International Journal of Research (IJR) Vol-1, Issue-10 November 2014 **ISSN 2348-6848**

Embedded Syestem: Microcontroller

Kuldeep Panwar[1] & Devanshu Sharma[2]
electronics and Computer Engineering.
[1]kuldeep.15715@Ggnindia.Dronacharya.Info ; [2] Devanshu.15706@Ggnindia.Dronacharya.Info

ABSTRACT
This research paper is dealing with the microcontroller of an embedded system. Introduction to microcontroller is represented here. Embedded and external microcontroller are elaborated here. This is just a basic approach towards embedded system. The basic details of embedded system is defined here.

I. INTRODUCTION

A **microcontroller** (sometimes abbreviated **µC**, **uC** or **MCU**) is a small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals. Program memory in the form of NOR flash or OTP ROM is also often included on chip, as well as a typically small amount of RAM. Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computers or other general purpose applications.[1]

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys and other embedded systems.[2] By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control even more devices and processes.[3] Mixed signal microcontrollers are common, integrating analog components needed to control non-digital electronic systems.[4]

Some microcontrollers may use four-bit words and operate at clock rate frequencies as low as 4 kHz, for low power consumption (single-digit milliwatts or microwatts).[5] They will generally have the ability to retain functionality while waiting for an event such as a button press or other interrupt; power consumption while sleeping (CPU clock and most peripherals off) may be just nanowatts, making many of them well suited for long lasting battery applications.[5] Other microcontrollers may serve performance-critical roles, where they may need to act more like a digital signal processor (DSP), with higher clock speeds and power consumption.[2]

II. TYPES OF MICROCONTROLLER

General Purpose

- Microcontroller manufacturers, such as Atmel and Microchip, offer general purpose microcontroller families.[4] Within the general purpose device types, there are often various configurations available such as 8-bit, 16-bit and 32-bit word sizes.[5] Word size refers to the size of binary numbers that can be handled by the microcontroller. Also, the general purpose devices come in different memory and peripheral configurations. General purpose microcontrollers normally have a set of features that would be useful in a variety of applications and can be designed into products such as home appliances and consumer products.

Signal Processing

- As the speed and processing power of microcontrollers has increased, manufacturers have combined features of a microcontroller with features of a digital signal processor (or DSP).[6] For example, Microchip offers the dsPIC line of

products that they refer to as digital signal controllers (or DSCs), which have microcontroller features and DSP features in a single core.[12] Signal processing microcontrollers typically combine the built-in memory and simple instruction sets of microcontrollers with the efficient signal processing arithmetic circuits found in DSPs.[11] Signal processing microcontrollers are used in applications such as intelligent power supplies that convert electrical power from one form to another.[10]

III. INTERRUPT LATENCY

In contrast to general-purpose computers, microcontrollers used in embedded systems often seek to optimize interrupt latency over instruction throughput. Issues include both reducing the latency, and making it be more predictable (to support real-time control).[8]

When an electronic device causes an interrupt, the intermediate results (registers) have to be saved before the software responsible for handling the interrupt can run.[8] They must also be restored after that software is finished. If there are more registers, this saving and restoring process takes more time, increasing the latency.[7] Ways to reduce such context/restore latency include having relatively few registers in their central processing units (undesirable because it slows down most non-interrupt processing substantially), or at least having the hardware not save them all (this fails if the software then needs to compensate by saving the rest "manually"). Another technique involves spending silicon gates on "shadow registers": One or more duplicate registers used only by the interrupt software, perhaps supporting a dedicated stack.[12]

Other factors affecting interrupt latency include:

- Cycles needed to complete current CPU activities. To minimize those costs, microcontrollers tend to have short pipelines (often three instructions or less), small write buffers, and ensure that longer instructions are continuable or restartable. Reduced instruction set computing/RISC design principles ensure that most instructions take the same number of cycles, helping avoid the need for most such continuation/restart logic.[8]
- The length of any critical section that needs to be interrupted. Entry to a critical section restricts concurrent data structure access. When a data structure must be accessed by an interrupt handler, the critical section must block that interrupt. Accordingly, interrupt latency is increased by however long that interrupt is blocked.[4] When there are hard external constraints on system l0atency, developers often need tools to measure interrupt latencies and track down which critical sections cause slowdowns[3].
  - One common technique just blocks all interrupts for the duration of the critical section. This is easy to implement, but sometimes critical sections get uncomfortably long.[3]
  - A more complex technique just blocks the interrupts that may trigger access to that data structure. This is often based on interrupt priorities, which tend to not correspond well to the relevant system data structures. Accordingly, this technique is used mostly in very constrained environments.[1]
  - Processors may have hardware support for some critical sections. Examples include supporting atomic access to bits or bytes within a word, or other atomic access primitives like the Load-link/store-conditional/LDREX/STREX exclusive access primitives introduced in the ARMv6 architecture.[2]
- Interrupt nesting. Some microcontrollers allow higher priority interrupts to interrupt lower priority ones. This allows software to manage latency by giving time-critical interrupts higher priority (and thus lower and

