The only other small core left is the LM32 core, and it was evaluated last. This core is a very well documented open source core actively maintained by Lattice. It comes with a plug-in for the Eclipse IDE. Further more a whole suite of peripherals are offered with it. Most of the peripherals are from the opencores.org web site, but they have been tested on the LM32 architecture. What is also worth noting is that Lattice offers a scaled down 8-bit version of the LM32. uClinux and uCOS ports are also offered with the core. Each RTOS was very well documented.

Section 4.1.2.1 LM32 soft-core architecture

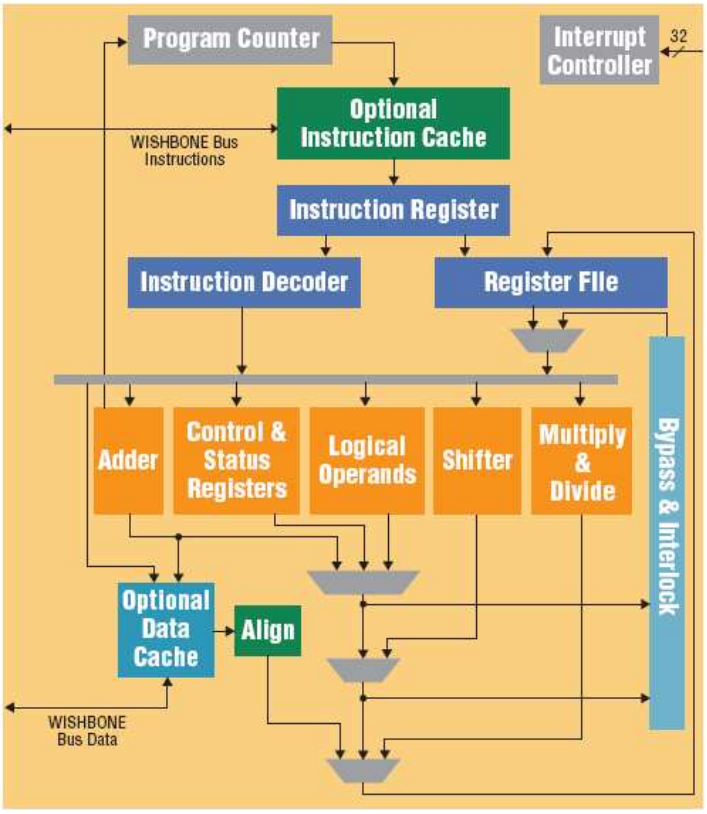

Figure 25. LM32 block diagram

Figure 25 shows the block diagram of the LM32 soft-core. The processor uses a 32-bit 6-stage pipeline with bypass and interlock logic. The six pipeline stages are:

- Address – The address of the instruction to execute is calculated and sent to the instruction cache.
- Fetch – The instruction is read from memory.

- ∞ Decode – The instruction is decoded, and operands are either fetched from the register file or bypassed from the pipeline.
- ∞ Execute – The operation specified by the instruction is performed. For simple instructions such as addition or a logical operation, execution finishes in this stage, and the result is made available for bypassing.
- ∞ Memory – For more complicated instructions such as loads, stores, multiplies, or shifts, a second execution stage is required.
- ∞ Write back – Results produced by the instructions are written back to the register file.

The processor has 32 32-bit registers. The first eight functional parameters are passed in the registers and the remaining arguments are placed on the stack.

The LM32 has an optional data and instruction cache, but for this design the cache is not enabled since the performance increase is not required. The LM32 has a flat byte addressable 32-bit address space that uses the big-endian standard (MSB is at lowest address).

All memory accesses must be aligned to the size of the access. No check is performed for unaligned access and it will result in undefined behaviour. 8-bit, 16-bit and 32-bit aligned access is allowed. The stack grows towards higher memory.

The LM32 architecture supports both software and hardware breakpoints. Software breakpoints should be used for setting breakpoints in code, which resides in volatile memory such as DDR or SRAM, while hardware breakpoints should be used for setting breakpoints in code, which resides in non-volatile memory, such as FLASH or ROM.

