

PARALLEL PROCESSING BY A MICROCONTROLLER BASED-SYSTEM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
ÇANKAYA UNIVERSITY

BY

SERDAR ÇETİNKAYA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JANUARY , 2009

Title of the Thesis: **Paralel Processing By a Microcontroller-Based System**
Submitted by **Serdar Çetinkaya**

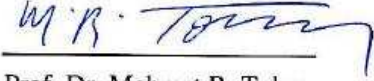
Approval of the Graduate School of Natural and Applied Sciences, Çankaya
University



Prof. Dr. Yahya K. BAYKAL

Acting Director

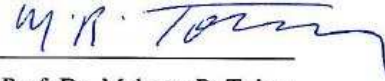
I certify that this thesis satisfies all the requirements as a thesis for the degree of
Master of Science.



Prof. Dr. Mehmet R. Tolun

Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully
adequate, in scope and quality, as a thesis for the degree Master of Science.



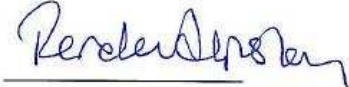
Prof. Dr. Mehmet R. Tolun

Supervisor


Examination Date : 19.01.2009

Examining Committee Members

Assoc. Prof. Ferda Nur ALPASLAN (METU)



Prof. Dr. Mehmet R. TOLUN (Çankaya Univ.)



Asst. Prof. Dr. Reza HASSANPOUR (Çankaya Univ.)



STATEMENT OF NON-PRAGIARISM

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name : Serdar Çetinkaya

Signature :



Date

: 19.01.2009

ABSTRACT

PARALLEL PROCESSING BY A MICROCONTROLLER-BASED SYSTEM

ÇETİNKAYA, Serdar

M.S.c., Department of Computer Engineering

Supervisor : Prof. Dr. Mehmet R. Tolun

JANUARY 2009, 62 pages

In this thesis, a microcontroller-based system which executes parallel process is developed. This system is designed to crack password getting the most performance. When system cracks password, system outputs execution time.

There are two software systems. One of them which is running on master and one another software system which is running on nodes compose the embedded software system. System architecture is based on MIMD architecture. Performance analysis is based on Amdahl's Law. Speedup is calculated.

Keywords: Parallel Processing, Embedded Software Development

ÖZ

MİKRODENETLEYİCİ TABANLI BİR SİSTEM İLE PARALEL İŞLEM

ÇETİNKAYA, Serdar

Yükseklisans, Bilgisayar Mühendisliği Anabilim Dalı

Tez Yöneticisi : Prof. Dr. Mehmet R. Tolun

OCAK 2009, 62 sayfa

Bu tez çalışmasında paralel işlem yapan microdenetleyici tabanlı gömülü bir sistem geliştirilmiştir. Sistem kullanıcı tarafından girilen şifreyi en yüksek performans ile çözmek amacıyla tasarlanmıştır. Sistem kullanıcının şifresini çözdüğü an harcanan işlem zamanını çıktı olarak vermektedir.

İki yazılım sistemi vardır. Master üzerinde koşan yazılım ve node'lar üzerinde koşan yazılım olmak üzere iki gömülü yazılım sistemi oluşturur. MIMD (Multiple instruction, multiple data) mimari temel alınmıştır. Değişik sayılarda node kullanılarak performans analizi için Amdahl's Law temel alınmıştır. Hızlanma(speedup) ve verimlilik(efficiency) hesaplamaları yapılmıştır.

Anahtar Kelimeler: Paralel İşlem, Gömülü Yazılım Geliştirme

ACKNOWLEDGEMENTS

The author wishes to express his deepest gratitude to his supervisor Prof. Dr. Mehmet Tolun for his guidance, advice, criticism, encouragements and insight throughout the research.

Additionally, I want to thank to my father Hasan for their support, encouragement and reliance throughout my life.

TABLE OF CONTENTS

STATEMENT OF NON-PRAGIARISM.....	iii
ABSTRACT.....	iv
ÖZ.....	v
ACKNOWLEDGMENTS.....	vi
TABLE OF CONTENTS.....	vii
LIST OF FIGURES.....	ix
CHAPTERS :	
1. INTRODUCTION.....	1
1.1 Theoretical foundations.....	2
1.1.1 Parallel Processing.....	2
1.1.2 Parallel Hardwares.....	3
1.1.3 Amdahl's Law, Speedup, Efficiency.....	4
2. CASE TOOLS AND TECHNOLOGICAL ASPECTS.....	8
2.1 Mpasm Library and Software Aspects.....	8
2.2 ISIS Simulation System.....	12
3. MICROCONTROLLER ARCHITECTURE.....	16
3.1 Memory Organization.....	16
3.2 Data EEPROM Memory.....	18
3.3 I/O Ports.....	22
3.4 Timer Modules.....	23
3.5 Special Features of the CPU.....	25
4. PARALLEL PROCESSING BY A MICROCONTROLLER-BASED SYSTEM.....	28
4.1 Ready State.....	28

4.2 Running State.....	30
4.3 Calculating Speedup.....	60
5. CONCLUSION AND FUTURE WORK.....	62
REFERENCES.....	R1

LIST OF FIGURES

FIGURES

1.1	Speedup under Amdahl's Law.....	5
1.2	Fixed-Size Model.....	6
1.3	Scaled-Size Model.....	6
2.1	Simple Assembler Process.....	9
2.2	Link Process.....	10
2.3	Library Build Process.....	11
3.1	Program Memory Map and Stack 16f84.....	17
3.2	Program Memory Map and Stack 16f877.....	18
3.3	Block diagram of the Timer0 / WDT Prescaler.....	24
4.1	System Design.....	29
4.2	System is Ready State.....	30
4.3	System is Running State.....	60
4.4	Calculating Speedup using different number of nodes and an input value.....	61
4.5	Speedup graph.....	61

CHAPTER 1

INTRODUCTION

Multiple CPU system offer the promise of both increasing the throughput of a computing system as well as decreasing job response time through the use of parallel processing. In such systems, throughput is increased by the addition of more microcontrollers and, which associated with this increased capacity, which is reduction in job response time due to decreasing queueing delays. Further reduction in job response time requires exploiting parallelism within the job by simultaneously executing a job's tasks on multiple microcontrollers.

As initially pointed out by Amdahl [1] there are fundamental limitation on speedup that are obtainable through parallel execution. Even under the ideal assumption that jobs have unbounded parallelism, cost associated with managing the cooperation between a job's tasks like synchronization or data sharing put limitations on possible speedups. An additional cost of sychronization arises from randomness in task execution times. These variations can arise either from resource sharing, such as memory conflicts, arise from the inherent randomness of a computations, or from both effects.

These random variations result in staggering task completion times and have different effects depending on parallel processing architecture [2]. An analysis of how different interconnection structure influences the efficiency. In this thesis, we focus on the task execution time to determine fundamental limitation of task sychronization. Our system assumes that all system cost associated with

synchronization are included in task execution times.

1.1. Theoretical foundations

1.1.1. Parallel Processing

Parallel processing is a form of computation in which many instructions are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level parallelism, instruction-level parallelism, data parallelism, and task parallelism. It has been used for many years, mainly in high-performance computing, but interest in it has grown in recent years due to the physical constraints preventing frequency scaling. Parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors.[3] However, in recent years, power consumption by parallel computers has become a concern.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks is typically one of the greatest barriers to getting good parallel program performance. The speed-up of a program as a result of parallelization is given by Amdahl's law which will be explained further in section 1.1.3.

Task parallelism (also known as function parallelism and control parallelism) is a form of parallelization of computer code across multiple processors in parallel

computing environments. Task parallelism focusses on distributing execution processes (threads) across different parallel computing nodes. It contrasts to data parallelism as another form of parallelism. In a multiprocessor system, task parallelism is achieved when each processor executes a different thread (or process) on the same or different data. The threads may execute the same or different code. In the general case, different execution threads communicate with one another as they work. Communication takes place usually to pass data from one thread to the next as part of a workflow. As a simple example, if we are running code on a 2-processor system in a parallel environment and we wish to do tasks "A" and "B", it is possible to tell CPU "a" to do task "A" and CPU "b" to do task "B" simultaneously, thereby reducing the runtime of the execution. The tasks can be assigned using conditional statements as described below. Task parallelism emphasizes the distributed (parallelized) nature of the processing (i.e. threads), as opposed to the data (data parallelism). Most real programs fall somewhere on a continuum between Task parallelism and Data parallelism.

1.1.2. Parallel Hardwares

The four classifications defined by Flynn [4] are based upon the number of concurrent instruction (or control) and data streams available in the architecture: Single Instruction, Single Data stream (SISD); a sequential computer which exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are the traditional uniprocessor machines like a PC or old mainframes. Single Instruction, Multiple Data streams (SIMD); a computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. For example, an array processor or GPU. Multiple Instruction, Single Data stream (MISD); multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the Space Shuttle flight control computer.

Multiple Instruction, Multiple Data streams (MIMD); multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD architectures; either exploiting a single shared memory space or a distributed memory space. Some further divide the MIMD category into the following categories: Single Program, Multiple Data streams (SPMD): Multiple autonomous processors simultaneously executing the same program (but at independent points, rather than in the lockstep that SIMD imposes) on different data. Also referred to as 'Single Process, multiple data'. SPMD is the most common style of parallel programming. Multiple Program Multiple Data (MPMD) : Multiple autonomous processors simultaneously operating at least 2 independent programs. Typically such systems pick one node to be the "host" ("the explicit host/node programming model") or "manager" (the "Manager/Worker" strategy), which runs one program that farms out data to all the other nodes which all run a second program. Those other nodes then return their results directly to the manager.

1.1.3. Amdahl's Law, Speedup, Efficiency

If N is the number of processors, s is the amount of time spent (by a serial processor) on serial parts of a program and p is the amount of time spent (by a serial processor) on parts of the program that can be done in parallel, then Amdahl's law says that speedup is given below.

$$Speedup = (s + p) / (s + p / N) = 1 / (s / p + 1 / N)$$

For $N = 1024$, this is an unforgivingly step function of s near $s = 0$ (see Figure1).

