Module 2

Embedded Processors and Memory

Lesson 10

Embedded Processors - I

In this lesson the student will learn the following

Architecture of an Embedded Processor
The Architectural Overview of Intel MCS 96 family of Microcontrollers

Pre-requisite

Digital Electronics

10.1 Introduction

It is generally difficult to draw a clear-cut boundary between the class of microcontrollers and general purpose microprocessors. Distinctions can be made or assumed on the following grounds.

- Microcontrollers are generally associated with the embedded applications
- Microprocessors are associated with the desktop computers
- Microcontrollers will have simpler memory hierarchy i.e. the RAM and ROM may exist on the same chip and generally the cache memory will be absent.
- The power consumption and temperature rise of microcontroller is restricted because of the constraints on the physical dimensions.
- 8-bit and 16-bit microcontrollers are very popular with a simpler design as compared to large bit-length (32-bit, 64-bit) complex general purpose processors.

However, recently, the market for 32-bit embedded processors has been growing.
Further the issues such as power consumption, cost, and integrated peripherals differentiate a desktop CPU from an embedded processor. Other important features include the interrupt response time, the amount of on-chip RAM or ROM, and the number of parallel ports. The desktop world values processing power, whereas an embedded microprocessor must do the job for a particular application at the lowest possible cost.

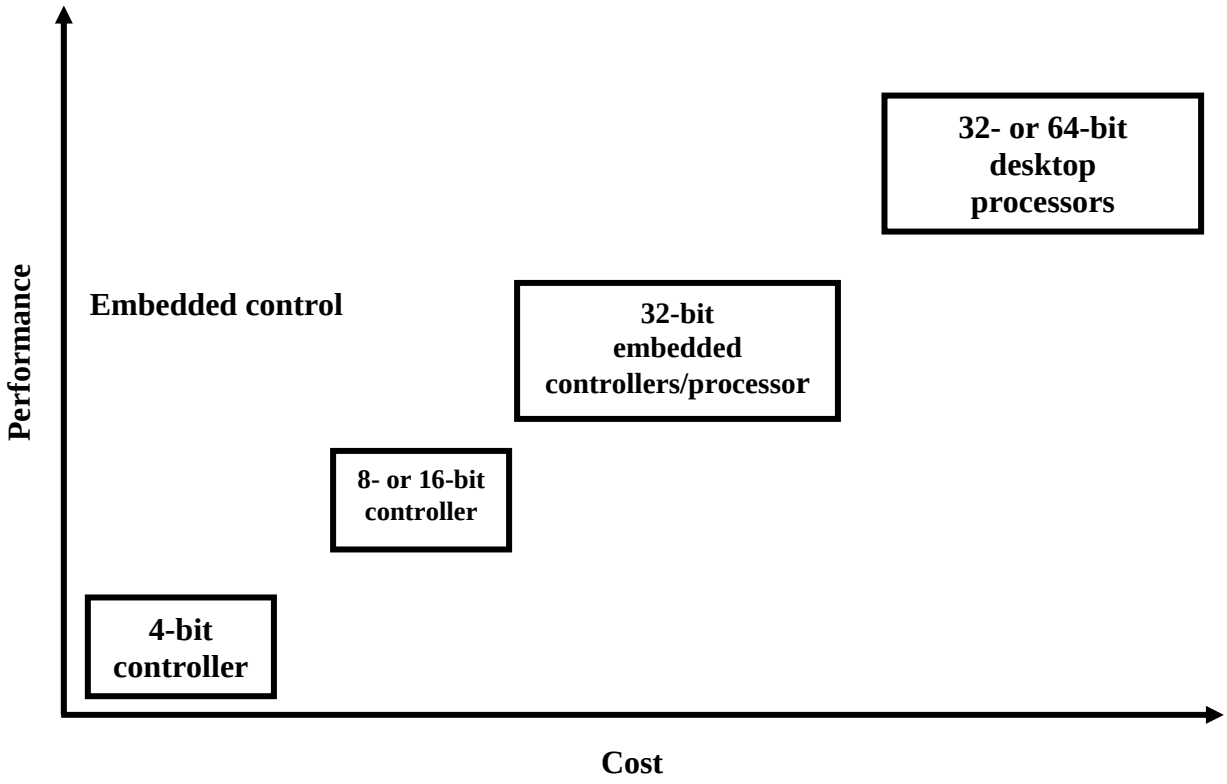

Fig. 10.1 The Performance vs Cost regions

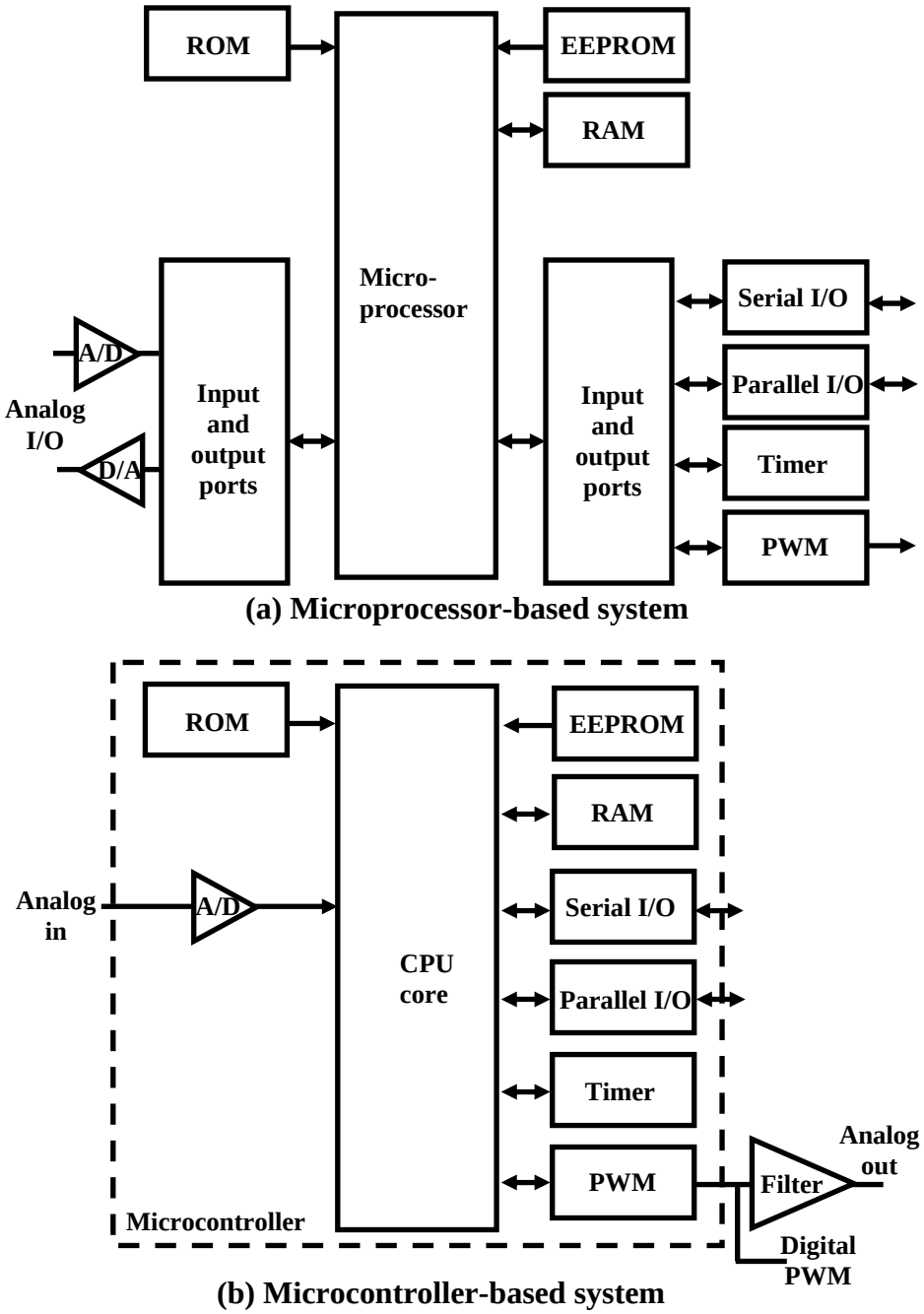

Fig. 10.2 Microprocessor versus microcontroller

Fig. 10.1 shows the performance cost plot of the available microprocessors. Naturally the more is the performance the more is the cost. The embedded controllers occupy the lower left hand corner of the plot.

