Sample SPARC functions for the LEON 3FT

Table 1: Cross Reference of Applicable Products

Product Name:	Manufacturer Part Number	SMD #	Device Type	Internal PIC
LEON 3FT	UT699	5962-08228	ALL	WG07

1.0 Overview

The LEON 3FT has the capability to make use of inline assembly routines that store data to memory and load data from memory. Assembly routines are callable as a C functions. This application note also describes some common LEON inline assembly routines:

- Clock gating for the SpaceWire Ports, CAN, Ethernet, PCI peripheral cores.
  - The clock gating unit provides a means to save power by disabling the clock to unutilized core blocks.
- Data/instruction cache flush
- Enabling/disabling instruction/data caches
- Instruction burst fetches.
- EDAC check bit manipulation

2.0 Assembly routines

Assembly language is a low-level programming language for computers, microprocessors and microcontrollers. Assembly language programs are line based, meaning each statement typically identifies a single instruction or data component. Below is the basic template for an inline assembly routine and a store and read memory inline function for the UT699.

Basic inline assembly:

```
asm ( assembler template
:output operands  /*optional*/
:input operands   /*optional*/
);
```

Store Memory inline function:

```
// storemem is passed two arguments, the address(addr) and a value(val).
// The value gets stored into the address

inline void storemem(int addr, int val)
{
      asm volatile (" st %0, [%1] "       // store val to addr
                  :                       // output
                  : "r" (val), "r" (addr) // inputs
            );
}
```

Read Memory inline function:

```
// loadmem is passed one argument, the address(addr)
// The value that is in the address gets returned by the function

inline int loadmem(int addr)
{
      int tmp;                              // used for returned value
      asm volatile (" ld [%1], %0 "         // load tmp from addr
                  : "=r" (tmp)              // output
                  : "r" (addr)              // input
            );
      return tmp;
}
```

3.0 Leon Functions

The LEON function described below is for enabling and disabling the Ethernet core. Please refer to `leon_routines.c` in Appendix A for enabling/disabling the CAN, PCI, SPW[1/2/3/4] peripheral cores, instruction/data cache and instruction/data cache flushes.

The operation of the clock gating unit is controlled through three registers: the unlock, clock enable and core reset registers. The clock enable register defines if a clock is enabled or disabled. A '1' in a bit location enables the corresponding clock, while a '0' disables the clock. The core reset register resets each core to a default state. A reset will be generated as long as the corresponding bit is set to '1'. The bits in clock enable and core reset registers can only be written when the corresponding bit in the unlock register is 1. If the bit in the unlock register is 0, the corresponding bits in the clock enable and core reset registers cannot be written.

To clock gate the core, the following procedure should be applied:

1. Write a 1 to the corresponding bit in the unlock register (0x80000600)
2. Write a 0 to the corresponding bit in the clock enable register (0x80000604)
3. Write a 0 to the corresponding bit in the clock unlock register (0x80000600)

To enable the clock for a core, the following procedure should be applied:

1. Write a 1 to the corresponding bit in the unlock register (0x80000600)
2. Write a 1 to the corresponding bit in the clock enable register (0x80000604)
3. Write a 0 to the corresponding bit in the clock unlock register (0x80000600)

Repeat this procedure to reset a specific core using the core reset register in place of the clock enable register.

Table 2: Clocks Controlled by CLKGATE Unit

Bit in CLKGATE unit	FUNCTIONAL MODE
0 (0x01)	GRSPW Spacewire link 0
1 (0x02)	GRSPW Spacewire link 1
2 (0x04)	GRSPW Spacewire link 2
3 (0x08)	GRSPW Spacewire link 3
4 (0x10)	CAN core 1 & 2
5 (0x20)	GRETH 10/100 Mbit Ethernet MAC (AHB Clock)
6 (0x40)	GRPCI 32-bit PCI Bridge (AHB Clock)

Table 3: Clock Unit Control Registers

APB ADDRESS	FUNCTIONAL MODULE	RESET VALUE
0x80000600	Unlock Register	0000000b (0x00)
0x80000604	Clock Enable Register	1111111b (0x7F)
0x80000608	Core Reset Register	0000000b (0x00)

In the `enable_eth`, `disable_eth` and `clk_eth` functions below are defined as 0x20 in the clock gating unit. The variable `tmp` is a temporary local integer, and the `storemem` and `loadmem` functions are called to clock gate the Ethernet peripheral core of the UT699.