more predictable latency) than less-critical ones.

- Trigger rate. When interrupts occur back-to-back, microcontrollers may avoid an extra context save/restore cycle by a form of tail call optimization.

Lower end microcontrollers tend to support fewer interrupt latency controls than higher end ones.

IV. MICROCONTROLLER EMBEDDED MEMORY TECHNOLOGY

Since the emergence of microcontrollers, many different memory technologies have been used. Almost all microcontrollers have at least two different kinds of memory, a non-volatile memory for storing firmware and a read-write memory for temporary data.

Data

From the earliest microcontrollers to today, six-transistor SRAM is almost always used as the read/write working memory, with a few more transistors per bit used in the register file. FRAM or MRAM could potentially replace it as it is 4 to 10 times denser which would make it more cost effective.[10]

In addition to the SRAM, some microcontrollers also have internal EEPROM for data storage; and even ones that do not have any (or not enough) are often connected to external serial EEPROM chip (such as the BASIC Stamp) or external serial flash memory chip.[11]

A few recent microcontrollers beginning in 2003 have "self-programmable" flash memory.[

Firmware

The earliest microcontrollers used mask ROM to store firmware. Later microcontrollers (such as the early versions of the Freescale 68HC11 and early PIC microcontrollers) had quartz windows that allowed ultraviolet light in to erase the EPROM.

The Microchip PIC16C84, introduced in 1993, was the first microcontroller to use EEPROM to store firmware. In the same year, Atmel introduced the first microcontroller using NOR Flash memory to store firmware.

V. CONCLUSION

Placing an embedded microprocessor system into a product makes the product smart. It can then be programmed to do things that are too difficult or expensive using conventional technologies such as logic, or time switches, and so on. Link such a smart product to the Internet and it can do even more. For example, products can be programmed to do self-diagnostic checks and to report back to the manufacturer. Not only does this provide the potential to collect data that can be used to improve products, it can also allow for the manufacturer to inform the user of potential problems, so that action can be taken. This opens up possibilities for improved customer service as well as new services. Basically, embedded microprocessors enable firms to compete on product and service innovation, by adding product and service features that customers value, but which would be largely impossible without this technology.

REFERENCES

1. Augarten, Stan (1983). "The Most Widely Used Computer on a Chip: The TMS 1000". *State of the Art: A Photographic History of the Integrated Circuit* (New Haven and New York: Ticknor & Fields). ISBN 0-89919-195-9. Retrieved 2009-12-23.
2. [b] "Oral History Panel on the Development and Promotion of the Intel 8048 Microcontroller". *Computer History Museum Oral History, 2008*. p. 4. Retrieved 2011-06-28.
3. ""Atmel's Self-Programming Flash Microcontrollers"". 2012-01-24.

Retrieved 2008-10-25. by Odd Jostein Svendsli 2003

4. Jim Turley. "The Two Percent Solution" 2002.
5. Tom Cantrell "Microchip on the March". Circuit Cellar. 1998.
6. Momentum Carries MCUs Into 2011 http://semico.com/content/momentum-carries-mcus-2011
7. Heath, Steve (2003). *Embedded systems design*. EDN series for design engineers (2 ed.). Newnes. pp. 11–12. ISBN 9780750655460.
8. Easy Way to build a microcontroller project
9. "8052-Basic Microcontrollers" by Jan Axelson 1994
10. Edwards, Robert (1987). *Optimizing the Zilog Z8 Forth Microcontroller for Rapid Prototyping*. Martin Marietta. p. 3. Retrieved 9 December 2012.
11. Microchip unveils PIC16C84, a reprogrammable EEPROM-based 8-bit microcontroller 1993