Fig.10.2 shows the architectural difference between two systems with a general purpose microprocessor and a microcontroller. The hardware requirement in the former system is more than that of later. Separate chips or circuits for serial interface, parallel interface, memory and AD-DA converters are necessary On the other hand the functionality, flexibility and the complexity of information handling is more in case of the former.

10.2 The Architecture of a Typical Microcontroller

A typical microcontroller chip from the Intel 80X96 family is discussed in the following paragraphs.

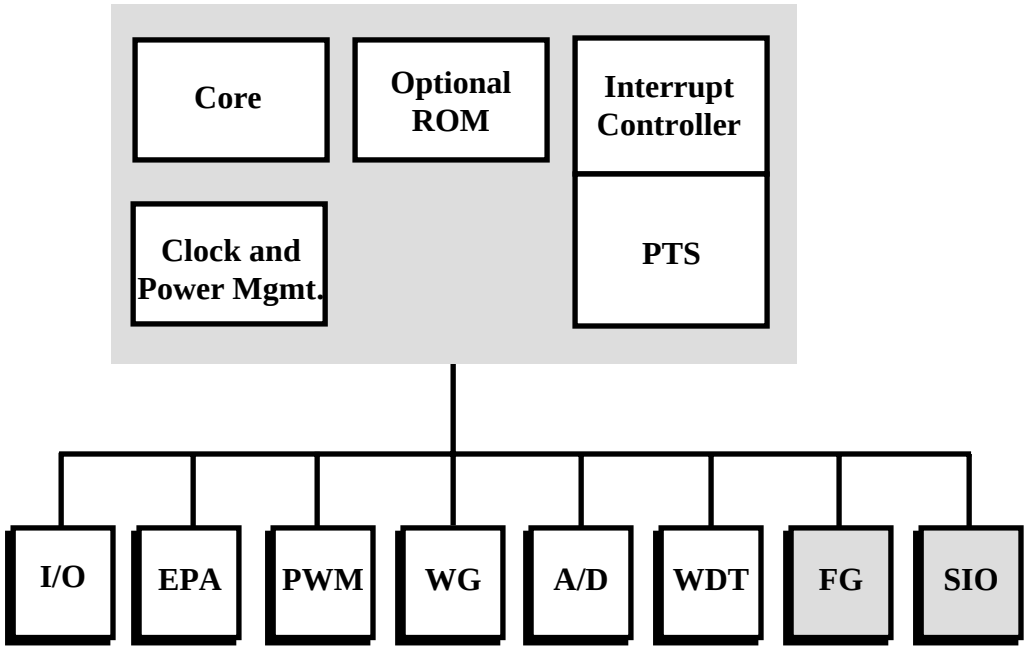

Fig. 10.3 The Architectural Block diagram of Intel 8XC196 Microcontroller

PTS: Peripheral Transaction Server; I/O: Input/Output Interface; EPA: Event Processor Array;

PWM: Pulse Width Modulated Outputs; WG: Waveform Generator; A/D- Analog to Digital Converter;

FG: Frequency Generator; SIO: Serial Input/Output Port

Fig. 10.3 shows the functional block diagram of the microcontroller. The core of the microcontroller consists of the central processing unit (CPU) and memory controller. The CPU contains the register file and the register arithmetic-logic unit (RALU). A 16-bit internal bus connects the CPU to both the memory controller and the interrupt controller. An extension of this bus connects the CPU to the internal peripheral modules. An 8-bit internal bus transfers instruction bytes from the memory controller to the instruction register in the RALU.

