DE0 Nano SoC – AMP System

Data prima stesura	**19/09/2016**
Data revisione	**22/10/2016**
Revisione	**01**
Autore	**Calzoni Pietro**
Riferimenti per Download	**www.lugman.org o su richiesta**

Con Asymmetric Multiprocessing (AMP) si intende la possibilità di eseguire su una stessa macchina con più core fisici, differenti software (semplici programmi o sistemi operativi completi) per ogni core in modo indipendente.
Nell'ambito dell'automazione e dei controlli industriali i sistemi embedded dominano in quanto spesso si realizzano software specifici per risolvere un problema che girano su una o più CPU con relativamente scarse risorse, ma tutte destinate al task principale.
Nonostante l'estrema specificità del mondo embedded, oggi si richiede ai prodotti di supportare tutta una serie di interfacce tipiche del modo consumer (Ethernet, USB, ecc) o di essere configurabili con un sistema operativo.
I sistemi AMP permettono di risolvere questo problema caricando su una CPU un sistema operativo di alto livello come LINUX per la gestione delle interfacce, mentre su un'altra CPU può essere eseguito il codice principale per il controllo del dispositivo/macchina/inverter che spesso deve soddisfare prestazioni di tipo hard-realtime o di affidabilità molto più elevate di quelle garantibili da un sistema operativo.

Obiettivi del documento:
1. Creare un applicativo Baremetal con interfaccia verso la FPGA caricato da UBoot
2. Creare un driver per Linux per essere eseguito su un solo Core
3. Caricare da Linux il codice baremetal ed eseguirlo sulla CPU1
4. Interagire tra Linux e Baremetal

Software Utilizzato:
- ALTERA Quartus Prime (versione 16.0)
- ARM DS5 Community Edition (facoltativo)

Prerequisiti:
- Conoscenza del C
- Utilizzo base di Linux
- Utilizzo minimale di UBoot

Scheda Target: DE0 Non SoC
Il sistema si ritiene configurato secondo quanto descritto in [26], ma in realtà si può utilizzare anche l'immagine SD fornita da Terasic o da Rocketboards.
Il collegamento alla scheda di sviluppo è fatto con una console seriale sulla porta UART (USBSerial). Il programma utilizzato per collegarsi è PuTTY, ma qualunque altro programma è valido; la velocità da impostare sarà 115200 boud senza controllo di flusso.

INDICE

1. Interfaccia ARM-FPGA per il controllo dei LED 3
1.1. Progetto FPGA: creazione della periferica LED 3
1.2. Crearezione e Test del File RBF – Raw Bit File 6
1.3. Creazione degli Headers C 7
2. Compilazione del Codice Baremetal 7
2.1. Test 1 – Accesso alla Periferica LED e a USER-KEY da UBoot 9
2.2. Test 2 – Accesso alla Periferica LED da LINUX 11
2.3. Test 2 – Accesso al timer di sistema 11
3. LINUX Kernel Driver – Passaggio da SMP a AMP 12
3.1. Boot ARM Multicore 12
3.2. Sistema di Attivazione delle CPU secondarie in LINUX 13
3.2.1. Struttura dei file 14
3.2.2. Routine e procedure di spegnimento e riattivazione delle CPU 14
3.3. AMP Driver 17
3.3.1. Implementazione 17
3.3.2. Variante per Caricamento ELF 20
3.3.3. Compilazione 21
3.3.4. Test 21
3.3.5. Bug e Miglioramenti 23
4. Cosa Manca 23
5. Riferimenti e Link 25

ACRONIMI, DEFINIZIONI E ABBREVIAZIONI

Sistema HOST	è il sistema sul quale sono installati i tool di sviluppo come i cross-compilatori: di fatto Linux Mint KDE 17.2
Sistema Target	sistema di destinazione, tipicamente Linux su ARM o l'FPGA
SoC	System On a Chip: sono chip che realizzano un sistema completo comprensivo di un certo numero di periferiche già implementate sul DIE.
HPS	*Hard Processor System*: in Cyclone V di fatto è l'ARM
AMBA	*ARM Advanced Microcontroller Bus Architectur*: attualmente alla versione 4, è un bus standard opensource di ARM utilizzato principalemnte su SoC.
AXI	*Advanced eXtensible Interface*: parte di AMBA, permette a sistemi multi-core di condividere memoria oltre ad altre tecnologie ARM e di effettuare transazioni ad alta velocità..
NIC-301	ARM CoreLink Network Interconnect: Sistema di interconnessione tra HPS e le periferiche, basato su bus AMBA, AXI, AHB e APB bus.
IP	Intellectual Property: componenti che è possibile installare, non necessariamente a pagamento, per estendere particolari funzionalità (es: sono IP le MegaFunction ALTERA o il soft core NIOS II)
EDS	SoCEmbedded Design Suite: di fatto DS-5 Altera Community Edition
ABI	*Application Binary Interface*: è l'interfaccia che garantisce che un programma sia compilato seguendo le regole del sistema che poi lo eseguirà; per esempio indica come viene chiamata una funzione, quanti parametri ha, come vengono allineati i dati, ecc.
EABI	*Embedded ABI*: tipicamente usate per ARM o per processori embedded.
Baremetal	programma eseguito direttamente sulla CPU senza l'ausilio di un sistema operativo.

1. INTERFACCIA ARM-FPGA PER IL CONTROLLO DEI LED

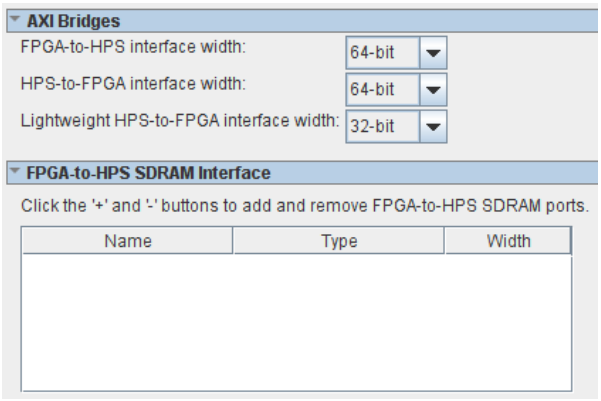
Il preloader creato come indicato in [26] non permette di controllare lo USER LED collegato all'ARM. Per questa ragione occorre creare una semplice periferica su FPGA in modo da poter controllare gli 8 LED collegati alla FPGA.
Il secondo scopo è creare tutti gli header file per i programmi baremetal in C che di fatto esportano le informazioni degli indirizzi delle periferiche FPGA.
L'obiettivo non è creare il board support package per compilare U-Boot o altro, ma ciò che serve veramente sono solo le periferiche FPGA; quindi la creazione della CPU è abbastanza semplice e ciò che importa è solo l'abilitazione dei bus per interfacciare la FPGA stessa.
Riassumendo i passi da eseguire sono:

1. Creazione di un progetto QSys e di uno Quartus per realizzare l'interfccia con l'FPGA
2. Testare il fimware creato e generare il file *.rbf* per poter programmare l'FPGA dall'ARM
3. Creare il BSP e quindi gli header per il controllo delle periferiche collegate al bus AXI

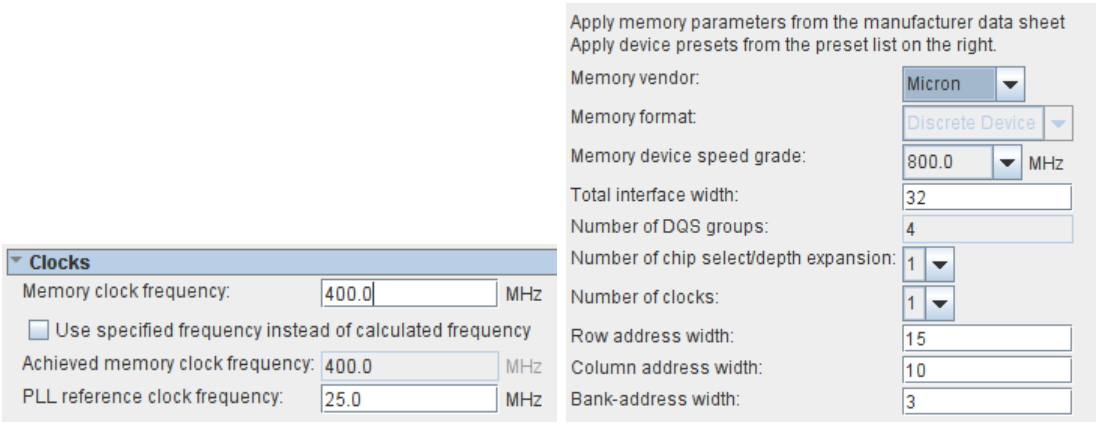
1.1. PROGETTO FPGA: CREAZIONE DELLA PERIFERICA LED

Le istruzioni che seguono prevedono già la conoscenza dell'ambiente di sviluppo:

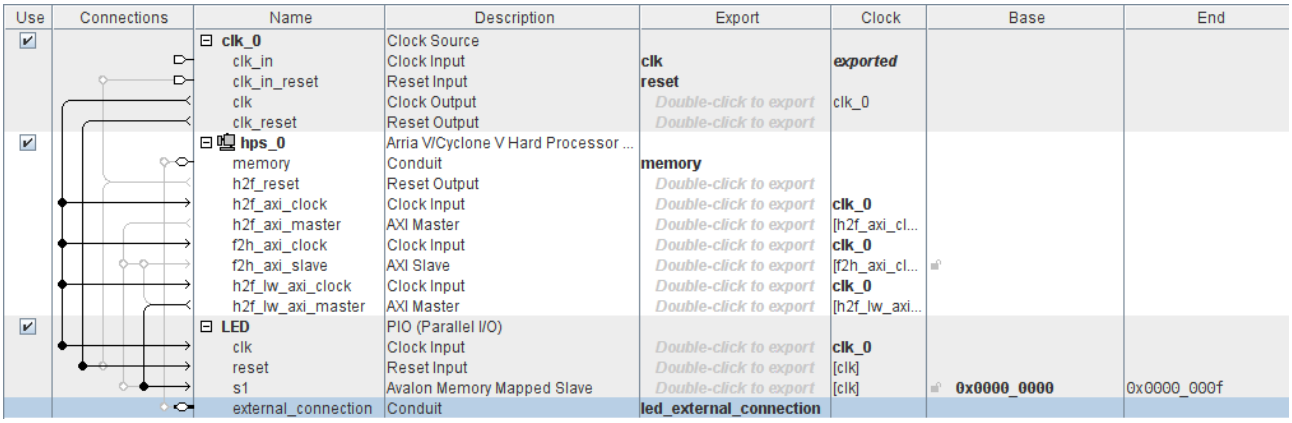
- Avviare *Quartus Prime*
- Creare un progetto vuoto per Cyclone V **5CSEMA4U23C6N** con nome del progetto *main*
- Avviare QSys
- Aggiungere la CPU ARM selezionando l'IP: **Arria V/Cycone V Hard Processing System**
  - Eliminare "*Enable MPU Standby and Event signal*"
  - Mantenere le interfacce: l'unica che conta è la Lightweight HPS-to-FPGA



