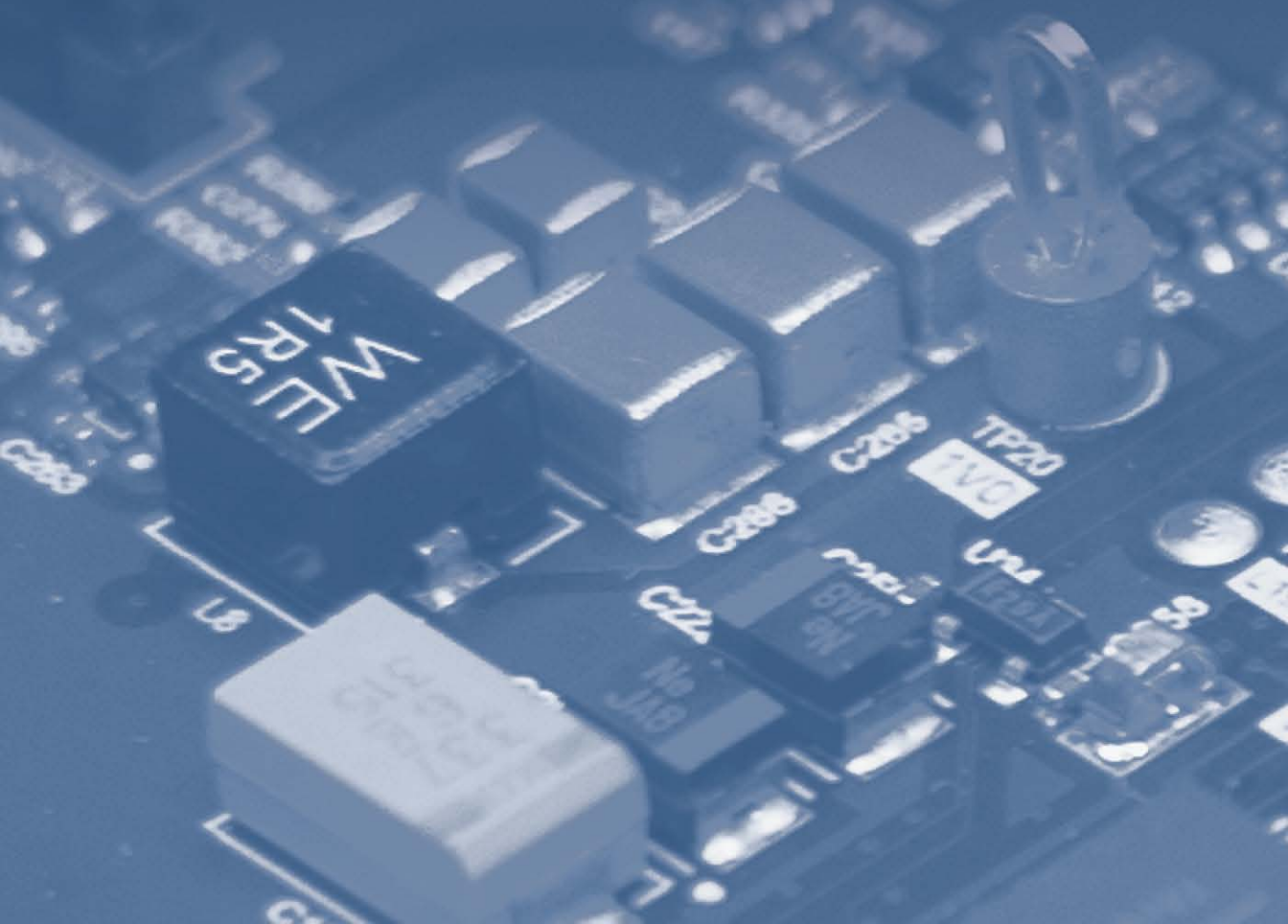
A HANDS-ON GUIDE TO

# DESIGNING EMBEDDED SYSTEMS

ADAM TAYLOR · DAN BINNUN
SAKET SRIVASTAVA



# A Hands-On Guide to Designing Embedded Systems

For a listing of recent titles in the
*Artech House Integrated Microsystems Library*,
turn to the back of this book.

# A Hands-On Guide to Designing Embedded Systems

Adam Taylor
Dan Binnun
Saket Srivastava

ARTECH
HOUSE
BOSTON | LONDON
artechhouse.com

**Library of Congress Cataloging-in-Publication Data**
A catalog record for this book is available from the U.S. Library of Congress.

**British Library Cataloguing in Publication Data**
A catalog record for this book is available from the British Library.

ISBN-13: 978-1-63081-683-4

**Cover design by Charlene Stevens**


**685 Canton Street**
**Norwood, MA 02062**



10 9 8 7 6 5 4 3 2 1

# Contents

# Acknowledgments

I have been truly lucky in my career to be able to work with some amazing engineers and companies on a range of exciting projects, many of which are literally out of this world. In recent years, I have been extremely fortunate to be able to run my own engineering consultancy focusing on embedded system development. The opportunity to run my own business came from a blog that I wrote called the MicroZed Chronicles (www.microzedchronicles) over 8 years. Now I write weekly blogs on embedded system development.

However, I have often wanted to go a little further than allowed in the blog or series of blogs and really walk new and younger engineers through the life cycle of an engineering project from concept to delivery. Hopefully, this book will act as a guide and aide to those who are starting out in what I consider to be one of the most rewarding careers there is. Unlike many engineering texts, my coauthors and I have set out to tell the story of an embedded development project, in parallel with the writing of the book, that we have designed, manufactured, and commissioned for the board.

All design information will be made available, along with additional technical content, on the book's website, looking further into deep technical areas than several books possibly could.

The creation of a book is a challenge and there are several people to thank, most especially my coauthors Dan and Saket. It has been truly a pleasure to work on the creation of this book with such talented engineers.

I would like to thank Steve Leibson, Mike Santarini, and Max Maxfield, who first published my articles; Rebecca To and the wider Xilinx community, who have been so supportive over the years as I work with their products to create blogs, projects, and real applications for clients; and the Avnet team of Kristine Hatfield, Bryan Fletcher, Kevin Keryk, Fred Kellerman, Dan Rozwood, and the wider Avnet/Hackster and Element 14 communities.

No man is an island, and I must thank my wife and son Jo and Dan for understanding the long hours that I spend both working for clients and creating content.

*—Adam Taylor*

It's with genuine humbleness that I write these acknowledgments, as I never thought I would be writing a book on how to do this kind of work. This is a professional accomplishment that I am very proud of and I would not be writing these words without the help of many people.

First and foremost, I want to thank my family for pushing me to go to college and pursue engineering. As a young man I didn't really know what I wanted to do—and with some pushing and prodding from my family, I ended up in a career that I truly find engaging and interesting. For that alone I consider myself very fortunate.

I also want to thank my wife, Katie. She gave me the courage to jump headfirst (after years of merely sticking a toe in the water) into running my own consulting company, E3 Designers. Without her push, I would not have taken the plunge mostly out of fear of failure. She believed that I could do it and be successful before I did.

In the context of this book, I have to start by thanking my coauthor Adam Taylor. Adam approached me about joining him as a coauthor after we had worked together on a project for a customer that he introduced me to. Without the invitation, I would certainly not have gone down the path of writing a book on engineering, trying to share the lessons I have been fortunate enough to be given by my more seasoned peers over my career.

Along the way I had a lot of help from wonderful people and companies in creating the design to support our story, I'd like to thank them all as well.

My colleagues Brandon Smith and Dave Chisholm, whom I consider among the best I've worked with in my career, helped an incredible amount with the PCB design, reviews, and layout. I truly couldn't have done it without their help and skillset. I have been fortunate enough to work with them professionally and consider them both friends as well. Their technical expertise and high standards are always valued, and I lean on them frequently in my consulting ventures as well.

The companies that helped me throughout this process also deserve a heap of thanks—Advanced Assembly, Altium, Analog Devices, Ansys, FLIR Systems, Mentor Graphics, Sparx Systems, and Trilogic all personally worked with my in some form or fashion in this process. I am grateful for all they did because none of it was owed to me, but was given out of their good will.

Lastly, I do want to thank my first mentor in this industry, Patrick Joyce. Pat was a senior engineer in the engineering team at my first job in Keene, NH at Markem (which would become Markem-Imaje shortly after I joined). Not only was he an extremely talented engineer and designer, but he dedicated time to teaching all of us in the group as well. It was because Pat was generous with his knowledge that I was able to grow professionally. My hope is that over the course of my career I can do that for others and pay that generosity forward.

*—Dan Binnun*

As an academic, it was always my dream to write a book that made a real difference in the life of everyday engineers. Writing a technical paper is bread and butter for academics, but the audience for technical papers is often limited to a particular research area and hardly reaches the general public. So when my dear friend Adam Taylor approached me to coauthor a book on this topic, my first question to

him was what was going to be the flavor of this book, as it was always my desire to write something that can help the readers not only solve real-life challenges but also is easy enough to understand and follow. When we had our preliminary discussions, it was exciting to see that all of us were aiming for the same goals, which really motivated me to join the team. That was the easy part; writing a book from scratch and, most importantly, making sure that it is technically sound but still easy enough to read and follow are the most challenging part. During the journey of writing this book, we all learned a lot from each other. Having never worked in the industry, personally it was enlightening for me to see the point of view of my coauthors, both of whom have extensive knowledge of what embedded design engineers in the industry are really like.

I would like to thank my wife, parents, and other family members, who have been encouraging me throughout my life to achieve higher academic goals and to write a book for as long as I could remember. I especially thank my wife Nupur, who helped me keep my focus even during the trying times of the Covid-19 pandemic, when major portions of this book were written and did not let me lose track when I was getting lost without new ideas.

I again thank my wonderful coauthors for the patience that they showed throughout this journey, even when I was missing deadlines, and to Artech House, for keeping faith in this project despite the delays. Without the understanding and commitment of the entire team, this book project may not have been successful.

*—Saket Srivastava*

# Introduction

Designing products is hard. A product is born out of a series of smaller developments done with a common goal in mind, and each of those smaller developments on their own is incredibly challenging.

Whether you are designing packaging for a product that needs to ship around the world in a cargo plane or shipping container or designing the mechanical enclosure of the product that goes inside that packaging, you are going to be faced with challenges from the very first day of the project.

When we decided to write this book on designing embedded systems, we did so because we felt that there was a need to address the challenges involved in designing products with a more holistic view than is often presented in technical books and publications.

There is certainly no shortage of wonderful references on the nitty-gritty technical information and nuance involved in electronics, firmware, software, systems, and reliability engineering; we reference many of them in this book. What we felt was missing was a framework for people to use that tries to explain how a product begins with an idea; that idea then becomes a set of requirements, and those requirements evolve into a focused product. Contrasted against an evaluation kit that can be overwhelmed with peripherals and features, a true embedded system is often a bare-bones collection of hardware required to perform a very specific set of tasks. Adding in superfluous interfaces, functionality, or features can cause further complexity to be introduced. This almost always results in additional technical risks and cost risks and, perhaps most significantly, can extend development timelines to the point of being untenable for a company to deal with.

The goal of this book is to illustrate the development cycle more holistically to provide better context as to why certain features are present on certain systems. Understanding this process holistically is critical to being a productive embedded systems engineer and technical contributor.

That is not to say that the why is more or less important than the how, merely that we wanted to bring the why into the picture as well.

In trying to come up with a concept for how to show this process in a realistic light, we decided that following the journey of a team of imaginary engineers at an imaginary company (SensorsThink) as they developed a new platform would be a fresh take on what can be a stale topic.

In addition to following along with this team of imaginary engineers, we also decided to practice what we preached and actually design the heart of the system to be available as reference material.

We started with a set of basic requirements, created a system architecture, and on down the line to designing a custom printed circuit board (PCB) and writing firmware, field programmable gate array (FPGA) code, and application code as well.

Our hope is that, in sharing those design artifacts and finished product in addition to this book, a more complete picture will be available and it will be easier to see how we got from the starting line to the finish line.

Lastly, we intend to make a wealth of other information available for reference as part of a larger ecosystem of information and content: things like simulations, test plans and results, design calculations and notes (as we took them in the course of the design work), and the schematics, layout, and source code.

We sincerely hope that you all find this book and ecosystem around it useful, no matter your experience level or where you are in your career.

CHAPTER 1

# Programmatic and System-Level Considerations

Before we can design, we must have a clear and frozen technical baseline, which allows for progressive assurance and risk retirement.

## 1.1 Introduction: SensorsThink

Another day begins bright and early at SensorsThink, a product design company that specializes in sensing technology solutions for all types and sizes of markets. A typical day for an engineering team member here is similar to a typical day for most engineering team members: chock-full of engineering change orders (ECOs), project status meetings, and other interesting, but mostly turn-the-crank engineering work. Today will be different and more exciting. There is a buzz in the office, a new product platform has been talked about for months, and the day may finally have arrived when management and marketing are ready to unveil the idea that will become the next great SensorsThink product. After months of research and speaking with potential customers, they have concluded that one product platform can be developed to suit the needs of several potentially large and growing markets:

- Smart buildings/smart infrastructure;
- Factory automation;
- Environmental monitoring.

In situations like this, there will often be a project kick-off meeting to generate some momentum for the work. This may take the form of a company-wide meeting or an email to the project team or engineers may be pulled aside by the project team leader and given an overview. So if you were to find yourself as part of a new product development team, perhaps pulled into a meeting with representatives from management and marketing, what would come next? How would you approach the problem of creating a product from what may be as vague as a bullet-point list of must-have features?

This is a situation in which almost every design engineer will be put at some point in their careers, and it can seem daunting, overwhelming, and intimidating. By the time you have read through this book, the task at hand will seem more

manageable and you will understand how to take that list of features and turn them into aspects of a well-engineered and well-designed product.

## 1.2 Product Development Stages

In order to set the stage for further understanding of what we consider best practices for doing product development work, it is vitally important that you understand the stages (or phases) of product design and development. It is equally important to note that there is a clear distinction between methods used in product design and development (Agile, for example, is an extremely popular method in the software engineering world) and the generalized stages that a development life cycle has. The focus here is not on the methods applied, but rather the stages that should be worked through.

For the purpose of this book, a classic stage-gate style approach will be discussed. However, before exploring that approach, it should be noted that there are many other approaches that are used in industry. These approaches are regularly tailored on a project-by-project basis for any number of reasons: project complexity, project timeline, product volumes, and cost target, for example. Any of these factors can change the approach or tailor the approach that a company uses.

Another key concept to understand is that the length of the entire process, or the length of a stage within the overall process, can vary wildly from project to project. Often, this is directly proportional to product complexity and anticipated product volumes.

A critical takeaway to have is that while the process may be documented in a certain way by your company, it may not always be followed to the letter of the law. Adaptability is a critical characteristic to have in the modern development world with aggressive timelines and aggressive requirements often seeming to be in direct opposition with each other. Remember the spirit of the process, but try not to be paralyzed from making progress by feeling restricted by it.

A classical stage-gate type product development typically consists of several stages of works, followed by checkpoints in the process referred to as gates. It is at these checkpoints where stakeholders in a project determine whether the project and/or product are still viable both from a business and technical perspective. One of several items that is evaluated at these gates is risk. Risk can be from any number of areas: financial risk, market risk, technical risk, and staffing risk, to name a few. In general, the best-performing projects tend to reduce risk early in this stage-gate process so that as you near the product launch date, the remaining risk is minimal, and no major risk items are left for the final hours. This book is not intended to serve as a complete reference for a stage-gate process; there are many excellent resources available both online and in reference texts for learning more about these types of processes and how they can be tailored to a specific company, product, or project.

For the purpose of this development journey, we will assume that SensorsThink has chosen to follow a 5-stage development path, shown as a simple flow diagram in Figure 1.1 [1].

A brief overview of each stage in the process is given here to provide further understanding and context.

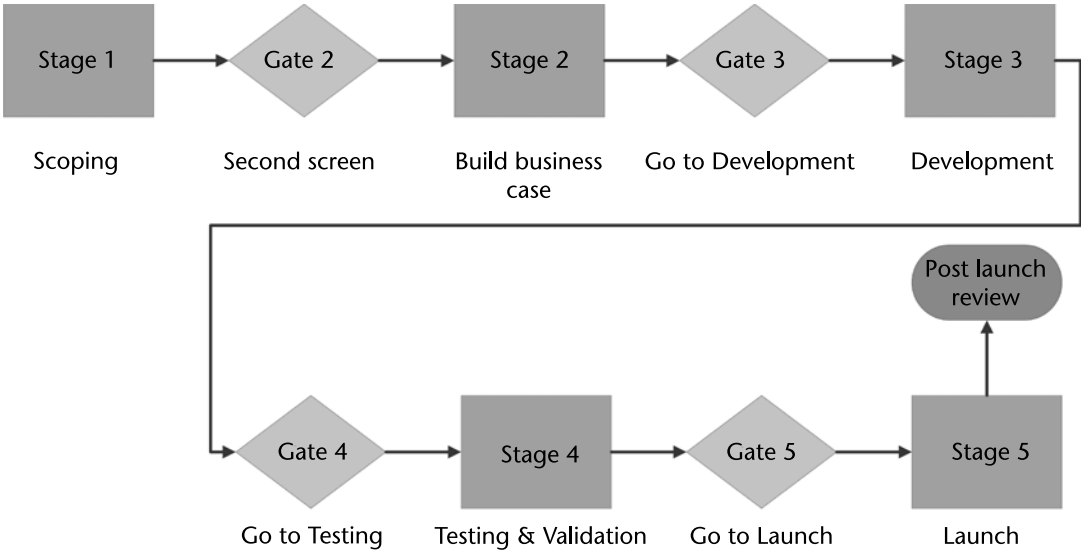



**Figure 1.1** A 5-stage development process.

- *Stage 1:* During this stage of the process, the main goals are essentially to evaluate the product and project from all angles. What size is the market? What is your competition doing? What competition already exists? What will differentiate your product from those of your competitors? Does your company possess the skills, experience, and know-how to deliver?
- *Stage 2:* Building a business case is exactly as it sounds; this is where the company must determine the following:
  - What is the product? What precisely is going to be offered and what technical capabilities will it have? What level of reliability is required by the consumers of the product?
  - What legal and regulatory requirements will be imposed upon the new product prior to it entering the marketplace? In safety-critical applications, the level of rigor and documentation can be quite daunting and is very resource-intensive. Knowing this well ahead of time can go a long way towards having a viable case for developing and launching that new product.
  - What risks are present in the product? This includes business risks, financial risks, and technical risks.
  - What is the plan for the project? This can vary in level of detail greatly, but most often will include key milestones, required staffing, and required financial resources (including any capital expenditures for new equipment or for expensive time at qualification labs at the end of the project).
  - Is this feasible? After spending a great deal of time and energy in analyzing the project and product, the company then decides if it is worth doing. If they feel it is worth doing, stage 3 will occur; this is typically when most design engineers will be involved in the process.

- *Stage 3:* This is the stage in the product development process with which most engineers are familiar. This is the execution of the (hopefully) detailed and rigorous analysis performed in the prior stages of work. This can include deployment of prototypes to customers during the process to get valuable customer feedback on the product as it is being developed. Ultimately, the product development team is tasked with delivering a prototype product ready for the next stage of the process, testing and validation.
- *Stage 4:* Testing and validation take on many forms, depending on in which department you work.
  - As a design engineer doing hardware, you will likely be rigorously testing your hardware against requirements and specifications to ensure performance.
  - As a software developer, you will be checking your code for coverage, perhaps compliance with coding standards, and regression testing releases.
  - In a regulatory and compliance role, you will be testing the product against worldwide standards for things such as electromagnetic interference/electromagnetic compatibility (EMI/EMC) and general product safety. Depending on what type of product you have designed and the markets in which you will be selling your product, there are different harmonized standards and directives against which your product will be evaluated.
  - There is also field testing, perhaps redeploying updated or further refined products to your early adopters from stage 3.
  - Lastly, there is market testing, where a business may tease the product and gauge interest prior to moving to the final stage of development, product launch.
- *Stage 5:* This stage is the culmination of the prior 4 stages of work (there are actually a few additional stages sometimes) to which the company has devoted countless resources. This is a very exciting time for all involved in the development process, from marketing, engineering, sales, and field support and service. This is also a very intense time for all involved in the process, with all involved hoping that the product will be successful.

In addition to understanding these 5 stages, it is equally important to understand what an effective gate can do for a development process. A gate must have "teeth," and it must be empowered to determine that a product is no longer feasible and "kill" the development life cycle. It can be for any number of reasons: too much residual risk, key staff departures, and changes in market conditions. This can be a disheartening thing for team members, as months (sometimes years) of hard work and effort are placed on a shelf and sometimes never continued. Healthy companies kill projects; it is part of the natural course of running a business and developing products. As a design engineer (or in whatever role you may have), it is crucial to remember that, if done for the right reasons, this is absolutely a healthy and viable choice for a business to make.

## 1.3 Product Development Stages: Tailoring

Before we proceed further into our journey, let's touch on some of the other aspects of the development process. As mentioned previously, this is not an in-depth guide to the product development process, just an overview. However, some additional discussion on the topic is useful to provide a broader understanding for anyone not familiar with a stage-gate process. The other aspects that are perhaps most important to touch on are:

- Ways that this process can be tailored based on project complexity;
- Ways that ideas can be fed into the process and turned into development projects.

A full 5-stage development process is not always necessary, even when considering the intent of always doing best-practice engineering development. This can be true if a product development is starting from ground zero. For example, consider the case of a value engineering (also known as cost reduction) or a customizable product family that may need a minor modification to complete a sale, sometimes referred to as a design-in-win. These are very different cases from the new idea to new product development cycle, and as such, the process can be tailored accordingly. This does not mean that steps should be skipped randomly and caution thrown to the wind. Generally speaking, the same deliverables are going to be required, but stages can be combined. We will examine these two cases in a bit more detail.

For the case of a value engineering project, it might be prudent to apply a slightly condensed version of this process; refer to Figure 1.2.

In this example, assume that the changes being made are to change the enclosure paint and hardware of an existing product. These changes may have an impact on performance in terms of regulatory or safety issues, but hopefully no impact to functionality. After scoping the work, the business chooses to go directly to development and test in one step, knowing that they intend to replace the current product offering with the new and lower-cost product. Once the engineering and product management teams are satisfied with the test results of the changed

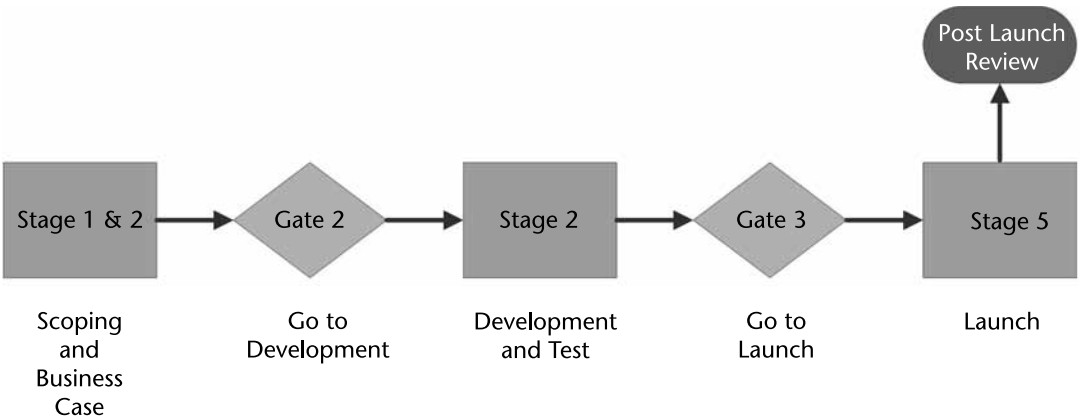



**Figure 1.2** A 5-stage development process condensed to 3 stages.

product (having skipped the gate to proceed to testing), the product simply "cuts in" to production (meaning that, as old inventory is depleted, the new, lower-cost model replaces it).

For the case of a design-in-win scenario, it might be prudent to condense the process even further; refer to Figure 1.3.

In this example, assume that an existing product needs only to have the paint color changed and a prospective customer will purchase 1,000 units. In a case like this, the necessity for complete engineering rigor does not have justification. After a quick scoping of the necessary changes and product management impact, this product can be launched as a new orderable part number for the customer in a rapid fashion. The same deliverables can be required of the team, but most can be recycled from the original product.

Being able to adapt to these conditions and differences in product development undertakings as a design engineer is very important in today's landscape. A truly brand-new product development is far less common than a derivative product; think, for example, of mobile phones or televisions or even cars and other vehicles. Perhaps there are a few new features, but upon close examination, the similarities almost always outweigh the differences.

Lastly, it is important to understand that this process of development must be fed ideas; this idea buffet can be filled from many different places: fundamental research (true research and development (R&D)), internal idea capture from employees, and voice of customer (which can be anything from direct interaction with potential customers to focus groups and even collaborative design efforts with the customer).

In this development adventure on which we are about to embark, we are assuming that this is a true ground-up development project for a new platform. We are also under the assumption that marketing and management have done their homework properly, and this idea has been vetted in terms of the marketplace and in terms of the available skills the company has to deliver upon it. This places the bulk of the content of our journey in what was previously described as stage 3 (development) and stage 4 (testing and validation).

## 1.4 Product Development: After Launch

As a final note, once a product is released, it enters a new life cycle of sorts, often divided into four distinct stages:

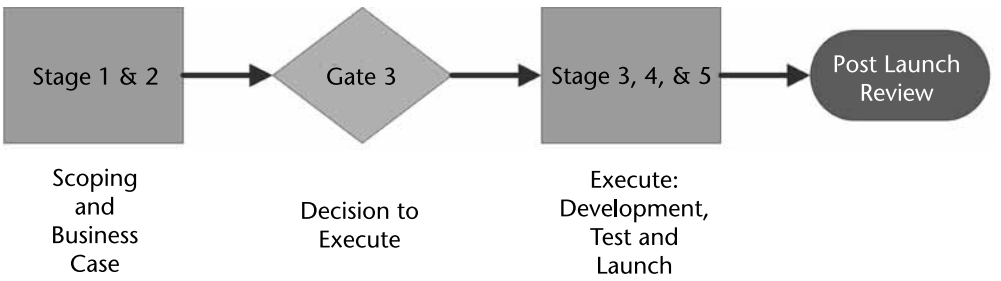



**Figure 1.3** A 5-stage development process condensed to 2 stages.

- *Product introduction:* This is where the development process that we have just discussed ends, and the product is introduced to the marketplace and is ready to be sold.
- *Growth stage:* This is hopefully where the product starts to sell better as the hard work and groundwork laid by marketing and sales starts to translate into sales. Demand and distribution hopefully both increase, and the product becomes a profitable one for the business.
- *Maturity stage:* At this point, the product may be considered to be in a maintenance phase; perhaps there are fewer sales because the market has become saturated or competition has increased. A business may opt for a value engineering project here to really try to squeeze the value out of the product, knowing it needs to be maintained for existing customers.
- *Product decline:* This is the point in time when the business case for support of the product no longer stands up to critique and analysis. It is time to end the product's active life and hopefully introduce its successor. This is also referred to as the end of life (EOL) or obsolescence stage.

Throughout this post-launch product life cycle, engineering is still needed to support products in the field, perhaps fix any bugs or defects that escaped initial launch, or even introduce new features. The launch of a new product is certainly not where the engineering work ends; the difference is in the mindset of the engineers and other contributors and the context of the work being done.

This hopefully has oriented us as we begin to follow the journey of SensorsThink as they start the development of their new platform and give some context as to where this part of the journey sits in the overall life cycle of both a design project and a product's own life cycle. What comes next is understanding precisely what must be delivered and capturing that in a clear and concise manner. Remember our key message: before we can design, we must have a clear and frozen technical baseline, which allows for progressive assurance and risk retirement.

## 1.5 Requirements

At this point in our journey, SensorsThink has announced their intention to develop and launch a new product platform; they have done their homework through the first two stages of a typical 5-stage development process. Marketing and management sit the broader engineering team down in a room (yourself included) and present a set of requirements to help the team start to better comprehend the task at hand (see Figure 1.4).

Figure 1.4 is a Systems Modeling Language (SysML) requirements diagram, meant to be a visual representation of the high-level requirements of the new product platform; it does not contain textual information when presented this way, but that textual information can be viewed in a number of ways depending on the systems engineering toolset that is in use. For SensorsThink, Enterprise Architect 14 (created by Sparx Systems) [2] is the in-house tool of choice. The diagram presented in Figure 1.4 is an abstracted view of the full set of textual requirements through which we will be working during our development process. (As an aside,

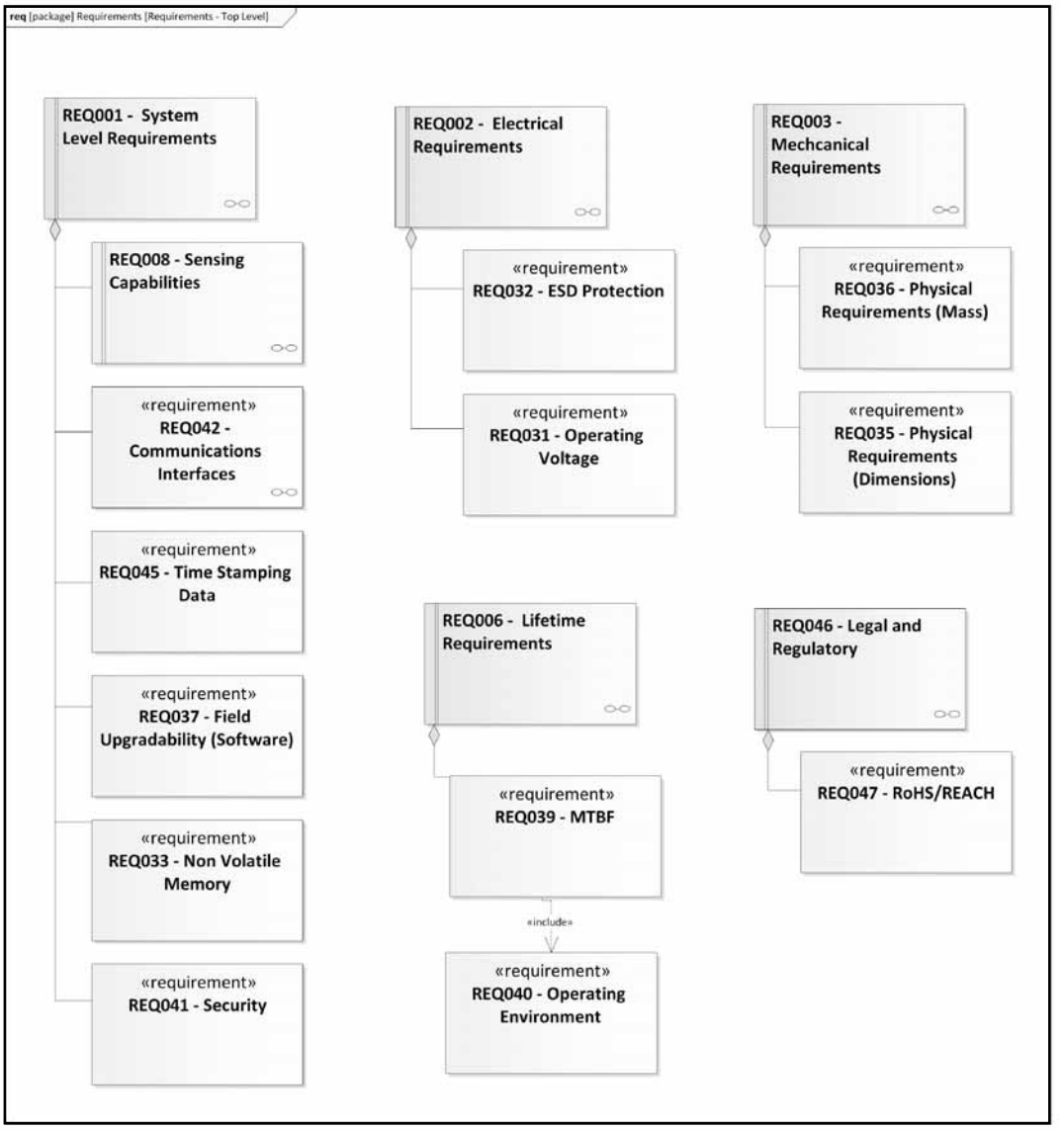



**Figure 1.4** Requirements tree as presented by marketing and management.

if marketing and management ever present something that looks like this, a celebration is in order. Starting out with requirements entered in a modeling tool is a fantastic starting point and often is not the norm.)

A modeling tool that supports SysML is extremely useful during product development. It makes the application of many modern engineering best practices much easier and, in some cases, may be the only way to implement them in a satisfactory way for a critical or safety relevant system. Some of these best practices supported by SysML tools such as Enterprise Architect include:

- The ability to trace multiple SysML element types to and from each other; requirements to other requirements, requirements to test cases, requirements to use cases, and requirements to design elements.
- The ability to monitor the status of a requirement (or any other element) as reviewed or approved with built-in attributes for modeled elements. This can be vitally important in a safety-critical system environment. Even better is the ability to make these things visible in diagrams.
- Traceability through the use of relationship matrices to view exiting traceability between element types and identify gaps in traceability. Refer to Figure 1.5.
- Checking for unrealized requirements; as an example, consider a large system with thousands of requirements. Manually checking that each requirement

| | Requirements::Checklist | Requirements::REQ001 - Syster | Requirements::REQ002 - Electri | Requirements::REQ003 - Mech | Requirements::REQ006 - Lifetin | Requirements::REQ007 - Introdi | Requirements::REQ008 - Sensin | Requirements::REQ011 - Infrare | Requirements::REQ012 - Infrare | Requirements::REQ013 - Infrare |
|---|---|---|---|---|---|---|---|---|---|---|
| Requirements::REQ027 - Local Temperature Sensing Ra... | | | | | | | | | | |
| Requirements::REQ029 - Local %RH Sensing Range | | | | | | | | | | |
| Requirements::REQ030 - Remote %RH Sensing Range | | | | | | | | | | |
| Requirements::REQ031 - Operating Voltage | | | ↑ | | | | | | | |
| Requirements::REQ032 - ESD Protection | | | ↑ | | | | | | | |
| Requirements::REQ033 - Non Volatile Memory | | ↑ | | | | | | | | |
| Requirements::REQ034 - Non Volatile Memory Contents | | | | | | | | | | |
| Requirements::REQ035 - Physical Requirements (Dimen... | | | | ↑ | | | | | | |
| Requirements::REQ036 - Physical Requirements (Mass) | | | | ↑ | | | | | | |
| Requirements::REQ037 - Field Upgradability (Software) | | ↑ | | | | | | | | |
| Requirements::REQ039 - MTBF | | | | | ↑ | | | | | |
| Requirements::REQ040 - Operating Environment | | | | | | | | | | |
| Requirements::REQ041 - Security | | ↑ | | | | | | | | |
| Requirements::REQ042 - Communications Interfaces | | ↑ | | | | | | | | |
| Requirements::REQ043 - Ethernet | | | | | | | | | | |
| Requirements::REQ044 - Bluetooth | | | | | | | | | | |
| Requirements::REQ045 - Time Stamping Data | | ↑ | | | | | | | | |
| Requirements::REQ046 - Legal and Regulatory | | | | | | | | | | |
| Requirements::REQ047 - RoHS/REACH | | | | | | | | | | |
| Requirements::REQ049 - Remote Temperature Sensing ... | | | | | | | | | | |

**Figure 1.5** Example of a traceability matrix for SensorsThink.

has been realized through a use case would be extremely cumbersome. Model validation in Enterprise Architect makes this task much more manageable.

These are just a few examples of the kinds of tasks that must be considered and properly managed throughout the development process and through a product's life cycle.

It is important to note that what is shown here is not intended to represent a full set of product requirements, if you are familiar with the requirements development process, you probably already know that a product of even moderate complexity can have thousands of requirements. A full elicitation and decomposition of requirements is not the focus of this journey. The requirements that are shown here will be addressed throughout the later parts of this text, so that the full process of requirement creation to requirement validation can be shown.

### 1.5.1 The V-Model

Before we dive any deeper into the requirements for our new product, a discussion on the V-model and what constitutes properly written requirements is needed in order to provide context and clarity for the next steps and how the pieces fit together.

Shown in Figure 1.6 is a diagram with which hopefully most readers are familiar. This development approach is a tried-and-true method for developing robust products and systems. The model emphasizes requirement-driven design and test. In the modern development, there are certainly detractors of this approach; many of these detractors are from the software engineering domain, and their objections to use of the model are certainly valid for their work. Some of the more common objections can be [3]:

- The model is too simple to reflect the realities of software development and is more suited towards management than developers or users.
- It is inflexible and has no inherent capability to respond well to changes.

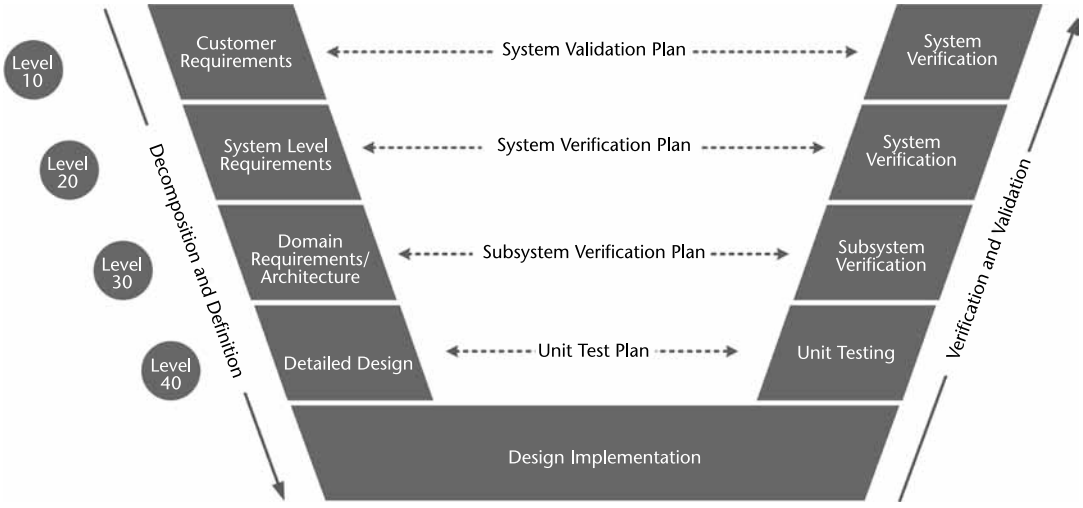



**Figure 1.6** The classical V-model.

- The approach often forces testing to be fit into a small window at the end of the development cycle as earlier stages take longer than anticipated, but the launch date remains fixed [4].

That said, there is certainly merit to the use of the V-model. It has been widely adopted by both the German government and U.S. government. The model provides assistance on how to implement an activity and the steps to take during that implementation. It also provides instructions and recommendations for those activities. Starting with the V-model (or any established model) is almost always better than trying to reinvent the wheel. A framework like this can be tailored and adapted to a company's culture or a specific project's needs quite easily, and most mature models allow for such tailoring, providing guidance on how to approach that task.

Taking a cursory look at the V-model, it is evident that there are both different sides and different levels, with links traversing the model. It is important to note that from a purely theoretical perspective, as you move down the left side, across the bottom, and up the right side of the model, you are always moving forward in time. This makes the V-model an extension of sorts of waterfall product development and project management.

#### 1.5.1.1 The Left Side: Decomposition and Definition (D&D)

The left side of the V is the D&D side of the model. During this stage of the project, customer requirements (which we will refer to as level 10) are decomposed into system-level requirements (which we will refer to as level 20), which are then decomposed into domain requirements and architecture (which we will refer to as level 30). For SensorsThink's project, domains are areas of engineering: electrical, mechanical, firmware, software, testing, and reliability.

As requirements are decomposed, they are always traced back through the levels such that they can point directly to a customer requirement that they fulfill or address.

Once level 30 is reached, the design implementation can begin, with execution on a firm set of requirements now the task at hand, the bottom of the V. Any documentation about the design implementation (engineering calculations, theory of operation-type documents for hardware, firmware, and software, for example) can be considered to be at level 40. These documents and calculations serve an important purpose in the overall life cycle of the product and should not be forgotten during the development process. They are often the basis for knowledge transfer internally and can be key artifacts for critical or safety-related systems.

#### 1.5.1.2 The Right Side: Verification and Validation (V&V)

The right side of the V is the V&V side of the model. During this stage of the project, the physical hardware and written code are tested against ascending levels of requirements. The further the ascension up the V, the more integrated the system is becoming. The first tests performed at level 40 are often deemed unit tests where individual components of software and hardware are tested.

As an example for software or firmware, consider a program that consists of several mathematical functions that ultimately integrate into a calculator. The developer would unit test the addition function, subtraction function, and so on, individually, prior to attempting to integrate all functions into a larger framework.

As an example for hardware, consider a printed circuit board (PCB) that has a microprocessor, a power supply, and a serial port. The designer would verify that the power supply performed as expected (hopefully against some simulation data and calculations performed on the left side of the V) in a separate set of tests as the serial port. This is to ensure that a solid hardware foundation is in place for the software to execute on.

Level 30 testing is typically testing subsystems of the product and may be the first time that software and hardware are integrated. This may also be integration of multiple pieces of hardware and multiple pieces of software that were previously running independently.

As an example, consider a case where developers have been working on evaluation kits for various components of the customized hardware. At this point in time, the application is likely headed for the final target, which may well be a custom PCB.

This stage of testing is also when things such as environmental testing can begin for hardware. With a basic set of functional code and all of the mechanical components assembled with temperature cycle testing, shock and vibration testing and other similar tests can begin. These tests often take many hours to run to completion, so an early start can be a good decision if the design is mature enough.

Level 20 testing is typically the system verification stage, when a fully implemented and integrated system is tested in situ. So rather than testing on the bench or in a controlled environment like an environmental chamber, the product goes into the end-use environment or into simulated end-use environments. Valuable system test data is gathered here, verifying the level 20 requirements.

As an example, consider the custom PCB with a verified serial port. Code has now been ported to this PCB to run a calculator via the serial port, and a test loop has been written to perform random mathematical operations on random numbers and verify the results. This test loop can run independently for weeks on end if needed and the results can be examined for any issues occurring due to varying the environment of the device under test (DUT).

Level 10 testing is the last stop on the right side of the V; the original level 10 customer requirements are validated here, sometimes against the customer's own test cases. System certification for safety, EMI/EMC, restriction of hazardous substances and registration (RoHS) and evaluation, authorization, and restriction of chemicals (REACH) (which are environmental regulations) also happen during this stage of validation. If all testing goes well, the product is ready to launch, having been put through rigorous testing and proven to meet all requirements at all levels of decomposition.

### 1.5.2 SensorsThink: Product Requirements

If we now jump back into SensorsThink's development path, we can see that we are firmly on the left side of the V-model and are sitting at level 20, perhaps a bit more decomposed than a true set of level 10 customer requirements. Let's now take

a look at the textual requirements themselves, rather than the pure diagrammatical view presented earlier. Since this is a sensing platform, let's look closely at the required sensing technologies and associated requirements as identified by marketing and management.

If we refer to the higher-level elements of the diagrams from Enterprise Architect 14, we can see that the basic sensing technology requirements are:

1. Temperature and humidity;
2. Accelerometer;
3. Magnetic field;
4. Infrared spectrum;
5. Vibration.

These are the key functional elements for the new sensing platform that SensorsThink will be attempting to integrate into a single product platform. It is not uncommon for the starting point of a project to be nothing more than a bullet-point list such as the one above, but in our case, we are in much better shape. Our management and marketing teams have gone above and beyond, putting in some details around the requirements for us to help engineering better understand the system and engineer the right product to meet the market demands. Required ranges have been provided for all sensor technologies:

- Degrees Celsius for temperature and percent relative humidity (%RH);
- Two types of acceleration with minimum required values of acceleration that must be measurable;
- Required axes of measurement and required field strength that must be measurable for the magnetic field;
- Required field of view, frame rate, and even wavelength specifications for the infrared spectrum;
- Required axes of measurement and required force and frequency that must be measurable for vibration.

We also see a nice-to-have feature request, which tells engineering that, if possible (after proper analysis and architecture), multiple sensors should be colocated or all sensors should be on a single sensing head. This is a preferred (but not required) solution for the platform. Note that this is not a requirement element, but rather an informational statement.

This type of definition is a wonderful starting point; if we refer back to the V-model, we can see how this can easily be decomposed into domain requirements for hardware and software capabilities and even lends itself nicely to the creation of accurate level 10 validation testing.

### 1.5.3 Creating Useful Requirements

Before we proceed further and show the rest of the product requirements with which we have been presented, it is certainly worth discussing what makes a proper requirement and how we can identify requirements that are weaker, or in need of

refinement, before they are actionable by engineering (or other disciplines, for that matter) [5].

The International Council on Systems Engineering (INCOSE) is an organization founded to develop and propagate the principles and practices that enable the creation of successful systems. The organization was formed in 1990 and has become the de facto standard for many aspects of systems engineering, including requirements writing. The *INCOSE Guide for Writing Requirements* [6] is a document that is often cited in company policy for how requirements are to be constructed. The document is comprehensive and will be heavily cited here as it is an excellent reference for all those tasked with creating, reviewing, designing to, or testing against requirements.

The document is specifically about how a requirement is properly expressed in a textual format, in other words: how to properly write a requirement. It is not a guide on how one might go about gathering requirements or turning those requirements into SysML diagrams. It is important to understand that, while often done by the same people, the tasks are quite different.

The general approach of the guide is to present characteristics and practical rules for crafting requirements. The characteristics express why a rule makes sense, and the rules express how to write the requirements. This helps to provide readers with context knowing that rules must often be adapted to fit or suit a certain need. For the purpose of this discussion, the focus will be on the characteristics of a well written requirement statement:

(When reading the following list, start each item with "a well-written requirement is…")

1. Necessary: Every requirement is necessary. This may seem obvious, but consider the points made below:
   (a) If you can remove the requirement and still solve the problem at hand, the requirement is not needed.
   (b) If the intent of the requirement is covered by other requirements, the requirement is not needed.
   (c) There is a cost (both in money and in time) associated with every requirement; unnecessary requirements can lead to additional cost, additional time, and unnecessary risk.
2. Implementation-independent: A requirement states what is required, not how the requirement should be met.
   (a) Being prescriptive can cause other, and perhaps better, implementations to be overlooked entirely.
3. Unambiguous: A requirement statement can only be interpreted in a single way.
   (a) The intent of the requirement must be understood the same way by the writer of the requirement, the designer, and those doing the verification and validation. (All sides of the V must understand the requirement's intent in the same way.)
   (b) Lack of clarity in a requirement can lead to many problems, like the wrong product being designed or a feature being implemented in a way that the customer did not want.

4. Complete: A requirement statement is complete in and of itself.
   (a) This does not preclude a requirement referencing another document (an international standard, for example) or even another requirement. However, the requirement itself should be a complete and proper sentence that does not rely on other requirements for comprehension.
5. Singular: A requirement statement addresses a single thought.
   (a) This characteristic is especially important to the steps that follow the initial gathering of the requirement; decomposition, allocation, and validation all depend on being able to point back to a singular statement.
   (b) Keeping a requirement limited in scope to a single function or constraint is ideal, but oftentimes referring to a diagram is the best way to express complex behavior.
      (i) If a requirement is that a particular state-machine shall be implemented to align with a safety standard, a diagram of that state-machine is a far better way to describe the requirement than a written explanation of each state and transition.
6. Feasible: A requirement statement always expresses something that is inherently feasible.
   (a) Unachievable requirements such as 100% reliability are frustrating for all parties involved (except for perhaps the customer). These requirements are not only likely to cause extensive overengineering, but they are also impossible to validate.
   (b) Oftentimes, these types of requirements are important features; someone asking for 100% reliability is clearly looking for a very robust solution. However, the analysis has not matured enough to be considered actionable by engineering.
7. Verifiable: A requirement statement is verifiable.
   (a) If a requirement statement is not in some way measurable, able to be inspected, or testable, there is no way to know if it has been satisfied.
   (b) A functional requirement typically requires a test and a performance requirement needs to clearly express the quantities and values associated with the performance.
   (c) Think of the four types of verification events when examining a requirement in this way: analysis, inspection, demonstration, and test.
8. Correct: A requirement statement is correct. This one is self-explanatory. Check the values, check the goals of the product, and make sure the requirement is correct.
9. Conforming: A requirement statement conforms to standards selected by the organization.
   (a) This characteristic is important for large projects with many thousands of requirements. Use of consistent language, terms, acronyms, and abbreviations goes a long way to ensuring that requirements are easier to read and review for the stakeholders.

For further information on the proper construction of a complete set of well-written requirements, refer to the INCOSE Organization website (http://www.INCOSE.org), in particular, the *INCOSE Guide for Writing Requirements*.

Let's examine one of the two example requirements shown in Figure 1.10 from the SensorsThink project for its completeness and correctness as a requirement statement. Enterprise Architect has a neat checklist that can be added to individual requirements or groups of requirements so that all aspects of a built-in (or customized) list can be checked by the project team during requirements development.

For example, consider REQ032–ESD Protection: "The system shall be designed to withstand ESD events on all external facing connections. Tested in accordance with the latest version of EN 61000-4-2" [7].

Review it against the list of items in the checklist (which matches the list of characteristics of a well-written requirement). See if you agree that the requirement meets those characteristics and qualifies as a well-written requirement.

### 1.5.4 Requirements: Finishing Up

Now that we understand the way that requirements are used and what makes a well-written requirement, the rest of the requirements will be presented; take some time and examine them, and think of yourself in the position of an engineer at SensorsThink. How would you feel if presented with this set of requirements (see Figures 1.7 through 1.15)? Hopefully with the context and information provided here, your answer will be that you feel ready to architect a system to meet these requirements [8–10].

## 1.6 Architectural Design

Once the requirements are agreed to at the customer level, several key engineering activities are ready to begin:

1. System-level requirements;
2. System-level architecture and design;
3. System validation plan.

System-level requirements are the next step down the decomposition ladder in requirements elicitation; for the SensorsThink product development project example, the given requirements are more of a hybrid level 10/level 20 requirements set. Remember that the key here is that the framework for approaching product design and development for embedded systems is the focus here, not on a fully and completely elicited requirements set.

Done concurrently with system requirements, system architecture is something that is absolutely essential to delivering a well-engineered product and solution that meets the needs of the customer and is neither overengineered nor underengineered. The right product is always the holy grail of development: the right features, the right performance, and the right price.

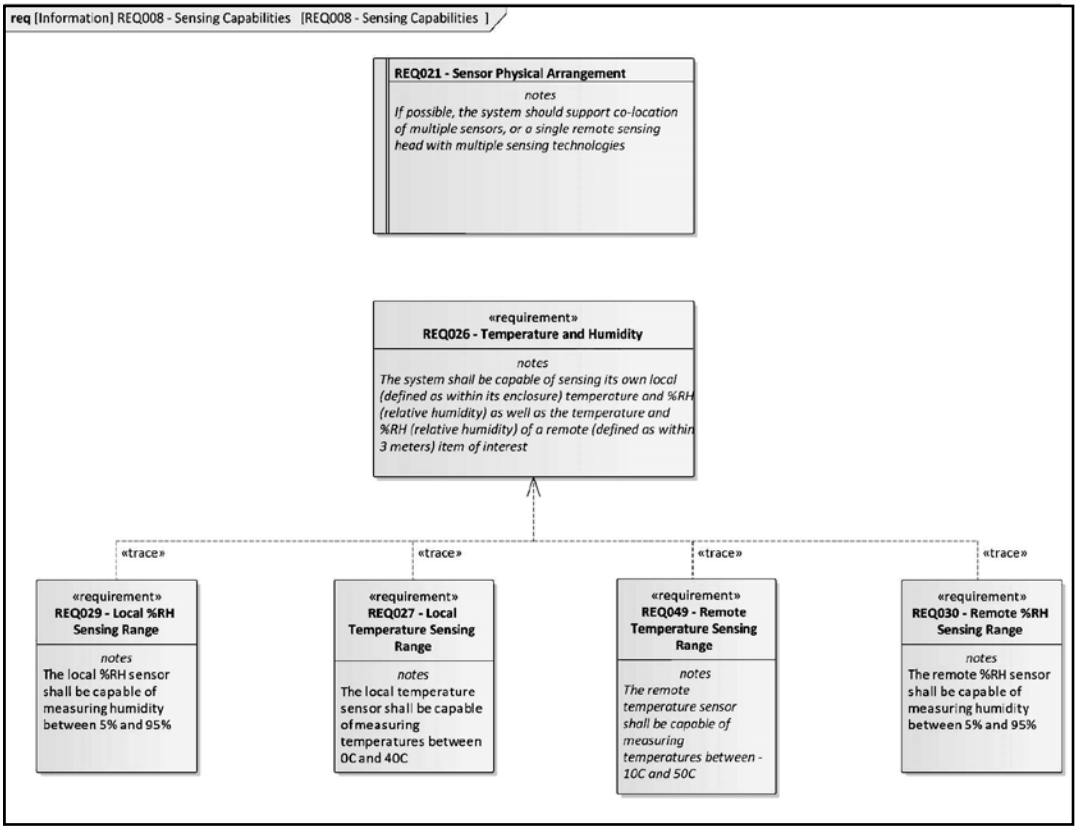



**Figure 1.7** Sensing capabilities: Part 1.

In our journey, we will not be taking a deep dive into modern systems engineering methodology, since it is tangential to our core focus. However, a discussion on the topic is certainly valuable and warranted to provide proper context and understanding.

### 1.6.1 SEBoK: An Invaluable Resource

The Systems Engineering Body of Knowledge (SEBoK) [11] is a curated body of information about systems engineering. It is a wonderful resource for those seeking more information and additional resources about systems engineering.

The SEBoK website (http://sebokwiki.org) provides six key purposes that the organization seeks to support; among them are [12]:

1. "Inform Practice": Inform systems engineers about the boundaries, terminology, and structure of their discipline and point them to useful information needed to practice systems engineering in any application domain.
2. "Inform Interactors": Inform engineers in interacting fields of engineering (mechanical, software, electrical, reliability) and other stakeholders (managers, assemblers, documentation creators) of the nature and value of systems engineering.

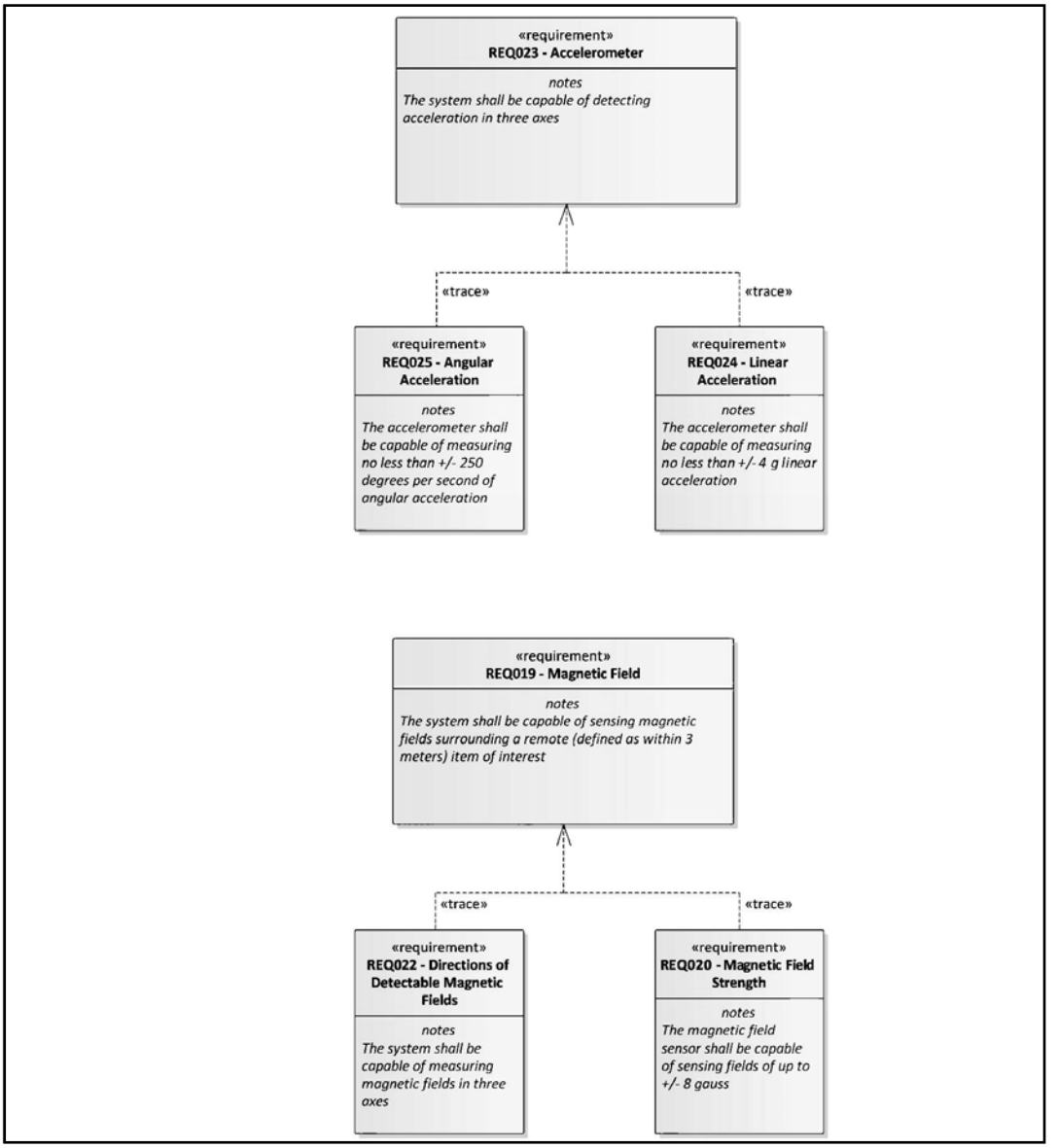



**Figure 1.8** Sensing capabilities: Part 2.

These two purposes are certainly relevant for us and the engineering staff at SensorsThink. Without a solid foundation in place that comes from proper systems engineering, the product being developed is not nearly as likely to be the right product, with the right features, performance, and price.

For technology development, there is a wonderful excerpt from the SEBoK website that should be kept in mind as we architect our new sensor platform [13]:

> Product systems engineering should bring awareness of technology changes and trends to the analysis of new product ideas or innovations. This affects the time and cost inputs into the technical feasibility analysis of the product. The result should include a road map of required technology developments, which is then

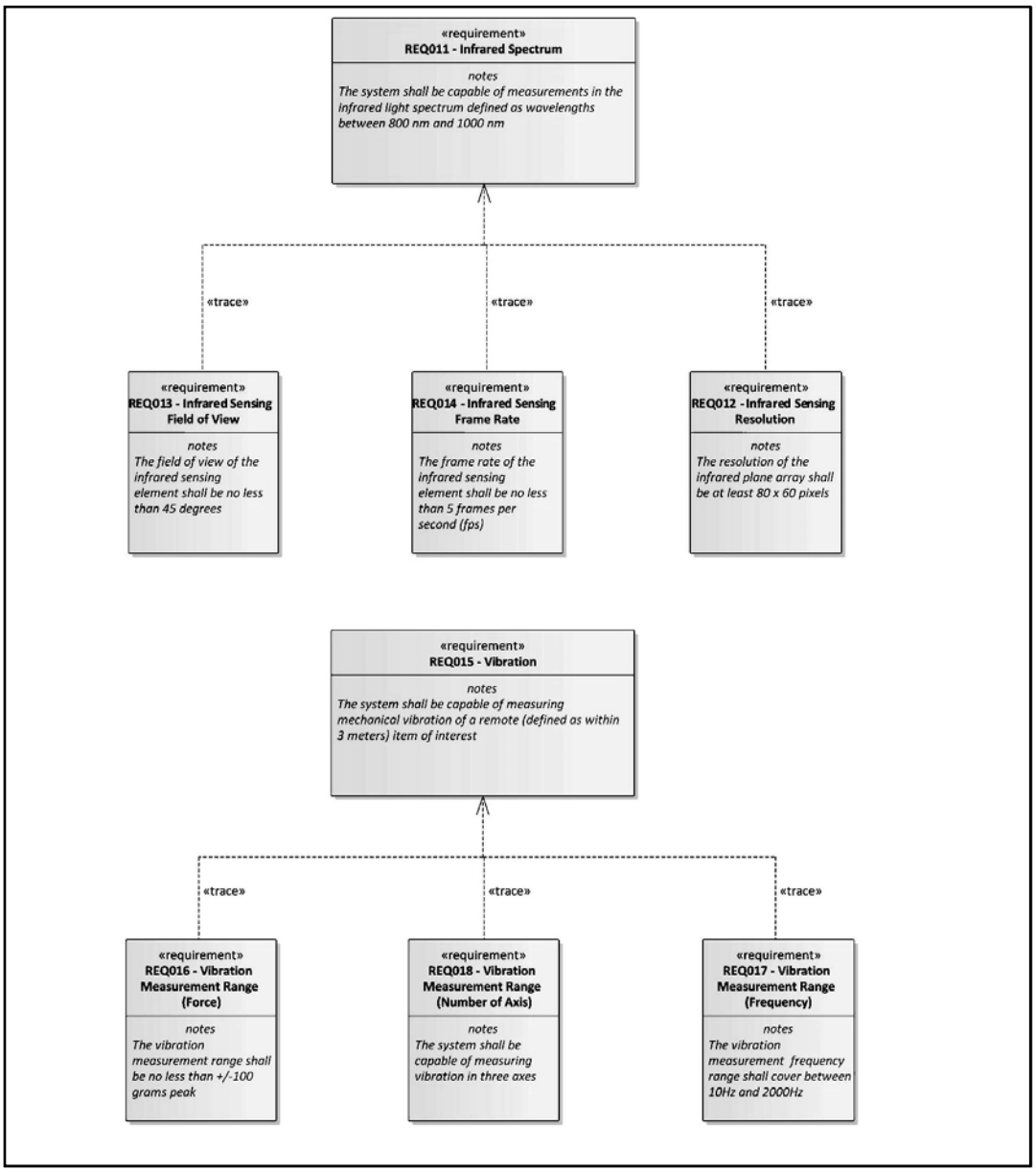



**Figure 1.9** Sensing capabilities: Part 3.

used to create the overall road map for the new product offering. In these cases, new product ideas impose requirements on new technology developments.

On the other hand, when technology developments or breakthroughs drive product innovation or the generation of new markets, the technology developments may also generate requirements on product features and functionalities.

At the systems engineering and architectural level, critical decisions should be made about what requirements are imposed on development; this is no trivial task and can be extremely challenging when new developments are thrust upon a market. It is this fundamental baseline, rooted in a thorough understanding of both the

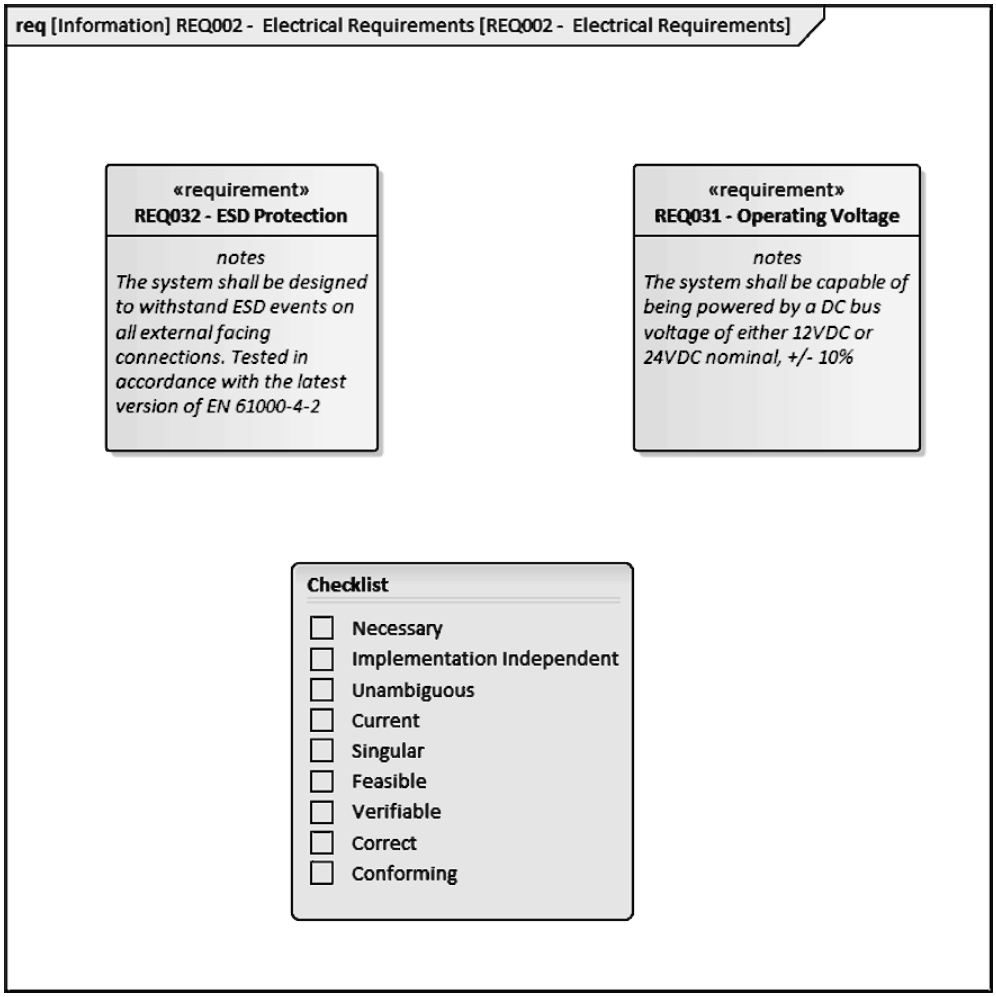



**Figure 1.10** Electrical requirements with a visible checklist.

technical concept at hand and the market (and markets in our case) being targeted, that is essential to a successful product development and launch.

### 1.6.2 SensorsThink: Back to the Journey

If we now immerse ourselves back into SensorsThink's development path, it may be easier to see why our given requirements really do straddle the level 10 and level 20 tiers (refer to Section 1.5.1), as we previously laid out. These requirements were derived by the analysis of potential markets and available technologies. Perhaps even requirements were compared and contrasted before arriving at the conclusion that several key applications could be served with a single sensor platform.

The systems engineer's job does not end at this point, as they would assuredly continue to be part of the requirements' decomposition down to the domains at level 30 while also assisting with the creation of verification plans at the system (and, if complex enough, the subsystem) level.

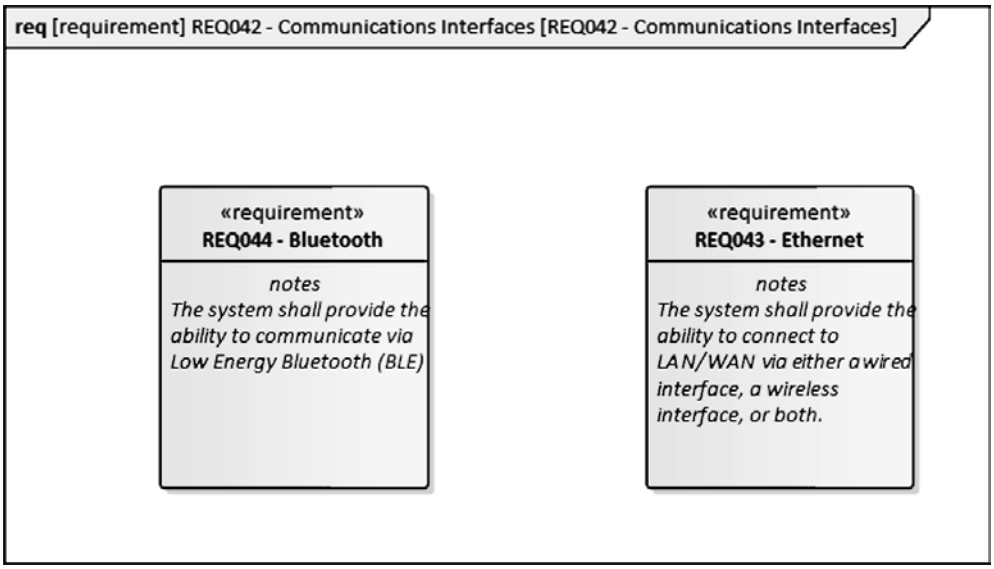



**Figure 1.11** Communications requirements.

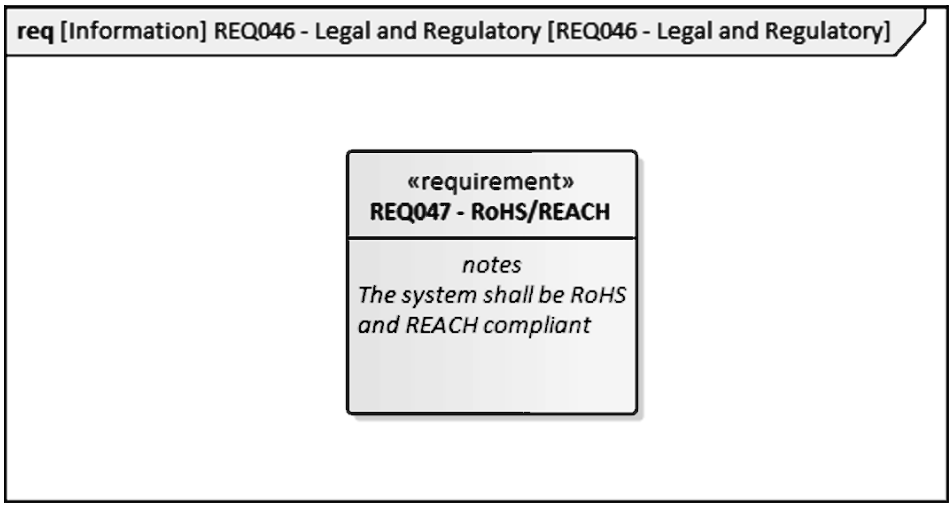



**Figure 1.12** Legal and regulatory requirements [8, 9].

### 1.6.3 Systems Engineering: An Overview

There are many different ways to apply the concepts of systems engineering to a development project. Generally speaking, the following topics are essential to keeping with modern best practices:

1. System requirements;
2. System architecture:
   (a) Logical architecture model development;
   (b) Physical architecture model development.