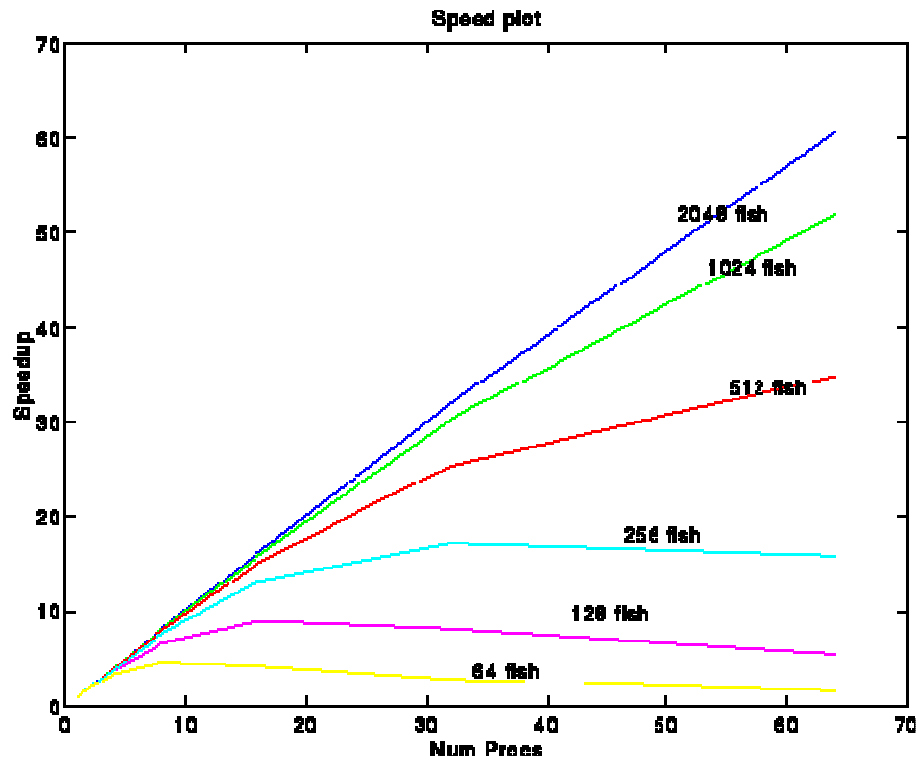


Figure 1.1: Speedup under Amdahl's Law

The expression and graph both contain the implicit assumption that p is independent of N , which is *virtually never the case*. One does not take a fixed-size problem and run it on various numbers of processors except when doing academic research; in practice, *the problem size scales with the number of processors*. When given a more powerful processor, the problem generally expands to make use of the increased facilities. Users have control over such things as grid resolution, number of timesteps, difference operator complexity, and other parameters that are usually adjusted to allow the program to be run in some desired amount of time. Hence, it may be most realistic to assume that *run time*, not *problem size*, is constant. As a first approximation, we have found that it is the parallel or vector part of a program that scales with the problem size. Times for vector startup, program loading, serial bottlenecks and I/O that make up. Component of the run do not grow with problem size. When we double the number of degrees of freedom in a physical simulation, we double the number of processors. But this means that, as a first approximation, the amount of work that

can be done in parallel *varies linearly with the number of processors*. For the three applications mentioned above, we found that the parallel portion scaled by factors of 1023.9969, 1023.9965, and 1023.9965. If we use s' and p' to represent serial and parallel time spent on the *parallel* system, then a serial processor would require time $s' + p' \times N$ to perform the task. This reasoning gives an alternative to Amdahl's law suggested by E. Barsis at Sandia:

$$\text{Scaled speedup} = (s' + p' \times N) / (s' + p') = s' + p' \times N = N + (1 - N) \times s'$$

In contrast with Figure 1, this function is simply a line, and one with much more moderate slope: $1 - N$.

It is thus much easier to achieve efficient parallel performance than is implied by Amdahl's paradigm. The two approaches, fixed-sized and scaled-sized, are contrasted and summarized in Figure 1.2 and 1.3.

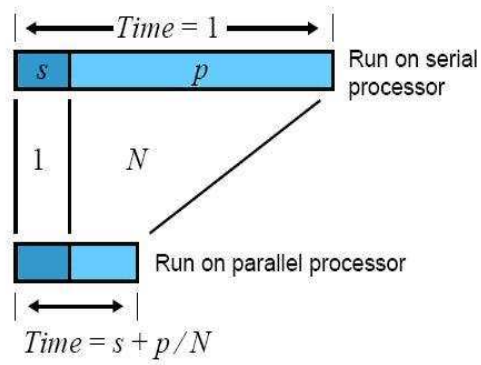


Figure 1.2 : Fixed-Size Model: $Speedup = 1 / (s + p / N)$

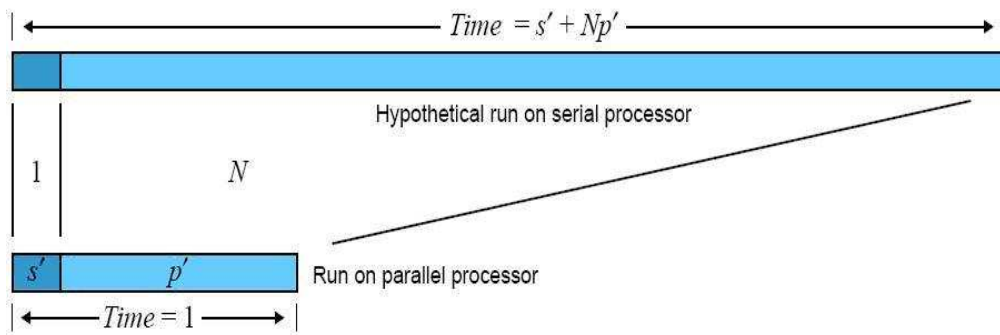


Figure 1.3 : Scaled-Size Model: $Speedup = s + Np$

The aim of the thesis is to obtain high performance using parallel processing with a suitable architecture. We want to show that depending on low cost for estimating execution time.

CHAPTER 2

CASE TOOLS AND TECHNOLOGICAL ASPECTS

2.1. Mpasm Library and Software Aspects

The MPASM assembler is a command-line or Windows-based PC application that provides a platform for developing assembly language code for Microchip's PIC[®] microcontroller (MCU) families. There are two executable versions of the assembler: The windows version (mpasmwin.exe). Use this version with MPLAB[®] IDE, in a stand-alone Windows application, or on the command line. This version is available with MPLAB IDE or with the regular and demo version of the MPLAB C18 C compiler. This is the recommended version. The command-line version (mpasm.exe). Use this version on the command line, either from a command shell or directly on the command line. This version is available with the regular and demo version of the MPLAB C18 C compiler. The MPASM assembler supports all PIC MCU devices, as well as memory and KeeLoq[®] secure data products from Microchip Technology Inc.

The MPASM assembler provides a universal solution for developing assembly code for all of Microchip's PIC MCUs. Notable features include: MPLAB IDE Compatibility, Command Line Interface, Windows/Command Shell Interfaces, Rich Directive Language, Flexible Macro Language.

Since the MPASM assembler is a universal assembler for all PIC MCU devices,

application code developed for the PIC16F877A can be translated into a program for the PIC18F452. This may require changing the few instruction mnemonics that are not the same between the devices (assuming that register and peripheral usage were similar). The rest of the directive and macro language will be the same. Included with MPLAB IDE are template files for all PIC MCUs. Template files allow you to quick set up a project in MPLAB with a generic file that can be filled in with code as you develop your application. Template files contain the basic structure of a source file, provide some examples for declaring variable storage and for setting device configuration bits.

The MPASM assembler can be used in two ways: To generate absolute code that can be executed directly by a microcontroller. To generate relocatable code that can be linked with other separately assembled or compiled modules. Relocatable code can not be executed by a microcontroller until it has been linked. Absolute code is the default output from the MPASM assembler. This process is shown below.

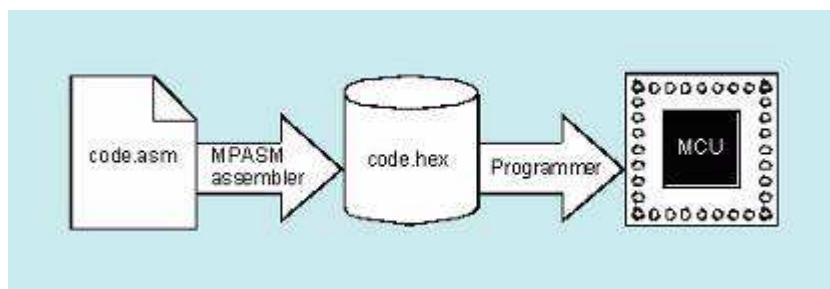


Figure 2.1: Simple Assembler Process

When a source file is assembled in this manner, all variables and routines used in the source file must be defined within that source file, or in files that have been explicitly included by that source file. If assembly proceeds without errors, a hex file will be generated, containing the executable machine code for the target

device. This file can then be used with a debugger to test code execution or with a device programmer to program the microcontroller.

The MPASM assembler also has the ability to generate a relocatable object module that can be linked with other modules using Microchip's MPLINK linker to form the final executable code. This method is very useful for creating reusable modules.

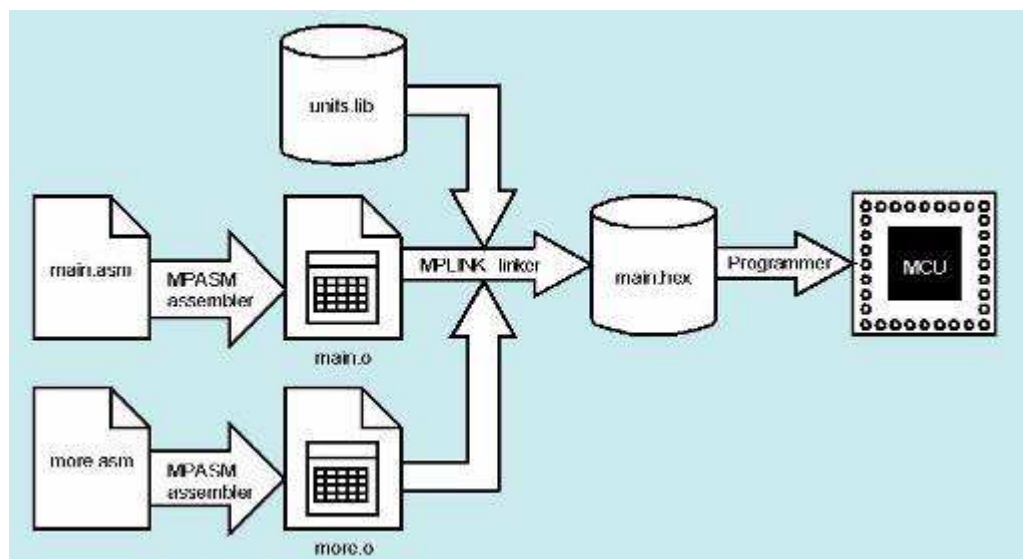


Figure 2.2: Link Process

Related modules can be grouped and stored together in a library using microchip's MPLIB librarian. Required libraries can be specified at link time, and only the routines that are needed will be included in the final executable.

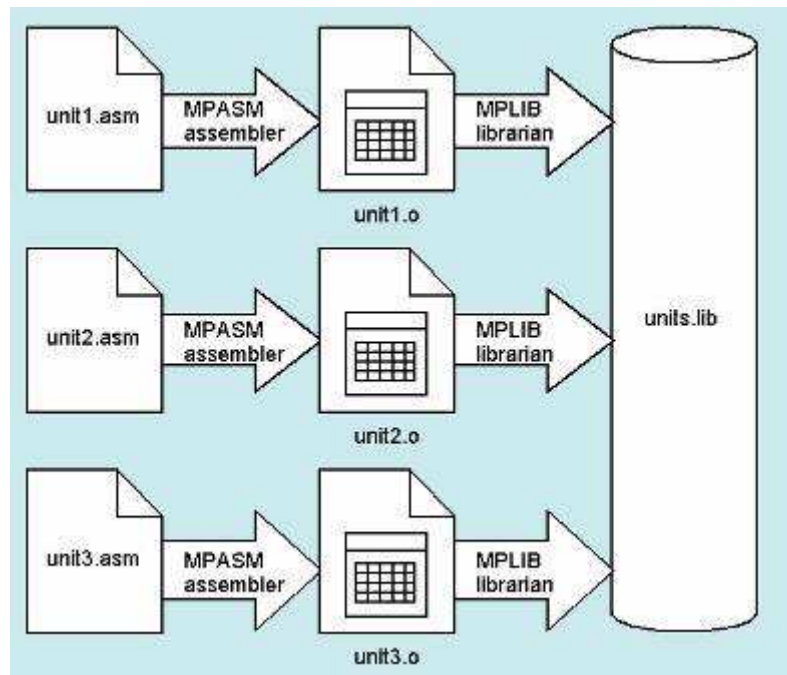


Figure 2.3: Library Build Process

Assembly is a programming language you may use to develop the source code for your application. The source code file may be created using any ASCII text file editor or using MPLAB’s Programmer’s Editor. Your source code should conform to the following basic guidelines. Each line of the source file may contain up to four types of information: labels – “tags” given to locations in source code, mnemonics – short names that are given for each machine instruction, operands – most mnemonics operate on operands, such as registers, labels or numbers: directives – special instructions to the assembler, macros – short cuts for defining commonly used assembly routines, comments – user notations to the code.