  - Nessuna periferica da configurare
  - Lasciare inalterati i clock per lavorare a 925MHz
  - Anche la RAM potrebbe essere lasciata invariata in quanto la configurazione verrà fatta dal bootloader. Ad ogni modo può essere configurato come segue:



- Aggiungere una porta parallela PIO configurata come output a 8 bit e con valore inizializzato a 0x0; rinominare come LED e assegnare il *Conduit*.
- Collegare le varie componenti come in figura:



- Salvare il progetto (es: CPUInterface) e generare il progetto
- Chiudere QSys
- Aggiungere ai file di progetto l'IP appena creata: *<generation_directory>/synthesis/CPUInterface.qip*
- Creare un modulo che contenga l'interfaccia per la RAM e per i LED. I LED verranno comandati o dalla CPU o dalla FPGA premento il tasto KEY0. Il reset viene messa sulla KEY1:

```
/** CPU Module Interface
 **/

module main(
    input  wire        clk,                          //     clk.clk
    output wire [7:0]  LED,                        //     led.export
    output wire [14:0] memory_mem_a,          //  memory.mem_a
    output wire [2:0]  memory_mem_ba,         //        .mem_ba
    output wire        memory_mem_ck,         //        .mem_ck
    output wire        memory_mem_ck_n,       //        .mem_ck_n
    output wire        memory_mem_cke,        //        .mem_cke
    output wire        memory_mem_cs_n,       //        .mem_cs_n
    output wire        memory_mem_ras_n,      //        .mem_ras_n
    output wire        memory_mem_cas_n,      //        .mem_cas_n
    output wire        memory_mem_we_n,       //        .mem_we_n
    output wire        memory_mem_reset_n,    //        .mem_reset_n
    inout  wire [31:0] memory_mem_dq,         //        .mem_dq
    inout  wire [3:0]  memory_mem_dqs,        //        .mem_dqs
    inout  wire [3:0]  memory_mem_dqs_n,      //        .mem_dqs_n
    output wire        memory_mem_odt,        //        .mem_odt
    output wire [3:0]  memory_mem_dm,         //        .mem_dm
    input  wire        memory_oct_rzqin,      //        .oct_rzqin
    input  wire        reset_n,                                 //   reset.reset_n
    input  wire                  KEY0
);

// HPS Wire
wire [7:0] led;

// Interfaccia della CPU
CPUInterface u0 (
    .clk_clk                       (clk),
    .led_external_connection_export (led),
    .memory_mem_a                  (memory_mem_a),       // memory.mem_a
    .memory_mem_ba                 (memory_mem_ba),      //       .mem_ba
    .memory_mem_ck                 (memory_mem_ck),      //       .mem_ck
    .memory_mem_ck_n               (memory_mem_ck_n),    //       .mem_ck_n
    .memory_mem_cke                (memory_mem_cke),     //       .mem_cke
    .memory_mem_cs_n               (memory_mem_cs_n),    //       .mem_cs_n
    .memory_mem_ras_n              (memory_mem_ras_n),   //       .mem_ras_n
    .memory_mem_cas_n              (memory_mem_cas_n),   //       .mem_cas_n
    .memory_mem_we_n               (memory_mem_we_n),    //       .mem_we_n
    .memory_mem_reset_n            (memory_mem_reset_n), //       .mem_reset_n
```

```
        .memory_mem_dq                  (memory_mem_dq),        //      .mem_dq
        .memory_mem_dqs                 (memory_mem_dqs),       //      .mem_dqs
        .memory_mem_dqs_n               (memory_mem_dqs_n),     //      .mem_dqs_n
        .memory_mem_odt                 (memory_mem_odt),       //      .mem_odt
        .memory_mem_dm                  (memory_mem_dm),        //      .mem_dm
        .memory_oct_rzqin               (memory_oct_rzqin),     //      .oct_rzqin
        .reset_reset_n                  (reset_n)
    );

    // Controllo dei LED
    // Assegnazione delle uscite
    assign LED = KEY0 ? led : 8'h55;

    endmodule
```

- Eseguire *Analysis & Synthesis* e successivamente procedere all'assegnazione dei pin:
  - Eseguire lo script per l'assegnazione dei pin *Tools → Tcl Script → hps_sdram_p0_pin_assignments.tcl*: potrebbero risultare degli errori, ma non curarsene.
  - Assegnare il RESET_N sulla KEY1: *PIN_AH16*, 3.3-V LVTTL
  - Assegnare gli altri pin con Pin Planner come di seguito:

in KEY0	Input	PIN_AH17	4A	B4A_N0	3.3-V LVTTL		16mA (default)
out LED[7]	Output	PIN_AA23	5A	B5A_N0	3.3-V LVTTL		16mA (default)
out LED[6]	Output	PIN_Y16	5A	B5A_N0	3.3-V LVTTL		16mA (default)
out LED[5]	Output	PIN_AE26	5A	B5A_N0	3.3-V LVTTL		16mA (default)
out LED[4]	Output	PIN_AF26	5A	B5A_N0	3.3-V LVTTL		16mA (default)
out LED[3]	Output	PIN_V15	5A	B5A_N0	3.3-V LVTTL		16mA (default)
out LED[2]	Output	PIN_V16	5A	B5A_N0	3.3-V LVTTL		16mA (default)
out LED[1]	Output	PIN_AA24	5A	B5A_N0	3.3-V LVTTL		16mA (default)
out LED[0]	Output	PIN_W15	5A	B5A_N0	3.3-V LVTTL		16mA (default)
in clk	Input	PIN_V11	3B	B3B_N0	3.3-V LVTTL		16mA (default)

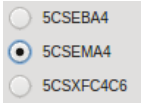
- Si consiglia di settare tutti i pin non assegnati dell'FPGA in "*As input tristate*" in *Settings → Device/Board → Device and Pin Option*, voce *Unused Pin*.
- Chiudere Pin Planner e compilare . Se tutto va a buon fine si dovrebbe ottenere:

Quartus Prime Version	16.0.0 Build 211 04/27/2016 SJ Lite Edition
Revision Name	main
Top-level Entity Name	main
Family	Cyclone V
Device	5CSEMA4U23C6
Timing Models	Final
Logic utilization (in ALMs)	177 / 15,880 (1 %)
Total registers	413
Total pins	83 / 314 (26 %)
Total virtual pins	0
Total block memory bits	0 / 2,764,800 (0 %)
Total DSP Blocks	0 / 84 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 5 (0 %)
Total DLLs	1 / 4 (25 %)

Test

Per testare il firmware appena creato, occorre:

- collegare il cavo USB alla poera USB-Blaster della scheda DE0-Nono-SoC
- collegarsi con *Quartus Programmer*
- selezionare da Hardware Setup DE0-SoC [2-1.1] (se non è stato trovato automaticamente)
- *AutoDetect*: selezionare la CPU montata sulla DE0, ovvero 5CSEMA4. Comparirà la catena JTAG per la programmazione della CPU ARM e dell'FPGA



- *Add File*: selezionare negli output file main.sof (comparirà un'altra FPGA in catena), cancellare la vecchia FPGA senza la selezione del file e programmare il dispositivo premendo *Start*. Il risultato è in Figura 1.

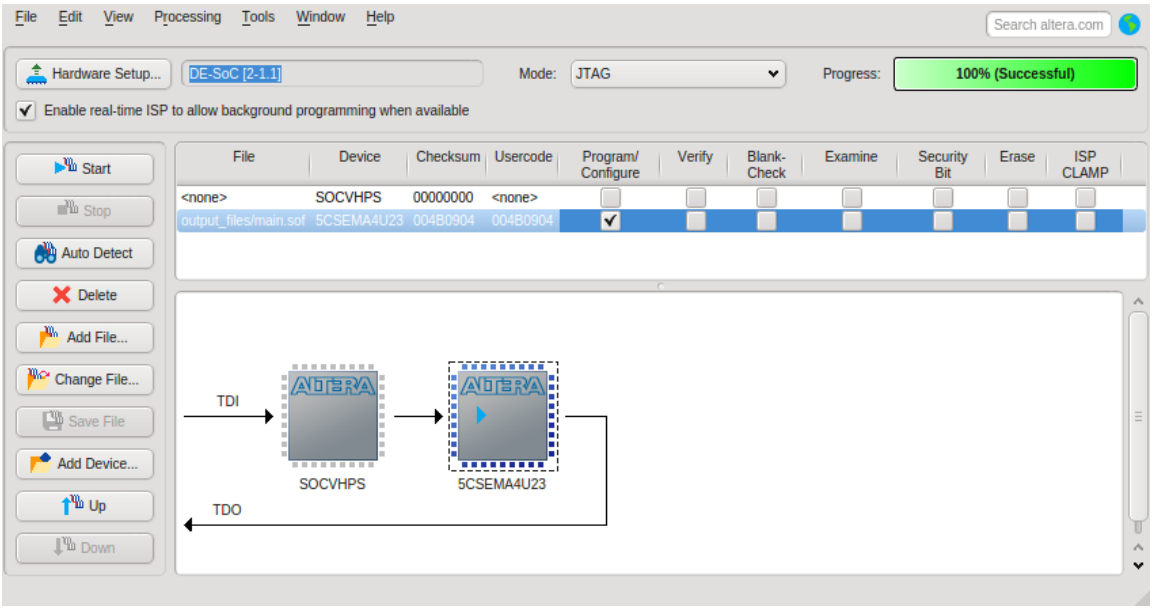


Figura 1 – Quartus Programmer: JTAG USB-Blaster II

Premendo la KEY0 i led dovrebbero accendersi secondo la codifica 0x55 (ovvero uno sì e uno no)

Nota: se il JTAG non viene rilevato occorre configurare UDEV per assegnare correttamente il dispositivo [27].
Nota: potrebbe essere necessario configurare i bit MSEL (*SW10*) per il boot in modo particolare. Per quanto segue la configurazione adottata è tutto ON.

1.2. CREAREZIONE E TEST DEL FILE RBF – RAW BIT FILE

Per caricare il firmware FPGA da LINUX o da UBoot è necessario creare il file .RBF dal file .SOF creato dal progetto Quartus. Per creare questo file si utilizza il seguente comando:

```
quartus_cpf -c main.sof main.rbf
```

Ovviamente occorre che i percorsi dei binari di Quartus siano compresi nel $PATH di sistema, altrimenti il comando va lanciato in modo assoluto. Analogamente, per compilare *main.sof* occorre che questo comando sia lanciato nella directory di output del progetto o dove è contenuto il file main.sof, altrimenti il percorso deve essere assoluto.
È possibile configurare Quartus per creare il file RBF oltre ai file di output .SOF o .JIC (Figura 2).

