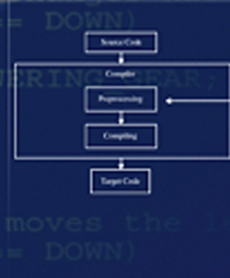




EMBEDDED TECHNOLOGY™
SERIES

Embedded Systems Architecture

A Comprehensive Guide for Engineers
and Programmers



Tammy Noergaard

Foreword by Jack Ganssle



CD-ROM included- Contains design and instructional tools



Embedded Systems Architecture

This Page Intentionally Left Blank

Embedded Systems Architecture

A Comprehensive Guide for Engineers and Programmers

By

Tammy Noergaard



ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier



Newnes

Newnes is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
Linacre House, Jordan Hill, Oxford OX2 8DP, UK

Copyright © 2005, Elsevier Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail: permissions@elsevier.com.uk. You may also complete your request on-line via the Elsevier homepage (<http://elsevier.com>), by selecting "Customer Support" and then "Obtaining Permissions."



Recognizing the importance of preserving what has been written,
Elsevier prints its books on acid-free paper whenever possible.

Library of Congress Cataloging-in-Publication Data

(Application submitted.)

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

ISBN: 0-7506-7792-9

For information on all Newnes publications
visit our Web site at www.books.elsevier.com

04 05 06 07 08 09 10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER

BOOK AID
International

Sabre Foundation

Dedication

To the engineer and man I respect and admire the most,
my father,
Dr. Al M. Zied

This Page Intentionally Left Blank

Contents

Foreword	xi
Acknowledgments	xiii
About the Author	xiv
<i>Section I: Introduction to Embedded Systems</i>	1
Chapter 1: A Systems Engineering Approach to Embedded Systems Design	5
1.1 What Is an Embedded System?	5
1.2 Embedded Systems Design	7
1.3 An Introduction to Embedded Systems Architecture	9
1.4 Why Is the Architecture of an Embedded System Important?	11
1.5 The Embedded Systems Model	12
1.6 Summary	13
Chapter 1 Problems	15
Chapter 2: Know Your Standards	17
2.1 An Overview of Programming Languages and Examples of Their Standards	30
2.2 Standards and Networking	46
2.3 Multiple Standards-Based Device Example: Digital Television (DTV)	65
2.4 Summary	67
Chapter 2 Problems	69
<i>Section II: Embedded Hardware</i>	73
Chapter 3: Embedded Hardware Building Blocks and the Embedded Board	77
3.1 Lesson One on Hardware: Learn to Read a Schematic!	77
3.2 The Embedded Board and the von Neumann Model	82
3.3 Powering the Hardware	87
3.4 Basic Hardware Materials: Conductors, Insulators, and Semiconductors	89
3.5 Common Passive Components on Boards and in Chips: Resistors, Capacitors, and Inductors	93
3.6 Semiconductors and the Active Building Blocks of Processors and Memory	101
3.7 Putting It All Together: The Integrated Circuit (IC)	117
3.8 Summary	121
Chapter 3 Problems	122

Contents

Chapter 4: Embedded Processors.....	129
4.1 ISA Architecture Models	131
4.2 Internal Processor Design	145
4.3 Processor Performance	203
4.4 Reading a Processor's Datasheet	206
4.5 Summary	218
Chapter 4 Problems.....	219
Chapter 5: Board Memory	223
5.1 Read-Only Memory (ROM)	227
5.2 Random-Access Memory (RAM).....	232
5.3 Auxiliary Memory	242
5.4 Memory Management of External Memory	247
5.5 Board Memory and Performance.....	249
5.6 Summary	250
Chapter 5 Problems.....	251
Chapter 6: Board I/O (Input/Output)	253
6.1 Managing Data: Serial vs. Parallel I/O	257
6.2 Interfacing the I/O Components	277
6.3 I/O and Performance	280
6.4 Summary	282
Chapter 6 Problems.....	283
Chapter 7: Board Buses.....	287
7.1 Bus Arbitration and Timing	289
7.2 Integrating the Bus with Other Board Components.....	299
7.3 Bus Performance.....	300
7.4 Summary	301
Chapter 7 Problems.....	302
Section III: Embedded Software Introduction.....	307
Chapter 8: Device Drivers	311
8.1 Example 1: Device Drivers for Interrupt-Handling	315
8.2 Example 2: Memory Device Drivers	332
8.3 Example 3: On-board Bus Device Drivers	351
8.4 Board I/O Driver Examples	358
8.5 Summary	379
Chapter 8 Problems.....	380
Chapter 9: Embedded Operating Systems	383
9.1 What Is a Process?	388
9.2 Multitasking and Process Management	390
9.3 Memory Management.....	421

9.4 I/O and File System Management	435
9.5 OS Standards Example: POSIX (Portable Operating System Interface).....	437
9.6 OS Performance Guidelines.....	439
9.7 Oses and Board Support Packages (BSPs).....	440
9.8 Summary	441
Chapter 9 Problems.....	442
Chapter 10: Middleware and Application Software.....	445
10.1 What Is Middleware?	445
10.2 What Is an Application?	447
10.3 Middleware Examples.....	447
10.4 Application Layer Software Examples	484
10.5 Summary	498
Chapter 10 Problems.....	499
Section IV: Putting It All Together: Design and Development.....	505
Chapter 11: Defining the System—Creating the Architecture and Documenting the Design	509
11.1 Creating an Embedded System Architecture	510
Stage 1: Have a Solid Technical Foundation	511
Stage 2: Know the ABCs (Architecture Business Cycles) of Embedded Systems.....	512
Stage 3: Define the Architectural Patterns and Reference Models	523
Stage 4: Define the Architectural Structures.....	530
Stage 5: Document the Architecture	533
Stage 6: Analyze and Evaluate the Architecture	535
11.2 Summary	537
Chapter 11 Problems.....	538
Chapter 12: The Final Phases of Embedded Design: Implementation and Testing.....	541
12.1 Implementing the Design.....	541
12.1.1 The Main Software Utility Tool: Writing Code in an Editor or IDE	542
12.1.2 Computer-Aided Design (CAD) and the Hardware	543
12.1.3 Translation Tools—Preprocessors, Interpreters, Compilers, and Linkers	545
12.1.4 Debugging Tools.....	548
12.1.5 System Boot-Up.....	555
12.2 Quality Assurance and Testing of the Design	563
12.3 Conclusion: Maintaining the Embedded System and Beyond.....	566
Chapter 12 Problems.....	567
Appendix A: Projects and Exercises	571
Section I Projects	574
Section II Projects	578