Section 4.1.3 Peripherals

The LM32 core contains a host of peripherals mostly from the www.opencores.org web site. In most instances these peripherals where designed with extensive feature set which makes them big. It was decided to create new small peripherals with limited features written in VHDL.

The LM32 soft core is written in Verilog. The author is fluent in VHDL therefore the high level instantiation of the core is done in VHDL, and all peripherals are written in VHDL with test benches written for Modelsim.

The following peripherals were developed for this project: external memory bus, general purpose I/O, 32-bit timer with PWM, SPI bus and UART.

The LM32 core is build on the Wishbone bus architecture (see www.opencores.org). Wishbone is an open source bus standard, and functions as a common interface to all IP cores and memories. A Wishbone master always talks to a Wishbone slave. The master is responsible for generating the bus cycle while the slave receives the bus cycle.

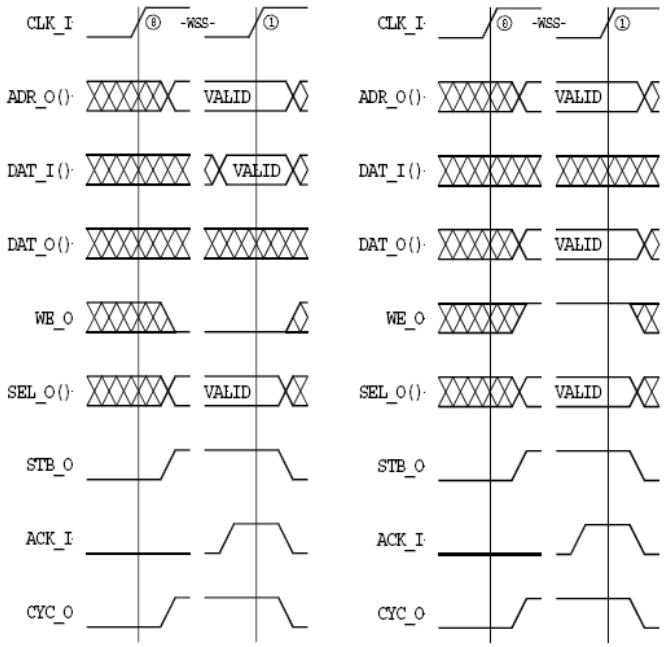

Figure 26. Wishbone read and write bus cycle

Figure 26 shows the Wishbone bus cycle. A data read takes place on the rising edge of the clock while STB_O and ACK_I is high and WE_O is low. The data write takes place on the rising edge of the clock while STB_O, ACK_I and WE_O is high. ACK_I is an asynchronous acknowledge signal.

SEL_O[3:0]	Byte selection
1000	MSB
0100	
0010	
0001	LSB

Table 3. Wishbone SEL_O byte map

The LM32 is a big Endean core and Table 3 shows how SEL_O maps to the corresponding byte within the 32-bits. Bus address lines A1 and A0 are not used by the core, and SEL_O is used for BYTE and WORD selection. Table 4 shows the byte location of the data bus.

WBS_DAT_I/WBS_DAT_O	Byte definition
[31:24]	MSB
[23:16]	
[15:8]	
[7:0]	LSB

Table 4. Wishbone data in and out bus definition

Section 4.1.4 HDL design

The LM32 has a separate bus for data and code and each of these is a Wishbone master. All peripherals and memory connected to these busses are Wishbone slave devices. For flexibility the hardware design has a data bus, instruction bus and a peripheral bus. It was decided to separate the peripheral bus from the data bus to eliminate bus stalling. Experiments showed long bus delays on the data bus when combined with slow peripherals.

To simplify the interfacing of the peripherals to the core and the debug architecture, a bus arbiter was designed to combine the instruction, peripheral and data bus into one shared bus. A shared bus arbiter is used to combine the instruction and data bus into one bus. Both the instruction and data bus are bus masters, and the arbiter will connect the single shared bus to a specific bus master if it is not used by the other bus master. All peripherals and memories connect through this single shared bus to the processor core.
In future, this simplification can be removed if better performance is required.