Enable Ethernet function:

```
void enable_eth(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_eth;
	storemem(unlreg, tmp);  /* unlock register */

	tmp = loadmem(clkres);
	tmp |= clk_eth;
	storemem(unlreg, tmp);  /* reset core */

	tmp = loadmem(clkenb);
	tmp |= clk_eth;
	storemem(clkenb, tmp);  /* enable clock */

	tmp = loadmem(clkres);
	tmp &= ~clk_eth;
	storemem(clkres, tmp);  /* reset core */

	tmp = loadmem(unlreg);
	tmp &= ~clk_eth;
	storemem(unlreg, tmp);  /* lock register */
}
```

Disable Ethernet function:

```
void disable_eth(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_eth;
	storemem(unlreg, tmp);  /* unlock register */

	tmp = loadmem(clkenb);
	tmp &= ~clk_eth;
	storemem(clkenb, tmp);  /* disable clock */

	tmp = loadmem(unlreg);
	tmp &= ~clk_eth;
	storemem(unlreg, tmp);  /* lock register */
}
```

4.0 Instruction Cache Functions

The cache functions use the load and store of the address space identifier 0x02. This is the ASI (Address Space Identifiers) of the Cache Control Register (CCR) with an offset of 0x0. The full list of Address Space Identifiers definitions are described in Table 1. The storemem_asi_02 function stores `val` to `addr` at ASI 0x02. The `loadmem_asi_02` function loads `tmp` to `addr` at ASI 0x02.

Table 4: ASI Usage

Address Space Identifiers (ASI)	Definitions
0x01	Forced Cache Miss
0x02	System Control Registers
0x08	User Instruction
0x09	Supervisor Instruction
0x0A	User Data
0x0B	Supervisor Instruction
0x0C	Instruction Cache Tags
0x0D	Instruction Cache Data
0x0E	Data Cache Tags
0x0F	Data Cache Data
0x10	Flush Instruction Cache
0x11	Flush Data Cache
0x1C	Leon Bypass

Store Memory inline ASI 0x02 function:

```
inline void storemem_asi_02(int addr, int val)
{
      asm volatile (" sta %0, [%1] 0x02 " // store val to addr at ASI 0x02
                  :                       // output
                  : "r" (val), "r" (addr) // inputs
            );
}
```

Read Memory ASI 0x02 inline function:

```
inline int loadmem_asi_02(int addr)
{
      int tmp;                            // used for returned value
      asm volatile (" lda [%1] 0x02, %0 " // load tmp from addr at ASI 0x02
                  : "=r" (tmp)            // output
                  : "r" (addr)            // input
            );
      return tmp;
}
```

To enable the instruction cache, 11b (0x3) must be written to the ICS bits of the CCR, see Table 5 for description. The `iCacheEnable` function on the next page reads ASI 0x02, OR's it with `0x00000003` and stores that value in `tmp`. Then, that value is stored into ASI 0x02, enabling the instruction cache.

To disable the instruction cache, x0b (00b or 10b) must be written to the ICS bits of the CCR, see Table 5 for description. The `iCacheDisable` function on the next page reads ASI 0x02, AND's it with `0xfffffffc` and stores that value in `tmp`. Then, that value is stored into ASI 0x02, disabling the instruction cache.

Table 5: Instruction Cache State

Bit Numbers	Bit Name	Description
1-0	ICS	Instruction Cache State Indicates the current instruction cache state: x0: Disabled 01: Frozen 11: Enabled x=don't care.