The order and position of these are important. For ease of debugging, it is recommended that labels start in column one and mnemonics start in column two or beyond. Operands follow the mnemonic. Comments may follow the operands,

mnemonics or labels, and can start in any column. The maximum column width is 255 characters. Comments can also be on a line by themselves.

White space or a colon must separate the label and the mnemonic, and white space must separate the mnemonic and the operand(s). Multiple operands must be separated by commas. “White space” is one or more spaces or tabs. White space is used to separate pieces of a source line. White space should be used to make your code easier to read. Unless within character constants, any white space means the same as exactly one space. Files associated with MPASM are listed here: source Code (.asm) Default source file extension input to assembler.

Include File (.inc) Include (header) file. Listing File (.lst) Default output extension for listing files generated by assembler. Error File (.err) Output extension from assembler for error files. Hex File Formats (.hex, .hxl, .hxh) Output extension from assembler for hex files. Cross Reference File (.xrf) Output extension from assembler for cross reference files. Symbol and Debug File (.cod) Output extension for the symbol and debug file. For absolute code, this file will be generated by the assembler. For relocatable code, this file and a file will be generated by the MPLINK linker. Object File (.o) Output extension from assembler for object files.

2.2. ISIS Simulation System

Many CAD users dismiss schematic capture as a necessary evil in the process of creating PCB layout but we have always disputed this point of view. With PCB layout now offering automation of both component placement and track routing, getting the design into the computer can often be the most time consuming element of the exercise. And if you use circuit simulation to develop your ideas, you are going to spend even more time working on the schematic.

ISIS has been created with this in mind. It has evolved over twelve years research and development and has been proven by thousands of users worldwide. The

strength of its architecture has allowed us to integrate first conventional graph based simulation and now - with PROTEUS VSM - interactive circuit simulation into the design environment. For the first time ever it is possible to draw a complete circuit for a micro-controller based system and then test it interactively, all from within the same piece of software. Meanwhile, ISIS retains a host of features aimed at the PCB designer, so that the same design can be exported for production with ARES or other PCB layout software.

For the educational user and engineering author, ISIS also excels at producing attractive schematics like you see in the magazines. It provides total control of drawing appearance in terms of line widths, fill styles, colours and fonts. In addition, a system of templates allows you to define a 'house style' and to copy the appearance of one drawing to another.

Other general features include: Runs on Windows 98/Me/2k/XP and later. Automatic wire routing and dot placement/removal. Powerful tools for selecting objects and assigning their properties. Total support for buses including component pins, inter-sheet terminals, module ports and wires. Bill of Materials and Electrical Rules Check reports. Netlist outputs to suit all popular PCB layout tools.

For the 'power user', ISIS incorporates a number of features which aid in the management of large designs. Indeed, a number of our customers have used it to produce designs containing many thousands of components.

Hierarchical design with support for parameterized component values on sub-circuits. Design Global Annotation allowing multiple instances of a sub-circuit to have different component references. Automatic Annotation - the ability to number the components automatically. ASCII Data Import - this facility provides the means to automatically bring component stock codes and costs into ISIS design or library files where they can then be incorporated or even totaled

up in the Bill of Materials report.

Users of ARES, or indeed other PCB software will find some of the following PCB design specific features of interest: Sheet Global Net Properties which allow you to efficiently define a routing strategy for all the nets on a given sheet (e.g. a power supply needing POWER width tracks). Physical terminals which provide the means to have the pins on a connector scattered all over a design. Support for heterogeneous multi-element devices. For example, a relay device can have three elements called RELAY:A, RELAY:B and RELAY:C. RELAY:A is the coil whilst elements B and C are separate contacts. Each element can be placed individually wherever on the design is most convenient.

Support for pin-swap and gate-swap. This includes both the ability to specify legal swaps in the ISIS library parts and the ability to back-annotate changes into a schematic. A visual packaging tool which shows the PCB footprint and its pin numbers alongside the list of pin names for the schematic part. This facilitates easy and error free assignment of pin numbers to pin names. In addition, multiple packagings may be created for a single schematic part.

ISIS provides the development environment for PROTEUS VSM, our revolutionary interactive system level simulator. This product combines mixed mode circuit simulation, micro-processor models and interactive component models to allow the simulation of complete micro-controller based designs. ISIS provides the means to enter the design in the first place, the architecture for real time interactive simulation and a system for managing the source and object code associated with each project. In addition, a number of graph objects can be placed on the schematic to enable conventional time, frequency and swept variable simulation to be performed.

Major features of PROTEUS VSM include: True Mixed Mode simulation based on Berkeley SPICE3F5 with extensions for digital simulation and true mixed mode operation. Support for both interactive and graph based simulation. CPU

Models available for popular microcontrollers such as the PIC and 8051 series.

Interactive peripheral models include LED and LCD displays, a universal matrix keypad, an RS232 terminal and a whole library of switches, pots, lamps, LEDs etc. Virtual Instruments include voltmeters, ammeters, a dual beam oscilloscope and a 24 channel logic analyser. On-screen graphing - the graphs are placed directly on the schematic just like any other object. Graphs can be maximised to a full screen mode for cursor based measurement and so forth. Graph Based Analysis types include transient, frequency, noise, distortion, AC and DC sweeps and fourier transform. An Audio graph allows playback of simulated waveforms. Direct support for analogue component models in SPICE format. Open architecture for 'plug in' component models coded in C++ or other languages. These can be electrical., graphical or a combination of the two.

Digital simulator includes a BASIC-like programming language for modelling and test vector generation. A design created for simulation can also be used to generate a netlist for creating a PCB - there is no need to enter the design a second time.

CHAPTER 3

MICROCONTROLLER ARCHITECTURE

3.1. Memory Organization

There are two memory blocks in the PIC16F84A. These are the program memory and the data memory. Each block has its own bus, so that access to each block can occur during the same oscillator cycle. The data memory can further be broken down into the general purpose RAM and the Special Function Registers (SFRs). The operation of the SFRs that control the “core” are described here. The SFRs used to control the peripheral modules are described in the section discussing each individual peripheral module. The data memory area also contains the data EEPROM memory. This memory is not directly mapped into the data memory, but is indirectly mapped. That is, an indirect address pointer specifies the address of the data EEPROM memory to read/write. The 64 bytes of data EEPROM memory have the address range 0h-3Fh.

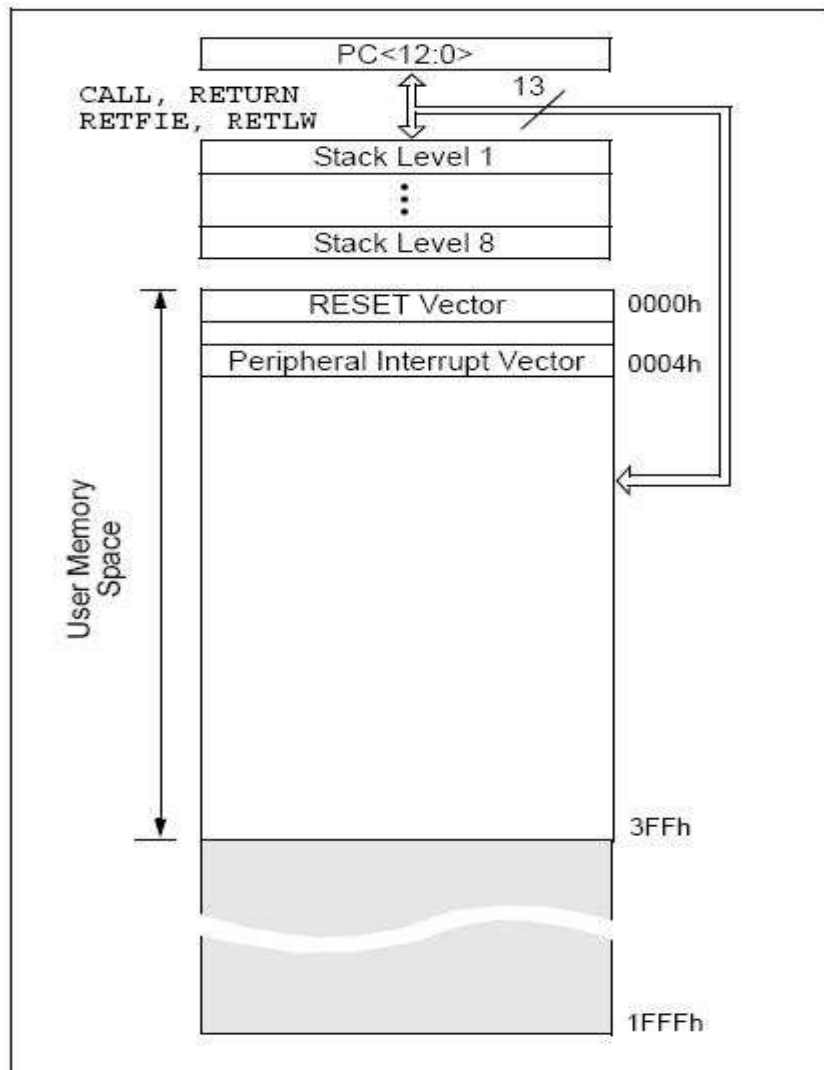


Figure 3.1: Program Memory Map and Stack 16f84

The PIC16F877A devices have a 13-bit program counter capable of addressing an 8K word x 14 bit program memory space. The PIC16F876A/877A devices have 8K words x 14 bits of Flash program memory, while PIC16F873A/874A devices have 4K words x 14 bits. Accessing a location above the physically implemented address will cause a wraparound. The Reset vector is at 0000h and the interrupt vector is at 0004h.

