Chapter 1

Introduction

The fundamental principle underlying parallel (or concurrent) processing is that once the limits on speed imposed by a certain computing technology have been reached, the most obvious way of building a faster computer is to perform operations simultaneously. Two fundamental ways of implementing parallelism have emerged:

- Pipelining
- Replication

Pipelining means overlapping parts of operations in time. Replication means providing more than one functional unit. Some form of parallelism has long been a feature of computer designs. By the 196Os, most scientific computers were processing the bits of a word in parallel - an example of replication. These forms of parallelism are probably ideal from the user's viewpoint, since they are entirely transparent to the user, and all he sees is a machine with faster through-put. The next stage of computer development involved pipelining, that is, the overlapping of operations in time. These operations could be instruction processing where the operation of instruction fetch, decode, address calculation, and operand fetch, are overlapped on successive operations.

Parallel processing enhancements can be divided into two broad categories: on-chip and off-chip. On-chip parallelism relies on architectural enhancements for improved performance, while off-chip parallelism incorporates additional processors.

1.1 On-Chip Parallel Processing

Architectural enhancements on RISC processors can be grouped into three distinct categories: superpipelining, superscaling, and multi-CPU integration.

1.1.1 Superpipelining

This technique breaks the instruction pipeline into smaller pipeline stages, allowing the CPU to start executing the next instruction before completing the previous one. The processor can run multiple instructions simultaneously, with each instruction being at a different stage of completion.

The main drawbacks of this technique are the increased level of control logic on the processor, difficulty in programming, and difficulty in task switching. Real-time multitasking on a superpipelined processor can become impossible if the pipeline grows too deep.

1.1.2 Superscaling

Instead of breaking the pipeline into smaller stages, superscaling creates multiple pipelines within a processor, allowing the CPU to execute multiple instructions simultaneously. However, when multiple instructions are executed simultaneously, any data dependency between the instructions (such as a conditional branch)increases the complexity of the programming. Programmers must make certain that simultaneously executed instructions don’t need the same-on-chip resource, or that one executing instruction doesn’t need the result of another whose result is not yet available.

1.1.3 Multi-CPU Integration

This technique goes a step further than the preceding techniques and integrates multiple CPUs into a single piece of silicon. The number of processors may vary, depending on chip size, power dissipation, and pin count.

All three of these parallel processing techniques increase processor performance without the need for dramatic cycle time improvement. None of the techniques, however, can achieve the BIPS performance required by today’s applications. If an application demands higher performance than on-chip processors can deliver, the solution must be multiple processors.

1.2 Off-Chip Parallel Processing

Off-chip parallel processing is not necessarily better — it's inevitable. No single processor, no matter how it is pipelined, how it is scaled, or how many CPUs it has on board, can handle all applications. Recognizing this, manufacturers developed techniques to integrate multiple processors efficiently. Like building blocks, off-chip parallel processors connect easily to form expandable systems of virtually infinite size and variety. Off-chip expansion is achieved by connecting multiple processors together with zero glue logic for direct processor-to-processor communication. While methods are different, the concept is the same: connect multiple processors together to create a topology or array of virtually any size to achieve the performance needed by high-end applications. The communication ports (or links) on the devices are supplemented by parallel memory buses and other support peripherals, allowing designers broad flexibility in designing their systems.

These are some benefits of off-chip parallelism:

- **Expandability** — you can easily add more processors to your system to meet performance requirements.
- **Flexibility** — you can implement a wide array of processor topologies that best fit your application needs. Unlike hardwired multi-CPU integration, off-chip processing can implement everything from 1D pipelines to 4D hypercubes.
- **Upgradability** — with processors that connect like building blocks, systems can be designed in a modular fashion, allowing extra processing power to be added at a later date to meet expanding processing needs.

Chapter 2

Literature Survey

For development of parallel processing embedded system, we have to find the two different processors which could support the parallel processing. Here I work on MSP430 & FPGA Virtex-4 processors. I choose MSP430 because it is a low power processor which supports parallel processing also. Likewise Virtex-4 is a high speed FPGA which can run on 50Mhz.

2.1 MSP430

The MSP430 is a 16-bit microcontroller that has a number of special features not commonly available with other microcontrollers:

- Complete system on-a-chip — includes LCD control, ADC, I/O ports, ROM, RAM, basic timer, watchdog timer, UART, etc.
- Extremely low power consumption — only 4.2 nW per instruction, typical
- High speed — 300 ns per instruction @ 3.3 MHz clock, in register and register addressing mode
- RISC structure — 27 core instructions
- Orthogonal architecture (any instruction with any addressing mode)
- Seven addressing modes for the source operand
- Four addressing modes for the destination operand
- Constant generator for the most often used constants (–1, 0, 1, 2, 4, 8)
- Only one external crystal required — a frequency locked loop (FLL) oscillator derives all internal clocks
- Full real-time capability — stable, nominal system clock frequency is available after only six clocks when the MSP430 is restored from low-power mode (LPM) 3; — no waiting for the main crystal to begin oscillation and stabilize.

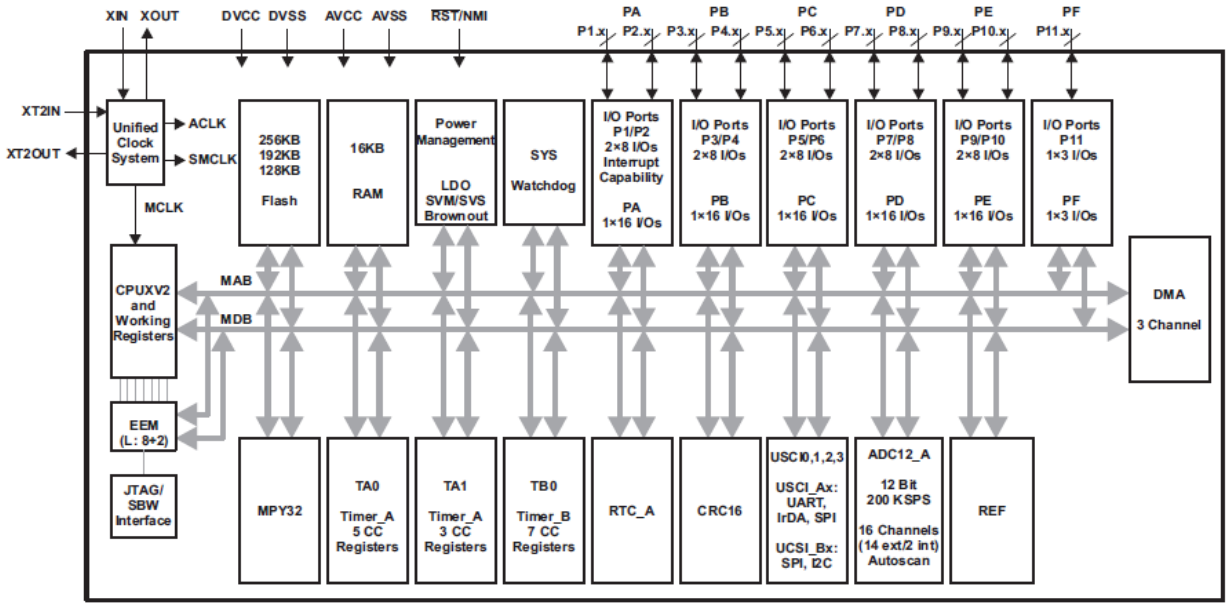


Fig 2.1 Functional Block Diagram of MSP430

2.1.1 Central Processing Unit

The CPU incorporates a reduced and highly transparent instruction set and a highly orthogonal design. It consists of a 16-bit arithmetic logic unit (ALU), 16 registers, and instruction control logic. Four of these registers are used for special purposes. These are the program counter (PC), stack pointer (SP), status register (SR), and constant generator (CGx). All registers, except the constant-generator registers R3/CG2 and part of R2/CG1, can be accessed using the complete instruction set. The constant generator supplies instruction constants, and is not used for data storage. The addressing mode used on CG1 separates the data from the constants.

The CPU control over the program counter, the status register, and the stack pointer (with the reduced instruction set) allows the development of applications with sophisticated addressing modes and software algorithms.

2.1.2 Program Memory

Instruction fetches from program memory are always 16-bit accesses, whereas data memory can be accessed using word (16-bit) or byte (8-bit) instructions. Any access uses the 16-bit memory data bus (MDB) and as many of the least-significant address lines of the

memory address bus (MAB) as required to access the memory locations. Blocks of memory are automatically selected through module-enable signals. This technique reduces overall current consumption. Program memory is integrated as programmable or mask-programmed memory.

In addition to program code, data may also be placed in the ROM section of the memory map and may be accessed using word or byte instructions; this is useful for data tables, for example. This unique feature gives the MSP430 an advantage over other microcontrollers, because the data tables do not have to be copied to RAM for usage. Sixteen words of memory are reserved for reset and interrupt vectors at the top of the 64- kilobytes address space from 0FFFFh down to 0FFE0h.

2.1.3 Data Memory

The data memory is connected to the CPU through the same two buses as the program memory (ROM): the memory address bus (MAB) and the memory data bus (MDB). The data memory can be accessed with full (word) data width or with reduced (byte) data width. Additionally, because the RAM and ROM are connected to the CPU via the same busses, program code can be loaded into and executed from RAM. This is another unique feature of the MSP430 devices, and provides valuable, easy-to-use debugging capability.

2.1.4 Operation Control

The operation of the different MSP430 members is controlled mainly by the information stored in the special–function registers (SFRs). The different bits in the SFRs enable interrupts, provide information about the status of interrupt flags, and define the operating modes of the peripherals. By disabling peripherals that are not needed during an operation, total current consumption can be reduced. The individual peripherals are described later in this manual.

2.1.5 Peripherals

Peripheral modules are connected to the CPU through the MAB, MDB, and interrupt service and request lines. The MAB is usually a 5-bit bus for most of the peripherals. The MDB is an 8- bit or 16-bit bus. Most of the peripherals operate in byte format. Modules with an 8-bit data bus are connected by bus-conversion circuitry to the 16-bit CPU. The data

exchange with these modules must be handled with byte instructions. The SFRs are also handled with byte instructions.

2.1.6 Oscillator and Clock Generator

The oscillator is designed for the commonly used 32,768 Hz, low-current consumption clock crystal. All analog components are integrated into the MSP430x3xx; only the crystal needs to be connected with no other external components required.

In addition to the crystal oscillator, all MSP430 devices contain a digitally controlled RC oscillator (DCO). The DCO is different from RC oscillators found on other microcontrollers because it is digitally controllable and tuneable. MSP430x3xx devices contain an additional logic block called the frequency locked loop (FLL).

The FLL continuously and automatically adjusts the frequency of the DCO relative to the 32768-Hz crystal oscillator to stabilize the DCO over voltage and temperature. This provides an effective, stable, ultralow-power oscillator for the CPU and peripherals. Clock source selection for peripherals is very flexible. Most peripherals are capable of using the 32768-Hz crystal oscillator clock or the DCO clock. The CPU executes from the DCO clock.