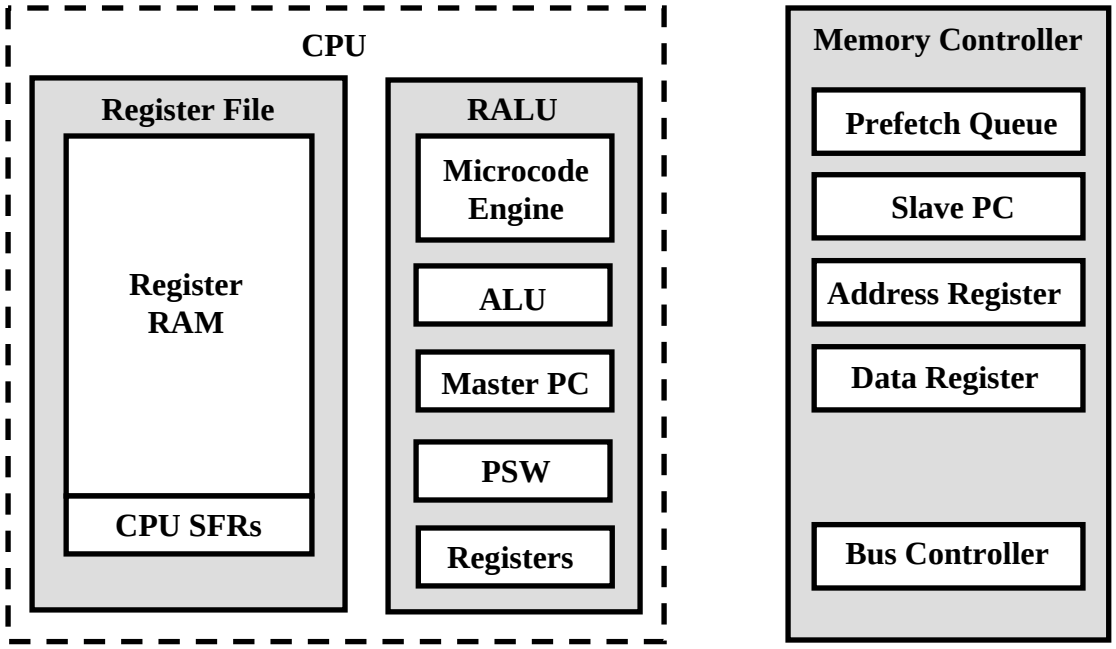

Fig. 10.4 The Architectural Block diagram of the core

CPU: Central Processing Unit; RALU: Register Arithmetic Logic Unit; ALU: Arithmetic Logic Unit;

Master PC: Master Program Counter; PSW: Processor Status Word; SFR: Special Function Registers

CPU Control

The CPU is controlled by the microcode engine, which instructs the RALU to perform operations using bytes, words, or double-words from either the 256-byte lower register file or through a window that directly accesses the upper register file. Windowing is a technique that maps blocks of the upper register file into a window in the lower register file. CPU instructions move from the 4-byte prefetch queue in the memory controller into the RALU's instruction register. The microcode engine decodes the instructions and then generates the sequence of events that cause desired functions to occur.

Register File

The register file is divided into an upper and a lower file. In the lower register file, the lowest 24 bytes are allocated to the CPU's special-function registers (SFRs) and the stack pointer, while the remainder is available as general-purpose register RAM. The upper register file contains only general-purpose register RAM. The register RAM can be accessed as bytes, words, or double words. The RALU accesses the upper and lower register files differently. The lower register file is always directly accessible with direct addressing. The upper register file is accessible with direct addressing only when windowing is enabled.

Register Arithmetic-logic Unit (RALU)

The RALU contains the microcode engine, the 16-bit arithmetic logic unit (ALU), the master program counter (PC), the processor status word (PSW), and several registers. The registers in the RALU are the instruction register, a constants register, a bit-select register, a loop counter, and three temporary registers (the upper-word, lower-word, and second-operand registers). The PSW contains one bit (PSW.1) that globally enables or disables servicing of all maskable interrupts, one bit (PSW.2) that enables or disables the peripheral transaction server (PTS), and six Boolean flags that reflect the state of your program. All registers, except the 3-bit bit-select register and the 6-bit loop counter, are either 16 or 17 bits (16 bits plus a sign extension). Some of these registers can reduce the ALU's workload by performing simple operations.
The RALU uses the upper- and lower-word registers together for the 32-bit instructions and as temporary registers for many instructions. These registers have their own shift logic and are used for operations that require logical shifts, including normalize, multiply, and divide operations. The six-bit loop counter counts repetitive shifts. The second-operand register stores the second operand for two-operand instructions, including the multiplier during multiply operations and the divisor during divide operations. During subtraction operations, the output of this register is complemented before it is moved into the ALU. The RALU speeds up calculations by storing constants (e.g., 0, 1, and 2) in the constants register so that they are readily available when complementing, incrementing, or decrementing bytes or words. In addition, the constants register generates single-bit masks, based on the bit-select register, for bit-test instructions.