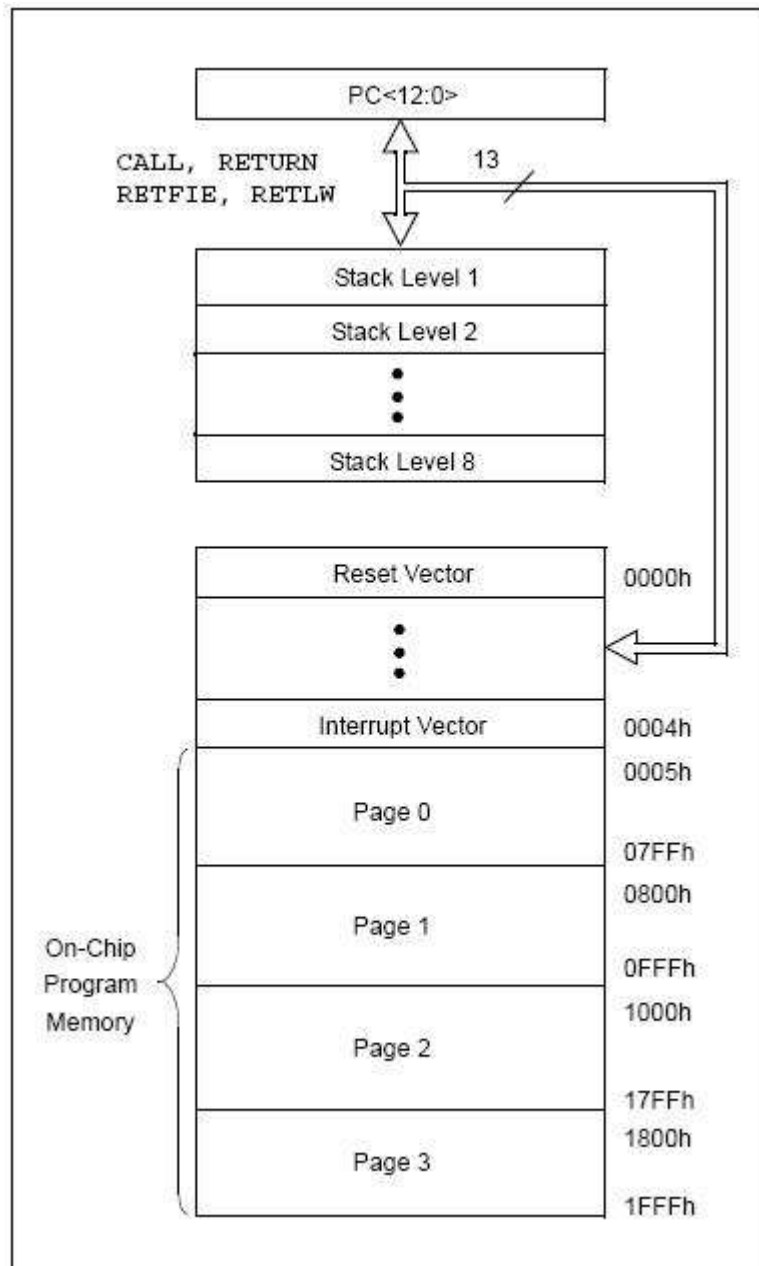


Figure 3.2: Program Memory Map and Stack 16f877

3.2. Data EEPROM Memory

The EEPROM data memory is readable and writable during normal operation (full VDD range). This memory is not directly mapped in the register file space.

Instead it is indirectly addressed through the Special Function Registers. There are four SFRs used to read and write this memory. These registers are: EECON1, EECON2 (not a physically implemented register), EEDATA, EEADR, EEDATA holds the 8-bit data for read/write, and EEADR holds the address of the EEPROM location being accessed. PIC16F84A devices have 64 bytes of data EEPROM with an address range from 0h to 3Fh. The EEPROM data memory allows byte read and write. A byte write automatically erases the location and writes the new data (erase before write). The EEPROM data memory is rated for high erase/write cycles. The write time is controlled by an on-chip timer. The writetime will vary with voltage and temperature as well as from chip to chip. Please refer to AC specifications for exact limits. When the device is code protected, the CPU may continue to read and write the data EEPROM memory.

The device programmer can no longer Access this memory. To read a data memory location, the user must write the address to the EEADR register and then set control bit RD (EECON1<0>). The data is available, in the very next cycle, in the EEDATA register; therefore, it can be read in the next instruction. EEDATA will hold this value until another read or until it is written to by the user (during a write operation).

To write an EEPROM data location, the user must first write the address to the EEADR register and the data to the EEDATA register. Then the user must follow a specific sequence to initiate the write for each byte.

The write will not initiate if the above sequence is not exactly followed (write 55h to EECON2, write AAh to EECON2, then set WR bit) for each byte. We strongly recommend that interrupts be disabled during this code segment.

Additionally, the WREN bit in EECON1 must be set to enable write. This mechanism prevents accidental writes to data EEPROM due to errant (unexpected) code execution (i.e., lost programs). The user should keep the WREN bit clear at all times, except when updating EEPROM. The WREN bit is not cleared by hardware. After a write sequence has been initiated, clearing the WREN bit will not affect this write cycle. The WR bit will be inhibited from being set unless the WREN bit is set. At the completion of the write cycle, the WR bit is cleared in hardware and the EE Write Complete Interrupt Flag bit (EEIF) is set. The user can either enable this interrupt or poll this bit. EEIF must be cleared by software.

Depending on the application, good programming practice may dictate that the value written to the Data EEPROM should be verified (Example 3-3) to the desired value to be written. This should be used in applications where an EEPROM bit will be stressed near the specification limit. Generally, the EEPROM write failure will be a bit which was written as a '0', but reads back as a '1' (due to leakage off the bit).

The data EEPROM and Flash program memory is readable and writable during normal operation (over the full VDD range). This memory is not directly mapped in the register file space. Instead, it is indirectly addressed through the Special Function Registers. There are six SFRs used to read and write this memory: EECON1, EECON2, EEDATA, EEDATH, EEADR, EEADRH.

When interfacing to the data memory block, EEDATA holds the 8-bit data for read/write and EEADR holds the address of the EEPROM location being accessed. These devices have 128 or 256 bytes of data EEPROM (depending on the device), with an address range from 00h to FFh. On devices with 128 bytes, addresses from 80h to FFh are unimplemented and will wraparound to the beginning of data EEPROM memory. When writing to unimplemented locations, the on-chip charge pump will be turned off. When interfacing the program memory block, the EEDATA and EEDATH registers form a two-byte word that

holds the 14-bit data for read/write and the EEADR and EEADRH registers form a two-byte word that holds the 13-bit address of the program memory location being accessed. These devices have 4 or 8K words of program Flash, with an address range from 0000h to 0FFFh for the PIC16F873A/874A and 0000h to 1FFFh for the PIC16F876A/877A. Addresses above the range of the respective device will wraparound to the beginning of program memory. The EEPROM data memory allows single-byte read and write. The Flash program memory allows single-word reads and four-word block writes. Program memory write operations automatically perform an erase-beforewrite on blocks of four words. A byte write in data EEPROM memory automatically erases the location and writes the new data (erase-before-write). The write time is controlled by an on-chip timer. The write/erase voltages are generated by an on-chip charge pump, rated to operate over the voltage range of the device for byte or word operations. When the device is code-protected, the CPU may continue to read and write the data EEPROM memory. Depending on the settings of the write-protect bits, the device may or may not be able to write certain blocks of the program memory; however, reads of the program memory are allowed. When code-protected, the device programmer can no longer access data or program memory; this does NOT inhibit internal reads or writes.

The EEADRH:EEADR register pair can address up to a maximum of 256 bytes of data EEPROM or up to a maximum of 8K words of program EEPROM. When selecting a data address value, only the LSByte of the address is written to the EEADR register. When selecting a program address value, the MSByte of the address is written to the EEADRH register and the LSByte is written to the EEADR register. If the device contains less memory than the full address reach of the address register pair, the Most Significant bits of the registers are not implemented. For example, if the device has 128 bytes of data EEPROM, the Most Significant bit of EEADR is not implemented on Access to data EEPROM.

3.3. I/O Ports

Some pins for these I/O ports are multiplexed with an alternate function for the peripheral features on the device. In general, when a peripheral is enabled, that pin may not be used as a general purpose I/O pin. PORTA is a 5-bit wide, bi-directional port. The corresponding data direction register is TRISA. Setting a TRISA bit (= 1) will make the corresponding PORTA pin an input (i.e., put the corresponding output driver in a hi-impedance mode). Clearing a TRISA bit (= 0) will make the corresponding PORTA pin an output (i.e., put the contents of the output latch on the selected pin).

Reading the PORTA register reads the status of the pins, whereas writing to it will write to the port latch. All write operations are read-modify-write operations. Therefore, a write to a port implies that the port pins are read. This value is modified and then written to the port data latch. Pin RA4 is multiplexed with the Timer0 module clock input to become the RA4/T0CKI pin. The RA4/T0CKI pin is a Schmitt Trigger input and an open drain output. All other RA port pins have TTL input levels and full CMOS output drivers. PORTB is an 8-bit wide, bi-directional port. The corresponding data direction register is TRISB. Setting a TRISB bit (= 1) will make the corresponding PORTB pin an input (i.e., put the corresponding output driver in a Hi-Impedance mode). Clearing a TRISB bit (= 0) will make the corresponding PORTB pin an output (i.e., put the contents of the output latch on the selected pin).

On 16f877, PORTA is a 6-bit wide, bidirectional port. The corresponding data direction register is TRISA. Setting a TRISA bit (= 1) will make the corresponding PORTA pin an input (i.e., put the corresponding output driver in a High-Impedance mode). Clearing a TRISA bit (= 0) will make the corresponding PORTA pin an output (i.e., put the contents of the output latch on the selected pin). Reading the PORTA register reads the status of the pins, whereas writing to it will write to the port latch. All write operations are read-modify-write operations. Therefore, a write to a port implies that the port pins are read, the

value is modified and then written to the port data latch. Pin RA4 is multiplexed with the Timer0 module clock input to become the RA4/T0CKI pin. The RA4/T0CKI pin is a Schmitt Trigger input and an open-drain output. All other PORTA pins have TTL input levels and full CMOS output drivers. Other PORTA pins are multiplexed with analog inputs and the analog VREF input for both the A/D converters and the comparators. The operation of each pin is selected by clearing/setting the appropriate control bits in the ADCON1 and/or CMCON registers.

3.4. Timer Modules

The Timer0 module timer/counter has the following features: 8-bit timer/counter, Readable and writable, Internal or external clock select, Edge select for external clock, 8-bit software programmable prescaler, Interrupt-on-overflow from FFh to 00h.

An 8-bit counter is available as a prescaler for the Timer0 module, or as a postscaler for the Watchdog Timer, respectively (Figure 5-2). For simplicity, this counter is being referred to as “prescaler” throughout this data sheet. Note that there is only one prescaler available which is mutually exclusively shared between the Timer0 module and the Watchdog Timer. Thus, a prescaler assignment for the Timer0 module means that there is no prescaler for the Watchdog Timer, and vice-versa. The prescaler is not readable or writable. The PSA and PS2:PS0 bits (OPTION_REG<3:0>) determine the prescaler assignment and prescale ratio. Clearing bit PSA will assign the prescaler to the Timer0 module. When the prescaler is assigned to the Timer0 module, prescale values of 1:2, 1:4, ..., 1:256 are selectable. Setting bit PSA will assign the prescaler to the Watchdog Timer (WDT). When the prescaler is assigned to the WDT, prescale values of 1:1, 1:2, ..., 1:128 are selectable. When assigned to the Timer0 module, all instructions writing to the TMR0 register (e.g., CLRF 1, MOVWF 1, BSF 1, etc.) will clear the prescaler. When assigned to WDT, a

CLRWDT instruction will clear the prescaler along with the WDT.