2.1.7 Operating Modes

The MSP430 family was developed for ultra-low power applications and uses different levels of operating modes. The MSP430 operating modes, shown in Figure, give advanced support to various requirements for ultra-low power and ultra-low energy consumption. This support is combined with an intelligent management of operations during the different module and CPU states. An interrupt event wakes the system from each of the various operating modes and the RETI instruction returns operation to the mode that was selected before the interrupt event.

The ultra-low power system design which uses complementary metal-oxide semiconductor (CMOS) technology, takes into account three different needs:

- The desire for speed and data throughput despite conflicting needs for ultralow-power
- Minimization of individual current consumption
- Limitation of the activity state to the minimum required by the use of low-power modes

There are four bits that control the CPU and the system clock generator: CPUOff, OscOff, SCG0, and SCG1. These four bits support discontinuous active mode (AM) requests, to limit the time period of the full operating mode, and are located in the status register. The major advantage of including the operating mode bits in the status register is that the present state of the operating condition is saved onto the stack during an interrupt service request.

As long as the stored status register information is not altered, the processor continues (after RETI) with the same operating mode as before the interrupt event. Another program flow may be selected by manipulating the data stored on the stack or the stack pointer. Being able to access the stack and stack pointer with the instruction set allows the program structures to be individually optimized, as illustrated in the following program flow:

- Enter interrupt routine

The interrupt routine is entered and processed if an enabled interrupt awakens the MSP430:

- The SR and PC are stored on the stack, with the content present at the interrupt event.
- Subsequently, the operation mode control bits OscOff, SCG1, and CPUOff are cleared automatically in the status register.

- Return from interrupt

Two different modes are available to return from the interrupt service routine and continue the flow of operation:

- Return with low-power mode bits set. When returning from the interrupt, the program counter points to the next instruction. The instruction pointed to is not executed, since the restored lowpower mode stops CPU activity.
- Return with low-power mode bits reset. When returning from the interrupt, the program continues at the address following the instruction that set the OscOff or CPUOff-bit in the status register. To use this mode, the interrupt service routine must reset the OscOff, CPUOff, SCGO, and SCG1 bits on the stack. Then, when the SR contents are popped from the stack upon RETI, the operating mode will be active mode (AM).

The software can configure one active mode and five operating modes:

1. Active mode AM; SCG1=0, SCG0=0, OscOff=0, CPUOff=0:
    - CPU clocks are active
2. Low-power mode 0 (LPM0); SCG1=0, SCG0=0, OscOff=0, CPUOff=1:
    - CPU is disabled
    - ACLK and MCLK remain active
    - Loop control for MCLK remains active
3. Low-power mode 1 (LPM1); SCG1=0, SCG0=1, OscOff=0, CPUOff=1:
    - CPU is disabled
    - Loop control for MCLK is disabled
    - ACLK and MCLK remain active
4. Low-power mode 2 (LPM2); SCG1=1, SCG0=0, OscOff=0, CPUOff=1:
    - CPU is disabled
    - MCLK and loop control for MCLK are disabled
    - DCO's dc-generator remains enabled
    - ACLK remains active
5. Low-power mode 3 (LPM3); SCG1=1, SCG0=1, OscOff=0, CPUOff=1:
    - CPU is disabled
    - MCLK and loop control for MCLK are disabled
    - DCO oscillator is disabled
    - DCO's dc-generator is disabled
    - ACLK remains active
6. Low-power mode 4 (LPM4); SCG1=X, SCG0=X, OscOff=1, CPUOff=1:
    - CPU is disabled
    - ACLK is disabled
    - MCLK and loop control for MCLK are disabled
    - DCO oscillator is disabled
    - DCO's dc-generator is disabled
    - Crystal oscillator is stopped

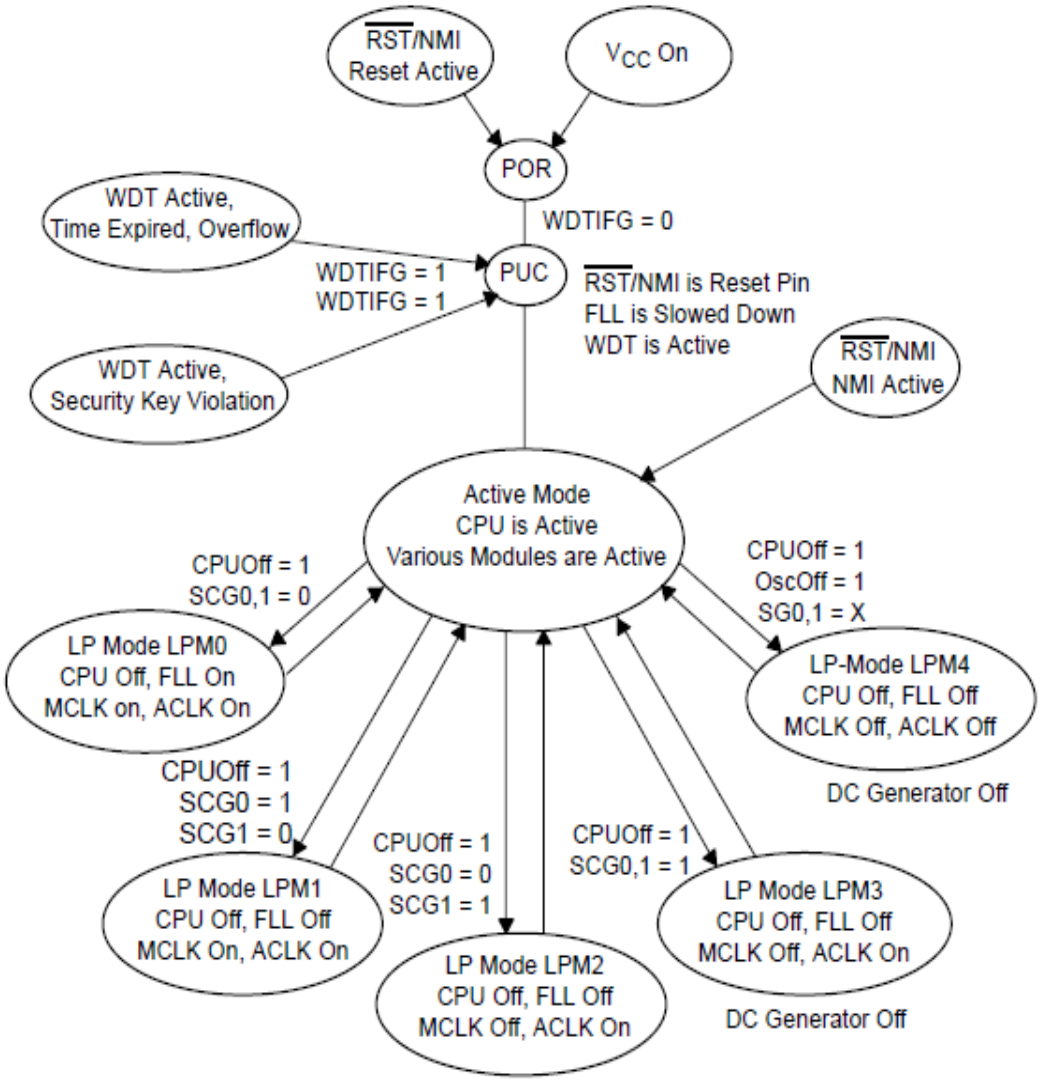


Fig 2.2 Active and Low power Modes

2.1.8 Interrupt Processing

The MSP430 programmable interrupt structure allows flexible on-chip and external interrupt configurations to meet real-time interrupt-driven system requirements. Interrupts may be initiated by the processor's operating conditions such as watchdog overflow; or by peripheral modules or external events. Each interrupt source can be disabled individually by an interrupt enable bit, or all maskable interrupts can be disabled by the general interrupt enable (GIE) bit in the status register.

Whenever an interrupt is requested and the appropriate interrupt enable bit and general interrupt enable (GIE) bit are set, the interrupt service routine becomes active as follows:

- CPU active: The currently executing instruction is completed.
- CPU stopped: The low-power modes are terminated.
- The program counter pointing to the next instruction is pushed onto the stack.
- The status register is pushed onto the stack.
- The interrupt with the highest priority is selected if multiple interrupts occurred during the last instruction and are pending for service.
- The appropriate interrupt request flag resets automatically on single source flags. Multiple source flags remain set for servicing by software.
- The GIE bit is reset; the CPUOff bit, the OscOff bit, and the SCG1 bit are cleared; the status bits V, N, Z, and C are reset. SCG0 is left unchanged, and loop control remains in the previous operating condition.
- The content of the appropriate interrupt vector is loaded into the program counter: the program continues with the interrupt handling routine at that address.

The interrupt latency is six cycles, starting with the acceptance of an interrupt request, and lasting until the start of execution of the appropriate interrupt-service routine first instruction. The interrupt handling routine terminates with the instruction: RETI (return from an interrupt service routine) which performs the following actions:

- The status register with all previous settings pops from the stack. All previous settings of GIE, CPUOFF, etc. are now in effect, regardless of the settings utilized during the interrupt service routine.
- The program counter pops from the stack and begins execution at the point where it was interrupted.

2.1.9 Memory Mapping

All of the physically separated memory areas (ROM, RAM, SFRs, and peripheral modules) are mapped into the common address space, as shown in Figure for the MSP430 family. The addressable memory space is 64KB. Future expansion is possible. The memory data bus (MDB) is 16- or 8-bits wide. For those modules that can be accessed with word data the width is always 16 bits. For the other modules, the width is 8 bits, and they must be accessed using byte instructions only. The program memory (ROM) and the data memory (RAM) can be accessed with byte or word instructions.

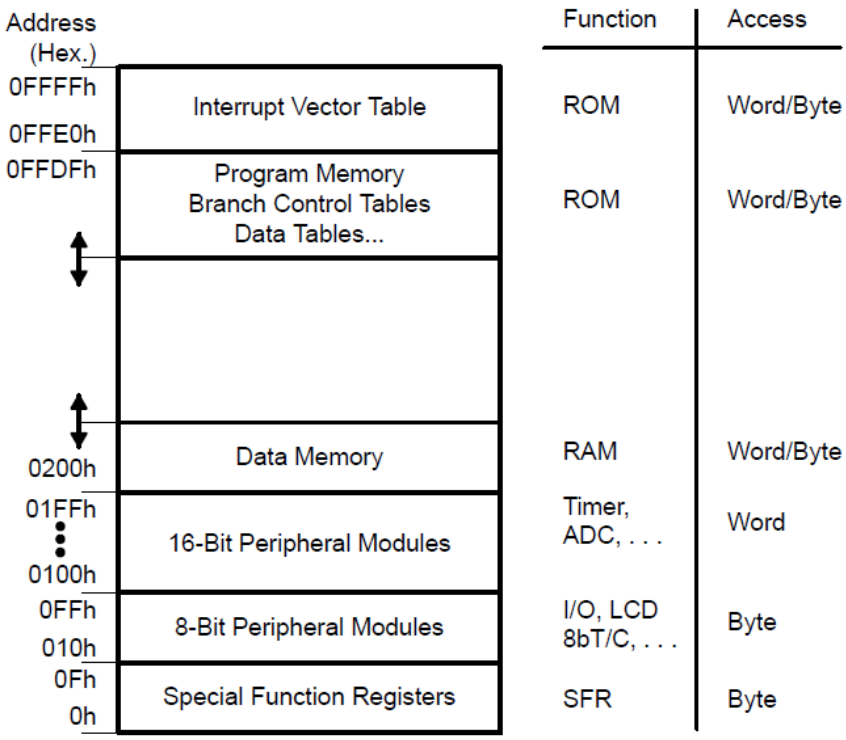


Fig 2.3 Memory Mapping

Bytes are located at even or odd addresses as shown in Figure. However, words are only located at even addresses. Therefore, when using word instructions, only even addresses may be used. The low byte of a word is always at an even address. The high byte of a word is at the next odd address after the address of the word. For example, if a data word is located at address xxx2h, then the low byte of that data word is located at address xxx2h, and the high byte of that word is located at address xxx3h.