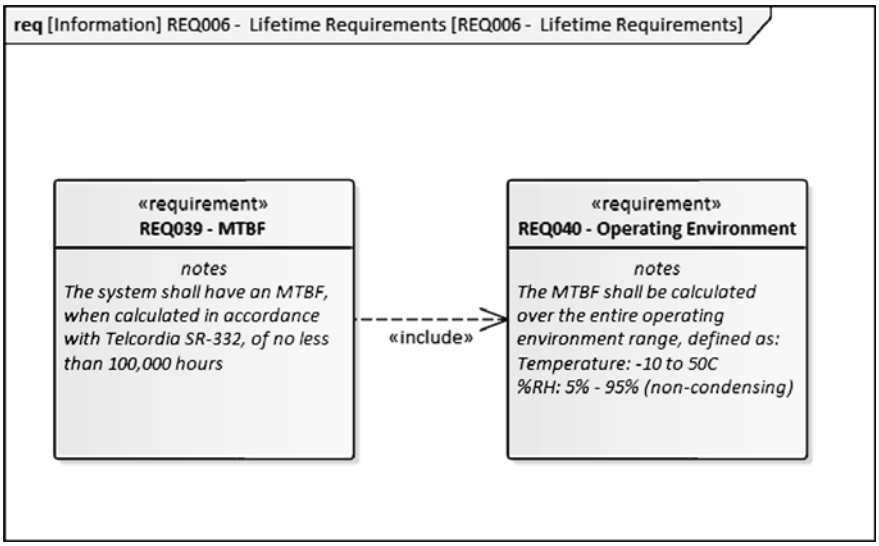



**Figure 1.13** Lifetime requirements [10].

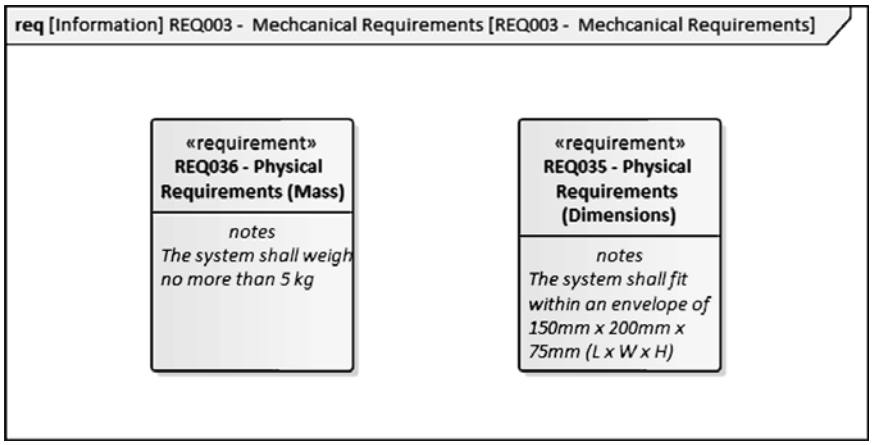



**Figure 1.14** Mechanical requirements.

3. System design: This is a way to supplement the architecture with useful (and necessary) data and documentation that enable domain engineers to implement the chosen architecture. This is the link between the architecture and the implementation of the system: things like equations, drawings, diagrams, and algorithms. SEBoK is a great place to grow knowledge in this area [14].
4. System analysis: This analysis allows developers to assess the system in a consistent way in order to choose the most efficient architecture. Trade-off studies, cost analysis, and risk analysis are all excellent ways to achieve this. Once again, SEBoK provides an excellent overview of this [15].

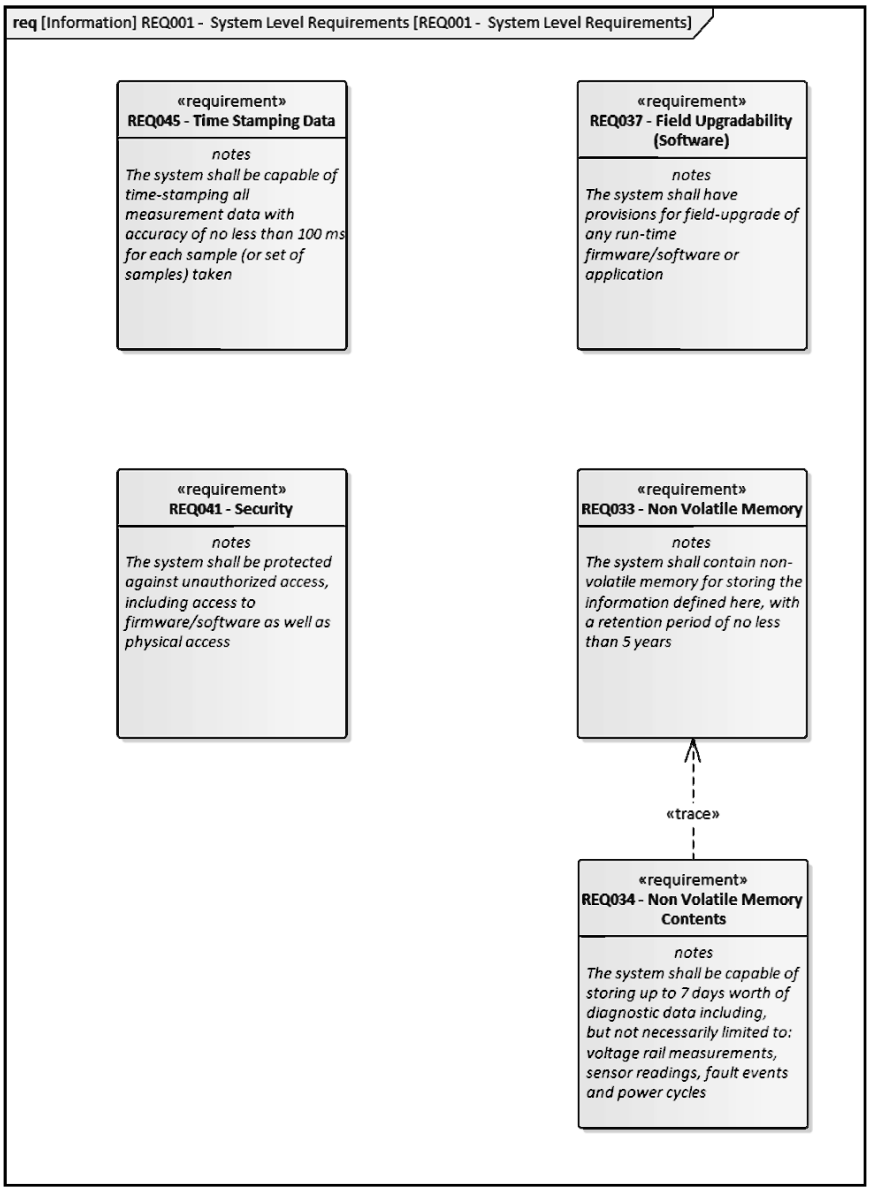



**Figure 1.15** Miscellaneous system-level requirements.

Let's proceed with the assumption that our set of approximately 30 requirements is the full set for our new product, even though we know better. The next step would be to architect the system.

### 1.6.4 Architecting the System, Logically

Usually the first step in architecting the system is to create a logical architecture model (although, depending on the system, it may not be necessary); the purpose of this type of architecture modeling is to create models of the major functionality of the system being developed. This should cover the product not only when in use

by customers, but also by other stakeholders in the process: test and quality assurance, production, and end-of-life procedures. Recall the four stages of a product's life from Section 1.2: introduction, growth, maturity, and decline. A well-executed architecture will consider all of these stages in addition to the functional requirements of the product in the marketplace.

It is very important that the intent of logical architecture is well understood. Logical architecture describes how a product works. This can be applied at a higher level of abstraction (consider the use case of a product being decommissioned) as well as lower levels of abstraction (how a certain piece of functionality should work); it is vital to separate this conceptual architecture from the physical way in which it will be implemented [16]. These logical architecture diagrams typically make use of three types of models:

- *Functional:* A set of functions and their subfunctions that defines the work done by the system to meet the system requirements.
- *Behavioral:* Arranging functions and interfaces (inputs and outputs) to define the execution sequencing, control flow, data flow, and performance level necessary to satisfy the system requirements.
- *Temporal:* Classifying the functions of a system based upon how often they execute. This includes, for example, the definition of what parts of a function are synchronous or asynchronous.

These architectural models are then linked to system requirements through a traceability matrix to ensure that every system requirement has been addressed by the logical architecture.

It is somewhat out of scope to perform a full logical architecture for the new sensor platform SensorsThink is developing; looking at one or two sample diagrams that may come from this portion of development is certainly useful, however.

### 1.6.5 Keep Things in Context (Diagrams)

One key diagram to consider is a context diagram, looking at the system being designed in its environment (or in many different environments). Enterprise Architect is an ideal tool for creating these types of architectural diagrams, so we can examine a few examples by expanding our system model to include them.

For our system, let's assume that we have three stakeholders in mind for the system when deployed in its operational environment during its useful life: the end user (customer), a remote server, and service personnel.

Each of these stakeholders (actors) will have a different interest and interaction with the system once it is deployed into operation. This is noted in the diagram (see Figure 1.16):

1. Service would have interest in diagnostic data or upgrading the system.
2. The remote server would likely take data for post-processing for all attached sensors and log system diagnostics.
3. The end user may only care about system status (fault/no fault) and any alarm conditions that the sensors are reading for their application.

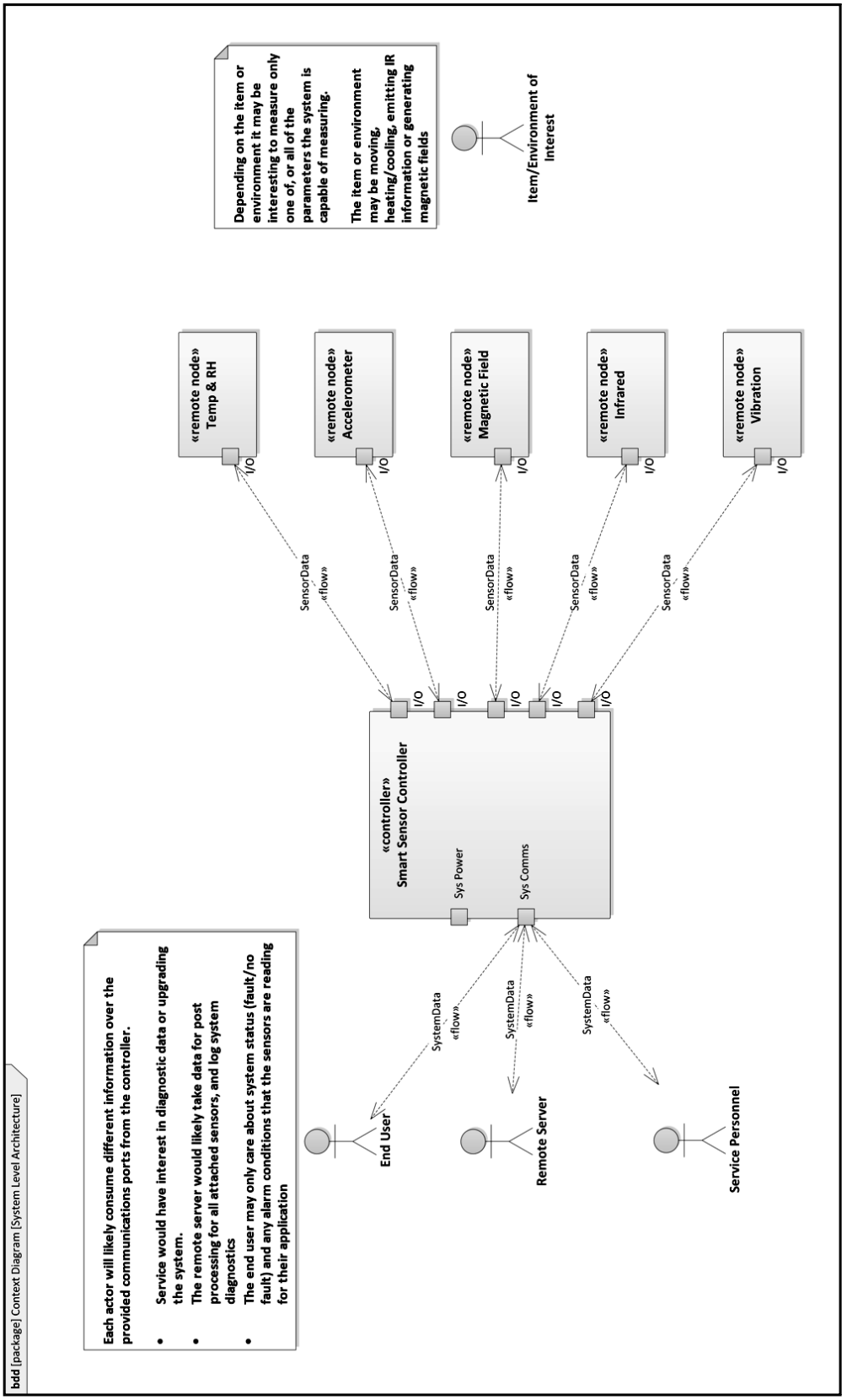



**Figure 1.16** Context diagram of the Smart Sensor Controller.

Note that all of these end users will be interacting with the system controller through the same communications interface (or interfaces, in our case). This helps to put definition to those interfaces and to things such as logging in to the system, access levels, and security needs for the interface.

We also see that our environment or item of interest is shown in this context diagram. This is what the sensor platform is monitoring and taking measurements of.

It is important to take note that this diagram is not prescriptive; it does not define anything about a particular interface or feature of the system. It is not defining what type or volume of information will flow over various interfaces. It simply provides the context for which the system will be operating in this particular use case.

### 1.6.6 Monitor Your Activity (Diagrams)

Another diagram type to consider is an activity diagram, a basic example being a very high-level activity diagram of the system's operational modes.

This is a very basic activity diagram, but still defines (at a high level) the way the system is intended to function during run time (see Figure 1.17); a few bullet points illustrate this nicely in conjunction with the diagram:

- System power is applied.
- Power on self-test (POST) is performed. This is when housekeeping diagnostics will take place; these diagnostic tests cannot be defined in meaningful detail until the system is better defined. Some typical examples are memory tests, processor self-tests, and voltage measurements.
  - POST has a decision point: if it passes the operating system (whatever it may be) is loaded; or, if it fails, the system halts and enters standby mode.
- Load operating system: This is when the processor can load the operating system. In some cases, like a bare metal system, this is really minimal. In other cases (Linux, for example), this can be a very large task and very complex. A further set of diagrams in a complex case may well be necessary to fully describe the system operation. If any faults occur during the operating system load function, the system immediately halts and enters standby mode.
- System operation begins: This is when the system is acquiring data and either writing it to memory or sending it over communications, for example. It will also be servicing and checking for interruptions during this time, as well as monitoring for any run-time faults, loss of power, or corrupted data. If any faults occur during operation, the system immediately halts and enters the standby mode.
- Shutdown request: This is when a user requests a graceful shutdown. The system performs whatever functions are necessary to ensure a safe and reliable end to operation and enters the standby mode.

Diagrams like this are extremely useful in defining the behavior of a system at varying levels of depth; the shutdown routine of a safety-critical system may warrant many layers of complex diagrams, contrasted against the shutdown routine of

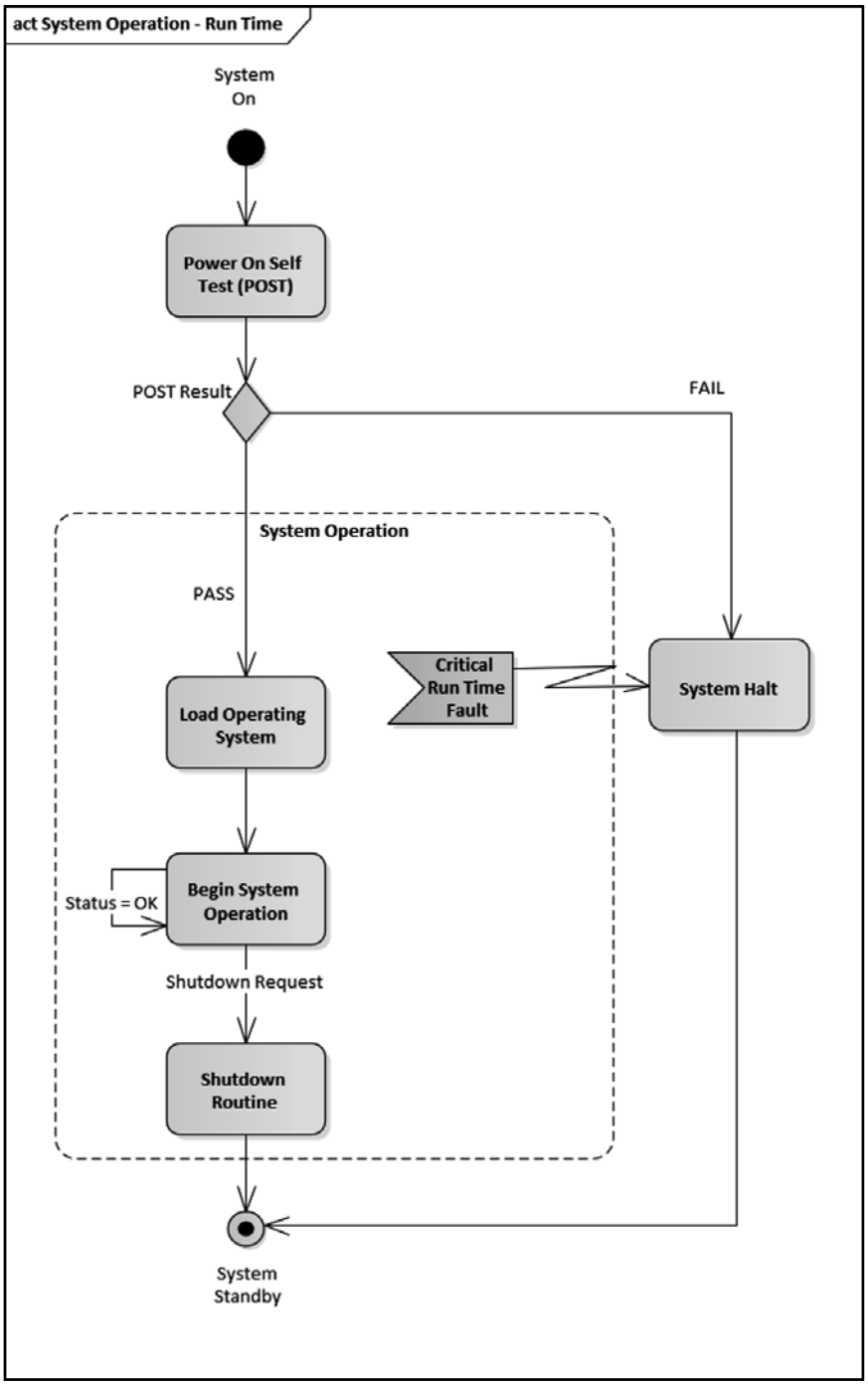



**Figure 1.17** System operation: run time.

a coffee maker, which may only require one or two actions be shown (it may be as simple as the user turning the power button off).

### 1.6.7 Know the Proper Sequence (Diagrams)

One more example diagram to consider is reading sensor data (see Figure 1.18): a core function in our new product design. In this diagram, our systems engineer has decided that, prior to getting sensor data, a sensor health check should be performed. Assuming that this is possible, this might be something as simple as verifying a "Who am I?" register and performing a cyclic redundancy check (CRC) on the data received to ensure that communications are reliable. If we recall Figure 1.17, a failure of this CRC may well cause a critical run-time fault and immediately halt operation. If the status check is fine, sensor data is requested, gathered from the environment, and returned back from the sensor. This data may be post-processed by the sensor node, depending on implementation, and then handed back to the core controller platform for further use.

### 1.6.8 Architecting the System Physically

Logical architecture really does tell the engineering team how the system should work and, from that, how we can begin to determine the implementation details in a physical architecture model [17]. The goal of physical architecture modeling

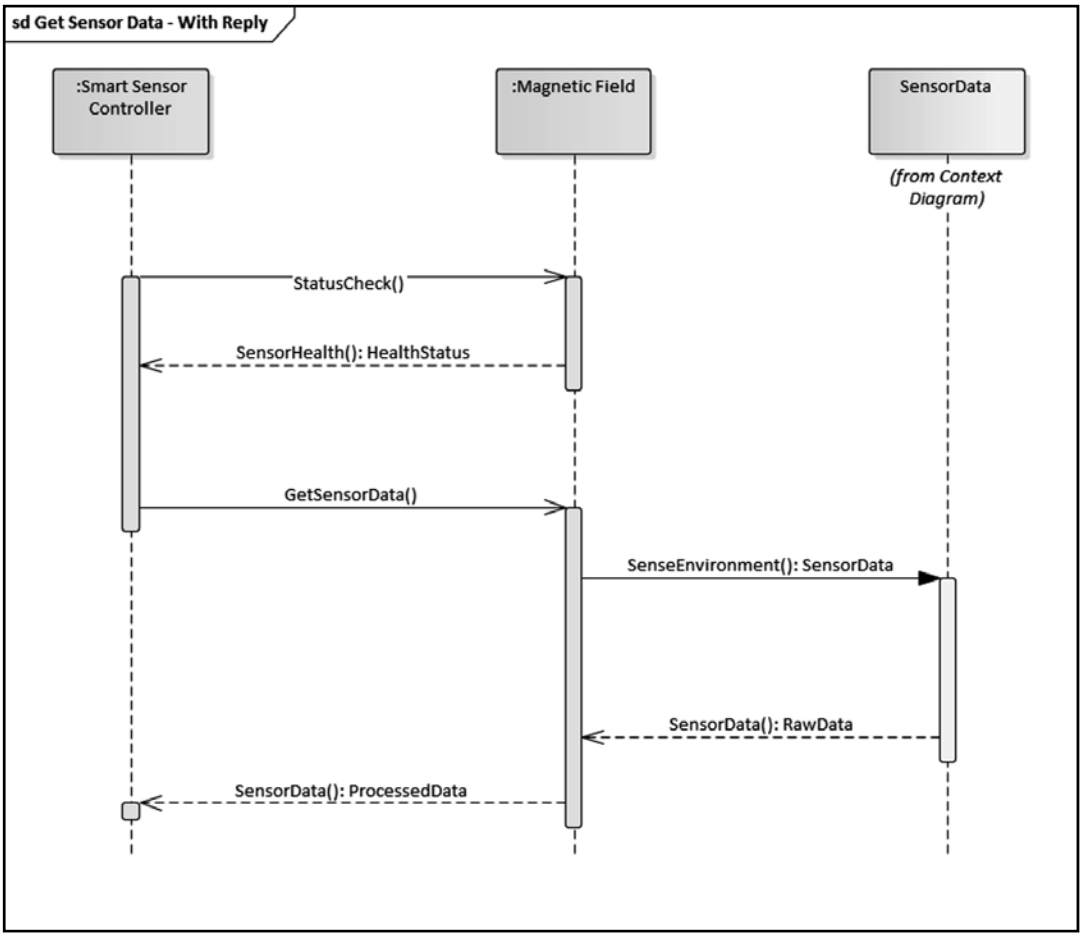



**Figure 1.18** Sequence diagram for reading a sensor.

is to create models and views of a physical solution that accommodates the logical architecture model (in other words, accounts for all functional, behavioral, and temporal models and diagrams) and satisfies system requirements. Physical elements (hardware, software, mechanical parts) have to be identified that can support these models and requirements.

One of the key drivers for the physical architecture model may be interface standards; they may be one of the key drivers for the overall system. However, it is also sometimes the case that interfaces are chosen later on in the model development process. In our system, we have both of these cases present: the communications interfaces that were stated as system requirements (Ethernet and Wi-Fi or Bluetooth) and the sensor communications interfaces that were not specified in any meaningful way.

Because the system will likely be communicating over established networks to the outside world, it makes sense that these protocols are defined and required at a very high level. On the other end of the system, there are several internal interfaces that are only exposed to the Smart Sensor Controller. These sensor communications interfaces can be of any type as long as the selected sensor meets its sensing performance requirements, the real driver for the sensors are those performance requirements, not the way in which data is transmitted between sensor node and controller.

### 1.6.9 Physical Architecture: Playing with Blocks

As a starting point for understanding physical architecture, let's look at a proposed solution for the Smart Sensor Controller (Figure 1.19). The physical architecture diagram shown in Figure 1.19 is intended to cover the large central block of the context diagram at a hardware level.

We can see from Figure 1.19 that the central processing portion of the system appears to be quite complex even though much of the system is still rather loosely defined. However, from covering all the requirements and in considering the various life cycles of the product, we end up with a rather complex processing board. It must be capable of performing a number of functions:

1. Ethernet and wireless communications;
2. Serving data to a cloud server;
3. Acquiring, time-stamping, and performing other post-processing and sensor fusion on as many as five sensors;
4. Being able to be upgraded in the field;
5. Being secure in the handling of data and information.

Note that most of the physical blocks here are represented (at first glance) by hardware only, but a closer look would reveal that there is a large volume of firmware and software work that must occur in the system to operate properly and reliably. There must be accurate date and time keeping for time-stamping data, data integrity must be guaranteed, sensor fusion must be performed, image processing (for our infrared sensor), data storage, and data transmission over multiple interfaces (perhaps up to three). When analyzing the system and what needs to be at the heart of it, there could be several implementations:

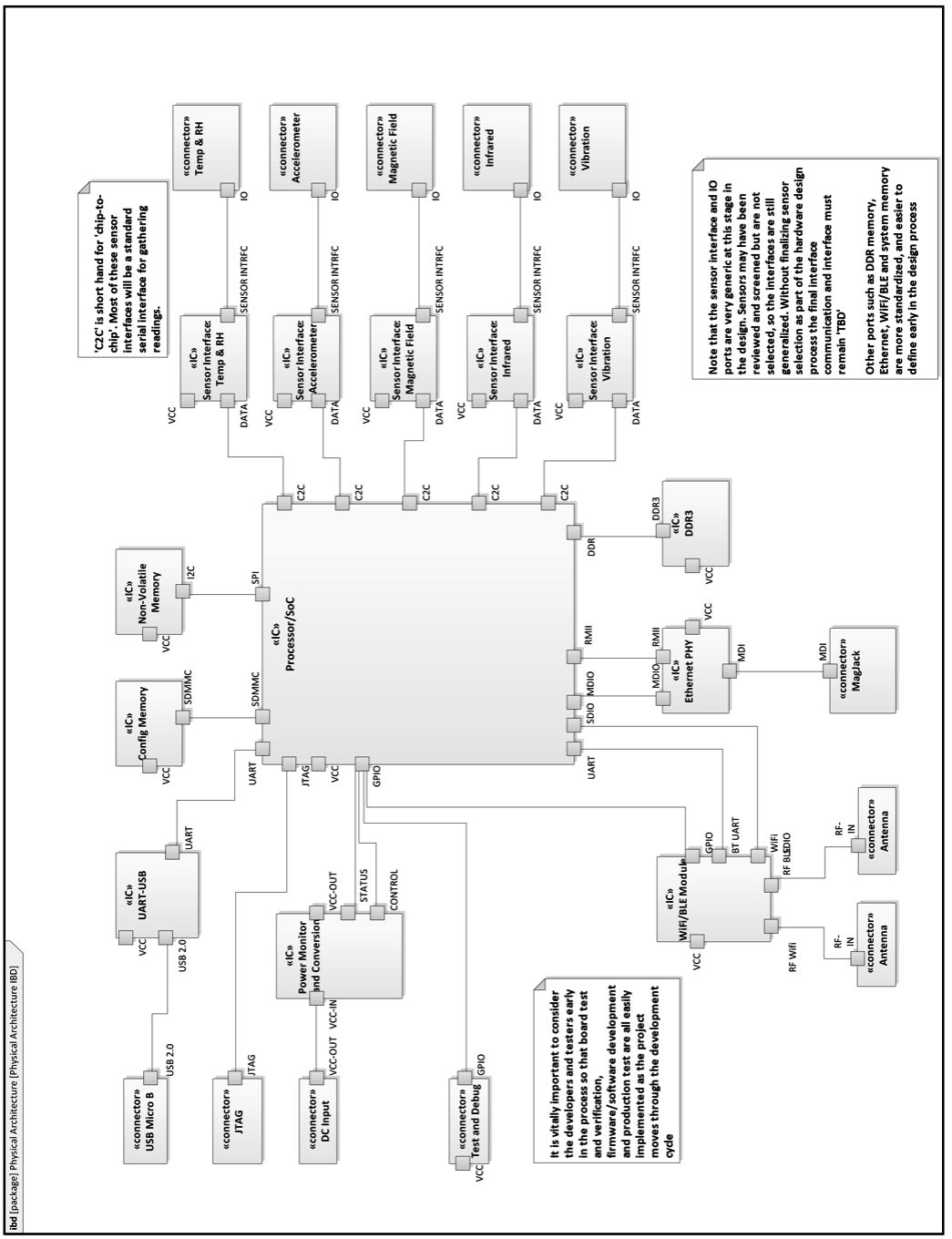



Figure 1.19 Smart Sensor Controller physical architecture.

1. Discrete microprocessor;
2. Discrete microprocessor and coprocessor/field programmable gate array (FPGA);
3. System on Chip (SoC).

More applications are lending themselves to the use of multiple processors and FPGAs. The natural progression of this trend led to higher integration, and the colocation of a relatively powerful Advanced Risc Machine (ARM) processor (an A class processor capable of running an application) with an FPGA for doing data processing or algorithmic acceleration. In our case, any of the three solutions mentioned above are feasible solutions; engineering teams could likely make any of the three work. However the ideal solution given the required functionality is using a SoC. A SoC, even an entry-level one, provides tremendous flexibility and processing power for many applications. Typically, this allows for a full Linux operating system to run and handle the high-level tasks such as serving data, and the inclusion of FPGA fabric allows for data processing to be done more efficiently in parallel and can even allow for a real-time coprocessor (like an M class ARM processor) to handle some of the lower-level, more time-critical functionality such as system diagnostics. These operating systems require a file system and run-time memory, which is why there are physical blocks for both double data rate three synchronous dynamic random-access memory (DDR3 SDRAM) and for configuration memory in the diagram.

The Bluetooth functionality has been paired with a Wi-Fi link. This comes from knowing what solutions exist in the marketplace (recall that it is important for a systems engineer to know what solutions are in the market and to use that information to sometimes drive design; this is one of those cases).

Things such as JTAG [18] (a very common test interface named after the Joint Test Action Group) and the UART (Universal Asynchronous Receiver/Transmitter) are in place because test and development were considered. The generic test and debug physical block is in place to accommodate a production test as well as field service. Consider something simple like a blinking heartbeat LED and some status LEDs that can be easily interpreted with a quick glance. The remaining blocks are directly driven by requirements as presented by the management and marketing team.

### 1.6.10 Trace Your Steps

A sample trace diagram, as shown in Figure 1.20, can be examined to understand how requirements can be allocated to various physical blocks of the architecture. This can be done exhaustively (and is required to be done this way for critical or safety systems) for the entire model if desired or only for a select set of requirements and diagrams.

At this point in time, a high-level set of architectural diagrams exists, the requirements are understood, and the engineering team has a physical architecture to start to implement (see Chapter 2). We are well on our way to developing this new product, and we continue to strive towards our key milestone.

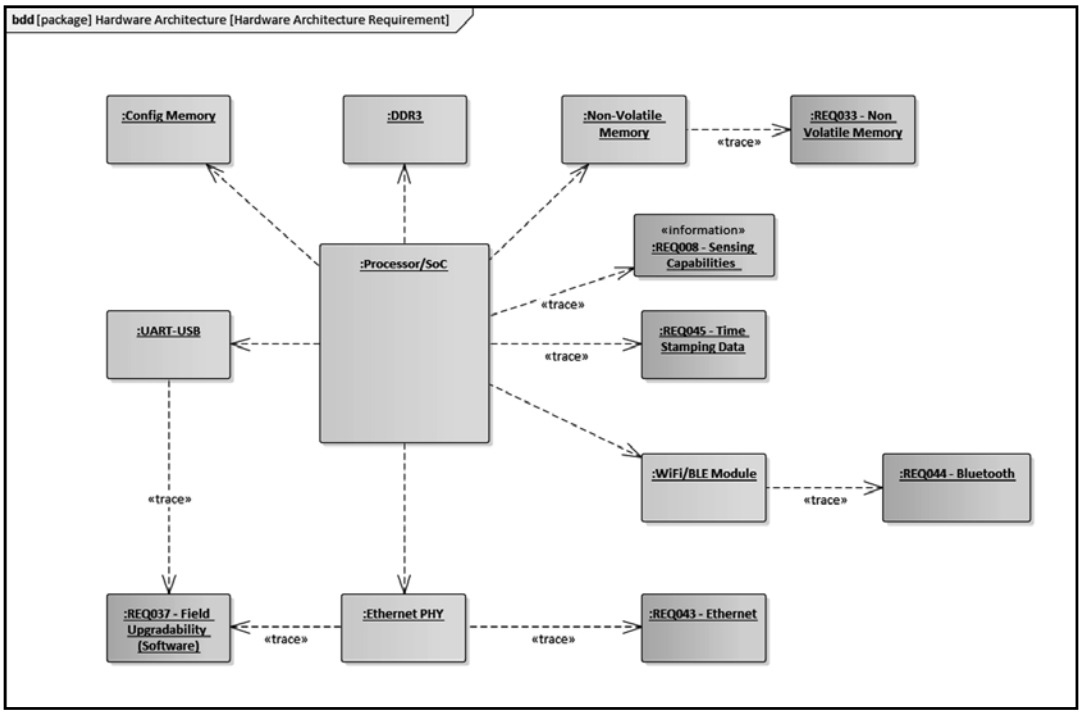



**Figure 1.20** Tracing requirements to physical blocks.

### 1.6.11 System Verification and Validation: Check Your Work

The system validation plan is the highest-level integration test plan; it should directly validate the requirements put forth by the customer (or business unit) at level 10. This test plan will typically spell out an objective, show a link to the tested requirements, and point out any necessary prerequisite tests.

As an example, consider an environmental test being performed at the system validation level. It would be considered a best practice to validate that the environmental chamber is actually meeting the temperature and humidity set points that we are expecting with calibrated measurement equipment prior to subjecting your product to that environment, defining the information to be recorded and how often that information must be recorded, and also very clearly defining pass-fail criteria. It also is often noted in these test plans if any assumptions or constraints exist that are pertinent to the task at hand. As an example, consider a test plan for the communication of many connected SensorsThink nodes back to a master server. This test plan would be created under the assumption that the server both existed and was able to communicate to many nodes concurrently. Another key concept is that of validation versus verification; this will be discussed in a bit more detail later. For now, it is sufficient to say that validation is often regarded as "was the right system built?" testing and verification is often regarded as "was the system built correctly?" testing.

## 1.7 Engineering Budgets

The goal of any engineering project is to deliver the final product on time, to quality, and on budget. Naturally, we consider the budget element to indicate the financial budget allocated for the development, which it primarily is. However, successful project delivery will require the achievement of many engineering budgets across several subsystems of the design.

Engineering budgets are a response to requirements, contained in the system requirement specification. The engineering budget allocates portions of the overall target defined by a system requirement to subsystems in the design, providing clear guidance and targets to the designers.

It is natural when doing this allocation to retain a portion of the overall budget as contingency, in case one subsystem is unable to meet its budgets. Retaining contingency at the highest level prevents each subsystem from retaining a contingency, which makes achieving its budget harder and therefore increases the technical risk and potentially impacts the delivery timescales and cost trying to achieve the budget.

### 1.7.1 Types of Budgets

When allocating an engineering budget, it requires some thought and analysis to ensure that the allocation is sensible. It is not sensible, for example, to allocate a mass budget equally between the enclosure and the circuit card. Nor is it good practice to allocate evenly the performance budget between a low noise analog front end and the associated digital processing element. Budgets should be allocated depending upon the needs of the subsystem.

The engineering budgets documents generated for a project will vary depending upon the requirements; however, commonly generated engineering budget documents include:

- *Unit price cost:* A financial target that defines the manufacturing cost of the assembled product. This includes not only the cost of the bill of materials but also the cost of assembly and manufacture.
- *Mass budget:* A physical target that defines the acceptable mass of the finished system. Achieving this budget is critical on applications for mass constrained applications (e.g., aerospace, space, automotive application). Mass budgets will allocate budgets to elements of the design such as enclosure, circuit card, power supplies, and actuators.
- *Power budget:* A physical target defines the maximum power that may be drawn by the application. A power budget may define different operating conditions and allowable power budget for those operating conditions. Examples of this include low power and reduced power operating requirements. Power budgets will also include allocations for direct current (dc)/dc conversion.
- *Memory budgets:* A storage target, this defines the required nonvolatile and volatile memory required for the overall application.

- *Performance budgets:* This will vary from project to project depending upon the application; however, the performance budget will allocate the overall performance requirement across the subsystems. This enables challenging subsystems to have a larger proportion of the budget than less challenging. For example, when measuring real-world parameters using a sensor, the analog element of the design that is more effected by temperatures, tolerance, and drift will be allocated more of the budget.
- *Thermal budget:* On many applications, the thermal dissipation can be challenging; examples include wearable products, space, and aerospace where thermal management is difficult or could result in harm. The thermal budget is therefore closely linked with the power budget and will address not only the external interface temperatures but also the allowable dissipation of the subsystems and even key components within the design that drive the thermal solution.

The creation of the engineering budget is a system-level task; as such, they should be generated as part of the requirements generation and prior to preliminary design review. This enables the designers to work with these budgets when they are creating the preliminary design. These budgets and how they are addressed are then able to be reviewed as part of the preliminary design review, which is held before the detailed design continues.

### 1.7.2 Engineering Budgets: Some Examples

Let's look at the engineering budgets, which can be generated using the requirements defined in Section 1.5. The one engineering budget which needs to be considered for an example application is the REQ036 - Physical Requirements (Mass): The system shall weigh no more than 5 kg. This defines the maximum permissible weight of the overall system; we need to define the allocation of each component of the system. At the architectural level, these components can be split into:

- *Enclosure and fixings:* The enclosure of the system, which protects it from its environment and provides thermal dissipation.
- *The ac/dc converter:* Power converter, which converts the input voltage to secondary voltage required by the system.
- *Circuit card assembly:* The circuit cards that contain the electronic components, which implement the required functionality of the system.

#### 1.7.2.1 Tipping the Scales: Mass Budget

Allocating the mass budget between these three elements can be pretty straightforward; the majority of the mass budget will be allocated to the enclosure and fixings, which accounts for the most mass of the system. Allocation for the enclosure includes any additional fixings or shielding required in the system, for example, shielding over mixed signal converters.

As such, the mass engineering budget can be allocated as:

- Contingency: 500g;
- Enclosure and fixings: 3,500g;
- The dc/dc converter: 500g;
- Circuit card assembly: 500g.

This budget allocation provides achievable budgets to the major elements of the solution while also retaining a contingency in case any element struggles in meeting these requirements.

#### 1.7.2.2 Powering the Pieces: Power Budgets

The smart sensor SoC requirements do not contain a maximum power consumption requirement as the system is intended to be powered from an electrical outlet. As such power budget would be constrained only by availability of the alternating current (ac)/dc converters to convert between ac power and the required dc input. As a design engineer, such a situation is nice as the only need is to accurately determine the power dissipation on the board once component selection has been completed to accurately size the ac/dc power converter.

However, we are not always as lucky as we are in the case of the Smart Sensor SoC development. To ensure that the benefit of a power budget is understood, we will go from a simple example of the Smart Sensor SoC to looking at a very challenging and power-constrained application in that of a CubeSat application.

Electrical power to power a CubeSat is generated from solar panels that are mounted on the external faces of the CubeSat. Due to the small surface area of the solar panels, CubeSat applications are typically very power-constrained.

Typical requirements placed on a CubeSat payload are:

- The payload shall provide regulated supply rails at 3.3V, 5.0V, and 12.0V.
- The total current drawn from each rail may be up to 600 mA.
- The total power dissipation during the sunlit orbit shall be 400 mW.

These requirements present an interesting challenge to the designer as the individual currents can supply reasonably high currents. Although the total potential power supplied is just over 12W, what limits the solution is the final requirement. The total power dissipation must be no more than 400 mW in the sunlit orbit, which is typically in the region of 45 minutes.

Higher powers can be drawn from the batteries; however, the average dissipation over the 45-minute orbit must be 400 mW or less. Higher-power payloads can therefore be powered for a shorter period; therefore, if we needed to draw 12W from the power supplies, we can operate the payload for only a fraction of the 45-minute sunlit orbit.

To keep battery life maximal, during the dark half of the orbit, the payloads are powered down.

Our power budget must therefore demonstrate that the design can operate within these constraints; as the project matures, the power budget needs to be kept

up-to-date to ensure that any power increases due to scope creep are addressed and are still within budget.

#### 1.7.2.3 Count Every Coulomb

When determining the budget for this payload, several factors must be considered, for example:

- Operating modes of the CubeSat payload;
- Proposed power architecture of the solution;
- Losses due to power conversion;
- Power switching to reduce the power dissipation.

The budgets that we must achieve for this CubeSat mission are:

- 3v3 Supplies less than 1.98W;
- 5v0 Supplies less than 3.0W;
- 12v Supplies less than 7.2W;
- The average power dissipation over 45 minutes is less than 400 mW.

In this example the CubeSat payload contains an FPGA along with configuration memories, SRAM memory, and support circuits such as oscillators.

Creating the power budget for this application requires the use of power estimation spreadsheets provided by the FPGA manufacturer (e.g., Xilinx XPE) to determine the maximum power required for the FPGA application. This estimation needs to include clocking rates, logic, and block random access memory (BRAM) resources, specialist macros (e.g., DSP, Processor, and Clocking). At early stages of the project, it is also good practice to include a margin factor in this estimation.

It also goes without saying that our power budget should be based upon the worst-case power dissipation taking into account process, voltage, and temperature (PVT) variation.

For this CubeSat application, the power estimation for the FPGA and support circuitry can be seen in Tables 1.1 and 1.2.

These power estimations identify the power drawn from each of the subregulated rails; they do not include the power required from the supply rail.

The proposed architecture uses the 3v3 supply to directly power the FPGA 3v3 IO, non-volatile random-access memory (NVRAM), and controller area network (CAN) interface. As such for the 3v3 supply rail, we do not need any further conversion, the load on the 3v3 Supply rail is 1.59W or 78% of the available 1.98W.

**Table 1.1** FPGA Power Estimation

| *Rail* | *Voltage* | *Tolerance* | *Voltage Maximum* | *Current (mA)* | *Power Maxiumum (W)* | *Power Nominal (W)* |
|---|---|---|---|---|---|---|
| FPGA Core | 1 | 5.00% | 1.05 | 750 | 0.7875 | 0.75 |
| FPGA IO | 2.5 | 5.00% | 2.625 | 250 | 0.65625 | 0.625 |
| FPGA IO | 3.3 | 5.00% | 3.465 | 250 | 0.86625 | 0.825 |

**Table 1.2** Support Circuitry Power Estimation

| *Rail* | *Voltage* | *Tolerance* | *Voltage Maximum* | *Current (mA)* | *Power Maxiumum (W)* | *Power Nominal (W)* |
|---|---|---|---|---|---|---|
| NVRAM | 3.3 | 5.00% | 3.465 | 50 | 0.17325 | 0.165 |
| Config Memory Core | 1.8 | 5.00% | 1.89 | 25 | 0.04725 | 0.045 |
| Config Memory IO | 2.5 | 5.00% | 2.625 | 10 | 0.02625 | 0.025 |
| CAN If | 3.3 | 5.00% | 3.465 | 150 | 0.51975 | 0.495 |

The power drawn from the 3v3 rail can therefore be said to be within the available budget, while the 1v0, 1v8, and 2v5 rails are generated using dc-dc switching converters connected to the 5v0 rail. When we convert dc-dc voltages, we must account for the switching efficiency; depending upon the efficiency, the power required from the supply rail may be considerably higher than that drawn by the load. When trying to meet tight power budgets using dc-dc converters, we want to select high-efficiency converters. For this example, a converter efficiency of 85% was assumed.

The total power drawn from the 5v0 rail is 2.1W, this is again within the allowable budget of 3W, providing a 70% load on the 5v0 rail.

The loads on the 3v3 and 5v0 rail provide margins of 22% and 30% respectively, we also need to ensure the power budget is able to address short duration start-up currents. Typically, these will be addressed in a worst-case analysis of the design.

With a total power draw of 3.66W the final determination is the time the CubeSat payload can be operational to achieve the 400-mW average during the 45-minute sunlit orbit.

When we calculate 400 mW/3.66W, we see the ratio required to achieve the average, as such the payload can be operational for just over one-tenth (10.92%) of the 45-minute operational time.

This becomes a limiting factor on the FPGA design, it must be able to achieve its task within the allocated timescale as determined by the power budget. This time limit can be therefore included as a derived requirement in the FPGA level requirements.

One final point of note on this example is throughout this example we used the worst-case figures. If we had used the nominal figures the power requirements from the 3v3 and 5v0 rails would be 1.485W and 1.445W, respectively, resulting in a total power of 3.185W. This nominal dissipation of 3.185W compared to the worse case of 3.66W is a 13% difference and would result in the 400-mW sunlit orbit average being exceeded if the operational time was based of the nominal power dissipation and process, voltage, and temperature variation resulted in power

**Table 1.3** Power Conversion Rail

| *Supply* | *POL Power (Watts)* | *Efficiency* | *Supply Power (Watts)* |
|---|---|---|---|
| 1v0 | 0.7875 | 0.85 | 0.926470588 |
| 1v8 | 0.04725 | 0.85 | 0.055588235 |
| 2v5 | 0.6825 | 0.85 | 0.802941176 |

dissipation closer to the maximum. This increased power dissipation may result in system level impacts as the battery experiences more dissipation than expected.

### 1.7.3 Engineering Budgets: Finishing Up

While this example is long it demonstrates the importance of determining a power budget at a subsystem levels. Without the correct analysis and budget allocation the hardware and FPGA design teams may inadvertently design a solution which is not capable of achieving the budgets.

If in either of the examples presented, we had been unable to achieve the allocated budget, the issue can be raised internally within the project. One method this issue can be raised internally is by the issue of a request for deviation (RFD). This request for deviation can then be reviewed by the systems engineering management team and if agreed, some of the agreed contingency budget can be freed up for that subsystem. If there is no contingency available, the RFD can be rejected and the engineering team will have to go back to the drawing board as to how to address the solution. These high-risk elements of the design should have been included in a technical risk register with associated allocated budgets to address the situation should this situation occur.

## 1.8 Interface Control Documents

When it comes to designing our system, the systems interfaces represent an area of risk. This risk arises as the system interfaces include multidisciplinary factors (e.g., electronic and mechanical design along with the complexity of interfacing with external systems). As engineers to mitigate issues arising due to incorrect interfacing between units or modules, we define the interfaces accurately in an interface control document (ICD). The ICD enable us to clearly define the different interfaces within our system and will cover a range of disciplines. As such, they will cover such areas as:

- *Mechanical ICD:* A mechanical ICD will define the mechanical aspects of the design. This will include not only the external dimensions and tolerances but also additional requirements such as finishing, emissivity, and flatness. External mechanical ICDs may be provided by the customer if they have a tightly defined space envelope, which the system must integrate within. If the external mechanical ICD must be developed as part of the development contract, then it is advisable to ensure agreement and sign off by the customer at an early design review. It is also common to define not only an external mechanical ICD but also one or several internal mechanical ICDs. Depending upon the complexity of the solution these internal mechanical ICDs may simply address how the circuit card mounts within the enclosure to how several circuit cards and a back plane are contained within the enclosure. These internal ICDs will also include mounting points, keep out areas and mechanical bracing and thermal paths for the circuit cards.

- *Thermal ICD:* A thermal ICD will define the thermal interfaces of the system. This will include the thermal conductivity between the enclosure and the mounting points, along with any additional thermal management methods. These additional thermal management methods may include forced air cooling, conductive paths, or liquid cooling. The thermal ICD enables the system designers to understand the thermal environment and design a system which can operate within it. The thermal ICD will also be very useful when (if) it comes to determining the part stress analysis and reliability figure.
- *Electrical ICD:* An electrical ICD will define all the electrical interfaces of the system. Often this will be split into an external electrical ICD and an internal electrical ICD. The electrical ICDs will define the power supplies, signal interfaces, and grounding. Obviously when working with electrical interfaces we should try to work with industry standard interfaces such as Ethernet, RS232, and USB depending upon the needs of the system. This is especially true for external interfaces where the system is required to interface with others. Internally within the system again standard interfaces should be used wherever possible. However, if they are not the interfaces should be defined in depth using the internal EICD. One critical element of the EICD along with the signal and interface types is the definition of the electrical connector types and pin allocations. Internally, this enables the individual circuit card designers to develop solutions from a single reference document ensuring they are fully aware of the connector type required and the exact pin allocations.

As electronic engineers it is the electrical ICDs which throughout our career we will both develop and use most often in our careers, although we will also use the mechanical and thermal ICDs at times.

The external electrical ICD will define the interfaces presented external by the system, this will include not only power and signals, but also connector type, maximum number of mating cycles and pin allocation. Within the ICD we should define several groups which can be used for each signal. Commonly used groups include power, analog, single-ended digital signal, differential digital signal, high reliability signals, and ground, the number of groups will depend upon the different interfacing standards used.

### 1.8.1 Sticking Together: Signal Grouping

Using groups means that each signal within the ICD can be allocated to one of the groups, as such updates or modifications to the signal group does not require significant rework, although each group will contain different parameters. A power group, for example, will contain at least the following elements:

- *Nominal voltage:* The nominal voltage supplied on the power rail.
- *Tolerance:* The maximum and minimum tolerance available across all worse-case conditions may be as simple as +/–5%, which is standard for many digital power rails. Alternatively, it could contain a range of tolerances across the operating conditions.

- *Maximum power:* The maximum power that can be drawn from the power rail.
- *Noise characteristics:* How much ac and dc ripple present on the rail.
- *Ground return:* The return path for the supply current.

For digital signal groups, the following elements may be used:

- *Logic type:* The logic standard used (e.g., LVCMOS1v8 or LVCMOS3v3); logic standard should also refer to the appropriate JEDEC standard. This is important, as it includes the necessary $V_{IH}$ and $V_{OH}$ minimum and maximum levels for that logic standard.
- *Characteristic impedance:* The characteristic impedance of the tracking required for the signal in the PCB or cabling.
- *Termination scheme:* If the interface terminated and, if so, what type of termination scheme is used.
- *Data rate:* The data rate of the signal.
- *Protocol:* How information is communicated over the link; this may be a reference to an industry standard, or if a bespoke approach is being used, then it may be a reference to an internal document that fully defines the communication standard.
- *Special requirements:* For example, the output open drain or tristate.
- *Reset:* Quiescent state following reset.

When it comes to differential digital interfaces, the interfacing specification is very similar to that of the single-ended logic groups, with the exception that the logic type is defined as either low-voltage differential signaling (LVDS), low-voltage positive emitter-coupled logic (LVPECL), or high-definition multimedia interface (HDMI). The characteristic impedance will also be different at typically 100Ω differential.

We also discuss several analog and radio frequency (RF) signals that we need to define as belonging to a group; these analog signals will typically contain elements such as:

- *Voltage range:* The operating voltage input range;
- *Current:* Maximum supply current;
- *Noise:* Ac and dc noise;
- *Return path:* Return path for the signal;
- *Characteristic impedance:* The characteristic impedance of the signal tracking/cabling.

Defining these different groups within the ICD is also useful in the later stages of the design process when layout is underway. These different groups can have different layout constraints applied to them regarding length matching and even separation between signal groups.

Once the different signal groups are created, we are then in a position to begin creating the actual interface information (e.g., signal, direction, and pin allocation).

### 1.8.2 Playing with Legos: Connectorization

The next step in the ICD creation process is to define each signal, its direction, and signal grouping. When we do this assignment, we also need to select the connectors as well. Connector selection will require consideration of the type, polarity, number of signal pins, mating cycles, and current rating.

We also need to consider carefully the connector type. The considerations here include the number of pins available, physical size, mounting technology, and polarity. If multiple connectors are required, it is a good idea to consider the use of different connector styles to prevent inadvertent incorrect connection. This incorrect connection could lead to damage and require significant design analysis depending upon the application. Using different styles of connectors prevents this, as it becomes impossible for the connectors to be mated incorrectly. If it is not possible to select a different connector style, then we can use connector keying on some connector styles to also prevent inadvertent connection.

The selection of socket or plug connectors will also be made during the connectorization phase. Here electrical safety also plays a part. For example, the main power input, either ac or dc, should for safety be a plug at the equipment, ensuring that the mating cable is a socket and prevents live pins on the power supply cable from being able to short out inadvertently when the cable is being connected or otherwise used.

As engineers, we can mitigate the number of mating cycles using connector savers. These are adaptors that mount with the connector and then enable all of the cable mating to be with the mating interface on the saver. Using such an approach allows the main connector to have one mating cycle (the connection of the saver) while the saver absorbs the wear and tear as the cable is mated and de-mated as required during development. While connector savers can be expensive, they can be very useful during development to ensure that the mating connectors are not used past their expected number of mating cycles.

To help achieve higher currents through the connector, multiple pins can be used to share the current for the same power rail. De-rating reduces the electrical and thermal stresses to help to ensure its reliability. With connector pins, it is normal to de-rate connector pins to carry only half their rated value. For example, a pin rated at 1A would be allowed to pass a de-rated current of 0.5A; therefore, if a power supply requires one amp over a connector two pins are required to meet de-rating rules.

We also need to consider careful signal placement on the connector to prevent failures. One way to do this is by careful consideration of the pin placement and how unused pins are addressed.

To prevent accidental shorting, especially power and return, it is good practice to separate the power pins from the return pins. This is achieved by physical separation between the power and return pins. One example of this would be locating the power pins at one side of the connector and the return pins at the opposite side on a sub-D-type connector.

The signals are then allocated to the pins between the return and power pins at either end. If the application requires high reliability, it may be further necessary to isolate different groups of signals from each other.

In this case, guard pins may be used to ensure that the critical signal cannot short. Guard pins therefore surround the critical signal pin. These guard pins should be connected to ground via a high value resistor, ensuring that the critical signal cannot be shorted to either return or the supply rail.

### 1.8.3 Talking Among Yourselves: Internal ICDs

ICDs are not only of interest at the system or circuit card level; they are also very useful within the FPGA and the SW world. However, when an internal ICD is generated, we focus less on the physical connector and pin-out and more on the formation sharing based on standardized interfaces (e.g., AXI Memory Map for FPGA or shared memory structure in an SW-based solution).

Such internal ICDs will therefore detail memory maps, which define the accessible registers, along with detailing the bit fields for each register. A common reference point for the register mapping and bit fields means engineers developing different modules can easily develop solutions based around the internal ICD arriving at a solution that is capable of functioning as intended.

Internal ICDs are not limited to only memory mapped peripherals. Within a programmable-logic based solution, there will be several signal lines acting as communication and trigger sources.

When internal ICDs are developed for programmable logic-based solutions, we also need to define the clock domains on both sides of the transaction source and the destination. This enables both the design and verification teams to ensure that clock domain crossings (CDC), if required, are addressed safely.

## 1.9 Verification

After completing design work, and pouring hundreds upon hundreds (sometimes thousands upon thousands) of hours of effort into just coming up with design concepts, and then even more hours piled on top for doing design work and design reviews, a milestone is reached. New PCBs, mechanical enclosures, critical components, and sensors are finally ordered. This is a major milestone, but it is far from the end of a product development process. (There is a reason those pesky revision blocks exist on drawings after all.)

Having already touched upon the V-model and looked at how systems engineering and architecture work lays the groundwork for product design, implementation and ultimately validation. We should now consider the other V in V&V: verification. Recall the anecdote from earlier in the book that we mentioned; validation is often regarded as “was the right system built?” testing, and verification is often regarded as “was the system built correctly?” testing.

### 1.9.1 Verifying Hardware

Verifying hardware is absolutely a requisite part of delivering a robust and well-engineered product. Without verifying that, for example, power supplies are stable under load transients, but also under temperature extremes and input voltage (referred to as line voltage quite frequently), there may well be many peculiarities hiding in the design that would otherwise be undetected until fielded in a certain set of conditions. The same goes for every electronic circuit. Values change with temperature; values change with time. Doing up-front work to mitigate and reduce the risk and uncertainty associated with these changes is a big part of ensuring that a robust design is going to launch and not a marginal one that will be the cause of sleepless nights and heartburn for years to come.

In our new SoC platform, we certainly have many requirements that can be verified without exhaustive testing. Product weight and size, for example, can be verified in a matter of a few minutes with a prototype. Put the product on a scale, and grab a measuring tape; this could not be simpler.

In other cases, more thought is required and more care must be put into the test approach and verification method to really give a high confidence level in the product.

In the discussion on requirements, we discussed the various levels of verification and validation, ascending from the low levels of unit testing at (what we called) level 40 and rising up to level 10, where system validation is performed. The focus of this particular discussion is really around level 40 and level 30 testing for hardware.

Level 40 testing for hardware consists primarily of the verification of individual circuits (or small groups of circuits that comprise one function). Think of this like you might think of building a Lego house; you check each piece out to make sure that it is the right piece and matches what the instructions have laid out for use. In product design, the instructions against which we check are the requirements, the architecture, and the detailed design documentation that is generated during the design process (such as SPICE (Simulation Program with Integrated Circuit Emphasis) and IBIS (Input/output Buffer Information Specification) modeling, thermal modeling, and FEA (finite element analysis)).

### 1.9.2 How Much Testing Is Enough?

One obvious question as a starting point is how we determine what level of testing is required for a particular product. As often is the case in engineering, the right answer is that it depends. It depends on what type of system is being tested, what level of safety performance is required, what level of robustness is required, and what level of residual risk is acceptable for the product.

There are many normative standards related to the design, verification, and validation of products. Depending on the standard, there may be more or less prescriptive testing and documentation requirements laid out for designers to conform to. A few examples are:

- ISO 13489: Safety of Machinery [19];

- IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems [20];
- ISO 26262: Road Vehicles: Functional Safety [21].

### 1.9.3 Safely Navigating the World of Testing

If for no reason other than to have an example, let's use ISO 26262 as a reference document for some means of understanding how hardware test cases can be derived, and what kinds of testing can be used to verify hardware performance under even the harshest levels of scrutiny. It is important to note that these test case derivation methods and verification methods are really in place for safety-critical pieces of hardware inside of vehicles. They may be somewhat overkill for a nonsafety-relevant system, but performing these kinds of tests if the means are available is never a bad idea or bad practice. The more well understood the design is, the better off the entire team is in the long run.

Verification has several stages: planning, specification, and execution [21]. Let's look more closely at these stages. Verification planning essentially consists of considering what work output or work products will be produced. However, quite a lot of thought needs to go into this in order to do it properly. For example:

- What methods will be used;
- What the pass/fail criteria are;
- What environment(s) will be used (these environments should be controlled by a quality management system if they are used in safety critical testing);
- What tools will be used (these tools should always be part of a quality management system for safety-critical testing);
- What actions must be taken if items fail to meet the pass criteria during the course of testing.

There are other aspects of verification planning that should be considered as well during the planning phase. Critically looking at the adequacy of the test being performed is one of the more important ones from a verification perspective. If, for example, SensorsThink wants to verify their Ethernet interface and they do so by sending a simple series of ping packets to a host machine, that is probably not a very thorough representation of real-world use and may lead to a false sense of security. Conversely, using a physical layer qualification test package that measures things such as eye width, power spectral density, jitter, and distortion is surely a better set of metrics against which to measure performance.

The verification specification essentially selects and specifies the methods to be used. This can be any number of things: simulation scenarios, physical test cases, and even something as simple as a checklist. As long as the adequacy of the specification is justifiable and reasonable, there usually is not much else to be considered here. That is not to say that the specification should lack detail or clarity; preconditions (input data) and configurations used (a subset or superset of variants) are wonderful things to document. The test environment conditions are also important

bits of information to specify to allow for repeatability of testing should failures or anomalies occur.

The execution of the verification activity is exactly as it sounds: performing the testing, capturing the results, and comparing them to the previously specified pass/fail criteria. It is generally the best practice to review these results with a broader team of subject matter experts and to note any discrepancies and justification for them and to essentially make sure the team feels as though the product is standing firmly in the camp of robust.

### 1.9.4 A Deeper Dive into Derivation (of Test Cases)

Now that we understand the process of verification, let's consider hardware in a bit more detail, again with an eye to ISO 26262 for a guiding source of information. Deriving test cases can be complex, and sometimes the best approach is not readily apparent.

Table 1.4, taken from ISO 26262 [21], is given as a resource to help engineers derive meaningful test cases using a variety of methods that may be applicable to any number of requirements. We can see here that each method is scored over each system level in the table. (Note: The actual heading of the table is ASIL (Automotive Safety Integrity Level), a level A is the lowest safety level, and a level D is the highest safety level; for the purpose of our development, which is not a safety product, those details are not germane.) Items denoted with ++ are highly recommended for a system of the corresponding level; items denoted with + are recommended for a system of the corresponding level, and items denoted with an "o" are not recommended for a system of the corresponding level. For a lower-level system, a walk-through of the hardware design can be a perfectly suitable means of verifying the hardware design meets the requirements; for a higher-level system, an inspection of the design is preferred and a walk-through is not preferred. (During a walk-through of a system the developer explains the system to a third-part assessor; typically, this does not involve operating the item/element. During inspection, an examination of the work products is conducted against a defined procedure or checklist and typically ends with a review of the results.)

Another aspect of hardware verification is a form of integration testing, and there are essentially three methods of testing that can be performed to verify proper hardware robustness.

Functional testing aims at verifying that the specified characteristics of the item or device under test have been achieved. The item is given standardized input data (consider a pseudorandom bit sequence), which adequately characterizes the expected normal operation. The outputs are then measured, and their response is

**Table 1.4** ISO 26262 Part 5 Table 3 (Partial)

| *_Methods* | *System Level* | *A* | *B* | *C* | *D* |
|---|---|---|---|---|---|
| A | Hardware design walk-through | ++ | ++ | o | o |
| B | Hardware design inspection | + | + | ++ | ++ |
| D | Simulation | o | + | + | + |
| E | Development by hardware prototyping | o | + | + | + |

*Source:* [21].

compared with that given by the specification (consider an eye-mask diagram for the Ethernet).

Fault injection testing means intentionally introducing faults in the hardware product and analyzing the response of the hardware. Model-based fault injection (e.g., SPICE, IBIS, FEA, and thermal) is also applicable, especially when fault injection testing is overly complex to do at the hardware product level. (Consider trying to perform single-bit error testing that might be expected from cosmic rays.)

Electrical testing aims at verifying compliance with requirements within the specified voltage range. Again, the example of a power supply performing over a wide input range is a good one to consider.

The next level up from the last couple of examples is what we would consider to be integration level testing, where the hardware has been unit-tested, and these individual components and circuits have been verified. Combining this set of hardware with software that performs nominal operation of the hardware is really the first big integration milestone for an embedded system. Once this is ready and the underlying hardware is believed to be adequately tested, the overall embedded system (or, if in a complex enough system, perhaps a subsystem) is ready for robustness testing.

During environmental testing with basic functional verification, the embedded system is put under various environmental conditions during which performance is assessed. For the SoC platform that we are designing, this may consist of performing a strenuous Ethernet traffic test over a wide range of temperature and humidity and over a very long Ethernet cable. This type of test gives strong confidence in the robustness of an interface if the test is successful.

Expanded functional testing is intended to verify the response of the system to input conditions that are expected to occur only rarely or that are outside the nominal specification of the hardware, that consider receiving an invalid command, or that interrupt a data transmission intentionally. If the system can gracefully handle these conditions, that is a good indicator of robustness.

**Table 1.5** ISO 26262 Part 5 Table 11

| *_Methods* | *System Level* | *A* | *B* | *C* | *D* |
|---|---|---|---|---|---|
| A | Functional Testing | ++ | ++ | ++ | ++ |
| B | Fault Injection Testing | + | + | ++ | ++ |
| C | Electrical Testing | ++ | ++ | ++ | ++ |

**Table 1.6** ISO 26262 Part 5 Table 12 (Partial)

| *_Methods* | *System Level* | *A* | *B* | *C* | *D* |
|---|---|---|---|---|---|
| A | Environmental testing with basic functional verification | ++ | ++ | ++ | ++ |
| B | Expanded functional test | o | + | + | ++ |
| C | Accelerated life test | + | + | ++ | ++ |
| D | Mechanical endurance test | ++ | ++ | ++ | ++ |
| E | EMC and ESD test | ++ | ++ | ++ | ++ |

Accelerated life testing is a topic worthy of its own book (or several for that matter) and its own subject matter experts. The short version of the intent of the testing is to predict the behavior and performance of the product as it ages by exposing it to conditions outside of its normal operating conditions. This is a test method based on analytical models of failure mode acceleration and can provide valuable information related to the robustness and margin available in the design. These tests must be properly designed and executed to provide value [22].

Mechanical endurance testing is intended to stress the product to the point of either failure or noticeable physical damage. Things such as vibration testing, drop testing, and impact testing are often performed to ensure that the product can withstand the environment in which it is going to operate. Performing the testing to failure gives information on the available margin in the design. Consider a drop test for the product that must pass at 10 feet. If the test passes at 10 feet, but fails at 10 feet and 1 inch, the engineering team may want to take a closer look at the design. If the test does not fail until 15 feet, the team can probably rest easy on that particular test.

EMI/EMC testing is always a hot topic in engineering circles and again warrants a book of its own; and is certainly its own area of expertise. For most product engineers, this EMI/EMC testing is the gateway into the marketplace. Without passing those tests, your product will not be approved for sale. That is why they are such a good marker for product robustness and a necessary one at that. During this testing, your product will be zapped with high-voltage transients, have high-power RF energy pointed at it, and have extremely sensitive RF antennas listening to it while it operates. If the product cannot take the heat or if it makes too much noise, the engineering team has to go back to the drawing board.

Luckily for us, the requirements provided to us are relatively easy for which to derive test cases and it would not be too demanding of a task to put together a cogent and defensible test plan to show that the hardware portion of the system was performing robustly for the application.

SensorsThink is getting off the hook a bit easily in this regard, as they are not a safety product and they are not operating in overly harsh environments. That does not mean that we do not want to do a good job verifying our design. It just means that the level of scrutiny placed on the engineering team is not as high as it might be in another situation. In the hardware design part of our journey, we will take a closer look at how we can design our product to make testing easier. For now, this should put in perspective the necessity of proper testing and some additional food for thought on how SensorsThink may want to approach their product testing.

## 1.10 Engineering Governance

Engineering governance is probably one of the least fun elements of the design process, but at the same time it is one of the most important. Engineering governance defines how we progress from the initial concept of the project to the completed delivery and even beyond to end of life and safe disposal.

As you would expect, engineering governance is tied closely with the design life cycles and the design reviews, which are held as the program progresses. The engineering governance program defines which of the checklists and reviews need

to be completed for a design review. Typically, the engineering governance program within a company is overseen by a chief engineer. The chief engineer has dual roles in the governance program. The first role is to ensure compliance with the program by the various projects under development at anyone point in time. The second role is to ensure the engineering governance program itself stays fit for purpose and stays up to date with the latest technologies, standards, and approaches. This means that the chief engineer has to be able to attend conferences and standards bodies and, if necessary, run research projects to verify the suitability of new technology.

### 1.10.1 Not Just Support for Design Reviews

One of the key roles of the engineering governance program is to ensure that the necessary design maturity evidence is available for significant design reviews (e.g., preliminary design review). However, correctly implemented, it is much more than that. A correctly implemented governance program will institute a series of design standards and design checklists and reviews to ensure the suitability, maturity, and risk of the design is known and suitable for progression to the next stage of development. An example of this would be the progression from schematic design to layout; to ensure minimal risk, it is necessary to demonstrate correctness of the schematic design and documentation of the layout constraints while also aligning with mechanical and thermal design teams to ensure the correct PCB size, connector locations, skylines, and thermal considerations.

As such, the engineering governance scheme is multidisciplinary and covers the whole host of engineering disciplines from systems engineering to electronic design, mechanical design, and software development.

Engineering governance can be split into two distinct elements. These are design rules, which define how the different elements of the design must be completed, and compliance, which checks that the design rules have been followed.

### 1.10.2 Engineering Rule Sets

Typically, the design engineers and other members of the project team will use the design rules as they create the design. Compliance with the design rules will normally be verified by an experienced independent engineer, who signs off the design to progress to the next stage of design.

For an embedded system design, there will be several different design rules against which engineers must demonstrate compliance:

- *Requirement definition and capture:* Present a common approach to requirement capture defining the expected contents of the requirement specification covering functional, nonfunctional, and environmental requirements. These rules will also define the language to be used to define the requirements (e.g., use of shall, should, may, and would).
- *Schematic design rules:* Define the rules for drawing schematics that are not only functionally correct but also easily understandable by other engineers. These rules may enforce elements such as part selection and derating

to ensure compliance with company manufacturing facilities and reliability requirements.

- *Layout design rules:* The layout rules will contain a number of rules to enable a high yield in the board manufacturing process, while also enabling a high yield in the assembly process. The PCB layout rules will also include sections on high-speed signaling, mixed signal design, and safety (e.g., creepage and clearance distances).
- *RTL coding rules:* Defines the acceptable register-transfer level (RTL) language constructions that may be used to capture the design. These rules will also define acceptable verification techniques, along with change and configuration control.
- *SW coding rules:* Very similar to the RTL coding rules, this rule set will define acceptable software language constructs that may be used.
- *Product safety:* Contains the design rules and analysis that must be followed to achieve regulatory compliance sign-off (e.g., IEC61508, ISO26262).

Complying with the design rule sets as the design progresses can be challenging, as often the rules are defined in published documents, which, despite best intentions, are easily overlooked.

### 1.10.3 Compliance

The best way to ensure compliance with engineering governance rules is to incorporate them within the CAD tools we are using to implement our embedded system. Within schematics and layout tools, we can implement rules using constraints and design for manufacturing settings. If additional checks are required, it is possible to create scripts in Python or Tool Command Language (TCL) to implement these additional checks. When it comes to ensuring compliance with the RTL and software coding standards, linting tools can be used to ensure compliance against rule sets that include naming conventions and design rules.

The automation of rule sets directly in the CAD tool itself obviously increases the compliance with the rule set. It also makes demonstrating compliance with the rule set easier when it is time for design sign-off.