Code Execution

The RALU performs most calculations for the microcontroller, but it does not use an *accumulator*. Instead it operates directly on the lower register file, which essentially provides 256 accumulators. Because data does not flow through a single accumulator, the microcontroller's code executes faster and more efficiently.

Instruction Format

These microcontrollers combine general-purpose registers with a three-operand instruction format. This format allows a single instruction to specify two source registers and a separate destination register. For example, the following instruction multiplies two 16-bit variables and stores the 32-bit result in a third variable.

```
MUL  RESULT, FACTOR_1, FACTOR_2     ;multiply FACTOR_1 and FACTOR_2
                                    ;and store answer in RESULT
                                    ;(RESULT)←(FACTOR_1 × FACTOR_2)
```

Memory Interface Unit

The RALU communicates with all memory, except the register file and peripheral SFRs, through the memory controller. The memory controller contains the prefetch queue, the slave program counter (slave PC), address and data registers, and the bus controller. The bus controller drives the memory bus, which consists of an internal memory bus and the external address/data bus. The bus controller receives memory-access requests from either the RALU or the prefetch queue; queue requests always have priority.

When the bus controller receives a request from the queue, it fetches the code from the address contained in the slave PC. The slave PC increases execution speed because the next instruction byte is available immediately and the processor need not wait for the master PC to send the address to the memory controller. If a jump interrupt, call, or return changes the address sequence, the master PC loads the new address into the slave PC, then the CPU flushes the queue and continues processing.

Interrupt Service

The interrupt-handling system has two main components: the programmable interrupt controller and the peripheral transaction server (PTS). The programmable interrupt controller has a hardware priority scheme that can be modified by the software. Interrupts that go through the interrupt controller are serviced by interrupt service routines those are provided by you. The peripheral transaction server (PTS) which is a microcoded hardware interrupt-processor provides efficient interrupt handling.

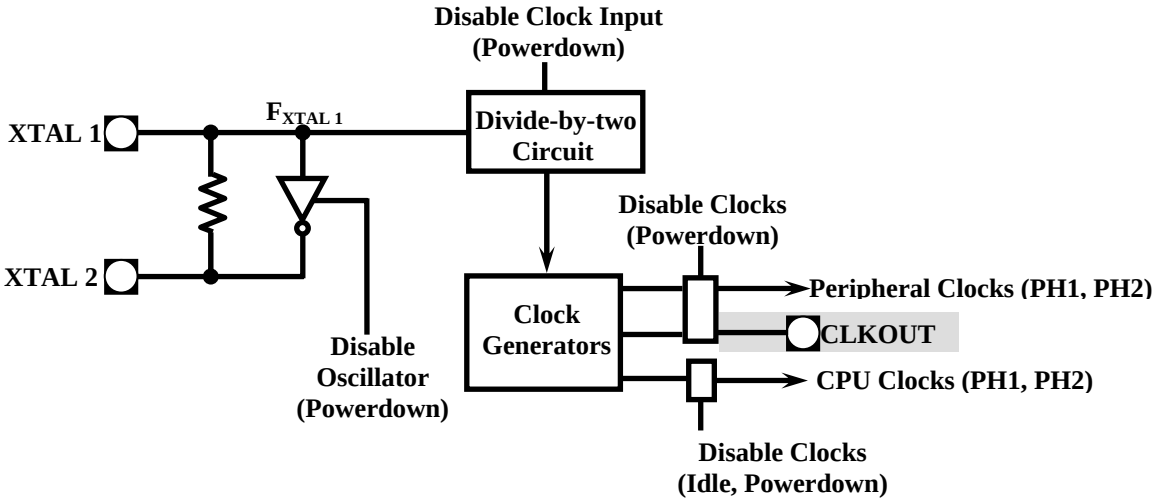

Fig. 10.5 The clock circuitry

Internal Timing

The clock circuitry (Fig. 10.5) receives an input clock signal on XTAL1 provided by an external crystal or oscillator and divides the frequency by two. The clock generators accept the divided input frequency from the divide-by-two circuit and produce two non-overlapping internal timing signals, Phase 1(PH1) and Phase 2 (PH2). These signals are active when high.