Contents

Section III Projects.....	586
Section IV Projects	589
Appendix B: Schematic Symbols.....	594
Appendix C: Acronyms and Abbreviations	601
Appendix D: Glossary	610
Index	627
What's on the CD-ROM.....	640

Foreword

When Tammy Noergaard first told me she wanted to write a soup-to-nuts book about building embedded systems I tried to dissuade her. This field is so vast, requiring insight into electronics, logic circuits, computer design, software engineering, C, assembly, and far more. But as we talked she showed me how the industry's literature lacks a definitive work on the subject. I warned her of the immensity of the project.

A year and many discussions later Fedex arrived with the review copy of this book. At over 700 pages it's appropriately twice the size of almost any other opus on the subject. The book you're holding truly is "A Comprehensive Guide for Engineers and Programmers." Sure, the minutia of programming a PIC's timer might have been left out, but the scope is vast and important.

Tammy starts with the first principles of electronics and advances through software to the expensive end-phase of maintenance. She treats hardware and software as an integrated whole, which sort of defines the nature of embedded systems. Ironically, though, developers are increasingly specialized. More than a few software folks haven't a clue about transistors while too many EEs can't accurately define middleware. I fear readers may skip those chapters that don't immediately pertain to the project at hand.

Resist any such temptation, gentle reader! Become a true master, an embedded sage, by broadening your horizons to cover all aspects of this fascinating field. We engineers are professionals; you and I know this in our hearts. Yet true professionals are those who learn new things, who apply newly evolving technologies to solve problems. Consider doctors: the discovery and production of penicillin in the 1940s changed the profession of medicine forever. Any doc who ignored this new technology, who continued to practice using only the skills learned in college, was suddenly rendered a butcher. Software and hardware developers are faced with the same situation. C wasn't taught when I went to school. The FPGA hadn't been invented. GOTOs were still just fine, thank you. We learned to program microprocessors in machine code using primitive toolchains. Today—well, we know how much has changed.

The rate of change is increasing; change's first derivative is an ever-escalating positive number. Professional developers will read this book from cover to cover, and will constantly seek out other sources of information. If you're not at least surfing through a half dozen technical magazines a month and reading a handful of books like this per year, then it won't take a Cretaceous asteroid to make you a dinosaur.

Foreword

Some of this book might surprise you. Ten pages about reading datasheets? Fact is, datasheets are dense formal compilations of contractual material. The vendor promises the part will do x as long as we use it in an agreed-on manner. Violate any of perhaps thousands of specifications and the part will either not work or will be unreliable. With some parts dissipating 100 watts or more, even such arcana as thermal characteristics are as important as the device's instruction set.

Tammy's generous use of examples elucidates the more obscure points. Engineering—whether hardware or software—is the art of building things and solving problems. The academics can work with dry theory; we practicing developers often learn best by seeing how something works. So the chapter on device drivers does explain the intricacies of building these often-complex bits of code, but couples the explanation to a wealth of real-world examples.

Finally, Tammy's words about the Architecture Business Cycle of embedded systems resonate strongly with me. We don't build these things just to have a good time (though we sure hope to have one along the way), but to solve important business problems. Every decision we make has business implications. Use too little horsepower and development costs skyrocket—sometimes to the point of making the project unviable. A poor analysis of the problem that leads you to toss in an excess of Flash might drive costs unacceptably high. Select a component (hardware or software) from a failing company and your outfit may share in the vendor's demise.

Enjoy this book, and futureproof your career at the same time.

—Jack Ganssle

Acknowledgments

My greatest debt in creating this book goes to the reviewers, who I hope will be pleasantly surprised to see how many of their suggestions have been incorporated into the book. They include Dr. Al M. Zied, both of my brothers (especially my younger brother who also provided me the inspiration to write this book in the first place), Jack Ganssle, Dr. Volker Enders, Dr. Stefan Frank, Dr. Karl Mathia, and Steve Bailey.

Thank you to my publisher Elsevier, specifically to my editor Carol Lewis and the rest of “my” Elsevier team for their hard work and dedication in making this book a reality.

I would also like to acknowledge my mentor when I was with Sony Electronics, Kazuhisa Maruoka, who patiently trained me to design televisions and gave me such a strong foundation upon which to grow, as well as my manager at Sony Electronics, Satoshi Ishiguro, who took a chance and hired me. My journey in the embedded systems field that has led me to writing this book began with the great people I worked with at Sony in Japan and in San Diego.

A very special thanks to my family for their support, which allowed me to write this book, and without whose support and encouragement I could never have completed it. To my husband, Christian, thank you for giving me the happiest days of my life, and the opportunity and encouragement to realize this book. For my beautiful baby girl Mia, thank you for being so patient with your Mama while I did my writing—always gifting me with endless smiles, hugs, and kisses when I really needed them. Thank you, Mom, for the times you flew out from Belgium, and thanks to my sister Mandy for flying up from Southern California, to help me at home so I could finish my writing, and for encouraging me every step of the way. Finally, a special thanks to Verónica Cervantes Gaona for taking such good care of Mia so that I could have the time and focus to write. I am very grateful for the quality time you spent with her.