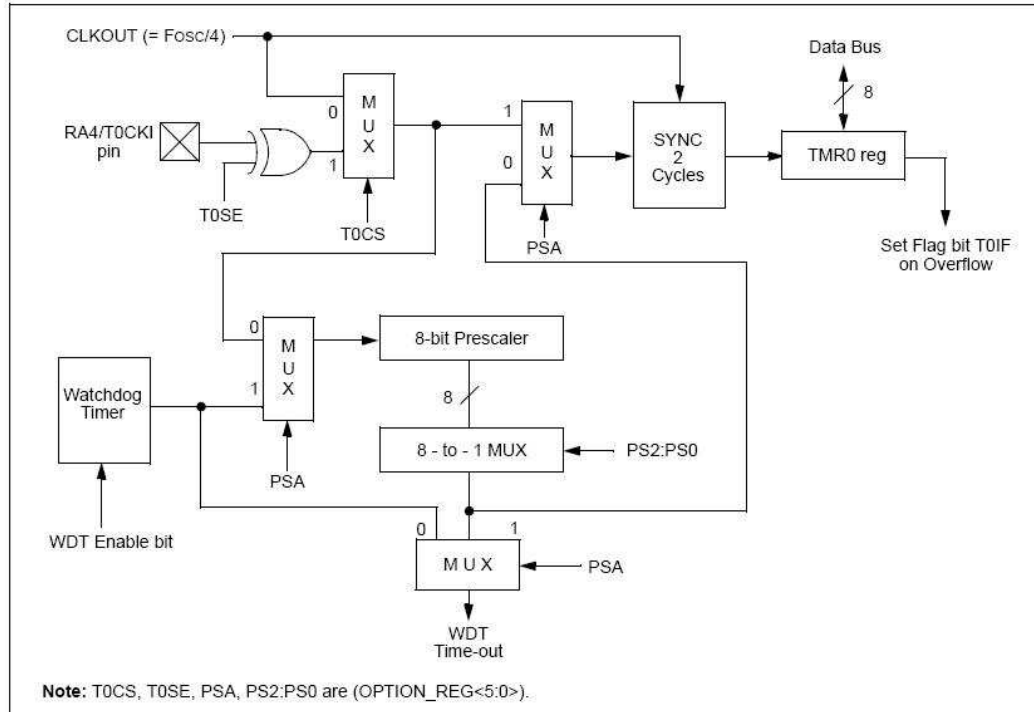


Figure 3.3: Block diagram of the Timer0 / WDT Prescaler

The Timer0 module timer/counter has the following features: 8-bit timer/counter, Readable and writable, 8-bit software programmable prescaler, Internal or external clock select, Interrupt on overflow from FFh to 00h, Edge select for external clock.

Figure 5-1 is a block diagram of the Timer0 module and the prescaler shared with the WDT. Additional information on the Timer0 module is available in the PICmicro® Mid-Range MCU Family Reference Manual (DS33023).

Timer mode is selected by clearing bit T0CS (OPTION_REG<5>). In Timer mode, the Timer0 module will increment every instruction cycle (without

prescaler). If the TMR0 register is written, the increment is inhibited for the following two instruction cycles. The user can work around this by writing an adjusted value to the TMR0 register. Counter mode is selected by setting bit T0CS (OPTION_REG<5>). In Counter mode, Timer0 will increment either on every rising or falling edge of pin RA4/T0CKI. The incrementing edge is determined by the Timer0 Source Edge Select bit, T0SE (OPTION_REG<4>). Clearing bit T0SE selects the rising edge. The prescaler is mutually exclusively shared between the Timer0 module and the Watchdog Timer. The prescaler is not readable or writable.

3.5. Special Features of the CPU

What sets a microcontroller apart from other processors are special circuits to deal with the needs of real time applications. The PIC16F84A has a host of such features intended to maximize system reliability, minimize cost through elimination of external components, provide power saving operating modes and offer code protection. These features are: OSC Selection, RESET, Power-on Reset (POR), Power-up Timer (PWRT), Oscillator Start-up Timer (OST), Interrupts, Watchdog Timer (WDT), SLEEP, Code Protection, ID Locations, In-Circuit Serial Programming™ (ICSP™).

The PIC16F84A has a Watchdog Timer which can be shut-off only through configuration bits. It runs off its own RC oscillator for added reliability. There are two timers that offer necessary delays on power-up. One is the Oscillator Start-up Timer (OST), intended to keep the chip in RESET until the crystal oscillator is stable.

The other is the Power-up Timer (PWRT), which provides a fixed delay of 72 ms (nominal) on power-up only. This design keeps the device in RESET while the power supply stabilizes. With these two timers on-chip, most applications need no external RESET circuitry. SLEEP mode offers a very low current power-down

mode. The user can wake-up from SLEEP through external RESET, Watchdog Timer Time-out or through an interrupt. Several oscillator options are provided to allow the part to fit the application. The RC oscillator option saves system cost while the LP crystal option saves power. A set of configuration bits are used to select the various options. The configuration bits can be programmed (read as '0'), or left unprogrammed (read as '1'), to select various device configurations. These bits are mapped in program memory location 2007h. Address 2007h is beyond the user program memory space and it belongs to the special test/configuration memory space (2000h - 3FFFh). This space can only be accessed during programming.

All PIC16F87XA devices have a host of features intended to maximize system reliability, minimize cost through elimination of external components, provide power saving operating modes and offer code protection. These are: PIC16F87XA devices have a Watchdog Timer which can be shut-off only through configuration bits. It runs off its own RC oscillator for added reliability. There are two timers that offer necessary delays on power-up. One is the Oscillator Start-up Timer (OST), intended to keep the chip in Reset until the crystal oscillator is stable. The other is the Power-up Timer (PWRT), which provides a fixed delay of 72 ms (nominal) on power-up only. It is designed to keep the part in Reset while the power supply stabilizes. With these two timers on-chip, most applications need no external Reset circuitry. Sleep mode is designed to offer a very low current power-down mode. The user can wake-up from Sleep through external Reset, Watchdog Timer wake-up or through an interrupt. Several oscillator options are also made available to allow the part to fit the application. The RC oscillator option saves system cost while the LP crystal option saves power. A set of configuration bits is used to select various options.

REGISTER 14-1: CONFIGURATION WORD (ADDRESS 2007h)⁽¹⁾

R/P-1	U-0	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	U-0	U-0	R/P-1	R/P-1	R/P-1	R/P-1
CP	—	DEBUG	WRT1	WRT0	CPD	LVP	BOREN	—	—	PWRTEN	WDTEN	Fosc1	Fosc0
bit 13													bit 0

bit 13 **CP:** Flash Program Memory Code Protection bit

1 = Code protection off

0 = All program memory code-protected

bit 12 **Unimplemented:** Read as '1'

bit 11 **DEBUG:** In-Circuit Debugger Mode bit

1 = In-Circuit Debugger disabled, RB6 and RB7 are general purpose I/O pins

0 = In-Circuit Debugger enabled, RB6 and RB7 are dedicated to the debugger

bit 10-9 **WRT1:WRT0** Flash Program Memory Write Enable bits

11 = Write protection off; all program memory may be written to by EECON control

10 = 0000h to 00FFh write-protected; 0100h to 1FFFh may be written to by EECON control

01 = 0000h to 07FFh write-protected; 0800h to 1FFFh may be written to by EECON control

00 = 0000h to 0FFFh write-protected; 1000h to 1FFFh may be written to by EECON control

CHAPTER 4

PARALLEL PROCESSING BY A MICROCONTROLLER BASED- SYSTEM

4.1. Ready State

Digital circuit is designed using ISIS case tool. On the our circuit, there are microcontrollers, registers, keypad, LCD display, capacitors. Each microcontroller has own program. There are four nodes which have got same program. Master runs different from nodes. The simultaneous usage of more than one microcontroller's CPU are used to execute same program. Ideally, parallel processing makes a program, running fast because there are more CPUs running it. In practice, it is often difficult to divide a program in such a way that separate CPUs can execute different portions without interfering with each other. It is possible to perform parallel processing by connecting the CPU in a network. However, this type of parallel processing requires very sophisticated software called distributed processing software.

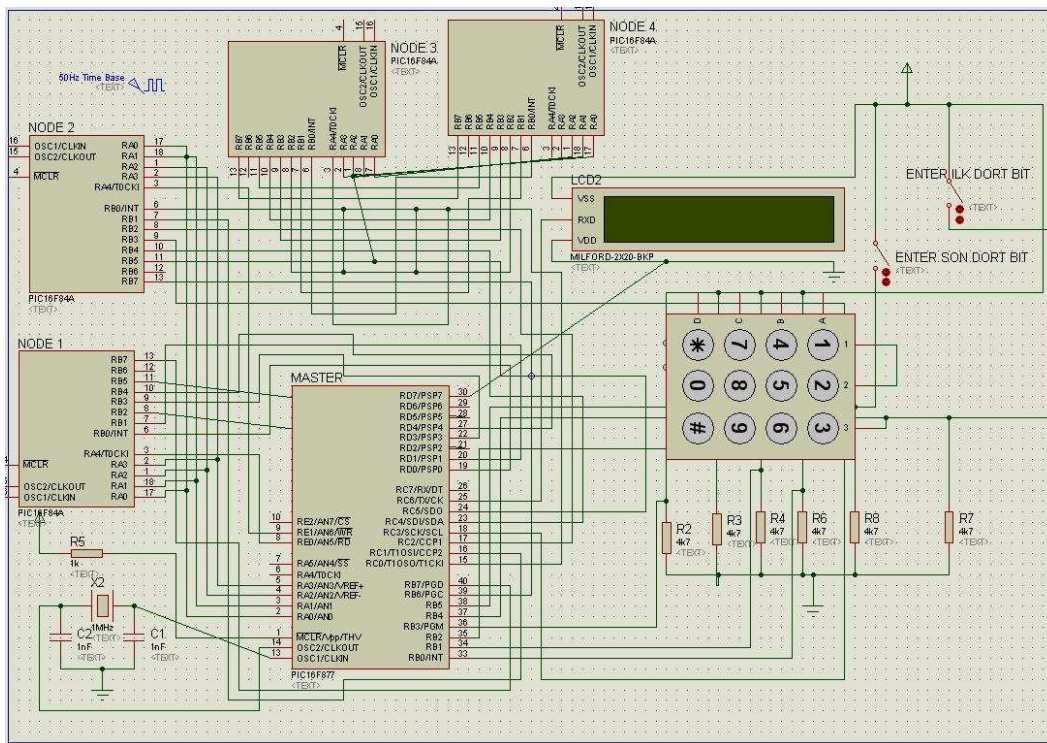


Figure 4.1: System Design

if ISIS simulation tool runs on initial state, the programs which is running on master and node waits a trigger from master. On this state, master reads data from ports when user enters any key on key pad. One register keeps data from ports for mission and in that case a pin reads data from start button. Start button initially is low. After user pushes the first four bits button, register record the data from port and then register waits for second four bits from the user. If start button is high register data is considered for data from user . User enters 8 bits data. On the blow, there are assembler code for initial state.