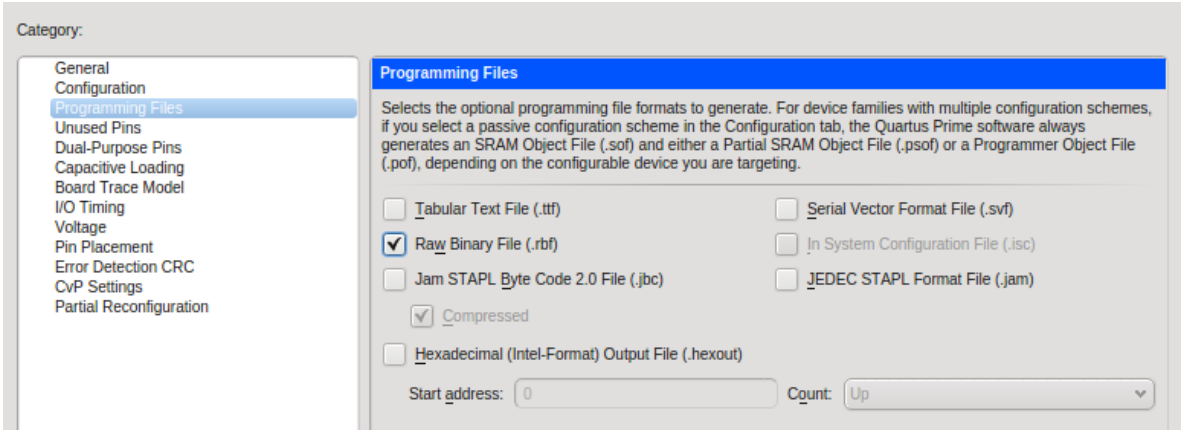


Figura 2 – Raw Bit File: *Settings* → *Device/Board* → *Device and Pin Option*

Il test del firmware può essere fatto programmando direttamente tramite JTAG come detto, oppure l'FPGA può essere programmata da LINUX o da UBoot. In entrambi i casi la CPU deve essere configurata come modalità di boot tale da

poter caricare il firmware FPGA da SD Card o comunque da HPS. Si consiglia di settare switch di SW10 a ON-OFF-ON-OFF-ON (ovvero tutti 01010 come in Figura 3) come indicato in [2].

MSEL[4:0]	Configure Scheme	Description
10010	AS	FPGA configured from EPCS
01010	FPPx32	FPGA configured from HPS software: Linux (default)
00000	FPPx16	FPGA configured from HPS software: U-Boot, with image stored on the SD card, like LXDE Desktop or console Linux with frame buffer edition.

Figura 3 – Configurazione della sorgente di configurazione della FPGA per DE0-Nano-SoC

Quindi occorre:

- copiare nella partizione della SD card il file *main.rbf*.
- Riavviare il sistema e bloccare il timeout di U-Boot e dare i seguenti comandi:

```
fatload mmc 0:1 $fpgadata main.rbf
fpga load 0 $fpgadata <size>
```

dove *fpgadata*=0x2000000 ed è una variabile già definita in UBoot, mentre <*size*> è la dimensione del file caricata in memoria da fatload, in questo caso è 1170232.

Test

Prima del caricamento della FPGA i led dovevano essere accesi in modo flebile. Caricando la FPGA si vedrà che i LED si spegneranno e premendo KEY0 si accenderanno con la codifica 0x55.

1.3. CREAZIONE DEGLI HEADERS C

La creazione degli header è sufficiente lanciare il comando seguente [26]:

```
mkdir headers
sopc-create-header-files CPUInterface.sopcinfo --output-dir headers
```

ovviamente occorre aver settato l'ambiente di compilazione e lanciato il comando nella directory del progetto QSys.
Al termine la directory *headers* verrà popolata con i vari headers C e dovrà essere inclusa nei vari programmi.

2. COMPILAZIONE DEL CODICE BAREMETAL

Gli esempi seguenti permetteranno di creare dei programmi baremetal da caricare prima in U-Boot e poi da LINUX. Per farlo gli indirizzamenti dovranno essere differenti in quanto U-Boot occupa solo la prima parte della RAM, mentre LINUX ne occupa decisamente di più ed occorrerà portare il codice bere metal "più aventi" rispetto a quando lo carica U-Boot. Si noti che non è stato possibile caricare con U-Boot indirizzi di memoria collocati a 128Mb o oltre.
Il software Baremetal di fatto parte su una CPU che può essere già configurata, anche se può ovviamente riconfigurarla se necessario. Se il baremetal viene caricato da U-Boot, si deve tenere conto che la configurazione della CPU0 è affidata prima al preloader e, se necessario, a U-Boot.
Quando sarà LINUX a caricare il Baremetal, la CPU1 verrà risvegliata per eseguire il programma. Fatto salvo per le periferiche shared (ovvero clock=915MHz, timer globale, bus e interfacce FPGA, ecc), la CPU1 potrebbe dover essere riconfigurata qualora si vogliano utilizzare altre periferiche (es: timer della CPU, vettori di interrupt, ecc).
La directory dei sorgenti dovrà contenere la directory precedentemente creata , ovvero *headers*.

Makefile

La compilazione del codice prevede di aver configurato l'ambiente di sviluppo lanciando lo script */<EDS install dir> /embedded/embedded_command_shell.sh*.
La compilazione è affidata al Makefile distribuito con i sorgenti (disponibili su www.lugman.org) la cui parte principale indica quale linker script usare e dove prelevare gli header principali:

```
EXAMPLE_SRC := file1.c io.c
C_SRC       := $(EXAMPLE_SRC)
```

```
LINKER_SCRIPT := cycloneV-dk-ram-calzo.ld

MULTILIBFLAGS := -mcpu=cortex-a9 -mfloat-abi=softfp -mfpu=neon
CFLAGS    := -g -O0 -Wall -Werror -std=c99 $(MULTILIBFLAGS) -I$(HWLIBS_ROOT)/include -I. -
Iqsys_headers -I$(SOCAL_ROOT)
LDFLAGS := -T$(LINKER_SCRIPT) $(MULTILIBFLAGS)
```

Il Makefile genererà un file in formato elf (con estensione .axf) e il binario (.bin) da caricare.

Linker Script

I linker script sono distribuiti con l'abiente di sviluppo:

<instal dir>/host_tools/mentor/gnu/arm/baremetal/arm-altera-eabi/lib/cortex-a9/

<instal dir>/host_tools/mentor/gnu/arm/baremetal/arm-altera-eabi/lib/

Il linker script scelto è quello che prevede di caricare il programma in RAM e non on OnChip RAM.

Nei sorgenti vengono creati due linker script targati "-uboot" e "-linux" che definiscono la memoria come segue:

- *cycloneV-dk-ram-uboot.ld* pone il baremetal appena sopra la on-chip ram, ovvero all'indirizzo 0x100000, dove U-Boot non è presente

```
MEMORY
{
  boot_rom (rx) : ORIGIN = 0xfffd0000, LENGTH = 64K
  oc_ram (rwx) : ORIGIN = 0xffff0000, LENGTH = 64K
  ram (rwx) : ORIGIN = 0x100000, LENGTH = 3M
}
```

- *cycloneV-dk-ram-.linux.ld* pone il baremetal a 0x10000000, ovvero a 128Mb dove LINUX non dovrebbe essere presente ne come kernel ne come applicazioni[1]

```
MEMORY
{
  boot_rom (rx) : ORIGIN = 0xfffd0000, LENGTH = 64K
  oc_ram (rwx) : ORIGIN = 0xffff0000, LENGTH = 64K
  ram (rwx) : ORIGIN = 0x10000000, LENGTH = 3M
}
```

Standard I/O

Per quanto riguarda lo standard input/output, questo è la UART su USB. Il linker script semi-hosted prevede di collegare le funzioni necessarie a che funzioni come la printf e la scanf del C possano restituire il loro output.

Il file *io.c*, copiato dall'esempio *Baremetal_fft* che si può trovare su Rocketboard permette di fare ciò e definisce solo lo standard output con la funzione:

```
int _write(int file, char * ptr, unsigned len, int flag ) {
    /* Fails if not STDOUT */
    if(file != STDOUT_FILENO)
        return -1;

    /* Print each character to UART */
    for(int i=0; i<len; i++) {
        /* Wait until THR is empty*/
        while(1 != ALT_UART_LSR_THRE_GET(alt_read_word(ALT_UART0_LSR_ADDR)));

        /* Write character to THR */
        alt_write_word(ALT_UART0_RBR_THR_DLL_ADDR, ptr[i]);
    }

    /* All printed fine */
    return len;
}
```

Di fatto non fa altro che trasferire sull'UART i caratteri scrivendo sul registro `ALT_UART0_RBR_THR_DLL_ADDR`.

Memory

La memoria vista dal Baremetal sarà di fatto sempre la DDR senza configurazione della MMU, quindi tutti gli indirizzi all'interno degli esempi seguenti saranno fisici.

[1] La SD card utilizzata sembra non occupare più di 30Mb di RAM, kernel e applicazioni comprese.

2.1. TEST 1 – ACCESSO ALLA PERIFERICA LED E A USER-KEY DA UBOOT

Questo esempio prevede che premendo la USER KEY collegata all'ARM il programma stampi la configurazione dei registri e incrementi il valore numerico a cui sono collegati i led. Quando la USER KEY non è premuta, verrà stampato il valore dei LED.