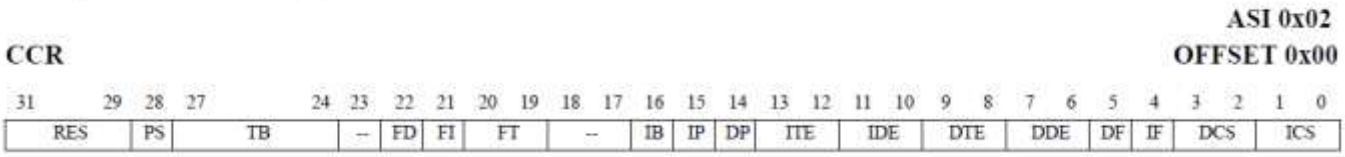

Figure 1. Cache Control Register

Instruction Cache Enable function using ASI 0x02 inline function:

```
void iCacheEnable(void)
{
	int tmp;
	tmp = loadmem_asi_02(0x00);
	tmp |= 0x00000003;
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
	storemem_asi_02(0x00, tmp);
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
}
```

Instruction Cache Disable function using ASI 0x02 inline function:

```
void iCacheDisable(void)
{
	int tmp;
	tmp = loadmem_asi_02(0x00);
	tmp &= 0xfffffffc;
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
	storemem_asi_02(0x00, tmp);
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
}
```

5.0 EDAC error insertions

To manipulate the EDAC check bits for SRAM/SDRAM do the following:

1. Disable data and instruction cache (see appendix A)
2. Enable read-modify-write cycles in memory configuration register 2, bit 6 (RMW)
3. Enable EDAC diagnostic write bypass in memory configuration register 3, bit 11 (WB)
4. Write to the Test Check bit field in memory configuration register 3, bits 7-0 (TCB[7:0])
5. Disable EDAC for SRAM/SDRAM in memory configuration register 3, bit 9 (SE)
6. Write to a memory location where the EDAC should occur
7. Enable EDAC and Disable EDAC diagnostic write bypass

```
// Disable dCache and iCache
dCacheDisable();
iCacheDisable();
//Enable Read-Modify-write
tmp = loadmem(0x80000004);
tmp = tmp | 0x40;
storemem(0x80000004, tmp);
//Enable EDAC Diagnostic write bypass, write to TCB field, disable EDAC for RAM
tmp = loadmem(0x80000008);
tmp = tmp | 0x82F;
storemem(0x80000008, tmp);
//Write to memory
storemem(0x404f1234, 0x12341234);
//enable EDAC for SRAM/SDRAM
tmp = loadmem(0x80000008);
tmp = tmp | 0x200;
storemem(0x80000008, tmp);
```

6.0 Conclusion

The remaining load/store memory ASI functions that are not described in this application note are located in `mem_routines.c` (Appendix A). The remaining LEON routines are located in `leon_routines.c` (Appendix A). The routines follow the same flow as the functions in this application note, but they preform different tasks in the UT699.

7.0 References

1. Aeroflex Colorado Springs Inc., UT699 LEON 3FT/SPARCTM V8 MicroProcessor Advanced User Manual, Aug. 2010
2. `mem_routnes.c, mem_routines.h, leon_routines.c, leon_routines.h` (Appendix A)