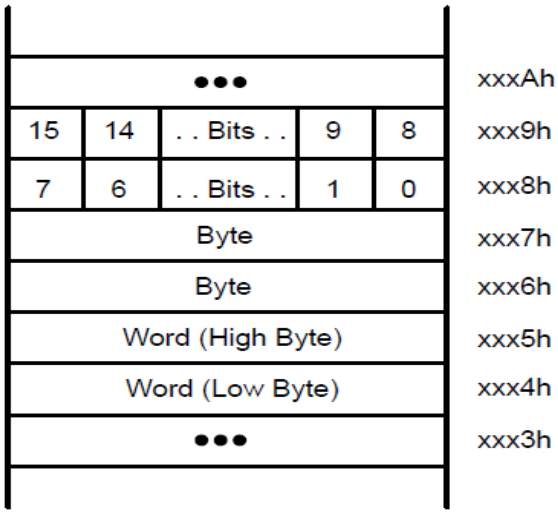


Fig 2.4 Memory Address

2.1.10 16-Bit CPU

The MSP430 von-Neumann architecture has RAM, ROM, and peripherals in one address space, both using a single address and data bus. This allows using the same instruction to access either RAM, ROM or peripherals and also allows code execution from RAM.

2.1.10.1 CPU Registers

Sixteen 16-bit registers (R0, R1, and R4 to R15) are used for data and addresses and are implemented in the CPU. They can address up to 64 Kbytes (ROM, RAM, peripherals, etc.) without any segmentation. The complete CPU-register set is described in Table. Registers R0, R1, R2, and R3 have dedicated functions.

Program counter (PC)	R0
Stack pointer (SP)	R1
Status register (SR)	R2
Constant generator (CG1)	
Constant generator (CG2)	R3
Working register R4	R4
Working register R5	R5
:	:
:	:
Working register R13	R13
Working register R14	R14
Working register R15	R15

Fig 2.5 CPU Registers

2.1.10.2 The Status Register (SR)

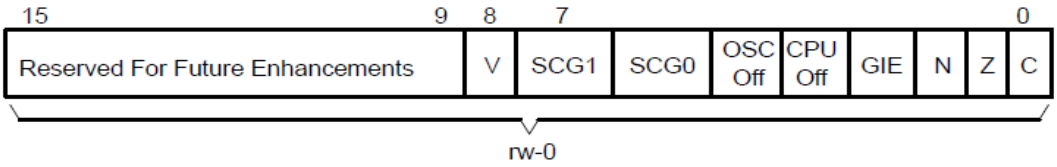


Fig 2.6 Status Register

The status register SR contains the following CPU status bits:

- V Overflow bit
- SCG1 System clock generator control bit 1
- SCG0 System clock generator control bit 0
- OscOff Crystal oscillator off bit
- CPUOff CPU off bit
- GIE General Interrupt enable bit
- N Negative bit
- Z Zero bit
- C Carry bit

2.1.11 FLL Clock Module

The frequency-locked loop (FLL) clock module (shown in Figure 7–1) follows the major design targets of low system cost and low-power consumption. The FLL operates completely using a 32768-Hz watch crystal. A second asynchronous high-speed clock signal is generated on-chip using a digitally-controlled oscillator (DCO). The DCO frequency is stabilized to a multiple of the watch crystal frequency by dividing the DCO frequency and digitally locking the two frequencies. This technique is known as frequency-locked loop.

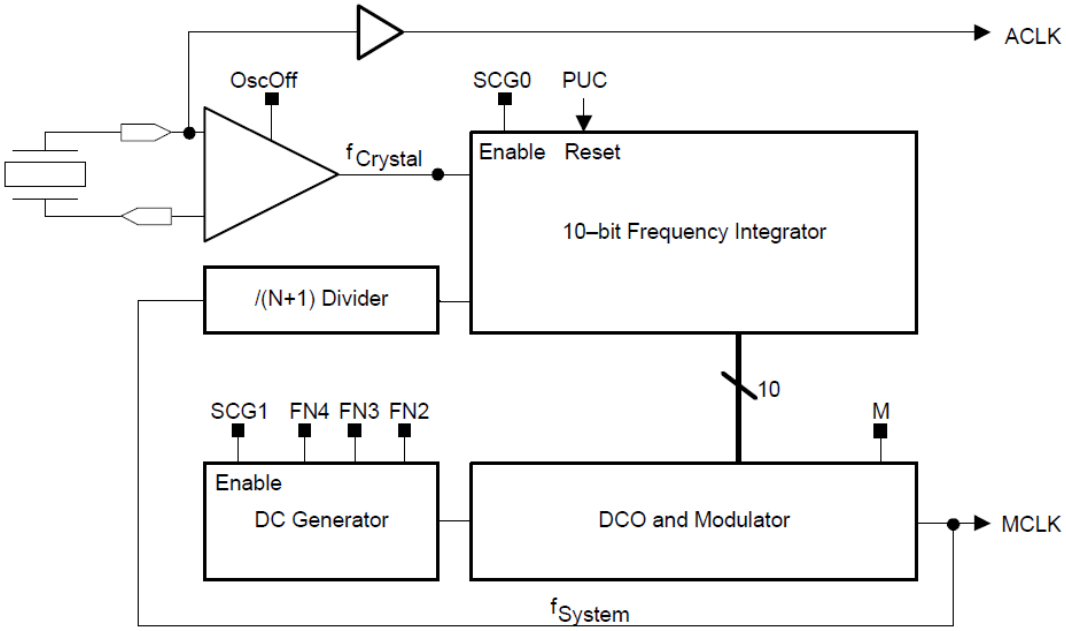


Fig 2.7 FLL Clock Module

2.1.12 Digital I/O Configuration

The general-purpose I/O ports of the MSP430 are designed to give maximum flexibility. Each I/O line is individually configurable, and most have interrupt capability. There are several different I/O port modules that function in slightly different ways. For this reason, names have been given to each port module. For example, port P0, P1, P2, etc. These names refer to specific port modules, and apply to all MSP430 devices. For example, port P0 and P1 may be available on a particular MSP430 device, while ports P1 and P3 may be available on another device. It is important for the user to understand the operating differences and which port(s) are available on the device in use. Additionally, the I/O port pins are often multiplexed with other pin functions on the devices to provide maximum flexibility while optimizing pin count on the devices.

- **Port P0**

The general-purpose port P0 contains 8 general-purpose I/O lines and the required registers to control and configure them. Each I/O line is capable of being controlled independently. In addition, each I/O line has interrupt capability. Six registers are used to control the port I/O pins.

Port P0 is connected to the processor core through the 8-bit memory data bus (MDB) and the memory address bus (MAB). Port P0 should be accessed using byte instructions in the absolute address mode.

- **Ports P1, P2**

Each of the general-purpose ports P1 and P2 contain 8 general-purpose I/O lines and all of the registers required to control and configure them. Each I/O line is capable of being controlled independently. In addition, each I/O line is capable of producing an interrupt.

Separate vectors are allocated to ports P1 and P2 modules. The pins for port P1 (P1.0–7) source one interrupt, and the pins for port P2 (P2.0–7) source another interrupt. Seven registers are used to control the port I/O pins. Ports P1 and P2 are connected to the processor core through the 8-bit MDB and the MAB. They should be accessed using byte instructions in the absolute address mode.

- **Ports P3**

P3 functions as general-purpose ports. Each pin can be selected to operate with the I/O port function, or to be used with a different peripheral module. This multiplexing of pins allows for a reduced pin count on MSP430 devices.

Four registers control each of the ports. Ports P3 is connected to the processor core through the 8-bit MDB and the MAB. They should be accessed with byte instructions using the absolute address mode.

2.1.13 Timers

2.1.13.1 Basic Timer1

The Basic Timer1 supplies other peripheral modules or the software with low frequency control signals. The Basic Timer1 operation supports two independent 8-bit timing/counting functions, or one 16-bit timing/counting function. Some uses for the Basic Timer1 include:

- Real-time clock (RTC)
- Debouncing keys (keyboard)
- Software time increments

2.1.13.2 8-Bit Interval Timer/Counter

The 8-Bit Timer/Counter supports three major application functions:

- Serial communication or data exchange
- Pulse counting or pulse accumulation
- Timing

2.1.13.3 Watchdog Timer

The primary function of the watchdog timer (WDT) module is to perform a controlled system restart after a software problem occurs. If the selected time interval expires, a system reset is generated. If the watchdog function is not needed in an application, the module can be configured as an interval timer and can generate interrupts at selected time intervals.

2.1.14 USART Peripheral Interface

The universal synchronous/asynchronous receive/transmit (USART) serial communication peripheral supports two serial modes with one hardware configuration. These modes shift a serial bit stream in and out of the MSP430 at a programmed rate or at a rate defined by an external clock. The first mode is the universal asynchronous receive/transmit (UART) communication protocol; the second is the serial peripheral interface (SPI) protocol.

Bit SYNC in control register UCTL selects the required mode:

SYNC = 0: UART – asynchronous mode selected

SYNC = 1: SPI – synchronous mode selected

2.1.15 ADC12+2 A-To-D Converters

The ADC12+2 features include:

- Eight analog or digital input channels
- A programmable current source on four analog pins
- Ratiometric or absolute measurement

- Built-in sample-and-hold
- End-of-conversion interrupt flag
- ADAT register that holds conversion results until the next start of conversion
- Low-power consumption
- Stand-alone conversion without CPU processing overhead
- Programmable 12-bit or 14-bit resolution
- Four programmable ranges that give 14-bit dynamic range
- Fast-conversion time
- Large supply-voltage range
- Monotonic conversion

2.1.16 Advantages of the MSP430 Concept

The MSP430 concept differs considerably from other microcontrollers and offers some significant advantages over more traditional designs.

2.1.16.1 RISC Architecture without RISC Disadvantages-

Typical RISC architectures show their highest performance in calculation- intensive applications in which several registers are loaded with input data, all calculations are made within the registers, and the results are stored back into RAM. Memory accesses (using addressing modes) are necessary only for the LOAD instructions at the beginning and the STORE instructions at the end of the calculations. The MSP430 can be programmed for such operation, for example, performing a pure calculation task in the floating point without any I/O accesses.

Pure RISC architectures have some disadvantages when running real-time applications that require frequent I/O accesses, however. Time is lost whenever an operand is fetched and loaded from RAM, modified, and then stored back into RAM. The MSP430 architecture was designed to include the best of both worlds, taking advantage of RISC features for fast and efficient calculations, and addressing modes for real-time requirements:

- The RISC architecture provides a limited number of powerful instructions, numerous registers, and single-cycle execution times.