About the Author

Tammy Noergaard is uniquely qualified to write about all aspects of embedded systems architecture. Since beginning her embedded systems career in 1995, she has had wide experience in product development, system design and integration, operations, sales, marketing, and training. She has design experience using many hardware platforms, operating systems, and languages. Noergaard worked for Sony as a lead software engineer developing and testing embedded software for analog TVs, and also managed and trained new embedded engineers and programmers. The televisions that she helped to develop were critically acclaimed and rated #1 in *Consumer Reports* magazines. At Wind River she was the liaison engineer between developmental engineers and customers to provide design expertise, systems configuration, systems integration, and training for Wind River embedded software (OS, Java, device drivers, etc.) and all associated hardware for a variety of embedded systems in the Consumer Electronic market. Most recently she was a Field Engineering Specialist and Consultant with Esmertec North America, providing project management, system design, system integration, system configuration, support and expertise for various embedded Java systems using Jbed in everything from control systems to medical devices to digital TVs. Noergaard has lectured to engineering classes at the University of California at Berkeley and Stanford, the Embedded Internet Conference, and the Java User's Group in San Jose, among others.

A Systems Engineering Approach to Embedded Systems Design

In This Chapter

- ▶ *Define embedded system*
- ▶ *Introduce the design process*
- ▶ *Define an embedded systems architecture*
- ▶ *Discuss the impact of architecture*
- ▶ *Summarize the remaining sections of the book*

1.1 What Is an Embedded System?

An embedded system is an applied computer system, as distinguished from other types of computer systems such as personal computers (PCs) or supercomputers. However, you will find that the definition of “embedded system” is fluid and difficult to pin down, as it constantly evolves with advances in technology and dramatic decreases in the cost of implementing various hardware and software components. In recent years, the field has outgrown many of its traditional descriptions. Because the reader will likely encounter some of these descriptions and definitions, it is important to understand the reasoning behind them and why they may or may not be accurate today, and to be able to discuss them knowledgeably. Following are a few of the more common descriptions of an embedded system:

- *Embedded systems are more limited in hardware and/or software functionality than a personal computer (PC).* This holds true for a significant subset of the embedded systems family of computer systems. In terms of hardware limitations, this can mean limitations in processing performance, power consumption, memory, hardware functionality, and so forth. In software, this typically means limitations relative to a PC—fewer applications, scaled-down applications, no operating system (OS) or a limited OS, or less abstraction-level code. However, this definition is only partially true today as boards and software typically found in PCs of past and present have been repackaged into more complex embedded system designs.
- *An embedded system is designed to perform a dedicated function.* Most embedded devices are primarily designed for one specific function. However, we now see devices such as personal data assistant (PDA)/cell phone hybrids, which are embedded systems designed to be able to do a variety of primary functions. Also, the latest digital TVs include interactive applications that perform a wide variety of general

functions unrelated to the “TV” function but just as important, such as e-mail, web browsing, and games.

- *An embedded system is a computer system with higher quality and reliability requirements than other types of computer systems.* Some families of embedded devices have a very high threshold of quality and reliability requirements. For example, if a car’s engine controller crashes while driving on a busy freeway or a critical medical device malfunctions during surgery, very serious problems result. However, there are also embedded devices, such as TVs, games, and cell phones, in which a malfunction is an inconvenience but not usually a life-threatening situation.
- *Some devices that are called embedded systems, such as PDAs or web pads, are not really embedded systems.* There is some discussion as to whether or not computer systems that meet some, but not all of the traditional embedded system definitions are actually embedded systems or something else. Some feel that the designation of these more complex designs, such as PDAs, as embedded systems is driven by nontechnical marketing and sales professionals, rather than engineers. In reality, embedded engineers are divided as to whether these designs are or are not embedded systems, even though currently these systems are often discussed as such among these same designers. Whether or not the traditional embedded definitions should continue to evolve, or a new field of computer systems be designated to include these more complex systems will ultimately be determined by others in the industry. For now, since there is no new industry-supported field of computer systems designated for designs that fall in between the traditional embedded system and the general-purpose PC systems, this book supports the evolutionary view of embedded systems that encompasses these types of computer system designs.

Electronic devices in just about every engineering market segment are classified as embedded systems (see Table 1-1). In short, outside of being “types of computer systems,” the only specific characterization that continues to hold true for the wide spectrum of embedded system devices is that *there is no single definition reflecting them all*.

Table 1-1: Examples of embedded systems and their markets ^[1-1]

Market	Embedded Device
Automotive	Ignition System
	Engine Control
	Brake System (i.e., Antilock Braking System)
Consumer Electronics	Digital and Analog Televisions
	Set-Top Boxes (DVDs, VCRs, Cable Boxes, etc.)
	Personal Data Assistants (PDAs)
	Kitchen Appliances (Refrigerators, Toasters, Microwave Ovens)
	Automobiles
	Toys/Games
	Telephones/Cell Phones/Pagers
	Cameras
	Global Positioning Systems (GPS)

Table 1-1: Examples of embedded systems and their markets ^[1-1] (continued)

Market	Embedded Device
Industrial Control	Robotics and Control Systems (Manufacturing)
Medical	Infusion Pumps
	Dialysis Machines
	Prosthetic Devices
	Cardiac Monitors
Networking	Routers
	Hubs
	Gateways
Office Automation	Fax Machine
	Photocopier
	Printers
	Monitors
	Scanners

1.2 Embedded Systems Design

When approaching embedded systems architecture design from a systems engineering point of view, several models can be applied to describe the cycle of embedded system design. Most of these models are based upon one or some combination of the following development models:^[1-5]

- The *big-bang* model, in which there is essentially no planning or processes in place before and during the development of a system.
- The *code-and-fix* model, in which product requirements are defined but no formal processes are in place before the start of development.
- The *waterfall* model, in which there is a process for developing a system in steps, where results of one step flow into the next step.
- The *spiral* model, in which there is a process for developing a system in steps, and throughout the various steps, feedback is obtained and incorporated back into the process.

This book supports the model shown in Figure 1-1, which I refer to as the Embedded Systems Design and Development Lifecycle Model. This model is based on a combination of the popular waterfall and spiral industry models.^[1-2] When I investigated and analyzed the many successful embedded projects that I have been a part of or had detailed knowledge about over the years, and analyzed the failed projects or those that ran into many difficulties meeting technical and/or business requirements, I concluded that the successful projects contained at least one common factor that the problem projects lacked. This factor is the process shown in Figure 1-1, and this is why I introduce this model as an important tool in understanding an embedded system's design process.