The design makes provision for a debug and operational processor mode. In debug mode the debug ROM kernel is mapped to the reset vector to ensure execution after POR. It is only through the execution of this kernel that the LM32 can talk to the GDB debugger on the PC.

The addressable area is broken up in 8KB pages and address lines A0 – A12 are ignored in address decoding. Even though this is a Harvard architecture none of the areas between the instruction and data bus overlaps. This was a deliberate decision to simplify the linker file as GCC does not inherently support Harvard architecture. All flash and ROM areas are

accessible through the bus arbiter. This is done to support in system programming and debugging.

Address range 0 to 0x7FFFFFFF are reserved for cached area leaving address area 0x80000000 to 0xFFFFFFFF for peripherals.

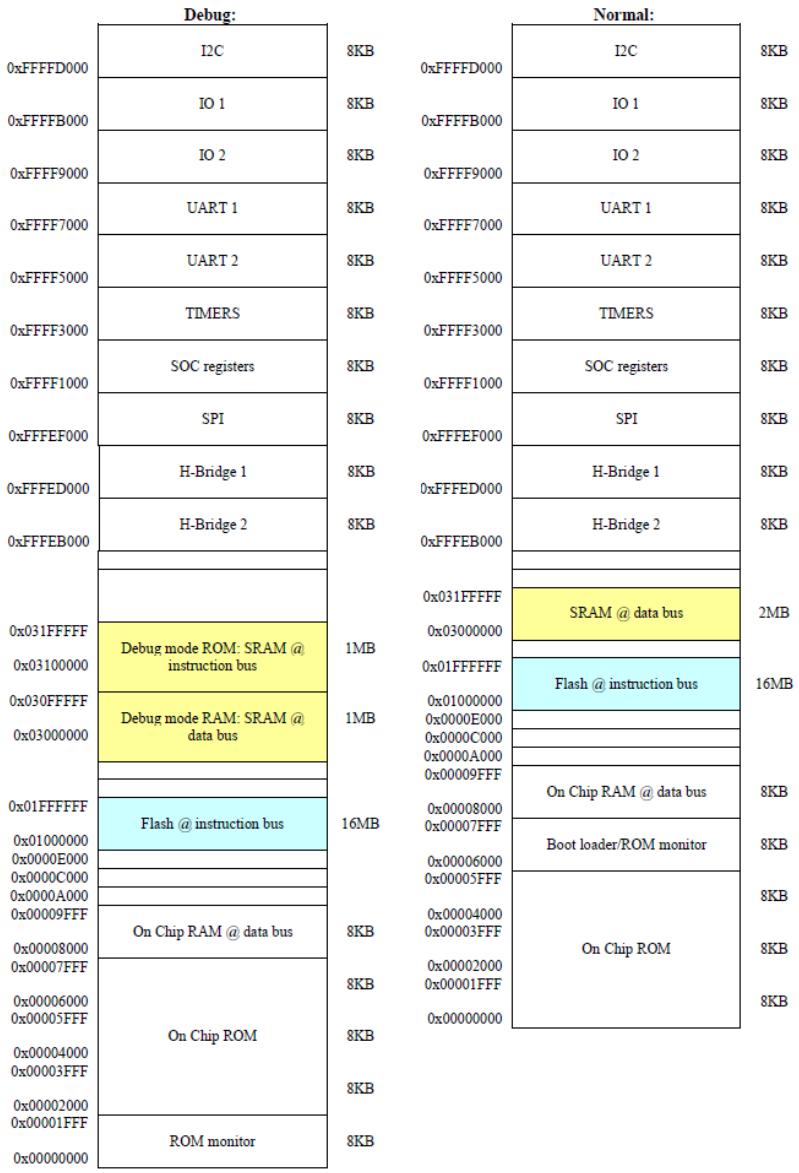

Figure 27. CPU address map for different modes

Figure 27 shows the processor address map for normal and debug mode. The ROM monitor in debug mode contains the GDB stub implementation (see Section 4.1.7.1).

Section 4.1.4.1 Peripherals

At the moment all the peripherals run at the same clock frequency as the processor.