If automation cannot be achieved, then checklists can be used to ensure that the necessary design rules have been complied with. To ensure repeatability, the checklists should be stored under a configuration control system, enabling the design team to access the latest version.

### 1.10.4 Review Meetings

Regardless of the method of creation, the design rules review needs to be reviewed by the engineering team and the chief engineer or his or her nominated representative when a subject matter expert is required (e.g., FPGA, digital electronics, high speed).

When performing the design rules review, there will often be rules that do not comply with the rule. This is to be expected, as the design rules are created to ad-

dress a wide range of developments. There will also be corner cases that do not align with the checklist.

This is where the roles of the chief engineer and independent expert come into play along with the engineering team. Each of the rule's violations needs to be analyzed and allocated to one of two categories:

- *Must be fixed:* These rule violations must be addressed before the next stage of the project can be started.
- *Will not be fixed:* These rule violations do not need to be addressed. For each violation, a detailed explanation of the reasoning being the exception must be recorded.

Once the design rules review has been completed, the design review notes, categorization of rule violations must be stored within a change management or product life-cycle management system. This provides a record of the review occurring and makes the data easily available for higher-level design reviews and certification reviews.

## References

[1] Cooper, R., *Winning at New Products: Creating Value Through Innovation*, 3rd edition, New York: Basic Books, 2011.

[2] Sparx Systems Pty Ltd., "Enterprise Architect 14," August 22, 2019. https://www.sparxsystems.com/products/ea/14/index.html.

[3] Reliable Software Technologies, "Mew Models for Test Development," 1999. http://www.exampler.com/testing-com/writings/new-models.pdf.

[4] Forsberg, K., and H. Mooz, "The Relationship of Systems Engineering to the Project Cycle," *National Council on Systems Engineering (NCOSE) and American Society for Engineering Management (ASEM) Conference*, Chattanooga, TN, October 21, 1991. https://web.archive.org/web/20090227123750/http://www.csm.com/repository/model/rep/o/pdf/Relationship%20of%20SE%20to%20Proj%20Cycle.pdf.

[5] INCOSE, "About: INCOSE," https://www.incose.org/about-incose.

[6] Requirements Working Group, International Council on Systems Engineering (INCOSE), "Guide for Writing Requirements," San Diego, CA, 2012.

[7] International Electrotechnical Commission (IEC), "Available Basic EMC Publications," August 22, 2019. https://www.iec.ch/emc/basic_emc/basic_emc_immunity.htm.

[8] RoHS Guide, "RoHS Guide," August 22, 2019. https://www.rohsguide.com/.

[9] European Commission, "REACH," August 22, 2019. https://ec.europa.eu/environment/chemicals/reach/reach_en.htm.

[10] Ericsson Inc., "Reliability Prediction Procedure for Electronic Equipment," SR-332, Issue 4, March 2016. https://telecom-info.telcordia.com/site-cgi/ido/docs.cgi?ID=SEARCH&DOCUMENT=SR-332&.

[11] BKCASE Editorial Board, "The Guide to the Systems Engineering Body of Knowledge (SEBoK), v. 2.0," Hoboken.

[12] SEBoK Authors, "Scope of the SEBoK," August 18, 2019. https://www.sebokwiki.org/w/index.php?title=Scope_of_the_SEBoK&oldid=55750.

[13] SEBoK Authors, "Product Systems Engineering Background," August 18, 2019. https://www.sebokwiki.org/w/index.php?title=Product_Systems_Engineering_Background&oldid=55805.

[14] SEBoK Authors, "System Design," August 18, 2019. https://www.sebokwiki.org/w/index.php?title=System_Design&oldid=55827.

[15] SEBoK Authors, "System Analysis," August 18, 2019. https://www.sebokwiki.org/w/index.php?title=System_Analysis&oldid=55956.

[16] SEBoK Authors, "Logical Architecture Model Development," August 18, 2019. https://www.sebokwiki.org/w/index.php?title=Logical_Architecture_Model_Development&oldid=55698.

[17] SEBoK Authors, "Physical Architecture Model Development," August 18, 2019. https://www.sebokwiki.org/w/index.php?title=Physical_Architecture_Model_Development&oldid=55874.

[18] IEEE, "1149.7-2009 - IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture," February 10, 2010. https://ieeexplore.ieee.org/servlet/opac?punumber=5412864.

[19] International Organization for Standardization, "ISO 13849-1:2015," August 22, 2019. https://www.iso.org/standard/69883.html.

[20] International Electrotechnical Commission, "Functional Safety and IEC 61508," August 22, 2019. https://www.iec.ch/functionalsafety/.

[21] International Organization for Standardization, "ISO 26262-1:2018," August 22, 2019. https://www.iso.org/standard/68383.html.

[22] McLean, H. W., *HALT, HASS, and HASA Explained: Accelerated Reliability Techniques, Revised Edition*, Milwaukee, WI: American Society for Quality, Quality Press, 2009.

CHAPTER 2

# Hardware Design Considerations

*Well-designed, tested, and documented underlying hardware is key to enabling high-performance systems.*

## 2.1 Component Selection

The System on Chip (SoC) Platform project is well underway at this point at SensorsThink. Requirements have been established (refer to Section 1.5: Requirements), system architecture has been defined (refer to Section 1.6), and the engineering domains (hardware, software, firmware, mechanical) are starting to do their engineering wizardry to turn those requirements and ideas into a real-life functioning product.

For hardware design, this process starts with component selection, more specifically, key component selection. Let's explore what we mean by a key component in a bit more detail before we dive into the selection process for some of the key components in our design.

For the sake of discussion, let's assume that the following two criteria are how we can identify a key component:

1. A component that, once selected, will not have a viable second source option or drop-in replacement. This excludes things such as resistors, capacitors, and diodes (with a few exceptions).
2. A component that is crucial to realizing the required functionality of the product and directly traces back to requirements. In our design, a great example of this is the selection of the actual sensors that will be providing data to the processing system.

### 2.1.1 Key Component Identification for the SoC Platform

Key components are important to choose early because they drive the rest of the design process directly. Their interfaces, power requirements, and support circuitry become critical aspects of the design to consider in order to ensure a successful hardware platform. Let's take another look back at the overall system architecture diagram to see if we can identify the key components (Figure 2.1).

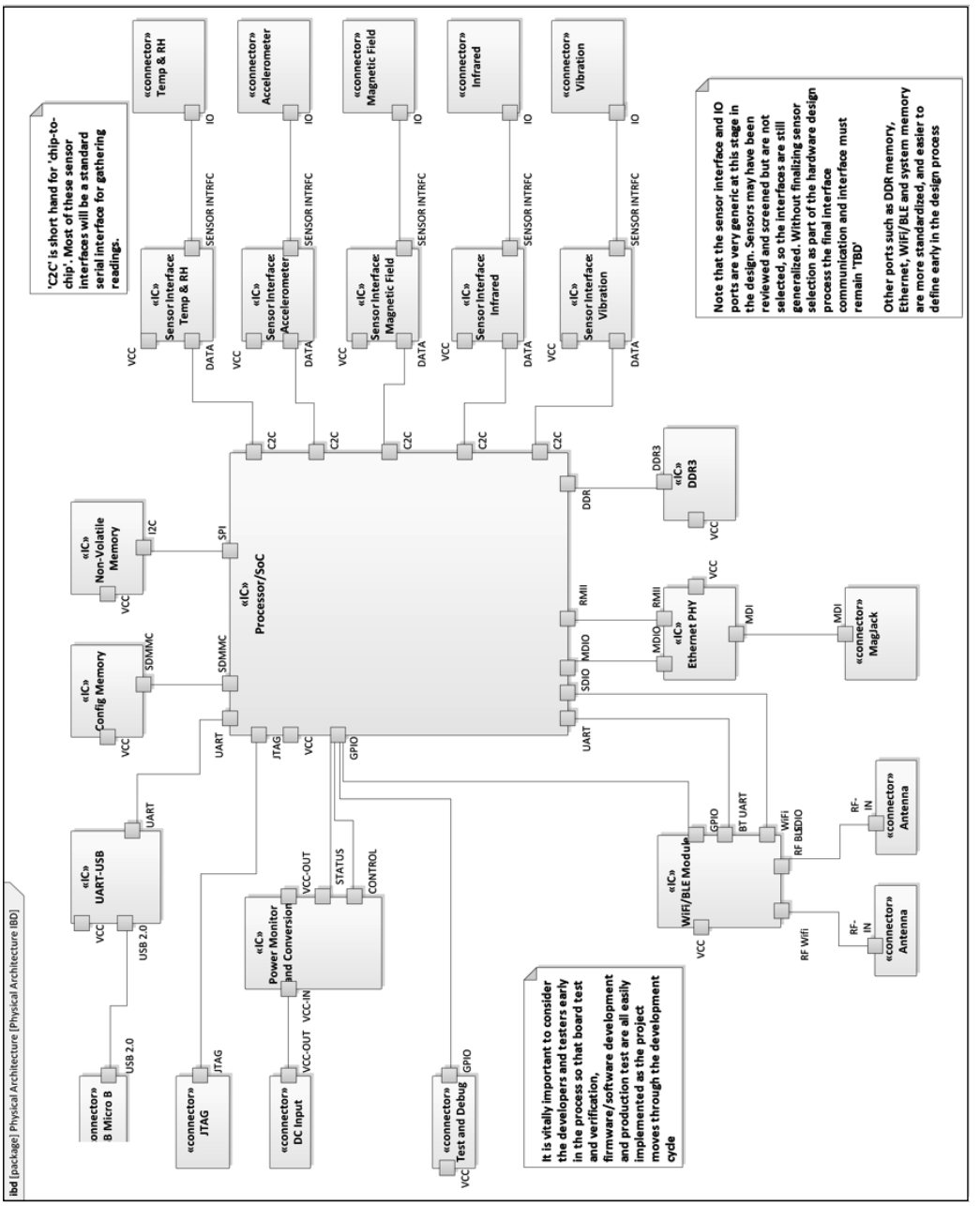



Figure 2.1 Hardware system architecture.

1. At the heart of the design is a processor/SoC; this is a key component for us. While there may be a migration path (more on this later) for our processor of choice, it is likely to not be a true drop-in replacement. In other words, another processor/SoC may fit physically on the printed circuit board (PCB), but likely would require some tweaks to firmware to properly run. This could lead to issues with production and tracking different product variants that exist once multiple devices are in customers' hands. Typically, unplanned variants are things that we want to avoid. The processor/SoC is also absolutely vital to realizing the functionality that we want to have in this SoC platform. It could be argued that it is as important as the sensors themselves.
2. The USB-UART converter, Wi-Fi/Bluetooth module, and Ethernet PHY are also key components solely due to the low probability of finding a drop-in replacement for these parts. The Wi-Fi/Bluetooth module and Ethernet PHY are also very important pieces of functionality for the platform. One could argue that the USB-UART converter is nice to have for a fielded product, but it is essential for development and test.
3. The sensors are key components. The performance of the sensors will be nearly solely responsible for the platform meeting the level 10 requirements. They are not depicted on the architecture diagram at the system hardware level, but they do appear in the context diagram (Figure 2.2).

It is natural to wonder why some of the other components are not considered key components, the answer (as it usually is) is that it depends. Things like memory can typically be found from at least two suppliers in a true drop-in replacement because of the JEDEC standards around memory.

In JEDEC's own words [1]:

> JEDEC's collaborative efforts ensure product interoperability, benefiting the industry and ultimately consumers by decreasing time-to-market and reducing product development costs.

The other major architectural blocks that we did not identify as key components are connectors and power conversion and monitoring. The reasoning here is that these components will be driven by the other components around them. We need to know the power requirements for the key components to design a power system, and we need to know what kinds of connectors may be required to interface to sensors or a debug adapter for whatever processor or SoC that we choose to be the core of the design.

So now that we have identified the key components that will drive our hardware design forward, there are a couple other key considerations to have in mind before we start to scour the internet and component supplier websites for the perfect parts:

1. *Component life cycle:* How long is the component expected to be available for sale? When was it introduced? What is the current life-cycle status? You never want to start a development project with a component that is already slated to go end-of-life. Just like we discussed in Section 1.4: Product

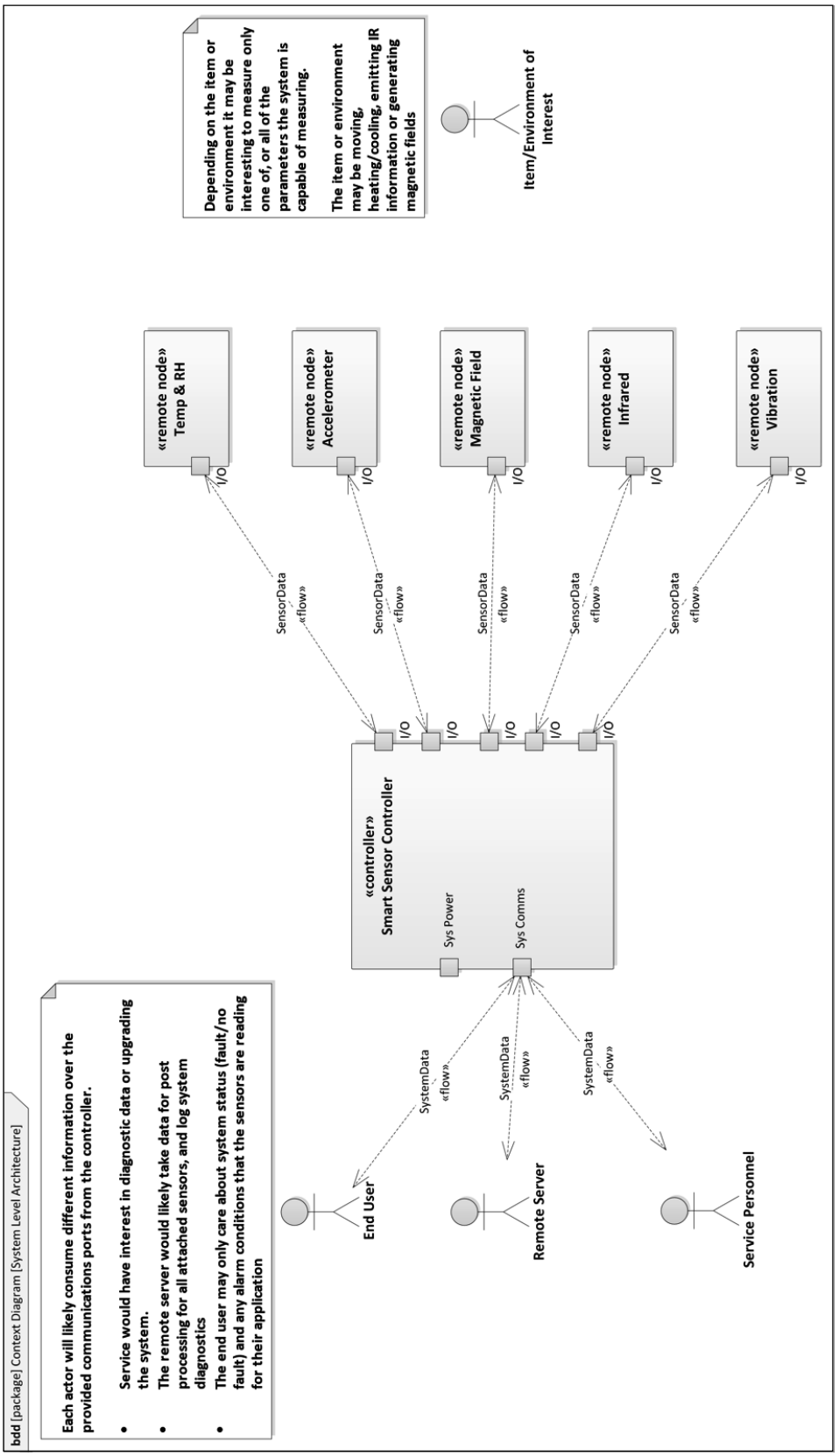



**Figure 2.2** System context diagram.

Develoment: After Launch, the component manufacturer has their own life-cycle considerations and product planning going on internally. It is important to know where your particular component sits on that roadmap.

2. *Component temperature range:* In what ambient temperature range is the component rated for operation? Does this temperature range meet our product requirements? If it does not, it is not the right component.
3. *Component/device packaging:* For integrated circuits (ICs), this is a key factor, as it can drive both the complexity and the cost of the PCB design. It is more costly to design a PCB with high-density interconnect (HDI) like micro-vias than it is to design a PCB with all through-hole vias. Sometimes our hands are forced in this regard, but it is something worth considering.

### 2.1.2 Key Component Selection Example: The SoC

We covered the rationale behind why a SoC made sense for this product (versus multiple devices or a discrete microprocessor) in Section 1.6. To summarize, it is essentially due to the inclusion of FPGA fabric for parallel sensor data processing and an applications class ARM processor to provide a Linux operating system for our connectivity requirements. In the FPGA/SoC space, there are several suppliers to choose from: Intel (formerly Altera), Xilinx, and Microsemi.

There are certainly more suppliers providing SoC solutions. for the sake of discussion, we will limit it to these three. Some other suppliers provide solutions with microprocessors and DSPs (digital signal processors) for example. These devices certainly have their place in the design world, but they are not relevant for our discussion and design consideration.

For SensorsThink, some important requirements for choosing a SoC device are as follows:

1. Ability to run Linux on an ARM A-Class processor;
2. Inclusion of FPGA fabric for sensor fusion;
3. Support for DDR3 memory;
4. Support for Gigabit Ethernet;
5. Inclusion of security features;
6. Inclusion of standard peripheral communications (I2C, SPI, UART, and GPIO).

Typically, when doing a key component selection exercise, something like a Pugh matrix can be really useful to make what can sometimes devolve into a "religious" debate about Supplier A versus Supplier B, a more data-driven decision. Let's do a quick Pugh matrix for the SoC selection and see if it helps us narrow the field of choices (Figure 2.3).

There are many very high-end and costly SoC devices on the market that could handle our application very easily. So we could just cherry-pick the latest and greatest new SoC and call it a day. In real-world applications and product development, cost is always a key factor. Part of choosing components wisely is choosing the component that has just the right feature set for your use. However, as part of this

| | Criteria 1 | Criteria 2 | Criteria 3 | Criteria 4 | Criteria 5 | |
|---|---|---|---|---|---|---|
| Criteria Description | Ability to run Linux on an ARM A-Class processor | Inclusion of FPGA fabric | DDR3 Support | 10/100/1000 Ethernet Support | Security Features | |
| | Criteria 1 | Criteria 2 | Criteria 3 | Criteria 4 | Criteria 5 | Weighted Score |
| Weight | 5 | 4 | 1 | 2 | 3 | 15 |
| | 33% | 27% | 7% | 13% | 20% | 100% |
| Options | Criteria 1 Scores | Criteria 2 Scores | Criteria 3 Scores | Criteria 4 Scores | Criteria 5 Scores | |
| Microsemi SmartFusion 2 SoC | 0 | 1 | 2 | 3 | 2 | 18 |
| Intel Cyclone V SoC | 3 | 2 | 3 | 3 | 1 | 35 |
| Xilink Zynq-7000 SoC | 3 | 3 | 3 | 3 | 3 | 45 |

**Figure 2.3** SoC selection: weighted Pugh matrix.

exercise, the cost will not be part of the Pugh matrix so as to keep the focus on technical decision making.

A quick review of the three supplier SoC offerings brings us our three options to put into the Pugh matrix:

1. Microsemi SmartFusion2 SoC FPGA family;
2. Intel Cyclone V SoC FPGA family;
3. Xilinx Zynq-7000 SoC FPGA family.

A basic Pugh matrix starts by defining a set of criteria that are scored and then summed. This gives each option a final score, which can then be ranked.

A weighted Pugh matrix evolves the concept further by weighting the criteria in order of importance. The more important the criteria, the higher the weight that it should be given (a higher number by which to multiply the ranking). This way, the resulting final score more accurately captures the fit of each option when the values are summed.

All of the information for the Pugh matrix was gathered by sorting through the device family documentation for each device type. This can be a time-consuming exercise and can require contacting applications engineers to really help with sorting through all the details and nuances of each family. For SensorsThink, the clear

| NOTES |
| --- |
| *ARM Cortex M3 Processor, no A-Class Option<br>*FPGA logic element count is the lowest among the three options (150K)<br>*The DDR3 controller is a 'soft' controller, versus a 'hardened IP' controller in the other options<br>*DPA Hardened, AES256, SHA256 |
| *ARM Cortex A9 Processor, single and dual core options<br>*FPGA logic element count up to 301K<br>*Some devices have 2 'hardened IP' DDR3 controllers in the FPGA fabric as well as one for the ARM Cortex A9<br>*AES256, SHA256 (requires 3rd party IP) |
| *ARM Cortex A9 Processor, single and dual core options<br>*FPGA logic element count up to 444K in mid-range devices<br>*The only hardened DDR controller is for the ARM Cortex A9. A soft IP controller can be placed into the FPGA fabric.<br>*RSA Authentication of First Stage Boot Loader, AES and SHA 256b Decryption and Authentication for Secure Boot for the ARM Cortex A9<br>*AES & SHA 256b Decryption & Authentication for Secure Programmable Logic Config |

**Figure 2.4** SoC selection: weighted Pugh matrix notes.

winner here is the Xilinx Zynq-7000 series of devices, and because we took an analytical approach to the selection it is difficult to argue against the decision.

From this key decision, we can now start to build the framework around our main SoC device. Choose support components such as:

- DDR3 memory (size, width, and configuration);
- Nonvolatile memory (size and interface);
- Configuration/filing system memory (size and interface);
- Ethernet PHY (interface type).

Before we get too far ahead of ourselves, we should also consider the component life cycle, temperature range, and packaging type. We previously identified these as important, so a review of the options is certainly worth a brief discussion.

#### 2.1.2.1 Xilinx Zynq-7000 SoC Family Options

When choosing a device that is part of a larger family, it is easy to be overwhelmed with options; package options, peripheral options, speed grade options, and migration path options can really make the decision seem daunting (Figure 2.5).

# Zynq®-7000 SoC Family

| | | Cost-Optimized Devices | | | | | | Mid-Range Devices | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Device Name | Z-7007S | Z-7012S | Z-7014S | Z-7010 | Z-7015 | Z-7020 | Z-7030 | Z-7035 | Z-7045 | Z-7100 |
| | Part Number | XC7Z007S | XC7Z012S | XC7Z014S | XC7Z010 | XC7Z015 | XC7Z020 | XC7Z030 | XC7Z035 | XC7Z045 | XC7Z100 |
| Processing System (PS) | Processor Core | Single-Core ARM® Cortex™-A9 MPCore™ Up to 766MHz | | | Dual-Core ARM Cortex-A9 MPCore Up to 866MHz | | | Dual-Core ARM Cortex-A9 MPCore Up to 1GHz[1] | | | |
| | Processor Extensions | NEON™ SIMD Engine and Single/Double Precision Floating Point Unit per processor | | | | | | | | | |
| | L1 Cache | 32KB Instruction, 32KB Data per processor | | | | | | | | | |
| | L2 Cache | 512KB | | | | | | | | | |
| | On-Chip Memory | 256KB | | | | | | | | | |
| | External Memory Support[2] | DDR3, DDR3L, DDR2, LPDDR2 | | | | | | | | | |
| | External Static Memory Support[2] | 2x Quad-SPI, NAND, NOR | | | | | | | | | |
| | DMA Channels | 8 (4 dedicated to PL) | | | | | | | | | |
| | Peripherals | 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO | | | | | | | | | |
| | Peripherals w/ built-in DMA[2] | 2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO | | | | | | | | | |
| | Security[3] | RSA Authentication of First Stage Boot Loader, AES and SHA 256b Decryption and Authentication for Secure Boot | | | | | | | | | |
| | Processing System to Programmable Logic Interface Ports (Primary Interfaces & Interrupts Only) | 2x AXI 32b Master, 2x AXI 32b Slave<br>4x AXI 64b/32b Memory<br>AXI 64b ACP<br>16 Interrupts | | | | | | | | | |
| Programmable Logic (PL) | 7 Series PL Equivalent | Artix®-7 | Artix-7 | Artix-7 | Artix-7 | Artix-7 | Artix-7 | Kintex®-7 | Kintex-7 | Kintex-7 | Kintex-7 |
| | Logic Cells | 23K | 55K | 65K | 28K | 74K | 85K | 125K | 275K | 350K | 444K |
| | Look-Up Tables (LUTs) | 14,400 | 34,400 | 40,600 | 17,600 | 46,200 | 53,200 | 78,600 | 171,900 | 218,600 | 277,400 |
| | Flip-Flops | 28,800 | 68,800 | 81,200 | 35,200 | 92,400 | 106,400 | 157,200 | 343,800 | 437,200 | 554,800 |
| | Total Block RAM (# 36Kb Blocks) | 1.8Mb (50) | 2.5Mb (72) | 3.8Mb (107) | 2.1Mb (60) | 3.3Mb (95) | 4.9Mb (140) | 9.3Mb (265) | 17.6Mb (500) | 19.2Mb (545) | 26.5Mb (755) |
| | DSP Slices | 66 | 120 | 170 | 80 | 160 | 220 | 400 | 900 | 900 | 2,020 |
| | PCI Express® | — | Gen2 x4 | — | — | Gen2 x4 | — | Gen2 x4 | Gen2 x8 | Gen2 x8 | Gen2 x8 |
| | Analog Mixed Signal (AMS) / XADC[2] | 2x 12 bit, MSPS ADCs with up to 17 Differential Inputs | | | | | | | | | |
| | Security[3] | AES & SHA 256b Decryption & Authentication for Secure Programmable Logic Config | | | | | | | | | |
| | Speed Grades: Commercial | -1 | | | -1 | | | -1 | | | -1 |
| | Speed Grades: Extended | -2 | | | -2,-3 | | | -2,-3 | | | -2 |
| | Speed Grades: Industrial | -1, -2 | | | -1, -2, -1L | | | -1, -2, -2L | | | -1, -2, -2L |

**Figure 2.5** Xilinx Zynq-7000 SoC family overview.

A great place to start is a product table (or product selection guide); suppliers will often summarize many of these options for us visually to make the information easier to consume. Another good resource can be parametric search tables from suppliers of devices, which allow the designer to sort and filter parts based on specific functionality, package size, or environmental ratings, among other things.

For SensorsThink, a dual-core ARM is the most sensible starting point. The reasoning behind this is that it allows for things such as asymmetric multiprocessing. One core can run the Linux operating system, and the other can run FreeRTOS (or another real-time operating system). This really opens up options for optimizing performance in the end application. You could also dedicate both cores to Linux if necessary. The flexibility is the attractive selling point for a dual-core system.

That leaves two device branches to explore: the cost-optimized family, built on Artix-7 fabric, and the mid-range family based on Kintex-7 fabric.

The analysis of Kintex-7 fabric versus Artix-7 fabric is out of the scope of our discussion; however, cost is a very easy barometer to indicate what might make sense given the features and trade-offs. A quick search on a distributor website such as Digikey or Mouser is a great way to get a rough-order estimate of the cost of each device relative to the other. Let's examine the highest-end cost-optimized device and the lowest-end mid-range device, the Z-7020 and Z-7030, respectively (Figure 2.6).

That is quite the increase in cost, essentially twice the cost for jumping the barrier from cost-optimized to mid-range. From Figure 2.5, we can see that some key differentiators are PCI Express lanes, logic elements, and DSP slices. In our application, we are doing some sensor fusion and some mathematical operations will certainly occur. However, without compelling input from a software engineer or systems engineer, the additional cost is hard to justify. For this reason, we can push closer to arriving at the major milestone of choosing the SoC for our platform. The Z-7010, Z-7015, and Z-7020 are the group of devices left to choose from (Figure 2.7).

Since this is an early stage of the development process, having a good migration path within our device family is something for which to aim if at all possible.

By examining this Figure 2.7, we can see that a CLG400 footprint provides us with a tremendous amount of flexibility, which could lead to cost reduction at the end of the project if we find that the initial selection has excessive horsepower for our needs. This extends down to the Z-7007S, a single core version of the Zynq-7000, up to the Z-7020, the highest density device available in our dual-core cost optimized family of components.

| Part Number | Device | Temperature | Package | Speed (Processor) | Cost (Qty: 1) |
|---|---|---|---|---|---|
| XC7Z020-1CLG400I | Z-7020 | -40°C-100°C | 400LFBGA | 667MHz | $131 |
| XC7Z020-2CLG400I | Z-7020 | -40°C-100°C | 400LFBGA | 766MHz | $140 |
| XC7Z030-1FBG484I | Z-7030 | -40°C-100°C | 484BBGA | 667MHz | $252 |
| XC7Z030-2FBG484I | Z-7030 | -40°C-100°C | 484BBGA | 800MHz | $301 |

**Figure 2.6** Cost comparison: Z-7020 and Z-7030.

**Zynq®-7000 Device Footprint Compatibility**
HR I/O, PS I/O, and GTP Transceivers — 13mm–35mm

| PCB Footprint Dimensions (mm) | 13x13 | 17x17 | 19x19 | 19x19 | 23x23 | 27x27 | 27x27 | 31x31 | 35x35 |
|---|---|---|---|---|---|---|---|---|---|
| Unique Footprint | CLG225 | CLG400 | CLG484 | CLG485 | FBG484 | FBG676 | FFG676 | FFG900 | FFG1156 |
| Z-7007S | 54, 84, 0 | 100, 128, 0 | | | | | | | |
| Z-7012S | | | | 150, 128, 4 | | | | | |
| Z-7014S | | 125, 128, 0 | 200, 128, 0 | | | | | | |
| Z-7010 | 54, 84, 0 | 100, 128, 0 | | | | | | | |
| Z-7015 | | | | 150, 128, 4 | | | | | |
| Z-7020 | | 125, 128, 0 | 200, 128, 0 | | | | | | |

**Figure 2.7** Xilinx Zynq-7000 SoC footprint compatibility.

Note: The shading indicates the footprint compatibility range. The numbers in each cell are the number of I/O available to the FPGA, processor system, and the number of high-speed GTP transceivers.

Based on this, we will start our development project with the Z-7020 device in a CLG400 package. We will also need the industrial temperature range to accommodate the requirement to operate down to –10°C. We will also start in the –2 speed grade, the faster of the two options available in our temperature range. This provides us the most processing power in our migration path and allows us to see if we can migrate down as we near completion of the development.

#### 2.1.2.2 Xilinx Zynq-7000 SoC Support Components

*Example: DDR3 Memory*
Choosing the amount of DDR3 memory, the memory bus width is driven directly by the capabilities of our SoC. The device family datasheet is a great place to start to find this type of information. On the first page of the Zynq-7000 series datasheet, the memory interface is pretty well constrained for us:

- 16-bit or 32-bit interfaces to DDR3, DDR3L, DDR2, or LPDDR2 memories;
- 1 GB of address space using single rank of 8-, 16-, or 32-bit-wide memories.

If we want to maximize our potential performance, a 32-bit-wide DDR3 bus with 1 GB of memory is the route to take. Companies such as Micron provide some helpful documentation for memory selection for widely used device families (such as ones from Xilinx or Intel). The MT41K/MT41J device families are listed as compatible and validated with the Zynq-7000 device family; it is available in a 16-bit-wide data interface with a density of up to 8 Gb (Gigabits) or 1 GB (Gigabyte). If we use two 16-bit-wide, 4-Gb density devices in parallel, we will have 1 GB of total memory available (Figure 2.8). Before finalizing this selection, it can also be helpful

| Technology | Component Configuration | Number of Components | Component Density | Total Width | Total Density |
|---|---|---|---|---|---|
| DDR3/DDR3L | X16 | 2 | 4 Gb | 32 | 1 GB |
| DDR2 | X8 | 4 | 2 Gb | 32 | 1 GB |
| LPDDR2 | X32 | 1 | 2 Gb | 32 | 256 MB |
| LPDDR2 | X16 | 2 | 4 Gb | 32 | 1 GB |
| LPDDR2 | X16 | 1 | 2 Gb | 16 | 256 MB |

**Figure 2.8** Zynq-7000 Technical Reference Manual: DDR Memory Section Table 10. (Adapted from Xilinx.)

to review more detailed documentation from Xilinx for the Zynq-7000 series to see any potential missed nuances for implementing DDR3. The technical reference manual for the device family is the deepest dive that you can take, and it is where we will go next.

The first row in this table confirms that our design intent is valid. If we choose a 4-Gb component at 16 bits wide and place 2 of them in parallel, we will have a 32-bit-wide data bus at 8 Gb (1 GB) of total memory density. A suitable part number for us to use is Micron MT41K256M16TW-107; this part has the right bus width and density, is on the supported list from Micron, and is currently an active part. This is one more design decision checked off the list for our SoC platform.

#### 2.1.2.3 Other Support Components for the Zynq-7000

A similar approach should be taken for the other key support components for the Zynq-7000. Start with basic parameters, find supported devices, and further examine the technical reference manual and even reference designs and evaluation kits to make sure that your selection is sound. For devices such as configuration memory, it is especially important as this often needs to be supported by the development environment for the SoC or processor. Careful selection of these components can really make the different in getting a design up and running in a quick manner.

### 2.1.3 Key Component Selection Example: Infrared Sensor

One of the sensor technologies for SensorsThink's new product platform is the infrared image spectrum. If we recall back to Section 1.5: Requirements, we know that the infrared spectrum requirements were fairly well defined (Figure 2.9).

There are several players in the infrared sensor space, but really there are not many alternative options for original equipment manufacturer (OEM) sensor design utilizing infrared imaging aside from FLIR.

> Note that there is one other commercially available option from Melexis, but it does not have the required plane array size, as it is only 32 × 24 pixels.

The FLIR Lepton 3.5 image sensor meets all of our stated requirements:

- Spectral range: 800 nm to 1,400 nm in the longwave infrared (LWIR) spectrum;

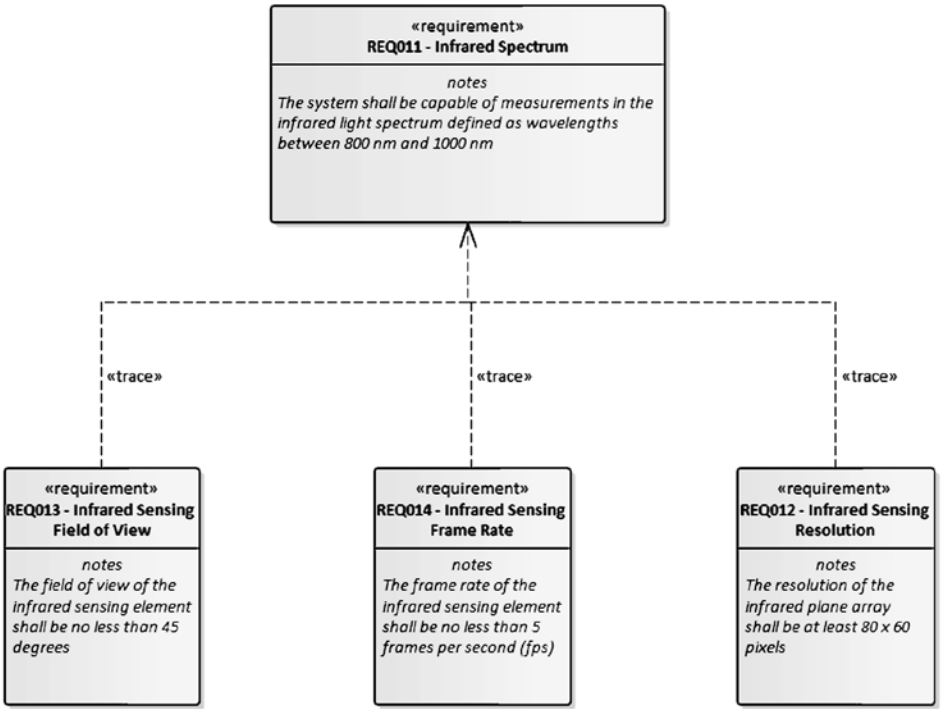



**Figure 2.9** Infrared spectrum requirements.

- Field of view (FOV): 57° horizontal, 71° diagonal;
- Frame rate (over SPI): 8.7 Hz (effective);
- Plane array size: 160 × 120 pixels.

Because of the limited options available, the selection of this sensor becomes merely an exercise in finding available options. If there was no sensor available that met all of the stated requirements, the hardware engineers and product management at SensorsThink would need to sit down and discuss alternative paths forward. That could mean reducing the requirement to align with available sensors, creating a partnership with a specialist partner company to develop a sensor, or removing the sensing technology altogether. Luckily for us in the hardware group, the requirements are able to be met.

#### 2.1.3.1 Infrared Sensor Interface

After choosing this sensor, we can then start to examine the impact of the choice on our system. This particular sensor requires a serial peripheral interface (SPI) communications link to transfer video data, an inter-integrated circuit (I2C) communications link to configure and control the sensor device, and some general-purpose inputs and outputs (GPIO) to monitor status. These things are important to note when putting together a more detailed hardware architecture diagram. We will cover hardware architecture in Section 2.2.

### 2.1.4 Key Component Selection: Finishing Up

All of the sensors should be selected in a similar way: refer back to requirements, find a sensor (or sensors) that are candidates, and, if necessary, create Pugh matrices to aid in decision making. Once all of the key components are selected, the hardware architecture can be fleshed out in more detail, which feeds directly into the design process: things such as power consumption, power supply voltages that are required (and current for each of these supply voltages), and SoC IO assignment. It is from these key components that our hardware design really begins to take shape.

At the end of this component selection exercise is a list summarizing the components that we have chosen for our SoC Platform project and key bullet-point items for the examples that we did not explore in detail.

- SoC: Xilinx Zynq-7000 Series; XC7Z020-2CLG400I;
- DDR3: (2) Micron 16-bit/4-Gb MT41K256M16TW-107;
- Configuration Memory, Micron MT25QL128ABA1EW7-0SIT:
  - Validated by Micron for the Zynq-7000;
  - 128-Mb size is large enough for our full migration path.
- Filing system storage: Micro SD Card:
  - SDHS (SD high speed) and SDHC (SD high capacity) card standards are acceptable for the Zynq-7000.
- Nonvolatile Memory (with Crypto-Security): Microchip AT88SC0808CA;
- Ethernet PHY: Microchip KSZ9031:
  - RGMII interface supported by the Zynq-7000;
  - Supported by the Xilinx PHY support in Linux kernel configuration.
- USB 2.0: Microchip USB3320:
  - ULPI interface supported by the Zynq-7000;
  - Used on the ZC702 evaluation board.
- USB to UART Converter: FTDI FT230X:
  - Simple UART interface;
  - Widely used industry standard part.
- Wi-Fi/Bluetooth module: Microchip ATWILC3000:
  - SPI, SDIO, I2C, and UART interfaces are supported by the Zynq-7000;
  - Used on the Ultra96 evaluation board.
- Local relative humidity/temperature sensor: Sensiron SHTW2:
  - I2C interface supported by the Zynq-7000;
  - Senses temperature and relative humidity within our required ranges.
- Accelerometer: ST Micro LSM6DS3:
  - SPI supported by the Zynq-7000;
  - Senses angular and linear acceleration within our required ranges in 3 axes.

- Infrared sensor: FLIR Lepton 500-0771-01:
  - Lepton 3.5.
- Magnetic field sensor: ST Micro LSM303AH:
  - SPI and I2C interfaces supported by the Zynq-7000;
  - Senses magnetic field strength (in gauss) within our required ranges in 3 axes.
- Vibration sensor: Analog Devices ADcmXL3021:
  - SPI supported by the Zynq-7000;
  - Measures vibrational force and frequency within our required ranges in 3 axes.
- Remote relative humidity/temperature sensor: Omega HX94BC:
  - Senses temperature and relative humidity within our required ranges in a single unit;
  - Provides output in analog voltage, which can be conditioned and sampled by the internal analog to digital converter (ADC) of the Zynq-7000.

## 2.2 Hardware Architecture

The next step in hardware design is to start to put the design into place. Finally, after months of research and legwork, requirements, and component selection, the design work starts to take shape. All the work that has been put into the project so far is essential for the hardware designer at SensorsThink to start the nitty-gritty design process. Without key components chosen and a system-level concept in place, the hardware design process is akin to assembling a puzzle with the pieces upside down. However, with those things well-defined, the puzzle pieces turn right-side up. It may be a 100-piece kids' puzzle sometimes and a 2,000-piece monster puzzle at others, but it is always easier to take that task on with the pieces flipped over.

A natural question at this juncture would be regarding the difference between the system-level physical architecture that was done previously and this current architectural design step. The diagrams can often look quite similar; however, this not always the case. In complex systems with more abstract black boxes, the diagrams can become more layered, with a system of systems architecture starting to reveal itself as the implementation path forward.

In our project, this is not the case, but the highest return on the investment for going through this part of the design process is always achieved by doing it upfront, before the real design work has started.

### 2.2.1 Hardware Architecture for the SoC Platform

Our attention can now shift to creating a block diagram for our hardware design for the SoC Platform, and, at SensorsThink, this diagram makes the most sense to complete in Enterprise Architect. By using the same diagramming tool and working in the same project, it becomes a very manageable task to trace hardware architecture

elements to system-level requirements and system-level elements to show how the hardware architecture satisfies the system-level requirements. If we refer to the discussion on the V-model in Section 1.5: Requirements, we are now working squarely in level 30, we are in a design domain, and we are putting the architecture in place for that domain's implementation.

In safety systems, such as those governed by ISO 26262, the hardware architecture is a key deliverable for being able to show how the hardware design satisfies the safety concept. The architecture helps to show ties between hardware and software and shows clear links to and from test cases as well.

To facilitate discussion regarding hardware architecture and its usefulness, we will refer again to ISO 26262. Most of the textual information in the standard refers to safety requirements and things such as the automotive safety integrity level (ASIL). However, there are some good concepts from which any project can benefit by utilizing and following [2].

In section 7.4 of part 5 of ISO 26262, some recommendations are given that are easily applicable to products of all kind by changing one word.

- "The hardware architecture shall implement the hardware safety requirements." If we change this to: "the hardware architecture shall implement the hardware system requirements," instead we see that this is just generally sound advice. The architecture implement shall implement the system requirements. It seems obvious, but some of the best advice often is.
- "The traceability between the hardware safety requirements and their implementation shall be maintained down to the lowest level of hardware components." If we change this to: "The traceability between the hardware system requirements and their implementation shall be maintained down to the lowest level of hardware components" instead, we see again that this is just generally sound advice. As one example of this, consider our prior discussion on verification in Section 1.6.11. Understanding what piece of hardware specifically satisfies a requirement makes the development of test cases and linking those test cases back to clear pass/fail criteria much easier down the road of development.
- "In order to avoid failures resulting from high complexity the hardware architectural design shall exhibit the following properties by use of the principles listed in table 1: modularity; adequate level of granularity; and simplicity." This is one of the most well-written statements regarding the importance of implementing the right solution and not one that is either overengineered or underengineered. Let's digest what is being suggested as advice here. Essentially, by having a modular architecture, that has enough granularity (which is to some extent subjective, but most engineers can usually agree when something is underdefined, even if most cannot agree when something is overdefined) and, most importantly, is simplified to the minimum required complexity to meet the requirements that will help to avoid failures. While future-proofing a system or project would be nice, it can often lead to unnecessary complexity. Complexity is a prime suspect in hardware systems failing. If a system requires only 1 Mbps of data throughput,

implementing a chip-to-chip interface that can handle 100 Mbps introduces additional design complexities that can cause implementation issues, as well as issues with EMI/EMC due to the faster clock frequencies and edge rates associated with faster interfaces. This is only one example, but it is an easy one to cherry-pick.

Hardware architecture ties into the discussion here in taking the time to document the key components, their interfaces, and the data that flows over those interfaces. It allows the entire engineering team to understand the real nuts-and-bolts impact to design decisions. If a new sensor comes along that would be a nice to have in the system but causes the SoC Platform to require that we make a technology jump to add high-speed transceivers to our SoC, we can much more clearly see that with a well-defined and documented architecture, fed by key component selection.

ISO 26262 also provides some guidance as to what makes a well-planned architecture; SensorsThink would be wise to consider Table 2.1.

As a final point, consider this last bit of advice from ISO 26262: "Nonfunctional causes for failure of a safety-related hardware component shall be considered during hardware architectural design, including the following influences, if applicable: temperature, vibrations, water, dust, EMI, cross-talk originating either from other hardware components of the hardware architecture or from its environment."

If we again simply remove the word safety and replace it with the word "system," we can see that this is wise advice to heed on all projects and product developments. Consider other failure modes aside from functional failures. Consider how to protect hardware against its operating environment, consider how to protect circuits from each other, and consider this early in the design process. It is much easier to populate a set of component pads to help with an EMI/EMC problem than it is to cut a board up, hack a part into place, and then respin your design to pass the testing. An ounce of prevention is worth a pound of cure.

#### 2.2.1.1 Hardware Architecture: The Diagram

Figure 2.10 shows the finalized hardware architecture for the SoC Platform. What we can see here is that the once-abstract blocks now have assigned part numbers due to our key component selection activity. From this, we can then start to define and understand the interfaces to each component and what will ultimately be required by our SoC.

**Table 2.1** ISO 26262, Part 5, Table 1

| *_Methods* | *System Level* | *A* | *B* | *C* | *D* |
|---|---|---|---|---|---|
| 1 | Hierarchical design | + | + | + | + |
| 2 | Precisely defined interfaces of hardware components | ++ | ++ | ++ | ++ |
| 3 | Avoidance of unnecessary complexity of interfaces | + | + | + | + |
| 4 | Avoidance of unnecessary complexity of hardware components | + | + | + | + |
| 5 | Maintainability (service) | + | + | ++ | ++ |
| 6 | Testability (during development and operation) | + | + | ++ | ++ |

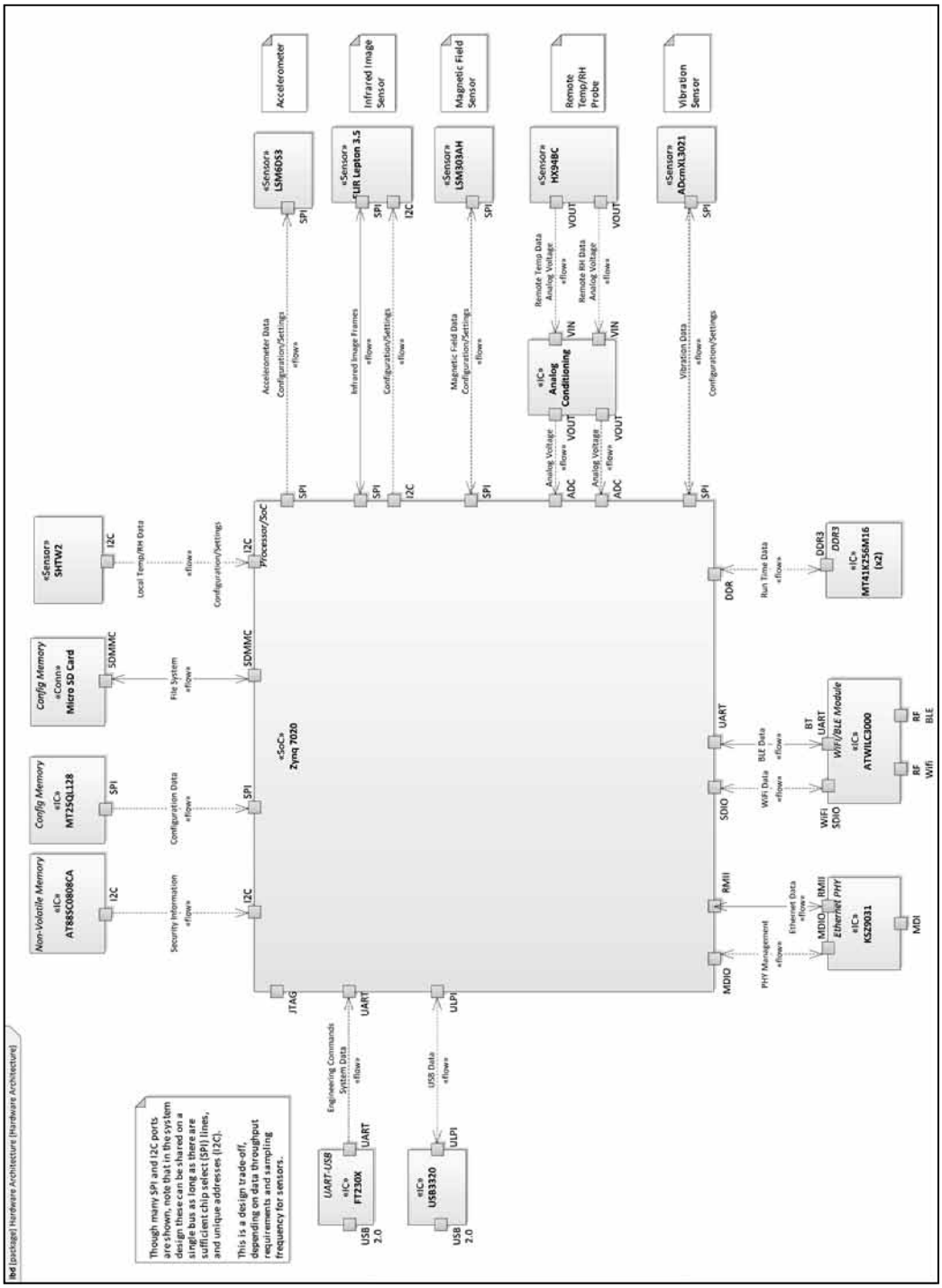



**Figure 2.10** Detailed hardware architecture.

### 2.2.2 Hardware Architecture: Interfaces

The way to understand the design in more detail is to really dig into the datasheets. The datasheets (at least the well-written ones) hold all the information that we need to better understand how we interface to and communicate with our components.

In addition to interfaces and communication, device datasheets hold a wealth of other information for integrated circuits: power ratings, operating voltages and currents, device functionality, characteristic curves and plots, and many other key bits of information. In order to really own a hardware design, you must understand the devices being used with an appropriate level of detail.

As an example, let's take a look at the FLIR Lepton 3.5 datasheet and better understand the interfaces it has and how they are used.

#### 2.2.2.1 Interface Example: FLIR Lepton 3.5

The diagram in Figure 2.11 [3] gives a high-level overview of the Lepton sensor. At this point in the design process, we are focused primarily on the interfaces to and from the sensor. We can see that leaving the sensor are several IO types: Video over Serial Peripheral Interface (VoSPI), I2C, GPIO, discrete controls, and a clock. The other arrow shown depicts power supplies, which are important, but come into play later in the design process. To better understand the interfaces and how they are used, we can start with the pin descriptions, this is generally a good place to go to for references to other parts of the datasheet that will better explain the connectivity of the pins.

Table 2.2 shows the entry from the Lepton datasheet for pins on the VoSPI. In addition to the reference to the appropriate page section, we also see that this is an SPI-compatible interface. This is key information that should be (and is) captured on the hardware architecture diagram.

If we jump into that section of the datasheet, there is a wealth of information in terms of how the interface is used, and how to properly communicate with it.

- VoSPI does not utilize the master-out slave-input (MOSI) line, and that pin should be set to a logic 0 or connected to signal GND.

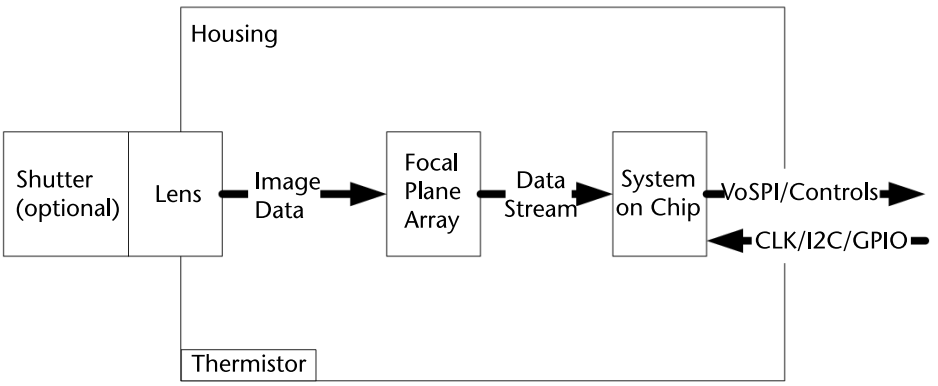



**Figure 2.11** FLIR Lepton block diagram.

**Table 2.2** VoSPI Pins: FLIR Lepton 3.5 Interfaces, VoSPI

| *Pin Number* | *Pin Name* | *Signal Type* | *Description* |
|---|---|---|---|
| 11 | SPI_MOSI | IN | Video Over SPI Slave Data In |
| 12 | SPI_MISO | OUT | Video Over SPI Slave Data Out |
| 13 | SPI_CLK | IN | Video Over SPI Slave Clock |
| 14 | SPI_CS_L | IN | Video Over SPI Slave Chip Select |

- Implementations are restricted to a single master and single slave.
- The SPI mode used by the device is SPI mode 3:
  - Clock polarity (CPOL) setting of 1 and a clock phase (CPHA) of 1.
  - For more information on SPI modes, refer to [4].
- Data is transferred by the most significant byte first and in big-endian order.
- The maximum clock rate is 20 MHz.

From these pieces of information, we can very clearly convey the requirements for the master SPI device (our SoC) to our firmware engineering team and also start to put the puzzle pieces together in terms of the number of peripherals required of our SoC to communicate to our sensors. Buried deeper in the section of the datasheet is the number of bits in each video frame and the different modes of communication in which the Lepton can operate.

As an example, let's consider the Raw14 video format mode. In this mode, there are 164 bytes per packet and 60 packets per video frame. That gives us approximately 10 kB per video frame. This is useful information to know for further consideration on where to place the peripheral and to be able to share as a starting point for firmware engineering. This is one of the two interfaces shown on the diagram. The other one was I2C, let's take a look at that interface next.

Table 2.3 shows the entry from the Lepton datasheet for pins on the Command and Control interface. In addition to the reference to the appropriate page section, we also see that this is an I2C-compatible interface. This is key information that should be (and is) captured on the hardware architecture diagram.

If we now jump into that section of the datasheet, we can find information on how the interface is used and how to properly communicate with it.

Lepton provides a command and control interface (CCI) via a two-wire interface similar to I2C. In this section, we are given a very useful reference to a separate interface description document (IDD) to fully explain the interface [5]. There is

**Table 2.3** I2C Pins: FLIR Lepton 3.5 Interfaces, I2C

| *Pin Number* | *Pin Name* | *Type* | *Description* |
|---|---|---|---|
| 21 | SCL | IN | Camera Control Interface Clock, I2C-compatible |
| 22 | SDA | IN/OUT | Camera Control Interface Data, I2C compatible |

also a list of some of the parameters that are controllable via this interface; things such as gain, telemetry, frame averaging, and radiometry mode selection are all done via this interface.

Again, all the pieces of information that we need are here to help us to make informed decisions in piecing the puzzle together to better understand our embedded system.

### 2.2.3 Hardware Architecture: Data Flows

The last really key piece of information to consider at this stage of architecture is data flows: what type of information is flowing over each interface and how much data is flowing. This is key information to consider on buses such as I2C, SPI, and CAN. These types of communications buses have multiple devices sharing a single bus, and, as a result, the throughput of the bus is shared. So for the Lepton, the SPI has a maximum frequency of 20 MHz, which is 20 Mbps. That means that the SPI over which the Lepton is communicating can only transfer that much data.

In some cases, it is possible to be more creative with sharing interfaces if you are really peripherally limited in your design. For the purpose of our discussion, we are heeding the advice of ISO 26262 and trying to keep the design as simple as possible to avoid unnecessary failures due to complexity.

Similarly, the I2C-compatible bus may be easily shared for the Lepton, as most of the settings appear to be setup type commands that are not frequently changed by the user or system. This may not be true in all use cases and should be carefully considered to ensure that there will not be any hidden traps or pitfalls later on in the development cycle.

### 2.2.4 Hardware Architecture: Finishing Up

When this exercise is completed, a very clear and concise picture of the hardware should be taking shape. A reasonably deep dive into the datasheets is a great place to start this exercise, and along the way you can make note of other bits of key information to use later in the process. Key components, interfaces, and data flows are now documented. This can be used as evidence to show that the hardware satisfies the systems' requirements. This type of design groundwork gives high confidence levels that the design intent will be fit-for-purpose for the product being developed. There is really no limit to the amount of information that you can embed in an Enterprise Architect model; how much you enter is a matter of how much is necessary for your development project, but the value in having the information in a single modeling environment increases exponentially as the system complexity increases.

## 2.3 Designing the System

With a solid underlying foundation in place, in the form of hardware architecture, the nuts and bolts of hardware design can begin in earnest. For hardware engineers who are new to PCB-level design work, this can seem daunting. Where to begin? Where to go next? It can be a very overwhelming task to undertake, even more so as the complexity of the system increases. As with most tasks in engineering, this

is where having a solid design process in place can make the insurmountable seem manageable.

For a system such as the one being designed by SensorsThink, the process to follow is the same as for a system being designed for a coffee maker or a server grade motherboard. The differences come in the complexity of those steps, and the amount of planning, checking, and replanning that likely go into each step.

Note that it is important to take a step back and acknowledge that, in engineering and in design work, there is always more than one way to get a job done properly. The intent of this story and the steps being laid out are to provide a framework that can be followed, which hopefully leads to successful and well-engineered designs and solutions. Over the course of your career, you will find what works best for you, and you will find what does not work for you at all. It is all part of the journey of learning the engineering trade and learning about a specific field of engineering. Design will evolve, software tools will change and improve, and the way that design is approached will have to change as well in order to keep up with the times. Now let's jump back into the story and outline some high-level steps to follow when designing an embedded system.

### 2.3.1 What to Worry About

If we look at the hardware architecture at which we arrived, we can start to look at the types of design analyses that are important to do in the early stages of design work. If you have never done a design like this with high-speed interfaces and an SoC, it may seem overwhelming. The key is to break the design down into manageable pieces and into subaspects of the design itself; for example, in most designs, a designer would be faced with at least some of the following design considerations and tasks.

### 2.3.2 Power Supply Analysis, Architecture, and Simulation

A key part of this on modern designs is power sequencing, especially on designs with multiple complex devices, each having their own requirements. Oftentimes, these requirements conflict with each other and require complex sequencing schemes to achieve compliance with all devices.

Simulation is a great way to verify your design intent before committing to a schematic capture or to a physical PCB. Once you have those designs on physical copper, mistakes are costly to rework and in some cases may not be possible. Simulation is your friend; use it.

A great way towards success is to analyze each power supply rail or, in other words, each discrete voltage level one component at a time. This will provide a great deal of insight into the current requirements for each voltage and is a necessary step in choosing appropriate power supply components.

Once you understand the current requirements, grouping the power supplies into stages of sequencing is another good idea; this maps out the order in which power supplies must be turned on (and sometimes turned off) in order to have a reliable and safe power system in which your components can operate.

At this point, you can put together a block diagram of the entire scheme using the tool of your choice. A tool that is particularly useful for this is LTpowerCAD.

This tool brings a number of very useful power system design tools together under one umbrella including individual power supply design and power system design and works hand-in-hand with LTspice to run more detailed simulations of power supplies. As an example, consider Figure 2.12 (tools used with the friendly permission of Analog Devices Inc.

This diagram is not for the SensorsThink design, but it is a good example of how you can visualize an entire power system, include all of the nominal loads, and really get a high level of confidence that your solution is a good one. After entering all of your design data, the tool also does power calculations on the system to show you things such as total power consumption, dissipation, and margin.

If we then wanted to look at the transient performance of a specific power supply, we can jump into LTspice and run that simulation as well, linking the simulation to this system-level view, making for a really nice design artifact.

### 2.3.3 Processor and FPGA Pinout Assignments

While it is true that FPGAs and modern processors (or in our case, one device that is both) are extremely versatile and flexible, it is equally true that there are seemingly infinite numbers of conflicting uses for multipurpose pins, invalid assignments of peripherals, and very specific requirements and uses for some pins that must be accounted for. The best way to prepare for this task is to start reviewing documentation. All of these "gotchas" are hiding in that documentation somewhere, and it is your job to find them and account for them.

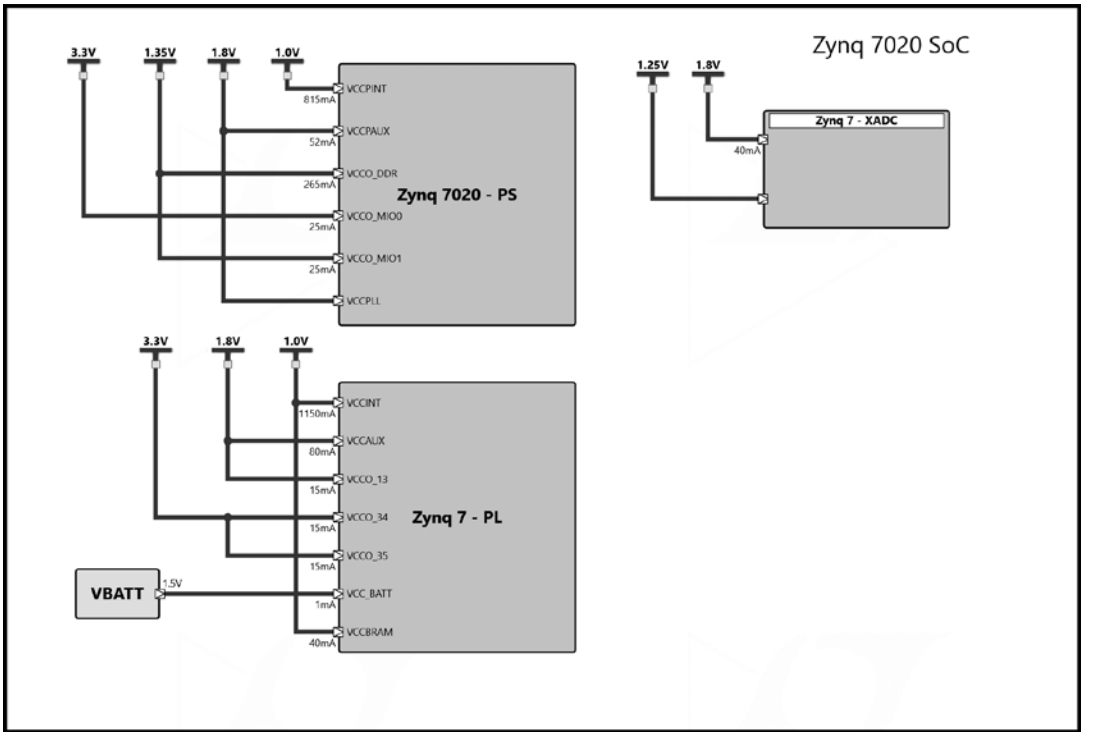



**Figure 2.12** LTpowerCAD diagram.

### 2.3.4 System Clocking Requirements

Processor clocks, FPGA logic clocks, local oscillators, and clocking domains: there is quite a bit of nuance and detail in understanding the clocking requirements of your system. If there were advanced protocols being used such as PCIe, those would require their own reference clocks as well.

Diagramming this scheme in a block diagram form can be a valuable exercise in understanding the right architectural solution for your design. It may be a programmable clock generator with 8 outputs, or it may be just a few local crystal oscillators. A great way to determine this is to break this part of the design down in a separate design activity.

### 2.3.5 System Reset Requirements

Every system has at least one kind of reset, a power-on reset. This reset ensures that a system starts from a known safe and valid state when power is applied. Other systems may have several other kinds of resets that need to be designed and accounted for:

- A cold reset essentially mirrors the logical behavior of a power-on reset in many cases, with the difference being that power is not removed from the board. This type of reset is useful during development and testing in the event that a system becomes unresponsive.
- A warm reset is typically reserved for resetting peripheral devices and can be used in development or in the system during operation to try to recover from run-time faults.
- External reset sources, such as a user push button, can also be reset sources that are intended to have a specific impact to the design; it may simply be a cold reset, or it may reset only select devices.

### 2.3.6 System Programming Scheme

How will your system get its code initially programmed? For external flash memories, they may be preprogrammed from the supplier once you are in production, but what about during development? Commonly, this is done via a JTAG interface for things such as FPGAs and for modest processors as well. For Linux-based solutions, often an external micro secure digital (SD) card is used as a quick way to develop new operating system environments and deploy them to the system. Another popular solution is embedded multimedia card (eMMC) memory, which is soldered to the board, but has the drawback of longer programming times in system.

In addition to the programming scheme itself, configuring the processor, FPGA, or SoC to accept the programming from the interface that you intend to use requires the configuration bits to be strapped (pulled to a logic 1 or logic 0) in a certain way to tell the device to look here for its code.

### 2.3.7 Summary

There is no shortage of design topics that can be concerning for engineers doing hardware design work. In high-speed digital systems, there can be months of work doing signal integrity simulations planning for routing structures (vias, fanout pattern, pin escape) on multigigabit-per-second interfaces. In sensitive analog systems, there can be many months of work doing analog simulation and analysis on op-amp circuit stability, looking at Bode plots, and doing noise analysis before arriving at what should be committed to the schematic. The more preparation work that is done, the more at ease the designer will ultimately feel when it comes time to stamp a schematic and PCB design with his or her approval. Let's take a look at an example of one of the topics we just mentioned in a bit more detail to get an idea of examples of what to expect when undertaking this type of work. Let's focus on the Zynq device, as it is the most complex device on the SensorsThink Platform towards which we have been marching.

### 2.3.8 Example: Zynq Power Sequence Requirements

In Xilinx terminology, there are two subsystems of the Zynq SoC, the PS is the processor system (the ARM Cortex processor complex), and the PL is the programmable logic portion of the device (the FPGA).

From the Zynq-7000 Technical Reference Manual [6], the following is a high-level view of the power domains: "The PS and PL power supplies are fully independent; however, the PS power supply must be present whenever the PL power supply is active. PL power up needs to maintain a certain timing relationship with the POR reset signal of the PS."

That is a total of 13 power domains in the device; that is not to say that you need 13 power supplies, as perhaps in your design (or ours), some of those may be combined together (Figure 2.13). A smart designer would wonder, "where do I find out how to combine these supplies, and in what order do I bring them up and down?" Let's take a look at a different document, the Zynq-7000 SoC DC and AC Switching Characteristics Specification [7], and try to get some more insight there.

| Type | Pin Name | Nominal Voltage | Power Pin Description |
|---|---|---|---|
| PS Power | VCCPINT | 1.0V | Internal logic |
| | VCCPAUX | 1.8V | I/O buffer pre-driver |
| | VCCO_DDR | 1.2V to 1.8V | DDR memory interface |
| | VCCO_MIO0 | 1.8V to 3.3V | MIO bank 0, pins 0:15 |
| | VCCO_MIO1 | 1.8V to 3.3V | MIO bank 1, pins 16:53 |
| | VCCPLL | 1.8V | Three PLL clocks, analog |
| PL Power | VCCINT | 1.0V | Internal core logic |
| | VCCAUX | 1.8V | I/O buffer pre-driver |
| | VCCO_# | 1.2V to 3.3V | I/O buffers drivers (per bank) |
| | VCC_BATT | 1.5V | PL decryption key memory backup |
| | VCCBRAM | 1.0V | PL block RAM |
| | VCCAUX_IO_G# | 1.8V to 2.0V | PL auxiliary I/O circuits |
| XADC | VCCADC, GNDADC | N/A | Analog power and ground |
| Ground | GND | Ground | Digital and analog grounds |

**Figure 2.13** Zynq-7000 SoC power domains.

#### 2.3.8.1 PS Power-On/Off Power Supply Sequencing

"The recommended power-on sequence is VCCPINT, then VCCPAUX and VCCPLL together, and then the PS VCCO supplies (VCCO_MIO0, VCCO_MIO1, and VCCO_DDR) to achieve minimum current draw and ensure that the I/O are 3-stated at power-on.

The recommended power-off sequence is the reverse of the power-on sequence. If VCCPAUX, VCCPLL, and the PS VCCO supplies (VCCO_MIO0, VCCO_MIO1, and VCCO_DDR) have the same recommended voltage levels, then they can be powered by the same supply and ramped simultaneously"[6].

This verbiage is taken verbatim from that document, but it does give a clear picture of the power supply sequence requirements and is an actionable statement that can help a hardware designer put a robust power scheme together for the PS system.

A note to the reader: There are more nuanced details provided and explained in the Xilinx documentation regarding proper design and sequencing for the Zynq-7000; for the sake of providing a relatively easy-to-understand example, they have been omitted.

#### 2.3.8.2 PL Power-On/Off Power Supply Sequencing

"The recommended power-on sequence for the PL is VCCINT, VCCBRAM, VCCAUX, and VCCO to achieve a minimum current draw and ensure that the I/O are 3-stated at power-on. The recommended power-off sequence is the reverse of the power-on sequence. If VCCINT and VCCBRAM have the same recommended voltage levels, then both can be powered by the same supply and ramped simultaneously. If VCCAUX and VCCO have the same recommended voltage levels, then both can be powered by the same supply and ramped simultaneously" [6].

This verbiage is again taken verbatim from that document, but it does give a clear picture of the power supply sequence requirements and is an actionable statement that can help a hardware designing put a robust power scheme together for the PL system. By comparison, it is of similar complexity as the PS requirements. Bear in mind that the PS and PL power supplies are fully independent. PS power supplies can be powered before or after any PL power supplies. The PS and PL power regions are isolated to prevent damage.

As the designer, you can now analyze your system, group power supplies together, and start to formulate a system that meets these requirements (among many others) to ensure that your system will work reliably.

A note to the reader: Another great source of sanity checking your design is reference designs provided by the manufacturer or its certified partners. They can offer a great deal of insight into how these requirements and constraints are managed by the company that designed the device. Leverage all of that knowledge and adapt it to your own design work.

## 2.4 Decoupling Your Components

During design reviews, it is often what seems to be the most mundane topics that get the most attention: the color of LEDs, the percentage tolerance of specific resistors, what sheet size was used for the design; on and on it can go, seemingly with no value added. One topic that can seem to fall into that category is decoupling capacitors, but doing so would be a considerable mistake, and, as such, the designers at SensorsThink are wise to consider decoupling strategy in great detail.

Despite the boring exterior, decoupling is absolutely essential to PCB design. Without it, you risk having a totally nonfunctional circuit. With too much or too little of it, you can have instability in all kinds of circuits: power supplies, loop compensation circuits, phase locked loops (PLLs), and more.

A note to the reader: In the course of my career, I have spent countless hours, days, and probably weeks discussing decoupling with seasoned engineers from analog, power, and digital perspectives. Decoupling is not a one-size-fits-all approach, and knowing that is a big part of conquering the battle with those little capacitors.