- The more traditional microcomputer features provide addressing modes for all instructions. This functionality is further enhanced with 100% orthogonality, allowing any instruction to be used with any addressing mode.

2.1.16.2 Real-Time Capability with Ultra-Low Power Consumption

The design of the MSP430 was driven by the need to provide full real-time capability while still exhibiting extremely low power consumption. Average power consumption is reduced to the minimum by running the CPU and certain other functions of the MSP430 only when it is necessary. The rest of the time (the majority of the time), power is conserved by keeping only selected low-power peripheral functions active.

But to have a true real-time capability, the device must be able to shift from a low-power mode with the CPU off to a fully active mode with the CPU and all other device functions operating nominally in a very short time. This was accomplished primarily with the design of the system clock:

- No second high frequency crystal is used — inherent delays can range from 20 ms to 200 ms until oscillator stability is reached
- Instead, a sophisticated FLL system clock generator is used — generator output frequency (MCLK) reaches the nominal frequency within 8 cycles after activation from low power mode 3 (LPM3) or sleep mode

2.1.16.3 Digitally Controlled Oscillator Stability

The digitally controlled oscillator (DCO) is voltage and temperature dependent, which does not mean that its frequency is not stable. During the active mode, the integral error is corrected to approximately zero every 30.5 µs. This is accomplished by switching between two different DCO frequencies. One frequency is higher than the programmed MCLK frequency and the other is lower, causing the errors to essentially cancel-out. The two DCO frequencies are interlaced as much as possible to provide the smallest possible error at any given time.

2.2 Virtex-4

The Virtex-4 greatly enhances programmable logic design capabilities, making it a powerful alternative to ASIC technology. Virtex-4 FPGAs comprise three platform families—LX, FX, and SX—offering multiple feature choices and combinations to address all complex applications. The wide array of Virtex-4 FPGA hard-IP core blocks includes the PowerPC® processors (with a new APU interface), tri-mode Ethernet MACs, 622 Mb/s to 6.5 Gb/s serial transceivers, dedicated DSP slices, high-speed clock management circuitry, and source-synchronous interface blocks. The basic Virtex-4

FPGA building blocks are enhancements of those found in the popular Virtex, Virtex-E, Virtex-II, Virtex-II Pro, and Virtex-II Pro X product families, so previous-generation designs are upward compatible. Virtex-4 devices are produced on a state-of-the-art 90 nm copper process using 300 mm (12-inch) wafer technology.

2.2.1 Features

Virtex-4 has following features-

- Three Families — LX/SX/FX
    - Virtex-4 LX: High-performance logic applications solution
    - Virtex-4 SX: High-performance solution for digital signal processing (DSP) applications
    - Virtex-4 FX: High-performance, full-featured solution for embedded platform applications
- Digital clock manager (DCM) blocks
    - Additional phase-matched clock dividers (PMCD)
    - Differential global clocks
- XtremeDS Slice
    - 18 x 18, two's complement, signed Multiplier
    - Optional pipeline stages
    - Built-in Accumulator (48-bit) and Adder/Subtracter
- Smart RAM Memory Hierarchy
    - Distributed RAM
    - Dual-port 18-Kbit RAM blocks

- Optional pipeline stages
- Optional programmable FIFO logic automatically remaps RAM signals as FIFO signals
- High-speed memory interface supports DDR and DDR-2 SDRAM, QDR-II, and RLDRAM-II.

➢ SelectIO Technology

- 1.5V to 3.3V I/O operation
- Built-in ChipSyn source-synchronous technology
- Digitally controlled impedance (DCI) active termination
- Fine grained I/O banking (configuration in one bank)

➢ Flexible Logic Resources

➢ Secure Chip AES Bitstream Encryption

➢ 90 nm Copper CMOS Process

➢ 1.2V Core Voltage

➢ Flip-Chip Packaging including Pb-Free Package

Choices

➢ IBM PowerPC RISC Processor Core [*FX only*]

- PowerPC 405 (PPC405) Core
- Auxiliary Processor Unit Interface (User Coprocessor)

➢ Multiple Tri-Mode Ethernet MACs [*FX only*]

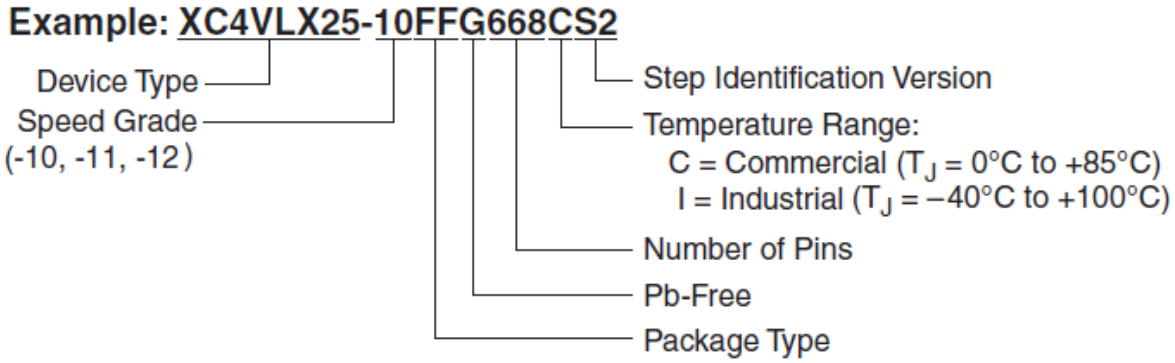


Fig 2.8 Virtex-4 FPGA Ordering Information

Chapter 3

Hardware Design

3.1 Hardware Design of MSP430

Here I use MSP430F5438A microcontroller in the embedded system design. It's a 100 pin IC which has following features.

- Low Supply Voltage Range: 1.8 V to 3.6 V
- Ultralow Power Consumption
- Wake-Up From Standby Mode in Less Than 5 μs
- 16-Bit RISC Architecture
- Unified Clock System
- Three 16-Bit Timer
- Four Universal Serial Communication Interfaces
- 12+2-Bit Analog-to-Digital (A/D) Converter
- Three Channel Internal DMA

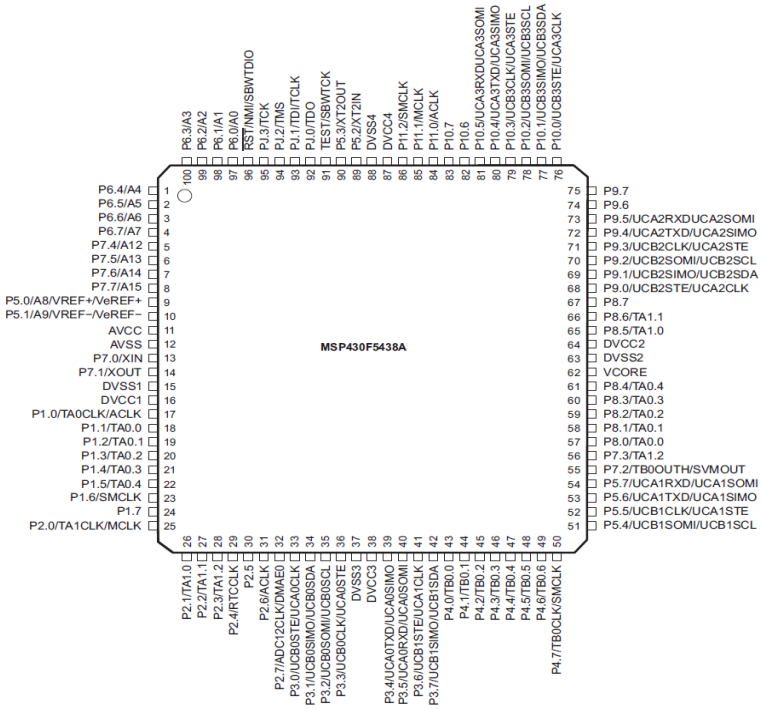


Fig 3.1 MSP430F5438A microcontroller

3.1.1 MSP-EXP430F5438 Experimenter Board

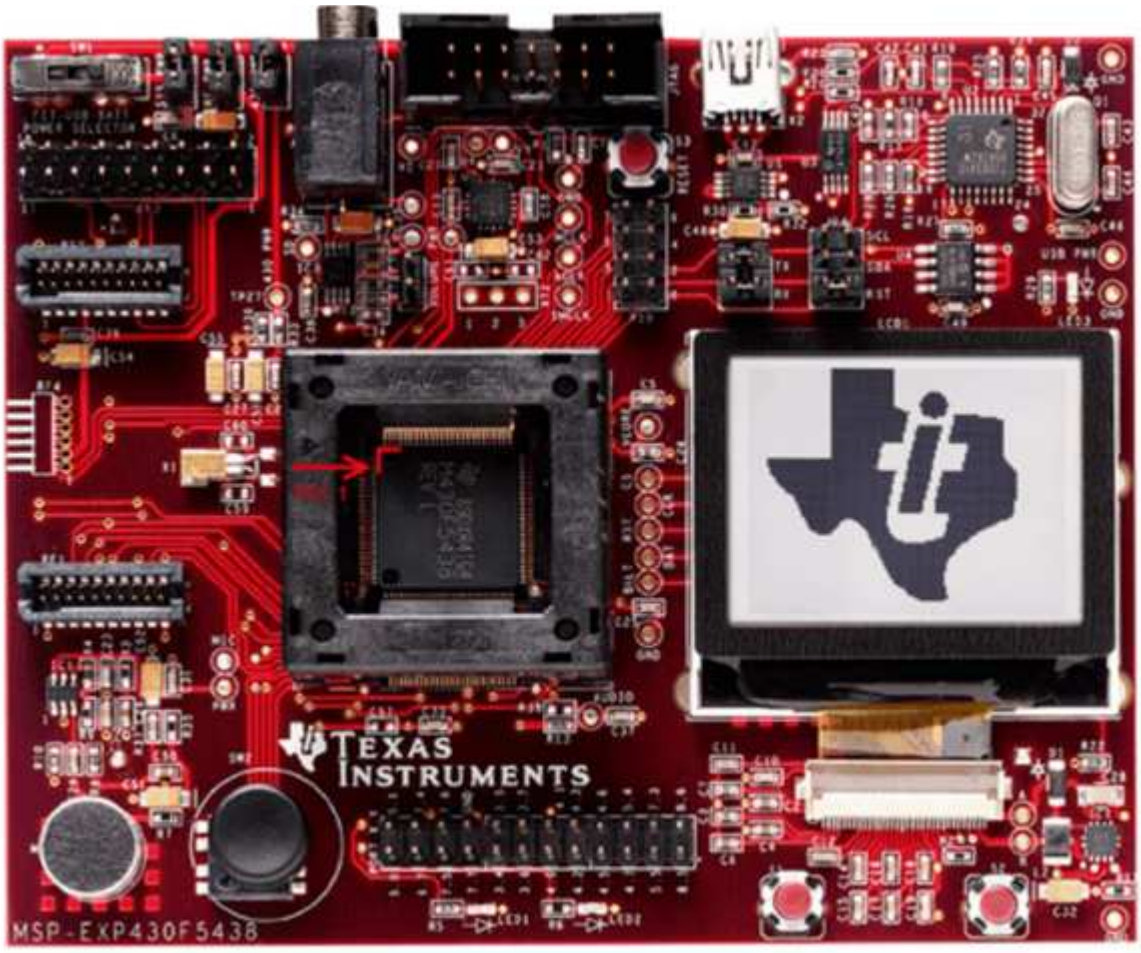


Fig 3.2 MSP-EXP430F5438 Experimenter Board

3.1.1 User Interfaces

Dot-Matrix LCD

The HD66753 is a Hitachi dot-matrix LCD with a resolution of 138 x 110, 4-level grayscale pixels. The LCD also has a built-in backlight driver that can be controlled by a PWM signal from the MSP430F5438A, pin P8.3. The MSP430F5438A communicates with the HD66753 via an SPI-like communication protocol. To supplement the limited set of instructions and functionalities provided by the on-chip LCD driver, an LCD driver has been developed for the MSP430F5438A to support additional functionalities such as font set and graphical utilities.