8.0 Appendix A

mem_routines.c

```
/* Inline assembly routines that store data to memory
 * and load data from memory. The routines are callable
 * as C functions.
 *
 * ASI definitions:
 * 0x01 forced cache miss
 * 0x02 system control registers
 * 0x08 user instruction
 * 0x09 supervisor instruction
 * 0x0A user data
 * 0x0B supervisor instruction
 * 0x0C instruction cache tags
 * 0x0D instruction cache data
 * 0x0E data cache tags
 * 0x0F data cache data
 * 0x10 flush instruction cache
 * 0x11 flush data cache
 * 0x1C Leon bypass */

#include "mem_routines.h"

inline void storemem(int addr, int val)
{
      asm volatile (" st %0, [%1] "       // store val to addr
                  :                       // output
                  : "r" (val), "r" (addr) // inputs
            );
}
inline int loadmem(int addr)
{
      int tmp;                            // used for returned value
      asm volatile (" ld [%1], %0 "       // load tmp from addr
                  : "=r" (tmp)            // output
                  : "r" (addr)            // input
            );
      return tmp;
}
inline void storemem_asi_01(int addr, int val)
{
      asm volatile (" sta %0, [%1] 0x01 "       // store val to addr at ASI 0x01
                  :                             // output
                  : "r" (val), "r" (addr)       // inputs
            );
}
inline int loadmem_asi_01(int addr)
{
      int tmp;                                  // used for returned value
      asm volatile (" lda [%1] 0x01, %0 "       // load tmp from addr at ASI 0x01
                  : "=r" (tmp)                  // output
                  : "r" (addr)                  // input
            );
      return tmp;
}
```

```
inline void storemem_asi_02(int addr, int val)
{
      asm volatile (" sta %0, [%1] 0x02 "       // store val to addr at ASI 0x02
                  :                             // output
                  : "r" (val), "r" (addr)       // inputs
            );
}
inline int loadmem_asi_02(int addr)
{
      int tmp;                                  // used for returned value
      asm volatile (" lda [%1] 0x02, %0 "       // load tmp from addr at ASI 0x02
                  : "=r" (tmp)                  // output
                  : "r" (addr)                  // input
            );
      return tmp;
}
inline void storemem_asi_0c(int addr, int val)
{
      asm volatile (" sta %0, [%1] 0x0c "       // store val to addr at ASI 0x0c
                  :                             // output
                  : "r" (val), "r" (addr)       // inputs
            );
}
inline int loadmem_asi_0c(int addr)
{
      int tmp;                                  // used for returned value
      asm volatile (" lda [%1] 0x0c, %0 "       // load tmp from addr at ASI 0x0c
                  : "=r" (tmp)                  // output
                  : "r" (addr)                  // input
            );
      return tmp;
}
inline void storemem_asi_0d(int addr, int val)
{
      asm volatile (" sta %0, [%1] 0x0d "       // store val to addr at ASI 0x0d
                  :                             // output
                  : "r" (val), "r" (addr)       // inputs
            );
}
inline int loadmem_asi_0d(int addr)
{
      int tmp;                                  // used for returned value
      asm volatile (" lda [%1] 0x0d, %0 "       // load tmp from addr at ASI 0x0d
                  : "=r" (tmp)                  // output
                  : "r" (addr)                  // input
            );
      return tmp;
}

inline void storemem_asi_0e(int addr, int val)
{
      asm volatile (" sta %0, [%1] 0x0e "       // store val to addr at ASI 0x0e
                  :                             // output
                  : "r" (val), "r" (addr)       // inputs
            );
}
```

```
inline int loadmem_asi_0e(int addr)
{
      int tmp;                                    // used for returned value
      asm volatile (" lda [%1] 0x0e, %0 "         // load tmp from addr at ASI 0x0e
                  : "=r" (tmp)                    // output
                  : "r" (addr)                    // input
            );
      return tmp;
}
inline void storemem_asi_0f(int addr, int val)
{
      asm volatile (" sta %0, [%1] 0x0f "         // store val to addr at ASI 0x0f
                  :                               // output
                  : "r" (val), "r" (addr)         // inputs
            );
}
inline int loadmem_asi_0f(int addr)
{
      int tmp;                                    // used for returned value
      asm volatile (" lda [%1] 0x0f, %0 "         // load tmp from addr at ASI 0x0f
                  : "=r" (tmp)                    // output
                  : "r" (addr)                    // input
            );
      return tmp;
}

inline void storemem_asi_10(int addr, int val)
{
      asm volatile (" sta %0, [%1] 0x10 "         // store val to addr at ASI 0x10
                  :                               // output
                  : "r" (val), "r" (addr)         // inputs
            );
}
inline void storemem_asi_11(int addr, int val)
{
      asm volatile (" sta %0, [%1] 0x11 "         // store val to addr at ASI 0x11
                  :                               // output
                  : "r" (val), "r" (addr)         // inputs
            );
}
inline void storemem_asi_1c(int addr, int val)
{
      asm volatile (" sta %0, [%1] 0x1c "         // store val to addr at ASI 0x11
                  :                               // output
                  : "r" (val), "r" (addr)         // inputs
            );
}
```

