Nios II マルチコア構成時の共有関数のリンク方法

ver.14

2015 年 5 月 Rev.1

Nios II マルチコア構成時の共有関数のリンク方法

目次

1. はじめに

この資料は、複数の Nios® II を使用してマルチコア構成とした場合に、一つの関数を複数の CPU コア（Nios II）から参照する一つの手法を紹介します。プログラム・メモリの容量が足りず、関数を共有して使用したいときなどにご活用ください。

2. 適用条件

■ 対応バージョン

－ Quartus® II 開発ソフトウェア v14.1

－ Nios II Software Build Tools (Nios II SBT) v14.1

■ 検証ハードウェア

－ Cyclone V E FPGA Development Board

・ FPGA : Cyclone V 5CEFA7F31I7ES

3. 概要

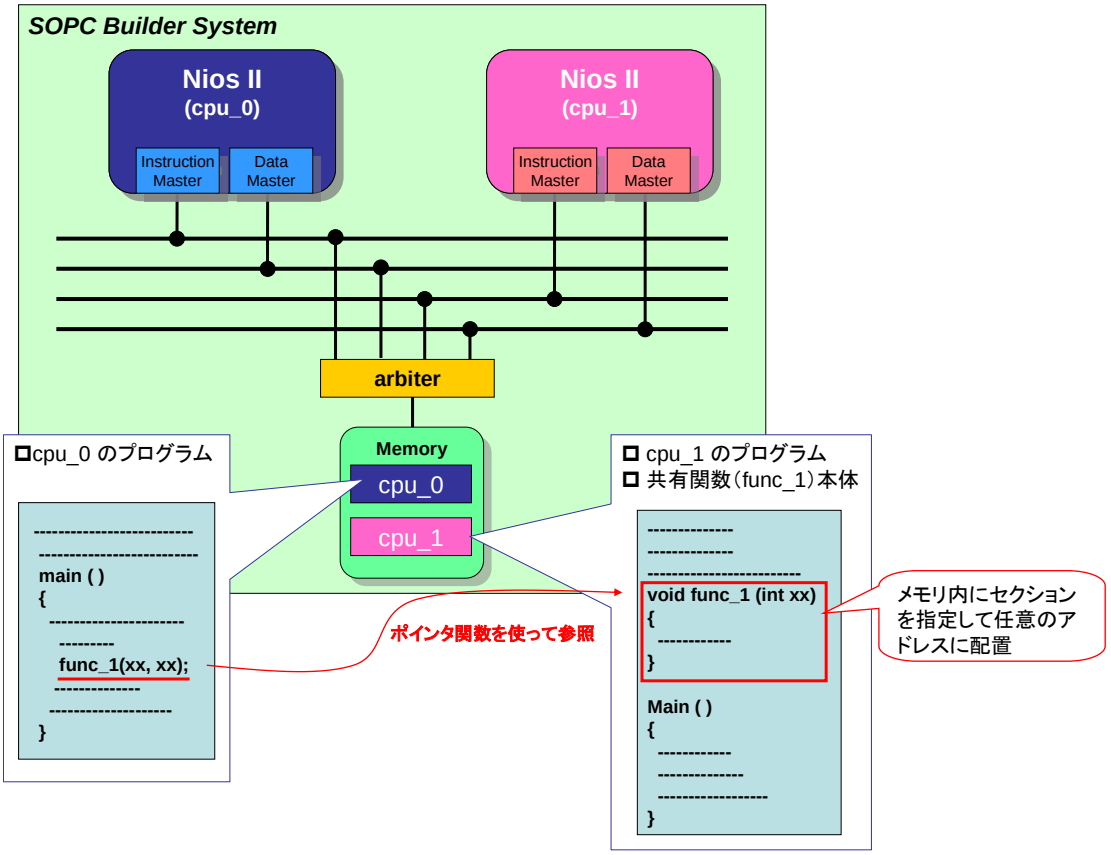
下記のような手法を用いて、複数の CPU コアから 他のコアが持つ関数へのアクセスを行います。

- 複数のコアが共有しているメモリにセクションを切り、共有したい関数（func_1）を配置します。
- 別の func_1 を持っていない関数から、関数ポインタで func_1 を呼び出して使用します。

例として下記のような構成を想定します。

システム内に Nios II が 2 つのマルチコアの構成とします。cpu_0、cpu_1 がプログラム・メモリとして一つのメモリを共有します。

cpu_1 のソース・コードにのみ共有関数（func_1）を記述します。cpu_0 のソース・コードの中には共有関数（func_1）はありませんので、cpu_1 の func_1 を参照して実行します。



