下の例は、Nios II C/C++ コンパイラが上記のコードをどのように処理するかを示します。

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