Five-Directional Joystick, Push Buttons, and LEDs

The following table describes the pin connections for the 5-directional joystick switch, the push button switches, and the on-board LEDs. The USB circuit on the board also sources an LED3, which indicates the presence of USB power from the mini-USB cable.

5-directional joystick (LEFT)	P2.1
5-directional joystick (RIGHT)	P2.2
5-directional joystick (CENTER)	P2.3
5-directional joystick (UP)	P2.4
Switch 1 (S1)	P2.6
Switch 2 (S2)	P2.7
RESET Switch (S3)	RST / NMI
LED1	P1.0
LED2	P1.1 / TA0 CCR0

3.1.2 Communication Peripherals

USB-UART

The USB interface on the MSP-EXP430F5438 Experimenter Board allows for UART communication with a PC host and also converts the USB power to 3.3-V power source for the entire board. The USCI module in the MSP430F5438A (UCA1) supports the UART protocol that is used to communicate with the TI TUSB chip for data transfer to the PC.

Two-Axis Accelerometer

The MSP-EXP430F5438 Experimenter Board supports a two-axis accelerometer, ADXL322. Two analog signals, one for each axis X and Y, are connected to input channels one and two of the MSP430F5438 ADC12 module, respectively. The layout also supports the

three-axis accelerometer, the ADXL330, by tracing the connection of a Z-axis to input channel three of the ADC12. To use the ADXL330, the user would need to remove the ADXL322 and correctly replace the part with the ADXL330. No further modifications to the board are required. The accelerometer is powered through pin P6.0. This interface, especially in conjunction with other on-board interfaces such as the LCD, enables several potential applications such as g-force measurement or tilt sensing.

Audio Input Signal Chain

The MSP-EXP430F5438 audio input chain is based on a noninverting op-amp gain stage positioned between the microphone and the MSP430F5438A ADC12. The circuit utilizes a Texas Instruments TLV2760, optimized for low-power operation. The power for the TLV2760 is supplied directly from MSP430F5438A port pin P6.4, which can be turned off to remove power consumption when the TLV2760 is not in use. The op-amp has a cutoff frequency of approximately 4 kHz, which targets typical speech frequency range.

The microphone is connected to the MSP430F5438A ADC12 input channel five via an analog filter circuit. The microphone is enabled or disabled via the same MSP430F5438A port pin as the TLV2760, P6.4.

Audio Output Signal Chain

The MSP430F5438A generates a high-frequency PWM signal to emulate the functionality of a DAC. The duty cycle of the PWM is derived from the ratio between the emulated voltage and the rail of 3.3 V. This PWM output signal is filtered heavily to emulate a constant voltage value. This output is then connected to a Texas Instruments TPA301 audio amplifier.

The audio output circuit utilizes the audio amplifier to amplify the filtered output signal from the PWM and feed the amplified signal into the audio output jack. The amplification is sufficient to support non-amplified headphones as well as amplified speakers.

3.2 Hardware Design of Virtex-4

In this system I use Virtex-4 ML401 evaluation board. The ML401evaluation platform enables designers to investigate and experiment with features of the Virtex-4 family of FPGAs.

Fig 3.3 Virtex-4 ML401evaluation platform

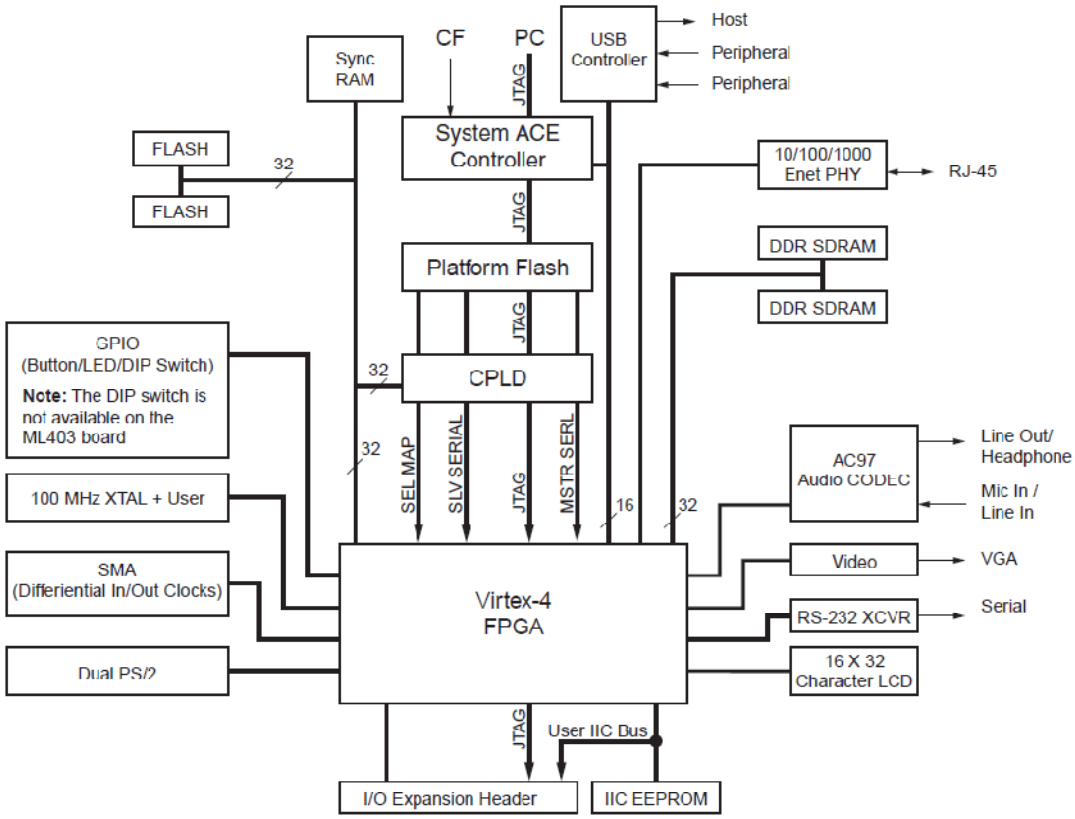


Fig 3.4 Block Diagram of ML401

3.2.1 Features

- Virtex-4 FPGA ML401: XC4VLX25-FF668-10
- 64-MB DDR SDRAM, 32-bit interface running up to 266-MHz data rate
- One differential clock input pair and differential clock output pair with SMA connectors
- One 100-MHz clock oscillator (socketed) plus one extra open 3.3V clock oscillator socket
- General purpose DIP switches (ML401/ML402 platform), LEDs, and push buttons
- Expansion header with 32 single-ended I/O, 16 LVDS capable differential pairs, 14 spare I/Os shared with buttons and LEDs, power, JTAG chain expansion capability, and IIC bus expansion
- Stereo AC97 audio codec with line-in, line-out, 50-mW headphone, and microphone-in (mono) jacks
- RS-232 serial port
- 16-character x 2-line LCD display

- One 4-Kb IIC EEPROM
- VGA output: ML401: 50 MHz / 24-bit video DAC
- PS/2 mouse and keyboard connectors
- System ACE CompactFlash configuration controller with Type I/II CompactFlash connector
- ZBT synchronous SRAM 9 Mb on 32-bit data bus with four parity bits
- Intel StrataFlash (or compatible) linear flash chips (8 MB)
- 10/100/1000 tri-speed Ethernet PHY transceiver
- USB interface chip (Cypress CY7C67300) with host and peripheral ports
- Xilinx XC95144XL CPLD to allow linear flash chips to be used for FPGA configuration
- Xilinx XCF32P Platform Flash configuration storage device
- JTAG configuration port for use with Parallel Cable III or Parallel Cable IV cable
- Onboard power supplies for all necessary voltages
- 5V @ 3A AC adapter
- Power indicator LED

Chapter 4

Parallel Architecture Design

I proposed a parallel design using MSP430 and Virtex-4 which are interfaced through USB UART with each other. In USB-UART connection the data is transferred with57600 baud rate.

Fig 4.1 Parallel architecture

4.1 Design Goal

In this parallel architecture my goal is that MSP430 works as a front hand device while Virtex-4 works as a backhand device. So the analog input data is given to the MSP430 which converts the analog data into digital form. When the conversion is completed then data is transferred to Virtex-4 platform for further processing. In Virtex-4, the data is received in digital form. After processing the data in Virtex-4, it is send to MSP430 back. Now in MSP430 the data is converted to analog from digital form and produces an output.

4.2 Design Considerations

First, we thoroughly examined MSP-EXP430F5438 Experimenter Board and Virtex-4 ML401 evaluation board. An asynchronous transfer protocol is implemented via USB UART data transfer so we have to make sure that both the platform is set for similar baud rate. If the baud rate is not similar then there is a big problem of mismatching the data synchronization.

In the design we should consider the process delay which causes the unwanted mismatch between incoming and outgoing data. As during the data communication, both the board send and receive the data simultaneously. But due to any reason, if there is a delay in any process so all the synchronization between the data is disturbed and board finds it hard to get the data correctly.

For asynchronous data transfer we have to consider SMCLK, ACLK and UCAxCLK signal responsible for generating the required baud rate. Through programming we can generate these clock signals as per the requirement.

4.3 Asynchronous Communication

The USART (Universal Synchronous/Asynchronous Receiver/Transmitter) module is a base unit for serial communications, supporting both asynchronous communications (RS232) and synchronous communications (SPI).

The USART module is available in the 5xx series devices, particularly in the sub-series MSP430F5438A.

The USART module supports:

- Low power operating modes (with auto-start);
- UART or SPI mode (I2C on ‘F15x/’F16x only);
- Double buffered TX/RX;
- Baud rate generator;
- DMA enabled;
- Error detection.