### 2.4.1 Decoupling: By the Book

Let's start by going back to the basics, literally, of the word decoupling: as per *Merriam-Webster* [8], to decouple something is "to eliminate the interrelationship of: SEPARATE."

So why is it that we would want to decouple our components? What is the benefit of separating them and from what are we separating them?

### 2.4.2 To Understand the Component, You Must Be the Component

Imagine for a moment that you are an integrated circuit; say, for the sake of argument, that you are a very simple one such as an inverting buffer. You are given a source of power (often called VCC or VDD) and a place to put your used power (often called GND). You have a bit presented to you, and your sole job in life is to take the state of that bit and invert it; make a 1 a 0 and make a 0 a 1.

When the time comes to do your job, you look to your power pin for energy to do your work, and sitting about 100 miles away is the power supply circuit that the hardware engineer designed for you to use. It takes a lot of effort to get the energy that you need to do your simple job, and as you gather up that energy, you are traveling a great distance, probably bumping into traffic along the way from your colleague ICs. This distance in the electronics world is impedance. Depending on how fast you have to do your work, you will choose one of two paths: the path of least resistance (R) or the path of least inductance (L). Either way, it would make your life a lot easier if you had something close by from which to grab that energy.

A note to the reader: This is a very oversimplified breakdown of the mathematics behind impedance and frequency and why integrated circuits need energy to do their jobs. Bear with me; it is merely to serve as an example to illustrate a point, not to be a mathematically accurate model of current flow and loop areas.

So let's reenact that scenario, only this time you look to your power pin, and sitting right next to it is a little bucket full of energy into which you can reach your shovel and grab the energy you need for inverting that bit: fantastic, a shorter

journey, less work, and you disturbed no one else on your journey to get that energy. Before you know it, that little bucket gets filled back up, and the next time that you need energy, it is sitting right there for you again.

By providing a local energy storage device (decoupling capacitor) for your circuits, you make their lives easier, they work better and more efficiently, and they are friendly to their neighbors. This is a win-win scenario.

### 2.4.3 Types of Decoupling

There are many different types of decoupling and many different types of capacitors, including:

1. Aluminum electrolytic;
2. Aluminum polymer;
3. Tantalum;
4. Ceramic;
5. Film.

Each type of capacitor has its own strengths and weaknesses, some due to the type of dielectric material, some due to packaging sizes, and some due to material properties and applications. A full discussion on each capacitor type and the advantages and disadvantages warrants a book all to itself. As a starting point, you can check the references at the end of the chapter for some very insightful discussion and information [9–12].

#### 2.4.3.1 Bulk Decoupling

Bulk decoupling most commonly refers to large (in value and, subsequently, in size) capacitors placed around a circuit or a group of circuits to act as a larger bucket of energy to more quickly refill the smaller buckets of energy that are placed closer to the devices. From a high level, think of these bulk capacitor components as a way to relocate the main source of power for an area on your design, further away from the power supply generating the energy, and closer to the loads that need them. The math behind this suggests that these types of capacitors are most effective in the range of dc to hundreds of kilohertz.

#### 2.4.3.2 High-Frequency Decoupling

High-frequency decoupling most commonly refers to small (in value and, subsequently, in size) capacitors placed directly near a specific pin or a group of pins on a specific integrated circuit to act as a small bucket of energy for higher-frequency demands that the circuit may have. From a high level, think of these capacitor components as a way to shorten the path of least inductance a bit for your high-frequency friends in between the big bulk decoupling components and the silicon inside the integrated circuit. The math behind this suggests that these types of capacitors are most effective up to frequencies around 500 MHz in the absolute best-case scenario.

#### 2.4.3.3 Interplane PCB Decoupling

This type of capacitance is a result of smart PCB stack-up design and refers to the power and ground planes of the PCB acting as the two plates of a capacitor. In order for this to be effective, the power and ground planes must be very close together, on the order of 2 mils (0.002 inch). Doing so can yield an interplane capacitance value of around 500 pF/in$^2$. In order to leverage this, the IC power pins (VCC and GND both) must be directly tied to the power and GND planes with vias. This type of decoupling can be effective up into the range of several gigahertz.

#### 2.4.3.4 On-Die Decoupling

This type of decoupling refers to capacitors placed on the substrate package right next to the silicon wafer. This is as close as decoupling capacitors can be placed, and for high-speed digital devices, they are almost always put there by the chip supplier, knowing that by the time the need for energy has reached the PCB to which the device is mounted, the impedance is too large for the capacitor to be effective.

As a designer, you need to be aware of and leverage all of these technologies and decoupling strategies, as they all play a role in a properly designed system with good power integrity. Let's take a look at the Zynq-7000 series documentation again for a good example of how these strategies play out in a real-world design example.

### 2.4.4 Example: Zynq-7000 Decoupling

From the Zynq-7000 Technical Reference Manual [6], we can find a mention of on-die decoupling in the section that discusses features of the PL system: "High-frequency decoupling capacitors within the package for enhanced signal integrity." Right away as a designer, you can check that box off mentally, knowing that Xilinx has done their part to ensure that your power integrity and decoupling scheme are set up for success.

In the Zynq-7000 PCB Design Guide [13], there is a discussion on several of these topics, starting with power planes, carrying through concepts on return currents and mounting inductance. All of these topics are fantastic evidence that Xilinx has a great handle on these concepts and should give you confidence as a designer that their suggestions come from a place of both knowledge and empirical data.

The documentation goes so far as to suggest capacitors based on your specific device. Recall that for SensorsThink, we eventually settled on the XC7Z020-2CLG400I as our device of choice. Based on that, we can extract the following recommendations from Xilinx, first for the PL subsystem (see Table 2.4).

Next, we can get the same information for the PS subsystem in the same document, for our package type, the CLG400; and our device type, the Z-7020 (see Table 2.5).

This is fantastic information and really takes the guesswork out of the decoupling of our device. However, Xilinx takes things a step further and even specifies acceptable capacitor sizes, types, and even part numbers to use. For full details, the latest PCB Design Guide (UG933) is a fantastic resource.

**Table 2.4** Number of Capacitors Per Power Domain: PL Subsystem

| *Capacitor Value* | *VCCINT* | *VCCBRAM* | *VCCAUX* | *VCCO per Bank* | *Bank 0* |
|---|---|---|---|---|---|
| 680 $\mu$F | 0 | 0 | 0 | 0 | 0 |
| 330 $\mu$F | 1 | 0 | 0 | 0 | 0 |
| 100 $\mu$F | 0 | 1 | 0 | 1* | 0 |
| 47 $\mu$F | 0 | 0 | 1 | 1* | 1 |
| 4.7 $\mu$F | 2 | 1 | 1 | 2 | 0 |
| 0.47 $\mu$F | 4 | 1 | 1 | 4 | 0 |

Note: The asterisk indicates that either a 100-$\mu$F or 47-$\mu$F capacitor is recommended.

**Table 2.5** Number of Capacitors Per Power Domain: PS Subsystem

| *Capacitor Value* | *VCCPINT* | *VCCPAUX* | *VCCO_DDR* | *VCCO_MIO0* | *VCCO_MIO1* |
|---|---|---|---|---|---|
| 100 $\mu$F | 1 | 1 | 1 | 1 | 1 |
| 4.7 $\mu$F | 1 | 1 | 1 | 1 | 1 |
| 0.47 $\mu$F | 3 | 1 | 4 | 1 | 1 |

As the designer, you have all the information that you need to properly decouple the Zynq at a schematic level, make smart component selection choices and substitutions for parts that may be hard to find, and understand the role each type of component plays.

### 2.4.5 Additional Thoughts: Specialized Decoupling

As mentioned earlier, decoupling is not a one-size-fits-all problem or solution. Some suppliers are great at giving guidance, and others are not so great. This is where understanding the fundamentals is so important and realizing that in some cases a 0.1-$\mu$F capacitor works as well as a 1.0-$\mu$F capacitor and vice versa. In other cases, they may not work interchangeably.

In some applications or in some devices (for example, RF devices), specific frequencies are targeted for aggressive decoupling, and others are seemingly ignored. Always fall back to fundamentals and make sure that you understand the goal of the decoupling scheme. Do not be afraid to ask the manufacturer questions. If you do not find the answer satisfactory, explore other options or leave yourself extra, just-in-case component pads based on your thoughts and your experience. Either way, this topic is one that can be the topic of a thesis. There is always more to learn; but, in many cases with enough time invested in reading and pouring over reference material, you can design robust power decoupling networks without too many headaches.

### 2.4.6 Additional Thoughts: Simulation

There are many tools available for simulation of power integrity and power analysis. If you have them, use them. Leverage those tools to make your life as a designer easier and to give yourself more confidence in the final product. You can simulate

almost anything these days with powerful simulation tools such as HyperLynx [14, 15] or Ansys SIwave. There are tools that can analyze your capacitor placement such as HyperLynx DRC [16] and alert you to physical placement issues that are easy to miss on large or complex designs. Using those tools takes time, but the time invested is almost always worth the return on that investment.

## 2.5 Connect with Your System

In Section 2.4, we talked about a seemingly mundane topic that has many more nuances and layers than meet the eye. A topic that is much more straightforward and requires less math and more common sense is connector choice or connectorization. To be clear, connectorization is a made-up word; you cannot find it in a dictionary or in any kind of conversation outside of engineering. That will not stop the word from being used in conference rooms at SensorsThink, however; and it will not stop you (nor should it) from using the word either.

A note to the reader: Sometimes picking connectors is really easy, and sometimes it is not. There are often organizational influences on connector choices: what tooling in which the company may have already invested, what companies are preferred partners, or what families are already in heavy use across different products. These can all be important in the bigger picture decision-making process, especially if you are working on a product with massive volumes (hundreds of thousands or millions of units). For the sake of discussion, we are going to keep the focus on technical reasons to choose a connector and assign signals to locations, because it is impossible to predict all of the organizational influences that may or may not be present in your situation.

### 2.5.1 Contemplating Connectors

What are some things that ought to be considered during the choice of a connector for an application? What are the engineers at SensorsThink considering when they set out to agree on connectors for the SoC Platform? There are many of them, more than may be obvious at first glance, for example:

1. Existing standards for the interface;
2. Maximum expected voltage and current;
3. How many times the connector will be used (mating cycles);
4. Environmental considerations (temperature, humidity, water ingress protection);
5. The end user of the connector (customer, test engineering, manufacturing, design engineers);
6. Available physical space within the enclosure or on the PCB;
7. Cable egress path and angle of escape (right angle, vertical).

With so many things to consider, it is a fair question to wonder why this topic was just stated to be straightforward. The easiest way to demonstrate that is to walk through an example of choosing a connector for an interface of our Smart Sensor Controller.

### 2.5.2 Example: System Communications

For the first decision that we will explore along with the engineers at SensorsThink, the system communications port seems like a great place to start (Figure 2.14). It is the interface with which the end user will interface, and we can run through our list of considerations with a lot of existing data about the application.

#### 2.5.2.1 Existing Standards

If we refer back to the requirements derived for the project, the interface is required to interface to a network via a wired network, a wireless network, or both. In our hardware architecture process, the decision was made to implement both. Our wired network interface implements a Gigabit Ethernet connection over copper. This is driven by the component selection that was made for the Ethernet PHY device (the KSZ9031).

Almost every wired network interface for the last 20 years has had the same style of connector, an RJ45 style connector.

A note to the reader: This statement is true for nearly all residential, light commercial, or light industrial computer or connected device. The RJ45 connector name is a bit misused, however. The technically proper name for the connector is really an 8p8c connector as an RJ45 connector is keyed. This is such a common misuse of the RJ45 standard that we will carry on with tradition and refer to it as such as well.

For this interface, we now know we need to have an RJ45 connector. It is a widely adopted standard, and it is an easy choice to make.

#### 2.5.2.2 Maximum Expected Voltage and Current

For this specific example, the expected voltage and current are less important to know because of how heavily standardized the interface is. However, we can still find this information based on the physical layer standards that our Ethernet PHY device implements: 10BASE-T/100BASE-TX/1000BASE-T. The Gigabit Ethernet interface is typically transformer coupled; in other words, it is typically isolated from end to end using magnetics (transformers) (Figure 2.15). The isolation of the interface is typically 1500VRMS; this is specified by the IEEE 802.3 standard [17]. The signal levels themselves are quite modest by comparison: approximately 1.0V differential signaling for 100BASE-TX and 1000BASE-T, and approximately 2.5-V differential signaling for 10BASE-T.

#### 2.5.2.3 Mating Cycles

This is an example of where the common-sense aspect of choosing a connector comes into play. If we consider the life cycle of the product (100,000 hours) and make some reasonable assumptions, we can arrive at a good guesstimate number.

First, let's convert hours to years: 100,000 hours is a bit over 11 years. For the sake of easy math (and to build in a bit of margin), we will round up to 12 years.

Over 12 years, let's say that the average customer does two service calls per year. At that service call, we know from our context diagram (Figure 2.14) that

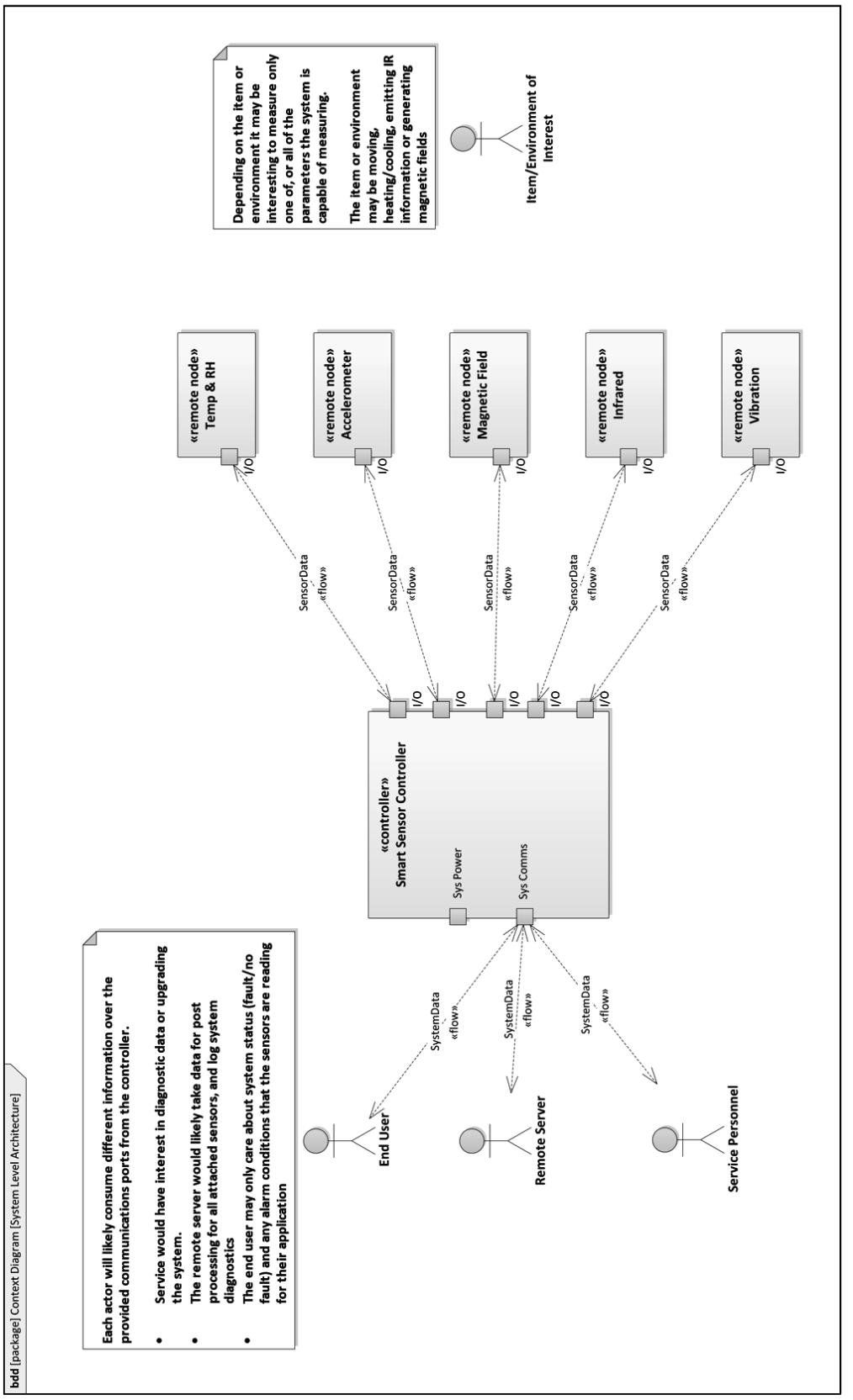



**Figure 2.14** System context diagram: SoC Platform.

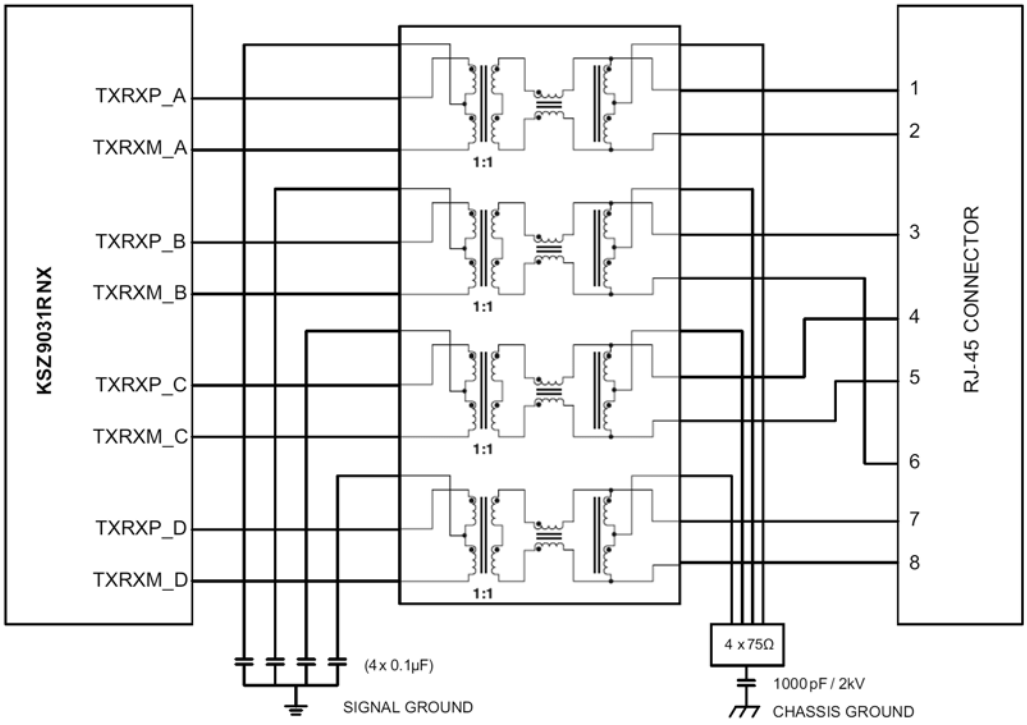



**Figure 2.15** KSZ9031 datasheet: magnetic interface.

the service personnel will use the interface. That means that at every service call, the RJ45 connector could be unplugged from the customer network, plugged into a service computer, and then reverted back to the customer network. That equals four mating cycles per visit, two visits per year: eight cycles. Eight cycles for 12 years equal 96 cycles. Let's keep with the theme of round numbers and round up to 100 cycles for maintenance.

Let's also assume that at least once per year the customer damages the wired connection and has to replace the cable to the interface; that is another two cycles per year for 12 years, or 24 cycles. We will go with 25 for our counting purposes.

For all intents and purposes, this is the bulk of the use that the interface will see. There are other times the interface will see action, during production testing, for example. There are the outlier customers that will change cables once a month or unplug the connector once a week for their application, maybe to clean around the system, for example.

To give ourselves some additional margin, and because the total number (125) is so modest, we will triple our calculation and assume that 375 mating cycles is the target.

#### 2.5.2.4 Environmental Considerations

In terms of environmental considerations, we can lean heavily on the excellent requirements work that the team at SensorsThink did and refer back to that list for guidance here.

1. Temperature range: –10°C to 50°C;
2. Humidity: 5% to 95% noncondensing.

There are no requirements for water or dust ingress protection, so this is all the information we need to make our choice.

#### 2.5.2.5 The User

The context diagram here provides us with the information we need very clearly in terms of the end user of the Ethernet interface:

1. The customer (end user);
2. Service personnel;
3. A remote server (another computer/networked device).

#### 2.5.2.6 Making the Choice

With all of the information gathered, making the choice becomes a matter of website filtering (choose your favorite supplier or distributor) and minding the details of the datasheet (Table 2.6).

In the case of the Ethernet, there are two types of RJ45 connectors from which to choose: those with integrated magnetics, and those without. Integrated magnetics make sense in many cases to simplify the PCB layout and to make design integration a bit easier. Let's presume that the design team and SensorsThink prefer that solution.

Bel Magnetic Solutions has a part number that checks off all of our considerations and requirement boxes: 0826-1X1T-32-F [18].

1. Existing standard: RJ45 Style;
2. Maximum expected voltage and current: 1,500-VRMS isolation;
3. Mating cycles: 375 (required confirmation with the supplier);
4. Environmental considerations: –10°C to 50°C;
5. End user: Because the connector type is so common, everyone is likely comfortable plugging in an RJ45 cable;
6. Datasheet details: Turns ratio, inductance, insertion loss, Hi-Pot.

This all points towards this being a good choice for our application and this interface. It did take a bit of information gathering and a bit of common sense and guesstimation (another made up word) to round things out but it was pretty

**Table 2.6** Magnetics Selection Criteria: KSZ9031

| *Parameter* | *Value* | *Test Conditions* |
|---|---|---|
| Turns ratio | 1 CT: 1 CT | — |
| Open-circuit inductance (min) | 350 $\mu$H | 100 mV, 100 kHz, 8 mA |
| Insertion loss (max) | 1.0 dB | 0 MHz to 100 MHz |
| High potential (HIPOT) testing (min) | 1,500 VRMS | — |

straightforward compared to many other design tasks that are encountered during hardware development.

If the team at SensorsThink follows a similar approach for all of the interfaces and connectors, they will end up making sound technical choices for the system.

## 2.6 Extend the Life of the System: De-Rate

In engineering, we nearly always consider things such as functionality and cost as driving considerations for making decisions. Other common factors can be things such as power consumption, physical size and weight, and reliability. Later we will cover reliability in great detail. One piece of reliability that can sometimes be overlooked is the de-rating of components. De-rating a component is the concept of operating a component at less than its rated capability with the intent of prolonging its operating life.

### 2.6.1 Why De-Rate?

Component de-rating is a deliberate step taken by engineering teams to try to ensure that a design is robust. Operating components at one or more of their maximum ratings can shorten their useful life, but it can also have impact on other ratings of the device. Sometimes these de-ratings need to be considered part of the base specification of a component. As an example, we can refer to a ceramic capacitor datasheet from Murata [19].

In the case of the working voltage of this ceramic capacitor value, we can see that for nominally rated 630-V capacitors, at the highest end of the operating temperature range, the value must be de-rated to 450V, which is approximately a 30% reduction in the working voltage for the device (Figure 2.16). This is extremely significant and is not something that readily presents itself on a single sheet specification. Knowing that this de-rating applies comes with experience and also with a careful examination of the detailed product specifications.

### 2.6.2 What Can Be De-Rated?

For the engineering team at SensorsThink, what types of component ratings can be considered for de-rating? The short answer is: anything with a rating. If it has a rating, you can de-rate it. It really is that simple. Some extremely common things that are rated in electronics components are:

1. Power;
2. Voltage;
3. Current;
4. Temperature;
5. Actuation cycles.

As we have done for other topics, such as verification, we can look to harmonized standards for a good starting point for discussion and investigation into this topic.

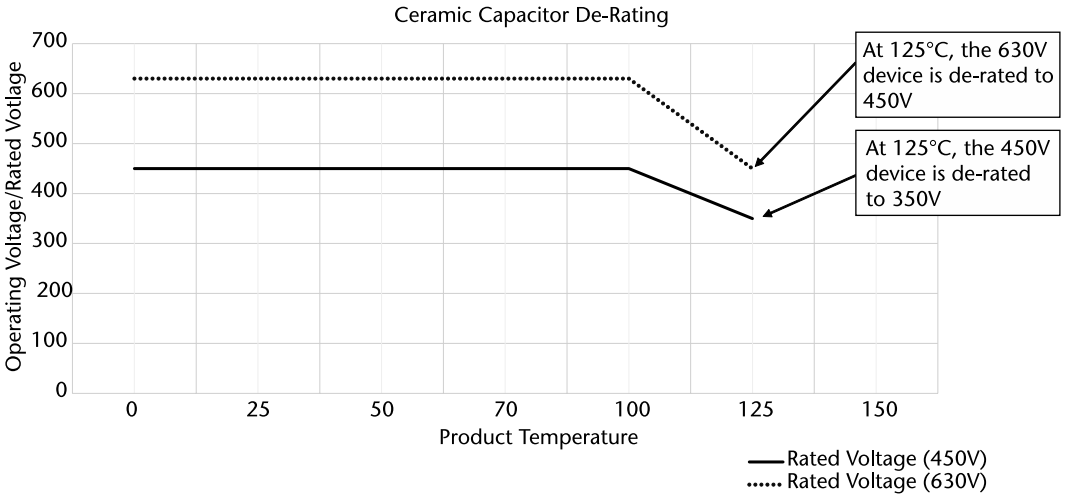



**Figure 2.16** Ceramic capacitor de-rating for temperature.

IEC 61508 [20] is an international standard for functional safety systems, and it is often cited as a baseline standard for other types of safety systems that develop their own standards for safety.

From the document's first section, we can refer to the introductory paragraph to better understand the intent of the IEC 61508 series:

> This International Standard sets out a generic approach for all safety lifecycle activities for systems comprised of electrical and/or electronic and/or programmable electronic (E/E/PE) elements that are used to perform safety functions. This unified approach has been adopted in order that a rational and consistent technical policy be developed for all electrically-based safety-related systems.

For the engineering team at SensorsThink, adhering to this standard is not required for the SoC Platform, as it is not a safety system or one used in a safety application. However, as we used the guidance from ISO26262 to better understand the value of, and methods for, proper verification, we can take similar lessons for de-rating from IEC 61508 and create a very robust hardware design, rooted in the principles of the most robustly designed systems in the world.

#### 2.6.2.1 What Does IEC 61508 Have to Say?

The guidance in IEC 61508 is extremely concise and simple; in the second entry of the series, the following text is given:

> De-rating (see IEC 61508-7) should be considered for all hardware components. Justification for operating any hardware elements at their limits shall be documented (see IEC 61508-1, Clause 5).
>
> NOTE: Where de-rating is appropriate, a de-rating factor of approximately two-thirds is typical.

Subsequently, referring to IEC 61508-7, we read the following:

> Aim: To increase the reliability of hardware components.
> Description: Hardware components are operated at levels which are guaranteed by the design of the system to be well below the maximum specification ratings. De-rating is the practice of ensuring that under all normal operating circumstances, components are operated well below their maximum stress levels.

There are two important takeaways from the standard:

1. Using components at their limits must be documented and justified.
2. A de-rating factor of approximately two-thirds (67%) is typical.

In other words, in a safety-related system that falls under the scope of IEC 61508, a 100-V rated capacitor can only be used on a voltage supply of 67V or less, unless there is supporting justification and documentation that shows why a higher applied voltage is acceptable.

### 2.6.3 Example: De-Rating the Zynq-7000

Based on the very straightforward guidance, we can apply these fundamentals to the Zynq-7000. Recall from the early discussion on system requirements that the SoC Platform must operate in an environment from –10°C to 50°C. While the platform is not a safety product, or part of a safety system, we can still apply the fundamental guideline from IEC 61508 to the temperature rating of the Zynq device that we chose to use as the heart of the system to determine the level of robustness that we can expect to see from that component, at least in terms of the impact from the temperature, during the product's life cycle.

Recall from Section 2.1 that the devices from which we chose belonged to the industrial grade of the Zynq-7000 series, with a nominal range of –40°C to 100°C. If we apply the de-rating factor of 67% to both the lower and upper limits of the range, we arrive at a new temperature range of –27°C to 67°C; and the good news for the engineers at SensorsThink is that the chosen device should not experience any reduction in working lifetime from the temperature range in which the product is expected to be used.

Suppose, however, that the device that was selected had a nominal temperature range of –20°C to 70°C instead of the –40°C to 100°C range with which we had to work. Applying the same de-rating factor would provide a result of –13.3°C to 46.6°C.

If the engineering team at SensorsThink wanted to design a very reliable and robust system, they may consider seeking a component with a wider temperature range, or they may want to consider doing extensive elevated temperature testing to understand the potential impact to the working life cycle of the product.

### 2.6.4 Additional Thoughts: Exceptions to the Rule

As with most rules, there are always exceptions, and de-rating is no different. There are certain types of components that are treated differently and must be de-rated differently if the engineering team truly needs a robust system to be deployed. Wire-wound resistors are often de-rated to 33% of their advertised power dissipation

factor in safety systems, and solid tantalum capacitors are usually de-rated to 40% of their listed working voltage due to their long-standing reputation as devices that self-ignite during failure.

Knowing the guidance from IEC 61508 is a great starting point, but it is not a substitute for engineering rigor and for deeper understanding of the exceptions to the rules, especially in safety-critical systems. The engineering team that ignores those rules and exceptions will not get very far in a safety assessment by functional safety experts.

Even if your system is not part of a functional safety product, that does not necessarily preclude the use of de-rating to ensure a robust system and product. It can sometimes drive cost and can lead to discussions about overengineering a particular system. There is little harm to overengineering on the side of reliability and robustness. Most importantly, engineers must be aware of all the trade-offs that they are making and make the right one for the product that they are designing.

## 2.7 Test, Test, Test

In an earlier part of our journey through developing this embedded system, we talked about the high-level process of verification and validation. We broke testing down into levels 10 through 40 to align with the V-model, and we used ISO-26262 as a framework and guide to better understand how to derive test cases and what kind of tests provided high confidence levels in critical systems.

After designing the PCB and getting it started through the process of fabrication, there is often a few weeks of downtime for the design engineering team to start to put a verification test plan together.

Recall that level 40 testing for hardware consists primarily of the verification of individual circuits (or small groups of circuits that comprise one function). In order to define a test plan, we must define all of the following:

- What methods will be used;
- What the pass/fail criteria are;
- What environment(s) will be used;
- What tools will be used.

This can be overwhelming on even moderately complex designs, but with a basic framework in place, this can be broken down into manageable steps and the process can be repeated from design to design. The detailed steps will certainly need to change to accommodate a specific PCB, or circuit, but the framework can be reused and can quickly become second nature as more designs are approached in a similar way.

### 2.7.1 Back to Basics: Pre-Power-On Checklist

The engineering team at SensorsThink has gathered in a conference room to talk about how they plan to verify the hardware that is expected to arrive at the office in a few short weeks. People start throwing detailed ideas out for looking at nuanced

performance of particular sensors, or elaborate stress tests for the high-speed portions of the design, a common occurrence when too many engineers gather in the same room.

Before discussing which pseudo-random bit pattern should be sent over a communications link or how many nanovolts of measurement accuracy are needed for a measurement, we should all remember the basics.

When a new PCB arrives, it must be treated as a complete unknown. Despite the thousands of hours of effort that went into conceptualizing and designing the circuits in the past several months, never forget that things happen. Human error is always a factor and, despite our best efforts, mistakes will be made. A well-constructed verification plan will push these errors to the forefront and allow them to be found, corrected, and protected from escaping into the field.

It is best to start with a very conservative approach and slowly expand the scope of testing as confidence in the design builds. Consider the following series of steps for taking a brand-new PCB out of the box and getting ready to test it:

1. Review provided documentation from the PCB fabrication and assembly contract manufacturers.
2. Perform visual registration checks.
3. Verify proper component orientation.
4. Verify that any components that were not to be included (DNP, DNI, no-load, no-stuff) are not present on the design.
5. Visually inspect solder joint quality; pay close attention to high-density components. Check for solder bridging.
6. Verify that any press-fit components are correctly installed.
7. Check that connectors and through-hole pins are not damaged or bent.
8. Before applying power, perform a mechanical fit check. Make sure that there are no interferences that were missed during three-dimensional (3-D) modeling work.
9. Measure impedances for each power domain to other power domains and to return nets.
10. Verify that fuses are not blown open, if they are present on the design.

This list of 10 steps can save a lot of headaches for engineers of all disciplines and only takes a short amount of time to perform on an appropriate sample size of the total lot of PCBs that have been received.

For particularly costly or complex designs, 100% inspection may well be warranted. In some cases, if very high-cost components are used, it is good practice to have several boards assembled without those components so that the underlying hardware environment can be verified to be safe for those high-value components. For example, consider a very high-end FPGA such as the Xilinx Virtex UltraScale+ Series of devices with on-board high bandwidth memory (HBM). These devices can cost tens of thousands (sometimes over hundreds of thousands) of dollars each. Taking extra precautions is absolutely warranted and is the right thing to do in cases like that.

It is reasonable to wonder what the value of each of the 10 steps listed provides to the engineering team, especially given that it can be quite exciting to get a new

piece of hardware in hand. Oftentimes, the instinct is to just plug it in, turn it on, and see what happens. Patience is a virtue, as the old saying goes, so let's explore that further.

#### 2.7.1.1 Hurry Up and Wait

Reviewing documentation from the PCB fabrication and assembly manufacturers is certainly not the most exciting activity an engineer will do, but it is important; if your design had controlled impedances, there should be reports with the boards showing that those impedances were verified and showing the measurements. Without seeing these results, there is no data to support that the high-speed traces on the PCB are actually fabricated with the correct characteristic impedance. This can cause many issues during testing (unstable interfaces, poor signal integrity) and sometimes can cause a design to not function. As speeds increase and rise times get faster, this becomes more important. Take the time to check the reports, and, if things are not clear, ask questions. It is better to ask early than to spend weeks chasing signal overshoot issues only to find that the board was not fabricated as you intended.

Visual checks are also a key aspect of bringing a design along from scratch. Imagine that an expensive component was loaded into a pick-and-place machine incorrectly, and pin 1 is not located properly. That could be disastrous for all involved. If caught before power is applied, the part can be removed and reworked to be installed correctly. If it is not caught, it might be too late to save the PCB. The same can be said for components that should not have been installed. Consider the case of things such as pull-up or pull-down resistors to configure a device in a certain way. If these are placed across a power supply rail and its return net, having both components present can be akin to applying a direct short.

Solder joint quality is one of the most important things to check; improperly flowed solder, too much solder, or not enough solder can cause the most frustrating of problems: things such as intermittent operation and fault behavior or board-to-board inconsistency. Many engineers have spent weeks chasing ghost problems caused by these kinds of issues.

Checking impedances between power domains and from power domains to return nets is essential as well. Remember always to check power and ground. The easiest way to derive the list of points to check is to open up the schematic, list every power supply net and return net, and make a spreadsheet or list of all supply nets to their return net, and if you have several return nets, check the impedance between them as well. If these values are not what you expect or are not consistent (within reasonable tolerance) across the lot of PCBs, then stop. Remove the outliers and figure out what is wrong with them. If this precaution is not taken, the magic-smoke can escape the design, and no one wants that.

Lastly, make sure that fuses survived being installed if they are present; there are fewer things more frustrating than trying to power a board on getting no activity, only to find a failed fuse as the culprit. It takes a few seconds to check a fuse, so save yourself some headaches and check them out.

### 2.7.2 Check for Signs of Life: Crawl Before Walking

After ensuring that the board has been properly fabricated and assembled and there are no drastic issues with short circuits, the time has come to apply power to the design. However, this does not mean that the engineering team should just slam the power on full-blast and start poking around. Having a plan is important.

The following set of steps is a good framework to keep in mind when bringing a new design online:

1. Basic power supply check:
   (a) For designs with programmable power supplies, this includes programming the supplies and verifying that the programming interface is functional.
   (b) A word of advice: always use a current limited power supply for these steps; better to hit a current limit and be mildly annoyed than to blow a board up and have to start over.
2. Clock verification:
   (a) As with power supplies, programmable clock generators are programmed at this point as well.
3. Power-on-reset/reset testing:
   (a) Many modern devices have requirements to be held in a reset state for a minimum amount of time. If the specific design under test has that as a requirement, verify that those times are being met. Doing so can prevent intermittent functionality/fault conditions from consuming hours (or days) of effort trying to debug.
4. Power sequence testing:
   (a) For designs with power-up and power-down sequencing constraints, they should be checked as early as possible. Not following those constraints can often cause device damage, so it is best to find those problems early.
5. Power supply noise and stability testing:
   (a) Before putting the devices through test cases that can cause load steps on power supplies, the stability of the supply's control loop should be tested over several conditions to ensure that catastrophic failures are not likely to occur.
   (b) Before checking the performance of interfaces, the noise performance should be looked at to ensure that in critical locations noise is within limits. Too much noise on high-speed device supply pins can cause the performance to degrade; understanding this and checking for that condition before testing in more detail can be invaluable.
   (c) Note that this step can be performed in the more detailed test stage in many cases; however, in designs with extremely expensive devices or with very high current supply rails, it is best to perform the testing early to try to catch any potential catastrophic issues early.
6. FPGA and processor programming:

(a) The last step before diving into the nitty-gritty verification world is to make sure that the processors and/or FPGA devices on the PCB can be programmed; this is usually accomplished via a JTAG (or another debugger) interface.

(b) In some cases, the JTAG interface can be used to program attached FLASH memory storage; if that is the case in a particular design, the time to check for basic functionality of that interface is early, before too much detailed checking has started.

By doing these tests for at least one representative PCB prior to distributing them to the other engineers who are part of the team, there is a reasonable level of confidence that a nominally functional hardware platform is in the hands of other engineers for further testing and development. Any necessary issues are found early and can be performed to all boards at the same time, ensuring a consistent deployment of hardware. This is crucial in teams of any size, but the larger the team, the more important.

### 2.7.3 Roll Up Your Sleeves and Get Ready to Run

With a reasonably high confidence level that the PCB that has arrived is not going to catch on fire and that it is able to be powered on and nominally programmed, the nitty-gritty testing can begin.

What does this mean exactly? What kinds of things can we leverage for test plans? Recall the earlier discussion on verification, in particular, verification specifications.

The verification specification essentially selects and specifies the methods to be used. This can be any number of things from simulation scenarios, physical test cases, and even something as simple as a checklist. As long as the adequacy of the specification is justifiable and reasonable, there usually is not much else to be considered here. That is not to say that the specification should lack detail or clarity; preconditions (input data) and configurations used (a subset or superset of variants) are wonderful things to document. The test environment conditions are also important bits of information to specify to allow for repeatability of testing should failures or anomalies occur.

We can refer to the design architecture and, even better, the schematics, to go page by page and construct our list of interfaces and circuits to be verified; for the SoC Platform here is a mostly complete list of interfaces to be verified.

- 10/100 Ethernet:
  - 100BASE-TX Physical Layer;
  - MII/RMII interface (payload data);
  - MDIO interface (PHY management).
- DDR3;
- SPI:
  - Accelerometer;
  - Infrared image sensor (video data);

    - Magnetic field sensor;
    - Vibration sensor;
    - Configuration memory.
- I2C interfaces:
    - Infrared image sensor (configuration);
    - Local temperature and relative humidity sensor;
    - Nonvolatile memory.
- SD/MMC;
- Debug UART:
    - USB 2.0 Physical Layer;
    - TTL UART interface.
- USB on-the-go port:
    - USB 2.0 Physical Layer;
    - ULPI interface.

That is quite the list. Going page by page and breaking it down, we essentially have created an outline for the majority of our test plan. Now the details can be filled in: how the tests will be executed, what measurements will be made, what the pass/fail criteria are, and so on.

#### 2.7.3.1 Don't Forget about Software

For many of these tests, hardware engineers cannot do it alone; the assistance of software and firmware is required. It is impossible to test an Ethernet interface without some code running that can generate and respond to messages, for example.

Once a rough idea of the plan is in place, engage with the software and firmware engineers, get feedback, and then, if compromises are needed, work through them so that there is little wasted work, but enough supporting code to thoroughly vet the hardware.

The earlier the other disciplines are involved, the smoother the testing process will go, and that is really the best-case scenario for all involved.

For these kinds of tests, a full-bore software image is not necessary. It can be completed piecemeal, with many small, simple programs loaded into the device one at a time, purpose-built to test a specific interface. This is all about verifying the underlying hardware platform. Keep it as simple as possible.

### 2.7.4 Example: I2C Interface

As a simple example of how to approach verifying an I2C interface, let's consider the nonvolatile memory in the SoC platform design.

The memory device is the Atmel/Microchip AT88SC0808CA; it runs a 2-wire I2C-compatible interface at up to 1 MHz. These simple datasheet facts provide an excellent starting point for defining our test case.

Let's consider the use case for the product; values will likely be periodically written to the device and also read back from it. (For some external memory

devices, such as those that hold a bootloader, readback of data from memory to host may be much more important to test than write performance to the memory. It is always good to understand the use case of each component.)

In this design, the interface communicates at 3.3-V logic levels and it is connected to the Zynq processor complex.

Let's summarize what we have so far:

1. I2C interface;
2. 3.3-V voltage levels;
3. 1-MHz maximum frequency;
4. Write values;
5. Read values.

We are starting to piece a test case together, which is excellent progress. Let's dig a bit deeper into the details. How can we really make sure that the device is working correctly?

#### 2.7.4.1 Read, Write, Verify

One of the best ways to verify a memory device is to write a pattern of alternating 0s and 1s to the memory and then read back those same memory locations. By doing so, we are putting a relatively high stress level on the device and also checking both directions of the interface. This checks the drivers and receivers on both interfaces, which is very important to do.

#### 2.7.4.2 Specifications

In nearly all digital communications devices, there will be both dc and ac characteristics provided for the device; these are very important to review during the design phase, and equally important to review when designing a test plan (Table 2.7).

Values for minimum and maximum voltage levels that are interpreted as 0s and 1s are given in the dc specification, and things such as rise time and fall time are given in the ac specification for the device.

From the dc characteristics, we know now what voltage levels to expect on the interface and what voltage levels must be met for proper operation for both inputs to the device and outputs from it. For the I2C interface, the clock is always an input to the device; during a write transaction, the data is an input, and during a read transaction, it is an output.

Therefore, when we write to the device, we need to provide signals that reach at least as low as 3.3 * 0.2 (remember, 3.3V is our VCC) or 0.66V. When we read from the device, we should expect that the voltage level for a logic state of 1 should be no less than 3.3 * 0.7, or 2.31V.

From the ac characteristics, we can now compare the measurements of the signals on our design to the requirements of the device (Table 2.8). In this case, they are dependent on the period of the clock cycle. For a 1-MHz clock, the period is 1 $\mu$s; 9% of 1 $\mu$s is 0.09 $\mu$s, or 90 ns. This is a really clear pass/fail criteria against which we can measure and verify our design's performance.

**Table 2.7** Selected DC Characteristics for AT88SC0808CA

| *Symbol* | *Parameter* | *Min* | *Typical* | *Max* | *Units* |
|---|---|---|---|---|---|
| VIL | Input low voltage | 0 | — | Vcc * 0.2 | V |
| VIH | Input high voltage | Vcc * 0.7 | — | 5.5 | V |
| VOL | Output low voltage | 0 | — | Vcc * 0.15 | V |
| VOH | Output high voltage | Vcc * 0.7 | — | Vcc | V |

**Table 2.8** Selected AC Characteristics for AT88SC0808CA

| *Symbol* | *Parameter* | *Min* | *Max* | *Units* |
|---|---|---|---|---|
| tR | Rise time | — | 9% * period | $\mu$s |
| tF | Fall time | — | 9% * period | $\mu$s |

#### 2.7.4.3 Other Options

In addition to these bench measurements, we can also use advanced functions on a logic analyzer or oscilloscope to decode the transactions as they occur in real time and verify that they are being executed as we expect. For example, if our test included a write to sequential locations in memory of the values 0x00 through 0x0F, we could capture that on a logic analyzer or oscilloscope and verify that the messages did contain that data. This does have a few caveats, such as configuring the thresholds of the decoding software to match the device that we are testing, but it can be a very valuable test with a simple visual cue of passing or failing.

### 2.7.5 Additional Thoughts

What we have outlined here is a good framework for which to approach testing a PCB, but there are more rigorous tests to consider. Once basic functionality and measurements have been made the test cases, if properly designed with some forethought, they can be reused and automated to run automatically. This can be very useful for collecting statistically relevant data and also for automated testing over temperature. Recall from ISO26262 that this kind of basic functional verification over temperature is a very strong data point for proving design robustness, so plan ahead and think through the whole plan before starting; it can really save time down the path of further verification and provide useful tools for manufacturing engineering, failure analysis, and regression testing. A little planning can go a long way in the world of testing, so do not be afraid to slow down up-front, so you can speed up at the back end of the process.

## 2.8 Integrity: Important for Electronics

In an earlier part of our journey through developing this embedded system, we talked about the high-level process of verification and validation. We broke testing down into levels 10 through 40 to align with the V-model, and we used ISO-26262 as a framework and guide to better understand how to derive test cases and what kind of tests provided high confidence levels in critical systems.

Part of verifying your design comes in the form of simulation, and this can take on many forms in embedded system design. For hardware specifically, we can consider two broad topics for simulation: power integrity and signal integrity.

A note to the reader: The topics of power and signal integrity are massive topics to try to explain in a concise manner, and they warrant books and conferences of their own. Because of that reality, it is not within the scope or spirit of this particular book to dive into great depths of detail for all things related to power and signal integrity. The intent of this brief discussion is to bring awareness to the topics for further consideration in your particular project or design. Many titans of the industry have written excellent books on the topics: Howard Johnson, Martin Graham, and Eric Bogatin are among our favorites, but there are many other excellent resources as well.

### 2.8.1 Power Integrity

Power integrity (in the context of the embedded system that the engineers at SensorsThink have been designing) refers to the quality of power being delivered to components within the system. It is an analysis (or series of analyses) that helps to determine whether or not the voltage and current requirements of the components in the design are being met. Without a properly designed power delivery system, many kinds of problems can occur in the system: increased bit-error rate (BER), reduced timing margin, and, in some cases, even brownout conditions within devices. The considerations outside of decoupling that are taken into account when designing for power integrity are mostly related to PCB stack-up design for copper weights, dielectric thicknesses for embedded capacitance, and trace widths that are necessary to carry currents with acceptable dc drop. Power integrity has several key factors that should be considered and analyzed during the design phase of the product: dc drop analysis, target impedance analysis, and simultaneous switching analysis.

### 2.8.2 Signal Integrity

Signal integrity, at its most basic and fundamental level, is a way to measure the quality of electrical signals. This is most commonly analyzed in high-speed digital systems: DDR2/3/4 memory, peripheral component interconnect (PCI) Express, USB 3.0, serial advanced technology attachment (SATA), and so on. As data rates of electronics have grown rapidly, into the range of tens of gigabits per second in recent years, this analysis has become mandatory for the design of systems that use these kinds of communications channels. The consideration for designing these interfaces into a system are far-reaching; component selection, PCB material choice, via size and spacing, PCB stack-up, trace width and space, acceptable channel characteristics such as insertion and return loss, and the power delivery network all play a role in ensuring a robust interface. In terms of types of analyses that are performed, it often depends on the type of interface being tested. For the sake of discussion, we can consider a few kinds: basic signal integrity analysis, DDR analysis, and serializer/deserializer (SERDES) analysis (PCIe, Ethernet, and USB are very common examples of SERDES interfaces).

### 2.8.3 Digging Deeper into Power Integrity

#### 2.8.3.1 DC Drop Analysis

The dc drop analysis is analysis that is done to ensure that the IR losses of the power delivery network (PDN) interconnect do not introduce excessive voltage drop. With modern devices, high current consumption on the order of 100A is not unheard of for core voltages of FPGA and SoC devices. It takes very little impedance to cause what can be a significant voltage drop on the interconnect from the power supply source to its destination. When performing dc drop analysis, several pieces of information should be readily available and well understood in order to get the most meaningful data from the simulations:

1. Copper weights (copper thickness) of power distribution layers of the PCB;
2. Voltage input requirements of critical loads, so that pass/fail criteria can be set appropriately;
3. Worst-case current consumption for loads, so that maximum IR drops are calculated correctly;
4. Any component-specific current limits for pins on connectors or device packages.

With all of the above information, a very in-depth analysis can be performed: extended networks through regulators and to downstream loads to provide high confidence levels in the capability of the power delivery interconnect to provide the power needed for the design.

#### 2.8.3.2 Target Impedance Analysis

Target impedance analysis characterizes the ac performance (e.g., performance across the frequency domain) of the PDN in the system. For high-frequency switching components especially, this is crucial to achieving expected performance of the components. Recall Section 2.4; we touched on four different categories of decoupling capacitor and how each type filled a specific role in providing adequate power to the component.

In order to gain the most value from this kind of analysis, the following pieces of information should be available as inputs to the simulation:

1. *The desired target impedance for critical components:* this type of information is sometimes available in datasheets; other times, more direct questions must be asked of IC suppliers to get the information.
2. *Accurate capacitor models for components used in the design:* Typically, these will be in the form of s-parameter files. S-parameter files are considered to be more accurate in many cases for high-frequency analysis because they can more accurately reflect nonlinear behaviors such as the variability of series inductance over frequency.
3. *Clear understanding of the specific pins of interest for the devices that require this analysis:* Not all package pins are created equally, and a good understanding of which pins supply which particular sections of a silicon

device can be important on devices with hundreds of supply voltage and return pins.

#### 2.8.3.3 Simultaneous Switching Noise

Simultaneous switching noise (SSN) occurs when a high number of signals inside of a particular component change state simultaneously. In this situation, a large amount of current is consumed by the device. These events, which we will refer to as SSN events, can cause serious issues with data integrity. It is possible, by using power-aware modeling techniques and software packages, to check for these kinds of issues and attempt to mitigate them before problems are committed to copper on a physical PCB. This particular type of analysis lives in the world of power integrity, yet also directly impacts signal integrity. Analysis like this can be performed with Power Aware IBIS simulations (from IBIS v5.1 onwards) and also with vendor-specific tools such as Vivado from Xilinx, which performs SSN analysis from within the tool chain. To accurately model SSN, accurate device models are required above all else; accurate modeling of the capacitance of the loads that the signal drives is crucial. What is actually occurring during SSN events is the charging or discharging of these capacitive loads; therefore, the necessity for accurate driver models and receiver models is paramount. Another requirement is a software package capable of performing Power Aware IBIS analysis; there are several excellent solutions for this on the market from Mentor Graphics and Ansys SIwave.

### 2.8.4 Digging Deeper into Signal Integrity

#### 2.8.4.1 Basic Signal Integrity Analysis

Basic signal integrity analysis is the first kind of analysis that probably comes to mind when engineers think about signal integrity. What we mean by basic signal integrity is running simulations on lower-speed signals and interfaces; in our design, we have many examples of where this type of analysis can and really should be performed:

- *SPI:* To sensors and memory;
- *RMII:* The interface to the Ethernet PHY;
- *ULPI:* The interface to the USB OTG PHY.

The reason that this type of analysis is becoming more common is due to the increased edge rates of modern silicon. Even devices such as logic gates from the 74LVC logic family boast rise time specifications on the order of only a few nanoseconds. An important concept to bear in mind with signal integrity is that the frequency of the signal is sometimes not the cause for concern, but rather the rise and fall times of the edges. A 1-ns rise time is equivalent to frequency content occurring at 1 GHz. At these frequencies, it is easy to see why designing with signal integrity in mind is so important. Performing basic signal integrity analysis requires accurate driver and receiver models and basic information about the interface such as:

1. Operating voltage level, in order to correctly select the model of the buffer to use in simulation. For even simple devices such as logic gates, there are often IBIS models describing the behavior of the driver at various nominal operating voltages. Choosing them incorrectly can lead to very misleading and strange results.
2. Operating frequency, in order to properly stimulate the interconnect. For an SPI that is intended to run at no faster than 10 MHz, using a 500-MHz input stimulus provides no value.
3. Specific drive strength and slew rate information for drivers. In modern devices, it is not uncommon to have multiple options for setting the drive strength (given in mA, typically) and slew rate (fast, medium, slow) for individual IO or specific peripherals. Changing these parameters can have a drastic impact on simulated and real-world performance.

#### 2.8.4.2 DDR Analysis

DDR analysis is a relatively new advancement in the world of signal integrity modeling. As DDR memory has become a staple in the world of embedded systems, software packages have evolved to accommodate the analysis of an entire DDR interface. These simulations can take into account various on-die termination schemes, analysis of timing margins, and voltage overshoot and undershoot of the interface. This provides a wealth of information to designers before committing to physical boards and can be invaluable during the design process. For advanced simulations, you can include things such as crosstalk between signals and coupling through plane structures; in some cases, the tools even support simulated calibration as an option. In the SoC Platform, we have a DDR3 interface that would absolutely be considered essential to simulate both in a pre-layout and post-layout environment. Understanding the available parameters in DDR controllers and memory devices is paramount in this kind of analysis, with so many options for operating speeds and internal termination; proper selection based on the specific use case of the design being simulated is essential.

#### 2.8.4.3 SERDES Signal Integrity Analysis

SERDES simulations are generally more involved, because they are typically going to require something called a passive channel analysis, where the interconnect of the differential signal is analyzed for a variety of characteristics such as insertion loss, return loss, and near-end and far-end (NEXT and FEXT) crosstalk (Figures 2.17 and 2.18). For many of these high-speed serial standards, there are plots that the simulation can be compared against to determine margins for the link. In addition to these passive channel analysis, virtual eye diagrams can be constructed in order to assess margins on the eye opening both vertically (voltage) and horizontally (time domain). This type of analysis is the most demanding, often requiring (in addition to what is required for other signal integrity analyses) s-parameter models for the transceivers and accurate s-parameters of the interconnect of the IC package and PCB geometry, and, in some cases, specialized software is recommended (Seasim, for example, is the supported tool used by PCI-SIG, the PCI special interest group).

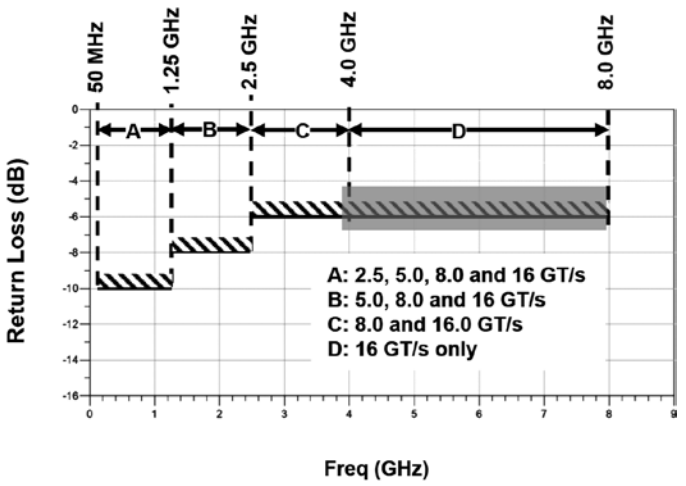



**Figure 2.17** Differential return loss mask for PCIe [21].

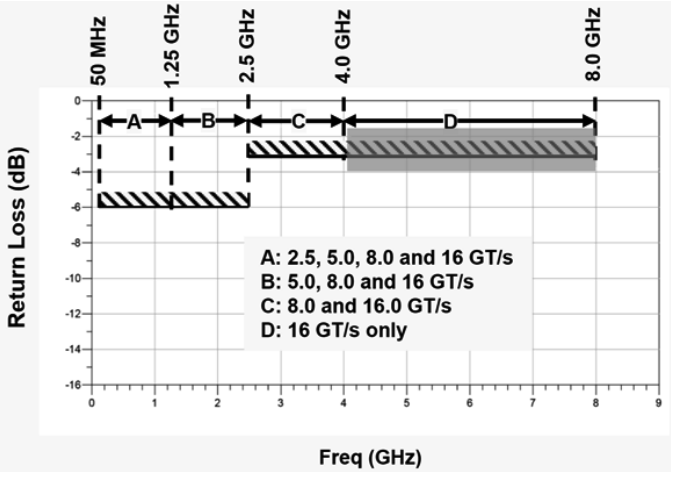



**Figure 2.18** Common-mode return loss mask for PCIe [21].

#### 2.8.4.4 Time-Domain Reflectometry (TDR)

Lastly, for all types of interconnect that runs at very high speeds (some example interfaces to consider as candidates would be PCIe, USB3.0, and Aurora LVDS), the simulation of the breakout from connectors and device packages is often modeled and then evaluated using a virtual TDR plot to judge the impedance of the traces, vias, and packages of the components in the time domain, essentially looking at the effective impedance that the signal experiences as it travels through the interconnect. Large impedance differences, often referred to as discontinuities, can cause signal reflections and have negative impact on functional performance.

### 2.8.5 Example: ULPI Pre-Layout Analysis

For one example, let's consider the ULPI interface between the Zynq SoC and the USB OTG PHY device. In this example, we will be using Mentor Graphics'

HyperLynx software to perform the pre-layout analysis and to verify that we have a sound design architecture in mind for the ULPI interface.

In the exercise of pre-layout simulation, the goal is to ensure that under idealized conditions the interfaces and design details (such as selected series termination) produce a very clean and idealized signal.

If the pre-layout simulation does not show nearly ideal signal behavior, it is an indicator that further investigation is warranted. There is something that is amiss, either with the setup of the simulation or with the design topology and component selection. Regardless of root cause, the time to find it and address it is certainly in the pre-layout stage and not later.

In addition to validating design decisions, pre-layout signal integrity analysis is a wonderful learning tool, as you can freely try things and look at the impact to the signals without any real-world risk or impact; this kind of learning opportunity is always welcomed in engineering.

For the ULPI interface, we have a relatively simple architecture to consider:

- Driver/Receiver 1: Xilinx Zynq FPGA;
- Transmission line;
- Driver/Receiver 2: USB3320 USB PHY.

Setting up this architecture in HyperLynx is relatively straightforward, using only a couple of components (Figure 2.19). Note that this also requires the IBIS models for the Zynq FPGA and USB3320.

By adding the two driver and receiver models to the design and an ideal 50Ω transmission line, we can quickly start to run an IBIS simulation, with only a few more pieces of input information provided to the tool:

- A sample stack-up of the PCB entered into the design tool;
- The simulation frequency or frequencies of interest;
- Configuration of any power supply nets required by the IBIS models;
- The voltage levels at which the design is intended to operate the ULPI interface;
- The expected distance between devices.

A note to the reader: PCB stack-up design is an art unto itself that is briefly covered later in this book. For the purposes of pre-layout signal integrity, the

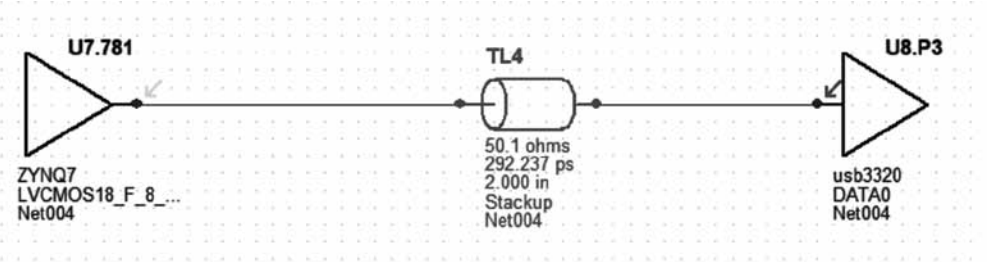



**Figure 2.19** HyperLynx schematic model of the ULPI data path.

requirements are only to have a stack-up well enough defined to facilitate the creation of transmission lines of a known impedance.

For our design, the interface is intended to operate from a 1.8-V supply, and, per the ULPI specification, the maximum clock frequency and data rate are 60 MHz. This allows us to run a baseline simulation and view the waveform result relatively quickly. In order to get a clear view of the quality of the signal, it is possible to run an edge simulation (either rising or falling) or an oscillator simulation at a specific frequency. Additional options can be set for the corner of interest: slow-weak, typical, or fast-strong. In this example, we will run an oscillator simulation at 60 MHz of the typical IBIS model. This simulation will be of data flowing from the Zynq to the USB3320 device.

We can observe some signal overshoot and ringing on this initial simulation (Figure 2.20), and this is something for us as designers to watch during post-layout analysis (which takes into account actual routing geometry). However, we can also study the impact of series termination so that if post-layout analysis reveals similar problems, we can have a fix ready to implement.

A note to the reader: In a classical sense, series source termination is placed as close as possible to the transmitting device in order to try to better match the source impedance of the driver to the transmission line. On a bidirectional bus, it is ideal to place the termination at the mid-point between the two devices in order to try to match both transmitting devices. It is important to keep in mind that the ideal signal is almost impossible to implement on real electronics, which is why pass/fail criteria and a thorough understanding of the way that an interface works are crucial to these kinds of exercises.

In the revised circuit (Figure 2.21), we have split the transmission line in half and placed a 22Ω resistor in series with the transmission line pair.

We can see that the signal quality is greatly improved on this simulation run, with significant reduction in overshoot and ringing (Figure 2.22).

These kinds of simulations can be repeated in all kinds of varying conditions: longer transmission lines, different driver models, additional fanout, and different frequencies. Knowing what to simulate at different stages of the design comes with experience and a little bit of experimentation. However, the value and insight gained along the way are invaluable and should not be discredited as a reason to err on the side of simulation if any doubt exists.

### 2.8.6 Suggested Additional Reading

For additional information on power and signal integrity, there are a few excellent resources that are highly recommended [22–25].

## 2.9 PCB Layout: Not for the Faint of Heart

In terms of the amount of work that goes into designing a PCB, it can be argued that the majority of the hours put into the design are accumulated during the layout phase. Layout is the phase of PCB design in which the schematic interconnect is turned into the physical pieces of copper that (hopefully) result in a functional piece

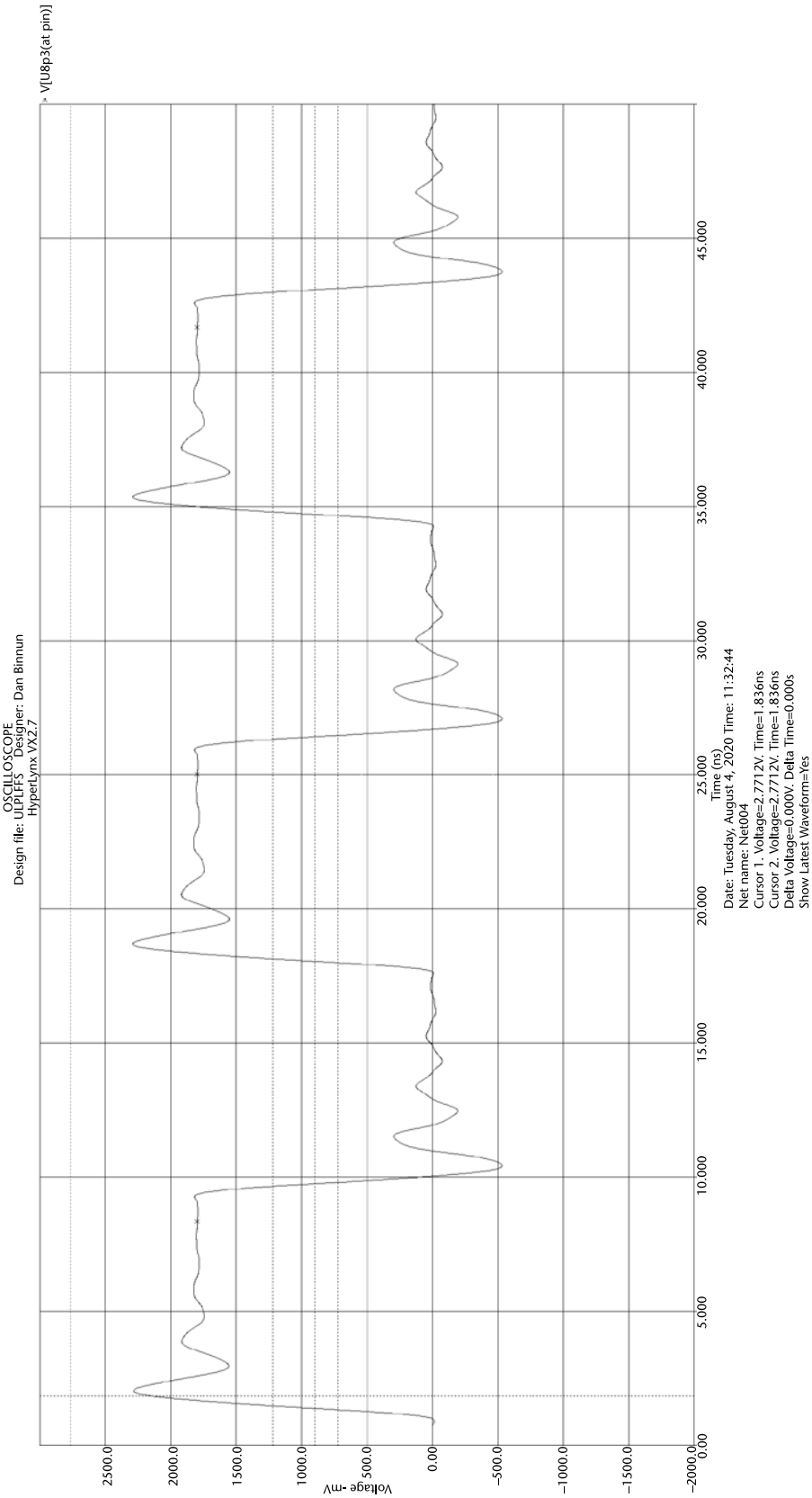



**Figure 2.20** HyperLynx pre-layout simulation: ULPI.

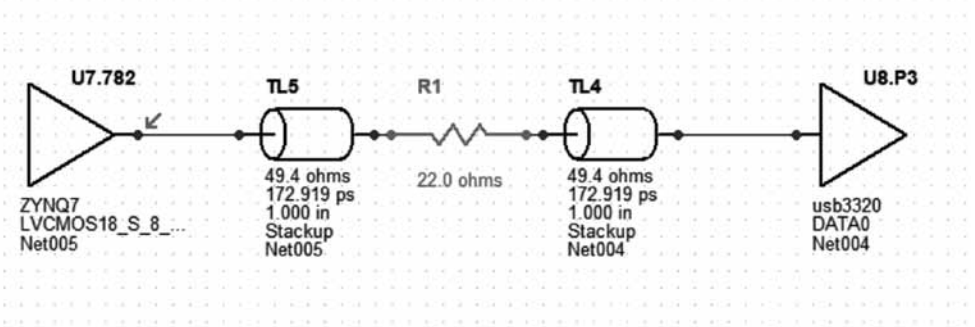



**Figure 2.21** HyperLynx schematic model of the ULPI data path: added series termination.

of hardware that implements the many months (and sometimes years) of planning, design work, and development efforts of countless engineers and management.

PCB layout is an often-misunderstood engineering discipline, sometimes even an overlooked one. Too often schematic netlists are "thrown over the wall" to an unsuspecting PCB design engineer who is given little input, less guidance, and the highest of expectations.

As with nearly all tasks in engineering, garbage in produces garbage out. If the engineers at SensorsThink provide a well-designed schematic along with a clear set of design constraints and layout considerations to follow and they engage in the layout as a collaborative effort, the team will be set up for success.

PCB layout is an art form for sure, but one that begins with a tremendous amount of planning and detailed consideration. There is nearly an endless amount of design trade-offs that will be made out of necessity throughout the process of designing a PCB, and navigating those trade-offs is both part of the fun and part of the battle of PCB layout.

### 2.9.1 Floor Planning

Floor planning is the part of the layout process that revolves around placing functional blocks of the schematic design in physical locations on the PCB. Like a well-designed and architected house, a good floorplan is important. It sounds easy enough to plop the circuits down in a linear flow and let them go, but it is not that easy.

There are countless constraints that must be considered in almost all embedded systems, the first of which is the rat's nest (Figure 2.23). This is the visual representation of the connectivity between components; it is at first overwhelming, like a knot that has been tied and retied repeatedly. Part of floor planning is like untangling the unruly bundled mess of lines with which every PCB design starts.