Baremetal01.c

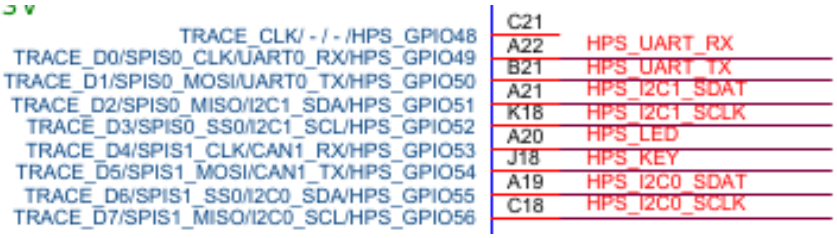
```
#ifndef soc_cv_av
#define soc_cv_av
#endif

#include "socal/hps.h"                    // bring in the view from the processor
#include "headers/hps_0.h"        // bring in the view from the dmas
#include "headers/CPUInterface.h"
#include <stdio.h>
int __auto_semihosting;

#define REG(x) (*((unsigned int*)(x)))

int main()
{
    volatile unsigned long int count = 0;
    volatile unsigned x;
    volatile unsigned *led = (void*)HPS_0_ARM_A9_1_LED_BASE;

    printf("\n\tCALZO - Baremetal Test 01\n\r");

    while(1)
    {
            count++;
            if(count > 2000000)    // circa 1 secondo
            {
                    count = 0;

                    if(REG(HPS_0_ARM_A9_1_HPS_0_GPIO1_BASE+0x50)&(1<<25))      // KEY0 NON premuta
                            printf("ADDR=%08X\tLED=%08X\n\n\r",
                                    HPS_0_ARM_A9_1_LED_BASE, REG(HPS_0_ARM_A9_1_LED_BASE));
                    else
                    {
                            printf("GPIO BASE:\t%08X\t%08X\t%08X\n\r",
                                            REG(HPS_0_ARM_A9_1_HPS_0_GPIO0_BASE+0x50),
                                            REG(HPS_0_ARM_A9_1_HPS_0_GPIO1_BASE+0x50),
                                            REG(HPS_0_ARM_A9_1_HPS_0_GPIO2_BASE+0x50));

                            x = REG(HPS_0_ARM_A9_1_LED_BASE);
                            x++;
                            *led = x;
                    }
            }
    }

    return 0;
}
```

I registri indirizzati sono in primis quelli delle GPIO dell'ARM descritti in [5]. In particolare i registri delle GPIO si dividono in 0, 1 e 2:

Registers in the GPIO module

Module Instance	Base Address
gpio0	0xFF708000
gpio1	0xFF709000
gpio2	0xFF70A000

Ogni registro GPIO è distanziato di 4k l'una dall'altra in quanto gli indirizzi riportati sono delle basi. In realtà i sottoregistri di 32bit l'uno configurano la porta in termini di direzione (In o Out), valore in uscita, valore in ingresso, ecc. Ogni porta GPIO indirizza 29 bit, tranne la GPIO2 che ne mappa 27.
Dalla documentazione della DE0-Nano-SoC si vede che lo USER LED e la USER KEY sono collocati sulla GPIO 53 e 54. Per quanto detto, per controllare la KEY va letto il bit 25 della GPIO1 (54-29=25).



Per leggere lo stato dei pin della porta, occorre leggere il registro GPIO_EXT_PORTA all'offset 0x50. Per questo nel codice viene indirizzato `HPS_0_ARM_A9_1_HPS_0_GPIO1_BASE+0x50`.

gpio_ext_porta on page 22-13	0x50	32	RO	0x0	External Port A Register

Per quanto riguarda i LED, questi sono invece mappati all'indirizzo `HPS_0_ARM_A9_1_LED_BASE = 0xff200000`, ovvero all'inizio dell'indirizzamento del bus AXI Lite. Questo indirizzo è fornito da *headers/CPUInterface.h*, ma è già noto da QSys durante il posizionamento della periferica.
Il programma va compilato con il liker script *–uboot*.

Test con U-Boot

Copiare il file nella partizione fat della SD card, riavviare la scheda, bloccare u-boot.
Per comunicare con l'FPGA è necessario caricare il firmware e abilitare i bridge (AXI Lite):

```
run bridge_disable
fatload mmc 0:1 $fpgadata main.rbf
fpga load 0 $fpgadata <size>
run bridge_enable_handoff
```

Se tutto va bene le applicazioni terminano con *rc = 0x0*.
A questo punto si può caricare il baremetal all'indirizzo definito nel linker script, ovvero 0x100000:

```
fatload mmc 0:1 $0x100000 baremetal01.bin
go 0x100000
```

Il risultato sarà qualche cosa come in Figura 4 dove si vede che il LED non parte da 0 perché erano già state fatte alcune prove e il registro non si azzera se la CPU viene riavviata o se c'è un soft reset. I led ovviamente si accenderanno in sequenza e nel caso in cui si prema KEY0 verrà ancora visualizzato il valore imposto dalla FPGA, che però non modificherà il valore memorizzato nella CPU.

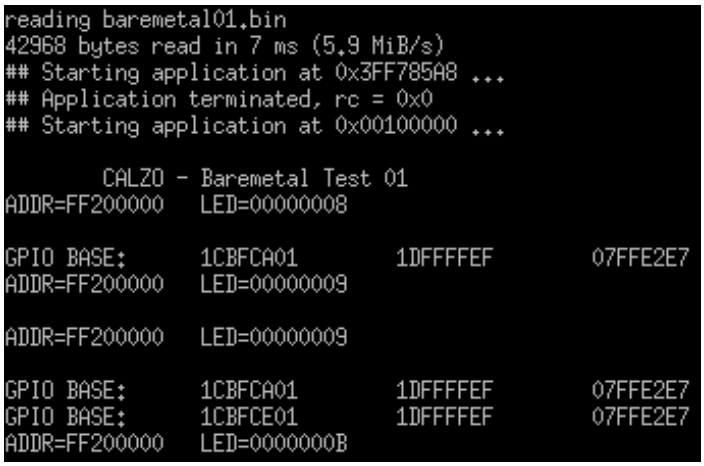


Figura 4 – Rusltato di Baremetal01 caricato da U-Boot premendo qualche volta USER KEY

2.2. TEST 2 – ACCESSO ALLA PERIFERICA LED DA LINUX

Questo test deve semplicemente eliminare l'accesso allo standard output seriale (UART) e deve essere collocato in un'area di memoria in cui LINUX non è stato caricato.

```
#include "socal/hps.h"                  // bring in the view from the processor
#include "headers/hps_0.h"              // bring in the view from the dmas
#include "headers/CPUInterface.h"

#define REG(x) (*((volatile unsigned int*)(x)))

volatile unsigned long int count = 0;
volatile unsigned int x;
volatile unsigned int *led = (void*)HPS_0_ARM_A9_1_LED_BASE;

int main()
{
    while(1) {
            count++;
            if(count > 1156250)   {       // attendo circa 1s
                    count = 0;

                    if(!(REG(HPS_0_ARM_A9_1_HPS_0_GPIO1_BASE+0x50) & (1<<25))  )
                    {
                            x = REG(HPS_0_ARM_A9_1_LED_BASE);
                            x++;
                            *led = x;
                    }
            }
    }

    return 0;
}
```

Il programma è esattamente come il Test1 senza le printf. Quando viene compilato, occorre usare un linker script che collochi il programma in 0x10000000 (o dove il driver prevede di caricarlo). Con questo indirizzo non è possibile testarlo con U-Boot.

2.3. TEST 2 – ACCESSO AL TIMER DI SISTEMA

Un terzo test prevede di portare sui LED il valore del timer 1 (0xFFC09004):

```
#include "socal/hps.h"                  // bring in the view from the processor
#include "headers/hps_0.h"              // bring in the view from the dmas
#include "headers/CPUInterface.h"

#define REG(x) (*((volatile unsigned int*)(x)))

volatile unsigned long int count = 0;
volatile unsigned int x;
volatile unsigned int *led = (void*)HPS_0_ARM_A9_1_LED_BASE;

volatile unsigned int counter  __attribute__((section(".fixed_ram")));

int main() {
    counter = 0;
    while(1) {
            counter++;
            count++;
            x = REG(0xFFC09004);
            *led = x >> 24;
    }

    return 0;
}
```

In questo test non vi è interazione: dopo il caricamento del driver dovrebbero cominciare ad accendersi in sequenza portando i bit più significativi del timer1 sui led.

3. LINUX KERNEL DRIVER – PASSAGGIO DA SMP A AMP

Il capitolo seguente è incentrato sui seguenti argomenti:

- Breve descrizione di come ARM gestisce il boot delle varie CPU secondarie
- Descrizione delle routine principali di LINUX per all'attivazione delle CPU nella fase di boot
- Realizzazione del driver:
    - Creazione del device tree per la definizione della funzionalità
    - Disattivazione della CPU secondaria
    - Caricamento del firmware (binario baremetal)
    - Attivazione della CPU secondaria assegnata al baremetal
    - Ripristino del sistema SMP una volta scaricato il driver

3.1. BOOT ARM MULTICORE

Secondo la documentazione ARM ([32]-4.7) relativa alla creazione di un sistema SMP per aumentare le prestazioni del sistema in un sistema LINUX, si vede come al boot le CPU secondarie siano in attesa di un interrupt da parte della CPU primaria che arriva tramite la chiamata `cpu_up()` (Figura 5).
La prima cosa che si nota è che una volta che la CPU viene svegliata, essa deve configurare tutte le periferiche che la riguardano (e che non sono quindi comuni come il timer globale o alcune periferiche I/O), compresa la MMU. Da qui la prima conclusione: se la CPU esegue un software non legato al sistema operativo, sembra essere completamente indipendente.

Reset Manager

Qualora la CPU in oggetto (Cortex-A9) spenga la CPU secondaria, la CPU viene mantenuta in RESET. Lo stato di reset viene gestito dal registro RSTMGR (0xFFD05000), ovvero il reset manager [5]. Questo registro è anch'esso base per 4k successivi.
In particolare all'offset 0x10 (0xFFD05010) vi è il registro MPUMODRST i cui primi due bit definiscono lo stato di reset della CPU primaria e di quella secondaria. In particolare, la CPU secondaria viene mantenuta in reset scrivendo 1 nel secondo bit (bit 1). Per sbloccare via software la CPU1 del CycloneV basta scrivere 0 in questo registro. Il problema è capire quale codice eseguirà una volta messa in funzione. ■

System Manager

Il secondo registro fondamentale è SYSMGR (0xFFD08000) che indirizza 16k di sottoregistri in quanto controlla di fatto ogni cosa: ID della CPU, interfacce dell'FPGA, e le varie tipologie di Boot.
Ciò che interessa ora sono i registri del gruppo Boot ROM localizzati all'offset 0xC0 (0xFFD080C0). Il campo che interessa principalmente è il CPU1STARTADDR (0xFFD08C4). Leggendo la documentazione [5] si nota che se la CPU1 è mantenuta nello stato di RESET, al suo risveglio essa partirà ad eseguire il codice a partire dal registro di memoria contenuto in questo registro (Figura 6). Se è vero quanto detto ai paragrafi precedenti, questo indirizzo è un valore fisico in quanto la CPU deve ancora configurare (eventualmente) la MMU. ■