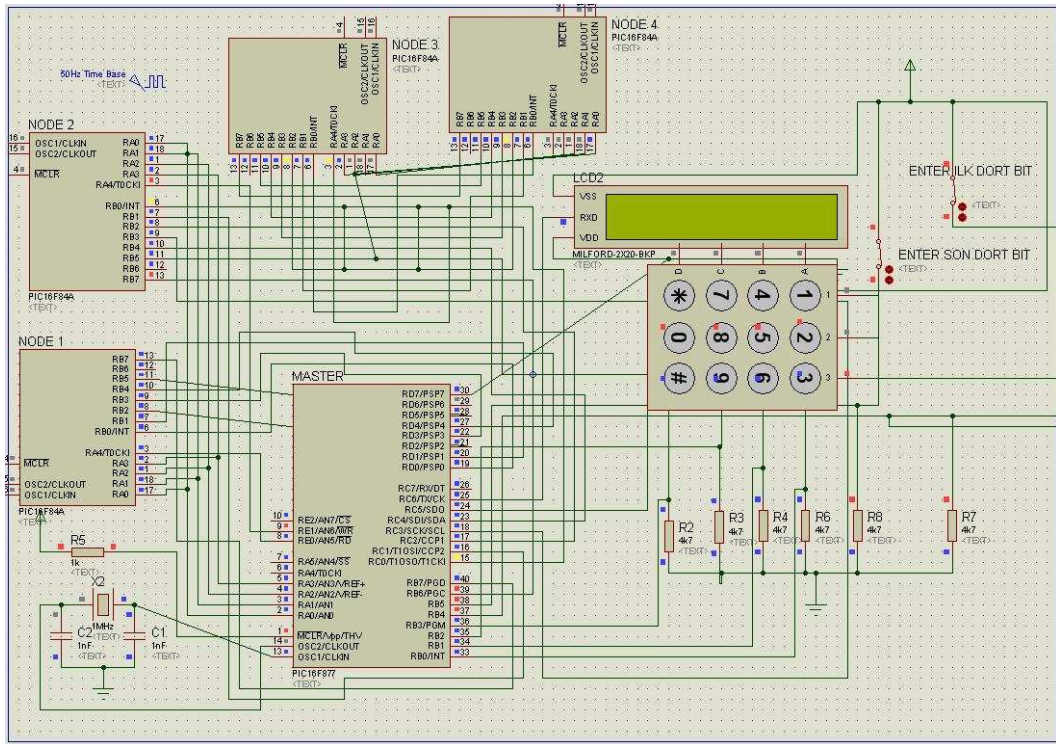


Figure 4.2: System is Ready State

4.2. Running State

When the register has the password, it specifies the nodes which one waits data from master. Program counter determines number of nodes. After this mode, programs divide register data which is password. If the number of nodes are two, program divides by two. In this way, program divides by number of nodes. The program, running on master send signal to node for activation of node. After this sign signal, node waits data from master. Assembler subroutine of master is shown on the below.

```
LIST p=16F877
INCLUDE "P16F877.INC"
```

```

ranl    equ H'0A'
ranm    equ H'0B'
ranh    equ H'0C'
beat    equ H'0D'
reg1    equ H'0E'
reg2    equ H'0D'
reg_FirstDortBit    equ H'0F'
reg_SecondDortBit    equ H'1F'
reg_ThirdDortBit    equ H'2F'
reg_ExecTimeFirst    equ H'FF'
reg_ExecTimeSecond    equ H'FF'

```

```

cblock 0x20
char,cmd,lc1,lc2;
endc

```

```

; org 0
goto G_INITIAL
; org 0x04 ; void interrupt(void)
; goto C_inthlr

```

```

G_INITIAL
clrwdt ; watchdog timer 1 temizle
movlw b'10110111'; assign prescaler, internal clock frekansı
option
bsf STATUS,RP0

```

```

movlw H'00'
movwf TRISA
movlw H'FF'

```



```

movwf TRISB
movlw H'3F'
movwf TRISC
movlw H'FF'
movwf TRISD
movlw H'00'
movwf TRISE
bcf STATUS,RP0
clrf PORTA
clrf PORTB
clrf PORTC
clrf PORTD
clrf PORTE
clrf reg1
clrf reg2
;clrf reg_ExeTimeFirst
movlw H'FF'
movwf reg_ExeTimeFirst

```

G_ENTER_NUMBER

```

call C_ENTERKONTROL ;Control enter
btfss reg2,0 ;is any bit set?
call C_SETILKDORTBIT ;get first four bits.
btfsc reg2,0
call C_SETIKINCIDORTBIT ;get second four bits
movlw H'03'
andwf reg2,0
sublw H'03'
btfsc STATUS,Z ;are 2nd cour bits set?
goto G_NUMBER_ENTERED ;if ok,crack password
goto G_ENTER_NUMBER

```

C_DISPLAY_ENTERED_NUMBER

```
btfs reg1,7
call C_ZERO
btfs reg1,7
call C_ONE
btfs reg1,6
call C_ZERO
btfs reg1,6
call C_ONE
btfs reg1,5
call C_ZERO
btfs reg1,5
call C_ONE
btfs reg1,4
call C_ZERO
btfs reg1,4
call C_ONE
btfs reg1,3
call C_ZERO
btfs reg1,3
call C_ONE
btfs reg1,2
call C_ZERO
btfs reg1,2
call C_ONE
btfs reg1,1
call C_ZERO
btfs reg1,1
call C_ONE
btfs reg1,0
```

```
call C_ZERO
btfsc reg1,0
call C_ONE
return
```

C_DISPLAY_EXECUTION_TIME

```
btfss reg_ExeTimeFirst,7
call C_ZERO
btfsc reg_ExeTimeFirst,7
call C_ONE
btfss reg_ExeTimeFirst,6
call C_ZERO
btfsc reg_ExeTimeFirst,6
call C_ONE
btfss reg_ExeTimeFirst,5
call C_ZERO
btfsc reg_ExeTimeFirst,5
call C_ONE
btfss reg_ExeTimeFirst,4
call C_ZERO
btfsc reg_ExeTimeFirst,4
call C_ONE
btfss reg_ExeTimeFirst,3
call C_ZERO
btfsc reg_ExeTimeFirst,3
call C_ONE
btfss reg_ExeTimeFirst,2
call C_ZERO
btfsc reg_ExeTimeFirst,2
call C_ONE
btfss reg_ExeTimeFirst,1
```

```
call C_ZERO
btfsc reg_ExeTimeFirst,1
call C_ONE
btfss reg_ExeTimeFirst,0
call C_ZERO
btfsc reg_ExeTimeFirst,0
call C_ONE
return
```

C_DISPLAY_RECEIVED_MSJ_NODE2

```
movlw 'N'
call G_putc
movlw 'o'
call G_putc
movlw 'd'
call G_putc
movlw 'e'
call G_putc
movlw '2'
call G_putc
movlw ''
call G_putc
movlw 'i'
call G_putc
movlw 's'
call G_putc
movlw ''
call G_putc
movlw 'r'
call G_putc
movlw 'u'
```

```

call G_putc
movlw 'n'
call G_putc
movlw 'n'
call G_putc
movlw 'i'
call G_putc
movlw 'g'
call G_putc
call C_WAIT
return

```

```

.....

```

```

G_DISPLAY_NODE2_RUNNING

```

```

clr
movwf PORTB
bcf STATUS,RP0
bsf RCSTA,SPEN
bsf RCSTA,CREN
bsf STATUS,RP0
movlw H'00'
clr
movwf TRISA
movwf TRISB
movlw H'19'
movwf SPBRG
movlw H'A4'
movwf TXSTA

```

```

movlw 100

```

```

call G_delay

call C_DISPLAY_RECEIVED_MSJ_NODE2

movlw H'0C'
call G_wrcmd
movlw H'0D'
call G_wrcmd
call G_loop
return

.....
G_DISPLAY_EXECUTION_TIME
clr
movlw H'00'
movwf PORTB
bcf STATUS,RP0
bsf RCSTA,SPEN
bsf RCSTA,CREN
bsf STATUS,RP0
clr
movlw H'00'
movwf TRISA
movwf TRISB
movlw H'19'
movwf SPBRG
movlw H'A4'
movwf TXSTA

movlw 100
call G_delay

```

```
movlw 'E'
call G_putc
call C_WAIT
movlw 'X'
call G_putc
call C_WAIT
movlw 'E'
call G_putc
call C_WAIT
movlw 'C'
call G_putc
call C_WAIT
movlw 'U'
call G_putc
call C_WAIT
movlw 'T'
call G_putc
call C_WAIT
movlw 'T'
call G_putc
call C_WAIT
movlw 'O'
call G_putc
call C_WAIT
movlw 'N'
call G_putc
call C_WAIT
movlw ':'
call G_putc
call C_WAIT
```

```
call C_DISPLAY_EXECUTION_TIME
```

```
movlw 'm'  
call G_putc  
call C_WAIT  
movlw 's'  
call G_putc  
call C_WAIT
```

```
movlw H'0C'  
call G_wrcmd  
movlw H'0D'  
call G_wrcmd  
call G_loop  
return
```

```
.....
```

```
        G_loop  
call G_getc  
movwf char  
sublw 0d  
btfsc STATUS,Z  
goto G_cls  
movf char,w  
sublw 08  
btfsc STATUS,Z  
goto G_bspace  
movf char,W
```



```
call G_putc
goto G_loop
```

```
    G_cls
    movlw H'01'
call G_wrcmd
goto G_loop
```

```
    G_bspace
movlw H'10'
call G_wrcmd
goto G_loop
```

```
G_hang clrwdt
goto G_hang
```

```
G_wrcmd movwf cmd
movlw 0xFE
call G_putc
movf cmd,W
goto G_putc
```

```
G_getc bcf STATUS,RP0
```

```
C_getc1 btfss PIR1,RCIF
goto C_getc1
movf RCREG,W ; read char
bcf PIR1,RCIF ; clear interrupt flag
return
```

```

G_putc bcf STATUS,RP0
movwf TXREG
bsf STATUS,RP0
movf TXSTA,W
C_putc1 btfss TXSTA,1
goto C_putc1
bcf STATUS,RP0
return

```

```

C_WAIT:
movlw 5
call G_delay
return

```

```

G_delay movwf lc2
G_sw2 movlw H'FF'
movwf lc1
C_sw3 nop
decfsz lc1,f
goto C_sw3
decfsz lc2,f
goto G_sw2
return
C_inthlr retfie

```

```

C_ENTERKONTROL
movlw H'10'
andwf PORTB,0
sublw H'10'
btfsc STATUS,Z ;are first four bits set?