There are many other constraints that must be considered during floor planning. Again, we can leverage the analogy of a house that is well-designed and architected. Mechanical systems must be considered; heating, ventilation, and air conditioning (HVAC) duct work, water and sewer lines, and structural loading of the building are all critical aspects to consider. In the PCB design, we can draw corollaries to power and return plane structure; the locations of (mostly) non-negotiable

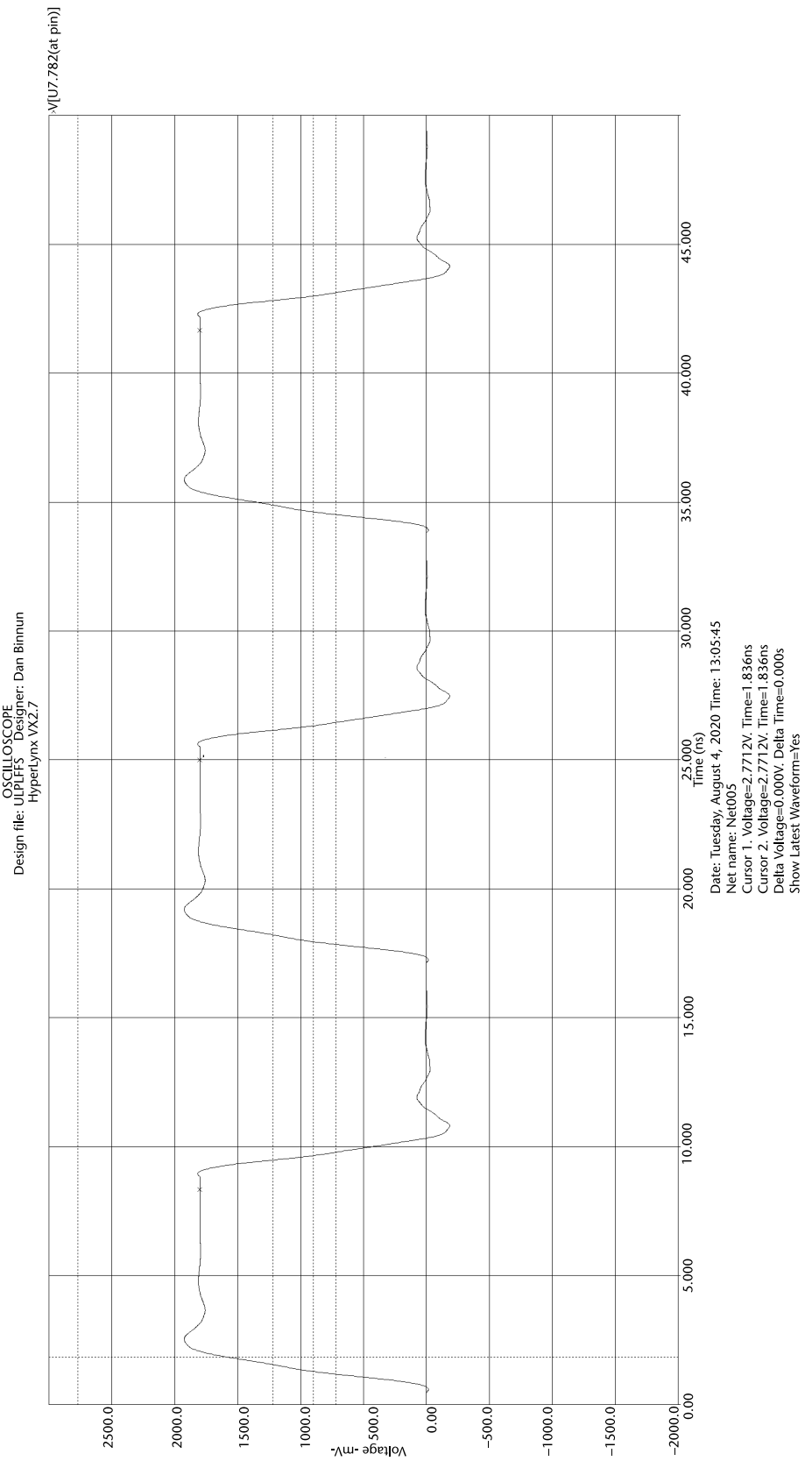



**Figure 2.22** Modified HyperLynx pre-layout simulation: ULPI.

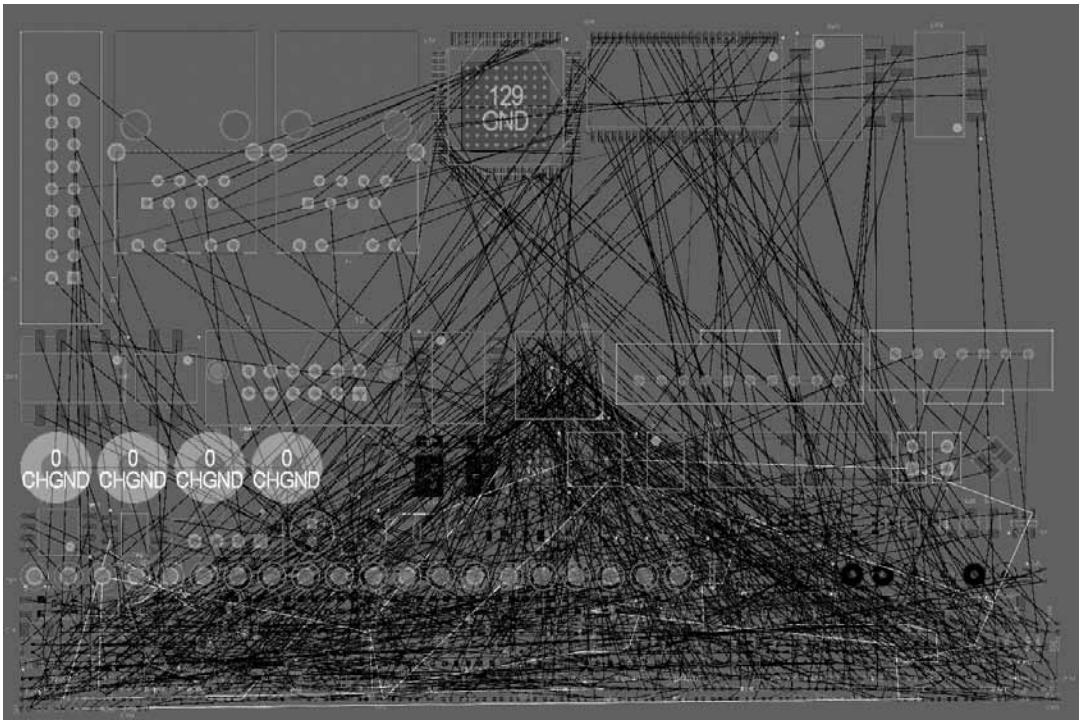



**Figure 2.23** A rat's nest view of a PCB design.

mechanical details such as mounting holes and the physical size of the PCB are all important to consider from the very start of floor planning.

Electrical constraints are yet another consideration. What circuits are sensitive and likely to be impacted by electrical noise? What circuits are likely to annoy their neighbors? In keeping with the theme of a house, where do we want our jacks for cable and internet? Where does the dishwasher go or the washing machine? These things are considered long before the carpenters and tradespeople begin the work of building the house. If they are not, we can imagine that the home would be built ad hoc, and the results would likely be a mixed bag at best with even the most talented tradespeople performing the labor.

### 2.9.2 Follow the Rats

As we touched on earlier, the rat's nest is one of the major constraints and design inputs to consider when doing PCB layout and floor planning. An example that seems straightforward to experienced engineers may not be so obvious to those starting out in the field. To illustrate this, consider an Ethernet interface and physical connectors. In Figure 2.24, we see an attempted placement of two RJ-45 connectors and an Ethernet PHY device. The components are placed close to each other, and the connections look fairly straightforward to make. This is probably a very "routable" and achievable electrical interconnect to make, but there is room for improvement (Figure 2.24).

Let's make an iterative change and try to improve things by moving the location of the connectors. In Figure 2.25, we see that this does appear to be an

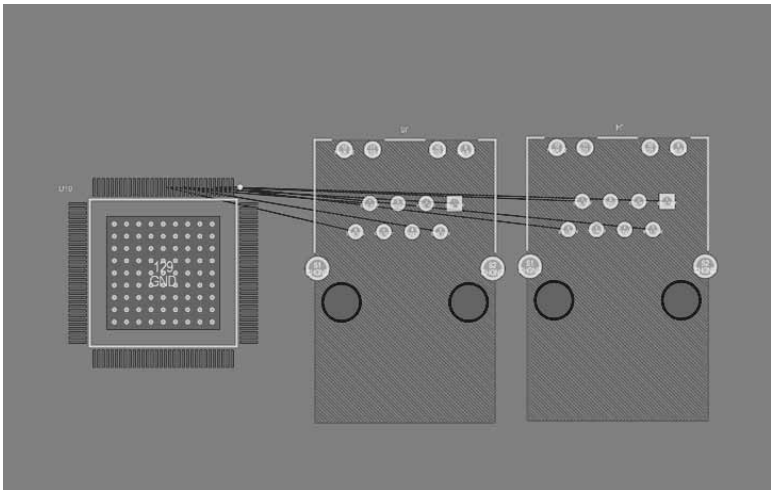


**Figure 2.24** Rat's nest placement: could this be better?

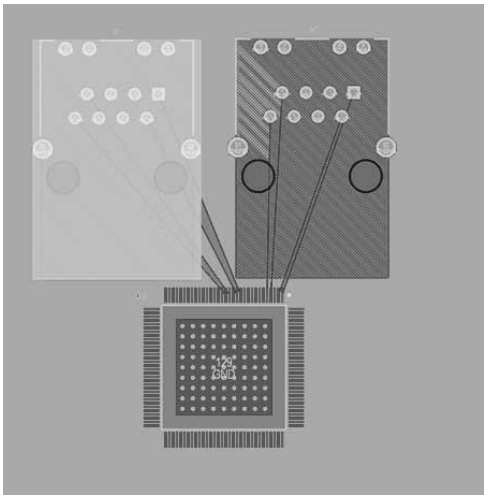


**Figure 2.25** Untangling the rat's nest.

improvement. The connections now all have a clear line of sight to the pins on the IC. While this does look promising, a quick look in a 3-D rendered environment will show the problem with the current arrangement (Figure 2.25).

We can imagine that it might be less than ideal to try to plug a cable into the connectors with the IC sitting right in front of the jack (Figure 2.26).

If we take one more attempt at this, we can probably arrive at a reasonable placement for these three components, relative to one another.

In Figure 2.27, we arrive at what is likely the best we can do with the information that we have at hand for these three components and the rat's nest connections that are showing. It looks like some of the lines are crossing over each other (Figure 2.27); it was much better before. It was, but it was not a reasonable physical approach when we viewed things in the 3-D world in which we live. So we had to make the first of the many trade-offs that will be necessary during the process of

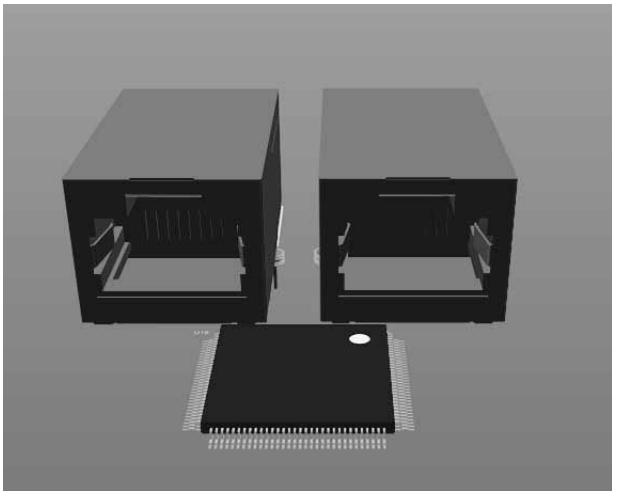


**Figure 2.26** What about the cable?

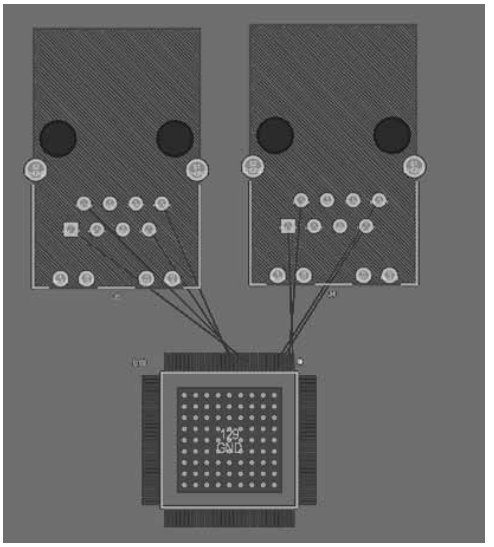


**Figure 2.27** As good as it gets?

laying out this (and any) PCB. There are many ways to unravel these crisscrossed lines during the routing phase of the design, and many of those tricks are learned by doing and come from years of practice. Experienced PCB layout engineers see these solutions as quickly as they see the problems, and the trade-offs that must be made are clear to them very quickly. Until that experience is something that can be relied upon, mistakes will be made and learned from; all of that is part of the process.

### 2.9.3 Mechanical Constraints

Mechanical constraints are often provided to the PCB layout engineer from the mechanical engineering team. The overall size of the PCB is usually not overlooked,

and mounting considerations are usually thought of as well, be it a card-guide system or old-fashioned screws that will hold the board to some kind of enclosure.

If PCB layout engineers are asked how much space they need for a PCB, they will almost always say "as much as possible." That is really their ideal answer. Unfortunately, there is rarely a situation where space is not at a premium. If product managers are asked how much space there is for a PCB, they will almost always say "as little as possible." That is really their ideal answer. Unfortunately, there is rarely a situation in which that is actionable by engineering. These two worldviews have to meet in the middle somewhere.

Aside from the available footprint and mounting considerations, you might be wondering what else mechanically is a true constraint on the PCB layout process. Here are a few to consider:

- *Connector locations:* Oftentimes, connector locations are determined by industrial design: what connectors have to be on the front, back, or sides of an enclosure based on how a system will be installed or used. This can be both a burden and a blessing to floor planning. If, for example, we know that the RJ-45 connectors must be on the right side of the PCB, we have a good idea of where the Ethernet PHY should be placed. An additional layer of complexity here is cable egress considerations: is the cable exiting from a specific side of the enclosure? If so, does a right angle connector or vertical connector allow that to be implemented more easily?
- *Height restrictions:* Height restrictions mean two things in this particular context. The first meaning is the overall height permissible in certain areas of the PCB, on the top and bottom sides of the board and in specific regions where perhaps the enclosure or environment otherwise might interfere with tall components. The second meaning is in card-based design, where the PCB thickness is based upon one or more existing standards because it has to mate with a specific type of connector in the end application; VPX systems, PCIe cards, and SODIMM form-factors are all prime examples of this.
- *PCB shape and cutouts:* In some instances, a PCB will need to have a unique shape or a shape that fits around a mechanical feature inside of an enclosure. This is generally known up-front when the product is conceptualized, but does not mean that there is any lack of challenge with integrating the constraint as the PCB layout process evolves.

Accommodating shrinking products and consumer expectations is challenging, and taking all of those fixed constraints into account during the floor-planning process is essential to reducing wasted effort and frustration for the entire product development team.

### 2.9.4 Electrical Constraints

This particular topic may seem a bit redundant in a discussion about PCB layout. Everything is electrical. However, in this context, we should reframe what we mean by electrical constraints to be a bit more direct. There are a few ways we can do this, so let's begin.

In nearly every design, there are going to be two types of circuits: circuits that create noise and interference, and circuits that are particularly susceptible to noise. In floor planning, understanding which circuits are noise creators is vitally important; equally important is knowing which circuits are susceptible to malfunction in the presence of noise. Once these circuits are identified, we can try to use physical space between the two types of circuits as our best ally for mitigation against unwanted interference and circuits that misbehave (Figure 2.28).

Some circuits that are typically considered noise creators are power supplies and high-speed switching digital electronics, and analog signal conditioning and data conversion are typically circuits that ideally are kept free and clear of these noise creators. In Figure 2.28, we see an example of a floor plan that can be used to help mitigate circuit interference using physical space as one of the primary means of achieving the goal.

Another type of electrical constraint is high-voltage considerations, be it in the form of ac mains-connected power or high-voltage dc power supply sources that are used in scientific instrumentation systems. Understanding where these interfaces and interconnect are going to be at the start of the design can make or break the success of a high-voltage PCB layout. Once those constraints are understood the PCB design engineer can begin considering how to address them; it could be slots in the PCB to prevent high-voltage breakdown, or if it is not possible to accommodate the required distances, encapsulation may be required. Surrounding the components with a material that can withstand and suppress high-voltage discharges better than air can may be necessary.

A note to the reader: High-voltage distances are called creepage and clearance distances and are the distance along a surface and through the air, respectively. IPC-2221B is an example of one reference for determining the required creepage and clearance distances needed in a specific situation. Also important to consider are product safety directives and standards, which may have additional constraints to consider.

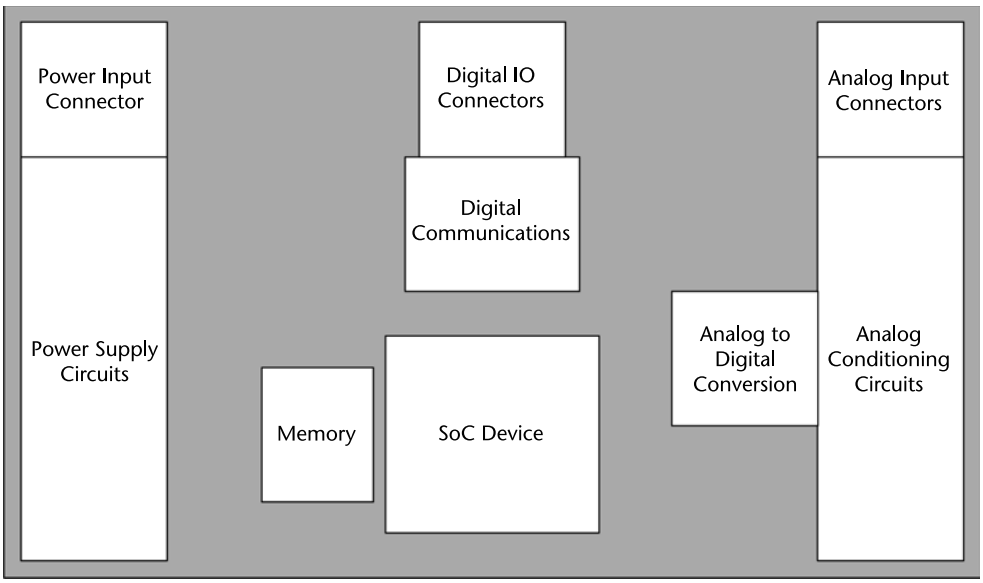



**Figure 2.28** PCB floor plan.

### 2.9.5 Stack-Up Design

All PCB designs rely heavily on their stack-up design to function well. Continuing on with the analogy of a PCB being similar to a house, we can think of the PCB stack-up as the foundation on which the design will be built. Stack-up design is a crucial step in PCB design, and, in a recurring theme, there is much to consider when designing a stack-up. The thickness of copper, the amount of routing layers, the type of dielectric material, the cost of manufacturing particular feature sizes, and the trade-off of making routing easier versus making the process go more quickly are all on the table for discussion and consideration.

What is a stack-up design? A stack-up design is the cross-section of the PCB, the layers of the PCB sandwich as it were. In a foundation for a home with a basement, there is a footing, a foundation wall, and a concrete pad that all must be poured. They must be poured with the right materials, at the right times, and with the right amounts to function as intended (Figure 2.29).

#### 2.9.5.1 Layer Count

Oftentimes, the first point of discussion will be the layer count of the PCB. Will it be a 4-layer board? Will it be an 8-layer board? For one reason or another (probably due to cost), this is usually where the discussion on stack-up design starts. When determining the number of layers needed, experience can be a helpful barometer for knowing how many layers will be necessary. However, it is not the only way to start.

With modern ball grid array (BGA) devices, there are some practical limits that can guide us to a reasonable starting point for the number of routing layers that we

| # | Name | Material | Type | Weight | Thickness | Dk |
|---|---|---|---|---|---|---|
| | Top Overlay | | Overlay | | | |
| | Top Solder | Solder Resist | Solder Mask | | 0.787mil | 3.5 |
| 1 | Top Layer | | Signal | 1oz | 2.008mil | |
| | Dielectric 1 | PP-006 | Prepreg | | 3.465mil | 3.77 |
| 2 | Layer 2(GND) | CF-004 | Plane | 1/2oz | 0.709mil | |
| | Dielectric 2 | Core-009 | Core | | 2.992mil | 4.25 |
| 3 | Layer 3(Mixed) | CF-004 | Signal | 1/2oz | 0.709mil | |
| | Dielectric 3 | PP-006 | Prepreg | | 5.551mil | 3.75 |
| 4 | Layer 4(PWR) | CF-004 | Plane | 1/2oz | 0.709mil | |
| | Dielectric 4 | Core-009 | Core | | 2.992mil | 4.25 |
| 5 | Layer 5(GND) | CF-004 | Plane | 1/2oz | 0.709mil | |
| | Dielectric 5 | PP-006 | Prepreg | | 5.551mil | 3.75 |
| 6 | Layer 6(Mixed) | CF-004 | Signal | 1/2oz | 0.709mil | |
| | Dielectric 6 | Core-009 | Core | | 2.992mil | 4.25 |
| 7 | Layer 7(GND) | CF-004 | Plane | 1/2oz | 0.709mil | |
| | Dielectric 7 | PP-006 | Prepreg | | 3.465mil | 3.77 |
| 8 | Bottom Layer | | Signal | 1oz | 2.008mil | |
| | Bottom Solder | Solder Resist | Solder Mask | | 0.787mil | 3.5 |
| | Bottom Overlay | | Overlay | | | |

**Figure 2.29** An example PCB stack-up.

will need. Xilinx provides an excellent resource that walks through a sound methodology for estimating the number of routing layers needed to accommodate a PCB design that includes BGA devices of various sizes. We will be referring heavily to this resource, UG1099 BGA Device Design Rules.

A very quick way to estimate the number of layers required is based on the following equation:

$$Layers = \frac{Signals}{Routing\ Channels \times Routes\ Per\ Channel}$$

For Xilinx devices, a ratio of 60% of the total BGA pads is given as an approximate rule of thumb for finding the number of signals that need to be routed. The other 40% are typically power supply and return pads.

A routing channel is defined as a path to get a trace from a BGA pad to the area of the PCB outside of that pad, and the number of routes per channel is the number of traces that can fit within each channel, almost always one or two.

If we use our Zynq device on the SoC Platform as an example, we can determine how many routing layers that we will need to reserve for the PCB design.

With 400 BGA pins in our device package (the CLG400), we can assume that at maximum we will have 60% of these pads to route out to other components; 60% of 400 is 240.

The number of routing channels can be determined by examining the PCB footprint, and we can refer to Xilinx UG865 for this.

With 20 total rows and columns, there are 19 total routing channels available per side; and with four sides, that calculates out to be 76 routing channels in total for the device.

The number of routes per channel can be either one or two; we will calculate the required layers for both of these cases. The reason for this is that, right now, we do not have all of the constraints in place to rule out one approach or the other.

We can now plug these values into the equation with which we started to get a rough idea for the required number of routing layers:

Case 1: One Route Per Channel

$$Layers = \frac{240}{76 \times 1}$$
$$Layers =\sim 3.2$$

Case 2: Two Routes Per Channel

$$Layers = \frac{240}{76 \times 2}$$
$$Layers =\sim 1.6$$

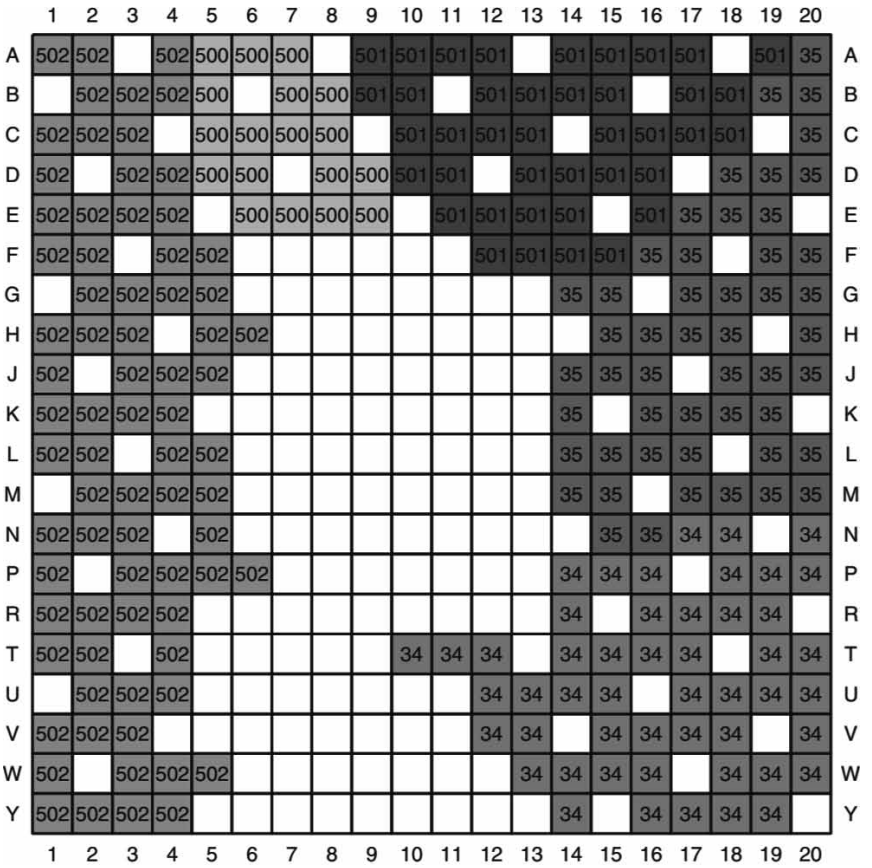



**Figure 2.30** The CLG400 land pattern.

We cannot have half of a routing layer, so we can say that we will need approximately two or three routing layers for this design in order to fully route the Zynq SoC device.

The additional steps of determining the possibility of routing one or two traces per channel can be followed step by step in UG1099. They will not be covered in detail here, but are mentioned for completeness and to give a clear overview of the complexities that can be involved.

In addition to this basic calculation, a slew of other considerations are important in determining what is feasible to expect, among them are:

- *Pad size and pad pitch:* The size of the BGA pad and the space between them are directly correlated to how many traces can fit in each routing channel. A 1.0-mm pitch device will be much easier to accommodate two traces per channel, and a 0.5-mm pitch device will almost assuredly not be able to do so.
- *Fabrication technologies:* Things such as via-in-pad, blind and buried vias, and micro vias (which are part of a category of PCB design technologies referred to as high-density interconnect (HDI) technologies) can greatly reduce the number of routing layers required by freeing up space in the PCB sandwich for other traces to be routed. We will cover HDI later in this chapter.

- *PCB thickness:* PCB thickness has a direct impact on the minimum mechanical drill size that can be performed on any particular design. A typical and safe aspect ratio is 10:1, meaning that the board cannot be any thicker than 10 times the diameter of the mechanically drilled hole. For a typical drilled hole of approximately 8 mils (0.008 inch), that means the thickest stack-up that we can design is 80 mils. If we try to add too many layers to a board and do not adjust the minimum drill size accordingly, our PCB will be impossible to manufacture.

Typically, performing this exercise for the highest pin-count and/or smallest pad-pitch device will yield a stack-up design and routing layer count that works for all the devices on the PCB.

However, having three routing layers is not the end of the story here, in addition to routing layers, we also have to consider power and return layers, and oftentimes the top and bottom layers of the PCB can be so full of components that routing on them is not always feasible.

Taking all of this into account, we can see that arriving at an 8-layer stack-up design is probably a reasonable starting point for this design. However, needing to add additional layers may be required, and ending up with a 10-layer or 12-layer stack-up would be completely reasonable as well.

#### 2.9.5.2 Copper Weights

Another piece of the puzzle with stack-up design is the weight or thickness of the copper. The copper weight is often one of the knobs that is turned when dealing with devices that have high current requirements. The thicker the copper conductors, the more current they can safely carry. However, it is important that this change in copper weight be done mindfully, as the thicker the copper is, the harder it is to manufacture fine pitch traces. So while 3.5 mil (0.0035 inch) can be created on a lighter copper weight of, say, 0.5 ounce, on a 2 or 3-ounce copper material, that trace geometry is not really feasible. One common practice is to use heavier copper weights on layers dedicated to power and return planes and lighter copper weights on routing layers or component layers with fine-pitch devices. This is a good way to get the best of both worlds: heavier copper for better power distribution and lighter copper for etching detailed features.

#### 2.9.5.3 Dielectric Types

A dielectric material is a material that is an insulating material or a very poor conductor of electric current. In the PCB stack-up design, these are the materials that sit between the conductive layers of copper, providing the necessary isolation between things such as power and return nets.

The most common materials used for dielectrics in PCB manufacturing are glass-reinforced epoxy, often referred to as FR-4 materials. These materials come in a wide variety, with differences in how the material is physically constructed (typically woven together) and how much glass fiber is present versus epoxy.

In addition to those fundamental construction differences, there are also varying thicknesses available, from as thin as 2 mils (0.002 inch) up to 125 mils (0.125

inch). The use of a particular style and thickness of dielectric depends greatly on the situation for which it is being designed.

A note to the reader: The best practice for creating and verifying PCB stack-up designs is to engage directly with the contract manufacturer that will be fabricating your PCB. This is the best way to make sure that your PCB stack-up design is able to be manufactured.

#### 2.9.5.4 Controlled Impedance

Controlled impedance is becoming a more common requirement in modern PCB design as the speeds of interfaces and edge rates increase. (As the edge rates increase, the rise and fall times decrease, making for sharper, faster transitions.) Controlled impedance refers to designing the physical geometry of a trace on your PCB to have a characteristic impedance, similar to the way that a BNC cable has a characteristic impedance of, say, 50Ω. Controlled impedance is important because without it, we cannot have robust transmission lines for our high-speed signals, RF signals, or analog signals. Without robust transmission lines, we run the risk of introducing enough degradation in the performance of the electrical channel to cause the interface to not function or not function well enough for the application at hand.

Designing for a controlled impedance or for several controlled impedances should be done as early as possible. Once we have established our routing geometries based on layer count and pad pitch, we at least have an idea of how wide our traces need to be in order to be properly routed. We can either provide this information to the contract manufacturer or leverage a PCB design tool that has the ability to calculate the characteristic impedance of a trace with respect to a reference layer.

The trace widths, required impedances, and material thicknesses and types all play a role in this calculation, and all the pieces of the puzzle can be difficult to fit together. Working hand in hand with the PCB contract manufacturer can be invaluable here in reducing wasted effort and rework; if a submitted set of files does not translate into the impedance that were targeted, the impact can be devastating to project timelines.

A note to the reader: Sometimes it is just not possible to have the ideal transmission line for all signals on a PCB for any number of reasons. If this is a situation that presents itself, remember that trade-offs are part of the game. Consider critical signal routes early on and design the rest of the stack-up around them, or perform signal integrity simulations to see if using narrower traces to route through a BGA has a significant impact on particular interfaces as early as possible to try to mitigate and manage the risks.

#### 2.9.5.5 HDI

Earlier in this chapter, we touched on several technology types that are part of a group of fabrication methods referred to as HDI. HDI is one of the more rapidly evolving spaces in the PCB manufacturing domain. IPC-2226 classifies PCB designs into six groups, or types, and also outlines material types and mechanical and physical properties that are acceptable for use in these types of designs. Let's refer back to our house analogy for PCB design. HDI is similar to adding fancy modern

gadgets and extras to your house. You may not need them, but they can make life easier. They almost always cost more than the old-fashioned way of doing things (like a tankless water heater versus a traditional tank-style water heater), but often come with performance benefits.

#### 2.9.5.6 What Is HDI?

In general, a PCB is considered to be HDI if it uses one or more of the following HDI technologies:

- *Micro-vias:* Typically, a via spanning two layers (layer 1 to layer 2, for example) and typically will be a laser-drilled via. These micro-vias usually require an aspect ratio of 1:1, meaning that they cannot drill deeper than the diameter of the hole that they create.
- *Via-in-pad:* This is exactly what it sounds like, drilling a via directly in the pad of a component. This technology requires an additional process step of plugging the via holes so that the solder does not drain into the pad holes. Via-in-pad can be a traditional through-via, micro-via, or blind-via.
- *Blind vias:* A via that only goes partially through the PCB, for example, on an 8-layer board, from layer 1 to layer 4. These vias can connect any two (or more) layers that they span.
- *Buried vias:* A via that is not exposed to the outer layers of the PCB, for example, on an 8-layer board, from layer 4 to layer 6. These vias can connect any two (or more) layers that they span.
- *Multiple laminations:* Any PCB assembly that needs to be laminated (pressed together) more than one time in order to accommodate blind or buried vias.

#### 2.9.5.7 When to Use HDI

HDI technologies have benefits and drawbacks, and sometimes as a designer your hand will be forced into using them. For example, on fine pitch devices, traditional mechanical drill sizes are simply too big to use.

- If you need to design a very small product with a lot of interconnect, you are likely going to need at least one HDI technology.
- If you have extremely high-speed digital interfaces, you are likely going to need some type of advanced HDI processing to achieve signal integrity goals.
- If you have very fine-pitch devices, you will likely need via-in-pad or micro-vias to even be able to fit a trace into the BGA area.

There are some handy graphs and plots for assessing when HDI may be the best path forward; the HDI Handbook [23] has one such region map that tells us that the key metrics for assessing this are component density (the number of parts per square inch) and component complexity (the number of pins per components). So just because you are working on a design with 1,000 components with 80 components per square inch, it does not mean that you need HDI if 998 of those

components have two pins. Conversely, just because a design has only 5 parts per square inch, it does not preclude the use of HDI if the components have an average of 80 pins.

#### 2.9.5.8 Exotic Materials

For some high-speed digital, RF, or other niche applications, standard glass reinforced epoxy simply is not up to the task at hand. Whether it is for transmission of extremely high-frequency signals or for dealing with high temperature environments, sometimes materials have limitations that cannot be overcome, even by the most seasoned designers. The most seasoned designers learn these kinds of situations and proactively solve the problem up-front. They can tell when an exotic material will probably be required and can save the product development team a lot of heartache by bringing these concerns to the forefront sooner, rather than later.

#### 2.9.5.9 An Example Material

For high-frequency RF or high-speed digital designs, it is common to use materials referred to as low-loss, high gigahertz. These materials demonstrate a couple of key characteristics that make them a more ideal fit for these applications. As an example, let's consider a popular material from Rogers Corp, RO4350B:

- Very tight control of the dielectric constant over frequency and environmental conditions;
- Low loss, or dissipation factor at high frequencies (specified as high as 40 GHz).

In the design of high-performance systems such as automotive radar or cellular base stations, these characteristics are essential to proper functionality. Compared to the performance of a standard FR-4 style material, the consistency and control of the dielectric constant are extremely important. In addition to changing with temperature, the dielectric constant shifts with frequency and can shift wildly within manufacturing tolerance of common materials. A material such as RO4350B is extremely consistent from lot to lot and can provide good peace of mind for designers working on high-performance RF systems.

### 2.9.6 Experience Matters

Last but certainly not least, it is important to know that the only way to get better at PCB design is to practice it. Experience matters. Knowing the tricks of the trade can help you to visualize solutions to problems before you head too far down a particular path or help you to determine whether or not a particular design is feasible to implement. These tricks of the trade and the confidence to tackle complex design work do not come without some hard lessons learned. This can be in the form of rework and lost hours, days, or weeks of PCB layout work or in expensive scrap piles of PCBs that just cannot be salvaged.

| Parameter | Frequency | Value |
|---|---|---|
| Dk, Permittivity | 100 MHz | 4.24 |
| | 1 GHz | 4.17 |
| | 2 GHz | 4.04 |
| | 5 GHz | 3.92 |
| | 10 GHz | 3.92 |

**Figure 2.31** Dielectric constant variation versus frequency of standard FR-4 style material.

There are many aspects of PCB design that we did not cover here, things such as design for manufacturability (DFM), design for assembly (DFA), and design for test (DFT). These topics are all broad and deep at the same time and really are worthy of entire books all on their own.

Hopefully, this has shed some light on the complexities and trade-offs to expect with PCB design work and why it truly takes a talented team of engineers to design a highly functioning PCB. If the engineering team working on the SoC Platform takes these factors into consideration, they will be on the path to success, even if there are many bumps along the way.

## References


[1] JEDEC, "About JEDEC," September 19, 2019. https://www.jedec.org/about-jedec.

[2] International Organization for Standardization, "ISO 26262-1:2018," August 22, 2019. https://www.iso.org/standard/68383.html.

[3] FLIR Commercial Systems, "FLIR LEPTON® Engineering Datasheet," 2014.

[4] Dhaker, P., "Introduction to SPI Interface," Analog Devices, Analog Dialogue 52-09, September 2018. https://www.analog.com/media/en/analog-dialogue/volume-52/number-3/introduction-to-spi-interface.pdf.

[5] FLIR Commerical Systems, "Lepton Software Interface Description Document, FLIR #110-0144-04," 2014.

[6] Xilinx, *Zynq-7000 SoC Technical Reference Manual*, UG585 (v1.13), April 2, 2021. https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[7] Xilinx, "Zynq-7000 SoC DC and AC Switching Characteristics," DS187 (v1.21), December 1, 2020. https://www.xilinx.com/support/documentation/data_sheets/ds187-XC7Z010-XC7Z020-Data-Sheet.pdf.

[8] *Merriam-Webster*, "Decouple," February 8, 2020. https://www.merriam-webster.com/dictionary/decouple.

[9] Murata, "Basics of Capacitors: Chapter 1," January 14, 2011. https://www.murata.com/en-us/products/emiconfun/capacitor/2011/01/14/en-20110114-p1.

[10] Murata, "Basics of Capacitors: Lesson 2," April 14, 2011. https://www.murata.com/en-us/products/emiconfun/capacitor/2011/04/14/en-20110414-p1.

[11] Panasonic, "Fundamentals of Capacitors and Hybrid Capacitors," July 8, 2019. https://industrial.panasonic.com/ww/ss/technical/lc3.

[12] Analog Devices, "Decoupling Techniques," March 2009. https://www.analog.com/media/en/training-seminars/tutorials/MT-101.pdf.

[13] Xilinx, "Zynq-7000 SoC PCB Design Guide," March 14, 2019. https://www.xilinx.com/support/documentation/user_guides/ug933-Zynq-7000-PCB.pdf.

[14] Mentor Graphics, February 2020. https://www.mentor.com/pcb/hyperlynx/signal-integrity/.

[15] Mentor Graphics, February 2020. https://www.mentor.com/pcb/hyperlynx/power-integrity/.

[16] Mentor Graphics, "HyperLynx Design Rule Checking," February 2020. https://www.mentor.com/pcb/hyperlynx/electrical-rule-check/.

[17] IEEE SA, "802.3-2018 - IEEE Standard for Ethernet," February 15, 2020. https://standards.ieee.org/standard/802_3-2018.html.

[18] Bel Magnetic Solutions, February 15, 2020. https://belfuse.com/resources/drawings/magneticsolutions/dr-mag-0826-1x1t-32-f.pdf.

[19] Murata, "Chip Multilayer Ceramic Capacitors," November 27, 2017. https://www.murata.com/~/media/webrenewal/support/library/catalog/products/capacitor/mlcc/c02e.ashx?la=en-us.

[20] International Electrotechnical Commission (IEC), "IEC 61508:2010 CMV," March 28, 2020. https://webstore.iec.ch/publication/22273.

[21] PCI SIG, "PCI Express 4.0 Electrical Previews," 2014. https://pcisig.com/sites/default/files/files/PCI_Express_4_0_Electrical_Previews.pdf.

[22] Bogatin, E., *Signal and Power Integrity: Simplified*, Upper Saddle River, NJ: Prentice Hall, 2009.

[23] Holden, H., et. al, The HDI Handbook, Seaside, OR: BR Publishing, 2009, https://www.hdihandbook.com/.

[24] Bogatin, E., and L. D. Smith, *Principles of Power Integrity for PDN Design—Simplified: Robust and Cost Effective Design for High Speed Digital Products*, Upper Saddle River, NJ: Prentice Hall, 2017.

[25] Graham, M., and H. Johnson, *High Speed Digital Design: A Handbook of Black Magic*, Upper Saddle River, NJ: Prentice Hall, 1993.

[26] Johnson, H., *High Speed Signal Propagation: Advanced Black Magic*, Upper Saddle River, NJ: Prentice Hall, 2003.

[27] Cooper, R., *Winning at New Products: Creating Value Through Innovation*, 3rd ed., New York: Basic Books, 2011.

[29] Sparx Systems Pty Ltd., "Enterprise Architect 14," August 22, 2019. https://www.sparxsystems.com/products/ea/14/index.html.

[29] Reliable Software Technologies, "Mew Models for Test Development," 1999. http://www.exampler.com/testing-com/writings/new-models.pdf.

[30] Forsberg, K., and H. Mooz, "The Relationship of Systems Engineering to the Project Cycle," *Joint Conference of the National Council on Systems Engineering (NCOSE) and American Society for Engineering Management (ASEM)*, Chattanooga, TN, October 21–23, 1991. https://web.archive.org/web/20090227123750/http://www.csm.com/repository/model/rep/o/pdf/Relationship%20of%20SE%20to%20Proj%20Cycle.pdf.

[31] INCOSE, "About: INCOSE," https://www.incose.org/about-incose.

[32] Requirements Working Group, International Council on Systems Engineering (INCOSE), "Guide for Writing Requirements," San Diego, CA, 2012.

[33] International Electrotechnical Commission (IEC), "Available Basic EMC Publications," August 22, 2019. https://www.iec.ch/emc/basic_emc/basic_emc_immunity.htm.

[34] RoHS Guide, "RoHS Guide," August 22, 2019. https://www.rohsguide.com/.

[35] European Commission, "REACH," August 22, 2019. https://ec.europa.eu/environment/chemicals/reach/reach_en.htm.

[36] Ericcson Inc., August 22, 2019. https://telecom-info.telcordia.com/site-cgi/ido/docs.cgi?ID=SEARCH&DOCUMENT=SR-332&.

[37] BKCASE Editorial Board, "The Guide to the Systems Engineering Body of Knowledge (SEBoK), v. 2.0," Hoboken.

[38] SEBoK Authors, "Scope of the SEBoK," August 18, 2019. https://www.sebokwiki.org/w/index.php?title=Scope_of_the_SEBoK&oldid=55750.

[39] SEBoK Authors, "Product Systems Engineering Background," August 18, 2019. https://www.sebokwiki.org/w/index.php?title=Product_Systems_Engineering_Background&oldid=55805.

[40] SEBoK Authors, "System Design," August 18, 2019. https://www.sebokwiki.org/w/index.php?title=System_Design&oldid=55827.

[41] SEBoK Authors, "System Analysis," August 18, 2019. https://www.sebokwiki.org/w/index.php?title=System_Analysis&oldid=55956.

[42] SEBoK Authors, "Logical Architecture Model Development," August 18, 2019. https://www.sebokwiki.org/w/index.php?title=Logical_Architecture_Model_Development&oldid=55698.

[43] SEBoK Authors, "Physical Architecture Model Development," August 18, 2019. https://www.sebokwiki.org/w/index.php?title=Physical_Architecture_Model_Development&oldid=55874.

[44] IEEE, "1149.7-2009 - IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture," February 10, 2010. https://ieeexplore.ieee.org/servlet/opac?punumber=5412864.

[45] International Organization for Standardization, "ISO 13849-1:2015," August 22, 2019. https://www.iso.org/standard/69883.html.

[46] International Electrotechnical Commission, "Functional Safety and IEC 61508," August 22, 2019. https://www.iec.ch/functionalsafety/.

[47] International Organization for Standardization, "ISO 26262-1:2018," August 22, 2019. https://www.iso.org/standard/68383.html.

[48] McLean, H. W., *HALT, HASS, and HASA Explained: Accelerated Reliability Techniques, Revised Edition*, Milwaukee, WI: American Society for Quality, Quality Press, 2009.

CHAPTER 3

# FPGA Design Considerations

## 3.1 Introduction

At SensorsThink, the development of the Smart Sensor SoC is making significant progress: the requirements have been agreed upon and baselined. The architecture of the solution has been defined, along with the subsystem allocation of functions between hardware, software, and firmware. Crucially, the hardware development has also begun creating the schematic and layout of the PCB. To ensure that SensorsThink have something to test on the hardware once the PCB is manufactured, the development of the FPGA and software needs to be started.

To ensure a high-quality FPGA design solution, SensorsThink FPGA engineers follow a comprehensive FPGA development process, which is aimed at managing the life cycle of the FPGA development from concept to delivery of the completed bit stream.

To create the FPGA application, engineers at SensorsThink face several distinct challenges:

1. Achieving the processing deadlines for sensor sampling;
2. Implementing algorithms that convert from raw sensor data to data values that are in the correct format, for example, resistance to temperature conversion;
3. Implementing algorithms that enable system noise to be removed, for example, filtering;
4. Implementing control structures to correctly interface with sensors and to configure the sensor for the required operation and read back the correct sensor data format;
5. Achieve the performance required across the operating environment.

These distinct challenges mean that engineers at SensorsThink need to consider a wide range of techniques that can be deployed in an FPGA in order to achieve these requirements. These techniques include:

1. Development flow and architectural implementation, including clock domains and clock domain crossing.

2. Development of control structures using state machines, along with considering defensive state machine design if the required operating environment demands it.
3. Implementation of algorithms requires engineers at SensorsThink to understand how math can be used in programmable logic. This also includes understanding different techniques that may present themselves for achieving complex algorithms (e.g., polynomial approximation, and COordinate Rotation DIgital Computer (CORDIC) algorithms).
4. Understanding how mixed signal converters work, as several of the sensors utilize the built-in Xilinx analog to digital converter (XADC) in the Zynq.
5. Understanding the verification of each module developed to ensure that its performance and functionality are correct prior to implementation within the target device once the board is made available for testing.
6. Understanding how digital filters can be implemented within the programmable logic, along with understanding how engineers can use the frequency domain and Fourier transforms to demonstrate the performance of filters, analog-to-digital converter (ADC), and digital-to-analog converter (DAC).
7. Consider the implementation of algorithms in higher-level languages such as C or C++. Implementing algorithms in a higher-level language can significantly reduce the development time.

In this chapter, we will explore these techniques in detail, thus enabling the SensorsThink engineering team to correctly architect and implement the FPGA solution.

## 3.2 FPGA Development Process

The development process for an FPGA follows elements of both hardware and software development.

The FPGA development flow starts in the systems engineering phase, when the subsystem segmentation allocates specific requirements to be implemented within the FPGA. These requirements are further expanded upon with the FPGA requirements, which includes not only the system-allocated requirements, but also derived requirements that are required for the FPGA. These derived requirements will include such requirements as FPGA technology (SRAM, flash, or One Time Programmable), operating temperature, operating frequency, and fixed-point performance requirements.

Once the requirements have been agreed upon, the next stage is the architectural design. During the architectural design, the architecture and hierarchy of the design are outlined, reviewed, and agreed upon. This architecture will identify the following:

- *Clock architecture:* The number of different clock domains and clock domain crossing approach.
- *Functional architecture:* The number of modules required to implement the overall requirement set.

- *Reset architecture:* Definition of the different reset domains and applicability to functions.
- *Test philosophy:* The verification strategy and tool chains used to demonstrate the modules and overall design, which achieves the requirements and performance targets.

Following sign-off of the architecture, the next stage of the development process is detailed design and verification; this will include:

- *Development functional modules:* This may use hardware descriptions languages (HDL) such as Verilog and VHDL. Alternatively, high-level languages may be used, such as C or C++, or proprietary tools such as MATLAB and LabVIEW.
- *Static code analysis:* Static analysis of the code enables nonfunctional issues to be detected and corrected. Static analysis therefore enables not only the enforcement of company naming standard, but also the enforcement of company/industry coding standards. Static analysis will also analyze the module to identify any dangerous or ambiguous coding structures that may lead to issues later in the development process. The goal of static code analysis is to start later stages of development with a higher quality of code, reducing the iteration time through verification and implementation.
- *Functional verification of modules:* Creation of test benches to verify the performance of the module under test. This may use industry standard frameworks, such as Universal Verification Methodology (UVM), Open Source VHDL Verification Methodology (OSVVM), or Universal VHDL Verification Methodology (UVVM). Depending upon the application, the verification may have targets for code coverage in the unit under test to ensure the module is sufficiently exercised.
- *Top-level design:* The assembly and interconnection of all modules within the design to complete the overall design.
- *Top-level verification:* The development of a test harness to check the top level of the design achieves the desired functionality. Depending upon the end application, the top-level verification may range from simply ensuring the correct connectivity of the modules to fully testing the functionality of the entire design.
- *Constraints generation:* Constraints are used by the FPGA implementation tool to guide the synthesis, placement, and routing stages. Commonly used constraints include:
  - *Input/output constraints:* Define the location of the signal on the FPGA pins and the logic standard to be used for that signal (e.g., LVCMOS1v8, LVCMOS2v5 for single-ended signals, or LVDS for differential signals). The IO constraints also can control the output drive strength, on-chip termination, slew rate, and pull-up or pull-down.
  - *Clocking constraints:* Defines the clock frequencies and interclocking and intraclocking relationships. The clocking constraints will also include definition of the input and output delays. Correctly defined timing

constraints will help to achieve the required performance across the operating conditions. Incorrectly defined timing constraints may result in the device not working as expected across the operating conditions. It may also significantly increase the implementation time as the implementation tool searches for a solution that is not realistic or achievable.

- *Placement constraints:* Define the physical location of logic elements/modules or groups of modules within the FPGA. Placement constraints can be used to help the tool achieve the timing performance and provide isolation between functional modules for security or safety.

- *Synthesis:* Synthesis translates the HDL written at a register transfer level into the logic resources, which are available in the targeted FPGA. To guide the synthesis tool, synthesis constraints may be used for timing-driven synthesis.
- *Placement:* Places the synthesized logic resources within the FPGA logic elements.
- *Routing:* The final stage of the implementation. Routing attempts to connect all the placed logic elements in the FPGA together, while achieving the timing performance defined by the constraints.
- *Bit stream generation:* Once the implementation has been completed, the implemented design needs to be converted into a format that can be downloaded or used to program the FPGA.

### 3.2.1 Introduction to the Target Device

The device selected in Chapter 2 to implement the FPGA design in is a Xilinx Zynq-7000 series device. This device is a heterogeneous SoC as it contains both a processing system and programmable logic. Within the processing system are dual Arm Cortex-A9 32-bit application processors capable of operation at up to 1 GHz. The programmable logic is based on the Xilinx 7 series logic and is interconnected with the programmable logic using several high bandwidth Arm eXtensible Interface Memory Mapped (AXI-MM) interfaces.

AXI is part of the Arm Advanced Microcontroller Bus Architecture (AMBA) tailored for implementation in FPGA and programmable logic devices. AXI provides three different interface standards, which provide for different performance and implementation requirements.

- *AXI Memory Mapped:* A full-duplex memory mapped interface that is capable of burst accesses to read and write memory locations. Both read and write channels are independent. AXI Memory Mapped is used for high bandwidth transfer to and from memory, either directly connected to the FPGA or within the processing system (e.g., DDR).
- *AXI Lite:* A simplified version of AXI Memory Mapped protocol that does not offer burst access. As such, it is intended for use in IP configuration, control, and status reporting.
- *AXI Stream:* A unidirectional point-to-point stream of data, intended for high bandwidth data transfer between IP within the programmable logic.

Architecturally, it is like the write channel of the AXI Memory Mapped interface.

AXI is a point-to-point interface, which always communicates between a master and a slave. Should we wish to connect a single master to several slaves, a crossbar switch is required to be implemented.

Within the processing system itself there are several peripherals commonly found within an embedded processing solution, for example, universal asynchronous receiver-transmitter (UART), controller area network (CAN), serial peripheral interrupt (SPI), I2C, and memory controllers. These peripherals are also interconnected using AXI buses; as such, it is possible if desired for the programmable logic to access peripherals within the processor system.

### 3.2.2 FPGA Requirements

At SensorsThink, we are developing a sensor platform that uses a programmable SoC. As such, we need to define the requirements for the programmable logic half of the design. These requirements need to be clear, concise, and unambiguous.

The FPGA will therefore be responsible for interfacing and processing with the following:

- FLIR Lepton 3.5: interface with the FPGA using I2C and SPI interfaces;
- LSM303AHTR Magnetometer Sensor: interfaces with the FPGA using a I2C interface;
- SHTW2 Temperature Sensor: interfaces with the FPGA using an I2C interface;
- ADCMXL3021 Vibration Sensor: interfaces with the FPGA using an SPI;
- LSM6DS3USTR Accelerometer Sensor: interfaces with the FPGA using an I2C sensor;
- HX94 Remote Humidity and Temperature Sensor: interfaces with the FPGA using XADC.

The FPGA is also able to help the processor expand its limited multiplexed IO (MIO). The processor has 54 MIO pins that can be implemented in interfaces such as I2C, SPI, Gigabit Ethernet (GigE), and Universal Serial Bus (USB), as shown in Figure 3.1. However, in some applications, the processor requires more MIO than are available. In this instance, for many of the low-speed interfaces, the peripheral signals can be routed to the programmable logic. This enables the PS interfaces to use the programmable logic IO for interfacing; this is called extended MIO (EMIO) as the MIO are extended into the programmable logic. In the SensorThink solution, EMIO is used to route out PL IO pins to support the Wi-Fi and Bluetooth interfaces.

At the FPGA requirements level, the requirements must trace back to the higher-level system requirements. The FPGA requirements can be seen in detail at SensorsThink.com.

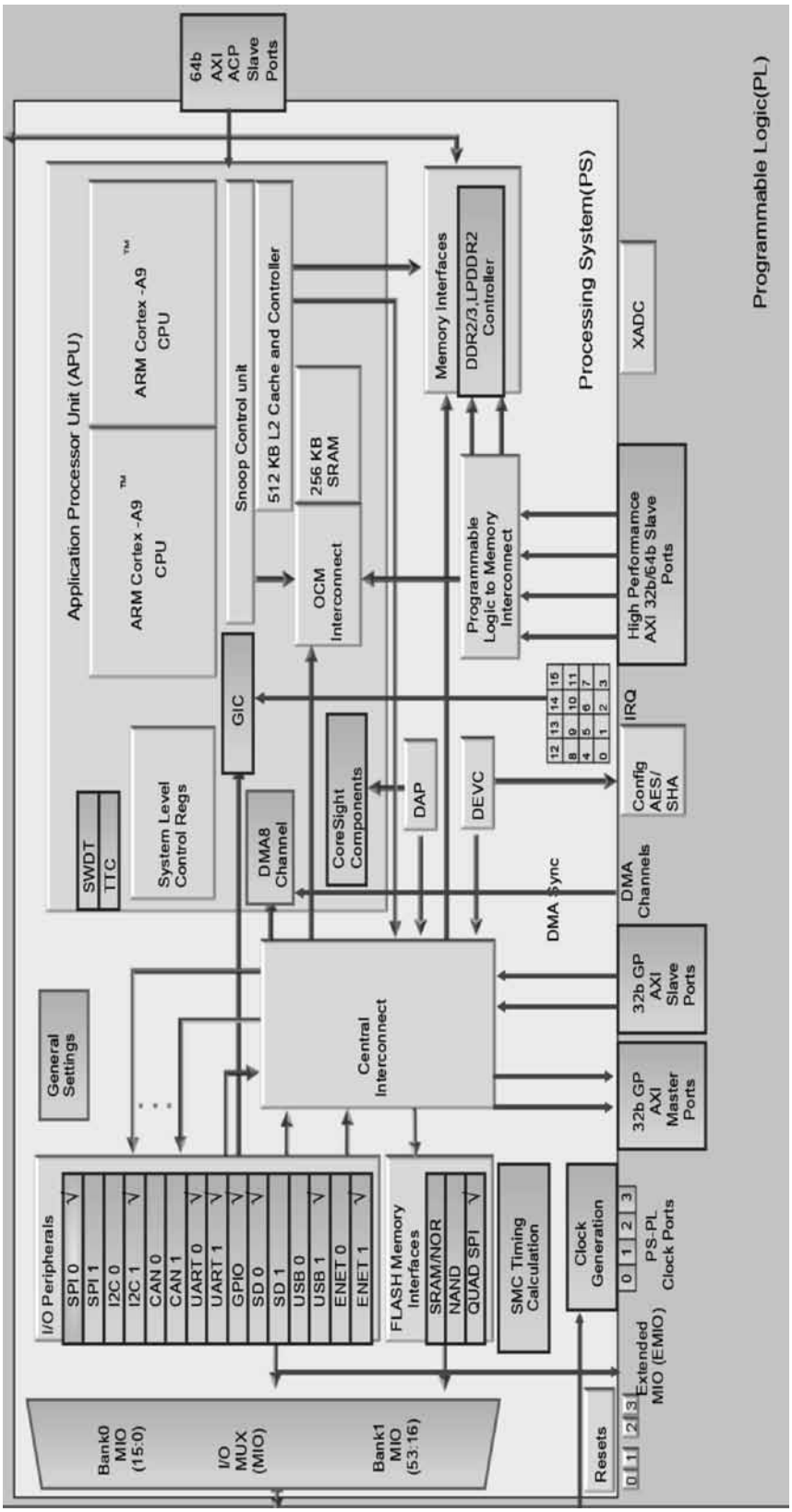



**Figure 3.1** Xilinx Zynq block diagram.

- There must be the ability to read temperature, magnetometer, vibration, and accelerometer sensors concurrently.
- Sensor data shall be tagged with a time stamp.
- The availability of new sensor data shall be indicated to the processor using an interrupt from the logic to the processor.
- The gathering of sensor data shall be initiated from the processor system.
- Frames from the FLIR Lepton 3.5 shall be read out at the maximum frame rate of 9 Hz.
- New FLIR image frames shall be indicated to the processor using an interrupt.

These requirements drive the overall architecture of the solution between the PS and the PL.

### 3.2.3 FPGA Architecture

The architecture of the FPGA design will implement several IP blocks that are used to interface with the sensors. Control functions will be used to gather the data and indicate to the processor that new data is ready.

The architecture of the FPGA solution can be considered in two distinct elements as shown in Figure 3.2: the sensor interfacing and the FLIR interfacing. Starting with the sensor interfacing to enable each sensor to be read in parallel, several I2C and SPI interfaces will be used. Each of these blocks will be under the control of a specific controller block. This controller block will consist of a state machine that will send the appropriate commands to the I2C or SPI interface. This will enable each of the sensors to be read in parallel. As each of the sequences are in parallel, this will allow the sensor controllers to wait for the sensor data to be available.

Starting the conversion process will be the responsibility of the controller and scheduling block that receives the cycle start signal from the processing system. The control and scheduling block also receive the sensor data when it is ready from

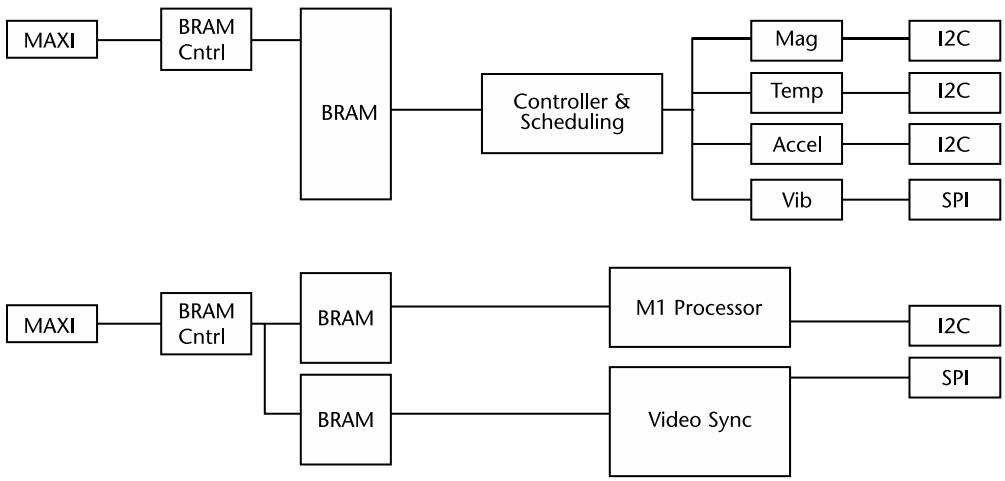



**Figure 3.2** FPGA architecture.

each of the dedicated sensor controllers. Once the sensor data from all four of the sensors has been received, the words are bundled together and written into the BRAM and an interrupt is issued to the processing system. The processing system can then, on receipt of this interrupt, read the BRAM via the AXI BRAM controller IP. Once in the processing system, the software is then able to process and work on the data.

Communicating with the controller and scheduling block using the BRAM enables the processing system to define several parameters used in the control and scheduling block, for example, the time between sampling cycles.

The FLIR input path is like the sensor path; however, the FLIR imager, once configured, will be free-running outputting frames at 9 Hz. The FLIR Lepton is configured over an I2C link; as such, in this application, the M1 processor is used connected to an I2C interface. The M1 processor can configure the FLIR Lepton under the control of the processing system, for example, changing the pixel output from grayscale to red, green, blue (RGB) false color or enabling the auto gain control. The M1 processor is operating from a program stored within BRAM; as this memory is dual-ported, updates to the M1 program can be executed by the processing system.

Video data output from the FLIR Lepton uses a video over SPI format. This means that packets of data are sent over the master in, serial out (MISO) port, when the master provides the serial clock (SCLK) and asserts chip select. In the video over SPI format, the master out, serial in (MOSI) port is not used.

The video packets output by the Lepton are either valid packets or invalid packets to comply with export regulations regarding frame rate. Once a valid packet is output with a valid line number, each of the line numbers increment until the entire frame has been read out. If the packet header indicates that the frame is incorrect, it needs to read out the entire corrupted frame and discard it.

As the frame rate is so low that a custom register-transfer level (RTL) block will be created that checks the frame header to determine if the header is valid or not. If the header is valid, the frame data will be written to block RAM, until the entire frame has been read out and written to BRAM. Once this has been completed, an interrupt will be generated to inform the processor a new image is available.

The processor can then access the image from the BRAM using the block random access memory (RAM) controller over the maser AXI interface.

Now that we understand the architecture, the next step is to start creating and verifying the RTL.

## 3.3 Accelerating Design Using IP Libraries

Before the SensorsThink engineering team begins to start RTL development, they should consider the use of existing intellectual property blocks, often called IP blocks. These IP blocks are provided by device vendors, third-party commercial IP vendors, and open-source initiatives (e.g., open cores). IP blocks fall within two groupings. The first group is those that are created to implement commonly used functions, for example, block RAM memories, first in, first out (FIFOs), and clocking structures. These basic IP building blocks are universal across a diverse range of FPGA developments and can be used to accelerate the design capture process

by providing IP blocks that enable the designer to focus on creating the value-added element of the solution, while the second group is domain-specific IP blocks. Domain-specific IP blocks focus upon a specific domain and implementation of functions commonly used within that domain. Example domains include quantitative finance, machine learning, and embedded vision. Domain-specific IP blocks are often significantly more complex than basic building blocks, for example, implementing algorithms such as color space conversion, video layering for embedded vision applications, or rate setting for quantitative finance. Using IP blocks provides a significant time-scale reduction as it reduces the number of custom blocks that need to be developed and verified. Many IP blocks also use the AXI standard outlined above that further reinforces the benefit of using a standard interface, as easy integration with existing IP blocks is provided.

## 3.4 Pin Planning and Constraints

The inclusion of a programable logic device on our Smart Sensor SoC means that we need to carefully consider the pin allocations. Programmable logic offers complete flexibility in assigning pins to signals; however, to get the optimal solution, we need to undertake careful pin planning. This pin planning is required to ensure that:

- Correct clocking resources are used: Xilinx Seven series provide both global and regional clocks.
- Correct pins are allocated for differential pairs.
- Correct IO standard is allocated to an IO bank.
- All IO standards used on a bank are compatible.

On a heterogeneous SoC such as the Zynq devices, not only do we have to consider the pin allocation for the programmable logic element of the device, but we also need to ensure that the MIO and DDR pin allocation is correct for the processor system.

Looking first at the processor system MIO, we need to create a design within the Vivado IP integrator, which targets the exact Zynq that will be placed on the board (xc7z020clg400-1). Once this project is created, we can create a new IP integrator design that contains a Zynq processing system, as shown in Figure 3.3.

Recustomizing the Zynq processor system allows us to implement the configuration of the PS MIO. Using the peripheral IO view, the MIO pin allocation to processing system peripherals can take place (Figure 3.4).

This visual view enables the MIO to be allocated as desired to the required processing system peripherals without the conflicts as design rule checking is live. Should the designer need to change any of the IO standards on the MIO pins, using the MIO configuration tab, the design can modify the IO speed and enable pull-up/pull-down on the signals (Figure 3.5). This can help greatly when the signal integrity is being analyzed.

While the DDR timing can be configured using the DDR configuration tab, this tab is where we define DDR type, speed, and arrangement (Figure 3.6).

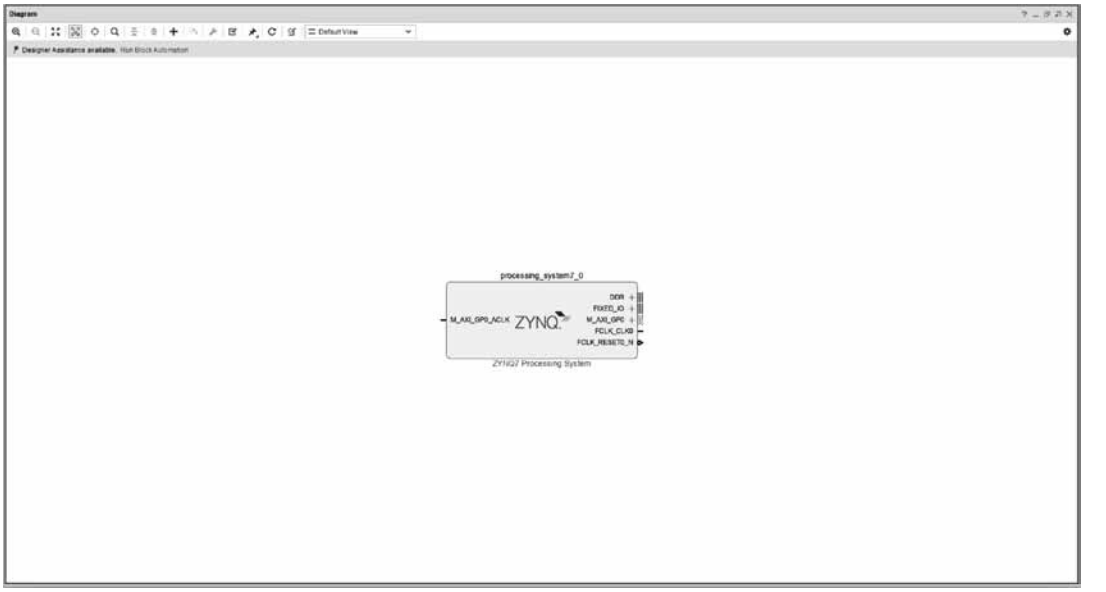



**Figure 3.3** Zynq processing system block diagram in an IP integrator.

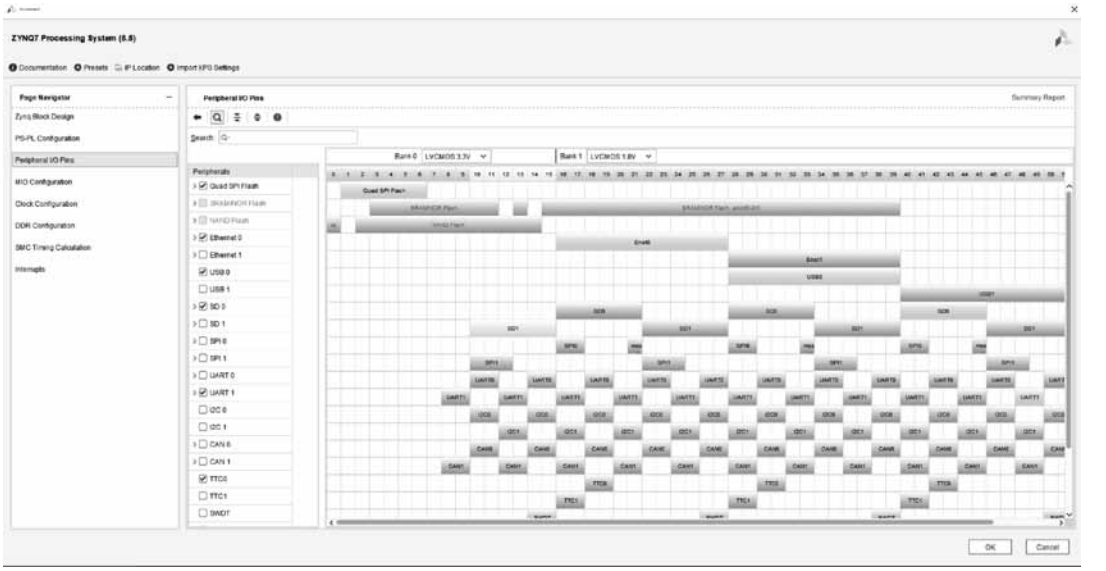



**Figure 3.4** Zynq processor system MIO allocation.

Once the MIO and DDR entries have been created, we are able to elaborate the design. From an elaborated design, we can export an IBIS file that describes the MIO and DDR interfaces. This can be critical for verifying that high-speed processor interfaces in the PCB layout have the correct signal integrity. As this can impact the design of the board, it is best to generate the IBIS file early on in the design process (Figure 3.7).[1]

1. The use of these IBIS files for signal integrity performance is discussed in detail in Chapter 2.

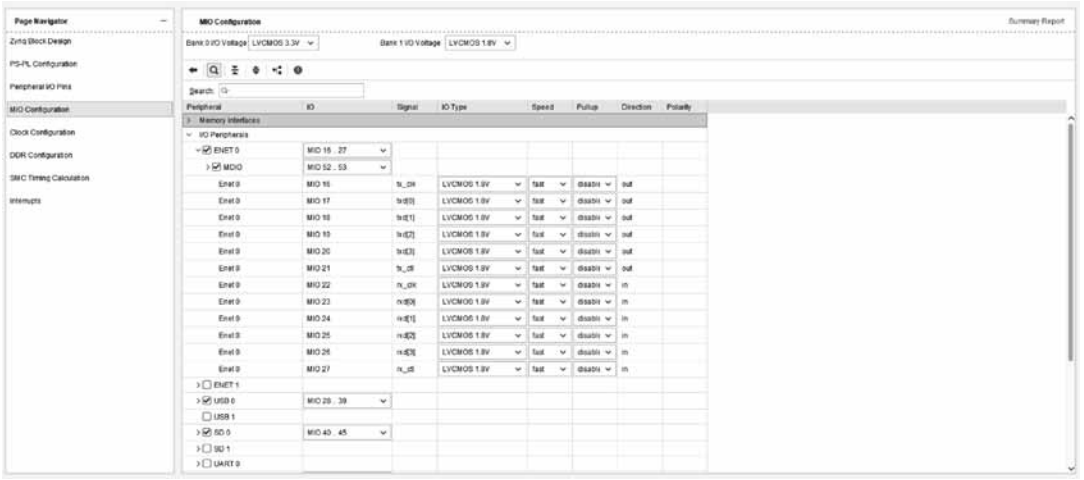


**Figure 3.5** Controlling the PS MIO drive strength.

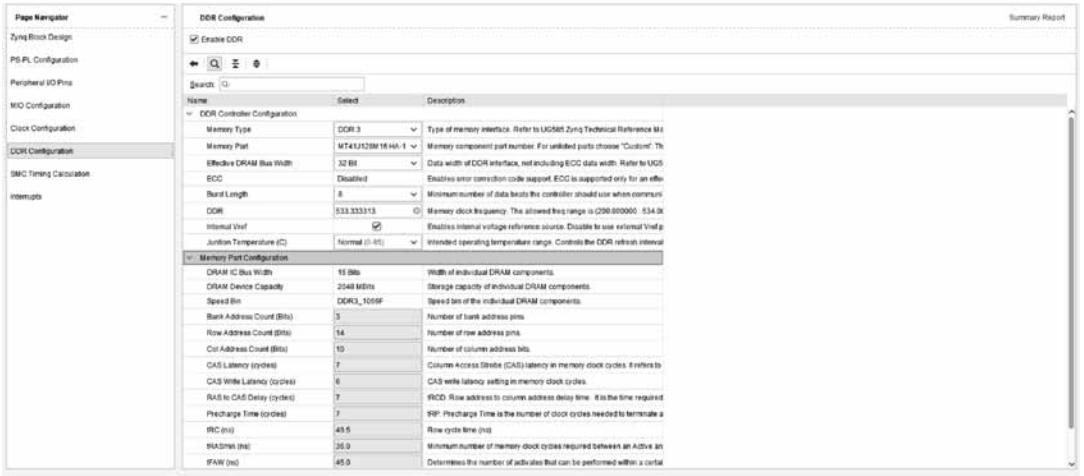


**Figure 3.6** Zynq PS DDR interfacing.

With the processor element of the pin allocation completed, we can create a pin planning project to verify the pins allocated to the programmable logic half of the Zynq.