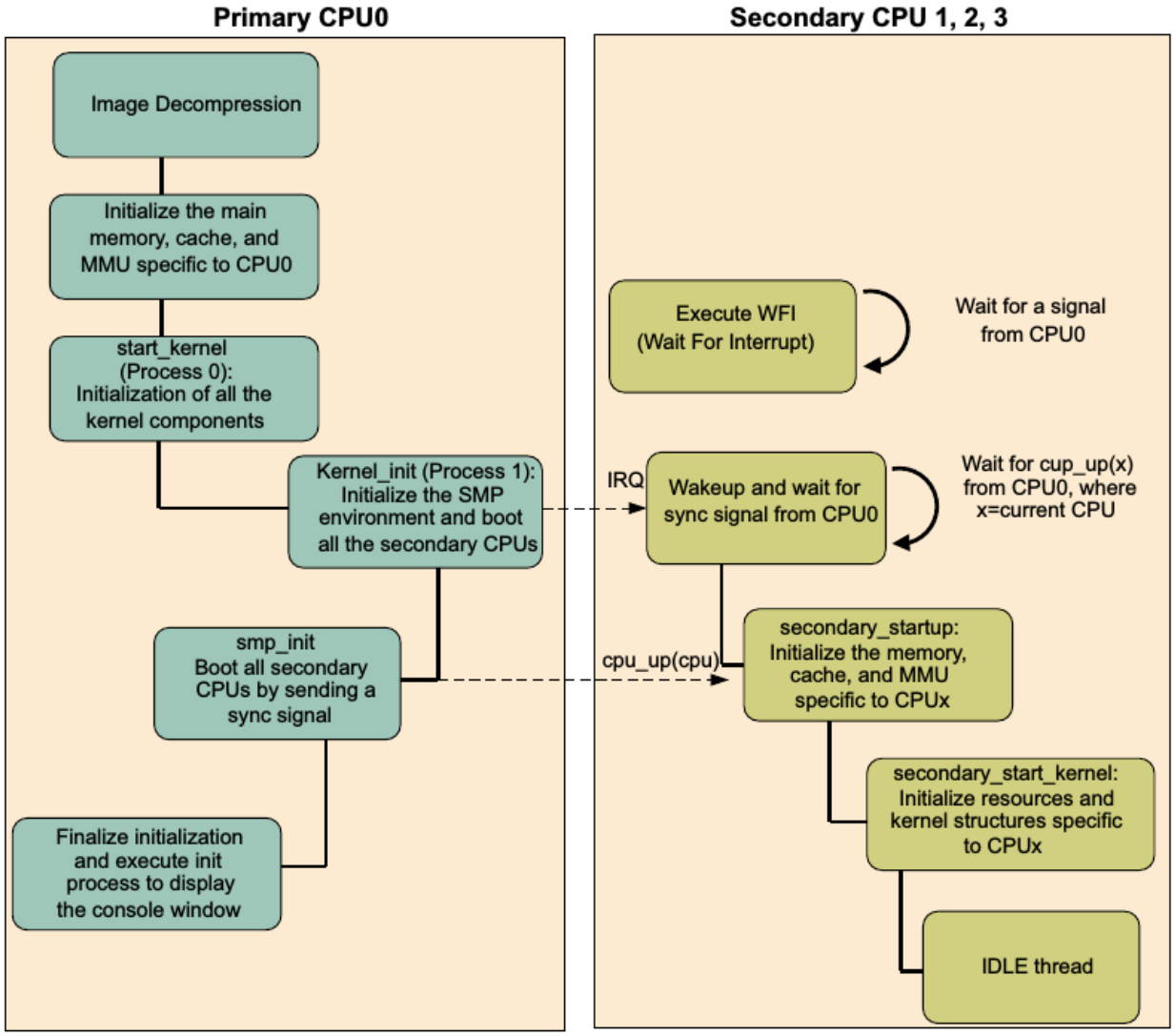


Figura 5 – Linux Multicore Boot Process

cpu1startaddr

When CPU1 is released from reset and the Boot ROM is located at the CPU1 reset exception address (the typical case), the Boot ROM reset handler code reads the address stored in this register and jumps it to hand off execution to user software.

Module Instance	Base Address	Register Address
sysmgr	0xFFD08000	0xFFD080C4

Figura 6 – CPU1 Start Address descritto in [5]

3.2. SISTEMA DI ATTIVAZIONE DELLE CPU SECONDARIE IN LINUX

In questo paragrafo verrà analizzata l'organizzazione del codice del kernel di LINUX relativa alla cpu che si ricorda essere un ARM Cortex-A9 CycloneV SoC Dual Core. I sorgenti in esame sono la versione 4.8 del kernel ottenuta come descritto in [26].

3.2.1. STRUTTURA DEI FILE

In linea generale, il codice base del kernel vero e proprio (non dei driver), ovverosia il codice strettamente legato all'architettura, è contenuto nella sotto-directory *arch/arm/*. La filosofia principale è astrarre le chiamate del kernel in chiamate "standard" che altro non sono che dei wrapper ad altre chiamate fortemente legate all'architettura.
Quindi le altre sottodirectory principali che interessano ora sono:

- *kernel/* contiene tutte le funzioni C che fanno da wrapper a quelle strettamente legate all'architettura. Il file principale per questo documento è c*pu.c* che definisce tutte le routine legate alla gestione delle CPU, come lo scaricamento dei "work", il monitoraggio, l'hot-plug tra cui `cpu_up()` e `cpu_down()` che permettono l'attivazione e la disattivazione delle CPU secondarie. In particolare espone le routine con suffisso "`cpu_`", mentre le routine statiche sono indicate con "`do_cpu_`" le quali sono il vero wraper per le funzioni dipendenti dall'architettura indicate con "`__cpu_`".
- *arch/arm/boot/* contiene le immagini del kernel una volta compilato (zImage, ecc)
- *arch/arm/boot/dts/* contiene i sorgenti e i binari dei vari device-tree. Quelli di interesse sono quelli il cui prefisso è *socfpga* relativi alla *Cyclone5*.
- *arch/arm/kernel/* è la directory più importante. Contiene i file per il boot e tutte le funzioni architecture-dependent chiamate dalle routine in *kernel/*. I principali file sono:
  - *head.S*: asm file che riconfigura la CPU in termini di SMP, la MMU, gli start address delle CPU secondarie, l'endianness, ecc
  - *smp.c*: definisce tutte le routine legate alla gestione delle CPU. Tali routine vengono indicate con il suffisso "`__`" per indicare la dipendenza dall'architettura (es: `__cpu_up()` e la vera funzione che esegue l'attivazione della CPU). A sua volta però molte delle funzioni devono chiamare funzioni legate alla variante della CPU.
- *arch/arm/ mach-socfpga/* è la variante di ARM che si sta utilizzando, ovvero implementa realmente le funzioni legate alla CPU specifica in oggetto. Tra i file vi è:
  - *headsmp.S* da non confondere con *head.S*, è il codice per il "trampolino" delle CPU secondarie.
  - *socfpga.c* contiene tutte le inizializzazioni della CPU (mappatura virtuale dei registri di configurazione, configurazione degli irq, ecc) in funzione del fatto che la CPU sia una Cyclone5 o una Arria10.
  - *platsmp.c* contiene tutte le funzioni specifica per configurare ed eseguire il "trampolino" che permette alla cpu secondarie di partire. La funzione più rilevante che permette di capire cosa accade quando una CPU viene attivata in LINUX è probabilmente la `socfpga_boot_secondary()`.

3.2.2. ROUTINE E PROCEDURE DI SPEGNIMENTO E RIATTIVAZIONE DELLE CPU

L'obiettivo del documento è caricare un software baremetal da linux il che implica che linux parta SMP e poi spenga la CPU; quindi conviene descrivere la modalità con cui prima linux spegne la CPU e poi la riaccende.
Per quanto detto al paragrafo 3.1, occorre però capire prima come LINUX riesca ad accedere ai registri di configurazione a causa della memoria virtuale.

mach-socfpga/socfpga.c: `void __init socfpga_sysmgr_init(void)`

la funzione _sysmgr_init assolve a due compiti:

- Scoprire quali sono i registri da rimappare leggendo il loro indirizzo dal device-tree
- per l'accesso ai registri LINUX ha bisogno di rimappare gli indirizzi fisici in funzione della MMU

Di seguito il codice della funzione:

```
void __init socfpga_sysmgr_init(void)
{
    struct device_node *np;

    np = of_find_compatible_node(NULL, NULL, "altr,sys-mgr");

    if (of_property_read_u32(np, "cpu1-start-addr",
                (u32 *) &socfpga_cpu1start_addr))
        pr_err("SMP: Need cpu1-start-addr in device tree.\n");

    /* Ensure that socfpga_cpu1start_addr is visible to other CPUs */
    smp_wmb();
    sync_cache_w(&socfpga_cpu1start_addr);

    sys_manager_base_addr = of_iomap(np, 0);
```

```
        np = of_find_compatible_node(NULL, NULL, "altr,rst-mgr");
        rst_manager_base_addr = of_iomap(np, 0);

        np = of_find_compatible_node(NULL, NULL, "altr,clk-mgr");
        clkmgr_base_addr = of_iomap(np, 0);
        WARN_ON(!clkmgr_base_addr);

        np = of_find_compatible_node(NULL, NULL, "altr,sdr-ctl");
        sdr_ctl_base_addr = of_iomap(np, 0);
    }
```

Questa funzione principalmente fa questo:

- individua il nodo del device-tree tramite `of_find_compatible_node()` che risponda ai criteri di compatibilità "*altr,sys-mgr*".
- Dal nodo vengono poi estratte le varie proprietà:
  - ✓ Carica il valore dell'indirizzo del registro CPU1STARTA DDR (ovvero 0xFFD080C4)
  - ✓ Rimappa tramite `of_iomap()` il registro system manager
  - ✓ Rimappa tramite `of_iomap()` il registro system reset
  - ✓ Rimappa tramite `of_iomap()` il timer di CPU

  Si noti che la rimappatura `of_iomap()` potrebbe essere eseguita anche con `phy_to_virt()` essendo noti dal manuale gli indirizzi fisici dei registri, ma questo metodo è più generale. Inoltre durante lo sviluppo del driver la funzione `phy_to_virt()` non ha funzionato su tutti i registri passati senza accedere al device-tree. Quindi anche nel proprio driver conviene utilizzare questa struttura.

Device-Tree per l'acesso ai registri: *socfpga.dtsi* e *socfpga_cyclone5.dtsi*

Il file *socfpga.dtsi* è il principale file da includere nel proprio device-tree in quanto definisce tutte le strutture standard della CPU in particolare dove sono collocati i registri principali; questo file vine incluso in *socfpga_cyclone5.dtsi* a sua volta incluso dei device tree veri e propri. Per informazioni più approfondite sul device-tree si rimanda a [25] e [33].
Sotto il nodo ***soc*** vi sono i nodi relativi ai vari registri:

- *CPU1 Start Address:* lettura valore registro

```
sysmgr@ffd08000 {
    cpu1-start-addr = <0xffd080c4>;
};
```

questa proprietà viene letta solo come valore e non viene rimappata (almeno inizialmente).

- *Reset Manager:* registro rimappato

```
rst: rstmgr@ffd05000 {
    #reset-cells = <1>;
    compatible = "altr,rst-mgr";
    reg = <0xffd05000 0x1000>;
    altr,modrst-offset = <0x10>;
};
```

il registro viene collocato all'indirizzo di memoria 0xFFD05000 a blocchi di 4k ed in particolare il registro in questione è collocato all'offset 0x10 esattamente come indicato in [5].