As shown in Figure 1-1, the embedded system design and development process is divided into four phases: creating the architecture, implementing the architecture, testing the system, and

- *Segmented.* A segmented address space is a portion of memory that is divided into sections called segments. Specific memory locations can only be accessed by specifying a segment identifier, a segment number that can be explicitly defined or implicitly obtained from a register, and specifying the offset within a specific segment within the segmented address space.

The offset within the segment contains a base address and a limit, which map to another portion of memory that is set up as a linear address space. If the offset is less than or equal to the limit, the offset is added to the base address, giving the unsegmented address within the linear address space.

- Containing any *special address regions*.
- *Limited* in any way.

An important note regarding ISAs and memory is that different ISAs not only define where data is stored, but also how data is stored in memory—specifically in what order the bits (or bytes) that make up the data is stored, or **byte ordering**. The two byte-ordering approaches are big-endian, in which the most significant byte or bit is stored first, and little-endian, in which the least significant bit or byte is stored first.

For example:

- 68000 and SPARC are big-endian
- x86 is little-endian
- ARM, MIPS and PowerPC can be configured as either big-endian or little-endian using a bit in their machine state registers

B. *Register Set*

A register is simply fast programmable memory normally used to store operands that are immediately or frequently used. A processor's set of registers is commonly referred to as the **register set** or the **register file**. Different processors have different register sets, and the number of registers in their sets vary between very few to several hundred (even over a thousand). For example, the SA-1110 register set has 37 32-bit registers, whereas the MPC823, on the other hand, has about a few hundred registers (general-purpose, special purpose, floating point registers, etc.).

C. *How Registers Are Used*

An ISA defines which registers can be used for what transactions, such as special purpose, floating point, and which can be used by the programmer in a general fashion (general purpose registers).

Chapter 4

- the control unit (CU) – *manages the entire fetching and execution cycle*
- the internal CPU buses – *interconnect the ALU, registers, and the CU*

Looking at a real-world processor, these four fundamental elements defined by the von Neumann model can be seen within the CPU of the MPC860 (see Figure 4-21).

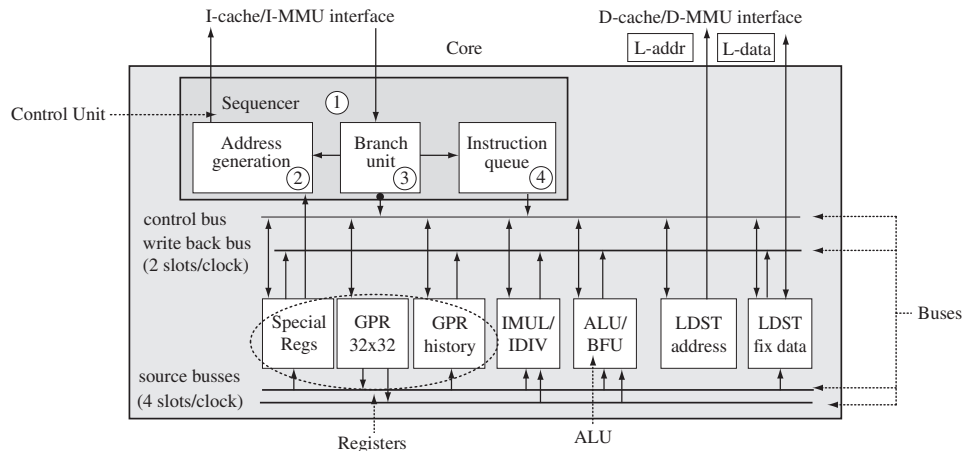


Figure 4-21: The MPC860 CPU – the PowerPC core ^[4-15]

Copyright of Freescale Semiconductor, Inc. 2004. Used by permission.

Remember: Not all processors have these components as strictly defined by the von Neumann model, but will have some combination of these components under various aliases somewhere on the processor. Remember that this model is a reference tool the reader can use to understand the major components of a CPU design.

Internal CPU Buses

The CPU buses are the mechanisms that interconnect the CPU's other components: the ALU, the CU, and registers (see Figure 4-22). Buses are simply wires that interconnect the various other components within the CPU. Each bus's wire is typically divided into logical functions, such as data (which carries data, bi-directionally, between registers and the ALU), address (which carries the locations of the registers that contain the data to be transferred), control (which carries control signal information, such as timing and control signals, between the registers, the ALU, and the CU), and so on.

Note: To avoid redundancy, buses will be discussed in more detail in Chapter 7.

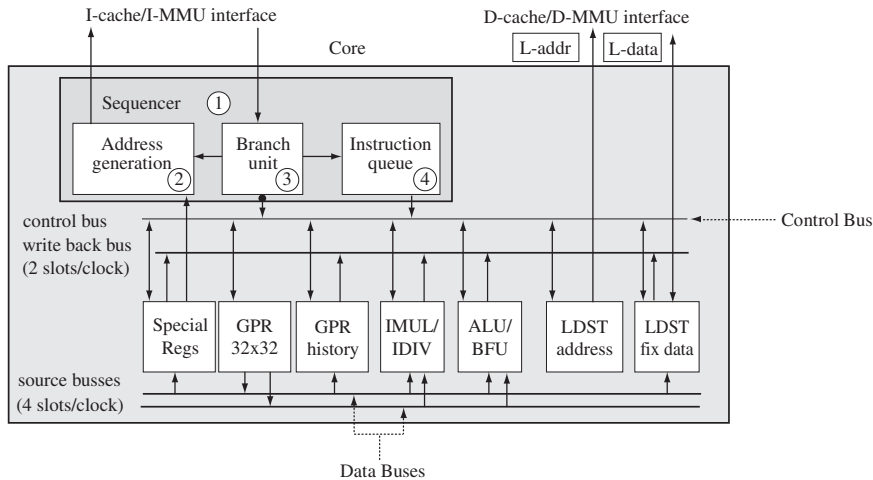


Figure 4-22: PowerPC core and buses ^[4-15]

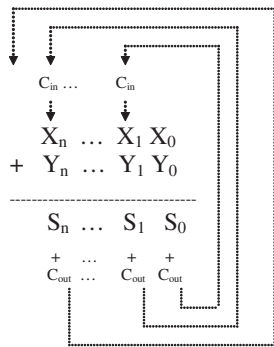
In the PowerPC Core, there is a Control Bus that carries the control signals between the ALU, CU, and registers. What the PowerPC calls “source buses,” are the data buses that carry the data between registers and the ALU. There is an additional bus called the write-back which is dedicated to writing back data received from a source bus directly back from the load/store unit to the fixed or floating point registers.