```

```

call C_ILKDORTBITGIRILDI
movlw H'20'
andwf PORTB,0
sublw H'20'
btfsc STATUS,Z ;are second four bits set?
call C_SONDORTBITGIRILDI
movlw H'30'
andwf PORTB,0
sublw H'30'
btfsc STATUS,Z
call C_TUMBITLERGIRILDI
;movlw H'C0'
;andwf PORTB,0
;sublw H'C0'
;btfsc STATUS,Z ;is sytem reset?
;goto INITIAL
return
C_ILKDORTBITGIRILDI
bsf reg2,0
return
C_SONDORTBITGIRILDI
bsf reg2,1
return
C_TUMBITLERGIRILDI
movlw H'03'
movwf reg2
return
C_SETILKDORTBIT
movlw H'01'
andwf PORTB,0
sublw H'01'

```

```
btfsc STATUS,Z
call C_SIFIR ;zero bit is set
```

```
movlw H'02'
andwf PORTB,0
sublw H'02'
btfsc STATUS,Z
call C_BIR ;first bit is set
```

```
movlw H'04'
andwf PORTB,0
sublw H'04'
btfsc STATUS,Z
call C_IKI ;second pin is set
```

```
movlw H'08'
andwf PORTB,0
sublw H'08'
btfsc STATUS,Z
call C_UC ;Third pin is set
return
```

C_SETIKINCIDORTBIT

```
movlw H'01'
andwf PORTB,0
sublw H'01'
btfsc STATUS,Z
call C_DORT ;fourth bit is set
movlw H'02'
andwf PORTB,0
```

```

sublw H'02'
btfsc STATUS,Z
call C_BES           ;fifth bit is set

movlw H'04'
andwf PORTB,0
sublw H'04'
btfsc STATUS,Z
call C_ALTI         ;sixth pin is set
movlw H'08'
andwf PORTB,0
sublw H'08'
btfsc STATUS,Z
call C_YEDI         ;seventh pin is set
return

```

C_SIFIR

```
bsf reg1,0
```

```
return
```

C_BIR

```
bsf reg1,1
```

```
return
```

C_IKI

```
bsf reg1,2
```

```
return
```

C_UC

```
bsf reg1,3
```

```
return
```

C_DORT

```
bsf reg1,4
```

return

C_BES

bsf reg1,5

return

C_ALTI

bsf reg1,6

return

C_YEDI

bsf reg1,7

return

C_ZERO:

movlw '0'

call G_putc

call C_WAIT

return

C_ONE:

movlw '1'

call G_putc

call C_WAIT

return

C_HAS_EXECUTION_TIME_RECEIVE_N3

btfss PORTB,6

goto C_HAS_EXECUTION_TIME_RECEIVE_N3

movlw H'3F'

andwf PORTC,0

movwf reg_ExeTimeSecond

return

C_HAS_EXECUTION_TIME_RECEIVE_N2

```

btfss PORTB,7
goto C_HAS_EXECUTION_TIME_RECEIVE_N2
movlw      H'FF'
andwf PORTD,0
movwf reg_ExeTimeFirst
return

```

C_HAS_DATA_RECEIVED_FROM_NODE2 ;FROM NODE 2

```

btfss PORTB,7
goto C_HAS_DATA_RECEIVED_FROM_NODE2 ;Has Node 2 received data
form Node 1
return

```

C_HAS_DATA_RECEIVED_FROM_NODE3 ;FROM NODE 3

```

btfss PORTB,6
goto C_HAS_DATA_RECEIVED_FROM_NODE3 ;Has Node 3 received data
form Node 1
return

```

C_SEND_DATA_NODE2

```

movf reg_FirstDortBit,0 ; w = reg_FirstDortBit
movwf PORTA             ; Data send node 2
return

```

C_SEND_DATA_NODE3

```

movf reg_SecondDortBit,0 ; w = reg_SecondDortBit
movwf PORTA             ; Data send node 3
return

```

G_NUMBER_ENTERED

```

movlw      H'0F'

```

```

andwf reg1,0          ; w = first 4 bit of reg1
movwf reg_FirstDortBit ; reg_FirstDortBit = w(first 4 bit of reg1)

swapf reg1,0         ; w = reg1(first 4 bit <-> second 4 bit)
andlw H'0F'          ; w = second 4 bit of reg1
movwf reg_SecondDortBit ; reg_SecondDortBit = w(second 4 bit of reg1)

movlw H'01'
movwf PORTE          ; Node 2 is set
call C_SEND_DATA_NODE2 ;Send data to node 2
call C_HAS_DATA_RECEIVED_FROM_NODE2
movlw H'FE'
andwf PORTE,0
movwf PORTE          ;Node 2 is reset
movlw H'00'
movwf PORTA          ;Data 0

movlw 2000
call G_delay
movlw H'02'
movwf PORTE          ; Node 3 is set
call C_SEND_DATA_NODE3
call C_HAS_DATA_RECEIVED_FROM_NODE3 ;Has Node 3 received data
form Node 1
movlw H'FD'
andwf PORTE,0
movwf PORTE          ;Node 3 is reset
movlw H'00'
movwf PORTA          ;Data 0

```



```

bsf PORTE,0 ;Node 2 set edildi.
movlw 2000
call G_delay
call C_HAS_EXECUTION_TIME_RECEIVE_N2 ;node1 <- execution time
bcf PORTE,0 ;Node 2 reset
edildi.

;goto G_DISPLAY_EXECUTION_TIME

bsf PORTE,1 ;Node 3 set edildi.
movlw 2000
call G_delay
call C_HAS_EXECUTION_TIME_RECEIVE_N3 ;node1 <- execution time
bcf PORTE,0 ;Node 3 reset

goto G_DISPLAY_EXECUTION_TIME

end

```

Assembler subroutine of nodes is shown on the below.

```

.....;NODE;.....

```

```

LIST p=16F84A

```

```

INCLUDE "P16F84A.INC"

```

```

lc1    equ 0x0a
lc2    equ 0x0b

msec   equ   0x0c           ; tens of milliseconds
sec     equ   0x0d           ; seconds
min     equ   0x0e           ; minutes
hour    equ   0x0f           ; hours
timef   equ   0x10           ; register for time flags
save    equ   0x11           ; save for ACCU

reg_number          equ 0x12
reg_timeMSec1       equ 0x13
reg_findOutNumber   equ 0x14
reg_timeSec1        equ 0x15
reg_timeSec2        equ 0x16
reg_ExecutionTime   equ 0x17

; constants
msf    equ   0x00           ; millisecond flag
sf     equ   0x01           ; second flag
mf     equ   0x02           ; minute flag
hf     equ   0x03           ; hour flag
df     equ   0x04           ; day flag

MSD    equ   0x4b           ; crystal divider (75)
PSD    equ   0x05           ; millisecond divider
XD     equ   0x4b           ; crystal divider (75)
minf   equ   0x02           ; minute flag

org    0

```

```

goto  _main
org   0x04      ; void interrupt(void)

_interrupt      ; {
movwf save      ; save(ACCU);
bcf   INTCON,T0IF      ; INTCON,T0IF = 0;
incf  msec,F      ; msec++;
bsf   timef,msf     ; msf = 1;
movf  msec,W      ; ACCU = msec;
sublw XD        ; if ((ACCU-XD) != 0)
btfss STATUS,Z   ; return;
retfie          ; else {
clrf  msec       ; msec = 0;
bsf   timef,sf   ; msf = 1;
incf  sec,F      ; sec++;
movf  sec,W      ; ACCU = sec;
sublw 0x3c       ; if ((ACCU-60) != 0)
btfss STATUS,Z   ; return;
retfie          ; else {
clrf  sec        ; sec = 0;
bsf   timef,minf ; sf = 1;
incf  min,F      ; min++;
movf  min,W      ; ACCU = min;
sublw 0x3c       ; if ((ACCU-60) != 0)
btfss STATUS,Z   ; return;
retfie          ; else {
clrf  min        ; min = 0;
bsf   timef,hf   ; hf = 1;
incf  hour,F     ; hour++;
movf  hour,W     ; ACCU = hour;

```

```

sublw 0x18      ; if ((ACCU-24) != 0)
btfss STATUS,Z ; return;
retfie         ; else {
clrf  hour     ; hour = 0;
bsf  timef,df  ; df = 1;
movf  save,W   ; }}}} restore(ACCU);
retfie         ; }

```

_initialize

```
bsf  STATUS,RP0; bank 1
```

```
movlw H'00'
```

```
movwf TRISB
```

```
movlw H'1F'
```

```
movwf TRISA
```

```
movlw 0x7d      ; RBPU=off, INTEDG=off, TOCS=osc, PSA=TMR0
```

```
addlw PSD      ; PSD = b'101' [64]
```

```
OPTION         ;
```

```
bcf  STATUS,RP0 ; bank 0
```

```
;movlw 0xa0      ; enable TMR0 interrupt
```

```
movlw b'10100000' ;
```

```
movwf INTCON    ;
```

```
clrf  msec      ; msec = 0;
```

```
clrf  sec       ; sec = 0;
```

```
clrf  min       ; min = 0;
```

```
clrf  hour      ; hour = 0;
```

```
clrf  timef     ; all flags off;
```

```
clrf  PORTB
```

```
clrf  PORTA
```

```
clrf  reg_number
```

```

clrf reg_timeSec1
movlw H'00'
movwf reg_timeSec2
clrf reg_timeMSec1
clrf reg_findOutNumber
return
C_DELAY
G_delay
movlw H'7F'
movwf lc2
G_sw2 movlw H'FF'
movwf lc1
C_sw3
nop
decfsz lc1,f
goto C_sw3
decfsz lc2,f
goto G_sw2
return
ONE
movlw H'01'
movwf reg_ExecutionTime
return
THREE
movlw H'03'
movwf reg_ExecutionTime
return
SEVEN
movlw H'07'
movwf reg_ExecutionTime
return

```

EIGHT

movlw H'08'

movwf reg_ExecutionTime

return

TEN

movlw H'0A'

movwf reg_ExecutionTime

return

ELEVEN

movlw H'0B'

movwf reg_ExecutionTime

return

THIRTEEN

movlw H'0D'

movwf reg_ExecutionTime

return

FOURTEEN

movlw H'0E'

movwf reg_ExecutionTime

return

ONE_7

movlw H'17'

movwf reg_ExecutionTime

return

TWO_A

movlw H'2A'

movwf reg_ExecutionTime

return

TWO_E

movlw H'2E'

movwf reg_ExecutionTime

```

return
THREE_4
movlw H'34'
movwf reg_ExecutionTime
return
THREE_9
movlw H'39'
movwf reg_ExecutionTime
return
THREE_B
movlw H'3B'
movwf reg_ExecutionTime
return
THREE_C
movlw H'3C'
movwf reg_ExecutionTime
return

C_NUMBER_SELECT
ZERO
movlw H'00'
andwf reg_findOutNumber,0
sublw H'00'
btfsc STATUS,Z
call ONE

movlw H'01'
andwf reg_findOutNumber,0
sublw H'01'
btfsc STATUS,Z
call THREE