Here we can allocate a pin to an FPGA signal and IO standard and control the speed and drive strength and termination. Once all the pins have been allocated, the developer can run a design rule check (DRC) to ensure that the FPGA pin allocation does not break the IO banking rules. Once complete, the pin out must be verified to be valid by passing the DRC.

These pins can be written out then as Xilinx Design Constraints File, which is how the Vivado understands which pin and IO standard to use for specific designs input and output.

```
set_property PACKAGE_PIN C20[get_ports HX94B_TEMP_BUF]
set_property PACKAGE_PIN E17[get_ports HX94B_TRH_BUF]
```

```
[Component]      ZYNQ
[Manufacturer]   Xilinx Inc.
[Package]
|CLG400
|variable        typ            min            max
R_pkg            409.3m         130.9m         730.5m
L_pkg            8.10nH         2.71nH         14.76nH
C_pkg            1.31pF         0.33pF         2.90pF
|
[Pin]    signal_name                          model_name  R_pin  L_pin  C_pin
A1       DDR_dm[0]                              SSTL135_DCI_F_PSDDR_MS
A2       DDR_dq[2]                              SSTL135_DCI_F_PSDDR_MS
A3       VCCO_DDR_502                           POWER
A4       DDR_dq[3]                              SSTL135_DCI_F_PSDDR_MS
A5       FIXED_IO_mio[6]                        LVCMOS33_S_8_PSMIO
A6       FIXED_IO_mio[5]                        LVCMOS33_S_8_PSMIO
A7       FIXED_IO_mio[1]                        LVCMOS33_S_8_PSMIO
A8       GND                                    GND
A9       FIXED_IO_mio[43]                       LVCMOS18_S_8_PSMIO
A10      FIXED_IO_mio[37]                       LVCMOS18_S_8_PSMIO
A11      FIXED_IO_mio[36]                       LVCMOS18_S_8_PSMIO
A12      FIXED_IO_mio[34]                       LVCMOS18_S_8_PSMIO
A13      VCCO_MIO1_501                          POWER
A14      FIXED_IO_mio[32]                       LVCMOS18_S_8_PSMIO
A15      FIXED_IO_mio[26]                       NC
A16      FIXED_IO_mio[24]                       NC
```

**Figure 3.7** IBIS file extract.

```
set_property PACKAGE_PIN F17[get_ports IO_L6N_35]
set_property PACKAGE_PIN G15[get_ports IO_L19N_35]
set_property PACKAGE_PIN K17[get_ports IO_L12P_35]
set_property PACKAGE_PIN M19[get_ports LTM4644_TEMP_BUF]
set_property PACKAGE_PIN N18[get_ports ATWILC_SPI_SCK]
```

Constraints are particularly important in FPGA design for not only ensuring external interfacing is defined, but also a range of other applications ranging from pin allocation to controlling timing performance and even location of logic in the programmable logic implementation. Now SensorsThink engineers have understood how we can pin plan our FPGA design (Figure 3.8); let us take a few moments to understand the wider role of constraints in our programmable logic development.

### 3.4.1 Physical Constraints

The most used physical constraints by an engineer are the placement of IO pins and the definition of parameters associated with the IO pin, for instance, standard and drive strength. However, the team at SensorsThink would be well served to understand and consider the following types of constraints.

- *Placement constraints:* These allow us to define the location of cells.
- *Routing constraints:* These allow us to define the routing of signals.
- *IO constraints:* These allow us to define the location and IO parameters.
- *Configuration constraints:* These allow the configuration methods to be defined.

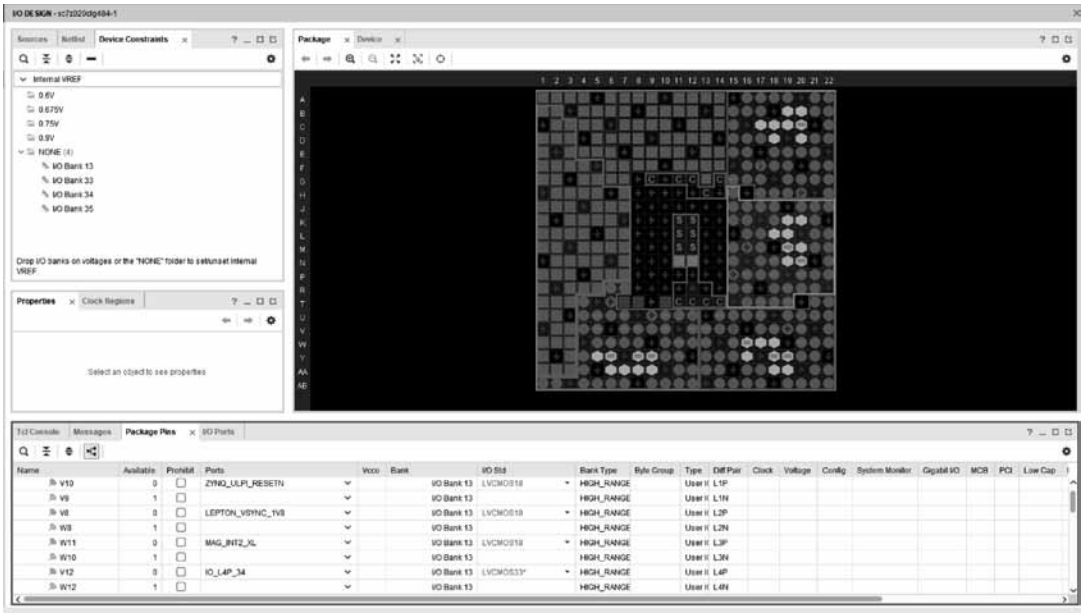


**Figure 3.8** Zynq project planning view.

As always, there are a few constraints which sit outside these groups. Vivado has three, and they are predominantly used on the netlist.

- *DONT_TOUCH:* This is used to prevent optimization; as such, it can be of great use when implementing a safety-critical/high-reliability system.
- *MARK_DEBUG:* This is used to preserve an RTL net such that it can be used for debugging later.
- *CLOCK_DEDICATED_ROUTE:* Identifies a route for the clock routing.

Modern FPGAs support several different single and differential IO standards; these are defined via the IO constraints. However, we must take care to ensure that we are following IO banking rules, which depends upon the final pin placement.

What are IO banking rules? Each of the user IOs within an FPGA is grouped together into a number of banks consisting of a number of IOs. These banks have independent voltage supplies enabling the wide range of IO standards to be supported. On the Zynq and other seven series devices, IO banks are further classified into belonging to one of two overall groups, which further constrains their performance and requires that the engineer use the correct class for the correct interface. The first is high performance (HP) optimized for higher data rates; as such, this uses lower operating voltages and does not support LVCMOS 3v3 and 2v5. The other class high range (HR) is optimized to support wider IO standards not supported by HP; as such, it supports traditional 3v3 and 2v5 interfacing. Figure 3.9 demonstrates these banks.

Once we have determined which banks will be used for which signal, we still can change the signal drive strength and slew rate. These will be of great interest to your hardware design team to ensure that the signal integrity upon the board is optimal. These selections will also affect the timing of the board design, as such a

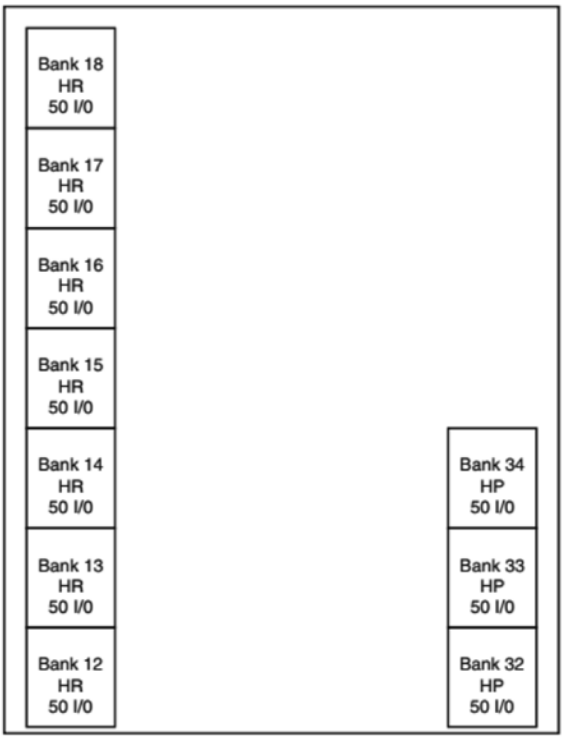



**Figure 3.9** HP and HR banks on a 7-series device.

signal integrity tool may be used; these tools require an IBIS model. We can extract an IBIS model of our design from Vivado when we have the implemented design open. Using the File->Export->Export IBIS model option, this file can then be used to close the system level signal integrity issues and timing analysis of the final PCB layout.

Once the design team is happy with the signal integrity performance and timing of the system as a whole, we end up with a number of constraints as below for the IOs in the design.

```
set_property PACKAGE_PIN G17 [get_ports {dout}]
set_property IOSTANDARD LVCMOS33 [get_ports {dout}]
set_property SLEW SLOW [get_ports {dout}]
set_property DRIVE 4 [get_ports {dout}]
```

With the HP IO banks, we can also use the digitally controlled impedance to terminate the IO correctly and increase the signal integrity of the system without the need for external termination schemes. We must also consider the effects of the IO if there is no signal driving it. For instance, if it is connected to an external connector, we can use the IO constraints to implement a pull-up or pull-down resistor to prevent the FPGA input signal from floating, which can cause system issues.

We can also use physical constraints to improve the timing of our design by implementing the final output flip-flop within the IO block itself; doing so reduces the clock to output timing. We can also do the same thing on input signals which allows the design to meet the pin-to-pin setup and hold timing requirements.

### 3.4.2 Timing Constraints

The next type of constraint for the development team at SensorsThink to consider is timing constraints. The most basic level of a timing constraint defines the operating frequency of the clock(s); however, more advanced constraints establish the relationships between clock paths. Why we do this is to determine if we need to analyze the path or if it can be discounted as there is no valid timing relationship between those clock paths. By default, Vivado will analyze all relationships; however, not all clocks within a design will have a timing relationship that can be accurately analyzed. One example of this is clocks that are asynchronous as their phase cannot be accurately determined, as shown in Figure 3.10.

We manage these relationships between clock paths using a constraints file and declaring clock groups. When a clock group is declared, Vivado performs no timing analysis in either direction between the clocks defined within it.

To aid the generation of our timing constraints, Vivado defines clocks as being within one of three categories:

- S*ynchronous clocks:* Synchronous clocks have a predictable timing/phase relationship; this is normally the case for a primary clock and integer-generated clocks, as they share a common root clock and will have a common period.
- *Asynchronous clocks:* Asynchronous clocks have no predictable timing/phase relationship between them; this is normally the case for different primary (and their generated) clocks or clocks generated with fractional division. Asynchronous clocks may or may not have a different root clock.
- *Unexpandable clocks:* Two clocks are unexpandable if over 1,000 cycles a common period cannot be determined. If this cannot be established, then the worst-case setup relationship over this, 1,000 cycles, will be used; however, there is no guarantee that it is the worst case.

To determine with which type of clock we are dealing with, we can use the clock report produced by Vivado to aid us in identifying asynchronous and unexpandable clocks.

With these identified, we can use the set clock group constraint to disable timing analysis between them. Vivado uses SDC-based constraints; as such, we can use the command below to define a clock group:

set_clock_groups –name –logically_exclusive –physically_exclusive –asynchronous –group

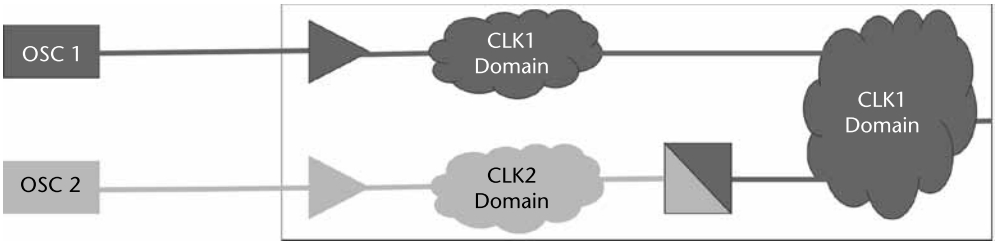



**Figure 3.10** Asynchronous clocks.

The –name is the name given to the group; the –group option is where one can define the members of the group (i.e., the clocks that have no timing relationship). The logically and physically exclusive options are used when we have multiple clock sources that are selected between to drive a clock tree (e.g., BUFGMUX or BUFGCTL); therefore, the clocks cannot be present upon the clock tree at the same time. As such, we do not want Vivado to analyze the relationship between these clocks, as they are mutually exclusive, while the –asynchronous is used to define asynchronous clock paths.

The final aspect of establishing the timing relationship is to consider the nonideal relationship of the clocks in particular jitter. Jitter needs to be considered in two forms: input jitter and system jitter. Input jitter is present upon the primary clock inputs and is the difference between when the transition occurs and when it should have occurred under ideal conditions, while system jitter results from noise existing within the design.

We can use the set_input_jitter constraint to define the jitter for each primary input clock, while the system jitter is set for the whole design (those are all the clocks) using the set_system_jitter constraint.

### 3.4.3 Timing Exceptions

We must also focus upon what happens within a defined clock group when we have an exception, but what is an exception?

One common example of a timing exception would be a result being captured only every other clock cycle, while another would be transferring data between a slow clock and a faster clock (or vice versa) where both clocks are synchronous. Both are a timing exception commonly referred to as a multicycle path as shown in Figure 3.11.

Declaring a multicycle path for these paths results in a more appropriate and less restrictive timing analysis allowing the timing engine to focus upon other more critical paths, increasing the quality of results.

Within our XDC file, we can declare a multicycle path using the XDC command:

```
set_multicycle_path path_multiplier [-setup|-hold] [-start|-end]
[-from <startpoints>] [-to <endpoints>] [-through
<pins|cells|nets>]
```

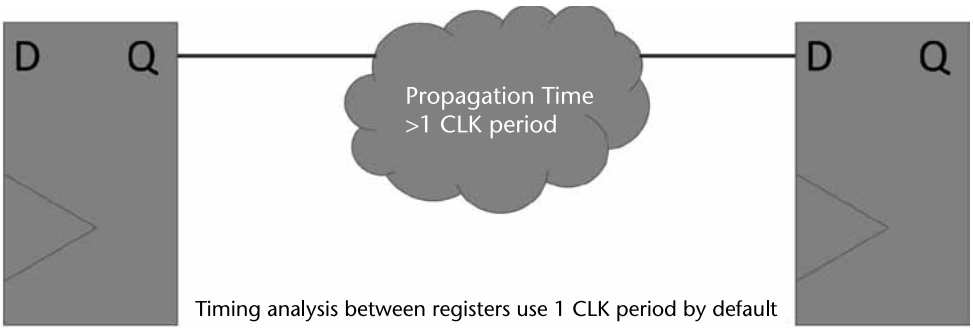



**Figure 3.11** Multicycle paths.

When we declare a multicycle path, we are in effect multiplying the requirements for the setup and hold (or both) analysis by the path_multiplier (e.g., in the first example above where the output occurs every 2 clock cycles, the path_multiplier would be 2 in the case of the setup timing).

As the multicycle path can be applied to either setup or hold, the engineer then has the choice of where to apply it. When we declare a setup multiplier, it is often best practice to also declare a hold time multiplier using the equation:

$$\text{Hold cycles} = \text{set up multiplier} - 1 - \text{Hold Multiplier}$$

What this means for the simple example that we have been following is that the hold multiplier is defined by the equation

Hold Multiplier = setup multiplier – 1 When using a common clock.

### 3.4.4 Physical Constraints: Placement

We may wish to constrain the physical placement for several reasons, perhaps to help to achieve timing or maybe to provide isolation between sections of the design. Before we delve too deeply, there are a few terms we need to define:

- *BEL:* The Basic Element of Logic (BEL) allows a netlist element to be placed within a slice.
- *LOC:* Location (LOC) places an element from the netlist to a location within a device.
- *PBlock (Physical Block):* This can be used to constrain logic blocks to a region of the FPGA.

Thus, while a LOC allows you to define a slice or other location within the device, a BEL constraint allows you to target at a finer granularity than the LOC and identify the flip-flop (FF) to use within the slice. PBlocks can be used to group logic together; they are also used for defining logical regions when we wish to perform partial reconfiguration.

The PBlock is great for when we wish to segment large areas of our design or if we wish to perform partial reconfiguration. In some instances, we will wish to group together smaller logic functions to ensure that the timing is optimal. While we could achieve this using PBlocks, it is more common for us to use relationally placed macros (RPMs).

RPMs allow design elements such as DSP, FF, lookup table (LUT), and RAMS to be grouped together in the placement. Unlike PBlocks, they do not constrain the location of these to a specific area of the device (unless you want it to) but instead RPMs group these elements together when they are placed.

Placing design elements close together allow us to achieve two things. It allows us to improve resource efficiency, and it means that we can fine-tune interconnection lengths to enable better timing performance.

To do this, we use three different types of constraints, which can be defined with the HDL source files:

- *U_SET:* Allows definition of an RPM set of cells regardless of hierarchy.
- *HU_SET:* Allows definition of an RPM set of cells with hierarchy.
- *RLOC:* Allows the assignment of relative locations to the SET.

The RLOC constraints use the definition RLOC = XmYm where the X and Y relate to the coordinates of the FPGA array. When we define an RLOC, we can make this either a relative or absolute coordinate. Depending upon if we add in the RPM_GRID attribute, this makes the definition absolute and not relative.

As these constraints are defined within the HDL, as in Figure 3.12, it is often necessary for a place and route iteration to be run initially before the constraints are added to the HDL file to correctly define the placement.

## 3.5 Clock Domain Crossing

As we start developing the application for our Smart Sensor SoC, engineers at SensorsThink need to be careful with the clocking architecture used within the programmable logic. Data is transferred between registers within the FPGA synchronously on a clock edge.

As we firm up the architecture of the programmable logic implementation, we may need to use several different clocks at different frequencies. The need for several clocks within the solution stems from the need to be able to interface with the sensor at specific clock frequencies. The four local sensors located on the board range in clock frequencies from 400 kHz to 14 MHz (Table 3.1).

The main clock to transfer data into the processing system is at a much higher rate, as it must be able to read multiple sensors in a short period of time. This means that sensor data must safely be transferred between the sensor clock and the processing system clock.

Transferring data between clocks can present several challenges to us as we develop the programmable logic solution. Failure to implement a safe transfer between clocks can result in a data corruption that could impact the performance of the system.

Transferring data between the clocks is often referred to as CDC; safely transferring data from one clock to another requires the designer to understand the concept of a clock domain.

```
SIGNAL ip_sync : STD_LOGIC_VECTOR(1 DOWNTO 0) :=(OTHERS =>'0');
SIGNAL shr_reg : STD_LOGIC_VECTOR(31 DOWNTO 0) :=(OTHERS =>'0');

ATTRIBUTE RLOC : STRING;
ATTRIBUTE HU_SET : STRING;
ATTRIBUTE HU_SET OF ip_sync : SIGNAL IS "ip_sync";
ATTRIBUTE HU_SET of shr_reg : SIGNAL IS "shr_reg";
```

**Figure 3.12** Constraints within the source code.

**Table 3.1** Sensor Clock Frequencies

| *Sensor* | *Maximum Clock Rate* |
|---|---|
| SHTW2 | 400 kHz |
| LSM6DS3 | 10 MHz |
| LSM303AH | 10 MHz |
| ADcmXL3021 | 14 MHz |

A clock domain is a grouping of clocks that are all related. Related means that the clocks are generated from the same source. A clock domain may include generated clocks from the source at different frequencies using a phase lock loop, multiclock manager, or traditional counter. Clocks generated in this way are related if the output clock is an integer multiple of the source clock. Related clocks therefore have a known and unchanging timing relationship between clock edges; this enables flip-flop setup and hold windows to be achieved. This means transferring data between clocks in the same clock domain does not require any special synchronization or handshaking (Figure 3.13).

Clocks are in a different clock domain if the source of the clock is different, even if the clock is the same frequency. Generated clocks are also in different clock domains if they are not an integer multiple of the source clock. This fractional relationship guarantees that the setup and hold windows of registers cannot be achieved (Figure 3.14).

Violating the setup and hold time of the registers results in registers going metastable; that is, the output of the flip-flop goes to a value between 0 and 1. Eventually, the flip-flop will recover, and the output will go to either a 0 or 1. This recovery time of the flip-flop and the random resulting output value mean that if we do not correctly synchronize CDC, our system can end up with corrupted data.

Multiclock designs therefore require additional analysis and design effort to ensure that CDC is safe and reliable.

Detecting CDC within a design can be complex; it is unlikely to be detected during RTL simulation, as this simulation does not include timing information for registers, whereas gate-level simulation that does include timing information requires considerable time and that the right question be asked (e.g., the unrelated clocks being timed exactly right to cause metastability). This is hard to create in a simulation but oddly easy to create on the deployed system in the field.

The engineers at SensorsThink would be well served to perform clock domain analysis; it is a critical analysis to be performed in multiclock domains. There are several ECAD tools available that can perform CDC analysis. For the development

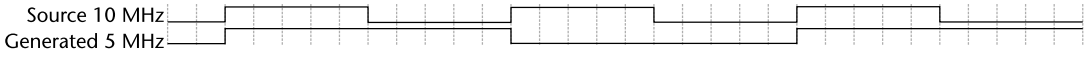



**Figure 3.13** Related clock: a 10-MHz source used to generate a 5-MHz clock.

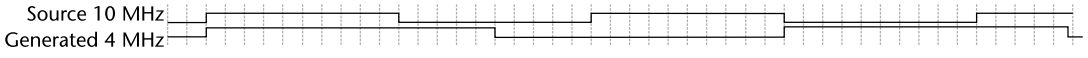



**Figure 3.14** Unrelated clock: a 10-MHz source used to generate a 4-MHz clock.

of the programmable logic design, engineers at SensorsThink use the Blue Pearl Software Visual Verification Suite (Figure 3.15). This provides us the ability for static code analysis and CDC analysis; Blue Pearl can analyze our design and identify not only the clock domains, but also signals that cross from one clock domain to the next.

Once the clock domain crossing signals have been identified, we can implement CDC structures to safely transfer data from one clock domain to another. Safely crossing clock domains requires the ability to resynchronize signals from one clock domain to another. The synchronization scheme used depends upon the type of signal being passed between clock domains.

- *Single static signal:* Transferring a single signal between clock domains can be implemented using a two-register synchronization structure.
- *Pulsed data:* Transferring pulsed data, especially from a faster clock domain to a slower clock domain, can present a challenge.
- *Data Bus:* Mux-based synchronizer.
- *Data Bus:* Handshake synchronizer.
- *Data Bus:* Asynchronous FIFO.
- *Data Bus:* Gray code.

We must also be careful to ensure that, when working with single logic signals, several single logic signals do not converge in a receiving clock domain (Figure 3.16), as the random delay through the synchronizer path can result in logic issues.

As we are implementing our design targeting a Xilinx programmable logic device, we should consider the specialist CDC structures provided. If we need to implement CDC structures within a Xilinx programmable logic, we can leverage the Xilinx Parameterized Macros (XPM), which provide several CDC structures.

- *Single-bit synchronizer:* Synchronizes a single bit vector between clocks.
- *Asynchronous reset synchronizer:* Synchronizes an asynchronous reset to the destination clock. Reset assertion is asynchronous, while the reset removal will always be synchronous and safe for the destination clock.
- *Gray code synchronizer:* Synchronizes a data bus between source and destination clocks, using Gray coding. Due to the implementation style, the input


**Figure 3.15** Clock domain identification using Blue Pearl (1 synchronized and 2 unsynchronized crossings).

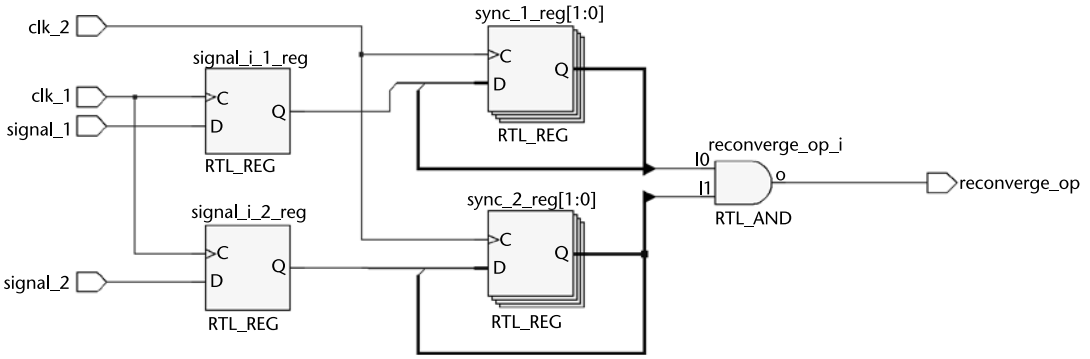



**Figure 3.16** Reconvergence of two standard logic vectors in a different clock domain.

can only increment or decrement by 1 between values, making it useful for counters.

- *Handshake synchronizer:* Synchronizes a data bus between source and destination clocks using handshaking of signals.
- *Pulse synchronizer:* Synchronizes a pulse of any width (one source clock or wider) from the source domain to the destination domain.
- *Single-bit synchronizer:* Synchronizes a single bit from source to destination domain.
- *Synchronous reset synchronizer:* Synchronizes a synchronous reset to the destination clock domain. This both asserts and de-asserts the reset synchronously, unlike the asynchronous reset macro.

Regardless of the implementation technology, when it comes to implementation within the device, we must ensure that synchronizing elements are placed close together to ensure the best timing performance.

## 3.6 Test Bench and Verification

As the SensorsThink FPGA design solution matures, the SensorsThink engineering team should be thinking about verification of both the FPGA and the individual models. The verification of a FPGA or RTL module can be a time-consuming process as the engineer ensures that the design will function correctly against its requirement specification and against corner cases that might make the module perform incorrectly. Engineers traditionally achieve this using a test bench; however, test benches can be simple or complicated affairs. Let us have a look at how we can get the most from our test bench while not overcomplicating it.

### 3.6.1 What Is Verification?

Verification ensures that the unit under test (UUT) meets both the requirements and specification and is suitable for its intended purpose. In many cases, verification is performed by a team independent to the design team. When considering FPGA

verification and test benches, we are ensuring that the UUT performs against its specification; with the size and complexity of modern FPGA designs, this can be a considerable task. The engineering team must therefore determine at the start of the project what the verification strategy will be; choices can include:

- *Functional simulation only:* This checks that the design is functionally correct.
- *Functional simulation and code coverage:* This checks that, along with the functional correctness of the design, all of the code within the design has been tested.
- *Gate-level simulation:* This verifies the functionality of the design when back-annotated with timing information from the final implemented design; this can take a considerable time to perform.
- *Static timing analysis:* This analyzes the final design to ensure that the timing performance of the module is achieved.
- *Formal equivalence checking:* This is used to check the equivalence of netlists against RTL files.

Whatever the verification strategy, the engineering team will need to produce a test plan that shows how the individual modules and the final FPGA are to be verified and how all requirements will be addressed.

Typically, to ensure that you can verify most requirements placed upon the FPGA, a test bench is required to stimulate the UUT, thus ensuring that the UUT performs as specified.

### 3.6.2 Self-Checking Test Benches

To ensure that the design functions as desired, we typically use a test bench to stimulate the UUT. This test bench can either be self-checking or not. If you opt for a nonself-checking test bench, then you will be required to visually confirm the test bench functions as desired. A self-checking test bench differs from a traditional test bench in that, along with applying the stimulus to the UUT, the outputs from the UUT are also checked against expected results as shown in Figure 3.17. This allows you to state that the UUT has passed or failed. Couple this with the control and reporting via using TEXTIO and you can create an immensely powerful verification tool.

The use of a self-checking test bench has the following advantages:

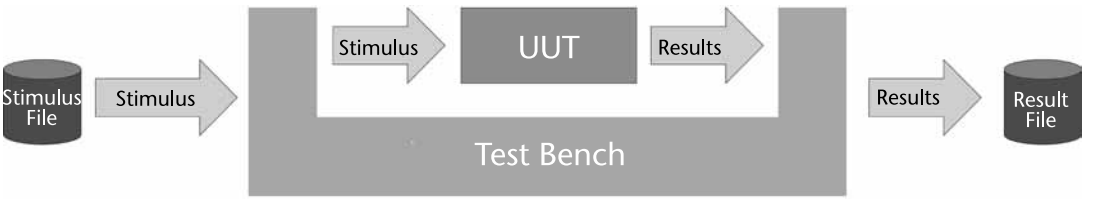



**Figure 3.17** Self-checking test bench architecture.

- Checking simulation waveforms can be very time-consuming, complicated to visually verify.
- Self-checking test benches can provide an overall pass or fail report that can be saved and used as test evidence later in the design flow.
- Should the UUT require design iterations at a later stage, rerunning the test bench and determining pass or fail can be achieved with a reduced timescale. This is often called regression testing.

### 3.6.3 Corner Cases, Boundary Conditions, and Stress Testing

When using a test bench, we want to ensure that the module performs as per the functional requirements, that any corner cases are addressed and most importantly that the test bench sufficiently exercises the UUT code. It is therefore common when verifying design modules to use glass box testing methodologies in that the contents of the UUT module are known and observed, whereas when the UUT is the FPGA top level, black box testing is used due to the increased time verification and simulation times at the top level.

To verify the functionality of the design against requirements, the test bench must apply stimulus as the modules expect to see in operation. However, testing all possible inputs to prove functional compliance can be a time-consuming and lengthy process; for this reason, it is often the case that the engineer will often focus upon boundary conditions and corner cases along with testing some typical operating values.

A boundary condition is when one of the inputs is at its extreme value, while a corner case is when all inputs are at their extreme value. A good example of this is illustrated by a simple 16-bit adder, which adds two numbers and produces a result. In this case, the boundary condition and corner cases are illustrated in Figure 3.18.

As you can clearly see, the corners cases would be as follows: when both A and B inputs equal 0, when A equals 0 and B equals 65,535, when A and B equal

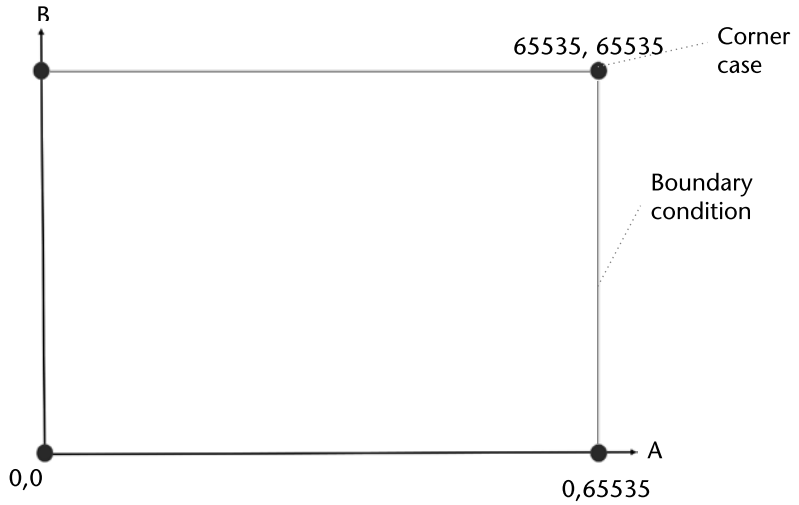



**Figure 3.18** Boundary condition and corner cases for a 16-bit adder.

65,535, and finally when A equals 65,535 and B equals 0. The boundary cases are the values between the corner cases.

In some applications, we may also want to stress test the UUT to ensure a margin above and beyond the normal operation. Stress testing will change depending upon the module, but it could involve repeated operation, attempting to overflow buffers and FIFOs. Stress testing a design allows the verification engineer to have a little fun trying to break the UUT.

### 3.6.4 Code Coverage

The verification team also requires a metric to ensure that the UUT is properly exercised by the test bench; this metric is typically provided by using code coverage to ensure that the UUT is correctly exercised. When looking at code coverage, there are several different parameters that the verification engineer should consider:

- Statements: Each executable statement is examined to determine how many times it is executed.
- Branch: All possible IF, CASE, and SELECT branches are executed.
- Conditions and subconditions within a code branch are tested to see what condition caused it to be true.
- Paths: All paths through the HDL are traversed to identify any untraveled paths.
- Triggering: Monitor signals in the sensitivity list of VHDL processes and wait statements.

Achieving 100% in the above parameters will not prove that the UUT is meeting functional requirements; however, it does easily identify sections of the UUT that have not been exercised by the test bench. It is especially important at this point to mention that code coverage only addresses what is within the UUT.

### 3.6.5 Test Functions and Procedures

The verification team should create a library of commonly used test routines that can be used by all members of the team and even across other projects. This will not only speed up the development, but it will also result in a standardized output from the test bench, which aids both analysis and verification of the module.

These standard functions and procedures can be used for numerous applications; however, a few commonly used methods are:

- *Stimulus generators:* For instance, a standard method of testing reset application and release or standard method of driving a communication interface.
- *Output checking:* Checking the output result against an expected result and reporting via the transcript or file.
- *Models:* Standardized models of devices with which the FPGA or module will interface during operation.

- *Logging functions:* Creating a standardized reporting format for the results; this enables easier analysis of the results. These also provide evidence of verification of requirements that can be used to demonstrate compliance.

Good output checking functions can make use of the VHDL signal attributes stable, delayed, last_value, and last_event. These attributes can be especially important when confirming that the UUT is compliant with the timing interfaces required for memories or other interfaces. These can also be useful for the ensuring that setup and hold times are achieved and reporting violations if they are not.

### 3.6.6 Behavioral Models

Sometimes the performance of a UUT can be verified against a nonsynthesizable behavioral model that implements the same function as the UUT as shown in Figure 3.19. The model and the UUT are then subjected to the same stimulus from the test bench; the outputs of the two modules are then compared on a cycle-by-cycle basis to ensure that both behave in the same manner. This may require that the latency of the UUT is accounted for by the model or test bench to allow comparison of the correct results.

### 3.6.7 Using Text IO Files

If you are using VHDL to verify your design, you can use the Text IO package to read in stimulus vectors from a text file and record the results including pass or fail within a results text file. The verification engineering team are then able to create several stimulus test files to test the UUT. This approach is beneficial as changing the stimulus becomes as simple as updating a text file; this allows for easy and quick updates as well to test scenarios that could be causing issues in the lab on the final hardware.

Within VHDL interaction with text files is supported by the std package, which if your simulator supports VHDL 2008, allows the use of std_logic and std_logic_vector. If your simulator does not yet support VHDL 2008, you can always use the nonstandard but commonly used std_logic_textio package to provide similar support.

Similar support is provided within Verilog to read and write text files.

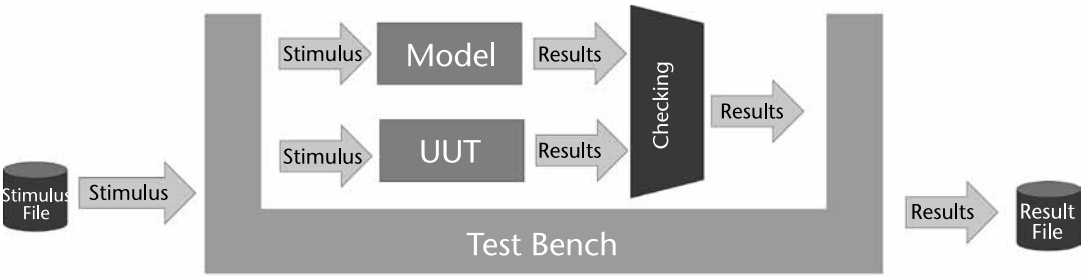



**Figure 3.19** Model checking against a UUT.

### 3.6.8 What Else Might We Consider?

The test bench should be time agnostic which means that it should be capable of testing either RTL or behavioral UUT along with post place and route netlists back annotated with SDF timing information. While simulating at a gate level can take considerable time due to the required resolution, at times, it is at times necessary.

The verification team should also consider the use of TCLscripts to control the simulation tool as it executes the test. This approach allows for a more repeatable process as all options that might be set within a graphical user interface (GUI) and change between simulation runs will be defined within the TCL file controlling the simulation. These files also become self-documenting reducing the documentation needed to support the verification of the UUT.

## 3.7 Finite State Machine Design

The SensorsThink engineering team is going to need to implement several control structures in the FPGA. These control structures will be used to configure and communicate with sensors, along with performing the overall sequencing of the programmable logic design. This means the SensorsThink engineering team will be required to implement several finite state machines in the design. Finite state machines are logical constructs that transition between a finite number of states. A state machine will only be in one state at a point in time; however, it will transition between states depending upon several triggers. Theoretically, state machines are divided into two basic classes, Moore and Mealy; they differ only in how the state machines outputs are generated.

- *Moore:* State machine outputs are a function of the present state only, a classic example is a counter.
- *Mealy:* State machine outputs are a function of the present state and inputs, a classic example of this is the Richards controller.

State machines are not only limited to implementation in programmable logic, but they can also be implemented in traditional discrete logic gates if required.

### 3.7.1 Defining a State Machine

When we are faced with the task of defining a state machine, the starting point is to develop a state diagram. The state diagram shows the states, the transitions between states, and the outputs from the state machine. Figure 3.20 shows the two state diagrams: one for a Moore state machine (left), the other a Mealy State Machine (right).

If we were to implement these using physical components (as we did before programmable logic was available), we would start generating the present state, next state tables and producing the logic required to implement the state machine. However, as we will be implementing this in a programmable logic, the engineer can work directly from the state transition diagram.

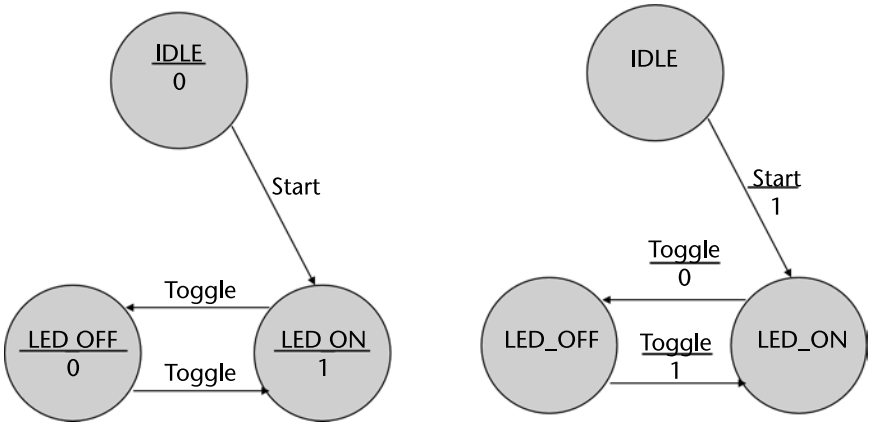



**Figure 3.20** FSM state diagrams Moore and Mealy.

### 3.7.2 Algorithmic State Diagrams

While many state machines are designed using the state diagram approach shown above, another method of describing a state machines behavior is the algorithmic state chart (see Figure 3.21). This algorithmic state machine ASM) chart is much closer in appearance to a software engineering flowchart. It consists of three basic constructs:

1. *State box:* This is associated with the state name and contains a list of the state outputs (Moore).
2. *Decision box:* This tests for a condition being true and allows the next state to be determined.
3. *Conditional output box:* This allows the state machine to describe Mealy outputs dependent upon the current state and inputs.

Some engineers at SensorsThink feel that a state machine described in ASM format is easier to map to implementation in a hardware description language such as VHDL.

### 3.7.3 Moore or Mealy: What Should I Choose?

The decision to implement either a Moore or Mealy machine will depend upon the function the state machine is required to implement along with any specified reaction times. The major difference between the two is in how the state machines react to inputs. A Moore machine will always have a delay of one clock cycle between an input and the appropriate output being set. This means that a Moore machine is incapable of reacting immediately to a change of input, as can be seen clearly in Figure 3.22. This ability of the Mealy machine to react immediately to the inputs can often mean that Mealy machines require fewer states to implement the same function as a Moore implementation would. However, the drawback of a Mealy machine is that when communicating with another state machine, there is the danger of race conditions due to the combinatorial nature of the outputs.

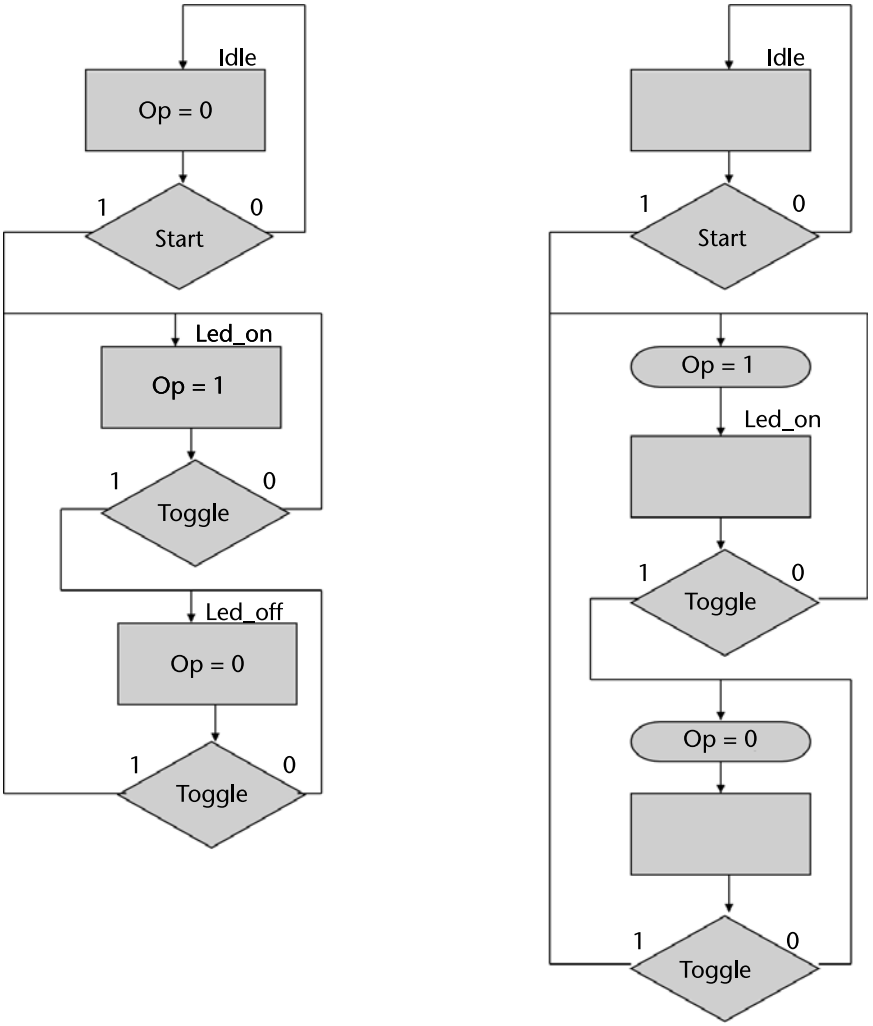



**Figure 3.21** Algorithmic state chart for the state machines in Figure 3.20 Moore (left) and Mealy (right).

A state machine does not have to be just Moore or Mealy; it is possible to create a hybrid state machine that uses both styles to create a more efficient implementation of the function required.

### 3.7.4 Implementing the State Machine

Using a high-level language like VHDL, it is easy to implement a state machine directly from the sate diagram. VHDL supports enumerated types that allows you as the engineer to define the actual state names, for example:

TYPE state IS (idle, led_on, led_off) ;

This type of declaration corresponds to the state diagrams shown in Figure 3.20, which waits for a button to be pushed then toggles a light emitting diode on and off.

There are many different methods of implementing a state machine; however, the two basic implementation methods are a one or two process approach. A one

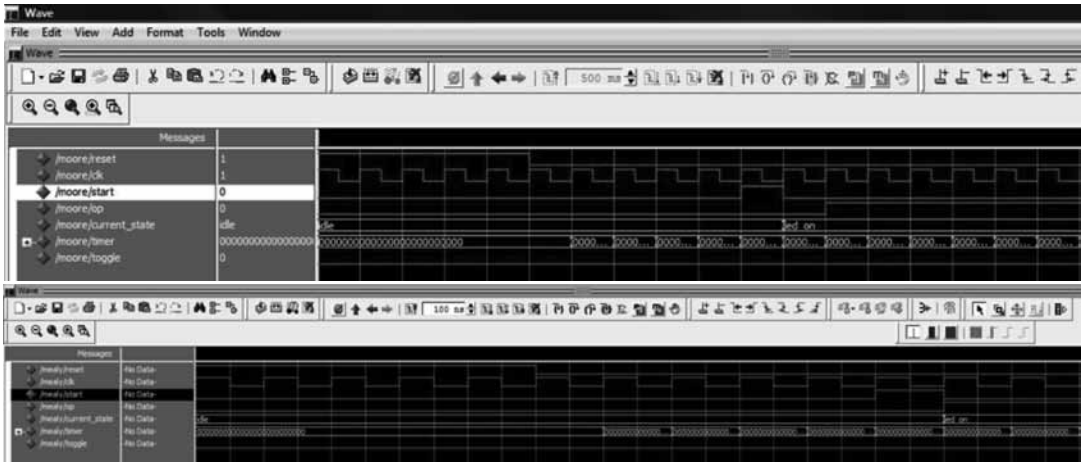



**Figure 3.22** Showing simulation results for Mealy and Moore outputs (Mealy bottom).

process contains both current and next state logic within a single process. The alternative two process approach separates the current and next state logic.

As a rule, engineers prefer to implement a single process state machine, these are seen as having several benefits over the traditional two process approach:

1. Removes the risk of incomplete coverage of signals in the combinatorial process creating latches.
2. State machine outputs are aligned to the clock.
3. Generally, they are easier to debug than their two-process implementation.

Regardless of which approach you decide to use to implement it, the next state determination and any outputs will be evaluated using a CASE statement. Figure 3.23 shows a side-by-side comparison between a Moore (left) and Mealy (right) using a single process approach.

### 3.7.5 State Machine Encoding

The state variable is stored within flip-flops that are updated with the next state on the next clock edge (even if there is no state change). How these flip-flops are used to represent the state value depends upon the number of states and if the engineer chooses to direct the synthesis tool in one particular method. The three most common types of state encoding are:

- *Sequential:* State encoding follows a traditional binary sequence for the states.
- *Gray:* Like the sequential encoding scheme with the exception of the states encoding using Gray code.
- *One-hot:* Has one flip-flop for each state within the state machine. Only one flip-flop is currently set high and the rest are low; hence, the name one-hot.

```
TYPE state IS (idle, led_on, led_off);
SIGNAL current_state : state := idle;

SIGNAL timer  : unsigned(24 DOWNTO 0) := (OTHERS =>'0');
SIGNAL toggle : std_logic := '0';

BEGIN

PROCESS(clk,reset)
BEGIN
  IF reset = '1' THEN
    current_state <= idle;
    op <= '0';
  ELSIF rising_edge(clk) THEN
  CASE current_state IS
    WHEN idle =>
      op <= '0'; --output is a function of the current state only
      IF start = '1' THEN
        current_state <= led_on;
      END IF;
    WHEN led_on =>
      op <= '1';
      IF toggle = '1' THEN
        current_state <= led_off;
      END IF;
    WHEN led_off =>
      op <= '0';
      IF toggle = '1' THEN
        current_state <= led_on;
      END IF;
    WHEN OTHERS =>
      op <= '0';
      current_state <= idle;
  END CASE;
 END IF;
END PROCESS;
```

```
TYPE state IS (idle, led_on, led_off);
SIGNAL current_state : state := idle;

SIGNAL timer  : unsigned(24 DOWNTO 0) := (OTHERS =>'0');
SIGNAL toggle : std_logic := '0';

BEGIN

PROCESS(clk,reset)
BEGIN
  IF reset = '1' THEN
    current_state <= idle;
    op <= '0';
  ELSIF rising_edge(clk) THEN
  CASE current_state IS
    WHEN idle =>
      IF start = '1' THEN
        current_state <= led_on;
        op <= '1'; --output is a function of the current state and inputs
      END IF;
    WHEN led_on =>
      IF toggle = '1' THEN
        current_state <= led_off;
        op <= '0';
      END IF;
    WHEN led_off =>
      IF toggle = '1' THEN
        current_state <= led_on;
        op <= '1';
      END IF;
    WHEN OTHERS =>
      op <= '0';
      current_state <= idle;
  END CASE;
 END IF;
END PROCESS;
```

**Figure 3.23** Moore and Mealy state machines in VHDL.

Both sequential and Gray encoding schemes will require several flip-flops that can be determined by:

$$FlipFlops = Ceil\left(\frac{LOG10(States)}{LOG10(2)}\right)$$

while one-hot encoding schemes require the same number of states as there are flip-flops.

The automatic assignment of state encoding depends upon the number of states that the state machine contains. While this will vary depending upon the synthesis tool selected, as a rule of thumb, the following encoding will be used:

- *Sequential:* <5 states;
- *One-hot:* Five to 50 states;
- *Gray:* >50 states.

Often, you will not necessarily think about what state encoding to use, instead allowing the synthesis engine to determine the correct implementation getting involved if the chosen style causes an issue. However, should the engineers need to take things into their own hands and define the state encoding, there is no need for them to define the state encoding long-hand, defining constants for each state within the state encoding. Instead, the engineers can use an attribute within the code to drive the synthesis tool to choose a particular encoding style as demonstrated below.

```
TYPE state IS (idle, led_on, led_off);
SIGNAL current_state : state := idle;
ATTRIBUTE syn_encoding STRING;
ATTRIBUTE syn_encoding OF current_state : SIGNAL IS "sequential";
```

where "sequential" can be also "gray" and "onehot"; these three choices can also be combined with the safe attribute to ensure that the state machine can recover to a valid state, should the state machine enter an illegal state.

You can also use the syn_encoding attribute to define the values of the state encoding directly. For example, suppose that the engineer desired to encode the 3-state state machine using the following state encoding; Idle = "11" led_on = "10," led_off = "01" as opposed to the more traditional sequence of "00," "01," and "10."

```
TYPE state IS (idle, led_on, led_off) ;
SIGNAL current_state : state := idle;
ATTRIBUTE syn_encoding STRING;
ATTRIBUTE syn_encoding OF current_state : SIGNAL IS "sequential";
```

As the engineer, you are responsible for setting the correct settings in the synthesis tool to ensure that attributes are not ignored.

### 3.7.6 Increasing Performance of State Machines

As a central part of the design, state machines and their implementation can have a significant impact on the achieved performance in the target device. Implementation performance does not come down only to the state encoding (e.g., one-hot versus sequential), but the actual structure of state machine itself.

One of the best methods of achieving better timing performance is to decouple as much functionality as possible from the state machine implementation. That is, within state machines, there is often a requirement to count, shift, or implement mathematical functions. These counters, shifters, and other functionality should be implemented as separate processes outside of the state machine. Communication with the state machine (e.g., to indicate a terminal count, reset a counter, or perform a mathematical operation) can be achieved by a single-bit signal, either generated by or received by the state machine.

Including this functionality within the state machine requires a larger implementation, making it harder for the implementation tool to optimize for performance due to more complex control logic, especially if wide buses are required.

Let us look at exactly what the difference in performance can be when using a state machine design to interface with a FLIR Lepton. This state machine will assert the chip select line and then received video over SPI frames back from the Lepton device. If the packet is valid, it will be written into a dual-port block RAM memory. If the packet is a corrupted packet, the state machine will read out the video frame as required for correct operation, but not write the corrupted frame to the block RAM storing valid pixels.

As can be expected, this design uses several counters to monitor bit and pixel positions along with controlling shift registers, RAM addresses, and RAM writes.

The first implementation uses a merged design that includes all functionality within the state machine. When this design was implemented within Xilinx Vivado 2020.1, the utilization was 45 LUTs and 102 flip-flops, giving a total performance of 164.56 MHz, as demonstrated in Figures 3.24 and 3.25.

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Slice LUTs | 45 | 0 | 32600 | 0.14 |
| LUT as Logic | 45 | 0 | 32600 | 0.14 |
| LUT as Memory | 0 | 0 | 9600 | 0.00 |
| Slice Registers | 102 | 0 | 65200 | 0.16 |
| Register as Flip Flop | 102 | 0 | 65200 | 0.16 |
| Register as Latch | 0 | 0 | 65200 | 0.00 |
| F7 Muxes | 0 | 0 | 16300 | 0.00 |
| F8 Muxes | 0 | 0 | 8150 | 0.00 |

(a)

| Design Timing Summary |
|---|

| WNS(ns) | TNS(ns) | TNS Failing Endpoints | TNS Total Endpoints | WHS(ns) | THS(ns) |
|---|---|---|---|---|---|
| 3.923 | 0.000 | 0 | 191 | 0.179 | 0.000 |

(b)

**Figure 3.24** Implementation and timing performance for the state machine when all functionality is included.

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Slice LUTs* | 36 | 0 | 32600 | 0.11 |
| LUT as Logic | 36 | 0 | 32600 | 0.11 |
| LUT as Memory | 0 | 0 | 9600 | 0.00 |
| Slice Registers | 102 | 0 | 65200 | 0.16 |
| Register as Flip Flop | 102 | 0 | 65200 | 0.16 |
| Register as Latch | 0 | 0 | 65200 | 0.00 |
| F7 Muxes | 0 | 0 | 16300 | 0.00 |
| F8 Muxes | 0 | 0 | 8150 | 0.00 |

(a)

| Design Timing Summary |
|---|

| WNS(ns) | TNS(ns) | TNS Failing Endpoints | TNS Total Endpoints | WHS(ns) | THS(ns) |
|---|---|---|---|---|---|
| 5.631 | 0.000 | 0 | 218 | 0.106 | 0.000 |

(b)

**Figure 3.25** Implementation and timing performance for the state machine when all functionality is decoupled.

Decoupling the state machine from the counters, RAM interfacing, and shift register results in slightly decreased utilization, but also a greatly increased maximum frequency of 228.88 MHz.

Note that the equation for conversion from WNS to Fmax is $1/(T - \text{WNS})$ where $T$ is the required target clock period. In this case, it is 10 ns, as both designs were targeted for 100-MHz operation.

### 3.7.7 Good Design Practices for FPGA Implementation

In addition to the decoupling of functionality outside of the FPGA, there are several additional good design practices that engineers at SensorsThink follow.

- *Use enumerated state definitions:* Using clearly defined enumerated state definitions makes the state machine much easier to understand and debug during commissioning. If the end application requires certification, clear state definitions and behavior enable independent reviewers to be able to understand the functionality and the design intent.
- *Single process state machine:* A single process state machine removes the potential for accidental latch creation in the combinatorial process when a branch is not fully defined. Single process state machines are also easier to debug and ensure that all the state machine outputs are registered, thus improving timing performance of the solution.
- *Consider unused states:* State machines implemented will always have a power of 2 states. The state machine may not use all the possible states for its implementation; these unused states can be an issue for high-reliability implementations and may need special handling to achieve reliability.
- *Check for terminal states:* A terminal state is one that, once entered, cannot be left; this may be intentional upon an error occurring. However, often, it is a design error in the state machine. Checking that each state has a path in and out prior to simulation can save significant time.
- *Reset:* Have a clear reset state defined for the current state.

Following these good design practices will enable our state machines to be not only effective, but also easier to debug and maintain as the life of the product evolves, which it no doubt will.

### 3.7.8 FLIR Lepton Interface

The FLIR Lepton deployed on the Smart Sensor SoC has a video over SPI interface. Further complicating the issue are U.S. export regulations, which limit the frame rate that the Lepton can output to 9 frames per second. To comply with this export requirement, the FLIR Lepton outputs corrupted frames intermixed with valid frames. Video over SPI requires the processor to maintain synchronization, across valid and corrupted frames; if synchronization is lost, the video output is lost.

The synchronization scheme is simple but time-critical: the FLIR Lepton transmits video packets over the SPI. Readout is started by the master, in this case, the Zynq de-asserting the SPI CS line for at least 5 image periods (185 ms). This causes the Lepton video output to time out and puts the Lepton in the right mode to resynchronize. To start new synchronization, the master asserts the CS line; following this, the Lepton will begin to output packets of data. Each packet consists of 4-byte

ID and CRC followed by 160 bytes of video when the imager is configured for raw output or 240 bytes when outputting RGB888. To read out the entire array, we need to read out 240 packets. To ensure frame compliance with export control rules, only 9 valid frames per second are output; in between these frames, the FLIR Lepton outputs corrupted frames. To maintain synchronization, these corrupted frames must be read out and discarded. After this has been achieved, the FLIR and the Zynq will be synchronized; to maintain synchronization, the Zynq must:

1. Not violate the intrapacket timeout. Once a packet starts, it must be completely clocked out within 3 line periods.
2. Not fail to read out all packets for a given frame before the next frame is available.
3. Not fail to read out all available frames.

While the Zynq processor has SPI interfaces, maintaining synchronization as the software application load increases becomes more difficult. Such a loading will also reduce the ability of the software to process other sensor data and complete analysis tasks.

Therefore, to ensure that the FLIR thermal imaging maintains synchronization in our Smart Sensor SoC, the synchronization and video capture will be implemented in the programmable logic.

To achieve synchronization, a simple state machine will be deployed that synchronizes with the output from the FLIR Lepton and writes valid video data to a block RAM memory that can be accessed by the processor. This enables the processor to access completed good frames from the programmable logic at the rate of 9 images per second.

The state diagram of the state machine is defined in Figure 3.26; this implementation uses several states to synchronize and capture the video data. The state machine works on the state of the 16-bit word that is received from the FLIR Lepton over the MISO SPI signal. This signal contains either a video packet information,

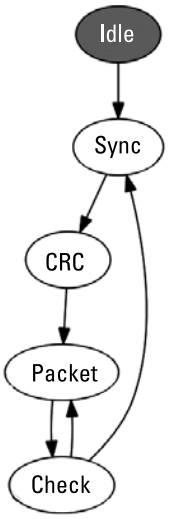



**Figure 3.26** FLIR Lepton FSM state diagram.

CRC, or video data. If the video packet is corrupted, it is discarded and the remaining video is read out. If the packet is valid, then the video is written to the connected block RAM to enable access from the processor for the image to be rendered.

The implementation for the state machine results in the images of FLIR Lepton being able to be synchronized and maintain synchronization with the Zynq and the images being made available to the process for further deployment in the Internet of Things (IOT) application.

## 3.8 Defensive State Machine Design

As we develop our Smart Sensor SoC, we must be careful about ensuring that it can work in harsh environments as outlined in the requirements in Chapter 1. This harsh environment could mean that the system has to be able to work in electrically noisy, high-altitude, or dynamic environments.

Engineers at SensorsThink must therefore carefully consider the operating environment against the need to design defensively in the implementation of state machines. If the SensorsThink engineering team considers it appropriate to design defensively, the engineer must take additional care in the design and implementation of the state machines (as well as all accompanying logic) inside the programmable logic device to ensure that they can function within the requirements.

One of the major causes of errors within state machines is single-event upsets caused by either a high-energy neutron or an alpha particle striking sensitive sections of the device silicon. Single event upsets (SEUs) can cause a bit to flip its state (0 -> 1 or 1 -> 0), resulting in an error in device functionality that could potentially lead to the loss of the system or even endanger life if incorrectly handled. Because these SEUs do not result in any permanent damage to the device itself, they are called soft errors.

Terrestrially single event upsets are more likely to occur at higher altitudes and certain latitudes.

The backbone of most programmable logic design is the finite state machine, a design methodology that engineers use to implement control, data flow and algorithmic functions. When implementing state machines within FPGAs, designers will choose one of two styles (binary or one-hot), although, in many cases, most engineers allow the synthesis tool to determine the final encoding scheme. Each implementation scheme presents its own challenges when designing reliable state machines for mission-critical systems. Indeed, even a simple state machine can encounter several problems (Figure 3.27). You must pay close attention to the encoding scheme and, in many cases, take the decision about the final implementation encoding away from the synthesis tool.

### 3.8.1 Detection Schemes

Let us first look at binary implementations (sequential or Gray encoding), which often have leftover, unused states that the state machine does not enter when it is functioning normally. Designers must address these unused states to ensure that the state machine will gracefully recover if it should accidentally enter an illegal state. There are two main methods of achieving this recovery. The first is to declare all

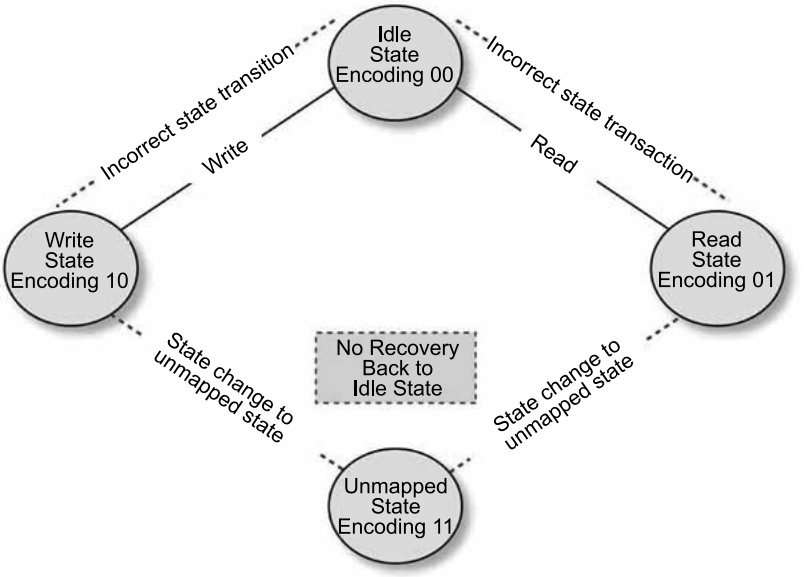



**Figure 3.27** Even a simple state machine can encounter several types of errors.

2N number of states when defining the state machine signal and cover the unused states with the "others clause" at the end of the case statement. The others clause will typically set the outputs to a safe state and send the state machine back to its idle state or another state, as identified by the design engineer. This approach will require the use of synthesis constraints to prevent the synthesis tool from optimizing these unused states from the design, as there are no valid entry points. This typically means synthesis constraints within the body of the RTL code ("syn_keep").

The second method of handling the unused states is to cycle through them at startup following reset release. Typically, these states also keep the outputs in a safe state; should they be accidentally entered, the machine will cycle around to its idle state again.

One-hot state machines have one flip-flop for each state, but only the current state is set high at any one time. Corruption of the machine by having more than one flip-flop set high can result in unexpected outcomes. You can protect a one-hot machine from errors by monitoring the parity of the state registers. Should you detect a parity error, you can reset the machine to its idle state or to another predetermined state.

With both methods, the state machine's outputs go to safe states and the state machine restarts from its idle position. State machines that use these methods can be said to be SEU detecting, as they can detect and recover from an SEU, although the state machines' operation will be interrupted. You must take care during synthesis to ensure that register replication does not result in registers with a high fan-out being reproduced and hence left unaddressed by the detection scheme. Take care also to ensure that the error does not affect other state machines with which this machine interacts.

Many synthesis tools offer the option of implementing a safe state machine option. This option often includes more logic to detect the state machine entering an illegal state and send it back to a legal one, normally the reset state. For

a high-reliability application, design engineers can detect and verify these illegal state entries more easily by implementing any of the previously described methods. Using these approaches, the designers must also consider what would happen should the detection logic suffer from an SEU. What effect would this have upon the reliability of the design? Figure 3.28 is a flowchart that attempts to map out the decision-making process for creating reliable state machines.

The techniques presented so far detect or prevent an incorrect change from one legal state to another legal state. Depending upon the end application, this could result in anything from a momentary system error to the complete loss of the mission.

### 3.8.2 Hamming Schemes

Techniques for detecting and fixing incorrect legal transitions are triple-modular redundancy and Hamming encoding. The latter provides a Hamming distance of 3 and covers all possible adjacent states (see Figure 3.29). A simpler technique for preventing legal transitions is the use of Hamming encoding with a Hamming distance of 2 (rather than 3) between the states and not covering the adjacent states. However, this will increase the number of registers that your design will require.

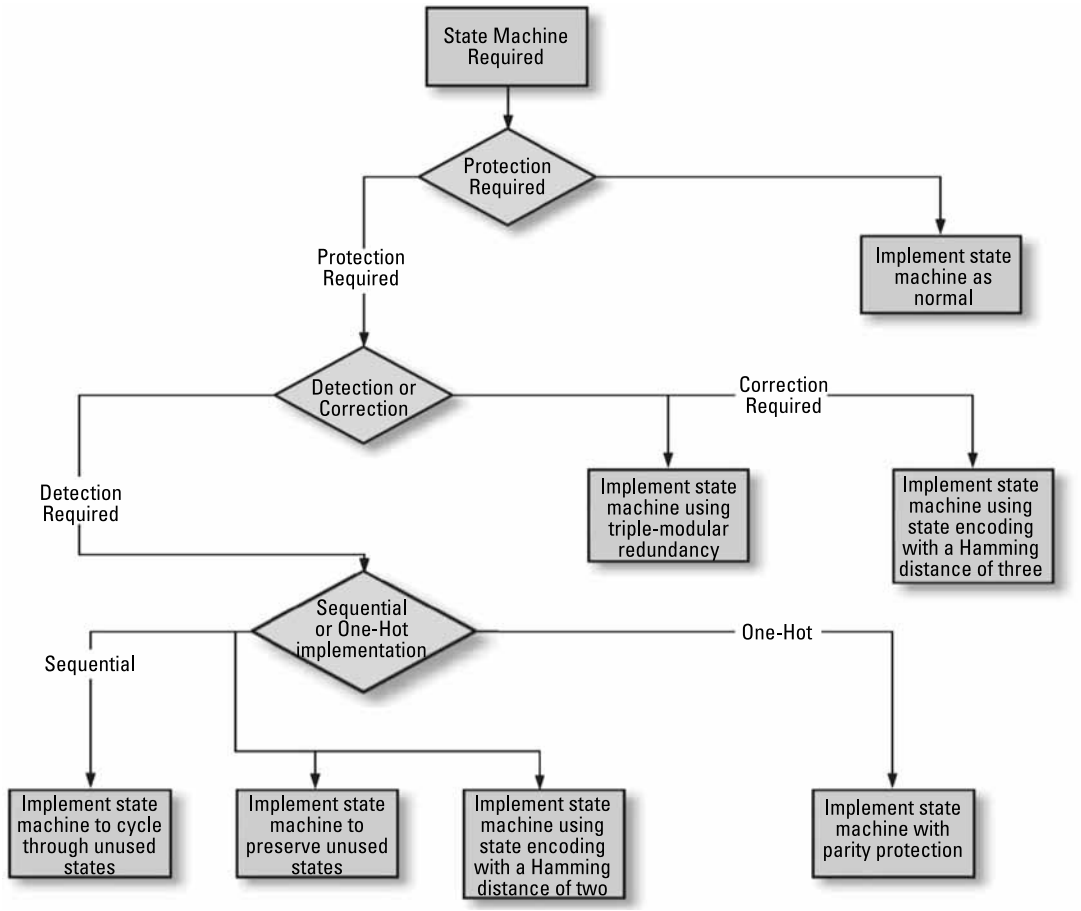



**Figure 3.28** This flowchart maps out the decision process for creating reliable state machines.

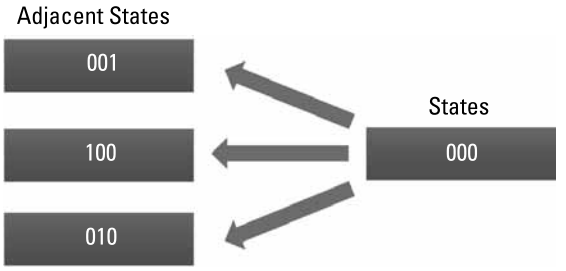



**Figure 3.29** Hamming code adjacent states.

Triple-modular redundancy (TMR), for its part, involves implementing three instantiations of the state machine with majority voting upon the outputs and the next state. This is the simpler of the two approaches and many engineers have used it in several applications over the years. Typically, a TMR implementation will require spatial separation of the logic within the FPGA to ensure that an SEU does not corrupt more than one of the three instantiations. It is also important to remove any registers from the voter logic because they can create a single-point failure in which an SEU could affect all three machines.

The use of a state machine with encoding that provides a Hamming distance of 3 between states will ensure both SEU detection and correction. This guarantees that more than a single bit will change between states, meaning that an SEU cannot make the state transition from one legal state to another erroneously. The use of a Hamming distance of 2 between states is like the implementation for the sequential machine, where the unused states are addressed by the "when others" clause or reset cycling. However, as the states are explicitly declared to be separate from each other by a Hamming distance of 2 within the RTL, the state machine cannot move from one legal state to another erroneously and will instead go to its idle state should the machine enter an illegal state. This provides a more robust implementation than the binary one mentioned above.

If you wish to implement a state machine that will continue to function correctly should an SEU corrupt the current state register, you can do so by using a Hamming-code implementation with a Hamming distance of 3 and ensuring its adjacent states are also addressed within the state machine. Adjacent states are those that are 1 bit different from the state register and hence achievable should an SEU occur. This use of states adjacent to the valid state to correct for the error will result in $N * (M + 1)$ states, where $N$ is the number of states and $M$ is the number of bits within the state register. It is possible to make a small high-reliability state machine using this technique, but crafting a large one can be so complicated as to be prohibitive. The extra logic footprint associated with this approach could also result in lower timing performance.