- *System Manager:* registro rimappato

```
sysmgr: sysmgr@ffd08000 {
    compatible = "altr,sys-mgr", "syscon";
    reg = <0xffd08000 0x4000>;
}
```

cpu_down(n) in kernel/cpu.c e arm/kernel/smp.c

Spegne la cpu n-esima dove n=0 indica la cpu primaria. Se è attiva l'opzione di hotplug-cpu è attiva e se il numero di cpu online è maggiore di 1, allora si può procedere allo spegnimento della CPU.
Prima di spegnere la CPU occorre scaricarla dei task ad essa assegnati. Quindi viene chiamata `cpuhp_down_callbacks()` che chiama, tramite una struttura di puntatori (`step->teardown`), la funzione `trardown_cpu()`: questa funzione esegue alcune operazione tra le quali aspetta la terminazione dei task ed infine chiama la funzione `__cpu_die()` (*arm/kernel/smp.c*) che effettivamente rimuove la CPU.

`__cpu_die()` chiama a sua volta `platform_cpu_kill()` che a sua volta chiama la vera funzione di kill tramite l'accesso alla struttura `smp_ops.cpu_kill()`.
Qui il flusso del codice si perde perché sembra che le chiamate siano assegnate o in modo statico (tramite linker script) o tramite device tree. Ad ogni modo si deve arrivare ad invocare le chiamate definite nelle strutture di *platsmp.c*, in particolare `socfpga_cpu_kill()` settata nella struttura `static const struct smp_operations` **`socfpga_smp_ops`**.

platsmp.c: socfga_cpu_kill

Questa funzione è una funzione dummy che ritorna sempre 1 serve solo per ragioni di compatibilità in quanto la CPU si spegne automaticamente chiamando l'istruzione `WFI` (wait for interrupt).
Non è chiaro però chi pone in RESET lo stato della CPU.

cpu_up(n) in kernel/cpu.c e arm/kernel/smp.c

Analogamente a `cpu_down()`, `cpu_up()` riattiva l'n-esima CPU chiamando `do_cpu_up()` e quindi `__cpu_up()` (*arm/kernel/smp.c*). La CPU viene attivata chiamando `smp_ops.smp_boot_secondary(cpu, idle)`. Questa call per Cyclone V è collegata alla funzione `socfpga_boot_secondary()` (*arm/mach-socfpga/platsmp.c*):

```
static int socfpga_boot_secondary(unsigned int cpu, struct task_struct *idle)
{
    int trampoline_size = &secondary_trampoline_end - &secondary_trampoline;

    if (socfpga_cpu1start_addr) {
        /* This will put CPU #1 into reset. */
        writel(RSTMGR_MPUMODRST_CPU1,
               rst_manager_base_addr + SOCFPGA_RSTMGR_MODMPURST);

        memcpy(phys_to_virt(0), &secondary_trampoline, trampoline_size);

        writel(virt_to_phys(secondary_startup),
               sys_manager_base_addr + (socfpga_cpu1start_addr & 0x000000ff));

        flush_cache_all();
        smp_wmb();
        outer_clean_range(0, trampoline_size);

        /* This will release CPU #1 out of reset. */
        writel(0, rst_manager_base_addr + SOCFPGA_RSTMGR_MODMPURST);
    }

    return 0;
}
```

La funzione viene eseguita solo se il registro `socfpga_cpu1start_addr` è stato configurato (cosa fatta in `socfpga_sysmgr_init`), viene resettata la CPU1 per precauzione, quindi viene caricato il codice contenuto in *headsmp.S*:

```
ENTRY(secondary_trampoline)
ARM_BE8(setend  be)
        adr     r0, 1f
        ldmia   r0, {r1, r2}
        sub     r2, r2, #PAGE_OFFSET
        ldr     r3, [r2]
        ldr     r4, [r3]
ARM_BE8(rev     r4, r4)
        bx      r4

        .align
1:      .long   .
        .long   socfpga_cpu1start_addr
ENTRY(secondary_trampoline_end)
```

Questo codice è molto simile a quello contenuto in arm/kernel e serve solo per poter calcolare l'indirizzo fisico da caricare in CPU1STARTADDR. L'ultima istruzione avvia la CPU1.
Questo codice implica ciò che era già noto dalla documentazione [5], ovvero che quando la CPU esce dallo stato di reset, tutto ciò che la riguarda deve essere configurato. Tra le varie periferiche vi è la MMU il che implica che CPU1STARTADDR sia necessariamente un indirizzo fisico.

3.3. AMP DRIVER

Specifica

Il driver deve:

- Spegnere la CPU secondaria (CPU1 in quanto il Cyclone V è un dual core)
- Caricare il codice baremetal dal filesystem utilizzando le firmware API leggendo il nome del file dal DTB
- Copiare il baremetal all'indirizzo di memoria fisico predefinito
- Configurare la CPU1 per partire eseguendo il codice dall'indirizzo di caricamento del baremetal
- Avviare la CPU agendo sui registri
- Ripristinare il sistema SMP qualora il driver venga disinstallato

Il driver verrà inserito nel kernel in *driver/calzo-amp/*. Le varie funzioni avranno suffisso `ct_` (*calzo test*). Il software baremetal dovrà essere compilato per essere caricato all'indirizzo fisico `0x10000000`.

3.3.1. IMPLEMENTAZIONE

Device Tree

Occorre ricreare il device tree per definire le varie informazioni relative al firmware. Per farlo è stato modificato il file DTS relativo alla DE0-Nano-SoC in *arch/arm/boot/dts/socfpga_cyclone5_de0_sockit.dts*. In particolare è stato aggiunto:

```
soc {
            calzo_test: calzo@0 {
                    compatible = "altr,calzotest";
                    reg = <0x10000000 0x300000>;
                    firmware = "BareMetal";
            };
    };
```

Questi parametri verranno letti dal driver.

Inizializzazione del driver tramite platform_driver

L'inizializzazione del driver avviene in modo standard utilizzando la struttura `platform_driver`:

```
static struct of_device_id calzo_test_match[] =
{
    { .compatible = "altr,calzotest", },
    { },
};

MODULE_DEVICE_TABLE(of, calzo_test_match);

static struct platform_driver calzo_test_diver =
{
    .probe = ct_probe,
    .remove = ct_remove,
    .driver = {
            .name = "calzo_test",
            .owner = THIS_MODULE,
            .of_match_table = calzo_test_match,
    },
};

module_platform_driver(calzo_test_diver);
```

L'utilizzo delle funzioni *.probe* e *.remove* al posto delle funzioni di `__init` o `__exit` (inizializzate come `module_init()` e `module_exit()` [34], [35]) è preferibile in quanto è più semplice l'accesso al device tree. La compatibilità con il device tree è garantita da "*altr,calzotest*".

Funzione di inizializzazione `ct_probe()`

Questa funzione ha il principale compito di eseguire il corretto caricamento in memoria del firmware. Per farlo deve:

- Configurare l'accesso ai registri della CPU

- Spegnere la CPU1
- Settare l'indirizzo di memoria di destinazione
- Caricare il firmware tramite le opportune API
- Copia il binario in RAM
- Configura e avvia la CPU1

La funzione utilizza un puntatore ai dati del firmware dove verrà copiato il binario e la struttura firmware che verrà utilizzata per caricare temporaneamente dalle funzioni *linux/firmware.h*.

```
static unsigned char *MyFirmwareAddr;
const struct firmware *fw;
```

La funzione è la seguente (sono state cancellate le printk di bug):

```
static int ct_probe(struct platform_device *pdev)
{
    const unsigned char *prop;    // property
    int i;
```

Viene chiamata la funzione per mappare i registri della CPU. Questa funzione è quasi identica a socfpga_sysmgr_init *del kernel standard*

```
    ct_sameas_socfpga_sysmgr_init();
```

Nel kernel Linux l'unico registro che non viene mappato è cpu1start *che può essere rimappato con* ioremap. *Si noti che la rimappatura di tutti i registri senza passare per il DTB ha causato degli errori, tranne per questo registio*

```
    cpu1start = ioremap(cpu1start_addr, 0x4);

    // Print dei registri
    […]
```

Viene quindi spenta la CPU secondaria (CPU1)

```
    cpu_down(1);
```

Siccome è noto dove occorre caricare il firmware, viene inizializzato il puntatore all'area di memoria di destinazione (0x10000000). Questo indirizzo fisico viene convertito nell'equivalente virtuale.

```
    ///     http://www.makelinux.net/ldd3/chp-9-sect-4
    MyFirmwareAddr = (unsigned char *)phys_to_virt(FIRMWARE_ADDR);
    […]
```

Per sapere quale file va caricato, viene interrogato il DTB alla ricerca della proprietà "firmware". Il nome del file verrà cercato in automatico dal kernel cercando nelle directory di default, tra cui /lib/firmware/

```
    prop = of_get_property(pdev->dev.of_node, "firmware", NULL);
    if(prop)
    {
        […]
```

Caricamento del firmware in un buffer temporaneo: la struttura contiene il puntatore ai binari del baremetal (se il file viene trovato) e la dimensione totale in byte del file appena caricato.

```
        if(request_firmware(&fw, prop, NULL) < 0)
            printk(KERN_INFO "Error while loading firmware.\n");
```

Copia il contenuto del buffer (ovvero il baremetal) all'indirizzo fisico tramite il puntatore virtuale a tale indirizzo

```
        for(i = 0; i < fw->size; i++)
            MyFirmwareAddr[i] = fw->data[i];
```

Il firmware viene poi rilasciato per liberare la memoria (che potrebbe essere parecchia se il file dati caricato è molto grande)

```
        release_firmware(fw);
        fw = NULL;
    }
    else
        printk(KERN_INFO "Firmware property not found\n");
```

Indico al kernel che il blocco di memoria fisico utilizzato è stato allocato onde evitare che il kernel sovrascriva questa area di ram. La memoria così allocata andrà poi liberata con kmemleak_free *e non con* kfree

```
    kmemleak_alloc(MyFirmwareAddr, FIRMWARE_MAX_SIZE, 1, GFP_ATOMIC);
```

Boot della CPU1, ma senza comunicarlo al kernel. Per farlo occorre una funzione apposita simile a socfpga_boot_secondary

```
    ct_sameas_socfpga_boot_secondary(FIRMWARE_ADDR);
    return 0;
```

```
}
```

Configurazione dei registri: `ct_sameas_socfpga_sysmgr_init()`

Questa funzione esegue le stesse funzioni della `socfpga_sysmgr_init()`, ovvero inizializza i puntatori ai registri di configurazione. I puntatori sono i seguenti:

```
static void __iomem *sys_manager_base_addr;
static void __iomem *rst_manager_base_addr;
static void __iomem *sdr_ctl_base_addr;
static void __iomem *clkmgr_base_addr;
static unsigned int __iomem *cpu1start = NULL;
```