```

```
movlw H'02'  
andwf reg_findOutNumber,0  
sublw H'02'  
btfsc STATUS,Z  
call SEVEN
```

```
movlw H'03'  
andwf reg_findOutNumber,0  
sublw H'03'  
btfsc STATUS,Z  
call SEVEN
```

```
movlw H'04'  
andwf reg_findOutNumber,0  
sublw H'04'  
btfsc STATUS,Z  
call EIGHT
```

```
movlw H'05'  
andwf reg_findOutNumber,0  
sublw H'05'  
btfsc STATUS,Z  
call TEN
```

```
movlw H'06'  
andwf reg_findOutNumber,0  
sublw H'06'  
btfsc STATUS,Z  
call ELEVEN
```



```
movlw H'07'  
andwf reg_findOutNumber,0  
sublw H'07'  
btfsc STATUS,Z  
call THIRTEEN
```

```
movlw H'08'  
andwf reg_findOutNumber,0  
sublw H'08'  
btfsc STATUS,Z  
call FOURTEEN
```

```
movlw H'09'  
andwf reg_findOutNumber,0  
sublw H'09'  
btfsc STATUS,Z  
call ONE_7
```

```
movlw H'0A'  
andwf reg_findOutNumber,0  
sublw H'0A'  
btfsc STATUS,Z  
call TWO_A
```

```
movlw H'0B'  
andwf reg_findOutNumber,0  
sublw H'0B'  
btfsc STATUS,Z  
call TWO_E
```

```
movlw H'0C'
```

```
andwf reg_findOutNumber,0
sublw H'0C'
btfsc STATUS,Z
call THREE_4
```

```
movlw H'0D'
andwf reg_findOutNumber,0
sublw H'0D'
btfsc STATUS,Z
call THREE_9
```

```
movlw H'0E'
andwf reg_findOutNumber,0
sublw H'0E'
btfsc STATUS,Z
call THREE_B
```

```
movlw H'0F'
andwf reg_findOutNumber,0
sublw H'0F'
btfsc STATUS,Z
call THREE_C
return
```

C_RECV_DATA_FROM_NODE1

```
movf PORTA,0
andlw H'0F' ; w = received 4 Bit data
movwf reg_number ; reg_number = w
return
```

C_GET_TIME1

```

movf  sec,0           ; w = sec
movwf reg_timeSec1   ; reg_time1 = w
return

```

C_GET_TIME2

```

movf  sec,0           ; w = sec
movwf reg_timeSec2   ; reg_time2 = w
return

```

_main

```

call  _initialize
C_IS_NODE_INIT
btfss PORTA,4        ;Is Node2 set
goto  C_IS_NODE_INIT
movf  sec,0           ; w = sec
movwf reg_timeSec1   ; reg_time1 = w
call  C_RECV_DATA_FROM_NODE1

```

```

movlw H'80'
movwf PORTB

```

```

movlw H'00'
movwf reg_findOutNumber
G_FIND_OUT_NUMBER
movf  reg_findOutNumber,0
incf  reg_findOutNumber,1
subwf reg_number,0
btfss STATUS,Z
goto  G_FIND_OUT_NUMBER
decf  reg_findOutNumber,1
movf  sec,0           ; w = sec

```

```

movwf reg_timeSec2          ; reg_time2 = w
movf  sec,0                 ; w = sec
movwf reg_ExecutionTime ; reg_time2 = w
call  C_NUMBER_SELECT
;movf  reg_timeSec1,0      ; w = reg_timeSec1
;subwf reg_timeSec2,0     ; w = reg_timeSec2 - w(reg_timeSec1)AMAN
;movwf reg_ExecutionTime ; reg_ExecutionTime = w

```

G_IS_NODE_RESET

```

btfsc PORTA,4              ;Is Node2 reset
goto  G_IS_NODE_RESET
movlw H'00'
movwf PORTB
call  C_DELAY

```

C_IS_NODE_SET

```

btfss PORTA,4             ;Is Node2 set
goto  C_IS_NODE_SET

```

G_SEND_EXECUTION_TIME

```

bsf      reg_ExecutionTime,7
movf  reg_ExecutionTime,0
movwf PORTB

```

C_IS_NODE_RESET

```

btfsc PORTA,4              ;Is Node2 reset
goto  G_SEND_EXECUTION_TIME
movlw H'00'
movwf PORTB
goto  C_IS_NODE_RESET

```

END

When data arrived at node, program on running node write data to own register.

Program starts estimation and on the same time, program interrupt is activated. When it starts to increase program counter by hexadecimal number 01, program check the data. If the generation of this data is same of sending data, interrupt is triggered. After this mode, using interrupt mission, execution time is estimated. After execution time is known, node send a sign signal to master for declaration. This declaration is about execution time on blocked state.

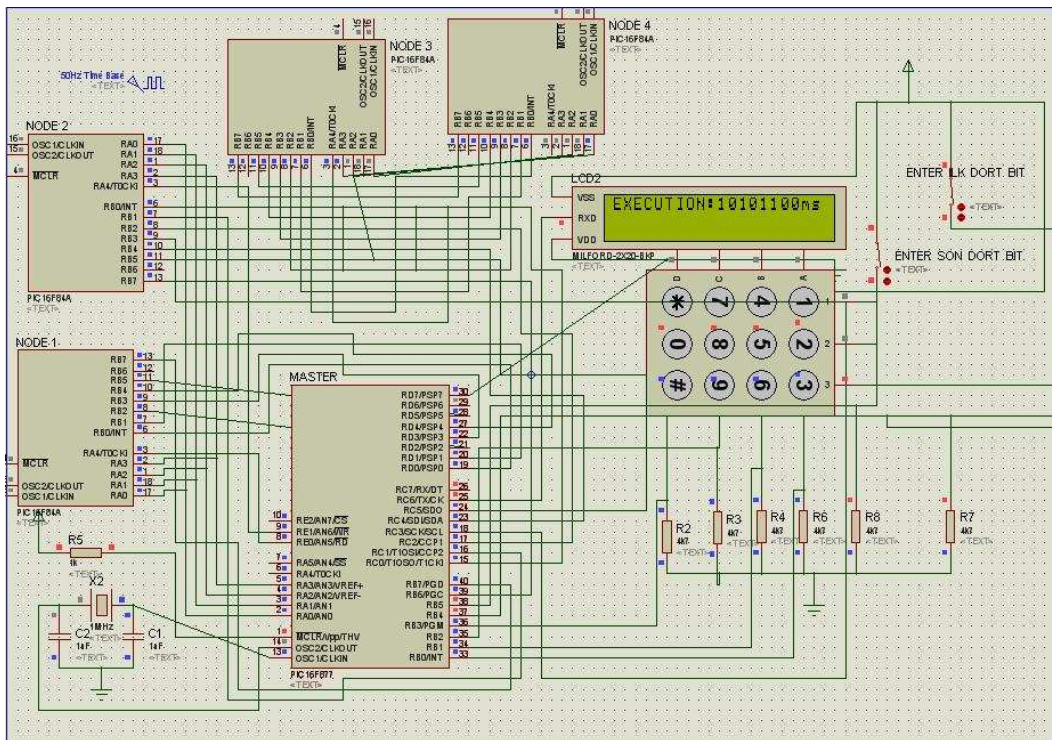


Figure 4.3: System is Running State

4.3. Calculating Speedup

In this thesis, we simulate the system with other input values. We enter values for different numbers of nodes as shown in the table below. The simulation enables us to calculate speedup and illustrate the speedup graph.



Input	A Node	Two Nodes	Four Nodes	Eighth Nodes (approximately)	two nodes	four nodes	eight nodes
Hex 01	7 msec	7-8 msec	7 msec	7 msec	0.934 speedup	1 speedup	1 speedup
Hex 02	9 msec	10 msec	12 msec	14 msec	0.9 speedup	0.75 speedup	0.645 speedup
Hex 04	13 msec	14 msec	13 msec	14 msec	0.93 speedup	1 speedup	0.93 speedup
Hex 08	21 msec	22 msec	12 msec	14 msec	0.95 speedup	1.75 speedup	1.5 speedup
Hex 10	37 msec	8 msec	11 msec	14 msec	4.625 speedup	3.364 speedup	2.645 speedup
Hex 20	69 msec	10 msec	12 msec	14 msec	6.9 speedup	5.75 speedup	4.9 speedup
Hex 40	133 msec	14 msec	12 msec	14 msec	9.5 speedup	11 speedup	9.5 speedup

Figure 4.4: Calculating Speedup using different number of nodes and an input value

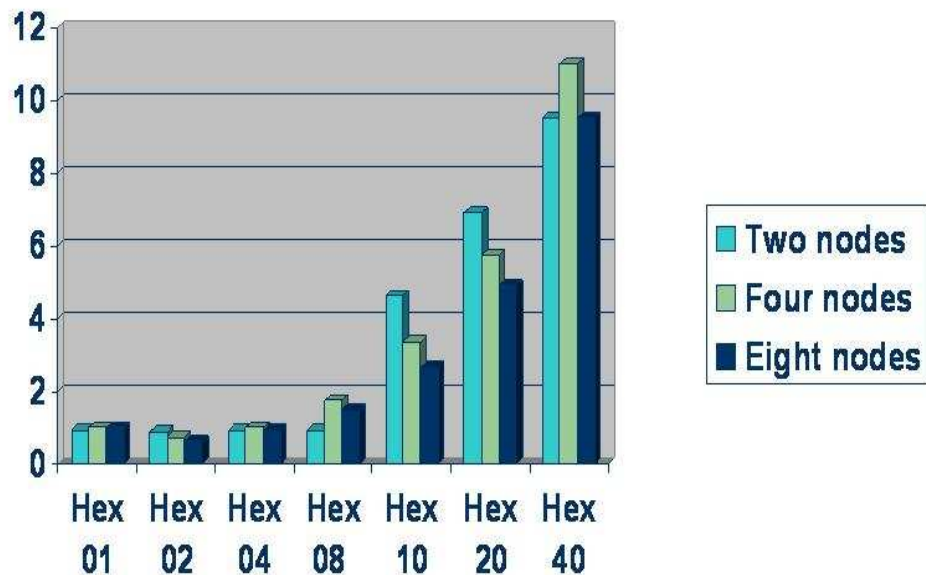


Figure 4.5: Speedup graph

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, we have derived a lower bound for the efficiency of executing a task graph. This corresponds to an iterative algorithm executing parallel on multiple microcontroller (this refers to CPU also) and requiring only partial synchronization. We have found out the most suitable number of microcontroller for our mission.

We try to calculate execution time using different number of CPU and thus system is saturated when there are four CPUs. We enter different key at different number of nodes. Consequently, our system is suitable on four nodes for most efficiency password cracker system.

REFERENCES

- [1] **Thompson, Washington D.C. (1967)** , G.M. Amdahl “ *Validity of theSingle Processor Approach to achieving Large Scale Computing Capabilities* “AFIPS conf. Proc., Thompson, Washington D.C., pp.483-485.
- [2] **F. Baccelli and A.M. Makowski (1989)**, “*Queueing Models for Systems with sychronization Constraints*”, Proceeding of the IEEE, 77(1), pp.138-161.
- [3] **Asanovic, Krste et al. (December 18, 2006)**, “*The Landscape of Parallel Computing Research: A View from Berkeley*”. University of California, Berkeley.
- [4] **Michael J. Flynn, Jones and Bartlett Publishers (1995)**, “*Computer Architecture: Pipelined and Parallel Processor Design*”, 1st edition.
- [5] **Lars Bengtsson, Kenneth Nilsson, Bertil Svensson (May 1994)**, “*a High - Performance Embedded Massively Parallel Processing System*”,Proceedings of MPC94: 1'st EUROMICRO International Conference on Massively Parallel Computing Systems, Ischia, Italy.
- [6] **Hua Bei, Tang Xinan (May 2006)**, “*High-performance IPv6 Forwarding Algorithm for Multi-core and Multi-threaded Network Processors*”, professor Hua Bei and USTC schoolfellow Dr. Tang Xinan issued on ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.