Arithmetic Logic Unit (ALU)

The arithmetic logic unit (ALU) implements the comparison, mathematical and logical operations defined by the ISA. The format and types of operations implemented in the CPU’s ALU can vary depending on the ISA. Considered the core of any processor, the ALU is responsible for accepting multiple n -bit binary operands and performing any logical (AND, OR, NOT, etc.), mathematical (+, −, *, etc.), and comparison (=, <, >, etc.) operations on these operands.

The ALU is a combinational logic circuit that can have one or more inputs and only one output. An ALU’s output is dependent only on inputs applied at that instant, as a function of time, and “no” past conditions (see Chapter 3 on gates). The basic building block of most ALUs (from the simplest to the multifunctional) is considered the **full adder**, a logic circuit that takes three 1-bit numbers as inputs and produces two 1-bit numbers. How this actually works will be discussed in more detail later this section.

To understand how a full adder works, let us first examine the mechanics of adding binary numbers (0’s and 1’s) together:



Starting with two 1-bit numbers, adding them will produce, at most, a 2-bit number:

X_0	Y_0	S_0	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$\Rightarrow 0b + 0b = 0b$
 $\Rightarrow 0b + 1b = 0b$
 $\Rightarrow 1b + 0b = 1b$
 $\Rightarrow 1b + 1b = 10b$ (or 2d) In binary addition of 2 1-bit numbers, when the count exceeds 10 (the binary of 2 decimal), the 1 (C_{out}) is carried and added to the next row of numbers thus resulting in a 2-bit number.

This simple addition of two 1-bit numbers can be executed via a *half-adder* circuit, a logical circuit that takes two 1-bit numbers as inputs and produces a 2-bit output. Half-adder circuits, like all logical circuits, can be designed using several possible combinations of gates, such as the possible combinations shown in Figure 4-23a.

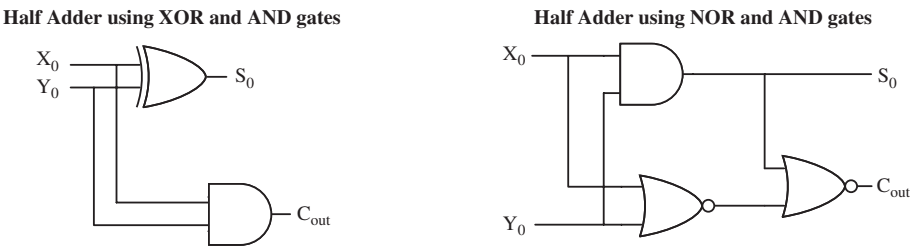


Figure 4-23a: Half-adder logic circuits [4-21]

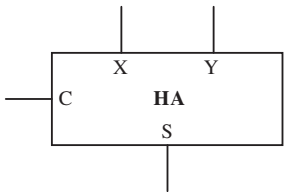


Figure 4-23b: Half-adder logic symbol [4-21]

In order to add a larger number, the adder circuit must increase in complexity, and this is where the full adder comes into play. When trying to add two-digit numbers, for example, a full adder must be used in conjunction with a half adder. The half adder takes care of adding the first digits of the two numbers to be added (i.e., x_0, y_0 , and so on)—the full adder’s three 1-bit inputs are the second digits of the two numbers to be added (i.e., $x_1, y_1, \dots$) along with the carry (C_{in}) in from the half adder’s addition of the first digits. The half adder’s output is the sum (S_0) along with the carry out (C_{out}) of the first digit’s addition operation; the two 1-bit outputs of the full adder are the sum (S_1) along with the carry out (C_{out}) of the second digit’s addition operation. Figure 4-24a shows the logic equations and truth table, Figure 4-24b shows the logic symbol, and Figure 4-24c shows an example of a full adder at the gate level, in this case, a combination XOR and NAND gates.

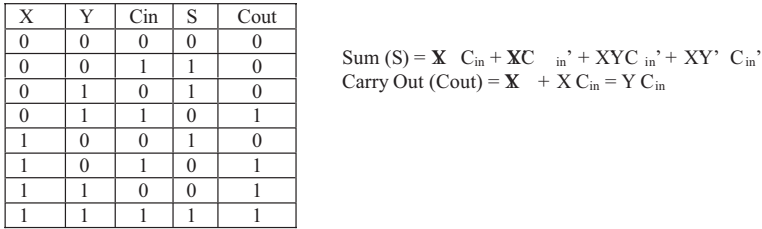


Figure 4-24a: Full adder truth table and logic equations [4-20]

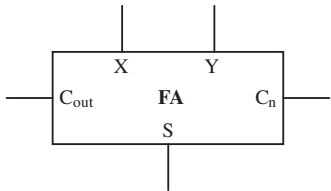


Figure 4-24b: Full adder logic symbol [4-20]

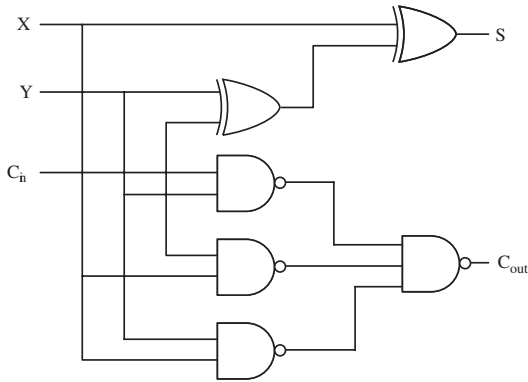


Figure 4-24c: Full adder gate-level circuit [4-20]

Chapter 4

Where the ALU saves the generated results varies with different architectures. With the PowerPC shown in Figure 4-25, results are saved in a register called an *Accumulator*. Results can also be saved in memory (on a stack or elsewhere), or in some hybrid combination of these locations.