è gli indirizzi fisici associati sono:

```
socfpga_cpu1start_addr =          0xFFD080C4
clkmgr_base_addr =                0xFFD04000
rst_manager_base_addr =           0xFFD05000
sdr_ctl_base_addr =               0xFFC25000
sys_manager_base_addr =           0xFFD08000
```

La funzione però rimappa virtualmente i registri leggendo l'indirizzo dal device tree:

```
void ct_sameas_socfpga_sysmgr_init(void)
{
    struct device_node *np;
```

viene prelevato il device node per accedere poi alle varie proprietà

```
    np = of_find_compatible_node(NULL, NULL, "altr,sys-mgr");
```

il registro cpu1start_addr viene letto come normale proprietà e non come puntatore

```
    if (of_property_read_u32(np, "cpu1-start-addr",
            (u32 *) &cpu1start_addr))
            pr_err("SMP: Need cpu1-start-addr in device tree.\n");
```

segue la configurazione del registro system manager, reset manager, clock e control

```
    sys_manager_base_addr = of_iomap(np, 0);

    np = of_find_compatible_node(NULL, NULL, "altr,rst-mgr");
    rst_manager_base_addr = of_iomap(np, 0);

    np = of_find_compatible_node(NULL, NULL, "altr,clk-mgr");
    clkmgr_base_addr = of_iomap(np, 0);

    np = of_find_compatible_node(NULL, NULL, "altr,sdr-ctl");
    sdr_ctl_base_addr = of_iomap(np, 0);
}
```

Start della CPU1: `ct_sameas_socfpga_boot_secondary(__u32 jump_addr)`

Questa è la procedura più importante e come la `socfpga_boot_secondary()` si occupa di avviare la cpu secondaria. Siccome tutti gli indirizzi di boot sono noti, e siccome la CPU1 quando parte legge indirizzi fisici, l'unica cosa che occorre fare è settare il registro CPU1STARTADDR con l'indirizzo fisico del baremetal.

```
static int ct_sameas_socfpga_boot_secondary(__u32 jump_addr)
{
    if (cpu1start) {
```

ci si assicura che la CPU1 sia nello stato di RESET

```
            /// This will put CPU #1 into reset.
            writel(RSTMGR_MPUMODRST_CPU1,
                   rst_manager_base_addr + SOCFPGA_RSTMGR_MODMPURST);
```

il registro CPU1STARTADDR *viene scritto con il valore fisico a cui si vuole*

```
            *cpu1start = jump_addr;

            flush_cache_all();
            smp_wmb();
            outer_clean_range(FIRMWARE_ADDR, FIRMWARE_MAX_SIZE);
```

Avvio della CPU1 scrivendo 0 nel registro MPUMODRST

```
            /// This will release CPU #1 out of reset.
```

```
            writel(0, rst_manager_base_addr + SOCFPGA_RSTMGR_MODMPURST);
        }
        return 0;
    }
```

Si noti che le funzioni `writel` o `readl` permettono di scrivere o leggere una variabile long a 32 bit. La scrittura potrebbe essere fatta anche direttamente accedendo tramite i puntatori, ma per ragioni di portabilità è meglio utilizzare queste funzioni. Per quanto riguarda invece le macro come SOCFPGA_RSTMGR_MODMPURST, queste sono definite in *mach-socfpga/core.h*.

Funzione di rimozione: `ct_remove()`

Rimuove il driver e libera la memoria:

```
    static int ct_remove(struct platform_device *pdev)
    {
Riattivazione della CPU dal punto di vista del kernel
        cpu_up(1);
tutto ciò che si alloca con kmemleak_ va liverato con kmemleak_free e non con kfree
        kmemleak_free(MyFirmwareAddr);

        return 0;
    }
```

3.3.2. VARIANTE PER CARICAMENTO ELF

Un'altra modalità di caricamento richiede il formato ELF. La compilazione dei baremetal porta di default ad avere i file *.axf* che sono sempre di tipo ELF.
Le soluzioni AMP in commercio tipicamente caricano il formato ELF. LINUX mette ovviamente a disposizione le API per il caricamento del formato. Il caricamento del binario avveniva copiando il firmware in modo diretto tramite un for. In questo caso si utilizza una funzione apposita in quanto il metodo è più complesso.

Funzione `ct_elf_load_segment` per il caricamento del firmware da ELF

La funzione che segue è stata adattata dal driver *remote_proc* per XILINX. Il sistema funziona e sembra leggermente più stabile, ma presenta sempre l'inconveniente (ad oggi non risolto) di dover caricare due volte il driver.

```
    static int ct_elf_load_segment(const struct firmware *fw, void* virtual_fw_addr)
    {
        struct elf32_hdr      *elf_header;
        struct elf32_phdr     *p_header;
        int i, ret = 0;
        const u8 *elf_data = fw->data;

        elf_header = (struct elf32_hdr*) elf_data;
        p_header = (struct elf32_phdr*) (elf_data + elf_header->e_phoff);

        /* Processa i vari segmenti dell'ELF */
        for(i=0; i<elf_header->e_phnum; i++, p_header++)    {
                u32 da = p_header->p_paddr;
                u32 memsz = p_header->p_memsz;
                u32 filesz = p_header->p_filesz;
                u32 offset = p_header->p_offset;
                void *ptr;

                if (p_header->p_type != PT_LOAD)
                        continue;

                printk(KERN_INFO "phdr: type %d da 0x%x memsz 0x%x filesz 0x%x\n",
                        p_header->p_type, da, memsz, filesz);

                if (filesz > memsz) {
                        printk(KERN_INFO "bad phdr filesz 0x%x memsz 0x%x\n", filesz, memsz);
                        ret = -EINVAL;
                        break;
                }

                if (offset + filesz > fw->size) {
```

```
                printk(KERN_INFO "truncated fw: need 0x%x avail 0x%zx\n",
                        offset + filesz, fw->size);
                ret = -EINVAL;
                break;
        }

        /* grab the kernel address for this device address */
        ptr = (void*)virtual_fw_addr;

        /* put the segment where the remote processor expects it */
        if (p_header->p_filesz)
                memcpy(ptr, elf_data + p_header->p_offset, filesz);

        /* Zero out remaining memory for this segment. */
        if (memsz > filesz)
                memset(ptr + filesz, 0, memsz - filesz);
    }

    return ret;
}
```

Nell'esempio 03 distribuito con i sorgenti, il firmware da caricare avrà il nome *bm.elf* e non sarà passato tramite DTB. Questa funzione viene chiamata in `ct_probe()` come segue:

```
ct_elf_load_segment(fw, (void*)MyFirmwareAddr);
```

3.3.3. COMPILAZIONE

Si ipotizza che la directory dei driver sia la sottodirectory dei sorgente del kernel *driver/calzo-amp/*:

- Editare il Makefile della directory superiore (*driver/Makefile*) includendo in fondo la directory dei sorgenti, indicando con *obj-m* il fatto che il driver verrà compilato come modulo:
  `obj-m += calzo-amp/`
- Editere il Makefile dei sorgenti (*driver/calzo-amp/Makefile*) indicando che il driver è un modulo:
  `obj-m += calzo_test01.o`
- Compilare i driver ricordandosi di abilitare il compilatore [26]:
  `make ARCH=arm CROSS_COMPILE=arm-altera-eabi- -j8 modules`
- Analogamente compilare il device tree (ipotizzando il file *arch/arm/boot/dtb/socfpga_calzo.dts*)
  `make ARCH=arm CROSS_COMPILE=arm-altera-eabi- -j8 socfpga_calzo.dtb`

Al termine della compilazione il driver sarà in *driver/calzo-amp/calzo_test01.ko*. Il file DTB sarà invece *arch/arm/boot/dtb/socfpga_calzo.dtb*. Il DTB andrà copiato nella directory di boot (FAT della SD card) con il nome *socfpga.dtb*.

3.3.4. TEST

Per il test si prevede di copiare numerosi file. Quindi se si usa una SD d default occorre settare la password di root (di solito assente) o di un eventuale utente creato.
I comandi alla scheda verranno invece dati sulla console seriale; si consiglia come interfaccia PuTTY.
Ciò che dovrà accadere è che una volta caricato il driver si spenga una CPU. Per vederlo basta fare leggere il file */proc/cpuinfo* che dovrà mostrare due CPU prima del caricamento del driver e una sola dopo. Inoltre premendo la USER-KEY i led lato FPGA si dovranno accendere.

> **Nota**: la parte FPGA deve essere già inizializzata, ovvero deve essere caricato il firmware e i bridge tra CPU e FPGA devono essere abilitati come indicato a pag. 10 o in [29]. Dopo di che occorre avviare LINUX.

Eseguire il test su LINUX

- Compilare il binario baremetal
- Copiare il binario sulla scheda rinominandolo in */lib/firmware/BareMetal* (o con il nome definito nel device tree). Se */lib/firmware* non esiste, occorre ovviamente crearla.
  `scp baremetal02.bin root@10.20.0.5:/lib/firmware/BareMetal`

- Copiare il driver nella directory di root (o dove si preferisce):

```
scp drivers/calzo-amp/calzo_test01.ko root@10.20.0.5:/home/root/
```

- Ora sulla scheda DE0-Nano-SoC caricare il driver con `insmod calzo_test01.ko`. Il driver dovrebbe venire caricato e linux dovrebbe restituire il prompt come in figura:

```
root@socfpga:~#
root@socfpga:~# insmod calzo_test01.ko
[   71.319609] Calzo test: PROBE
[   71.322573] Initialize CycloneV registers...
[   71.327085] socfpga_cpu1start_addr = 0xFFD080C4
[   71.331598] cpu1start = 0xF09C40C4
[   71.335004] sys_manager_base_addr = 0xF09B8000
[   71.339435] rst_manager_base_addr = 0xF09BE000
[   71.343858] sdr_ctl_base_addr = 0xF09C2000
[   71.347949] cpu1start value = 0x001017C0
[   71.351861] cpu1start value = 0x001017C0
[   71.384795] CPU1: shutdown
[   71.387773] Physical Addr 0x10000000 = Virtual 0xD0000000
[   71.393152] Firmware <BareMetal> will be loaded.
[   71.398098] Copy firware into RAM
[   71.401424] Calzo: Prepare to JUMP
[   71.404872] Calzo: jump to 10000000
root@socfpga:~#
```

Tutte le varie scritte sono solo di debug. Vedere i sorgenti per maggiori informazioni.