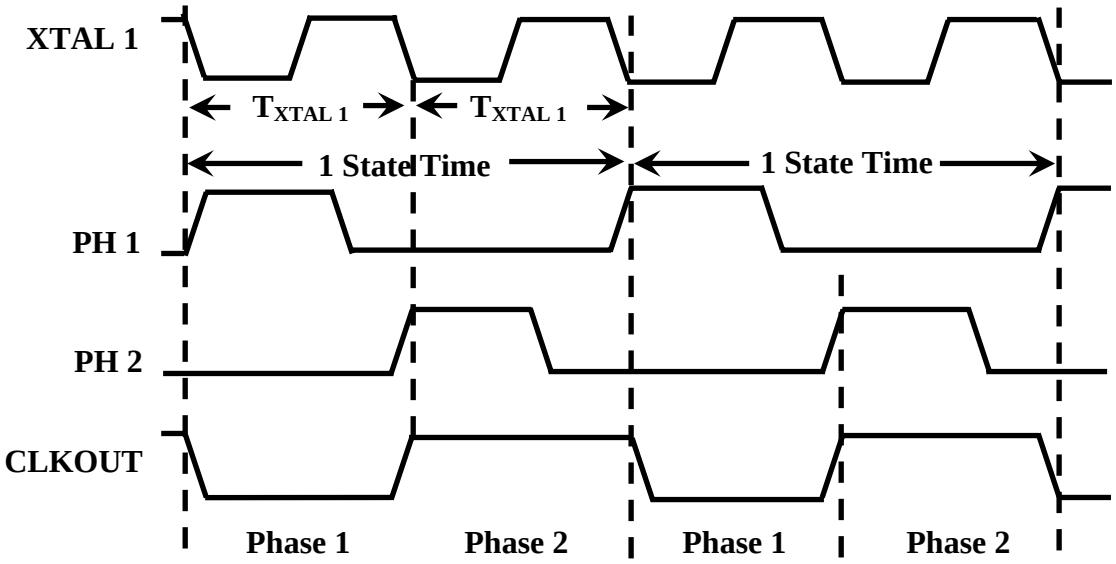

Fig. 10.6 The internal clock phases

The rising edges of PH1 and PH2 generate the internal CLKOUT signal (Fig. 10.6). The clock circuitry routes separate internal clock signals to the CPU and the peripherals to provide flexibility in power management. Because of the complex logic in the clock circuitry, the signal on the CLKOUT pin is a delayed version of the internal CLKOUT signal. This delay varies with temperature and voltage.

I/O Ports

Individual I/O port pins are multiplexed to serve as standard I/O or to carry special function signals associated with an on-chip peripheral or an off-chip component. If a particular special-function signal is not used in an application, the associated pin can be individually configured to serve as a standard I/O pin. Ports 3 and 4 are exceptions; they are controlled at the port level. When the bus controller needs to use the address/data bus, it takes control of the ports. When the address/data bus is idle, you can use the ports for I/O. Port 0 is an input-only port that is also the analog input for the A/D converter. For more details the reader is requested to see the data manual at
www.intel.com/design/mcs96/manuals/27218103.pdf.

Serial I/O (SIO) Port

The microcontroller has a two-channel serial I/O port that shares pins with ports 1 and 2. Some versions of this microcontroller may not have any. The serial I/O (SIO) port is an asynchronous/synchronous port that includes a universal asynchronous receiver and transmitter (UART). The UART has two synchronous modes (modes 0 and 4) and three asynchronous modes (modes 1, 2, and 3) for both transmission and reception. The asynchronous modes are full duplex, meaning that they can transmit and receive data simultaneously. The receiver is buffered, so the reception of a second byte can begin before the first byte is read. The transmitter is also buffered, allowing continuous transmissions. The SIO port has two channels (channels 0 and 1) with identical signals and registers.