I/O Port

To accept input and control actuators the core requires general purpose input and output ports. The port design is bidirectional and bit addressable and multiple instances can be instantiated. Table 5 shows the component definition for the I/O port. The SOC contains 2 I/O ports.

```
component IO
        Port (ADR_I         : in std_logic_vector(3 downto 0);
              DAT_I         : in std_logic_vector(7 downto 0);
              DAT_O         : out std_logic_vector(7 downto 0);
              CLK_I         : in std_logic;
              RST_I         : in std_logic;
              ACK_O         : out std_logic;
              STB_I         : in std_logic;
              WE_I          : in std_logic;
              CYC_I         : in std_logic;
             --Non wishbone signals
             HW_PORT_PINs          : inout std_logic_vector(7 downto 0));
end component;
```

Table 5. I/O Port VHDL component definition

Timer

Timers are also important especially when using a RTOS in which case the operating system tick is generated by a time overflowing periodically. An array of timers was created in VHDL and during instantiation the required timer count (N_TIMERS) is specified. The width of the timer (TIMER_WIDTH) is also specified and the default value is 32-bits. Optionally the timer can be configured as a 32-bit Pulse Width Modulator (PWM). Table 6 shows the component definition for the Timer. The SOC contains 4 timers of which one is dedicated to the RTOS.

```
component Timers
   generic( N_TIMERS            : integer range 1 to 8 := 1;
            TIMER_WIDTH         : integer range 8 to 32 := 32   );
   Port (   CLK_I       : in  STD_LOGIC;
            RST_I       : in  STD_LOGIC;
            ADR_I       : in  STD_LOGIC_VECTOR (31 downto 2);
            DAT_I       : in  STD_LOGIC_VECTOR ((TIMER_WIDTH-1) downto 0);
            DAT_O       : out STD_LOGIC_VECTOR ((TIMER_WIDTH-1) downto 0);
            ACK_O       : out STD_LOGIC;
            STB_I       : in  STD_LOGIC;
            SEL_I       : in  STD_LOGIC_VECTOR (3 downto 0);
            CYC_I       : in  STD_LOGIC;
            WE_I        : in  STD_LOGIC;

            timers_out          : out std_logic_vector((N_TIMERS-1) downto 0);
            timers_pwm_out      : out std_logic_vector((N_TIMERS-1) downto 0));
end component;
```

Table 6. Timer VHDL Component definition

UART

To communicate to peripherals like a wireless modem and GPS receiver the core requires a serial port. A lightweight UART was designed specifically for this task and to use as a debug port. The UART design can be instantiated multiple times. The current implementation contains 4 UART's. Table 7 shows the component definition for the UART.

The baud generator of the UART does not need a specific clock frequency like 1.8432MHz but can work of the CPU frequency of 32MHz. Normally the divisor for 115200 baud at 32MHz would be 277.78. To achieve a stable baud rate we sometimes divide by 277 and sometimes by 278, through the cycle, this average out to 277.78. Table 8 shows the VHDL implementation for the baud generator.

```
component serial
generic ( BAUD_RATE     : integer;
          CORE_FREQ_KHZ : integer);
Port ( WBS_ADR_I   : in std_logic_vector(3 downto 2);
       WBS_DAT_I   : in std_logic_vector(31 downto 0);
       WBS_DAT_O   : out std_logic_vector(31 downto 0);
       WBS_CLK_I   : in std_logic;
       WBS_RST_I   : in std_logic;
       WBS_ACK_O   : out std_logic;
       WBS_STB_I   : in std_logic;
       WBS_WE_I    : in std_logic;
       WBS_SEL_I   : in std_logic_vector(3 downto 0);
       WBS_CYC_I   : in std_logic;

       serial_intr  : out std_logic;

       -- Serial port Interface
      txpin        : out std_logic;
      rxpin        : in  std_logic);
end component;
```