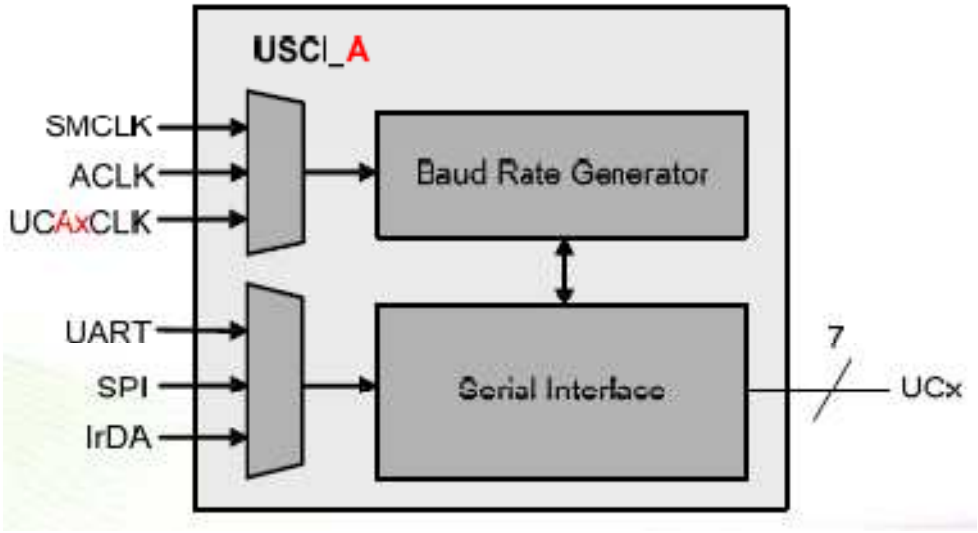


Fig 4.2 USART

Chapter 5

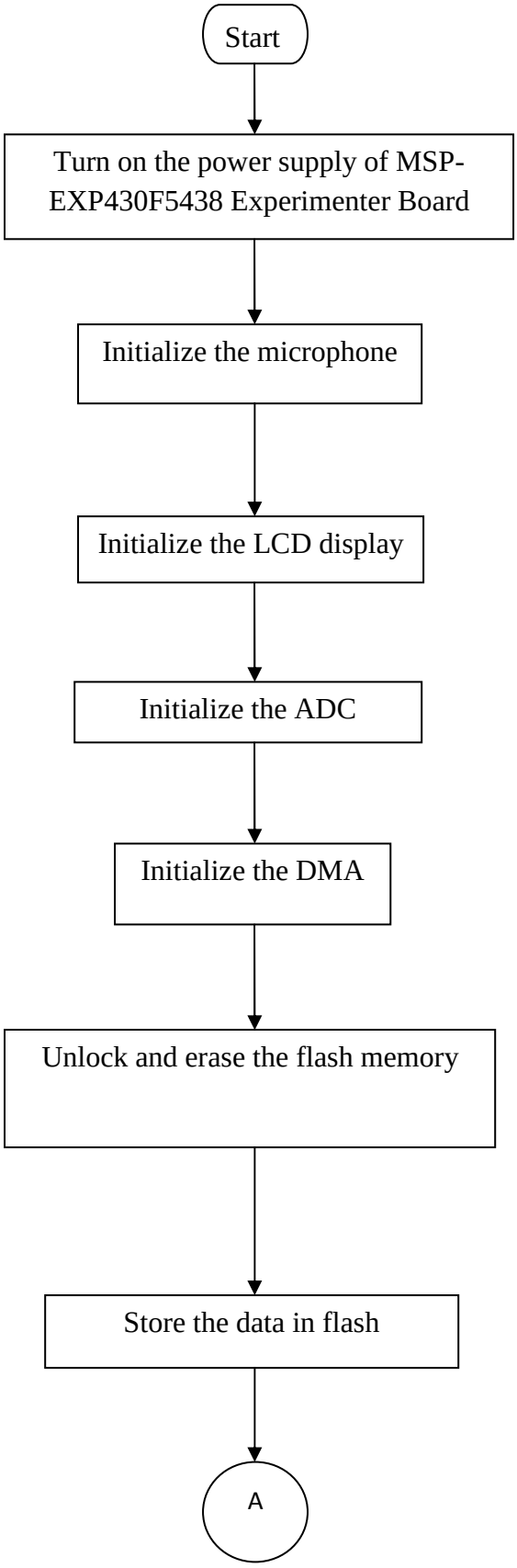
Data Flow Design

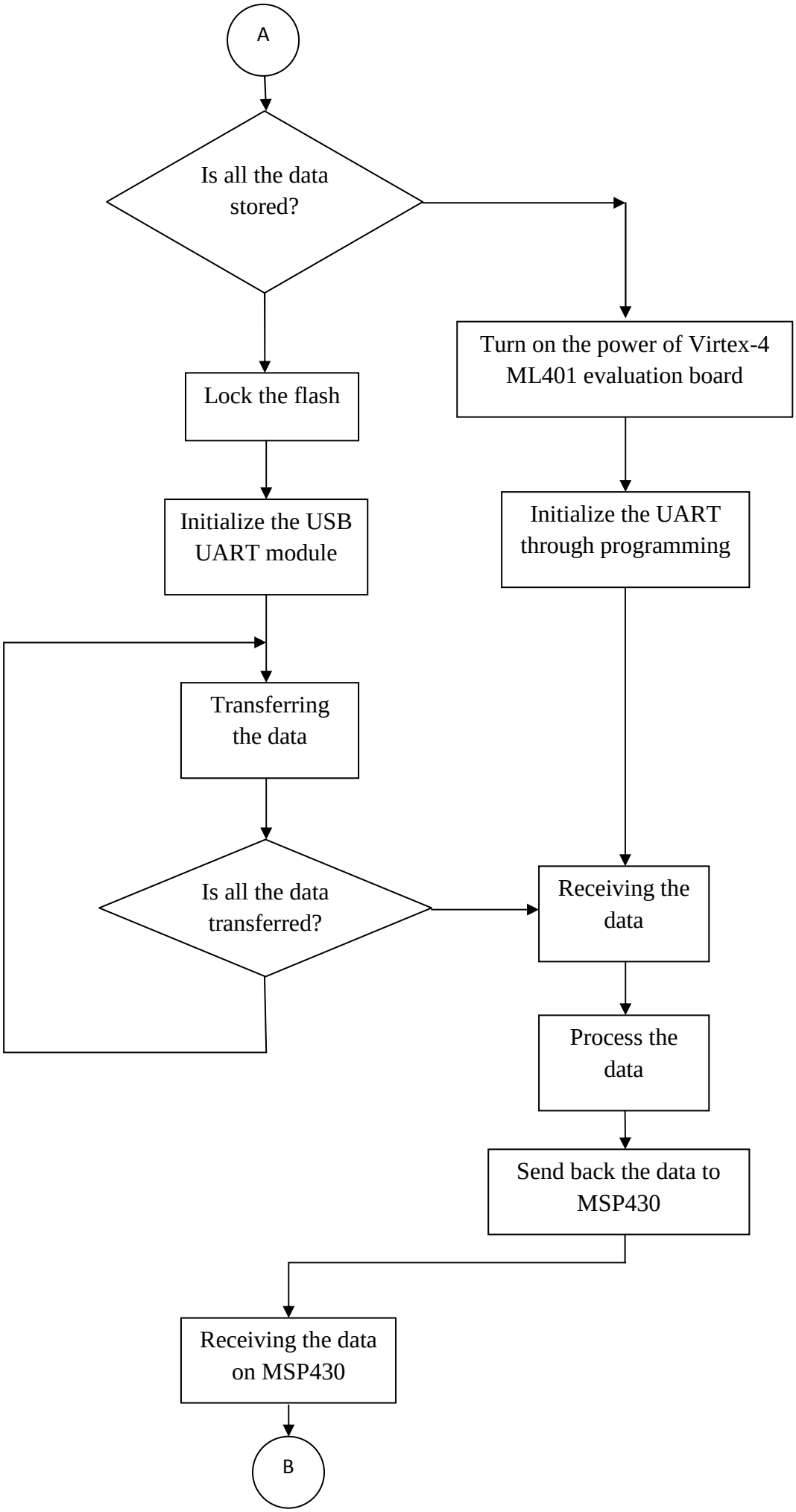
One of the main focuses of this architecture is the design of hardware organizations that support the parallel execution of instructions. Data flow parallel architectures continue to receive a great deal of attention. In a data flow architecture an instruction may execute as soon as its operands become available, permitting a degree of parallelism bounded only by the flow of data between instructions.

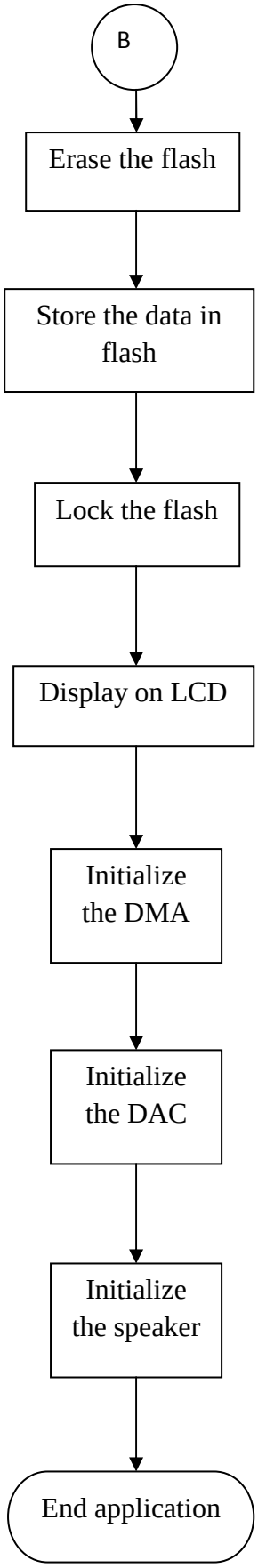
The objective is to develop inexpensive parallel architectures that can exploit parallelism without sacrificing compatibility with existing software. Compatibility with existing software is important because it represents an enormous investment for the computer user, and it is necessary to preserve this investment.

The architecture is a MIMD machine that allows the degree of parallelism to vary with time. Two types of parallelism are supported, static parallelism where the degree of parallelism is determined at compile time and dynamic parallelism where the degree of parallelism depends in part on the data being processed. There is no upper limit on the degree of parallelism. On a sequential machine each instruction has exactly one predecessor and exactly on successor, while on a data-flow machine, each instruction has several predecessors and successors. In order to support parallelism whose degree varies with time, it is necessary to have instructions that have more than one predecessor and successor.

5.1 Algorithm




A
Is all the data stored?
Turn on the power of Virtex-4 ML401 evaluation board
Lock the flash
Initialize the USB UART module
Initialize the UART through programming
Transferring the data
Is all the data transferred?
Receiving the data
Process the data
Send back the data to MSP430
Receiving the data on MSP430
B


B
Erase the flash
Store the data in flash
Lock the flash
Display on LCD
Initialize the DMA
Initialize the DAC
Initialize the speaker
End application

First we initialize the both evaluation board. We turn on the power supply of MSP-EXP430F5438 Experimenter Board. After that for running the voice application, we initialize the microphone so that we record the voice sample for this application. To convert this voice sample into digital form we initiate the 12+2 bit ADC which is already embedded on the board. After converting the data from analog to digital form, the data is stored in the flash memory by using DMA which is initializing to transfer the data from ADC channel to flash memory without interrupting the processor. Before storing the data in flash, we have to unlock and erase the flash. After this the data is stored in flash and flash is then locked. Simultaneously we initialize the LCD module to display the status of the process.