mem_routines.h

```
#ifndef MEM_ROUTINES_H_
#define MEM_ROUTINES_H_

#endif /*MEM_ROUTINES_H_*/

/* Inline assembly routines that store data to memory
 * and load data from memory. The routines are callable
 * as C functions.
 *
 * ASI definitions:
 * 0x01 forced cache miss
 * 0x02 system control registers
 * 0x08 user instruction
 * 0x09 supervisor instruction
 * 0x0A user data
 * 0x0B supervisor instruction
 * 0x0C instruction cache tags
 * 0x0D instruction cache data
 * 0x0E data cache tags
 * 0x0F data cache data
 * 0x10 flush intruction cache
 * 0x11 flush data cache
 * 0x1C Leon bypass */

inline void storemem(int addr, int val);
inline int loadmem(int addr);
inline void storemem_asi_01(int addr, int val);
inline int loadmem_asi_01(int addr);
inline void storemem_asi_02(int addr, int val);
inline int loadmem_asi_02(int addr);
inline void storemem_asi_08(int addr, int val);
inline int loadmem_asi_08(int addr);
inline void storemem_asi_09(int addr, int val);
inline int loadmem_asi_09(int addr);
inline void storemem_asi_0a(int addr, int val);
inline int loadmem_asi_0a(int addr);
inline void storemem_asi_0b(int addr, int val);
inline int loadmem_asi_0b(int addr);
inline void storemem_asi_0c(int addr, int val);
inline int loadmem_asi_0c(int addr);
inline void storemem_asi_0d(int addr, int val);
inline int loadmem_asi_0d(int addr);
inline void storemem_asi_0e(int addr, int val);
inline int loadmem_asi_0e(int addr);
inline void storemem_asi_0f(int addr, int val);
inline int loadmem_asi_0f(int addr);
inline void storemem_asi_10(int addr, int val);
inline void storemem_asi_11(int addr, int val);
inline void storemem_asi_1c(int addr, int val);
```