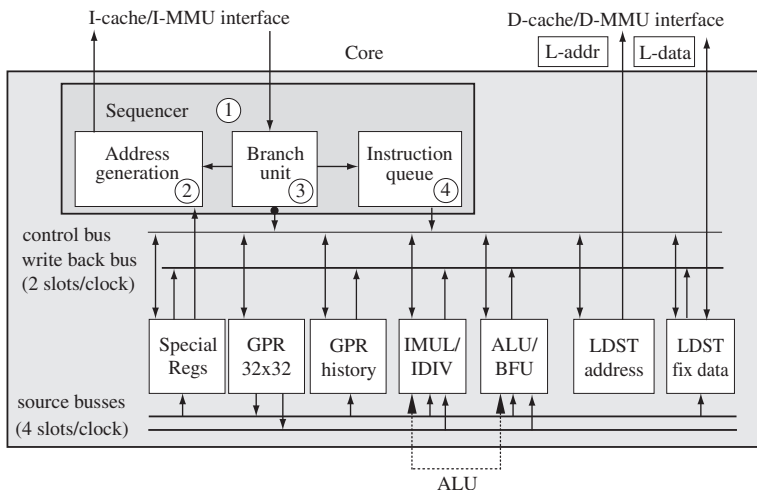


Figure 4-27: PowerPC core and the ALU ^[4-15]

In the PowerPC core, the ALU is part of the “Fixed Point Unit” that implements all fixed-point instructions other than load/store instructions. The ALU is responsible for fixed-point logic, add, and subtract instruction implementation. In the case of the PowerPC, generated results of the ALU are stored in an Accumulator. Also, note that the PowerPC has an IMUL/IDIV unit (essentially another ALU) specifically for performing multiplication and division operations.

Registers

Registers are simply a combination of various flip-flops that can be used to temporarily store data or to delay signals. A **storage register** is a form of fast programmable internal processor memory usually used to temporarily store, copy, and modify operands that are immediately or frequently used by the system. **Shift registers** delay signals by passing the signals between the various internal flip-flops with every clock pulse.

Registers are made up of a set of flip-flops that can be activated either individually or as a set. In fact, it is the *number of flip-flops in each register* that is actually used to describe a processor (for example, a 32-bit processor has working registers that are 32 bits wide containing 32 flip-flops, a 16-bit processor has working registers that are 16 bits wide containing 16 flip-flops, and so on). The number of flip-flops within these registers also determines the width of the data buses used in the system. Figure 4-28 shows an example of how eight flip-flops could comprise an 8-bit register, and thus impact the size of the data bus. In short, registers are made up of one flip-flop for every bit being manipulated or stored by the register.

- ***Some mechanism for both to exchange data.*** This refers to how data is actually exchanged between the I/O controller and the master processor. In a *programmed transfer*, the master processor receives data from the I/O controller into its registers, and the CPU then transmits this data to memory. For memory-mapped I/O schemes, DMA (direct memory access) circuitry can be used to bypass the master CPU entirely. DMA has the ability to manage data transmissions or receptions directly to and from main memory and an I/O device. On some systems, DMA is integrated into the master processor, and on others there is a separate DMA controller. Essentially, DMA requests control of the bus from the master processor.

6.3 I/O and Performance

I/O performance is one of the most important issues of an embedded design. I/O can negatively impact performance by *bottlenecking the entire system*. In order to understand the type of performance hurdles I/O must overcome, it is important to understand that, with the wide variety of I/O devices, each device will have its own unique qualities. Thus, in a proper design, the engineer will have taken these unique qualities on a case by case basis into consideration. Some of the most important shared features of I/O that can negatively impact board performance include:

- ***The data rates of the I/O devices.*** I/O devices on one board can vary in data rates from a handful of characters per second with a keyboard or a mouse to devices that can transmit in Mbytes per second (networking, tape, disk).
- ***The speed of the master processor.*** Master processors can have clocks rates anywhere from tens of MHz to hundreds of MHz. Given an I/O device with an extremely slow data rate, a master CPU could have executed thousands of times more data in the time period that the I/O needs to process a handful of *bits* of data. With extremely fast I/O, a master processor would not even be able to process anything before the I/O device is ready to move forward.
- ***How to synchronize the speed of the master processor to the speeds of I/O.*** Given the extreme ranges of performance, a realistic scheme must be implemented that allows for either the I/O or master processor to process data successfully regardless of how different their speeds. Otherwise, with an I/O device processing data much slower than the master processor transmits, for instance, data would be lost by the I/O device. If the device is not ready, it could hang the entire system if there is no mechanism to handle this situation.

- **How I/O and the master processor communicate.** This includes whether there is an intermediate dedicated I/O controller between the master CPU and I/O device that manages I/O for the master processor, thus freeing up the CPU to process data more efficiently. Relative to an I/O controller, it becomes a question whether the communication scheme is interrupt driven, polled, or memory mapped (with dedicated DMA to, again, free up the master CPU). If interrupt-driven, for example, can I/O devices interrupt other I/O, or would devices on the queue have to wait until previous devices finished their turn, no matter how slow.

To improve I/O performance and prevent bottlenecks, board designers need to examine the various I/O and master processor communication schemes to ensure that every device can be managed successfully via one of the available schemes. For example, to synchronize slower I/O devices and the master CPU, status flags or interrupts can be made available for all ICs so that they can communicate their status to each other when processing data. Another example occurs when I/O devices are faster than the master CPU. In this case, some type of interface (i.e., DMA) that allows these devices to bypass the master processor altogether could be an alternative.