Table 7 UART VHDL Component definition

```
baud_inc = ((Baud<<(BaudGeneratorAccWidth-4))+(ClkFrequency>>5))/(ClkFrequency>>4);

process (WBS_CLK_I, WBS_RST_I)
begin
   if WBS_RST_I = '1' then
      baud_inc <= conv_std_logic_vector(gen_inc_val,(ACC_WIDTH+1));
      baud_accum <= (others => '0');
   elsif rising_edge(WBS_CLK_I) then
       if baud_inc = conv_std_logic_vector(0,ACC_WIDTH) then
          baud_inc <= conv_std_logic_vector(gen_inc_val,ACC_WIDTH+1);
       elsif divreload = '1' then
         baud_inc <= data_in;--WBS_DAT_I;
         baud_accum <= (others => '0');
       end if;
       baud_accum <= ('0' & baud_accum((ACC_WIDTH-1) downto 0)) + baud_inc;
   end if;
end process ;
baudx16_clk <= baud_accum(ACC_WIDTH);
```

Table 8. Baud generator code

Memory

The BRAM inside the FPGA is grouped together in 2KB x 32-bit blocks and form the building elements for ROM and RAM storage. The only difference between ROM and RAM is write access. Table 9 shows the definition of each component.

```
component BRAM_2K_32bits_WB port(
        CLK_I   : in std_logic;
        RST_I   : in std_logic;
        ACK_O   : out std_logic;
        STB_I   : in std_logic;
        CYC_I   : in std_logic;
        WE_I    : in std_logic;
        DAT_I   : in std_logic_vector(31 downto 0);
        DAT_O   : out std_logic_vector(31 downto 0);
        SEL_I   : in std_logic_vector(3 downto 0);
        ADR_I   : in std_logic_vector(12 downto 2));
end component;
```

```
component BROM_2K_32bits_WB port(
        CLK_I   : in std_logic;
        RST_I   : in std_logic;
        ACK_O   : out std_logic;
        CYC_I   : in std_logic;
        STB_I   : in std_logic;
        DAT_O   : out std_logic_vector(31 downto 0);
        ADR_I   : in std_logic_vector(12 downto 2));
end component;
```

Table 9. Onboard ROM/RAM VHDL Component definition

When the FPGA goes through it's configuration cycle the BRAM blocks are initialised with data from the bitmap file. It is possible to add the application binary data in the FPGA bit file (see data2mem in Section 4.1.7.1) so that the BRAM (simulated ROM) contains the application after FPGA configuration. Once the FPGA configuration is completed the soft processor core can start executing the application binaries.

The design also caters for external flash and ram through a 16-bit external instruction and data bus. This bus is connected to the 32-bit wishbone bus through a separate RAM/Flash controller. This controller can include wait states (COUNT_CYCLE_EXTEND) as required. It supports byte, word and double word read and write operations.

For each external access the controller does two consecutive external bus cycles, thereby increasing the external bus access time by a factor of two. The power consumption requirement has a higher priority than performance. A 32-bit external bus would require 2 additional components that would increase the total power consumption. Table 10 shows the VHDL component definition for the external memory bus interface.

```
COMPONENT SRAM
generic(    COUNT_CYCLE_EXTEND          : integer := 1;
            WORD_ADR_WIDTH              : integer := 20 );
port (      CLK_I   : std_logic;
            RST_I : std_logic;

            -- Wishbone slave interface
            ACK_O    : inout std_logic;
            STB_I    : in std_logic;
            CYC_I    : in std_logic;
            WE_I     : in std_logic;
            DAT_I    : in std_logic_vector(31 downto 0);
            DAT_O    : out std_logic_vector(31 downto 0);
            SEL_I    : in std_logic_vector(3 downto 0);
            ADR_I    : in std_logic_vector(WORD_ADR_WIDTH downto 2);

            --RAM chip interface
            ADR                 : out std_logic_vector(WORD_ADR_WIDTH downto 1);
            DQ                  : inout std_logic_vector(15 downto 0);
            nWE                 : out std_logic;
            nOE                 : out std_logic;
            nUB                 : out std_logic;
            nLB                 : out std_logic;
            nCE                 : out std_logic         );
END COMPONENT;
```

Table 10. External memory bus VHDL component definition.