Now the data is stored in flash as a digital form. This data is transferred to other Virtex-4 ML401 board for further processing. The data is transferred through USB-UART which has to be initialized for transferring the data from one board to another evaluation board. Simultaneously we initialize the USB UART in Virtex-4 ML401 board for receiving the data from MSP430 board. We should keep the fact in mind that the baud rate should be same otherwise there is a mismatching in the data. In virtex-4 ML401 board, the data is then processed. It can be passed through FIR low pass filter to get better frequency response in lower frequency range. After this processing we send the data back to MSP430 experimental board through USB UART module.

After getting the data through USB UART, we stored the data in the flash memory. Now the data is transferred to DAC for converting the data from digital to analog format. We have to initialize DMA module for transferring the data from flash to DAC channel. After converting the data, it is given to the speaker module as output. Thus we give the input to micro phone and get back the output from speaker. The data is travelled from one processor to another FPGA processor where the data is processed in parallel by both the processors.

Chapter 6

Implementation and Result

The application is implemented on the MSP-EXP430F5438 Experimenter Board and Virtex-4 ML401 evaluation board. So we have to programme both the boards separately to work together. There are several modules that have to be programmed for initialization likewise LCD, microphone, USB-UART etc.

First we set a watchdog timer to watch the MSP430 processor. If the MSP430 is struck in any infinite loop or hang then the watchdog timer reset the processor. For initialize the watchdog timer, the programme is as follows-

```
WDTCTL = WDTPW + WDTHOLD;
```

In the voice recording, we have to initialize the microphone So that we can store the sample voice data. After that we convert the data from analog to digital form with the help of ADC. After conversion the data is stored in flash memory so we have to initialize the DMA. For initialization the microphone, ADC and DMA the software module is as follows-

```
AUDIO_PORT_OUT |= MIC_POWER_PIN;

AUDIO_PORT_OUT &= ~MIC_INPUT_PIN;

AUDIO_PORT_SEL |= MIC_INPUT_PIN;

TBCTL = TBSSEL_2;                    // Use SMCLK as Timer_B source

TBR = 0;

TBCCR0 = 2047;                       // Initialize TBCCR0

TBCCR1= 2047- 100;

TBCCTL1 = OUTMOD_7;

UCSCTL8 |= MODOSCREQEN;
```

```
ADC12CTL0 &= ~ADC12ENC;                 // Ensure ENC is clear
ADC12CTL0 = ADC12ON + ADC12SHT02;
ADC12CTL1 = ADC12SHP + ADC12CONSEQ_2 + ADC12SSEL_2 + ADC12SHS_3;
ADC12CTL2 = ADC12RES_0;                 // Select 8-bit resolution
//Sequence of channels, once
ADC12MCTL0 = MIC_INPUT_CHAN | ADC12EOS  ; //VeREF+ and VeREF-
ADC12CTL0 |= ADC12ENC;                  //Enable
ADC12CTL0 |= ADC12SC;                   //Start conversion
 DMACTL0 = DMA0TSEL_24;                 // ADC12IFGx triggers DMA0
__data16_write_addr((unsigned long)&DMA0SA & 0xffff, (unsigned
long)&ADC12MEM0);         // Src address = ADC12 module
```

For writing the flash the software module is as follows-

```
FCTL3 = FWKEY;                          // Unlock the flash for write
FCTL1 = FWKEY + BLKWRT;
DMA0CTL = DMADSTINCR_3 + DMAEN + DMADSTBYTE +  DMASRCBYTE +
DMAIE;
// Enable Long-Word write, all 32 bits will be written once
// 4 bytes are loaded
TBCCTL1 &= ~CCIFG;
TBCTL |= MC0;
  __bis_SR_register(LPM0_bits + GIE);       // Enable interrupts, enter LPM0
__no_operation();
```

```
  TBCTL &= ~MC0;
  DMA0CTL &= ~( DMAEN + DMAIE);
    FCTL3 = FWKEY + LOCK;                    // Lock the flash from write
```

For playing back the audio data stored in flash, we have to initialize the speaker, DMA and DAC modules on that board. The programme is as follows-

```
  AUDIO_PORT_DIR |= AUDIO_OUT_PWR_PIN;
  AUDIO_PORT_OUT &= ~AUDIO_OUT_PWR_PIN;
  AUDIO_OUT_SEL |= AUDIO_OUT_PIN;
// Use SMCLK as Timer0_A source, enable overflow interrupt
  TBCTL = TBSSEL_2 + TBIE;
  // Set output resolution (8 bit. Add 10 counts of headroom for loading TBCCR1
  TBCCR0 = 255;
  TBCCR4 = 255 >> 1;                    // Default output ~Vcc/2
  // Reset OUT1 on EQU1, set on EQU0. Load TBCCR1 when TBR counts to 0.
  TBCCTL4 = OUTMOD_7 + CLLD_1;
  // Start Timer_B in UP mode (counts up to TBCCR0)
  TBCTL |= MC0;
__bis_SR_register(LPM0_bits + GIE);       // Enable interrupts, enter LPM0
  __no_operation();
```

For displaying the status of the process, we have to initialize the LCD module which is programmed as follows-

```
volatile unsigned int i=0;
  LCD_CS_RST_OUT |= LCD_CS_PIN | LCD_RESET_PIN ;
  LCD_CS_RST_DIR |= LCD_CS_PIN | LCD_RESET_PIN ;
  LCD_BACKLT_SEL |= LCD_BACKLIGHT_PIN;
   LCD_CS_RST_OUT &= ~LCD_RESET_PIN;         // Reset LCD
  __delay_cycles(0x47FF);                  //Reset Pulse
  LCD_CS_RST_OUT |= LCD_RESET_PIN;
  // UCLK,MOSI setup, SOMI cleared
  LCD_SPI_SEL |= LCD_MOSI_PIN + LCD_CLK_PIN;
  LCD_SPI_SEL &= ~LCD_MISO_PIN;
  LCD_SPI_DIR &= ~(LCD_MISO_PIN + LCD_MOSI_PIN);
// Pin direction controlled by module
   // Set both pins to input as default
   // Initialize the USCI_B2 module for SPI operation
  UCB2CTL1 = UCSWRST;              // Hold USCI in SW reset mode while configuring it
  UCB2CTL0 = UCMST+UCSYNC+UCCKPL+UCMSB;     // 3-pin, 8-bit SPI master
  UCB2CTL1 |= UCSSEL_2;                  // SMCLK
  UCB2BR0 = 4;                                // Note: Do not exceed D/S spec for UCLK!
  UCB2BR1 = 0;
  UCB2CTL1 &= ~UCSWRST;              // Release USCI state machine
  UCB2IFG &= ~UCRXIFG;
```

```
// Wake-up the LCD as per datasheet specifications
halLcdActive();
// LCD Initialization Routine Using Predefined Macros
halLcdSendCommand(&LcdInitMacro[ 1 * 6 ]);
halLcdSendCommand(&LcdInitMacro[ 2 * 6 ]);
halLcdSendCommand(&LcdInitMacro[ 4 * 6 ]);
halLcdSendCommand(&LcdInitMacro[ 5 * 6 ]);
halLcdSendCommand(&LcdInitMacro[ 6 * 6 ]);
halLcdSendCommand(&LcdInitMacro[ 7 * 6 ]);
```

For Shutting down the LCD display, the programme is as follows-

```
halLcdStandby();
LCD_CS_RST_DIR |= LCD_CS_PIN | LCD_RESET_PIN ;
LCD_CS_RST_OUT &= ~(LCD_CS_PIN | LCD_RESET_PIN );
LCD_CS_RST_OUT &= ~LCD_RESET_PIN;
LCD_SPI_SEL &= ~(LCD_MOSI_PIN + LCD_CLK_PIN + LCD_MISO_PIN);
LCD_CS_RST_DIR |= LCD_MOSI_PIN + LCD_CLK_PIN + LCD_MISO_PIN;
LCD_CS_RST_OUT &= ~(LCD_MOSI_PIN + LCD_CLK_PIN + LCD_MISO_PIN);
UCB2CTL0 = UCSWRST;
```

For initializing the serial communications peripheral, the programme module is as follows-

```
unsigned char i;
for (i = 0; i < 255; i++)
halUsbReceiveBuffer[i]='\0';
bufferSize = 0;
USB_PORT_SEL |= USB_PIN_RXD + USB_PIN_TXD;
USB_PORT_DIR |= USB_PIN_TXD;
USB_PORT_DIR &= ~USB_PIN_RXD;
UCA1CTL1 |= UCSWRST;                    //Reset State
UCA1CTL0 = UCMODE_0;
UCA1CTL0 &= ~UC7BIT;                    // 8bit char
UCA1CTL1 |= UCSSEL_2;
UCA1BR0 = 16;                           // 8Mhz/57600=138
UCA1BR1 = 1;
UCA1MCTL = 0xE;
UCA1CTL1 &= ~UCSWRST;
UCA1IE |= UCRXIE;
__bis_SR_register(GIE);                 // Enable Interrupts
```

The above all programming modules are for MSP430 experimental board. All the modules are used as a function and can be called any time whenever it required.

In the Virtex-4 ML401 evaluation board, we have to programme for initialization the USB UART. In this board the receiver and transmitter module both are differently programmed. For receiver module, the programme is as follows-

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_rx is
   generic(
      DBIT: integer:=8;     -- # data bits
      SB_TICK: integer:=16  -- # ticks for stop bits
   );
   port(
      clk, reset: in std_logic;
      rx: in std_logic;
      s_tick: in std_logic;
      rx_done_tick: out std_logic;
      dout: out std_logic_vector(7 downto 0)
   );
end uart_rx ;
architecture arch of uart_rx is
   type state_type is (idle, start, data, stop);
   signal state_reg, state_next: state_type;
```

```
  signal s_reg, s_next: unsigned(3 downto 0);
  signal n_reg, n_next: unsigned(2 downto 0);
  signal b_reg, b_next: std_logic_vector(7 downto 0);
begin
  -- FSMD state & data registers
  process(clk,reset)
  begin
    if reset='1' then
      state_reg <= idle;
      s_reg <= (others=>'0');
      n_reg <= (others=>'0');
      b_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
      s_reg <= s_next;
      n_reg <= n_next;
      b_reg <= b_next;
    end if;
  end process;
  -- next-state logic & data path functional units/routing
  process(state_reg,s_reg,n_reg,b_reg,s_tick,rx)
  begin
    state_next <= state_reg;
```

```
s_next <= s_reg;
n_next <= n_reg;
b_next <= b_reg;
rx_done_tick <='0';
case state_reg is
  when idle =>
    if rx='0' then
      state_next <= start;
      s_next <= (others=>'0');
    end if;
  when start =>
    if (s_tick = '1') then
      if s_reg=7 then
        state_next <= data;
        s_next <= (others=>'0');
        n_next <= (others=>'0');
      else
        s_next <= s_reg + 1;
      end if;
    end if;
  when data =>
    if (s_tick = '1') then
      if s_reg=15 then
```