The most common units measuring performance relative to I/O include:

- **Throughput** of the various I/O components (the maximum amount of data per unit time that can be processed, in bytes per second). This value can vary for different components. The components with the *lowest* throughput are what drives the performance of the whole system.
- The **execution time** of an I/O component. The amount of time it takes to process all of the data it is provided with.
- The **response time** or *delay* time of an I/O component. It is the amount of time between a request to process data and the time the actual component begins processing.

In order to accurately determine the type of performance to measure, the benchmark has to match how the I/O functions within the system. If the board will be accessing and processing several larger stored data files, benchmarks will be needed to measure the *throughput* between memory and secondary/tertiary storage medium. If the access is to files that are very small, then *response time* is the critical performance measure, since execution times would be very fast for small files, and the I/O rate would depend on the number of storage accesses per second, including delays. In the end, the performance measured would need to reflect how the system would actually be used, in order for any benchmark to be of use.

12.1.5 System Boot-Up

With the development tools ready to go, and either a reference board or development board connected to the development host, it is time to start up the system and see what happens. System boot-up means that some type of power-on or reset source, such as an internal/external hard reset (i.e., generated by a check-stop error, the software watchdog, a loss of lock by the PLL, debugger, etc.) or an internal/external soft reset (i.e., generated by a debugger, application code, etc.), has occurred. When power is applied to an embedded board (because of a reset), start-up code, also referred to as **boot code**, **bootloader**, **bootstrap** code, or **BIOS** (basic input output system) depending on the architecture, in the system's ROM is loaded and executed by the master processor. Some embedded (master) architectures have an internal program counter that is automatically configured with an address in ROM in which the start of the boot-up code (or table) is located, while others are hardware wired to start executing at a specific location in memory.

Boot code differs in length and functionality depending on where in the development cycle the board is, as well as the components of the actual platform that need initialization. The same (minimal) general functions are performed by boot code across the various platforms, which are basically initializing the hardware, which includes disabling interrupts, initializing buses, setting the master and slave processors in a specific state, and initializing memory. This first hardware initialization portion of boot-up code is essentially the executing of the initialization device drivers, as discussed in Chapter 8. How initialization is actually done—that is, the order in which drivers are executed—is typically outlined by the master architecture documentation or in documentation provided by the manufacturers of the board. After the hardware initialization sequence, executed via initialization device drivers, the remaining system software, if any, is then initialized. This additional code may exist in ROM, for a system that is being shipped out of the factory, or loaded from an external host platform (see the callout box with `bootcodeExample`).

```
bootcodeExample ()
{
....
// Serial Port Initialization Device Driver
initializeRS232(UART,BAUDRATE,DATA_BITS,STOP_BITS,PARITY);

// Initialize Networking Device Driver
initializeEthernet(IPAddress,Subnet, GatewayIP, ServerIP);

//check for host development system for down loaded file of rest of code to RAM
// through ethernet
// start executing rest of code(i.e. define memory map, load OS, etc.)

...
}
```

MPC823-Based Board Booting Example

The MPC823 processor contains a reset controller that is responsible for responding to all reset sources. The actions taken by the reset controller differ depending on the source of a reset event, but in general the process includes reconfiguring the hardware, and then sampling the data pins or using an internal default constant to determine the initial reset values of system components.

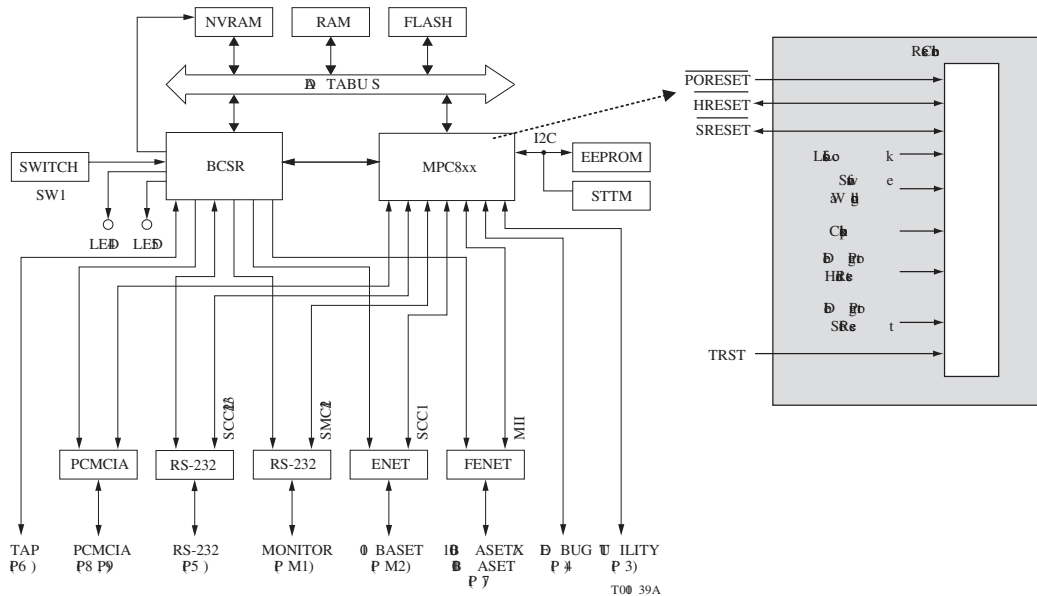


Figure 12-7a: Interpretation diagram [12-3]

Copyright of Freescale Semiconductor, Inc. 2004. Used by permission.