leon_routines.c

```
/* Common C routines used in Leon code execution */

#include "leon_routines.h"
#include "mem_routines.h"

void enable_eth(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_eth;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkres);
	tmp |= clk_eth;
	storemem(unlreg, tmp);			/* reset core */

	tmp = loadmem(clkenb);
	tmp |= clk_eth;
	storemem(clkenb, tmp);			/* enable clock */

	tmp = loadmem(clkres);
	tmp &= ~clk_eth;
	storemem(clkres, tmp);			/* reset core */

	tmp = loadmem(unlreg);
	tmp &= ~clk_eth;
	storemem(unlreg, tmp);			/* lock register */
}

void enable_can(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_can;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkres);
	tmp |= clk_can;
	storemem(unlreg, tmp);			/* reset core */

	tmp = loadmem(clkenb);
	tmp |= clk_can;
	storemem(clkenb, tmp);			/* enable clock */

	tmp = loadmem(clkres);
	tmp &= ~clk_can;
	storemem(clkres, tmp);			/* reset core */

	tmp = loadmem(unlreg);
	tmp &= ~clk_can;
	storemem(unlreg, tmp);			/* lock register */
}
```

```
void enable_pci(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_pci;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkres);
	tmp |= clk_pci;
	storemem(unlreg, tmp);			/* reset core */

	tmp = loadmem(clkenb);
	tmp |= clk_pci;
	storemem(clkenb, tmp);			/* enable clock */

	tmp = loadmem(clkres);
	tmp &= ~clk_pci;
	storemem(clkres, tmp);			/* reset core */

	tmp = loadmem(unlreg);
	tmp &= ~clk_pci;
	storemem(unlreg, tmp);			/* lock register */
}

void enable_spw0(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_spw0;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkres);
	tmp |= clk_spw0;
	storemem(unlreg, tmp);			/* reset core */

	tmp = loadmem(clkenb);
	tmp |= clk_spw0;
	storemem(clkenb, tmp);			/* enable clock */

	tmp = loadmem(clkres);
	tmp &= ~clk_spw0;
	storemem(clkres, tmp);			/* reset core */

	tmp = loadmem(unlreg);
	tmp &= ~clk_spw0;
	storemem(unlreg, tmp);			/* lock register */
}
```

```
void enable_spw1(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_spw1;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkres);
	tmp |= clk_spw1;
	storemem(unlreg, tmp);			/* reset core */

	tmp = loadmem(clkenb);
	tmp |= clk_spw1;
	storemem(clkenb, tmp);			/* enable clock */

	tmp = loadmem(clkres);
	tmp &= ~clk_spw1;
	storemem(clkres, tmp);			/* reset core */

	tmp = loadmem(unlreg);
	tmp &= ~clk_spw1;
	storemem(unlreg, tmp);			/* lock register */
}

void enable_spw2(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_spw2;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkres);
	tmp |= clk_spw2;
	storemem(unlreg, tmp);			/* reset core */

	tmp = loadmem(clkenb);
	tmp |= clk_spw2;
	storemem(clkenb, tmp);			/* enable clock */

	tmp = loadmem(clkres);
	tmp &= ~clk_spw2;
	storemem(clkres, tmp);			/* reset core */

	tmp = loadmem(unlreg);
	tmp &= ~clk_spw2;
	storemem(unlreg, tmp);			/* lock register */
}
```

```
void enable_spw3(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_spw3;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkres);
	tmp |= clk_spw3;
	storemem(unlreg, tmp);			/* reset core */

	tmp = loadmem(clkenb);
	tmp |= clk_spw3;
	storemem(clkenb, tmp);			/* enable clock */

	tmp = loadmem(clkres);
	tmp &= ~clk_spw3;
	storemem(clkres, tmp);			/* reset core */

	tmp = loadmem(unlreg);
	tmp &= ~clk_spw3;
	storemem(unlreg, tmp);			/* lock register */
}

void disable_eth(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_eth;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkenb);
	tmp &= ~clk_eth;
	storemem(clkenb, tmp);			/* disable clock */

	tmp = loadmem(unlreg);
	tmp &= ~clk_eth;
	storemem(unlreg, tmp);			/* lock register */
}

void disable_can(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_can;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkenb);
	tmp &= ~clk_can;
	storemem(clkenb, tmp);			/* disable clock */

	tmp = loadmem(unlreg);
	tmp &= ~clk_can;
	storemem(unlreg, tmp);			/* lock register */
}
```

```
void disable_pci(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_pci;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkenb);
	tmp &= ~clk_pci;
	storemem(clkenb, tmp);			/* disable clock */

	tmp = loadmem(unlreg);
	tmp &= ~clk_pci;
	storemem(unlreg, tmp);			/* lock register */
}

void disable_spw0(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_spw0;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkenb);
	tmp &= ~clk_spw0;
	storemem(clkenb, tmp);			/* disable clock */

	tmp = loadmem(unlreg);
	tmp &= ~clk_spw0;
	storemem(unlreg, tmp);			/* lock register */
}

void disable_spw1(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_spw1;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkenb);
	tmp &= ~clk_spw1;
	storemem(clkenb, tmp);			/* disable clock */

	tmp = loadmem(unlreg);
	tmp &= ~clk_spw1;
	storemem(unlreg, tmp);			/* lock register */
}
```

```
void disable_spw2(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_spw2;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkenb);
	tmp &= ~clk_spw2;
	storemem(clkenb, tmp);			/* disable clock */

	tmp = loadmem(unlreg);
	tmp &= ~clk_spw2;
	storemem(unlreg, tmp);			/* lock register */
}

void disable_spw3(void)
{
	int tmp;
	tmp = loadmem(unlreg);
	tmp |= clk_spw3;
	storemem(unlreg, tmp);			/* unlock register */

	tmp = loadmem(clkenb);
	tmp &= ~clk_spw3;
	storemem(clkenb, tmp);			/* disable clock */

	tmp = loadmem(unlreg);
	tmp &= ~clk_spw3;
	storemem(unlreg, tmp);			/* lock register */
}

void iCacheEnable(void)
{
	int tmp;
	tmp = loadmem_asi_02(0x00);
	tmp |= 0x00000003;
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
	storemem_asi_02(0x00, tmp);
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
}

void dCacheEnable(void)
{
	int tmp;
	tmp = loadmem_asi_02(0x00);
	tmp |= 0x0000000c;
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
	storemem_asi_02(0x00, tmp);
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
}
```

```
void iCacheDisable(void)
{
	int tmp;
	tmp = loadmem_asi_02(0x00);
	tmp &= 0xfffffffc;
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
	storemem_asi_02(0x00, tmp);
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
}

void dCacheDisable(void)
{
	int tmp;
	tmp = loadmem_asi_02(0x00);
	tmp &= 0xfffffff3;
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
	storemem_asi_02(0x00, tmp);
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
}

void iCacheFlush(void)
{
	int tmp, cnt;
	tmp = loadmem_asi_02(0x00) | FI;
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
	storemem_asi_02(0x00, tmp);
	for(cnt=0; cnt<8192; cnt++)          /* Wait the correct amount of time until */
		doNothing();                     /* the bit IP in the CCR is cleared */
}

void dCacheFlush(void)
{
	int tmp, cnt;
	tmp = loadmem_asi_02(0x00) | FD;
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
	storemem_asi_02(0x00, tmp);
	for(cnt=0; cnt<8192; cnt++)          /* Wait the correct amount of time until */
		doNothing();                     /* the bit DP in the CCR is cleared */
}
```

```
void burstFetchEnable(void)
{
	int tmp;
	tmp = loadmem_asi_02(0x00) | IB;
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
	storemem_asi_02(0x00, tmp);
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
}

void burstFetchDisable(void)
{
	int tmp;
	tmp = loadmem_asi_02(0x00) & ~IB;
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
	storemem_asi_02(0x00, tmp);
	asm (" nop ");
	asm (" nop ");
	asm (" nop ");
}

void doNothing(void)
{
	asm (" nop ");
}
```