```
            s_next <= (others=>'0');
            b_next <= rx & b_reg(7 downto 1) ;
            if n_reg=(DBIT-1) then
              state_next <= stop ;
            else
              n_next <= n_reg + 1;
            end if;
          else
            s_next <= s_reg + 1;
          end if;
        end if;
      when stop =>
        if (s_tick = '1') then
          if s_reg=(SB_TICK-1) then
            state_next <= idle;
            rx_done_tick <='1';
          else
            s_next <= s_reg + 1;
          end if;
        end if;
    end case;
  end process;
  dout <= b_reg;
```

```
end arch;
```

For transmitter module, the programme is as follows-

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_tx is
  generic(
    DBIT: integer:=8;     -- # data bits
    SB_TICK: integer:=16  -- # ticks for stop bits
  );
  port(
    clk, reset: in std_logic;
    tx_start: in std_logic;
    s_tick: in std_logic;
    din: in std_logic_vector(7 downto 0);
    tx_done_tick: out std_logic;
    tx: out std_logic
  );
end uart_tx ;

architecture arch of uart_tx is
  type state_type is (idle, start, data, stop);
```

```
  signal state_reg, state_next: state_type;
  signal s_reg, s_next: unsigned(3 downto 0);
  signal n_reg, n_next: unsigned(2 downto 0);
  signal b_reg, b_next: std_logic_vector(7 downto 0);
  signal tx_reg, tx_next: std_logic;
begin
  -- FSMD state & data registers
  process(clk,reset)
  begin
    if reset='1' then
      state_reg <= idle;
      s_reg <= (others=>'0');
      n_reg <= (others=>'0');
      b_reg <= (others=>'0');
      tx_reg <= '1';
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
      s_reg <= s_next;
      n_reg <= n_next;
      b_reg <= b_next;
      tx_reg <= tx_next;
    end if;
  end process;
```

```
-- next-state logic & data path functional units/routing
process(state_reg,s_reg,n_reg,b_reg,s_tick,
        tx_reg,tx_start,din)
begin
   state_next <= state_reg;
   s_next <= s_reg;
   n_next <= n_reg;
   b_next <= b_reg;
   tx_next <= tx_reg ;
   tx_done_tick <= '0';
   case state_reg is
      when idle =>
         tx_next <= '1';
         if tx_start='1' then
            state_next <= start;
            s_next <= (others=>'0');
            b_next <= din;
         end if;
      when start =>
         tx_next <= '0';
         if (s_tick = '1') then
            if s_reg=15 then
               state_next <= data;
```

```
          s_next <= (others=>'0');
          n_next <= (others=>'0');
        else
          s_next <= s_reg + 1;
        end if;
      end if;
    when data =>
      tx_next <= b_reg(0);
      if (s_tick = '1') then
        if s_reg=15 then
          s_next <= (others=>'0');
          b_next <= '0' & b_reg(7 downto 1) ;
          if n_reg=(DBIT-1) then
            state_next <= stop ;
          else
            n_next <= n_reg + 1;
          end if;
        else
          s_next <= s_reg + 1;
        end if;
      end if;
    when stop =>
      tx_next <= '1';
```

```
            if (s_tick = '1') then
              if s_reg=(SB_TICK-1) then
                state_next <= idle;
                tx_done_tick <= '1';
              else
                s_next <= s_reg + 1;
              end if;
            end if;
        end case;
    end process;
    tx <= tx_reg;
end arch;
```

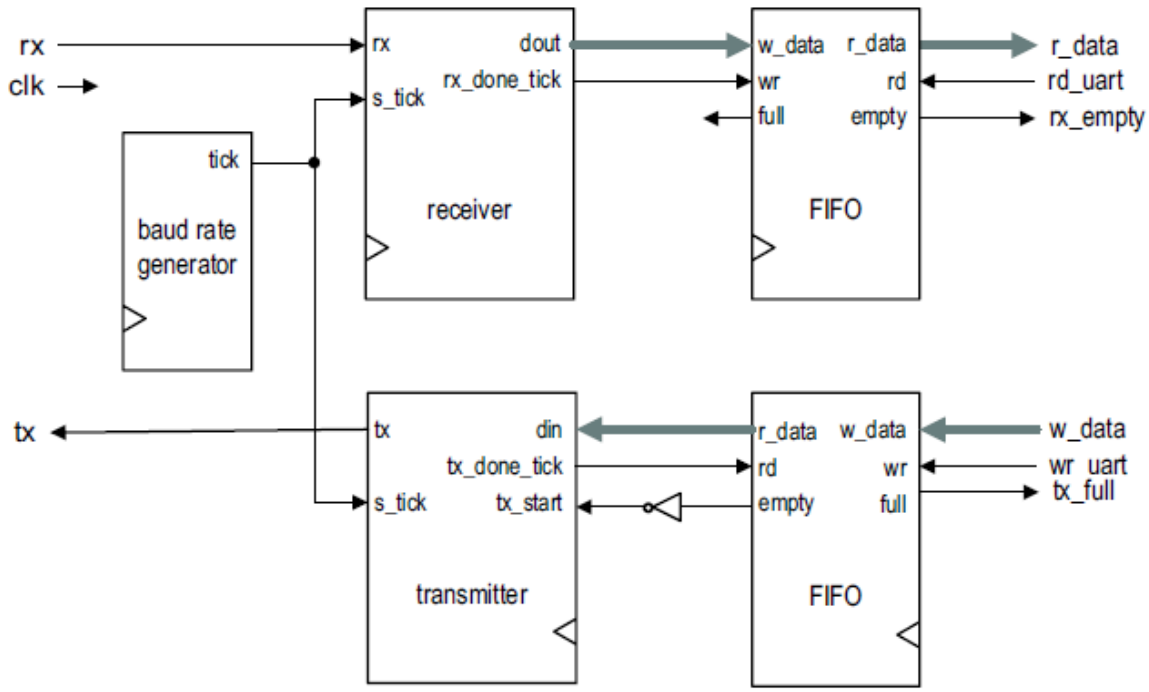


Fig 6.1 Transmitter and Receiver module

For complete UART can be constructed by combining the receiver and transmitter module. The programme code is as follows—

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart is
   generic(
     -- 19,200 baud, 8 data bit, 1 stop bit, 2^2 FIFO
     DBIT: integer:=8;     -- # data bits
      SB_TICK: integer:=16; -- # ticks for stop bits, 16/24/32
                        --   for 1/1.5/2 stop bits
      DVSR: integer:= 163;  -- baud rate divisor
                        -- DVSR = 50M/(16*baud rate)
      DVSR_BIT: integer:=8; -- # bits of DVSR
      FIFO_W: integer:=2    -- # addr bits of FIFO
                        -- # words in FIFO=2^FIFO_W
   );
   port(
      clk, reset: in std_logic;
      rd_uart, wr_uart: in std_logic;
      rx: in std_logic;
      w_data: in std_logic_vector(7 downto 0);
```

```
    tx_full, rx_empty: out std_logic;
    r_data: out std_logic_vector(7 downto 0);
    tx: out std_logic
  );
end uart;

architecture str_arch of uart is
  signal tick: std_logic;
  signal rx_done_tick: std_logic;
  signal tx_fifo_out: std_logic_vector(7 downto 0);
  signal rx_data_out: std_logic_vector(7 downto 0);
  signal tx_empty, tx_fifo_not_empty: std_logic;
  signal tx_done_tick: std_logic;
begin
  baud_gen_unit: entity work.mod_m_counter(arch)
    generic map(M=>DVSR, N=>DVSR_BIT)
    port map(clk=>clk, reset=>reset,
          q=>open, max_tick=>tick);
  uart_rx_unit: entity work.uart_rx(arch)
    generic map(DBIT=>DBIT, SB_TICK=>SB_TICK)
    port map(clk=>clk, reset=>reset, rx=>rx,
          s_tick=>tick, rx_done_tick=>rx_done_tick,
          dout=>rx_data_out);
```

```
  fifo_rx_unit: entity work.fifo(arch)
    generic map(B=>DBIT, W=>FIFO_W)
    port map(clk=>clk, reset=>reset, rd=>rd_uart,
          wr=>rx_done_tick, w_data=>rx_data_out,
          empty=>rx_empty, full=>open, r_data=>r_data);
  fifo_tx_unit: entity work.fifo(arch)
    generic map(B=>DBIT, W=>FIFO_W)
    port map(clk=>clk, reset=>reset, rd=>tx_done_tick,
          wr=>wr_uart, w_data=>w_data, empty=>tx_empty,
          full=>tx_full, r_data=>tx_fifo_out);
  uart_tx_unit: entity work.uart_tx(arch)
    generic map(DBIT=>DBIT, SB_TICK=>SB_TICK)
    port map(clk=>clk, reset=>reset,
          tx_start=>tx_fifo_not_empty,
          s_tick=>tick, din=>tx_fifo_out,
          tx_done_tick=> tx_done_tick, tx=>tx);
  tx_fifo_not_empty <= not tx_empty;
end str_arch;
```

Chapter 7

Conclusion

This example shows an application with parallel processors. These processors are interfaced with each other through USB UART. It should also be commented that parallel systems are likely to open up new fields of research on modelling methodologies that are inherently highly parallel. Another big drawback is that the standardized platforms for code development disappear with parallelization, since there is such a diversity of parallel hardware on the market, with attendant language extensions virtually for each machine. Since we are using the experimental board so there is a lot of limitations in our design.

In this embedded system design, we have to programme both the processors separately. After programming they can be interfaced with each other to work in parallel. We have to initialize the different modules embedded on boards through software programming.

In the parallel architecture design, there is a synchronizing problem in system. We should consider that the process is going to be step by step. If there is any delay in any process then all the synchronization is disturbed and system may not work accurately.

Chapter 8

Future Scope of Work

We have developed a prototype interface between the MSP430 and Virtex-4. The design is a reconfigurable, programmable interface for other processor also. While the initial transfer speed is not very fast but you can improve it by increasing the clock rate and eliminating unnecessary states in the prototype code. Ultimately, the knowledge gained from this effort could be used to develop an FPGA interface that improves both speed and size.

Another thing is that this system is developed on the evaluation board so there is a lot of limitation for designing the system. We can developed this system by using separate chips of processors, ADC, DAC, LCD and USB UART modules and connecting them accordingly. We can develop such type of application on this system so that processors can have more and more work on it.

References

- www.ti.com
- www.xilinx.com
- MSP430 User's Guide, Texas Instruments 2009
- Virtex-4 User's Guide, Xilinx
- A.K.Rath and P.K. Meher "Design of a Merged DSP Microcontroller for Embedded Systems using Discrete Orthogonal Transform" International Journal of Computer Science, pp 388-394, 2006,USA.
- Arvind, D. E. Culler, "Dataflow Architectures," Annual Reviews in Computer Science, 1986,.Vol 1, Annual Reviews Inc., 1986, pp. 225-253.
- R. H. Kuhn, D. A. Padua (eds.) "Tutorial on Parallel Processing," IEEE Computer Society Press, Silver Spring Md, 1981.
- D. B. Davidson, "A parallel processing tutorial," IEEE Antennas PropagationMagazine**,** 32**,** pp. 6-19, April, 1990.
- Texas Instruments "Parallel Processing with TMS320C4X".