SPI Bus

The SPI bus is a low pin-count interconnection bus and was chosen as the main interface to peripheral devices. The HDL design caters for all variations like clock phase, clock polarity, delay time and shift direction. Each SPI port can only function as a master device. The SOC contains one SPI port but makes provision for a second SPI port. Table 11 shows the SPI component definition. SHIFT_DIRECTION specifies whether the most or least significant bit is send first. CLOCK_PHASE specifies if data is latched on the leading or trailing edge of the clock. CLOCK_POLARITY specifies the idle logic stat of the clock line.

```
COMPONENT spi
        generic(    SHIFT_DIRECTION : integer := 0;
                    CLOCK_PHASE     : integer := 0;
                    CLOCK_POLARITY  : integer := 0;
                    SLAVE_NUMBER    : integer := 1;
                    DATA_LENGTH     : integer := 3;
                    DELAY_TIME      : integer := 2;
                    INTERVAL_LENGTH : integer := 2);
        PORT(       CLK_I           : IN  std_logic;
                    RST_I           : IN  std_logic;
                    SPI_ADR_I       : IN  std_logic_vector(31 downto 0);
                    SPI_DAT_I       : IN  std_logic_vector(31 downto 0);
                    SPI_DAT_O       : OUT  std_logic_vector(31 downto 0);
                    SPI_ACK_O       : OUT  std_logic;
                    SPI_STB_I       : IN  std_logic;
                    SPI_SEL_I       : IN  std_logic_vector(3 downto 0);
                    SPI_CYC_I       : IN  std_logic;
                    SPI_WE_I        : IN  std_logic;
                    SPI_CTI_I       : in std_logic_vector(2 downto 0);
                    SPI_BTE_I       : in std_logic_vector(1 downto 0);
                    SPI_LOCK_I      : in std_logic;
                    SPI_ERR_O       : out std_logic;
                    SPI_RTY_O       : out std_logic;

                    MISO_MASTER     : IN  std_logic;
                    MOSI_MASTER     : OUT  std_logic;
                    SS_N_MASTER     : OUT  std_logic_vector((SLAVE_NUMBER-1) downto 0);
                    SCLK_MASTER     : OUT  std_logic;
                    SPI_INT_O       : OUT  std_logic;
END COMPONENT;
```

Table 11. SPI VHDL Component definition

H-Bridge controller

The H-bridge controller is designed to drive a DC motor with the following modes: forward speed control, reverse speed control, braking and stop. The speed and level of breaking is configurable through the duty cycle register. The period of the PWM signal is also configurable. The SOC contains 2 H-Bridge controllers.

```
entity H_bridge is
   Port ( CLK_I    : in  STD_LOGIC;
          RST_I    : in  STD_LOGIC;
          ADR_I    : in  STD_LOGIC_VECTOR (31 downto 2);
          DAT_I    : in  STD_LOGIC_VECTOR (31 downto 0);
          DAT_O    : out STD_LOGIC_VECTOR (31 downto 0);
          ACK_O    : out STD_LOGIC;
          STB_I    : in  STD_LOGIC;
          SEL_I    : in  STD_LOGIC_VECTOR (3 downto 0);
          WE_I     : in  STD_LOGIC;

          -- Due to the hardware desighn, highside driver is inverted.
          n_left_top_plus          : out std_logic;
          left_bottom_plus      : out std_logic;
          n_right_top              : out std_logic;
          right_bottom             : out std_logic);
end H_bridge;
```

Table 12. H-Bridge VHDL Component definition

Quadrature decoder

A quadrature device is used to sense position and rotation by converting the displacement into digital pulses. The phase difference between output signal A and output signal B determines the direction of rotation. For example, if pulse output A leads pulse output B the shaft is rotating in the clockwise direction.

```
component quad_decoder is
   generic(    QUAD_COUNT_WIDTH    : integer   := 32 );
   port(   --Wishbone bus signals
           wb_clk_i     : in    std_logic;
           wb_rst_i     : in    std_logic;
           wb_stb_i     : in    std_logic;
           wb_cyc_i     : in    std_logic;
           wb_ack_o     : out   std_logic;
           wb_adr_i     : in    std_logic_vector(1 downto 0); --assumes 32 bit alignment
           wb_dat_o     : out   std_logic_vector(31 downto 0);
           wb_dat_i     : in    std_logic_vector(31 downto 0);
           wb_we_i      : in    std_logic;
           --Quadrature inputs
           quad_cha_i  : in    std_logic;  --Qudrature channel A
           quad_chb_i  : in    std_logic;  --Quadrature channel B
           quad_idx_i   : in    std_logic;  --Quadrature index
           quad_lat_i   : in    std_logic;  --Quadrature latch cnt input
           quad_irq_o  : out   std_logic   --Quadrature IRQ out
   );
end component;
```