Event Processor Array (EPA) and Timer/Counters

The event processor array (EPA) performs high-speed input and output functions associated with its timer/counters. In the input mode, the EPA monitors an input for signal transitions. When an event occurs, the EPA records the timer value associated with it. This is called a *capture* event. In the output mode, the EPA monitors a timer until its value matches that of a stored time value. When a match occurs, the EPA triggers an output event, which can set, clear, or toggle an output pin.
This is called a *compare* event. Both capture and compare events can initiate interrupts, which can be serviced by either the interrupt controller or the PTS. Timer 1 and timer 2 are both 16-bit up/down timer/counters that can be clocked internally or externally. Each timer/counter is called a *timer* if it is clocked internally and a *counter* if it is clocked externally.

Pulse-width Modulator (PWM)

The output waveform from each PWM channel is a variable duty-cycle pulse. Several types of electric motor control applications require a PWM waveform for most efficient operation. When filtered, the PWM waveform produces a DC level that can change in 256 steps by varying the duty cycle. The number of steps per PWM period is also programmable (8 bits).

Frequency Generator

Some microcontrollers of this class has this frequency generator. This peripheral produces a waveform with a fixed duty cycle (50%) and a programmable frequency (ranging from 4 kHz to 1 MHz with a 16 MHz input clock).

Waveform Generator

A waveform generator simplifies the task of generating synchronized, pulse-width modulated (PWM) outputs. This waveform generator is optimized for motion control applications such as driving 3-phase AC induction motors, 3-phase DC brushless motors, or 4-phase stepping motors. The waveform generator can produce three independent pairs of complementary PWM outputs, which share a common carrier period, dead time, and operating mode. Once it is initialized, the waveform generator operates without CPU intervention unless you need to change a duty cycle.

Analog-to-digital Converter

The analog-to-digital (A/D) converter converts an analog input voltage to a digital equivalent. Resolution is either 8 or 10 bits; sample and convert times are programmable. Conversions can be performed on the analog ground and reference voltage, and the results can be used to calculate gain and zero-offset errors. The internal zero-offset compensation circuit enables automatic zero offset adjustment. The A/D also has a threshold-detection mode, which can be used to generate an interrupt when a programmable threshold voltage is crossed in either direction. The A/D scan mode of the PTS facilitates automated A/D conversions and result storage.

Watchdog Timer

The watchdog timer is a 16-bit internal timer that resets the microcontroller if the software fails to operate properly.

Special Operating Modes

In addition to the normal execution mode, the microcontroller operates in several special-purpose modes. Idle and power-down modes conserve power when the microcontroller is inactive. On circuit emulation (ONCE) mode electrically isolates the microcontroller from the system, and several other modes provide programming options for nonvolatile memory.

Reducing Power Consumption

In idle mode, the CPU stops executing instructions, but the peripheral clocks remain active. Power consumption drops to about 40% of normal execution mode consumption. Either a hardware reset or any enabled interrupt source will bring the microcontroller out of idle mode. In power-down mode, all internal clocks are frozen at logic state zero and the internal oscillator is shut off. The register file and most peripherals retain their data if V_{CC} is maintained. Power consumption drops into the μW range.

Testing the Printed Circuit Board

The on-circuit emulation (ONCE) mode electrically isolates the microcontroller from the system. By invoking the ONCE mode, you can test the printed circuit board while the microcontroller is soldered onto the board.

Programming the Nonvolatile Memory

The microcontrollers that have internal OTPROM provide several programming options:

- Slave programming allows a master EPROM programmer to program and verify one or more slave microcontrollers. Programming vendors and Intel distributors typically use this mode to program a large number of microcontrollers with a customer's code and data.
- Auto programming allows an microcontroller to program itself with code and data located in an external memory device. Customers typically use this low-cost method to program a small number of microcontrollers after development and testing are complete.
- Run-time programming allows you to program individual nonvolatile memory locations during normal code execution, under complete software control. Customers typically use this mode to download a small amount of information to the microcontroller after the rest of the array has been programmed. For example, you might use run-time programming to
- download a unique identification number to a security device.
- ROM dump mode allows you to dump the contents of the microcontroller's nonvolatile memory to a tester or to a memory device (such as flash memory or RAM).