leon_routines.h

```
/* Header file for leon_routines.c */

#define     clk_spw0    0x01                    /* clock gating bit for SpW0 */
#define     clk_spw1    0x02                    /* clock gating bit for SpW1 */
#define     clk_spw2    0x04                    /* clock gating bit for SpW2 */
#define     clk_spw3    0x08                    /* clock gating bit for SpW3 */
#define     clk_can           0x10              /* clock gating bit for CAN0 and CAN1
*/
#define     clk_eth           0x20              /* clock gating bit for Ethernet */
#define     clk_pci           0x40              /* clock gating bit for pci */
#define     FD                0x00400000        /* flush data cache bit */
#define     FI                0x00200000        /* flush instruction cache bit */
#define     IB                0x00010000        /* instruction burst fetch bit */

/* Registers */
#define     unlreg            0x80000600        /* clock gating unlock register */
#define     clkenb            0x80000604        /* clock gating enable register */
#define     clkres            0x80000608        /* clock gating reset register */

/* Function prototypes */
void enable_eth(void);
void enable_can(void);
void enable_pci(void);
void enable_spw0(void);
void enable_spw1(void);
void enable_spw2(void);
void enable_spw3(void);
void disable_eth(void);
void disable_can(void);
void disable_pci(void);
void disable_spw0(void);
void disable_spw1(void);
void disable_spw2(void);
void disable_spw3(void);
void iCacheEnable(void);
void dCacheEnable(void);
void iCacheDisable(void);
void dCacheDisable(void);
void iCacheFlush(void);
void dCacheFlush(void);
void burstFetchEnable(void);
void burstFetchDisable(void);
void doNothing(void);
```