Table 13. Quadrature decoder VHDL component definition

A quadrature decoder takes the output signals (A, B, and Index) from the quadrature encoder as inputs and converts these signals into a numerical value that can be used to determine position, distance, velocity, and other functions. Table 13 shows the definition for the VHDL component. The SOC contains two quadrature decoders.

Section 4.1.4.2 Clocks

Inside the FPGA the delayed-locked loop of the DCM is used to remove any clock skew. The clock is then split into two paths each going into a programmable divider. The one path goes to the LM32 soft core while the other one is used for the peripheral clock. There is also an enable/disable bit for the peripheral clock. Each output goes through a BUFG buffer specifically design to buffer and gate clock resources inside the FPGA.

Section 4.1.4.3 Power management

A FPGA based design is usually not a low power platform. Power management aims to limit the power consumption by implementing intelligent power control algorithms. When a peripheral is not used it is placed in a sleep mode and its clock is disabled. This reduces the power consumption since there are no signal transitions in the peripheral due to the absence of the clock.

The clock to a peripheral is disabled by using a BUGCE or a BUFGMUX primitive. These are resources inside the FPGA that were designed by the manufacturer to select and gate clocks. If not explicitly specified, the FPGA will use a multiplexer for the enable to the CLB that will not disable the clock to the flip flops inside the CLB. Thus the clock will continue to charge and discharge the input capacitance of the flip flop.

The power management also controls the LM32 core clock frequency, and when high processing speed is not required the clock frequency is scaled down. The LM32 core clock frequency is also scaled at power-up. After POR the LM32 runs at a reduced clock frequency placing a smaller demand on the power supply. This was only done for a brief moment after which the clock frequency is scaled back to the full value.

The FPGA core temperature is monitored in an attempt to reduce the static power consumption. This is done through a diode inside the FPGA silicon. The temperature is used

to scale the LM32 core clock frequent. When the temperature is too high the clock frequency is reduced, but when it is under a threshold value the clock frequency is not scaled.

Section 4.1.4.4 Dynamic Partial Self Reconfiguration

The design is partitioned into blocks. There is one fixed block containing the LM32 soft-core and the ICAP interface. This block will never change dynamically but will always initiate the dynamic reconfiguration of the other blocks (see Figure 29). The layout of the blocks is done in PlanAhead which export a user constraint file (UCF). The UCF file is imported into the ISE project containing the soft-core and all the VHDL code for each block. The ISE project must also contain the clock network which cannot change during a partial reconfiguration.

Section 4.1.4.5 Floorplanning

Floorplanning the design proved to be a very difficult step in the design process. There are conflict priorities and the optimal floor plan is a compromise between partial reconfiguration layout, IOB's voltage compatibility and peripheral location on the PCB.

To improve signal integrity it is important to have the peripheral pins as close to the actual peripherals as possible. This reduces the track length and track impedance. The problem arises when there are multiple peripherals and different external busses. This design has a data bus, instruction bus, peripheral bus and numerous peripherals. All the peripherals were grouped in an essential core group and a reconfigurable group. The core peripherals and the CPU busses should all fit inside the fixed area (partial reconfigurable design should have a fixed area). This area should be as small as possible to ensure that a large area is reserved for the partial reconfigurable blocks, and if possible, should not cover multiple clock regions as these define the minimum size of the PR blocks.

To reduce the power consumption all IOB's are used at the lowest possible voltage. This voltage is dictated by the peripherals connected to it. There are two voltages used: 3.3V and 2.5V. Similar voltage peripherals and busses are grouped together and placed on IOB's. There are some instances where it was not possible to do this, then level shifting was done to ensure proper operation.