10.3 Conclusion

This lesson discussed about the architecture of a typical high performance microcontrollers.
The next lesson shall discuss the signals of a typical microcontroller from the Intel MCS96 family.

10.4 Questions and Answers

1. What do you mean by the Microcode Engine?

Ans: This is where the instructions which breaks down to smaller micro-instructions are executed.
Microprogramming was one of the key breakthroughs that allowed system architects to implement complex instructions in hardware. To understand what microprogramming is, it helps to first consider the alternative: direct execution. With direct execution, the machine fetches an instruction from memory and feeds it into a hardwired control unit. This control unit takes the instruction as its input and activates some circuitry that carries out the task. For instance, if the machine fetches a floating-point ADD and feeds it to the control unit, there's a circuit somewhere in there that kicks in and directs the execution units to make sure that all of the shifting, adding, and normalization gets done. Direct execution is actually pretty much what you'd expect to go on inside a computer if you didn't know about microcoding.

The main advantage of direct execution is that it's fast. There's no extra abstraction or translation going on; the machine is just decoding and executing the instructions right in hardware. The problem with it is that it can take up quite a bit of space. Think about it. If every instruction has to have some circuitry that executes it, then the more instructions you have, the more space the control unit will take up. This problem is compounded if some of the instructions are big and complex, and take a lot of work to execute. So directly executing instructions for a CISC machine just wasn't feasible with the limited transistor resources of the day.

With microprogramming, it's almost like there's a mini-CPU on the CPU. The control unit is a **microcode engine** that executes microcode instructions. The CPU designer uses these microinstructions to write microprograms, which are stored in a special control memory. When a normal program instruction is fetched from memory and fed into the microcode engine, the microcode engine executes the proper microcode subroutine. This subroutine tells the various functional units what to do and how to do it.

As you can probably guess, in the beginning microcode was a pretty slow way to do things. The ROM used for control memory was about 10 times faster than magnetic core-based main memory, so the microcode engine could stay far enough ahead to offer decent performance. As microcode technology evolved, however, it got faster and faster. (The microcode engines on current CPUs are about 95% as fast as direct execution) Since microcode technology was getting better and better, it made more and more sense to just move functionality from (slower and more expensive) software to (faster and cheaper) hardware. So ISA instruction counts grew, and program instruction counts shrank.

As microprograms got bigger and bigger to accommodate the growing instructions sets, however, some serious problems started to emerge. To keep performance up, microcode had to be highly optimized with no inefficiencies, and it had to be extremely compact in order to keep memory costs down. And since microcode programs were so large now, it became

much harder to test and debug the code. As a result, the microcode that shipped with machines was often buggy and had to be patched numerous times out in the field. It was the difficulties involved with using microcode for control that spurred Patterson and others began to question whether implementing all of these complex, elaborate instructions in microcode was really the best use of limited transistor resources.

2. What is the function of the Watch Dog Timer?

Ans: A fail-safe mechanism that intervenes if a system stops functioning. A hardware timer that is periodically reset by software. If the software crashes or hangs, the watchdog timer will expire, and the entire system will be reset automatically.
The Watch Dog Unit contains a Watch Dog Timer.

A watchdog timer (WDT) is a device or electronic card that performs a specific operation after a certain period of time if something goes wrong with an electronic system and the system does not recover on its own.

A common problem is for a machine or operating system to lock up if two parts or programs conflict, or, in an operating system, if memory management trouble occurs. In some cases, the system will eventually recover on its own, but this may take an unknown and perhaps extended length of time. A watchdog timer can be programmed to perform a warm boot (restarting the system) after a certain number of seconds during which a program or computer fails to respond following the most recent mouse click or keyboard action. The timer can also be used for other purposes, for example, to actuate the refresh (or reload) button in a Web browser if a Web site does not fully load after a certain length of time following the entry of a Uniform Resource Locator (URL).

A WDT contains a digital counter that counts down to zero at a constant speed from a preset number. The counter speed is kept constant by a clock circuit. If the counter reaches zero before the computer recovers, a signal is sent to designated circuits to perform the desired action.