- A questo punto verificare la presenza di una sola cpu attiva con *cat /proc/cpuinfo*:

```
[ 1182.401433] Calzo: Prepare to JUMP
[ 1182.404880] Calzo: jump to 10000000
root@socfpga:~#
root@socfpga:~# cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 0 (v7l)
BogoMIPS        : 200.00
Features        : half thumb fastmult vfp edsp thumbee neon vfpv3 tls vfpd32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x3
CPU part        : 0xc09
CPU revision    : 0

Hardware        : Altera SOCFPGA
Revision        : 0000
Serial          : 0000000000000000
root@socfpga:~#
```

- Rimuovendo il driver (`rmmod calzo_test01.ko`) il sistema torna dual core:

```
root@socfpga:~# rmmod calzo_test01.ko
[ 1375.019219] Calzo test: REMOVE
root@socfpga:~# cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 0 (v7l)
BogoMIPS        : 200.00
Features        : half thumb fastmult vfp edsp thumbee neon vfpv3 tls vfpd32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x3
CPU part        : 0xc09
CPU revision    : 0

processor       : 1
model name      : ARMv7 Processor rev 0 (v7l)
BogoMIPS        : 200.00
Features        : half thumb fastmult vfp edsp thumbee neon vfpv3 tls vfpd32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x3
CPU part        : 0xc09
CPU revision    : 0

Hardware        : Altera SOCFPGA
Revision        : 0000
Serial          : 0000000000000000
root@socfpga:~#
```

Test di caricamento ELF

Il caricamento del firmware tramite ELF darà il seguente risultato:

```
root@socfpga:~# insmod calzo_test03.ko
[ 1003.539648] Calzo test: PROBE
[ 1003.542613] Initialize CycloneV registers...
[ 1003.547127] socfpga_cpu1start_addr = 0xFFD080C4
[ 1003.551641] cpu1start = 0xF09D80C4
[ 1003.555047] sys_manager_base_addr = 0xF09D0000
[ 1003.559477] rst_manager_base_addr = 0xF09CC000
[ 1003.563899] sdr_ctl_base_addr = 0xF09D6000
[ 1003.567989] cpu1start value = 0x001017C0
[ 1003.571900] cpu1start value = 0x001017C0
[ 1003.604789] CPU1: shutdown
[ 1003.607777] cpu_down(1) = 0
[ 1003.610564] Physical Addr 0x20000000 = Virtual 0xE0000000
[ 1003.615976] Firmware <bm.elf> will be loaded.
[ 1003.620904] Copy firware into RAM
[ 1003.624216] phdr: type 1 da 0x20000000 memsz 0x460 filesz 0x438
[ 1003.630165] phdr: type 1 da 0x20020000 memsz 0x4 filesz 0x0
[ 1003.635730] truncated fw: need 0x20000 avail 0x11b9c
[ 1003.640686] Calzo: Prepare to JUMP
[ 1003.644095] Calzo: jump to 20000000
root@socfpga:~#
```

Si noti che è cambiato l'indirizzo di caricamento (ora 0x20000000) solo per eseguire un test. Inoltre quasi sicuramente occorrerà caricare il driver in questo modo:

- Caricare il driver con `insmod calzo_test03.ko`
- Il sistema potrebbe non accendere i led, segno che il sistema è instabile anche se ha restituito il prompt
- Riavviare (`reboot`) oppure se si pianta riavviare con un warm reset (USER KEY2)
- Ricaricare il driver: questa volta tutto deve funzionare.

3.3.5. BUG E MIGLIORAMENTI

- Può capitare che la CPU si blocchi dopo il Jump sul core secondario. Non è chiaro il motivo di ciò visto che è una condizione rara che tipicamente si verifica all'inizio, ma si è notato che:
  - Se l'FPGA viene programmata da JTAG, il problema è molto meno frequente
  - Spesso occorre premere KEY1 per resettare l'FPGA
- Per sbloccare l'esecuzione occorre (normalmente) installare la prima volta il driver e il sistema si pianterà. Resettare quindi il bus (KEY1) e riavviare la CPU con la USERKEY3. Al riavvio di Linux rieseguire la procedura. Può accadere che il sistema non si pianti, ma che i LED non funzionino. Questa è una situazione instabile che necessita comunque il riavvio. Il motivo di tutto ciò non è al momento chiaro.

- I dati relativi all'indirizzo di caricamento e alla dimensione del firmware potrebbero essere caricati da DTB
- Linux potrebbe essere caricato ad un indirizzo di ram molto superiore a 0x0 e lasciare la ram più bassa a disposizione del firmware, o viceversa caricare il firmware nell'ultima parte della ram (questo dovrebbe evitare di dover rimappare il "trampolino" per le CPU secondarie collocato a 0x4000).

4. COSA MANCA

Avere due CPU che non dialogano è nella stragrande maggioranza dei casi inutile. È bene che il sistema di alto livello (LINUX) dialoghi con il baremetal per scambiare informazioni, stati, configurazioni, ecc.
Tipicamente occorre sviluppare un sistema per il dialogo con le due CPU che può avvenire a vari livelli di cui seguono alcune idee non testate.

/proc o mmap

La directory virtuale */proc* potrebbe mettere a diposizione l'accesso ad un'area di memoria condivisa con il baremetal. Analogamente potrebbe fare mmap, ma tramite proc la cosa sarebbe più flessibile.
Vantaggi: è semplicissimo
Svantaggi: le interrogazioni sarebbero tutte in polling, e sull'area di memoria dovrebbe essere disabilitata la cache, cosa che potrebbe rallentare un po' le prestazioni in favore della sincronizzazione dei dati

SWI

Le Software Interrupt vengono utilizzate per invocare le chiamate di sistema (ovverosia passare in spazio kernel) modificando lo stato della CPU.
In generale la SWI possono essere configurate con un normale interrupt dove l'unico accorgimento è capire quale SWI è stata chiamata. Come per tutti gli interrupt esiste un registro che indica quale CPU deve soddisfare la SWI.
Si potrebbe quindi creare una SWI invocabile da LINUX soddisfatta dal baremetal e viceversa e i dati potrebbero essere passati tramite shared ram (o al limite alcuni registri).
Vantaggi: estremamente veloce e versatile, permette di creare un sistema di comunicazione su misura
Svantaggi: sistema molto invasivo che obbliga a modificare LINUX a basso livello riducendo la portabilità tra i vari kernel. Inoltre tutto il lavoro è a carico dello sviluppatore.

Virtual IO

LINUX mette già a disposizione una serie di sistemi di comunicazione tra cui i virtual I/O (virt_io). Queste API sono lo standard de facto delle soluzioni AMP in commercio (come Texas Instruments e Xilinx) nei driver remote_proc. Di fatto si tratta di aprire un canale di comunicazione a messaggi uno verso LINUX e uno da LINUX.
Vantaggi: si utilizza un sistema già presente e collaudato implementabile direttamente nel driver AMP/remote_proc
Svantaggi: occorre implementare lato baremetal la stessa cosa e potrebbe non essere banale

5. RIFERIMENTI E LINK

[1] Terasic DE0 Nano SoC board
[2] Manuale DE0 Nano SoC – DE0 Nano SoC User Manual
[3] Cyclone V Device Handbook volume 1 – Device Interface and Integration – cv_5v2.pdf
[4] Cyclone V Device Handbook volume 2 – Transceivers – cv_5v3.pdf
[5] Cyclone V Device Handbook volume 3 – Hard Processor System – Technical Reference Manual – cv_5v4.pdf (indirizzo alternativo versione 2012)
[6] Cyclone V Address Map
[7] *DE0-Nano-SoC_My_First_FPGA.pdf*: Tutorial Terasic (distribuiti con la documentazione della scheda)
[8] *DE0-Nano-SoC_My_First_HPS.pdf*: Tutorial Terasic (distribuiti con la documentazione della scheda)
[9] *DE0-Nano-SoC_My_First_HPS-Fpga.pdf*: Tutorial Terasic (distribuiti con la documentazione della scheda)
[10] Lista dei workshop di riferimento: Altera SoC Workshop Series
[11] RocketBoards Workshop 1: WS1– Intro to Altera SoCDevices for SW Developers
[12] RocketBoards Workshop 2: WS2 Linux KernelIntroduction for Altera SoCDevices
[13] RocketBoards Workshop 3: WS3 Developing Drivers for Altera SoC Linux
[14] Golden System Reference Design: AV CV GSRD 16.0 User manual – fasi per la generazione di un sistema Linux per dispositivi ALTERA.
[15] ALTERA SoC Embedded Design Suite User Guide – *ug_soc_eds.pdf* (ug-1137 del 2015.08.06) https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_soc_eds.pdf
[16] SoC HPS System Generation Using Qsys - https://www.youtube.com/watch?v=8BehnPg8IvM
[17] SoC HPS Quartus II Integration (HW/SW Hand-off) - https://www.youtube.com/watch?v=L8FMSy7Uxjc
[18] Preloader and U-boot Generation for Altera Cyclone V SoC - https://www.youtube.com/watch?v=vS7pvefsbRM
[19] Introduzione alle FPGA (*FPGA-Introduction.pdf*): http://lugman.org/index.php/Documentazione
[20] Elenco documentazione Cyclone V - http://rocketboards.org/foswiki/view/Documentation/CycloneVSoCLinks
[21] Talk ARM, FPGA & Linux - Linux Day 2015 – LUGMan
[22] BareMetal Application per DE0_NANO_SOC caricato al boot
[23] Ottenere e compilare il Kernel Linux, U-Boot, ecc per Cyclone V SoC per DE0-Nano-SoC
[24] Building External Modules: indica quali parametri e macro servono per la compilazione del kenel
[25] Embedded Linux Beginner Guide – Guida di Rocketboards abbastanza chiara e completa sulla creazione di u-Boot, Kernel e devicetreee anche in relazione all'FPGA partendo dalla configurazione dell'HPS. Comprende anche link a guide e video di approfondimento.

[26] 160929-Compilare_UBoot_Baremetal_e_Kernel.pdf – Documentazione LUGMan
[27] Creazione di una periferica FPGA – 151024-LinuxDay2015-ARM_FPGA_LINUX.pdf – Documentazione Linux Day 2015 sul sito (www.lugman.org)
[28] Embedded Linux Beginner Guide: configurazione degli switch di boot (MSEL)
[29] Programming FPGA from HPS: dal Preloader, UBoot o Linux
[30] Programmare la FPGA da Linux con kernel 4.x: https://forum.rocketboards.org/t/missing-dev-fpga0-device-on-de0-nano-soc/526/2
[31] Forum dove viene indicato quali registri fanno "saltare" l'esecuzione della CPU1 al risveglio: http://www.alteraforum.com/forum/showthread.php?t=43368
[32] ARM – Application Note 425 – Migrating a Software Application from ARMv5 to ARMv7-A/R: spiega come configurare MMU, cache, sistemi SMP, ecc.
[33] Device Tree Specification: specifiche di come scrivere i nodi del device tree (devicetree-specification-v0.1-20160524.pdf)
[34] Linux Device Driver 3° Edition
[35] The Linux Kernel Module Programming Guide
[36] Linux Kernel Memory Leak Detection – Linux Foundation – Catalin Madinas @ LinuxCon Europe 2011
[37] CPU Hotplug – Linux Kernel Documentation