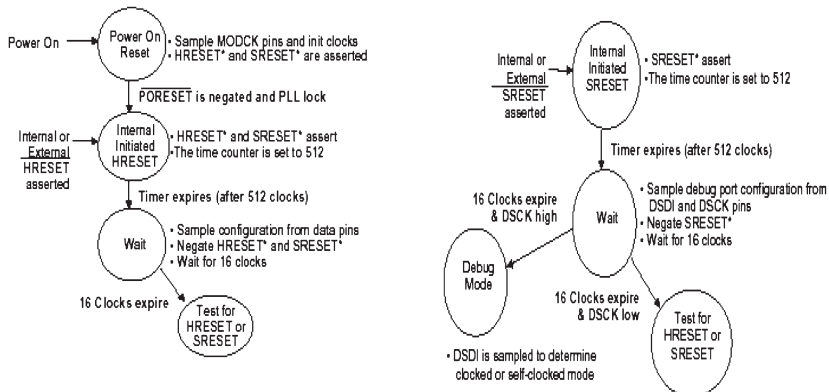


Figure 12-7b: Interpretation diagram [12-3]

The Final Phases of Embedded Design: Implementation and Testing

The data pins sample represents initial configuration (setup) parameters as shown in Figure 12-7c.

if..		then..		
No external arbitration	SIUMCR.EARB = 0	D0 = 0	D0	
External arbitration	SIUMCR.EARB = 1	D0 = 1		
EVT at 0	MSR.IP = 0	D1 = 1	D1	D0 specifies whether external arbitration or the internal arbiter is to be used.
EVT at 0xFFFF0000	MSR.IP = 1	D1 = 0		
Do not activate memory controller	BR0.V = 0	D3 = 1	D3	D1 controls the initial location of the exception vector table, and the IP bit in the machine state register is set accordingly.
Enable CS0	BR0.V = 1	D3 = 0		
Boot port size is 32	BR0.PS = 00	D4 = 0, D5 = 0	D4 D5	D3 specifies whether Chip Select 0 is active on reset.
Boot port size is 8	BR0.PS = 01	D4 = 0, D5 = 1		
Boot port size is 16	BR0.PS = 10	D4 = 1, D5 = 0	D7 D8	If Chip Select 0 is active on reset pins D4 and D5 specify the port size of the boot ROM, with a choice of 8, 16, or 32 bits.
Reserved	BR0.PS = 11	D4 = 1, D5 = 1		
DPR at 0	immr = 0000xxxx	D7 = 0, D8 = 0	D7 D8	D7 and D8 specify the initial value for the IMMR registers. There are four different possible locations for the internal memory map.
DPR at 0x00F00000	immr = 00F0xxxx	D7 = 0, D8 = 1		
DPR at 0xFF000000	immr = FF00xxxx	D7 = 1, D8 = 0	D9 D10	D9 and D10 select the configuration for the debug pins.
DPR at 0xFFFF0000	immr = FFF0xxxx	D7 = 1, D8 = 1		
Select PCMCIA functions, Port B	SIUMCR.DBGC = 0	D9 = 0, D10 = 0	D9 D10	D11 and D12 select the configuration of the debug port pins. This selection involves configuring these pins either as JTAG pins, or development support communication pins.
Select Development Support functions	SIUMCR.DBGC = 1	D9 = 0, D10 = 1		
Reserved	SIUMCR.DBGC = 2	D9 = 1, D10 = 0	D11 D12	D13 and D14 determine which clock scheme is in use; one clock scheme implements GCLK2 divided by one, and the second implements GCLK2 divided by two.
Select program tracking functions	SIUMCR.DBGC = 3	D9 = 1, D10 = 1		
Select as in DBGC + Dev. Supp. comm pins	SIUMCR.DBPC = 0	D11 = 0, D12 = 0	D11 D12	
Select as in DBGC + JTAG pins	SIUMCR.DBPC = 1	D11 = 0, D12 = 1		
Reserved	SIUMCR.DBPC = 2	D11 = 1, D12 = 0	D13 D14	
Select Dev. Supp. comm and JTAG pins	SIUMCR.DBPC = 3	D11 = 1, D12 = 1		
CLKOUT is GCLK2 divided by 1	SCCR.EBDF = 0	D13 = 0, D14 = 0	D13 D14	
CLKOUT is GCLK2 divided by 2	SCCR.EBDF = 1	D13 = 0, D14 = 1		
Reserved	SCCR.EBDF = 2	D13 = 1, D14 = 0		
Reserved	SCCR.EBDF = 3	D13 = 1, D14 = 1		

Figure 12-7c: Interpretation diagram ^[12-3]

The Embedded Planet RPXLite Board assumes that onboard ROM (FLASH) contains the bootloader monitor/program, called PlanetCore, originally created by Embedded Planet. The PowerPC processor and on-board memory start up in a default configuration set via hardware (CS0 is an output pin that can be configured to be the global chip select for the boot device, HRESET/SRESET, data pins,...) , and has no dedicated accessible PC register.

Chip Select	Port Size	Function/Address	Comment
CS0#	x32	FLASH (x32) FFFF FFFF minus actual FLASH size	Reset vector at IP = 1: 0000 0100 Vector set at IP = 1 in hardware BRO set at FFFF minus FLASH size 2, 4, 8, or 16 Mbytes
CS1#	x32	SDRAM (x32) 0000	16, 32, or 64 Mbytes
CS2#		Expansion Header UUUU	Routed to expansion receptacle
CS3#	x32	Control and Status Registers FA00	Byte and/or word accessible
CS4#	x8	NVRAM/RTC or SRAM/RTC FA00	0K, 32K, 128K, or 512 Kbytes Also available at Expansion Receptacle
CS5#		Expansion Header UUUU	Routed to expansion receptacle
CS6#	x16 or U	PCMCIA Slot B Chip Select Even Bytes or Chip Select 6 to I/O Header UUUU	OP2 in MPC850 PCMCIA control register selects mode: L = PCMCIA Slot B enabled H = CS6# to expansion header enabled
CS7#	x16 or U	PCMCIA Slot B Chip Select Odd Bytes or Chip Select 7 to I/O Header UUUU	OP2 in MPC850 PCMCIA control register selects mode: L = PCMCIA Slot B enabled H = CS7# to I/O expansion header enabled
IMMR	x32	Value at reset = FF00 0000, then set to FA20 0000	

Figure 12-7d: Interpretation diagram ^[12-3]