4. セクション定義

cpu_1 のソフトウェア・プロジェクトのプログラム・メモリ内に任意のセクションを定義し、このセクションに共有関数を配置します。

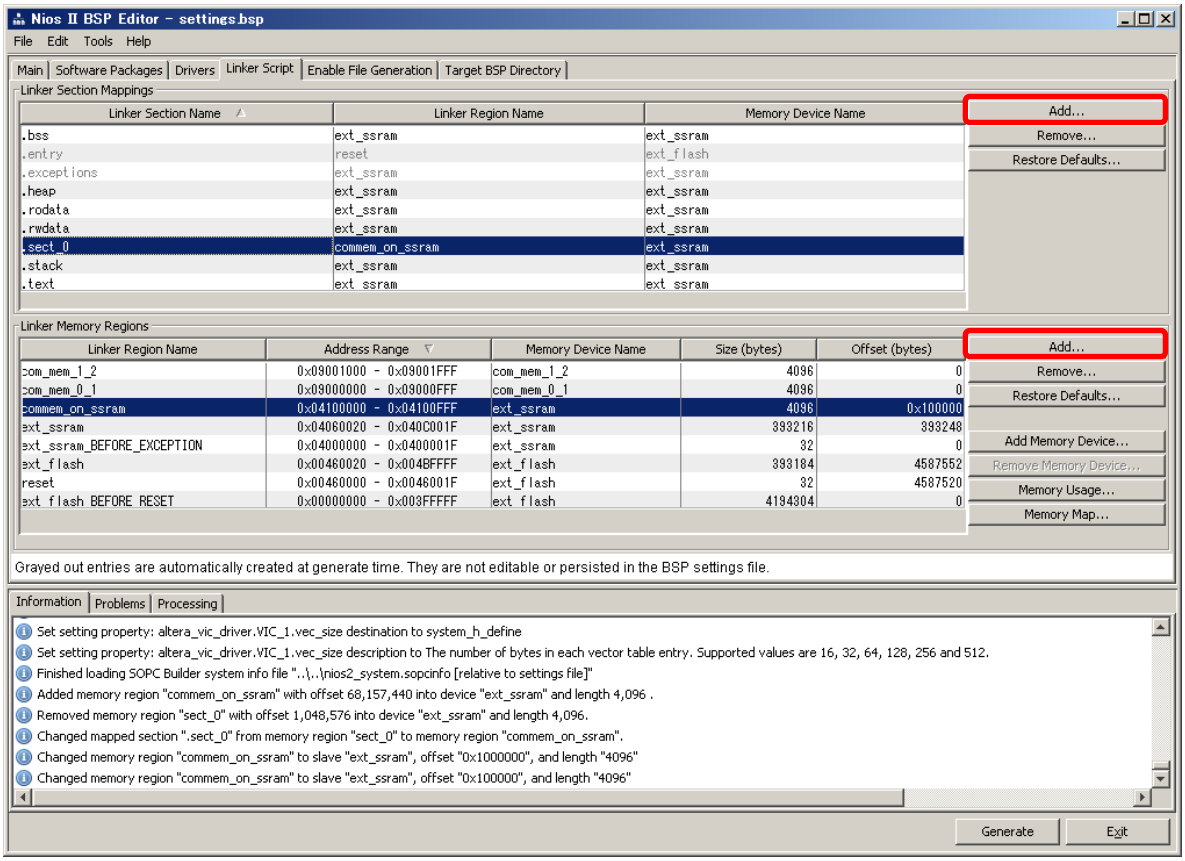
Nios II SBT にてプロジェクトを作成した際にリンカ・スクリプトが自動生成されますが、このリンカ・スクリプトをユーザが編集し使用することが可能です。

① BSP プロジェクト(bsp のプロジェクト)から BSP Editor を起動します。
<プロジェクト名>_bsp を右クリックから Nios II ⇒ BSP Editor
フォルダに generated.x ファイルが生成されます。こちらがプロジェクトのリンカ・スクリプトになります。

② Linker Script タブを開きます。

③ Linker Memory Regions の “Add” ボタンをクリックして、新しい Region を追加します。以下の例では、commem_on_ssram という名前の Region を作成しています。Size を 4096 に Offset を 0x100000 に設定しています。この設定で、SSRAM の空き領域に 4096 バイトの共有メモリ領域が確保されました。サイズやオフセットをどのように設定するかは、共有関数のサイズと使用する RAM 領域の空き領域がどのくらいあるかで決定します。

④ 次に、Linker Section Mappings の “Add” ボタンをクリックしてに新しいセクション .sect_0 を生成します。Linker Region Name には、commem_on_ssram を選択します。



5. 共有関数を使用する

cpu_1 のプログラム内にある共有関数 func_1 を cpu_0 のソース上から使用します。

① cpu_1 のプログラム中(cpu_1.c)に下記のように __attribute___ 記述で、3 章で指定したセクション(.sect_0)に func_1 を配置します。func_1 の関数本体も cpu_1 のプログラム中に記述します。

cpu_1 のプログラム

cpu_1.c

```
#include <stdio.h>
#include <io.h>
#include <unistd.h>
#include "system.h"

// ******************************
// 共有関数 func_1
// ******************************
int func_1(alt_u32 led_pio_base) __attribute__ ((section (".sect_0")));
```

func_1 本体

```
int func_1(alt_u32 led_pio_base)
{
    while(1)
    {
        IOWR(led_pio_base, 0, 0x0);
        usleep(500000);
        IOWR(led_pio_base, 0, 0x1);
        usleep(500000);
```

※ *注) 共有する関数(例:func_1)内で printf 等の標準関数や HAL を使用している場合には、cpu_1 のプログラムのライブラリを使用しますので、それらの関数がスレッド・セーフかどうかを確認してご使用ください。*

※ *共有関数自体を mutex 等のコアを使用し、それぞれの CPU からの共有関数へのアクセスを排他制御することもできます。*

② cpu_0 のプログラム中で func_1 を使用します。cpu_0 のプログラムは func_1 の関数を持っていませんので、cpu_1 のプログラム中の func_1 を参照して使用します。① で func_1 を .sect で指定したアドレスに配置しましたので、このアドレスを関数ポインタとして宣言し使用します。

cpu_0 のプログラム

cpu_1.c　cpu_0.c

```
#include <stdio.h>
#include <io.h>
#include <unistd.h>
#include "system.h"

int main(void)
{
    int (*func_p)(alt_u32 led_pio_base);
    func_p = (int(*)(alt_u32))0x04100000;

    printf("Hello from Nios II 0, start!!");

    (*func_p)(LED_PIO_0_BASE);

    return 0;

}
```

- 関数ポインタ宣言
- .sect_0 で指定したアドレスをポインタに入力

関数 func_1 を関数ポインタ *func_p を使って呼び出します

改版履歴

Revision	年月	概要
1	2015 年 5 月	初版