### 3.8.3 Deadlock and Other Issues

There are other issues to consider when designing a high-reliability state machine beyond the state encoding schemes. Deadlock can occur when the state machine enters a state from which it is never able to leave. An example would be one state machine awaiting an input from a second state machine that has entered an illegal

state and hence been reset to idle without generating the needed signal. To avoid deadlock, it is therefore good practice to provide timeout counters on critical state machines. Wherever possible, these counters should not be included inside the state machine but placed within a separate process that outputs a pulse when it reaches its terminal count. Be sure to write these counters in such a way as to make them reliable.

When checking that a counter has reached its terminal count, it is preferable to use the greater-than-or-equal-to operator, as opposed to just the equal-to operator. This is to prevent an SEU from occurring near the terminal count and hence no output being generated. You should declare integer counters to a power of 2, and, if you are using VHDL, they should also be modulo to the power of 2 to ensure in simulation that they will wrap around as they will in the final implementation [count <= (count + 1) Mod 16; for a 0-bit to 15-bit integer counter]. Unsigned counters do not require this because there is no simulation mismatch between RTL and post-route simulation regarding wraparound. You can replicate data paths within the design and compare outputs on a cycle-by-cycle basis to detect whether one of them has been subjected to an SEU event. Wherever possible, edge-detect signals to enable the design to cope with inputs that are stuck high or stuck low. You should analyze each module within the design at design time to determine how it will react to stuck-high or stuck-low inputs. This will ensure that it is possible to detect these errors and that they cannot have an adverse effect upon the function of the module.

### 3.8.4 Implementing Defensive State Machines in Xilinx Devices

The defensive design techniques considered by engineers at SensorsThink to protect state machines may at first glance seem to require considerable design effort in the creation and definition, in both in the RTL and the synthesis tool. However, when developing for Xilinx targets and the using the Xilinx Synthesis engine within Vivado, engineers at SensorsThink can use synthesis attributes to implement defensively designed state machines.

The synthesis attribute FSM_SAFE_STATE, defined in either the source code or the XDC file in Vivado, allows us to implement the following state machines:

- *auto_safe_state:* Hamming 3 encoded to ensure tolerance of SEU.
- *reset_state:* Hamming 2 encoding to force the state machine to its reset state on error.
- *power_on_state:* Hamming 2 encoding to force the state machine to its power-on state on error.
- *default:* Hamming 2 forces the state machine to the default / when others case-define in the HDL.

Using the state machine presented in Figure 3.27 as an example, the attribute can be applied in the RTL source code as shown.

```
type state is (idle, rd, wr, unmapped);
signal current_state : state;
signal transfer_word : std_logic_vector(31 downto 0);
```

```
attribute fsm_safe_state    : string;
attribute fsm_safe_state of current_state : signal is
"auto_safe_state";
```

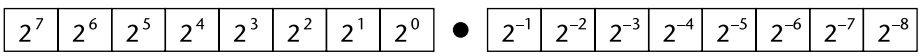
Proof the state machine has been implemented using a Hamming 3 encoding can be observed in both the synthesis report and the elaborated design.

## 3.9 How Does FPGA Do Math?

Along with the finite state machine control structures, SensorsThink FPGA engineers are going to need to implement several mathematical algorithms. It is therefore crucial that all the SensorsThink engineering team understand how math is implemented in programmable logic. One of the many benefits of an FPGA-based solution is that we engineers can implement the mathematical algorithm in the best possible solution for the problem at hand. For example, if response time is critical, then we can pipeline stages of mathematics. If accuracy of the result is more critical, we can use more bits to ensure the desired accuracy is achieved. Many modern FPGAs also provide us the benefit of embedded multipliers and DSP slices, which can be used to ensure the optimal implementation is achieved in the target device.

### 3.9.1 Representation of Numbers

There are two methods of representing numbers within a design, fixed, or floating-point number systems that we can use. Fixed-point representation maintains the decimal point within a fixed position allowing for straightforward arithmetic operations. The major drawback of fixed-point representation is that, to represent larger numbers or to achieve a more accurate result with fractional numbers, a larger number of bits are required. A fixed-point number consists of two parts called the integer and fractional parts. Floating point representation allows the decimal point to float to different places within the number depending upon the magnitude; floating point numbers are divided into two parts: the exponent and the mantissa. This is remarkably like scientific notation, which represents a number as A times 10 to the power B where A is the mantissa and B is the exponent; however, the base of the exponent in a floating-point number is base 2, that is, A times 2 to the power B. The floating-point number is standardized by the IEEE/ANSI Standard 754-1985; the basic IEEE floating point number (there are many) utilizes an 8-bit exponent and a 24-bit mantissa.




Due to the complexity of floating-point numbers, we as designers tend wherever possible to point representation. The above fixed-point number can represent an unsigned number between 0.0 and 255.9906375 or a signed number between –128.9906375 and 127.9906375 using two's complement representation. Within a design, we have the choice to use either unsigned or signed numbers; typically, this will be constrained by the algorithm being implemented. Unsigned numbers can represent a range of 0 to $2n - 1$ and always represent positive numbers, while

the range of a signed number depends upon the encoding scheme used: sign and magnitude, one's complement, or two's complement.

Sign and magnitude utilize the leftmost bit to represent the sign of the number (0 = positive, 1 = negative); the remainder of the bits represents the magnitude. Therefore, in the sign and magnitude system, both positive and negative numbers have the same magnitude; however, the sign bit differs. Due to this, it is possible to have both a positive and negative zero within the sign and magnitude system.

The one's complement uses the same unsigned representation for positive numbers as sign and magnitude representation. However, for negative numbers, the inversion (one's complement) of the positive number is used.

The two's complement is the most widely used encoding scheme for representing signed numbers. Just like sign and magnitude, one's complement schemes that are positive are represented in the same manner as an unsigned number, whereas negative numbers are represented as the binary number that you add to a positive number of the same magnitude to get zero. A negative two's complement number is calculated by first taking the one's complement (inversion) of the positive number and then adding 1 to it. The two's complement number system allows subtraction of one number from another by performing an addition of the two numbers. The range that a two's complement number can represent is given by

$$-(2n-1) \text{ to } +(2n-1-1)$$

One method that we can use to convert a number to its two's complement format is to work right to left, leaving the number the same until the first one is encountered; after this, each bit is inverted.

## 3.10 Fixed Point Mathematics

The normal way of representing the split between integer and fractional bits within a fixed-point number is x,y where x represents the number of integer bits and y represents the number of fractional bits. For example, 8,8 represents 8 integer bits and 8 fractional bits, while 16,0 represents 16 integer buts and 0 fractional bits. In many cases, the correct choice of the number of integer and fractional bits required will be undertaken at design time normally following conversion from a floating-point algorithm. Thanks to the flexibility of FPGAs, we can represent a fixed-point number of any bit length; the number of integer bits required depends upon the maximum integer value that the number is required to store, while the number of fractional bits will depend upon the accuracy of the final result. To determine the number of integer bits required, we can use the following equation

$$\text{Integer Bits Required} = Ceil\left(\frac{LOG_{10}\text{Integer\_Maximum}}{LOG_{10}2}\right)$$

For example, the number of integer bits required to represent a value between 0.0 and 423.0 is given by

$$9 = Ceil\left(\frac{LOG_{10}423}{LOG_{10}2}\right)$$

meaning that we would need 9 integer bits, allowing a range of 0 to 511 to be represented. If we want to represent the number using 16 bits, this would allow for 7 fractional bits. The accuracy that this representation would be capable of providing is given by

$$accuracy = \left(\frac{Actual\_Value - FPGA\_Value}{2^{\text{Fractional Bits}}}\right) \times 100$$

We can increase the accuracy of a fixed-point number by using more bits to store the fractional number. When we are designing, there are times when you may wish to store only fractional numbers (0,16) depending upon the size of the number that you wish to scale up; scaling up by $2^{16}$ may result in a number that still does not provide an accurate enough result. In this case, we can multiply up by power of 2 such that the number can be represented within a 16-bit number. We can then remove this scaling at a further stage within the implementation. For example, to represent the number $1.45309806319 \times 10^{-4}$ in a 16-bit number, the first step is to multiply it by $2^{16}$:

$$65{,}536 * 1.45309806319 \times 10–4 = 9.523023$$

Storing the integer of the result (9) will result in the number being stored as $1.37329101563 \times 10^{-4}$ (9/65,536). This difference between the number required to be stored and the stored number is substantial and could lead to an unacceptable error in the calculated result. We can obtain a more accurate result by scaling the number up by a factor of 2, which produces a result between 32,768 and 65,535, therefore still allowing storage in a 16-bit number. Using the earlier example of storing $1.45309806319 \times 10^{-4}$, we can, by multiplying by factor of $2^{28}$, obtain a number that can be stored in 16 bits and will be highly accurate of the desired number.

$$268{,}435{,}456 * 1.45309806319 \times 10–4 = 39{,}006.3041205$$

The integer of the result will result in the stored number being $1.45308673382 \times 10^{-4}$, which will give a much more accurate calculation, provided that the scaling factor of $2^{28}$ can be addressed at a later stage within the calculation. For example, a multiplication of the scaled number with a 16-bit number scaled 4,12 will produce a result of 4, 40 (28 + 12); however, the result will be stored in a 32-bit result.

### 3.10.1 Fixed-Point Rules

To perform addition, subtraction, or division, the decimal points of both numbers must be aligned. That is, an x,8 number can only be added to, subtracted from,

or divided by a number that is also in an x,8 representation. To perform arithmetic operations on numbers of different x,y formats, we must first ensure that the decimal points are aligned. To align a number to a different format, you have two choices: either multiply the number with more integer bits by $2^X$, or divide the number with the least number of integer bits by $2^X$. When dividing by $2^X$, the accuracy will be reduced and may lead to a result that is outside the allowable tolerance. As all numbers are stored in base 2, scaling up or down can be achieved easily in an FPGA through shifting one place to the left or right for each power of 2 required to balance the two decimal points. To add together two numbers that are scaled 8,8 and 9,7, you can either scale up the 9,7 number by a factor of $2^1$ or scale the 8,8 format down to a 9,7 format if the loss of a least significant bit is acceptable, for example, adding 234.58 and 312.732, which are stored in an 8,8 and 9,7 formats, respectively. The first step is to determine the actual 16-bit numbers, which will be added together.

$$234.58 * 28 = 60{,}052.48$$

$$312.732 * 27 = 40{,}029.69$$

The two numbers to be added together are 60,052 and 40,029; however, before the two numbers can be added together the decimal point must be aligned. To align the decimal points by scaling up the number with the largest number of integer bits, the 9,7 format number must be scaled up by a factor of $2^1$:

$$40{,}029 * 2^1 = 80{,}058$$

The result can then be calculated by performing an addition of

$$80{,}058 + 60{,}052 = 140{,}110$$

This represents 547.3046875 in a 10,8 format ($140{,}110/2^8$).

When multiplying two numbers together, the decimal points do not need to be aligned, as the multiplication will provide a result that is X1 + X2, Y1 + Y2 wide. Multiplying two numbers, which are formatted 14,2 and 10,6, will produce a result, which is formatted 24 integer bits and 8 fractional bits. Multiplication by a fractional number can be used in the place of a division within an equation, by multiplying by the reciprocal of the divisor. Using this approach can reduce the complexity of the design significantly. For example, to divide the number 312.732 represented in a 9,7 (40,029) format by 15, the first stage is to calculate the reciprocal of the divisor.

$$\frac{1}{15} = 0.066666'$$

This reciprocal must then be scaled up to be represented within a 16-bit number:

$$65{,}536 * 0.06666 = 4{,}369$$

This will produce a result that is formatted 9,23 when the two numbers are multiplied together:

$$4{,}369 * 40{,}029 = 174{,}886{,}701$$

The result of this multiplication is thus

$$\frac{174886701}{8388608} = 20.8481193781$$

While the expected result is 20.8488, if the result is not accurate enough, then the reciprocal can be scaled up by a larger factor to produce a more accurate result. Therefore, never divide by a number when you can multiply by the reciprocal.

### 3.10.2 Overflow

When implementing algorithms, we must ensure that the result is not larger than what is capable of being stored within the result register. When this condition occurs, it is known as overflow; when overflow occurs, the stored result will be incorrect and the most significant bits are lost. A remarkably simple example of overflow would occur if two 16-bit numbers each with a value of 65,535 were added together and the result was stored within a 16-bit register.

$$65{,}535 + 65{,}535 = 131{,}070$$

The above calculation would result in the 16-bit result register containing a value of 65,534, which is incorrect. The simplest way to prevent overflow is to determine the maximum value, which will result from the mathematical operation, and use the first equation in this chapter to determine the size of the result register required. If an averager were being developed to calculate the average of up to 50 16-bit inputs, the size of the required result register can be calculated.

$$50 * 65{,}535 = 3{,}276{,}750$$

Using the first equation in this chapter would require a 22-bit result register to prevent overflow occurring. Care must also be taken when using signed numbers to ensure that overflow does not occur when using negative numbers. Using the averager example again, this time taking 10 averages of a signed 16-bit number and returning a 16-bit result:

$$10 * -32{,}768 = -327{,}680$$

As it is easier to multiply the result by a scaled reciprocal of the divisor, this number can be multiplied by 1/10 * 65,536 = 6,554 to determine the average.

$$-32{,}768 * 6{,}554 = -2{,}147{,}614{,}720$$

This number, when divided by 216 = –32,770, cannot be represented correctly within a 16-bit output. The module design must therefore take into account the overflow and detect it to ensure that an incorrect result is not output.

### 3.10.3 Real-World Implementation

A module is required to implement a transfer function that is used to convert atmospheric pressure measured in millibars into altitude measured in meters.

$$-0.0088 \times 2 + 1.7673x + 131.29$$

The input value will range between 0 and 10 millibars with a resolution of 0.1 millibar. The output of the module is required to be accurate to +/–0.01m. As the module specification does not determine the input scaling, this can be determined by the following equation.

$$4 = Ceil\left(\frac{LOG_{10}10}{LOG_{10}2}\right)$$

Therefore, to ensure maximum accuracy, the input data can be formatted as 4 integer bits and 12 fractional bits. The next step in the development of the module is to use a spreadsheet to calculate the expected result of the transfer function across the entire input range using the unscaled values. If the input range is too large to reasonably achieve this, an acceptable number of points should be calculated. For this example, 100 entries can be used to determine the expected result across the entire input range.

Once the initial unscaled expected values have been calculated, the next step is to determine the correct scaling factors for the constants and calculate the expected outputs using the scaled values. To ensure maximum accuracy, each of the constants used within the equation will be scaled by a different factor.

The first polynomial constant ($A$) scaling factor is given by

$$8 = Ceil\left(\frac{LOG_{10}133.29}{LOG_{10}2}\right)$$

The second polynomial constant ($B$) scaling factor is given by

$$1 = Ceil\left(\frac{LOG_{10}1.7673}{LOG_{10}2}\right)$$

The final polynomial constant ($C$) can be scaled up by a factor of $2^{16}$, as it is completely fractional.

**Table 3.2** Pressure to Altitude Conversion

| *Input (millibar)* | *Output (meters)* |
|---|---|
| 0 | 131.2900 |
| 0.1 | 131.4666 |
| 0.2 | 131.6431 |
| 0.3 | 131.8194 |
| 0.4 | 131.9955 |
| 0.5 | 132.1715 |
| 0.6 | 132.3472 |

**Table 3.3** Calculated Scale and Unscaled Constants

| *Polynomail Constant* | *Unscaled* | *Scaled* |
|---|---|---|
| A | 133.29 | 33610 |
| B | 1.77 | 57910 |
| C | –0.01 | –577 |

These scaling factors allow the scaled spreadsheet to be calculated, this can be seen in Table 3.4. The results of each stage of the calculation will produce a result that will require more than 16 bits.

The calculation of the $Cx^2$ will produce a result that is 32 bits long when formatted 4,12 + 4,12 = 8,24; this is then multiplied by the constant C producing a result that will be 48 bits long when formatted 8,24 + 0,16 = 8,40. For the accuracy required in this example, 40 bits of fractional representation are excessive; therefore, the result will be divide by $2^{32}$ to produce a result with a bit length of 16 bits formatted as 8,8. The same reduction to 16 bits is carried out upon the calculation of Bx to produce a result that is formatted as 5,11. The result is the addition of columns $Cx^2$, Bx, and A; however, to obtain the correct result, the radix points must first be aligned. This can be achieved through either the shifting up of A and $Cx^2$ to align the numbers in an x,11 format. The alternate approach would be to shift down the calculated Bx to a format of 8,8 aligning the radix points with the calculated values of A and $Cx^2$.

In this example, the calculated value was shifted down by $2^3$ to align the radix points in an 8,8 format. This approach simplified the number of shifts required and therefore reduces the logic need to implement the example. Note that if the accuracy could not be achieved through by shifting down to align the radix points, then the radix points must be aligned by shifting up the calculated values of A and $Cx^2$. In this example, the calculated result is scaled up by a power of $2^8$; the result can then be scaled down and compared against the result calculated obtained with unscaled values. The difference between the calculated result and the expected result is then the accuracy; using the spreadsheet commands of MAX() and MIN(), the maximum and minimum error of the calculated result can be obtained across the entire range of spreadsheet entries. Once the calculated spreadsheet confirms that the required accuracy can be achieved, the RTL code can be written and

**Table 3.4** Expected Results and Scaled Result Accuracy

| *Input Scaled* | *C* | *B* | *A* | *Result* | *Result Scaled* | *Expected Result* | *Difference* |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 33610 | 33610 | 131.289 | 131.2900 | 0.0009 |
| 409 | –6 | 361 | 33610 | 33655 | 131.465 | 131.4666 | 0.0018 |
| 819 | –24 | 723 | 33610 | 33700 | 131.641 | 131.6431 | 0.0025 |
| 1228 | –52 | 1085 | 33610 | 33745 | 131.816 | 131.8194 | 0.0030 |
| 1638 | –93 | 1447 | 33610 | 33790 | 131.992 | 131.9955 | 0.0033 |
| 2048 | –145 | 1809 | 33610 | 33835 | 132.168 | 132.1715 | 0.0035 |
| 2457 | –208 | 2171 | 33610 | 33880 | 132.344 | 132.3472 | 0.0035 |
| 2867 | –283 | 2533 | 33610 | 33925 | 132.520 | 132.5228 | 0.0033 |

simulated. If desired, the testbench can be designed such that the input values are the same as those used in the spreadsheet; this allows the simulation outputs to be compared against the spreadsheet's calculated results to ensure the correct RTL implementation.

### 3.10.4 RTL Implementation

The RTL example uses signed parallel mathematics to calculate the result within 4 clock cycles. Because of the signed parallel multiplication, care must be taken to ensure the extra sign bits generated by the multiplications.

{start_code}

```
ENTITY transfer_function IS PORT(
sys_clk : IN std_logic;
reset : IN std_logic;
data : IN std_logic_vector(15 DOWNTO 0);
new_data : IN std_logic;result : OUT std_logic_vector(15 DOWNTO
0);
new_res : OUT std_logic);
END ENTITY transfer_function;
ARCHITECTURE rtl OF transfer_function IS
-- this module performs the following transfer function -0.0088x2
+ 1.7673x + 131.29
-- input data is scaled 8,8, while the output data will be scaled
8,8.
-- this module utilized signed parallel mathematics
TYPE control_state IS (idle, multiply, add, result_op);
CONSTANT c : signed(16 DOWNTO 0) := to_signed(-577,17);
CONSTANT b : signed(16 DOWNTO 0) := to_signed(57910,17);
CONSTANT a : signed(16 DOWNTO 0) := to_signed(33610,17);SIGNAL
current_state : control_state;
SIGNAL buf_data : std_logic; --used to detect rising edge upon
the new_data
SIGNAL squared : signed(33 DOWNTO 0); -- register holds input
squared.
SIGNAL cx2 : signed(50 DOWNTO 0); --register used to hold Cx2
SIGNAL bx : signed(33 DOWNTO 0); -- register used to hold bx
SIGNAL res_int : signed(16 DOWNTO 0); --register holding the tem-
porary result
BEGIN
```

```
fsm : PROCESS(reset, sys_clk)
BEGIN
IF reset = '1' THEN
buf_data <= '0';
squared <= (OTHERS => '0');
cx2 <= (OTHERS => '0');
bx <= (OTHERS => '0');
result <= (OTHERS => '0');
res_int <= (OTHERS => '0');
new_res <= '0';
current_state <= idle;
ELSIF rising_edge(sys_clk) THEN
buf_data <= new_data;
CASE current_state IS
WHEN idle =>
new_res <= '0';
IF (new_data = '1') AND (buf_data = '0') THEN --detect rising
edge new data
squared <= signed( '0'& data) * signed('0'& data);
current_state <= multiply;
ELSE
squared <= (OTHERS =>'0');
current_state <= idle;
END IF;
WHEN multiply =>
new_res <= '0';
cx2 <= (squared * c);
bx <= (signed('0'& data)* b);
current_state <= add;
WHEN add =>
new_res <= '0';
res_int <= a + cx2(48 DOWNTO 32) + ("000"& bx(32 DOWNTO 19));
current_state <= result_op;
WHEN result_op =>
result <= std_logic_vector(res_int(res_int'high -1 DOWNTO 0));
new_res <= '0';
current_state <= idle;
END CASE;
END IF;
END PROCESS;
END ARCHITECTURE rtl;
```

## 3.11 Polynomial Approximation

As the SensorsThink team develops the FPGA solution, they may be required to implement several significant mathematical algorithms. The SensorsThink engineering team may find implementing these algorithms directly will result in a complex implementation. This complex implementation may have knock-on effects in timing closure and verification. One technique that the SensorsThink engineering team might consider for complex algorithms is a different approach to direct implementation. This alternative approach is to leverage polynomial approximation.

### 3.11.1 The Challenge with Some Algorithms

Due to their flexibility and performance, FPGAs are used for a number of industrial, science, military, and other applications that require the calculation of complex mathematical transfer functions.

One example of this would be within an FPGA that monitors a Platinum Resistance Thermometer (PRT) and converts the resistance of the PRT into a temperature. This is typically achieved using a Calendar Van Duson equation that is shown in the simplified form here (valid between 0°C and 660°C)

$$R = R_0 \times \left(1 + a \times t + b \times t^2\right)$$

where $R_0$ is the resistance at 0°C and $a$ and $b$ are coefficents of the PRT.

Most systems that use a PRT will know the resistance from the design of the electronic circuit and the calculating temperature is what is desired; this requires rewriting the equation as:

$$t = \frac{-R_0 \times a + \sqrt{R_0^2 \times a^2 - 4 \times R_0 \times b \times \left(R_0 - R\right)}}{2 \times R_0 \times b}$$

Plotting the resistance against temperature will result in a graph as shown in Figure 3.30. From the graph, you can clearly see the nonlinearity.

Implementing this directly could be a significant challenge for an engineer and many will look for different methods to implement the function in an easier manner. One possible approach would be a look-up table storing several points on the curve with linear interpolation between the points within the look-up table. Depending upon the accuracy requirement and number of elements stored within

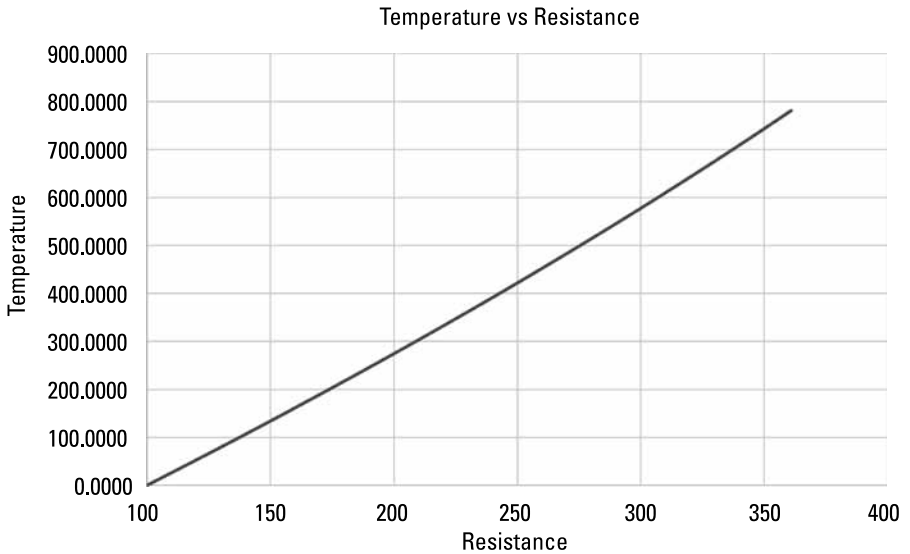



**Figure 3.30** The plotted transfer function.

a look-up table, this may prove acceptable; however, you will still need a linear interpolator function which can be complex mathematically.

### 3.11.2 Capitalize on FPGA Resources

Modern FPGAs contain much more than the traditional look-up table and flip-flop. These days, FPGA contain DSP slices and block and distributed RAM, along with many more advanced hard-core IP such as PCIe endpoints and high-speed serial links.

It is these internal RAM structures and DSP slices that can be used to implement transfer functions with greater ease. The DSP slices are often referred to as DSP48s due to the 48-bit accumulator that these slices provide.

One method that utilizes the DSP slice-rich architecture of the FPGA is to plot the mathematical function, including the input value range across in mathematical program, for instance, MATLAB or Excel. The engineer can then add a polynomial trend line to this dataset such that the equation for the trend line can then be implemented within the FPGA in place of the mathematically complex function, providing that the trend line equation meets the accuracy requirements. The accuracy can be easily verified by calculating the resultant output using the polynomial equation and comparing it against the result of the transfer function. When this is performed for the temperature transfer function, we use our example in Microsoft Excel. We obtain for this example a trend line that sits right on top of the original data set, as seen in Figure 3.31.

Having obtained the polynomial fit for the transfer function that we want to implement, we can then double-check this for accuracy against the transfer function again using the same analysis tool, in this case, Excel. For the case in point where temperature is being monitored, it may be the case that the end measurement has to be accurate to +/–1°C, for example. Depending upon the range of measurement, this may prove difficult to achieve using just one polynomial equation.

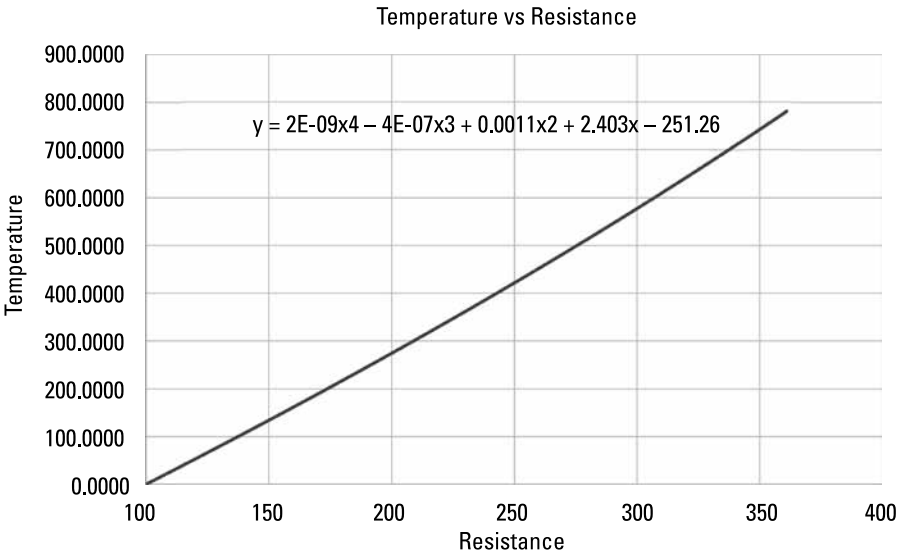



**Figure 3.31** Trend line and polynomial equation for the temperature transfer function.

### 3.11.3 Multiple Trend Lines Selected by Input Value

This can be addressed by changing the polynomial constants used above once the input goes above a specified value. Continuing with the temperature example, the first polynomial equation provides +/–1°C of accuracy between 0°C and 268°C. While for many applications this will be more than sufficient, we may require this tolerance across a range out to 300°C, which would limit our approach. However, we can address this by plotting the range between 269°C and 300°C (see Figure 3.32) and obtaining a different polynomial equation that will provide more accuracy for this output range.

This approach allows the engineering team to break the transfer function into several segments to achieve the desired accuracy. These segments can be either uniformly spaced across the transfer function (i.e., split into 10 segments of equal value of X). Alternatively, they can be nonuniform and segmented as required to achieve the desired accuracy.

## 3.12 The CORDIC Algorithm

Implementing mathematical algorithms can be a significant challenge; however, techniques such as polynomial approximation can significantly help the SensorsThink engineering team develop mathematical algorithms. Another complementary approach with which SensorsThink engineers must be familiar is the CORDIC algorithm. This algorithm can help us to implement trigonometric and transcendental functions, thanks to its linear and hyperbolic extensions.

Most engineers tasked with implementing a mathematical function within an FPGA such as sine, cosine, or square root may initially think of implementing this via a look-up table, possibly combined with linear interpolation or a power series if multipliers are available. However, the CORDIC (COordinate Rotation DIgital Computer) algorithm invented by Jack Volder for the B58 program in 1959 is one

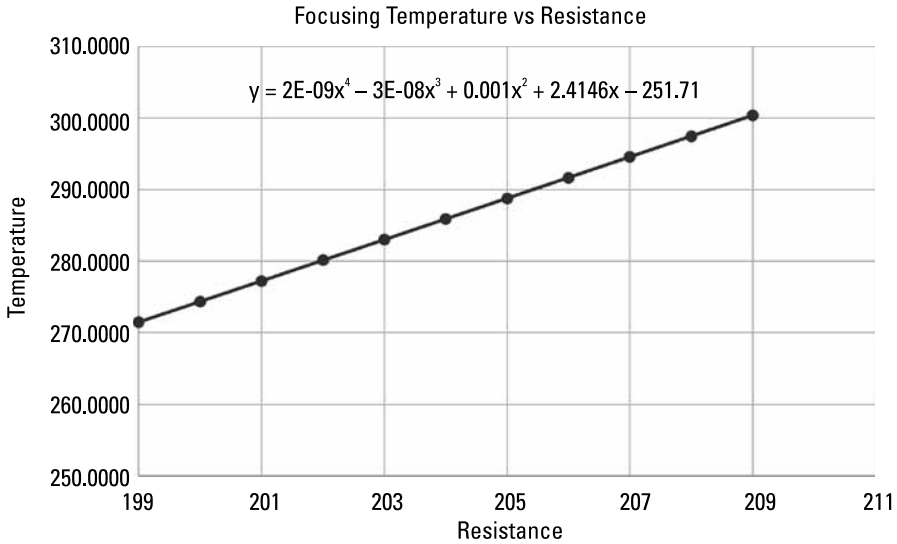



**Figure 3.32** Plotting between 269°C and 300°C to provide a more accurate result.

of the most important algorithms in the FPGA engineer's toolbox and one that few engineers are aware of, although as engineers they will almost certainly use results calculated by a scientific calculator that uses the algorithm for trigonometric and exponential functions, starting with the HP35 and continuing to present-day calculators. The real beauty of this algorithm is that it can be implemented with a very small FPGA footprint and requires only a small look-up table, along with logic to perform shifts and additions; importantly, the algorithm requires no dedicated multipliers or dividers to implement.

This algorithm is one of the most useful for DSP and industrial and control applications. Depending upon its mode and configuration, the CORDIC implements some very useful mathematical functions. The CORDIC algorithm can operate in one of three configurations: linear, circular, or hyperbolic. Within each of these configurations, the algorithm functions in one of two modes: rotation or vectoring (see Table 3.5). In the rotation mode, the input vector is rotated by a specified angle, while in the vectoring mode, the algorithm rotates the input vector to the x axis while recording the angle of rotation required.

Additionally, there are other functions that can be derived from the CORDIC outputs. In many cases, these could even be implemented by the use of another CORDIC in a different configuration.

$$\text{Tan} = \text{Sin}/\text{Cos}$$
$$\text{Tan}\,H = \text{Sinh}/\text{Cosh}$$
$$\text{Exponential} = \text{Sinh} + \text{Cosh}$$
$$\text{Natural Logarithm} = 2 * \text{Arc Tan}\,H^{\text{Note 1}}$$
$$\text{SQR} = \left(X^2 - Y^2\right)^{0.5\ \text{Note 2}}$$

Note 1: where $X$ = Argument +1 $Y$ = Argument – 1.
Note 2 where $X$ = Argument +0.25 $Y$ = Argument – 0.25.

The unified CORDIC algorithm can be seen below; it covers all three configurations. The algorithm has three inputs X, Y, and Z; how these are initialized at startup depends upon the mode of operation (vectoring or rotation) in accordance with Table 3.6.

$$X_{i+1} = X_i - m * Y_i * d_i * 2^i$$
$$Y_{i+1} = Y_i + X_i * d_i * 2^i$$
$$Z_{i+1} = Z_i - d_i * e_i$$

**Table 3.5** Functions Resulting from CORDIC Configurations and Mode

| *Configuratoin* | *Rotation* | *Vectoring* |
|---|---|---|
| Linear | Op Y = X * Y | Op Z = X/Y |
| Hyperbolic | Op X = CosH(X)<br>Op Y = SinH(Y) | Op Z = Arc TanH |
| Circular | Op X = Cos(X)<br>Op Y = Sin(Y) | Op Z = ArcTan(Y)<br>Op X = SQR($X^2 + Y^2$) |

**Table 3.6** CORDIC Angle of Rotation

| *Configuration* | $e_i$ |
|---|---|
| *Linear* | $2^{-i}$ |
| *Hyperbolic* | Arc TanH($2^{-i}$) |
| *Circular* | Arc Tan($2^{-i}$) |

where $m$ defines the configuration for either hyperbolic ($m = -1$), linear ($m = 0$), or circular ($m = 1$) correspondingly the value of $e_i$ as the angle of rotation changes depending upon the configuration. The value of $e_i$ is normally implemented as a small look-up table within the FPGA (Table 3.6).

$d_i$ is the direction of rotation that depends upon the mode of operation for rotation mode $d_i = -1$ if $Z_i < 0$, else +1, while in the vectoring mode $d_i = +1$, if $Y_i < 0$, else –1.

When configured in either circular or hyperbolic using rotation mode, the output results will have gain that can be precalculated using the number of rotations defined using the equation.

$$A_n = \sum\sqrt{(1+2^{-2i})}$$

This gain is typically fed back into the initial setting of the algorithm to remove the need for post-scaling of the result (Table 3.7).

While the algorithm presented above is particularly important to the design engineer, it has to be noted that the CORDIC algorithm only operates within a strict convergence zone that may require the engineer to perform some prescaling to ensure the algorithm performs as expected. It is worth noting that the algorithm will get more accurate the more iterations (serial) or stages (parallel) the engineer decides to implement. A general rule of thumb is that for $n$ bits of precision, $n$ iterations or stages are required. However, all of this is easily modeled in simple tools such as Excel or MATLAB prior to cutting code to ensure the accuracy is obtained with the selected iterations.

**Table 3.7** Initialization Settings Depending upon Mathematical Operation as Demonstrated in Table 3.5

| *Mode* | *Y* | *X* | *Z* |
|---|---|---|---|
| Circular Rotation | $A_n$ | 0 | Argument Z |
| Circular Vectoring | 1 | Argument X | 0 |
| Hyperbolic Rotation | $A_n$ | 0 | Argument Z |
| Hyperbolic Vectoring | Argument + 1 | Argument – 1 | 0 |
| Linear Rotation | 0 | Argument X | Argument Z |
| Linear Vectoring | Argument Y | Argument X | 0 |

## 3.13 Convergence

The CORDIC algorithm as defined will only converge (work) across a limited range of input value. For circular configurations of CORDIC algorithms, convergence is guaranteed for the angles below the sum of the angles in the look-up table (i.e., between –99.7° and 99.7°). For angles outside of this, the engineer must use a trigonometric identity to translate one within. This is also true for convergence within the linear configuration. However, in the hyperbolic mode to gain convergence, certain iterations must be repeated (4, 13, 40, $K$… $3K + 1$). In this case, the maximum input of $\theta$ is approximately 1.118 radians.

## 3.14 Where Are These Used?

CORDICs are used in a wide range of applications, including digital signal processing, image processing, and industrial control systems. The most basic method of using a CORDIC is to generate sine and cosine waves when coupled with a phase accumulator. The use of the algorithm to generate these waveforms can, if correctly done, result in a high spurious free dynamic range (SFDR). Good SFDR performance is required for most signal processing applications. Within the field of robotics, CORDICs are used within kinematics where the addition of coordinate values with new coordinate values can be easily accomplished using a circular CORDIC in the vectoring mode. Within the field, image processing 3-D operations such as lighting and vector rotation are the perfect candidates for implementation using the algorithm. However, perhaps the most common use of the algorithm is in implementing traditional mathematical functions as shown in Table 3.1 where multipliers, dividers, or more interesting mathematical functions are required in devices where there are no dedicated multipliers or DSP blocks. This means that CORDICs are used in many small industrial controllers to implement mathematical transfer functions, with true RMS measurement being one example. CORDICs are also being used in biomedical applications to compute fast Fourier transforms (FFTs) to analyze the frequency content of many physiological signals. In this application, along with performing the traditional mathematical functions, the CORDIC is used to implement the FFT twiddle factors.

## 3.15 Modeling in Excel

One of the simplest methods of modeling your CORDIC algorithm before cutting code is to put together a simple Excel spreadsheet that allows you to model the number of iterations and gain ($A_n$) initially using a floating-point number system and later using a scaled fixed-point system providing a reference for verification of the code during simulation (Table 3.8).

As can be seen from the Excel implementation, the initial X input is set to $A_n$ to reduce the need for postprocessing the result, and the initial argument is set in Z that is defined in radians, as are the results.

**Table 3.8** Excel Model of CORDIC Configured for Circular Rotation

| Iteration | x | y | z | Direction | Gain |
|---|---|---|---|---|---|
| Initial | 0.607253 | 0.000000 | 0.017453 | | |
| 0 | 0.607253 | 0.607253 | -0.767945 | 1 | 0.707107 |
| 1 | 0.910879 | 0.303626 | -0.304298 | -1 | 0.894427 |
| 2 | 0.986786 | 0.075907 | -0.059319 | -1 | 0.970143 |
| 3 | 0.996274 | -0.047442 | 0.065036 | -1 | 0.992278 |
| 4 | 0.999239 | 0.014826 | 0.002617 | 1 | 0.998053 |
| 5 | 0.998776 | 0.046052 | -0.028623 | 1 | 0.999512 |
| 6 | 0.999496 | 0.030446 | -0.012999 | -1 | 0.999878 |
| 7 | 0.999734 | 0.022637 | -0.005186 | -1 | 0.999969 |
| 8 | 0.999822 | 0.018732 | -0.001280 | -1 | 0.999992 |
| 9 | 0.999859 | 0.016779 | 0.000673 | -1 | 0.999998 |
| 10 | 0.999842 | 0.017756 | -0.000304 | 1 | 1 |
| 11 | 0.999851 | 0.017268 | 0.000185 | -1 | 1 |
| 12 | 0.999847 | 0.017512 | -0.000060 | 1 | 1 |
| 13 | 0.999849 | 0.017390 | 0.000063 | -1 | 1 |
| 14 | 0.999848 | 0.017451 | 0.000001 | 1 | 1 |
| | | | | | |
| | Cos | Sin | | $A_n$ | 0.607253 |
| CORDIC | 0.999848 | 0.017451 | | | |
| Actual | 0.999848 | 0.017452 | | | |

## 3.16 Implementing the CORDIC

Unless there is a good reason not to, the simplest method of implementing a CORDIC algorithm within an FPGA is to utilize a tool such as the Xilinx Core Generator. The Core Generator provides a comprehensive interface allowing the engineer to define the exact functionality of the CORDIC (rotate, vector).

Unfortunately, the Core Generator does not provide options for working with CORDICs within the linear mode (the Core Generator does provide separate cores to perform these functions). However, the VHDL code required to implement the algorithm can be written in a very few lines, as the simple example of a circular implementation below shows. There are two basic topologies for implementing a CORDIC in an FPGA, using either a state machine-based approach or a pipelined approach. If the processing time is not critical, the algorithm can be implemented as a state machine that computes one CORDIC iteration per cycle until the desired number of cycles has been completed. If a high calculation speed is required, then a parallel architecture is more appropriate. This code implements a 15-stage parallel CORDIC operating within the rotation mode. It uses a simple look-up table of ArcTan, coupled with a simple array structure to implement the parallel stages.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;


ENTITY synth_cordic IS PORT(


clk   : IN std_logic;
resetn : IN std_logic;
```

```
z_ip  : IN std_logic_vector(16 DOWNTO 0); --1,16
x_ip  : IN std_logic_vector(16 DOWNTO 0); --1,16
y_ip  : IN std_logic_vector(16 DOWNTO 0); --1,16
cos_op : OUT std_logic_vector(16 DOWNTO 0); --1,16
sin_op : OUT std_logic_vector(16 DOWNTO 0)); --1,16
END ENTITY synth_cordic;


ARCHITECTURE rtl OF synth_cordic IS


TYPE signed_array IS ARRAY (natural RANGE <> ) OF signed(17 DOWN-
TO 0);


--ARCTAN Array format 1,16 in radians
CONSTANT tan_array : signed_array(0 TO 16) := (to_
signed(51471,18), to_signed(30385,18), to_signed(16054,18),to_
signed(8149,18), to_signed(4090,18), to_signed(2047,18),
to_signed(1023,18), to_signed(511,18), to_signed(255,18),
to_signed(127,18), to_signed(63,18), to_signed(31,18), to_
signed(15,18), to_signed(7,18),to_signed(3,18), to_signed(1,18),
to_signed(0, 18));


SIGNAL x_array : signed_array(0 TO 14) := (OTHERS => (OTHERS
=>'0'));
SIGNAL y_array : signed_array(0 TO 14) := (OTHERS => (OTHERS
=>'0'));
SIGNAL z_array : signed_array(0 TO 14) := (OTHERS => (OTHERS
=>'0'));


BEGIN


--convert inputs into signed format


PROCESS(resetn, clk)
BEGIN
IF resetn = '0' THEN
x_array <= (OTHERS => (OTHERS => '0'));
z_array <= (OTHERS => (OTHERS => '0'));
y_array <= (OTHERS => (OTHERS => '0'));
ELSIF rising_edge(clk) THEN
IF signed(z_ip)< to_signed(0,18) THEN
x_array(x_array'low) <= signed(x_ip) + signed('0' & y_ip);
y_array(y_array'low) <= signed(y_ip) - signed('0' & x_ip);
z_array(z_array'low) <= signed(z_ip) + tan_array(0);
ELSE
x_array(x_array'low) <= signed(x_ip) - signed('0' & y_ip);
y_array(y_array'low) <= signed(y_ip) + signed('0' & x_ip);
z_array(z_array'low) <= signed(z_ip) - tan_array(0);
END IF;
FOR i IN 1 TO 14 LOOP
IF z_array(i-1) < to_signed(0,17) THEN
```

```
x_array(i) <= x_array(i-1) + (y_array(i-1)/2**i);
y_array(i) <= y_array(i-1) - (x_array(i-1)/2**i);
z_array(i) <= z_array(i-1) + tan_array(i);
ELSE
x_array(i) <= x_array(i-1) - (y_array(i-1)/2**i);
y_array(i) <= y_array(i-1) + (x_array(i-1)/2**i);
z_array(i) <= z_array(i-1) - tan_array(i);
END IF;
END LOOP;
END IF;
END PROCESS;
cos_op <= std_logic_vector(x_array(x_array'high)(16 DOWNTO 0));
sin_op <= std_logic_vector(y_array(y_array'high)(16 DOWNTO 0));


END ARCHITECTURE rtl;
```

## 3.17 Digital Filter Design and Implementation

The Smart Sensor SoC being developed by SensorsThink interfaces with several sensors that gather real-world data. This data will be subject to noise. As such, SensorsThink engineers may want to consider using digital filters to remove noise from the signals, especially the remoteHX94 Remote Humidity and Temperature sensor, which is sampled directly by the XADC.

Digital filters are a key part of any signal-processing system, and as modern applications have grown more complex, so has filter design. FPGAs provide the ability to design and implement filters with performance characteristics that would be exceedingly difficult to re-create with analog methods. These digital filters are immune to certain issues that plague analog implementations, notably component drift and tolerances (over temperature, aging and radiation, for high-reliability applications). These analog effects significantly degrade the filter performance, especially in areas such as passband ripple.

Digital models have their own quirks. The rounding schemes used within the mathematics of the filters can be a problem, as these rounding errors will accumulate, impacting performance by, for example, raising the noise floor of the filter. The engineer can fall back on a number of approaches to minimize this impact, and schemes such as convergent rounding will provide better performance than traditional rounding. In the end, rounding-error problems are far less severe than those of the analog component contribution.

One of the major benefits of using an FPGA as the building block for a filter is the ability to easily modify or update the filter coefficients late in the design cycle with minimal impact, should the need for performance changes arise due to integration issues or requirement changes.

### 3.17.1 Filter Types and Topologies

Most engineers familiar with digital signal processing will be aware that there are four main types of filters. Lowpass filters only allow signals below a predetermined cutoff frequency to be output.

Highpass filters are the inverse of the lowpass and will only allow through frequencies above the cutoff frequency. Bandpass filters allow a predetermined bandwidth of frequencies, preventing other frequencies from entering. Finally, band-reject filters are the inverse of the bandpass variety and therefore reject a predetermined bandwidth, while allowing all others to pass through.

Most digital filters are implemented by one of two methods: finite impulse response (FIR) and infinite impulse response (IIR). Let's take a closer look at how to design and implement FIR filters, which are also often called windowed-sinc filters. So why are we focusing upon FIR filters? The main difference between the two filter styles is the presence or lack of feedback. The absence of feedback within the FIR filter means that for a given input response, the output of the filter will eventually settle to zero. For an IIR filter subject to the same input that contains feedback, the output will not settle back to zero. The lack of feedback within the filter implementation makes the FIR filter inherently stable, as all the filter's poles are located at the origin. The IIR filter is less forgiving. Its stability must be carefully considered as you are designing it, making the windowed sinc filter easier for engineers new to DSP to understand and implement. If you were to ask an engineer to draw a diagram of the perfect lowpass filter in the frequency domain, most would produce a sketch similar to that shown in Figure 3.33.

The frequency response shown in Figure 3.33 is often called the brick-wall filter. That is because the transition from passband to stopband is very abrupt and much sharper than can realistically be achieved. The frequency response also exhibits other perfect features, such as no passband ripple and perfect attenuation within the stopband. If you were to extend this diagram such that it was symmetrical around 0 Hz extending out to both +/–FS Hz (where FS is the sampling frequency) and perform an inverse discrete Fourier transform (IDFT) upon the response, you would obtain the filter's impulse response, as shown in Figure 3.34.

This is the time-domain representation of the frequency response of the perfect filter shown in Figure 3.33, often called the filter kernel. It is from this response that FIR or windowed-sinc filters get their name, as the impulse response is what is achieved if you plot the sinc function.

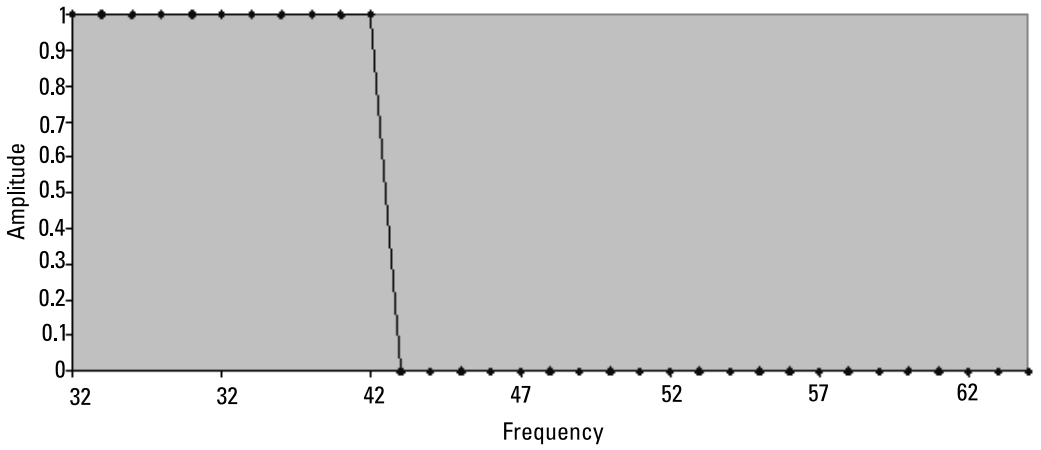



**Figure 3.33** Ideal lowpass filter performance.

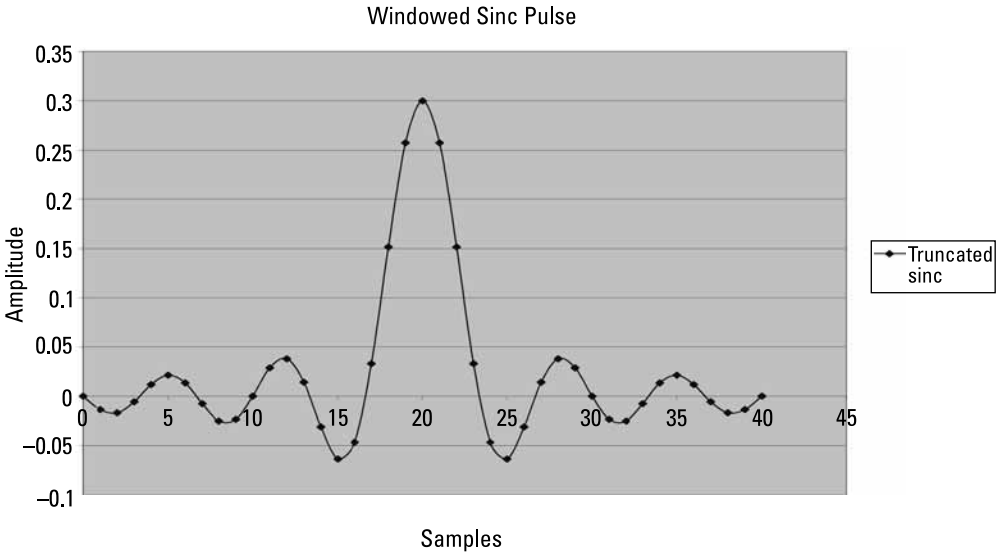



**Figure 3.34** IDFT or impulse response of the perfect lowpass filter.

$$h[i] = \frac{\sin(2 * PI * Fc * i)}{i * PI}$$

Combined with the step response of the filter, these three responses—frequency, impulse, and step—provide all the information on the filter performance that you need to know to demonstrate that the filter under design will meet the requirements placed upon it.

### 3.17.2 Frequency Response

The frequency response is the traditional image engineers consider when thinking about a filter. It demonstrates how the filter modifies the frequency-domain information.

The frequency response allows you to observe the cutoff frequency, stopband attenuation, and passband ripple. The roll-off between passband and stopband—often called the transition band—is also apparent in this response. Ripples in the passband will affect the signals being filtered. The stopband attenuation demonstrates how much of the unwanted frequencies remain within the filter output.

### 3.17.3 Impulse Response

It is from the impulse response that the coefficients for your filter are abstracted. However, to achieve the best performance from your filter, the standard practice is to use a windowing function. Windowing is the technique of applying an additional mathematical function to a truncated impulse response to reduce the undesired effects of truncation. Figure 3.33 demonstrates the impulse response extending out infinitely with ripples that, although they diminish significantly in amplitude, never settle at zero. Therefore, you must truncate the impulse response to $N + 1$ coefficients

chosen symmetrically around the center main lobe, where $N$ is the desired filter length (please remember that $N$ must be an even number). This truncation affects the filter's performance in the frequency domain due to the abrupt cutoff of the new, truncated impulse response. If you were to take a discrete Fourier transform (DFT) of this truncated impulse response, you would notice ripples in both the passband and stopband along with reduced roll-off performance. Therefore, it is common practice to apply a windowing function to improve the performance.

### 3.17.4 Step Response

The step response, which is obtained by integrating the impulse response, demonstrates the time-domain performance of the filter and how the filter itself modifies this performance. The three parameters of importance when you are observing the step response are the rise time, overshoot, and linearity. The rise time is the number of samples that it takes to rise between 10% and 90% of the amplitude levels, demonstrating the speed of the filter. To be of use within your final system, the filter must be able to distinguish between events in the input signal; therefore, the step response must be shorter than the spacing of events in the signal. Overshoot is the distortion that the filter adds to the signal as it is processing it. Reducing the overshoot in the step response allows you to determine if the signal distortion results from either the system or the information that it is measuring. Overshoot reduces the uncertainty of the source of the distortion, degrading the final system performance, and possibly means that the system does not meet the desired performance requirements. If the signal's upper and lower halves are symmetrical, the phase response of the filter will have a linear phase, which is needed to ensure the rising edge and falling edge of the step response are the same. It is worth nothing at this point that is very difficult to optimize a filter for good performance in both the time and frequency domains. Therefore, you must understand which of those domains contains the information that you are required to process. For FIR filters, the information required is within the frequency domain. Therefore, the frequency response tends to dominate.

### 3.17.5 Windowing the Filter

Using a truncated impulse response will not provide the best-performing digital filter, as it will demonstrate none of the ideal characteristics. For this reason, designers use windowing functions to improve the passband ripple, roll-off, and stopband attenuation performance of the filter. There are many window functions that you can apply to the truncated sinc, for example, Gaussian, Bartlett, Hamming, Blackman, and Kaiser. However, two of the most popular window functions are the Hamming and Blackman. Let us explore them in more detail. Both windows, when applied, reduce the passband ripple and increase the roll-off and attenuation of the filter. Figure 3.35 shows the impulse and frequency responses for a truncated sinc and with Blackman and Hamming windows applied.

As you can see, both windows significantly improve the passband ripple. The roll-off of the filter is determined not only by the window but also by the length highpass filter; you can then use combinations of the two to produce band-reject and bandpass filters. Let us first examine how to convert a lowpass filter into a

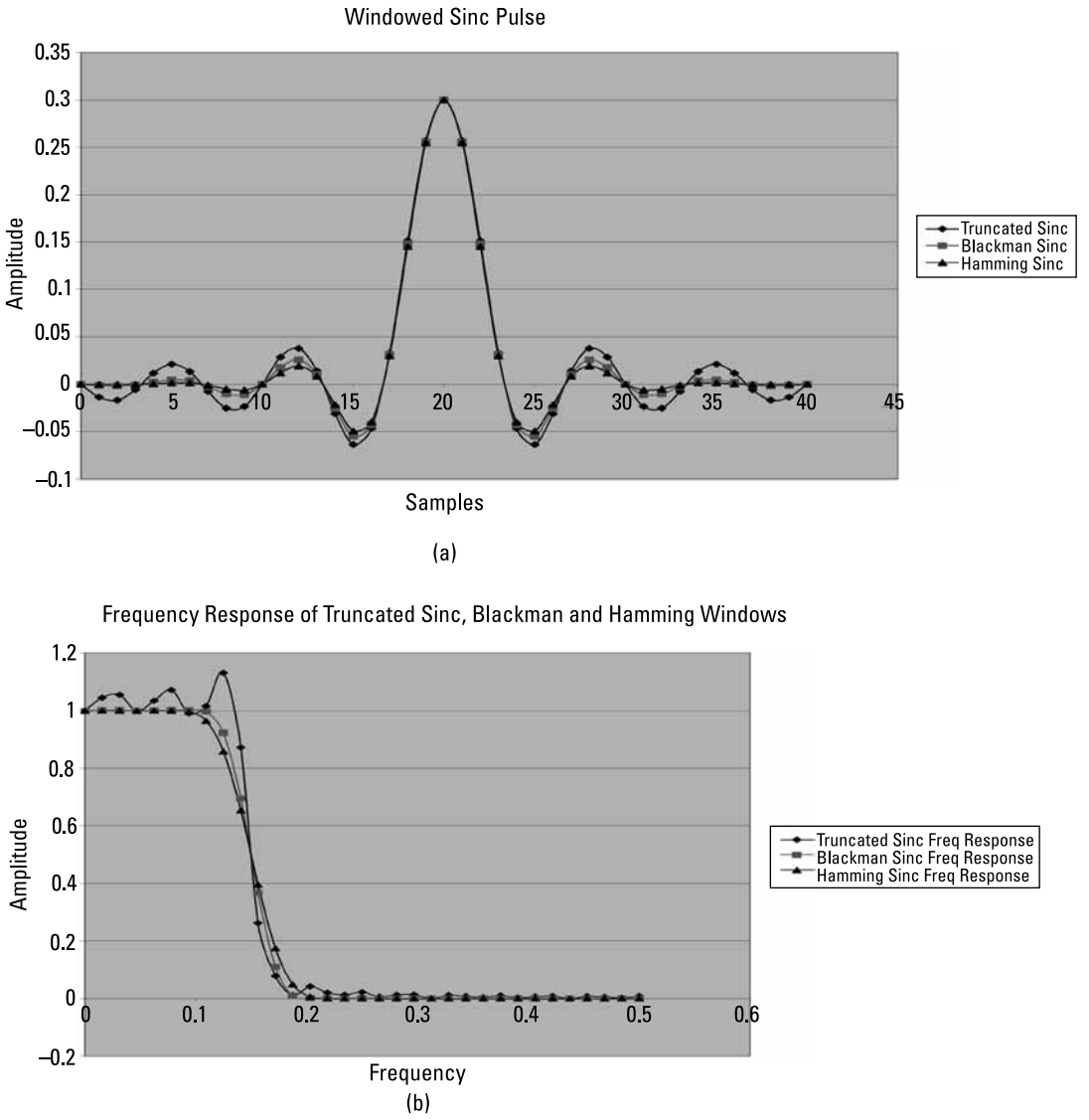



**Figure 3.35** Windowed impulse and frequency responses.

highpass one. The simplest method, called spectral inversion, changes the stopband into the passband and vice versa. You perform spectral inversion by inverting each of the samples, while at the same time adding one to the center sample. The second method of converting to a highpass filter is spectral reversal, which mirrors the frequency response; to perform this operation, simply invert every other coefficient. Having designed the lowpass and highpass filters, it is easy to make combinations that you can use to generate bandpass and band-reject filters. To generate a band-reject filter, just place a highpass and a lowpass filter in parallel with each other and sum together the outputs. Meanwhile, you can assemble a bandpass filter by placing a lowpass and a highpass filter in series with each other.

## 3.18 Fast Fourier Transforms

The SensorsThink engineering team will also be working with ADCs and filters; several of these may require verification in the frequency domain during testing. As such, it is important that the SensorsThink development team understands how to operate in the frequency domain. For many engineers, working in the frequency domain does not come as naturally as working within the time domain; however, the engineer needs the ability to work within both domains to unlock the true potential of FPGA-based solutions.

### 3.18.1 Time or Frequency Domain?

As engineers, we can analyze and manipulate signals in either the time or frequency domains, knowing when to do which forms one of the core reasons that an engineer is required on a project.

The time domain enables the engineer to analyze how a signal changes over time. Typically, in electronic systems, the signal in question is a changing voltage, current, or frequency, which has been output by a sensor or generated by another part of the system. Within this domain, we can measure a signal's amplitude, frequency, and period and more interesting parameters such as the rise and fall times of the signal. If the engineer wishes to observe a time-domain signal in a laboratory environment, it is common to use an oscilloscope or logic analyzer.

However, there are parameters of a signal that require analysis within the frequency domain to access the information contained within. These parameters are present within the frequency domain, where we can identify the frequency components of the signal, their amplitudes and the phase of each frequency. Working within the frequency domain also makes it much simpler to manipulate signals due to the ease with which convolution can be performed in the frequency domain. As with the time-domain analysis, if the engineers wish within the laboratory environment to observe a frequency-domain signal, they can use a spectrum analyzer.

For some applications, the engineer will wish to work within the time domain, for example, monitoring systems that monitor the voltage or temperature of a system. While noise may be an issue, taking an average of several samples will in many cases be sufficient. However, for other applications, it is preferable to work within the frequency domain, for example, signal processing applications that require the filtering of one signal from another or to separate it from a noise source.

Working within the time domain requires less postprocessing on the quantized digital signal, while working within the frequency domain first requires the application of a transform to the quantized data to convert from the time domain. Similarly, to output the postprocessed data from the frequency domain, we need to perform the inverse transform again back to the time domain.

## 3.19 How Do We Get There?

Depending upon the type of the signal, repetitive or nonrepetitive, discrete or nondiscrete, there are several methods that one can use to covert between the time and frequency domains, for instance, Fourier series, Fourier transforms, or Z

transforms. Within electronic signal processing and FPGA applications in particular, the engineer is most often interested in one transform, the discrete Fourier transform (DFT), which is a subset of the Fourier transform. The DFT is used by the engineer to analyze signals that are periodic and discrete (i.e., they consist of several *n* bit samples evenly spaced at a sampling frequency, which in many applications are supplied by an ADC within the system).

At its most simple explanation, DFT decomposes the input signal into to two output signals that represent the sine and cosine components of that signal. Thus, for a time-domain sequence of *N* samples, the DFT will return two groups of *N*/2 + 1 cosine and sine wave samples, respectively, referred to as the real and imaginary components as shown in Figure 3.36. The real and imaginary sample width will also be *n*/2 for an input signal width of *n* bits.

The algorithm to calculate the DFT is straightforward; we can use the equation shown here:

$$ReX[k] = \sum_{i=0}^{N-1} x[i]\cos\left(\frac{2\pi ki}{N}\right)$$
$$ImX[k] = \sum_{i=0}^{N-1} x[i]\sin\left(\frac{2\pi ki}{N}\right)$$

where $x[i]$ is the time-domain signal, $i$ ranges from 0 to $N - 1$, and $k$ ranges from 0 to $N/2$. The algorithm above is called the correlation method, and what it does is multiply the input signal with the sine or cosine wave for that iteration to determine its amplitude.

We will wish at some point in our application to transform back from the frequency domain into the time domain, as such we can use the synthesis equation, which combines the real and imaginary waveforms to recreate a time-domain signal as such:

$$x[i] = \sum_{k=0}^{N/2} Re\bar{X}[k]\cos\left(\frac{2\pi ki}{N}\right) + \sum_{k=0}^{N/2} Im\bar{X}[k]\sin\left(\frac{2\pi ki}{N}\right)$$

However, $Re\bar{X}$ and $Im\bar{X}$ are scaled versions of the cosine and sine waves as such we need to scale them; hence, $Im\bar{X}[k]$ or $Re\bar{X}[k]$ are divided by $N/2$ to determine

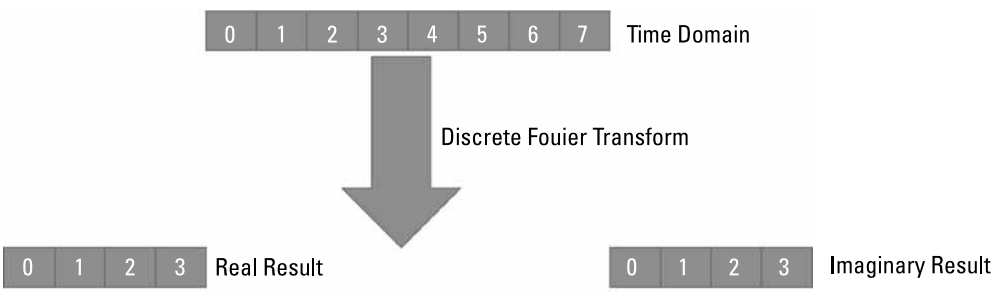



**Figure 3.36** N bits in the time domain to n/2 real and imaginary parts in the frequency domain.

the values for $Re\bar{X}$ and $Im\bar{X}$ in all cases except when $Re\bar{X}[0]$ and $Re\bar{X}[N/2]$; in this case, they are divided by $N$.

For obvious reasons, this is called the IDFT. Having explained the algorithms used for determining the DFT and IDFT, it may be useful to know what we can use these for.

We can use tools such as Octave, MATLAB, and even Excel to perform DFT calculations upon captured data and many lab tools such as oscilloscopes can perform DFT upon request.

However, before we progress, it is worth pointing out that both the DFT and IDFT above are referred to as real DFT and real IDFT in that the input is a real number and not complex; why we need to know this will become apparent.

### 3.19.1 Where Do We Use These?

From telecommunications to image processing, radar, and sonar, it is hard to think of a more powerful and adaptable analysis technique that we can implement within our FPGA than the Fourier transform. Indeed, the DFT forms the basis of one of the most used FPGA-based applications in being the basis for the generation of the coefficients for the finite input response filter.

However, its use is not just limited to filtering; the DFT and IDFT are used in telecommunications processing to perform channelization and recombination of the telecommunication channels. In spectral monitoring applications, they are used to determine what frequencies are present within the monitored bandwidth, while in image processing the DFT and IDFT are used to perform convolution of images with a filter kernel to perform, for example, image pattern recognition. All the above are typically implemented using a more efficient algorithm to calculate the DFT than that shown above.

All told, the ability to understand and implement a DFT within your FPGA is a skill that every FPGA developer should have.

### 3.19.2 FPGA-Based Implementation

The implementation of the DFT and IDFT as described above is often implemented as a nested loop each performing $N$ calculations. As such, the time taken to implement the DFT calculation is

$$DFTtime = N * N * Kdft$$

where $Kdft$ is the processing time for each iteration to be performed; as such, this can become quite time-consuming to implement. As a result of this, DFTs within FPGAs are normally implemented using an algorithm called the fast Fourier transform to calculate the DFT. This algorithm has often been called the "the most important algorithm of our lifetime" as it has had such an enabling impact on many industries. The FFT differs slightly from the DFT algorithms explained previously, in that it calculates the complex DFT, that is, it expects real and imaginary time-domain signals and produces results in the frequency domain that are $n$ bits wide as opposed to $n/2$. This means that when we wish to calculate a real DFT, we must

first address this by setting the imaginary part to zero and moving the time-domain signal into the real part. If we wish to implement an FFT within our Xilinx FPGA, we have two options; we can write one from scratch using the HDL of our choice or use the FFT IP provided within the Vivado IP catalog or another source. Unless there are pressing reasons not to use the IP, the reduced development time resulting from using the IP core should drive its selection and use.

The basic approach of the FFT is that it decomposes the time-domain signal into several single-point time-domain signals. This is often called bit reversal as the samples are reordered. The number of stages that it takes to create these single-point time-domain signals is calculated by Log2 $N$ where $N$ is the number of bits, if a bit reversal algorithm is not used as a short cut.

These single-point, time-domain signals are then used to calculate the frequency spectra for each of these points. This is straightforward as the frequency spectra are equal to the single-point time domain.

It is in the recombination of these single frequency points that the FFT algorithm gets complicated. We must recombine these spectra points one stage at a time, which is the opposite of the time-domain decomposition; thus, it will again take Log2 $N$ stages to recreate the spectra. This is where the famous FFT butterfly comes into play.

When compared with the DFT execution time, the FFT takes

$$FFTtime = Kfft * N \log 2N$$

which results in significant improvements in execution time to calculate a DFT.

When implementing an FFT within our FPGA, we must also consider the size of the FFT; this will determine the noise floor below which we cannot see signals of potential interest. The FFT size will also determine the spacing of the frequency bins; the equation here is used:

$$FFTNoise\ Floor(dB) = 6.02n + 1.77 + 10\log 10\left(\frac{FFTsize}{2}\right)$$

where $n$ is then number of quantized bits within the time domain and FFTsize is the FFT size. For FPGA-based implementation, this is normally a power of 2, for example, 256, 512, and 1,024. The frequency bins will be evenly spaced at

$$Bin\ Width = \frac{\frac{Fs}{2}}{FFTsize}$$

For a quite simple example, an FS of 100 MHz with an FFT size of 128 would have a frequency resolution of 0.39 Hz. This means that frequencies within 0.39 Hz of each other cannot be distinguished.

### 3.19.3 Higher-Speed Sampling

Many applications for FFTs within higher-performance systems and FPGAs are based upon applications that operate at very high frequencies, which can present their own challenges for the engineer implementing them.

At high frequencies, the Nyquist sample rate (sampling at least 2 samples per cycle) simply cannot be maintained as such a different approach must be undertaken. An example of this would be using an ADC to sample a 3-GHz full-power-bandwidth analog input with a 2.5-GHz sample rate. Using Nyquist rate criteria, signals above 1.25 GHz will be aliased back into the first Nyquist zone to be of use. These aliased images are harmonic components of the fundamental signal and thus contain the same information as the nonaliased signal as shown in Figure 3.37.

To determine the resultant frequency location of the harmonic or harmonic content, you can use the following algorithm:

```
Fharm=N ×Ffund
IF (Fharm=Odd Nyquist Zone)
Floc=Fharm Mod Ffund
Else
Floc=Ffund-(Fharm Mod Ffund)
End
```

where *N* is the integer for the harmonic of interest.

Continuing our example further with a sample rate of 2,500 MHz and a fundamental of 1,807 MHz, there will be a harmonic component at 693 MHz within the first Nyquist zone, which we can further process within our FFT.

Having explained a little about the frequency spectrum, the way you interface these devices to the FPGA is another important factor, as it is not possible for the data from the ADC to be received at FS/2 where in the example above the sampling frequency is 2.5 Gbps. For this reason, high-performance data converters use multiplexed digital inputs and outputs that operate at a lower data rate with respect to the converter's sample rate typically FS/4 or FS/2.

Having received the data from the FPGA in a number of data streams you may be wondering how you can process the data internally within the FPGA, if you wished to perform a DFT. One common method used for a number of applications, including telecommunication processors and radio astronomy, is to use combined or split FFT structures as shown in Figure 3.38.

While this application is more complicated than a straightforward FFT, such an approach enables the higher-speed processing to be achieved.

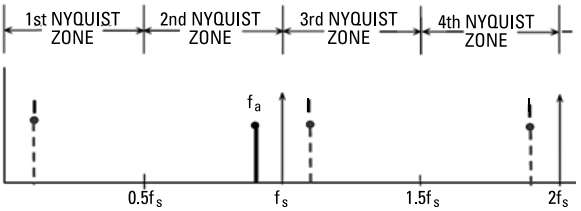



**Figure 3.37** Nyquist zones and aliasing.

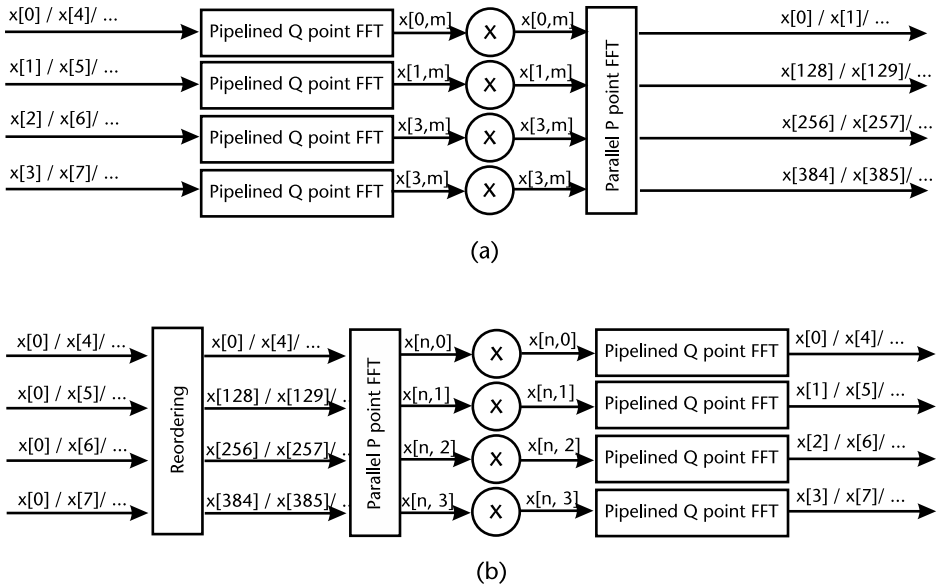



**Figure 3.38** Split and combined FFT structures.

## 3.20 Working with ADC and DAC

Commonly, FPGA-based systems must interface with the real world having performed the task that it was designed to do. Sadly, the real world tends to function around an analog signal as opposed to digital signals; therefore conversion is required to and from the digital domain from the analog domain. As with selecting the correct FPGA for the job, the engineers are faced with a multitude of choices when selecting the correct ADC or DAC for their system. The first thing for the engineers to establish is the sampling rate required for the signal to be converted, as this will drive not only the converter selection but also impact the FPGA selection as well to ensure that the processing speed and logic footprint required can be addressed. As all engineers should be aware, the sampling rate of the converter needs to be at least twice that of the signal being sampled. Therefore, if the engineers require sampling a signal at 50 MHz, the sampling rate needs to be at least 100 MHz; otherwise, the converted signal will be aliased back upon itself and not be correctly represented. This aliasing is not always a bad thing and can be used by engineers to fold back signals into the useable bandwidth if the converter bandwidth is wide enough.

### 3.20.1 ADC and DAC Key Parameters

ADCs are implemented by many different techniques; however, some of the most common are flash, ramp, and successive approximation.

- Flash converters are known for their speed and use a series of scaled analog comparators to compare the input voltage against a reference voltage and use the outputs of these comparators to determine digital code.

- Ramp converters utilize a free-running counter connected to a DAC. The output of the DAC is used to compare against the input voltage; when the two are equal, the count is held.
- Successive approximation converters are an adaption of ramp converters and utilize a DAC and a comparator against the analog input. However, instead of counting, the SAR converter determines if the analog representation of the count is above or below the input signal, allowing a trial-and-error-based approach to determine the digital code.

DACs are also provided in several different implementations; some of the most common are binary-weighted, R-2R Ladder, and pulse width modulation.

- *Binary-weighted:* These are one of the fastest DAC architectures and sum the result of individual conversions for each logic bit; for example, a resistor-based one will switch on or out resistors depending upon the current code.
- *R-2R Ladder:* These converters use a structure of cascaded resistors of value R-2R. Due to the ease with which precision resistors can be produced and matched, these are more accurate than the binary-weighted approach.
- *Pulse width modulation:* The simplest type of DAC that passes the pulse width modulation waveform through a simple lowpass analog filter. This is commonly used in motor control but also forms the basis for delta sigma converters.

Many manufacturers of specialist devices have developed their own internal conversion architectures that provide the best possible performance in specific areas depending upon the intended use. Each of these has advantages and disadvantages relating to the speed of conversion, accuracy, and resolution of the conversion. As when selecting an FPGA, the engineer will look at the number of IO, IO standards, clock management, logic resources, and memory. Similar parameters are used to compare the performance of ADCs; these are the maximum sampling rate, the signal-to-noise ratio (SNR), SFDR, and the effective number of bits (ENOB). The sampling frequency is simple and is the maximum rate at which the ADC can sample its input. SNR represents the quantization noise; this is assumed to be uncorrelated with the input signal and can be theoretically determined using the equation:

$$\text{SNR} = 6.02N + 1.76 \text{ dB}$$

where $N$ is the resolution.

The actual SNR can be determined during system test by taking an FFT of the output and measuring between the value of the input signal and the noise floor. The SFDR is the ratio between the input signal and the next highest peak; this is usually a harmonic of the fundamental. Normally the SFDR is given in terms of dBc and will degrade as the input signal power reduces. From these measurements of the converter, the engineer can calculate the effective number of bits using the equation:

$$\text{ENOB} = (\text{SNR} - 1.77/6.02)$$

When performing this testing, the engineer must take care to ensure that the FFT used is correctly sized to ensure that the noise floor shown is correct and not inadvertently incorrect due to the size of the FFT selected. The FFT noise floor is given by

$$\text{FFT Noise Floor} = 6.02N + 1.76\ \text{dB} + 10\ \text{LOG10}(\text{FFT Size}/-)$$

These tests should be performed using a single tone test, normally a simple sine wave to reduce the complexity of the output spectrum. For the engineer to get the best results, performing coherent samples of the output is a must. Coherent sampling occurs when there are an integer number of cycles within the data window for this to be correct; then

$$\text{FS}/\text{Fin} = \text{Ncycles}/\text{FFT}$$

### 3.20.2 The Frequency Spectrum

As mentioned above, the engineer must be aware of the Nyquist criteria when implementing the system to ensure that the signal is correctly converted or quantized. This means that the signal must be sampled at least twice the maximum frequency of the signal of interest to ensure correct conversion. When signals are sampled outside of this criterion, aliasing will occur, although this can result in unwanted performance if it is not correctly understood. It is also for this reason that ADCs require anti-aliasing filters to prevent signals or noise being aliased back into the quantized signal. However, aliasing can be very useful to the engineer especially if the ADC has a wide input bandwidth. This can allow, with careful consideration, the direct conversion of signals without the need for downconverters; for this reason, the frequency spectrum is divided up to several zones. Using the information presented in Table 3.9, it is possible to alias signals from one Nyquist band to another if the converter has a wide enough bandwidth.

### 3.20.3 Communication

As with all external devices, ADCs and DACs, interfaces are provided in several different interfacing options. The most common differentiation is the interface being either parallel or serial. Typically, higher-speed devices will implement a parallel

**Table 3.9** Nyquist Zones and Aliasing

| *Nyquist Zone* | *Range Lower* | *Range Upper* | *Aliasing* |
|---|---|---|---|
| First | DC | 0.5 FFS | None |
| Second | 0.5 FS | FS | Folds |
| Third | FS | 1.5 FS | Direct |
| Fourth | 1.5 FS | 2 FS | Folds |

interface, while slower-speed devices will utilize a serial interface. However, your choice of application may drive the engineer down a particular route. It is easier, for instance, to detect a stuck at bit in a serial interface than it is a parallel one. High-speed interfaces provide multiple output buses (I and Q) or use double data rate outputs; some devices might even offer both options. This allows the data rate to be maintained while reducing the frequency of operation required for the interface. For example, an interface sampling at 600 MHz will produce an output at 300 MHz (half the sampling frequency); it is much easier to recover if the clock frequency is 75 MHz (FS/4) and there are two data buses that provide the samples from the device using double data rate (DDR), as is the case with the 10D1000ADC from National Semiconductor. This allows the relaxation of the input timings required to be achieved by the FPGA engineer. Many high-speed converters utilize LVDS signaling on their IO as the lower-voltage swing and low current reduce the coupling that can occur using other signaling standards (i.e., LVCMOS), which can affect the mixed signal performance of the converter.

### 3.20.4 DAC Filtering

Most DACs will hold the analog output until the next sampling period; this results in the engineer observing an interesting effect upon the output frequency domain. They will notice both images existing across the output spectrum and the output signal in the first Nyquist zone exhibiting a roll-off due to the SINC effect being nearly 4 dB (3.92 dB) lower at 0.5 FS. The solution to both issues is to utilize filters; the first filter can be implemented digitally within the FPGA before the DAC and can correct for the roll-off using a simple digital filter, while the anti-image filter needs to be located after the DAC output as the images are a function of the conversion.

The sinc correction filter can be implemented very simply as an FIR filter; the simplest method for the engineer to undertake in developing this filter is to plot the sinc roll-off using the equation:

$$A = \frac{\sin\frac{(\pi * Fin)}{FS}}{\frac{\pi * Fin}{FS}}$$

Create the correction factor that is the reciprocal of those calculated for the roll-off. The engineer can then take an inverse Fourier transform to obtain the coefficients needed to implement the filter. Typically, this filter can be implemented with a few taps. Table 3.10 shows the first 10 coefficients for the filter, while Figures 3.39 and 3.40 show the implementation of the filter in the frequency domain.

### 3.20.5 In-System Test

Many of these systems will require the converter to achieve specific performance characteristics for the end application (e.g., CDMA or GSM). This can require significant investment in test systems (arbitrary waveform generators, logic analyzers,

**Table 3.10** The First 10 Coefficients for a DAC Compensation FIR Filter

| *Tap* | *Coefficient* |
|---|---|
| 1 | –6.22102953898351E-003 |
| 2 | 9.56204928971727E-003 |
| 3 | –1.64864415228791E-002 |
| 4 | 3.45071042895427E-002 |
| 5 | –0.107027889432584 |
| 6 | 1.166276 |
| 7 | –0.107027889432584 |
| 8 | 3.45071042895427E-002 |
| 9 | –1.64864415228791E-002 |
| 10 | 9.56204928971727E-003 |
| 11 | –6.22102953898351E-003 |

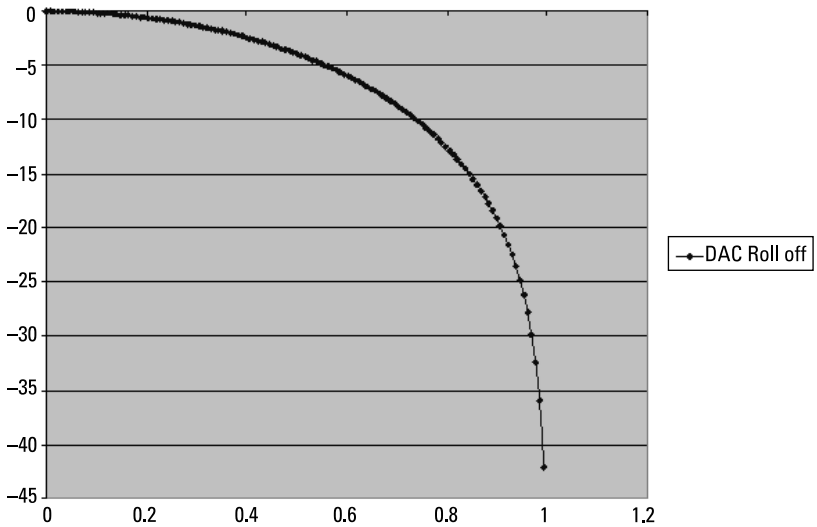



**Figure 3.39** DAC roll-off between 0 and FS.

pattern generators, spectrum analyzers) to achieve the desired testing and determine the actual performance. However, the reprogrammable flexibility of the FPGA allows specific test programs to be inserted into the device that either capture and analyze the output of an ADC or allow the stimulus of a DAC reducing hopefully the need for additional extra test equipment.

## 3.21 High-Level Synthesis

So far, the approaches to logic design for SensorsThink have focused on one of the two hardware description languages that is VHDL. Both hardware description languages, VHDL and Verilog, operate at what is commonly known as the register transfer level (RTL), as the languages are used to create registers and describes the transfer of data between registers. Both developing and verifying RTL designs is

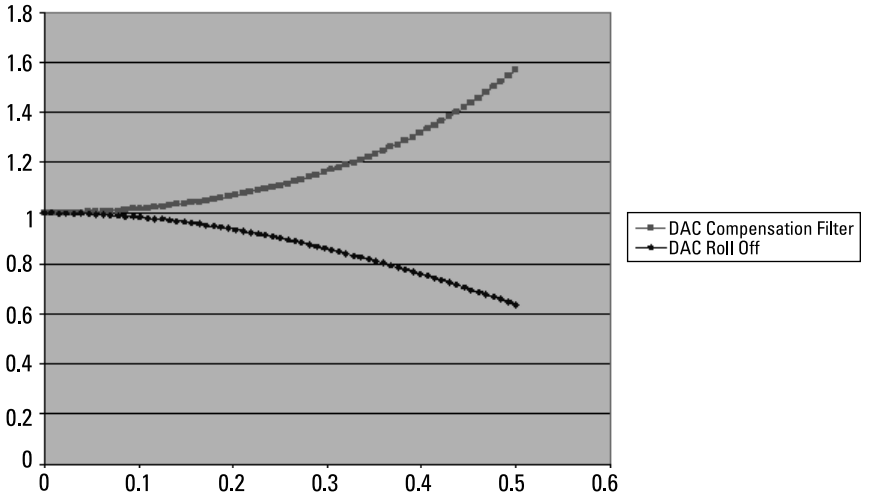



**Figure 3.40** DAC roll-off and compensation filter to FS/2.

time-consuming, especially so in verification where the design must be simulated clock cycle by clock cycle.

Increasingly, FPGA developers are using C-based High Level Synthesis (HLS) to accelerate both the design and verification cycles. This approach is something engineers at SensorsThink need to be familiar with C-based HLS translates the C-based design into Verilog and VHDL, which can then be implemented using a traditional implementation flow. Using a C-based design flow enables the developer to focus on the algorithm implementation, while verification can also use untimed C simulation. This allows both the algorithm and its verification to be developed much quicker than using a traditional RTL.

Just like when we develop with Verilog and VHDL, there is a subset of the language that can be implemented, and there is also a similar subset of C/C++ that can be implemented via HLS. The exact details vary from HLS tool to HLS tool; however, obvious elements such as system calls cannot be synthesized.

To convert an HLS design into C, the HLS tool will perform 3 stages of analysis to generate the target RTL.

1. *Scheduling:* In this phase, the untimed C operations are allocated to clock cycles by the compiler. The compiler will consider the desired clock frequency, the cost in time of the operation in the target technology to correctly schedule the operations.
2. *Binding:* Once the operations have been allocated to a clock cycle, the next stage is to assign the operations to resources available within the target device. This could be look-up tables, block RAMS, or DSP elements.
3. *Control extraction:* Once the scheduling and resource allocation has been created, the final stage is for the HLS tool to implement the necessary control structures to enable the block to be started on demand, perform memory accesses and report the result is valid. This stage will typically implement finite state machines to perform the control actions required.

The output of these 3 stages is Verilog and VHDL source code, which can be imported as an IP block into the implementation tool.

However, C does not describe parallel operations as hardware description languages do. To address this, the HLS compiler can be instructed by the developer where parallel structures exist in the source code. The user identifies parallel structures in the code using pragmas; these pragmas are used to control the compilers' implementation of the design. Example pragmas include:

- *Interval:* The number of clock cycles between the synthesized IP module being able to start processing a new input.
- *Unrolling:* Where loops are used to process data elements, by default, HLS tools will keep loops rolled up, ensuring minimal resource requirements. However, keeping loops rolled results in decreased performance as each iteration of the loop uses the same hardware and incrementing a loop counter requires at least one clock. Unrolling a loop results in the creation of hardware for each iteration of the loop significantly reducing the execution time of the loop, while at the same time increasing the required resources. This resource for performance trade-off is one of the key aspects of optimizing HLS designs.
- *Pipelining:* Pipelining is another technique that can be used to increase throughput in your HLS implementation. Pipelining is an alternative to unrolling and enables loops to be pipelined such that a new input can be processed on each clock cycle (other constraints permitting). This does not affect the latency of the pipeline, but it increases the throughput enabling the design to achieve higher performance than a nonpipelined design.
- *Memory partitioning and reshaping:* By default, arrays in C code are synthesized to the block RAMs. Block RAMs can act as bottlenecks when arrays are accessed, preventing unrolling or pipelining with an optimal interval. The bottleneck occurs as block RAMs have limited access ports that prohibit multiple accesses at the same time. Partitioning block RAMs allows a single block RAM to be split into several smaller block RAMs. The HLS tool is then able to arrange the data across the smaller block RAMs to enable multiple accesses in parallel, removing the bottleneck. In the most extreme cases, the block RAM contents can be partitioned into individual registers. Reshaping is like partitioning; however, instead of splitting the words across block RAMS, reshaping changes the size of the words stored and accessed.

Along with the optimization pragmas, HLS compilers can also use pragmas to define the interfacing standard or standards that are used in the synthesized HDL to interface with larger programmable logic design.

HLS is ideal for accelerating the development of signal processing and image processing applications, as engineers developing our IOT sensor application, we may need to perform filtering on sensor data to remove noise; this is an ideal application for HLS.

The HLS development will implement a 16-bit block average; that is, 16 elements will be sampled and the average of the 16 elements will be output. As the

IOT board application contains a Xilinx Zynq SoC, the HLS tool used for the development of this filter module is Vitis HLS.

Creating such an example in HLS is amazingly simple; we can implement this in less than 10 lines of C, as shown in Figure 3.41.

Before we can synthesize the HLS design, we need to know that the C algorithm works as expected; within Vitis HLS, we have two verification methods:

- *C simulation:* The C algorithm is executed as a C program and is tested by a C wrapper. As both elements of the verification are C-based, the simulation time is significantly reduced. C simulation is used to verify the algorithm implementation achieves requirements, before performing synthesis.
- *Cosimulation:* Combines the C test bench with the synthesized RTL to verify the behavior of the generated RTL. Cosimulation therefore uses an RTL simulator to simulate the behavior of the RTL. As this simulator will be clock cycle accurate, cosimulation may take require a longer run time than C simulation.

The C simulation created to verify the average filter can be seen in Figure 3.42; this test bench is very flexible, loading in the verification values from a text file and applying them to the average filter. The resulting average value generated by the average IP core is printed for verification.

Synthesizing this code results in an IP core that can calculate one average every 18 clock cycles and begin to process a new input every 19 clock cycles.

To ensure the most optimal implementation in our FPGA, we can use pragmas to remove potential bottlenecks in the design. Examining the design analysis view in Vitis HLS provided in Figure 3.43, we see first clock cycle used to start the IP core. The main accumulation loop takes 2 clock cycles and is run 16 times. However, as the HLS tool has been able to pipeline the loop access to memory (indicated by the II=1 in the loop descriptor) the loop is able to be completed in only 16 clock cycles, while the final clock cycle is used to output the result of the average considering the accumulated value.

Examining the analysis flow shows that the main bottleneck in being able to increase performance and reduce the number of clock cycles taken is the reading of the 16 input values from the external block RAM.

```
#include "sensor.h"

void sensor_filter( ip_type input[16], op_type *output)
{
int32_t accumulator=0;
accum: for(int i = 0; i<16; i++)
{
accumulator += (int32_t ) input[i];
}
*output = accumulator / 16;
}
```

**Figure 3.41** HLS code.

```
#include "sensor.h"

#define MAX_LINE_LENGTH 80

int main()
{
    FILE *fp;
    char line[80];
    ip_type input[16];
    op_type output;

    printf("HLS Example\n\r");
// Open a file for the output results
fp=fopen("input.dat","r");
if (fp == NULL)
printf("error opening file\n\r");
int i = 0;
while (fgets(line, MAX_LINE_LENGTH, fp) != NULL) {
       input[i] = atoi(line); //convert string to integer format
       i++;
}
fclose(fp);
//run the sensor
sensor_filter(input, &output);
printf("Result from HLS module = %d \n\r",output);
}
```

**Figure 3.42** C test bench code.

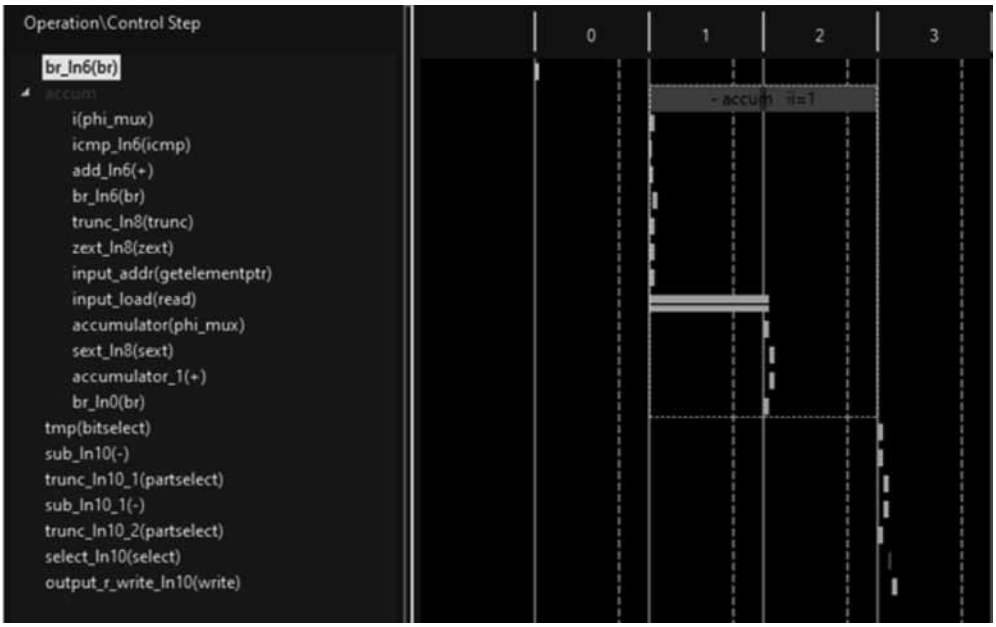



**Figure 3.43** Unoptimized HLS implementation timing analysis.

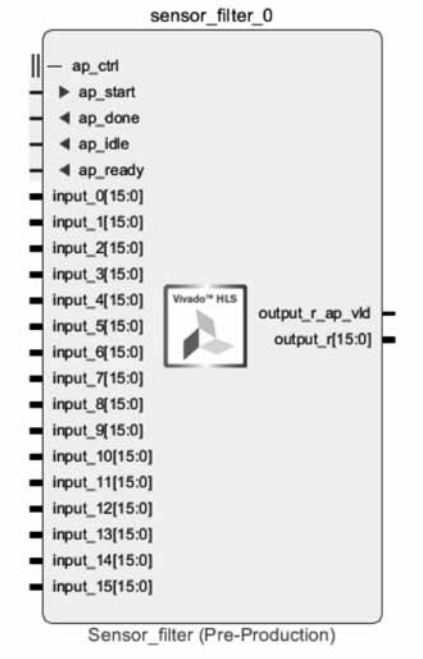



**Figure 3.44** HLS Average block IP symbol in Vivado Integrator.

Partitioning the input values to be stored as 16 discrete registers in place of the block RAM enables the loop to be unrolled and significantly reduces the latency.

Implementing the unrolling and partitioning pragmas results in the generated IP core being able to implement a new calculation every clock cycle when the clock frequency is 25 MHz.

This enables us, if necessary, to increase the throughput of the averaging IP core, should we wish to remove noise.

With the performance as desired, the IP block can then be packaged as an IP block for use within our larger IoT FPGA design if desired.

To be able to control and monitor the status of the sensor filter IP block, the HLS tool implements a control port on the synthesized RTL. The ports, idle and ready, are used to indicate that the block is ready for executing a new command. While the done signal indicates the core has completed its operation, the start signal is used to start a new operation. As the sensor filter module was optimized to be able to process a new input on each clock cycle, provided that new data is present on for the 16 averaging sample, the start pin can be tied high.

Should we want to perform more complex interfacing within an FPGA design, we can instruct the HLS compiler to implement AXI interfaces.

High-level synthesis is a tool within the FPGA designers toolbox, and, as we develop our IOT board, we should consider using HLS for signal processing applications as desired.

CHAPTER 4

# When Reliability Counts

*In the end, a well-designed system is only as good as the reliability it offers. While designers may have worked hard to meet all technical requirements and system specifications, none of it would matter if the system failure occurred as reliability concerns were not addressed.*

## 4.1 Introduction to Reliability

Reliability is defined as the ability of a product to perform as expected over certain time. In the simplest sense, reliability means how long a component or a system will deliver its expected outcome without a breakdown. In mathematical terms, reliability can be defined as the probability that a component or a system will perform its intended function for a stated period under specified operating conditions. To understand this better, let us look at the most important elements that define the reliability of a system:

- Probability is a value between 0 and 1. It is a ratio of the number of times that an event occurs (success or failure) divided by the total number trials (e.g., an embedded system with a failure probability of 0.007 means that only 7 of 1,000 items could fail to perform as stated at any given time under stated conditions).
- Performance is a set of criteria that defines what actions constitute a success and which actions constitute a failure of a component or a system. It can be very detailed or very specific in defining success or failure criterion (e.g., an embedded system with 4 sensors, a microcontroller, and a memory unit could be considered a failure if any one of the components fail or the same system could be considered as working if at least one of the sensors is working).
- Time duration for a working system. The probability of survival or failure is generally given as a function of time.
- Operating conditions such as environmental factors, humidity, vibration, shock, temperature cycle, and operational profile have an impact on the working of a system and the duration of a stated product life.

Over the past few years, penetration of embedded systems in all aspects of human life has put more focus on the importance of incorporating reliability aspect in design practices. While the issue of reliability is almost a certainty when starting any new project, the emphasis laid on reliable design could vary greatly from project to project. In some projects, reliability aspect during the system design process is given far greater importance than some other considerations, such as time to market. In general, reliability is given the highest consideration in industries where system failure could lead to loss of life (e.g., automobile or aeronautical industries) or a severe impact on the company's reputation (e.g., financial systems or large-scale factory automation). However, there are several other major industrial sectors where reliability as a design aspect is not given a similar importance as time to market. One good example is the present-day smartphone industry. Given the nature and dynamics of the market and the fast pace of technology, if the designers test the devices for all possible reliability concerns over a period of time, the technology used in the smartphone model could be out of date even before it hits the market. Keeping this in mind, designers must understand what level of emphasis is required for addressing reliability in the project that they have undertaken and what industry or end user it will serve.

While it is easy to recognize the importance of reliability in mass-produced systems that could lead to loss of life (such as automobiles and aircrafts), the reliability of a system generally has a far greater financial impact on the brand and profitability of companies engaged in other electronics industries such as mobile devices or consumer electronics.

The emphasis on reliability of a component varies greatly from one industry to another. A component deployed in a product used in the smartphone industry could have a completely different requirement for reliability in as compared to the same component being used in another industry (e.g., aerospace). As shown in Table 4.1, errors can occur during any stage of the design life cycle and can have various degrees of impact on the reliability of a system. Therefore, it is important to study reliability from a technical point of view. Before we dive into the technical definitions of reliability and several other parameters associated with it, let us consider a few scenarios to emphasize the importance of reliability in the design process.

- The component that you have designed would be used in a mass-produced automobile and you have estimated that the design failure rate is just 1 in 1,000 (99.9% reliable). Imagine that for each 1 million vehicles sold with that component, 1,000 vehicles could potentially break down and cause an accident. This automobile failure rate is just due to one component (designed by you). The overall number for system reliability will be much lower when the reliability of each individual component is considered, assuming that the average life span of an automobile is between 10 and 15 years.
- Let us now consider a case of an aircraft where 100,000 individual components, each with an average failure rate of 1 in 10,000 (99.99% reliability), are used. Let us further assume that 1,000 components can cause critical failure. In this case, the overall system failure probability can be calculated as $(0.9999)^{1,000} = 0.905$ or 90.5%. Now imagine a scenario, if this were to

**Table 4.1** Different Sources of Error in Various Design Phases

| *Design Stage* | *V-Model* | *Potential Error Source* | *Error Detection* |
|---|---|---|---|
| Specifications and design | Level 10, Level 20 | Algorithm design, formal specification | Consistency checks, simulation |
| Prototyping | Level 30, Level 40 | Algorithm design, wiring and assembly, timing component failure | Stimulus/response testing |
| Manufacturing | Level 20 | Wiring and assembly, component failure | System testing, diagnostics |
| Installation/unit testing | Level 20 | Assembly, component failure | System testing, diagnostics |
| Field operation/system validation | Level 10 | Component failure, operator errors, environmental factors | Diagnostics |

really happen, would mean that 1 of 10 aircrafts would have an accident. While there is not much that can be done to reduce the number of critical components, there is certainly a lot that can be done to increase the reliability of individual critical components that would improve the overall system reliability, even if it take a lot more years of testing, especially considering that the average life of an aircraft can be 25 to 40 years.

- Finally, let us consider a case for another mass-produced device that contains several individual components, a smartphone. Assuming that a smartphone consists of 1,000 different components and half of them are critical, in this case, the overall system failure probability can be calculated as $(0.9999)^{500} = 0.9512$ or 95.12%. Since a smartphone's failure is not considered as critical as that of an automobile or aircraft, manufacturers would be able to digest those numbers in their business model if it helps them to reduce the time to market. Considering the fact that the latest smartphones have, at best, 1 or 2 years of life span before being replaced, it would not justify years of testing to improve those system reliability numbers while risking the technology to get older by the time it is ready to be sold.

Having considered system-level reliability for mass-produced products, it should be emphasized that sometimes it is smarter to consider the expected market size of the product while finalizing design considerations. If a design is proven to be 99.99% reliable over its useful life and there will be less than 10,000 such products being built, then there is probably not much use to improve the reliability any further.

## 4.2 Mathematical Interpretation of System Reliability

As defined by the IEEE, "Reliability is the ability of an item (or component) to perform its required functions under stated conditions and for a specified period." Similarly, reliability of a system (or subsystem) can be defined as the probability of survival during the effective life span of the system. Since a system (or subsystem) consists of one or more critical components, reliability of the entire system (or

subsystem), $R_S(t)$, is calculated from the individual probability of failure (reliability) of each critical component that forms a part of the system (or subsystem). In mathematical terms, $R_S(t)$ can be calculated as follows:

$$R_S(t) = Prob\left(t_o \le t \le t_f \forall f \in F\right)$$

Here $t_o$ is the time at which the system is introduced into service and $t_f$ is the time at which the first critical fault $f$ takes place. $f$ belongs to a set of critical faults $F$ that can cause the system to fail. The failure probability $Q_S(t)$ of a system is complementary to the reliability $R_S(t)$ of the system. Hence, failure rate (or failure probability) of a system can be calculated by using the relation:

$$Q_S(t) + R_S(t) = 1$$

As mentioned above, each system (or subsystem) consists of one or more individual critical components and the reliability of the system $R_S(t)$ is dependent on the individual reliability of each critical component. Let us denote the reliability of an individual critical component as $R_C(t)$. Using the same formula as above, the failure rate (probability) for a component $Q_C(t)$ is complementary to the reliability $R_C(t)$ of the component and can be calculated by using the relation:

$$Q_C(t) + R_C(t) = 1$$

The failure rate $Q_C(t)$ generally varies throughout the life of the component; hence, it is not very useful to calculate the reliability. Failure of a system can be defined as a moment of time at which the system stops to fulfill its specified function. Mathematically, failure rate of a system is defined as:

$$Failure\,Rate = \frac{Total\,number\,of\,items\,failed}{Total\,Operating\,Time}$$

### 4.2.1 The Bathtub Curve

The failure rate is of a system or a component is expected to vary over the expected life of a product and is typically represented by a bathtub curve as shown in Figure 4.1. The bathtub model depicts the failure rate of a component or a system during three key phases: burn-in, useful life, and phase-out.

It is a standard industry practice to use the bathtub model for your system and use it as a reference for your failure calculations. A summary of failure rate causes and possible remedies during different phases of the bathtub model is presented in Table 4.2.

### 4.2.2 Failure Rate ($\lambda$)

In most practical systems, the failure rate is expressed using a different variable ($\lambda$) that represents failure rates and how many failures occur every $10^9$ hours. $\lambda$ is

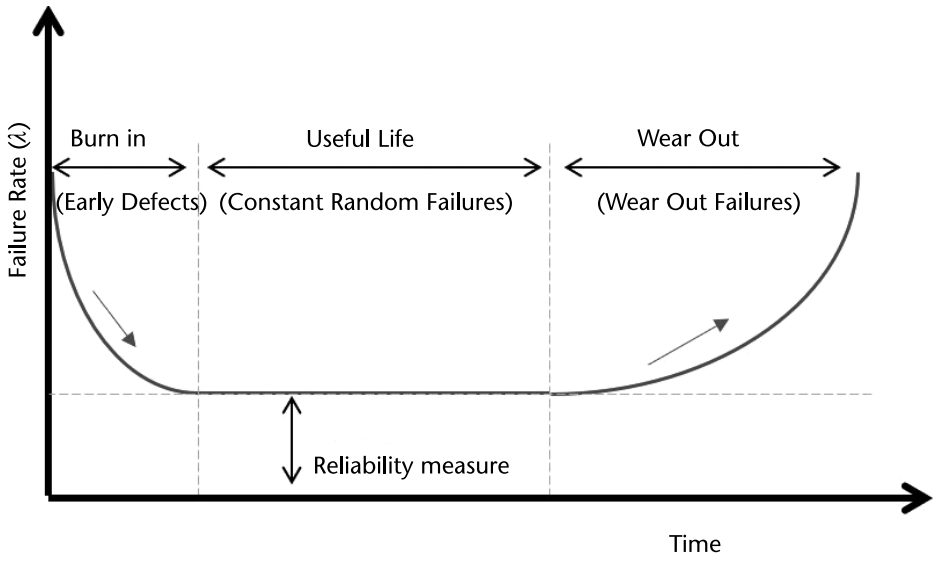



**Figure 4.1** The bathtub curve (also known as the component reliability model).

**Table 4.2** Summary of Failure Rates and Possible Causes During Different Phases of Bathtub curve

| *Phase* | *Failure Rate* | *Possible Causes* | *Possible Improvement Actions* |
|---|---|---|---|
| Burn-in | Decreasing | Manufacturing defects, soldering, assembly errors, part defects, poor QC, poor design | Better QC, acceptance testing, burn-in testing, screening, highly accelerated stress screening |
| Useful life | Constant | Environment, random loads, human errors, random events | Excess strength, redundancy, robust design |
| Wear-out | Increasing | Fatigue, corrosion, aging, friction | Derating, preventive maintenance, parts replacement, better material, improved designs, technology |

assumed to be constant during the useful life of the component as shown in Figure 4.1 and is generally expressed as failure in time (FIT). For example, if there is an occurrence of 10 failures for every $10^9$ hours, then $\lambda$ will be equal to 10 FIT. The value for $\lambda$ can also be calculated using the formulas given below with the MTBF or MTTF shown in the reliability data. By making use of $\lambda$, we can model the reliability of a component using an exponential distribution.

$$R_C(t) = e^{-\lambda t}$$

*Example 4.1*

During the trial of a key component for the SoC Platform project, sensor company SensorsThink observed that that 95 components failed during a test with a total operating time of 1 million hours (total time for all items: both failed and passed). Calculate the reliability of the key component after 1,000 hours (i.e., t = 1,000).

*Solution:*
Failure rate is given by:

$$Failure\ Rate = \lambda = \frac{Total\ number\ of\ failed\ components}{Total\ Operating\ Time}$$

$$\therefore \lambda = \frac{95}{1{,}000{,}000} = 9.5 \times 10^{-5}\ per\ hour$$

$$Reliability\ of\ a\ component\ at\ time\ t = R_C(t) = e^{-\lambda t}$$

$$R_C(1{,}000) = e^{-\left(9.5\times10^{-5}\times1{,}000\right)} = 0.909$$

Therefore, there is a 90.9% probability that the designed component for the sensor platform project will survive after 1,000 hours.

### 4.2.3 Early Life Failure Rate

Early defect rate or early life failure rate (ELFR) of a product is calculated during product qualifications or as part of ongoing product reliability monitoring activities. ELFR measures the reliability of the product during performance testing phase during the first few months in the field. As $\lambda$ is assumed to be a constant and is generally expressed in FIT, some texts also find it acceptable to express ELFR in terms of FIT. However, it is often desired to express the ELFR in parts per million (PPM), since PPM is a measure of the cumulative fraction failing per device, whereas FIT is a measure of fraction failing per device-hour. Therefore, it is a common practice to specify the early life time period ($t_{ELF}$), whenever ELFR is expressed in PPM. The value of $t_{ELF}$ is generally defined by the user or the supplier. The value of ELFR can be calculated using the following equation:

$$MTBF = \int_0^\infty R(t)\,dt = \frac{1}{\lambda}$$

ELFR provides a good indication of reliability during performance testing phase; however, once the critical issues have been identified and addressed, it is the failure rate ($\lambda$) that is usually considered to estimate system reliability.

As design engineers, ELFR can be regarded in two contexts. The first context is the components that the design engineers are choosing to build their own prototype. In this context, EFLRs of those components are useful only to estimate the overall reliability of the system being developed. In the second context, the design engineers have to understand that the system that they are developing will have its own bathtub curve and hence its own three phases: burn-in (ELFR), useful life, and phase-out. Ideally, the designers would like to minimize the early life time period and achieve a constant failure rate ($\lambda$) as soon as possible.

The reliability of a system is a study of factors that lead to its failure, and it is essential to provide this feedback to the designers and production engineers in the early stages of design prototyping and production so that they can make suitable

adjustments to prevent future failures. It is obvious that the final onus on causes of failure thus lies with either the designer or the production engineer, which may not be welcomed by either of them unless a robust classification of failure types is already agreed upon. When speaking about the reliability of a system, we end up talking more about the following three aspects: failure causes, failure modes, and failure mechanisms.

### 4.2.4 Key Terms

Before we go any further, it is important for the readers to understand some standard terms and definitions that are used quite frequently in the context of reliability.

- *Failure:* A failure is an event when an item is not available to perform its function at specified conditions when scheduled or is not capable of performing functions to specification.
- *Failure rate:* The number of failures per unit of gross operating period in terms of time, events, and cycles.
- *Hazard:* The potential to cause harm, including ill health and injury, damage to property, plant, products, or the environment, production losses, or increased liabilities.
- *Risk:* The likelihood that a specified undesired event will occur due to the realization of a hazard by or during work activities or by the products and services created by work activities.
- *Mean time between failures (MTBF):* The average time between failure occurrences. For repairable items, MTBF is calculated using number of items and their operating time is divided by the total number of failures.
- *Mean time to failure (MTTF):* The average time to failure occurrence. For repairable items and nonrepairable items, MTTF is calculated using number of items and their operating time divided by the total number of failures.
- *Mean time to repair (MTTR):* The average time to restore the item to specified conditions.
- *Maintenance load:* The repair time per operating time for an item.
- *Maintainability:* A characteristic of design, installation, and operation, usually expressed as the probability that an item can be retained in or restored to a specified operable condition within a specified interval of time when maintenance is performed in accordance with prescribed procedures.
- *Availability:* A measure of the time that a system is operating versus the time that the system was planned to operate. It is the probability that the system is operational at any random time $t$.

While it is important for all embedded system design engineers to know these terms and their importance while working on a new system design, it is also important to understand that most big companies have a separate reliability and fault testing division to undertake such studies. This is mainly because there is an inherent conflict of interest between designers and testers, even though they both are aiming towards the same goal of providing the best product to their clients or customers.

However, for smaller and mid-sized companies, it may not be financially viable to maintain a completely different departed dedicated for reliability testing; hence, the risk of conflict of interest is managed using other operating protocols.

### 4.2.5 Repairable and Nonrepairable Systems

In order to provide a robust classification of failure of a system, there also needs to be an agreement on the type of system being designed. In general, there can be two types of systems: nonrepairable systems and repairable systems.

A system that has been classified as nonrepairable is one that leads to the failure of the entire system even if one of the key (critical) components fail. For example, if a system has 5 temperature sensors and a malfunction in any one of the sensors constitutes a system failure, even if the faulty sensor could be easily replaced, then such a system can be considered as nonrepairable. However, if the agreed definition of system failure is based on the fact that at least 3 of 5 sensors should be working at any given time, then the system could be classified as repairable and the faulty sensor components can be replaced easily. The key characteristics of repairable and nonrepairable systems are listed in Table 4.3.

### 4.2.6 MTTF, MTBF, and MTTR

MTTF is a term used for nonrepairable systems and is defined as the expected value of $t_f$. In other words, it is the expected time of first failure in a nonrepairable system and can be calculated using:

$$MTTF = E\left(t_f\right) = \int_0^\infty R(t)\,dt = \frac{1}{\lambda}$$

MTBF is a term used for repairable electronic systems and is defined as the expected time between two failures of a repairable system as shown in Figure 4.2. It is calculated in the same way as MTTF, with an assumption that the system failure is repairable.

$$MTBF = \int_0^\infty R(t)\,dt = \frac{1}{\lambda}$$

MTTR is a term used in the context of repairable systems and is the amount of time required to repair a system and make it operational again.

**Table 4.3** Key Reliability Characteristics for Repairable and Nonrepairable Systems

| | *Nonrepairable Systems* | *Repairable Systems* |
|---|---|---|
| *Availability* | Function of reliability | Function of reliability and maintainability |
| *Key Factor* | Failure rate | Failure rate and repair rate |
| *Time to Failure* | MTTF | MTBF |
| *Expected Life* | Mean residual life (MRL) | MRL (Economic justification) |

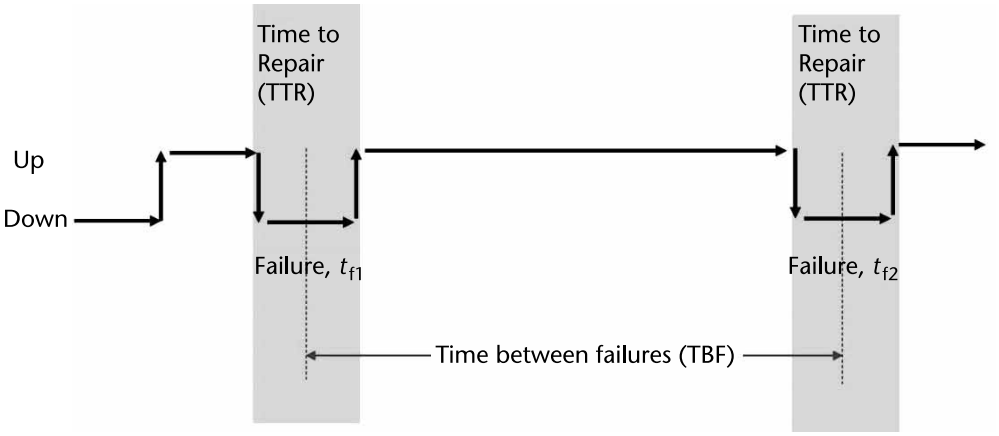



**Figure 4.2** MTBF for repairable systems. MTTF, which is a characteristic of nonrepairable systems, is a special case of this plot that calculates the first instance of failure (tf1).

$$MTTR = \frac{Total\ maintanance\ downtime}{Total\ number\ of\ failures (or\ maintenance\ actions)}$$

While calculating both MTBF and MTTR, it is safe to assume that that there are more than one failure and repair during the operational life of the system.

*Example 4.2*
Let us assume that the SoC Platform project by Sensor Company SensorsThink will eventually be deployed as a subsystem to detect fuel efficiency in a car. The prototype was tested in 200 cars for a total of 28,000 hours. Overall, 7 failures were observed. Calculate the MTBF and failure rate.

*Solution:*
Assuming that the overall car system is a repairable system (since it is highly unlikely that someone will change the car due to a fuel efficiency detector failure), we can use MTBF

$$MTBF = \frac{1}{\lambda} = \frac{Total\ operational\ time}{Total\ number\ of\ failures} = \frac{28{,}000}{7}$$

$$\therefore MTBF = 4{,}000\ hours$$

$$Average\ Failure\ Rate(\lambda) = \frac{7}{28{,}000} = 0.00025\ per\ hour$$

---

*Example 4.3*
Let us consider a scenario where SoC Platform project by sensor company SensorsThink is being deployed as the primary control system to detect ignition temperature at a chemical plant. Five such systems were tested in with failure hours of 157, 201, 184, 239, and 225. Calculate the MTTF and failure rate.

*Solution:*
Considering the primary control system used in the chemical plant as a nonrepairable system (needs replacement), we use MTTF:

$$MTTF = \frac{1}{\lambda}\frac{(157+201+184+239+225)}{5}$$
$$\therefore MTTF = 201.2 \; hours$$
$$Average\ Failure\ Rate(\lambda) = \frac{5}{(157+201+184+239+225)} = 0.00497 \; per\ hour$$
$$\therefore MTTF = 1{,}125 \; hours$$

### 4.2.7 Maintainability

The maintainability of an embedded system is the measure of the ability of the system or component to be retained or restored to a specified condition when maintenance is performed by qualified personnel using specified procedure and resources. It is an important parameter to be considered during the design phase as low maintenance leads to both reduced downtime as well as reduced costs. Maintainability also helps one determine the level of fault tolerance and redundancy techniques required for a certain system. Maintainability is measured by MTTR.

$$MTTR = \frac{\sum_{i=1}^{n} \lambda_i Rt_i}{\sum_{i=1}^{n} \lambda_i}$$

Here $n$ is the number of systems available and $\lambda_i$ is the failure rate of $i$th system. $Rt_i$ is the repair time for the $i$th system. The probability of performing a maintenance action within an allowable time interval can be calculated using:

$$M(t) = 1 - e^{-t/MTTR}$$

where $M(t)$ is the probability of completion of repair within time period $t$.

*Example 4.4*
Let us assume that sensor company SensorsThink has ordered 4 prototypes from the SoC Platform project that will be used for solar panel direction control systems for maximizing the angle of incidence of solar rays on the panel. During testing, it was found that one of the components on each the control systems fail at 1,100 hours, 1,150 hours, 1,050 hours, and 1,200 hours, respectively. The system could be repaired by replacing the faulty component, and the total time to repair all 4 systems was 48 hours. Calculate the time to failure of this system if it was:

1. Deployed in a farm house to provide renewable energy to a small farm;
2. Deployed on a mini satellite to control the solar panels.

*Solution:*
MTTR for this system = 48/4 = 12 hours.

1. In this type of application, the solar panel controller can easily be classified as a repairable system; therefore, we use MTBF

$$MTBF = \frac{1}{\lambda} = \frac{Total\ Operational\ Time}{Total\ Number\ of\ Failures} = \frac{(1{,}100 + 1{,}150 + 1{,}050 + 1{,}200) - 48}{4}$$

$$\therefore MTBF = 1{,}113 \text{ hours}$$

2. In this application, the system would be classified as nonrepairable (as it is practically impossible to repair or replace such systems in space); hence, we make use of MTTF:

$$MTTF = \frac{1}{\lambda} = \frac{Total\ Time}{Total\ Number\ of\ Failures} = \frac{(1{,}100 + 1{,}150 + 1{,}050 + 1{,}200) - 48}{4}$$

$$\therefore MTTF = 1{,}125 \text{ hours}$$

As seen in this example, the average time to failure of a system can vary significantly based on whether it is classified as a repairable system or a nonrepairable system.

---

*Example 4.5*
**Let us consider a scenario where** SoC Platform project requested by sensor company SensorsThink will be deployed as a subsystem in an automatic airflow control system, developed for an industrial warehouse, with an MTTR of 5 hours. What is the probability that the system can be repaired within 3 hours?

*Solution:*
Assuming that the sensor system used in the chemical plant is a repairable system, the probability of completion of repair within 3 hours can be found using:

$$M(t) = 1 - e^{-t/MTTR}$$
$$= 1 - e^{-3/5}$$
$$\therefore M(t) = 1 - 0.549 = 0.451$$

Therefore, there is a 45% probability of repairing this system within 3 hours.

---

### 4.2.8 Availability

Availability of an embedded system is a measure of the time during which the system is operational versus the time that the system was planned to operate. It is the probability that the system is operational at any random time *t*. The availability of

a system is very critical measure for repairable systems as it determines the overall operating efficiency and costs of the bigger system. For example, an airplane with an availability of 50% would take twice the number of years to just break even the cost of purchase and it could also result in higher cost expenditure with regard to buying more planes to serve the same amount of traffic. There are three common measures of availability: inherent availability, achieved availability, and operational availability

However, in the context of this book, we will just cover inherent availability, as there are other texts available to study this topic in more detail.

The inherent availability ($A_i$) is one of the most common measures to calculate the availability of a system. It does not consider the time for preventive maintenance and assumes that there is a negligible time lag between the time of failure and the start of repair. For repairable systems, it is defined as follows:

$$A_i = \frac{\mu}{\lambda + \mu} = \frac{MTBF}{MTTR + MTBF}$$

where $\mu$ is the repair rate (=1/MTTR) and $\lambda$ is the failure rate (=1/MTBF).

As evident in the examples given above, classification of a system as repairable or nonrepairable depends on several parameters and scenarios. Moreover, there are multiple parameters that define the reliability of the system. Some of these parameters are more important than others depending on the application and the environment in which the sensor system is used.

*Example 4.6*

The sensors company SensorsThink develops an SoC Platform to be used within a gas detection system developed for a food storage warehouse, which has an MTTR of 15 hours and an MTBF of 3,050. What is the inherent availability of the system?

*Solution:*

$$A_i = \frac{MTBF}{MTTR + MTBF} = \frac{3,050}{15 + 3,050} = 0.9951$$

Therefore, the availability of this system is 99.51%.

## 4.3 Calculating System Reliability

Like the bathtub curve shown in Figure 4.1 for key components of the system, it is important to plot a bathtub curve for the embedded system that you have designed, which will include one or many such key (critical) components. Once designers have enough information to plot the bathtub curve for the system being designed,

they have a powerful visualization tool available to estimate and predict system failure probability during different phases of the design cycle. The bathtub curve is also a good indicator of the useful life of the product that you are developing. As mentioned earlier, the useful life of a product is the time during which the system operates with a constant failure rate. Useful life is an important measure as it determines the cost of the system and the frequency of replacements as well as the duration of warranty. It is vital to understand that the product reliability (for the embedded system that you have designed) will vary greatly on the following factors:

1. Total number of components in the system including both critical and noncritical components;
2. Total number of critical components in the system;
3. The configuration in which components have been placed during the design and production phase;
4. Criteria that determines what constitutes a system failure.

Keeping the above factors in mind, designers need to come up with a bathtub model for the designed system that can capture the most basic estimate of overall system reliability. Even a simple embedded system can have hundreds, if not thousands, of individual components, making it an onerous task to calculate system reliability. At the same time, it is almost impossible to design a system and sell it to your client without having even a rough estimate of system reliability and the components that have the highest probability of failure and if the reliability of the system can be improved by clever design techniques, good programming, or providing fault tolerance. Let us try to understand the effect of these factors on the reliability of the system SoC Platform that you have been asked to design.

The first task is to determine the total number of components used in your design. The sensor platform project requested by SensorsThink, in Chapter 2, has a total 941 individual components. Trying to find the individual failure rate of each of those 941 components and then trying to come up with a single mathematical formula for system reliability that includes the individual failure rate of all 941 components is an almost impossible exercise. However, a good design engineer still needs to know all the components used in the system design. The more important task is to understand which of the components are critical and noncritical.

We have provided a technical definition of the key component for a sensor platform project requested by SensorsThink in Chapter 2. In simple terms, a critical (key) component of an embedded system can be defined as a piece of hardware or software, the failure of which would cause the system to fail altogether or seriously affect its capability to perform the desired task. A good example of a critical component would be a microprocessor chip in a laptop or a power supply unit. A noncritical component are those components that could potentially affect the performance of the system, but would not lead to critical failure. For example, a microphone or webcam on a laptop could potentially have a serious effect on user experience but would not probably lead to replacement of the laptop. There are many components, in each system, that could be labeled as critical or noncritical depending on how they are designed and how they are used. For example, a system designed to handle only one high capacity memory chip would fail if the memory device failed. However, if the same system can handle multiple chips, each with

a lower capacity, it can continue to work at a reduced efficiency if one or more memory chips fail.

Before designers can label an individual system component as critical or non-critical, they must first answer what the meaning of system failure is in the context of the embedded system being designed. Would the failure of a webcam in a laptop be considered as a critical system failure? The answer to that question is not with the designer, but with the user. If the laptop is being used for general browsing or as a standalone PC, then a webcam failure may not be considered as a critical failure, as there a number of good external webcams available to purchase at a reasonable cost. However, if the laptop is used to provide face-to-face online support service in a particular industry or to record online lectures on the go for live web-based delivery, then even a simple webcam failure could make the laptop worthless to its user and hence that failure will constitute a critical failure. In the context of the sensor platform project we designed in earlier chapters, it would be worthwhile to provide a definition of critical failure. For example, if you have only one sensor malfunctioning of 10 different sensors in your system, would that constitute a system failure? To keep it simple, we can assume that all key components in the sensor platform project are critical and even a single component failure will lead to system failure.

Once the designer has identified a minimum possible set of critical components of the system, it is quite possible to reduce the complexity of the equation that determines system reliability. For the sake of simplicity, let us assume that there are only $n$ critical components (of a total of 941) in the system SoC Platform. We need to calculate $R_S(t)$, which is the overall reliability of the system SoC Platform. Let us assume that and the failure rates of all $n$ critical components in the system SoC Platform are statistically independent of each other. Furthermore, we also assume that there is no redundancy of fault tolerance techniques deployed in the system SoC Platform. Let us see how we can use this to determine system reliability under the following scenarios.

### 4.3.1 Scenario 1: All Critical Components Connected in Series

In this case, the critical path of the design passes through all the critical components in a sequential manner (Figure 4.3). Assuming that $R_{Ci}(t)$ is the reliability of an individual critical component and $i$, from the equation of reliability given earlier:

$$R_{Ci}(t) = e^{-\lambda_i t}$$

As the failure rates of all individual critical components are statistically independent, the overall system reliability in this case will be a product term of $n$ individual critical component reliability $R_S(t)$

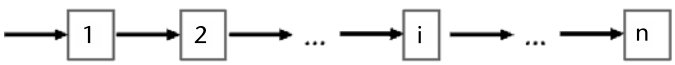



**Figure 4.3** Critical path of a system if all key components are connected in series.

$$R_S(t) = R_{C1}(t) \times R_{C2}(t) \times R_{C3}(t) \times \ldots \times R_{Cn}(t)$$
$$\Rightarrow R_S(t) = \prod_{i=1}^{n} R_{Ci}(t)$$

The overall system reliability can therefore be represented in terms of the individual failure rate ($\lambda_i$) of each component:

$$R_S(t) = e^{-t\left(\sum_{i=1}^{n} \lambda_i\right)} = e^{-t\lambda_S}$$

where $\lambda_S$ is the overall serial failure rate and can be represented by:

$$\lambda_S = \sum_{i=1}^{n} \lambda_i$$

A special case of series connection can be if the failure rates of critical components are not statistically independent. In that case, we need to derive a failure rate equation based on the subset of critical components that form the critical path.

In such a case, the failure probability of the system can be found by determining the subsets of critical path components, whose functioning ensures the functioning of the system even if other critical components outside the critical path fail. For example, in Figure 4.4, we can have four such critical paths: {1,2,6}, {1,2,4,5}, {3,4,5}, and {3,6}. However, this is a special case scenario and most likely results in system malfunction whenever the faulty critical path is chosen and properly functioning otherwise. In a way, it will again come down to how we define system failure and critical components.

*Example 4.7*
Consider a sensor system that has 50 critical components connected in series. If the reliability of each component is 0.999, what will be the overall system reliability?

*Solution:*
Since the components are connected in series, the system will operate or fail if any one of the critical components fail. The reliability of the system is given by the equation:

$$R_S(t) = \prod_{i=1}^{n} R_{Ci}(t)$$

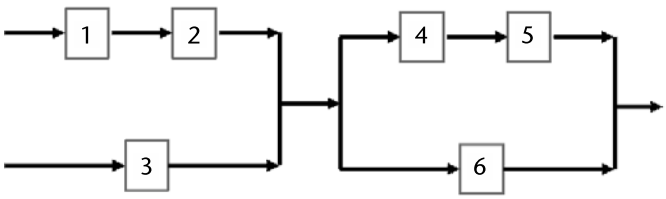



**Figure 4.4** Critical path of a series system with dependent critical components.

Here, individual reliability for all critical components is the same; therefore, the equation reduces to:

$$R_S(t) = \left(R_C(t)\right)^n$$

Plugging the values for $R_{Ci} = (t) = 0.985$ and $n = 50$, we get

$$R_S(t) = (0.999)^{50} = 0.9512$$

The probability of all 5 components failing is $(0.02)^5 = 3.2 \times 10^{-9}$. Therefore, the reliability of the system is RS = $1 - 3.2 \times 10^{-9} = 0.9999999968$

---

*Example 4.8*

Let us take an example of the embedded system design for SensorsThink and determine the reliability of the system being developed after 5 years. Refer to Chapter 2, where we performed component selection and listed the key components used in the design. For the sake of simplicity, we will choose only a subset of key components used in the design.

C1. SoC: Xilinx Zynq-7000 Series; XC7Z020-2CLG400I
C2. DDR3: (2) Micron 16-bit/4 Gbit MT41K256M16TW-107
C3. Filing System Storage: Micro SD Card
C4. Ethernet PHY: Microchip KSZ9031
C5. USB 2.0: Microchip USB3320
C6. USB to UART Converter: FTDI FT230X
C7. Wi-Fi/Bluetooth Module: Microchip ATWILC3000
C8. Local Relative Humidity/Temperature Sensor: Sensiron SHTW2
C9. Infrared Sensor: FLIR Lepton 500-0771-01

| *Component No.* | *Component Description* | *Failure rate (λ)* | *Component Reliability ($R_C T = e–\lambda t$)* |
|---|---|---|---|
| C1 | Xilinx Zynq-7000 series SoC | 22 FIT | 0.999 |
| C2 | DDR3 RAM module | 100 FIT | 0.996 |
| C3 | Micro SD card | 220 FIT | 0.990 |
| C4 | Ethernet PHY IC | 45 FIT | 0.998 |
| C5 | USB 2.0 IC | 165 FIT | 0.993 |
| C6 | USB to UART converter module | 425 FIT | 0.982 |
| C7 | WiFi/Bluetooth module | 185 FIT | 0.992 |
| C8 | Humidity/Temperature Sensor | 400 FIT | 0.983 |
| C9 | Infrared Sensor | 35 FIT | 0.998 |

[Note 1: 1 FIT is equal to one failure in billion working hours or in other words FIT = $\lambda_{hours} \times 10^9$.]
[Note 2: In the table above, the value of $R_C(t)$ is calculated using $t$ = 5 years = 43,800 hours.]

Since each of the above components is critical, we can assume that the components are connected in series for the purpose of reliability calculation. The reliability of the system is given by the equation:

$$R_S(t) = \prod_{i=1}^{n} R_{Ci}(t)$$

Therefore, the reliability of the system after 5 years is $R_S = 0.9324$ or 93.24%

### 4.3.2 Scenario 2: All Critical Components Connected in Parallel

As shown in Figure 4.5, each of the critical paths in the design contains only one critical component or in other words, all critical components are parallel to one another. Assuming $R_{Ci}(t)$ is the reliability of an individual critical component and $Q_{Ci}(t)$ is the failure rate of the component, we get

$$Q_{Ci}(t) = 1 - R_{Ci}(t)$$
$$\Rightarrow Q_{Ci}(t) = 1 - e^{-\lambda_i t}$$

Assuming that the failure rates of all critical components are statistically independent, we get:

$$Q_S(t) = \prod_{i=1}^{n} Q_{Ci}(t)$$

Hence, the equation for overall system reliability will be:

$$R_S(t) = 1 - \prod_{i=1}^{n} Q_{Ci}(t)$$
$$\Rightarrow R_S(t) = 1 - \prod_{i=1}^{n} \left[1 - R_{Ci}(t)\right]$$

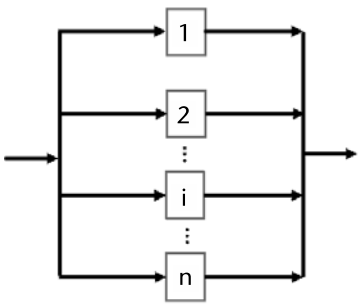



**Figure 4.5** Critical path of a system if all key components are connected in parallel.

*Example 4.9*
Consider a sensor system that has 5 critical components connected in parallel. If the reliability of each component is 0.98, what will be the overall system reliability?

*Solution:*
Since the components are connected in parallel, the system will operate reliably if at least one of the critical components does not fail. The failure probability of a single component is:

$$Q_{Ci}(t) = 1 - R_{Ci}(t)$$
$$\Rightarrow Q_{Ci}(t) = 1 - 0.98 = 0.02$$

The probability of all the 5 components failing is $(0.02)^5 = 3.2 \times 10^{-9}$
Therefore, the reliability of the system is $R_S = 1 - 3.2 \times 10^{-9} = 0.9999999968$

### 4.3.3 Scenario 3: All Critical Components Are Connected in Series-Parallel Configuration

In this scenario, the critical path of the design consists of several blocks connected in a series. For example, in Figure 4.6, block 1 is in series with block 2, but with each block there are several critical components that are connected in a series/parallel configuration. In other words, critical failure of one block leads to the failure of the system. However, critical failure of one component in the block may or may not result in failure of the block depending on the series/parallel configuration in which it is aligned.

## 4.4 Faults, Errors, and Failure

So far, we have seen that reliability of a system is a function of the individual reliabilities of its key components. As a system designer, most of the time, you would not have control over the reliability and failure rates of the components that you use for your design. The little leverage you have in this regard is to make an informed choice to choose the best possible components that are most reliable based on their technical documentation. However, this does not mean that designers cannot do

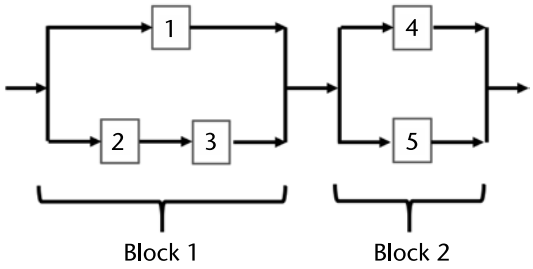



**Figure 4.6** Critical path of a system with key components in series-parallel combination.

anything to improve system reliability, even when they are forced to choose a component that may not be offering the best reliability (lowest possible failure rate).

In simple terms, the reliability of a system is a function of its failure rate. Failure in a system occurs when the delivered service deviates from specified service. Generally, the failure of a system occurs when a key component of the system (hardware or software) was erroneous. The cause of this error is termed as the fault. In other words, a fault creates a latent error, which becomes effective when activated. When this activated error affects the delivered service, it results in a failure. This relationship between fault, error, and system failure is shown in Figure 4.7.

There can be several reasons for a fault to occur. Some of the most common ones are:

- Design errors;
- Software or hardware (either mechanical or electronic);
- Manufacturing problems (related to production and quality control);
- Damage, fatigue, and deterioration (operating the system in the wearout phase of the bathtub curve);
- External disturbances (can be environmental, situational, operational, or manual);
- Harsh environmental conditions such as electromagnetic interference and ionization radiation;
- System misuse (one of the leading causes of system failure).

Faults can be broadly classified under the following four categories shown in Figure 4.8.

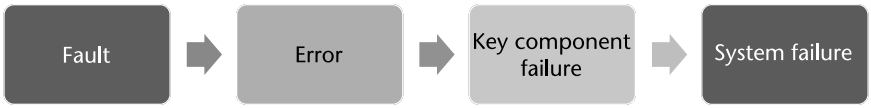



**Figure 4.7** Fault leading to system failure.

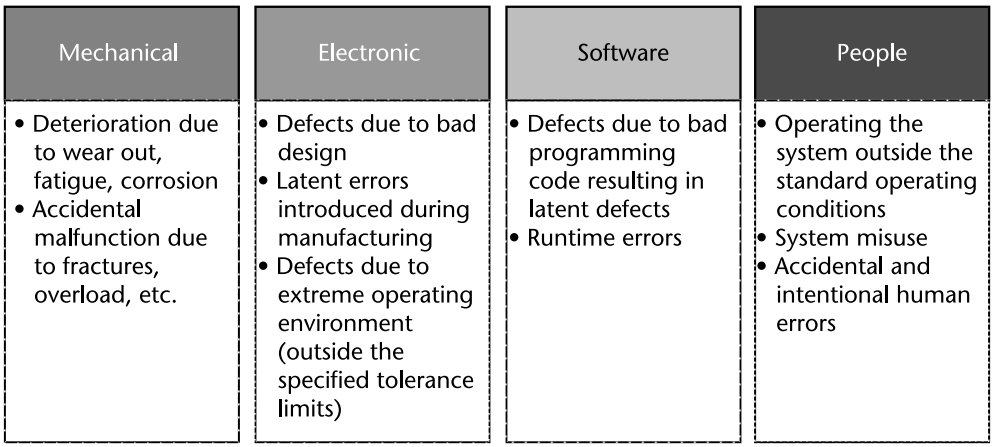



**Figure 4.8** Classification of system faults.

No system can be designed as 100% reliable. In other words, even in a high reliability system, there is still a probability of system failure, which could occur due to any of the reasons stated above. All that a designer can do is to work on the design to reduce that probability of fault occurring that could lead to a system failure. This approach towards reliable system design, when we are aware of the most likely causes of faults present in the system, is often termed as fault tolerant computing. In other words, it is acceptable fact that all systems will have key components (hardware or software) that will develop faults, but as a designer, we have to work with the same set of components and still increase the overall reliability of the system using clever design techniques.

It is important to understand that, just like reliability engineering is an established practice in manufacturing industry, fault tolerant computing is also an established practice in the field of software development. In recent years, both these techniques are being fine-tuned and deployed in an embedded systems design as it encompasses both hardware and software development. While it is difficult to study and summarize all aspects of fault tolerant computing in a section of this book, we would try to provide a snapshot of some of the most common fault tolerance techniques used in low-cost, reliable design.

### 4.4.1 Classification of Faults

Before we discuss different types of fault tolerance techniques, we need to classify faults either a hardware fault or a software fault. Hardware faults can be further classified based on the time/duration of the fault (transient, permanent, or intermittent faults).

A transient fault, as the name suggests, occurs at a random instance of time and is evident for a small duration before disappearing. Unless there is a clear correlation of the fault timing with an external trigger, such faults are very difficult to isolate and diagnose (e.g., hardware component failure due to an exposure to radioactivity or a component failure due to sudden extreme change in weather conditions). Some systems are quite prone to transient faults and are designed based on the assumption that these faults will happen (e.g., communication systems). Permanent faults remain in the system until they are repaired (e.g., a broken wire or a software design error). Intermittent faults are a special case of transient faults that occur from time to time. However, the biggest difference between the two is that intermittent faults are more predictable in nature (e.g., over the period if we are aware that a particular sensor component is sensitive to low temperatures and high humidity). In such a scenario, it may need to be brought back to within the operating range of temperature/humidity and it will start to work again.

Software faults are generally called bugs and are classified into two categories.

- *Bohrbugs:* Software faults that are easily reproducible and identifiable.
- *Heisenbugs:* Faults that are only active under rare conditions (e.g., when there is a race condition existing within a state machine).

Unlike hardware faults, it is not a common practice to classify software faults as being transient, intermittent, or permanent. Since software upgrade is a much

easier option than hardware changes (which sometimes leads to system redesign design), the presence of software faults is generally unacceptable within a system. Software does not deteriorate with age, so it is either correct or incorrect, but some faults can remain dormant for long periods. Moreover, in the case of embedded systems, a change in software due to hardware changes could potentially lead to bugs or errors being introduced in the system (as is the case when we upload new drivers or update libraries). That kind of fault can be classified as both hardware and software-related.

Figures 4.9 and 4.10 show the classification of faults and errors based on a number of factors discussed above.

### 4.4.2 Fault Prevention Versus Fault Tolerance: Which One Can Address System Failure Better?

There are two main approaches to building a reliable embedded system: fault prevention and fault tolerance. Fault prevention attempts to eliminate the possibility of a fault occurring in a system while it is operational. Fault tolerance is the ability of a system to deliver expected performance even in the presence of faults. Both these approaches attempt to produce reliable systems that have well-defined failure modes. As discussed earlier in this section, faults lead to errors and errors cause failure. Therefore, failure mode analysis needs to consider different types and sources of faults and errors that lead to system failure as shown in Figure 4.11.

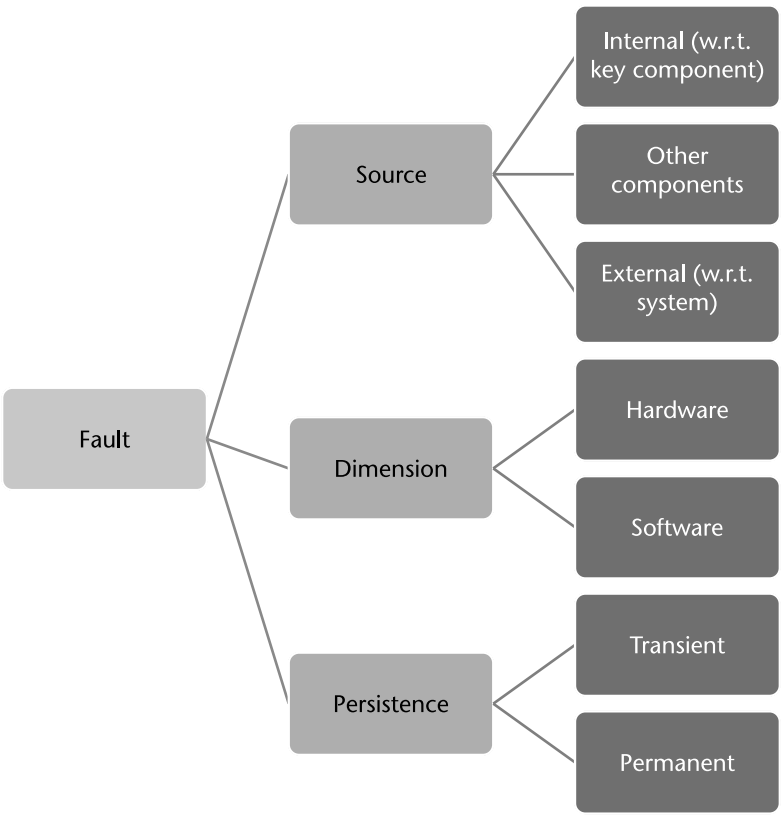



**Figure 4.9** Sources and types of faults.

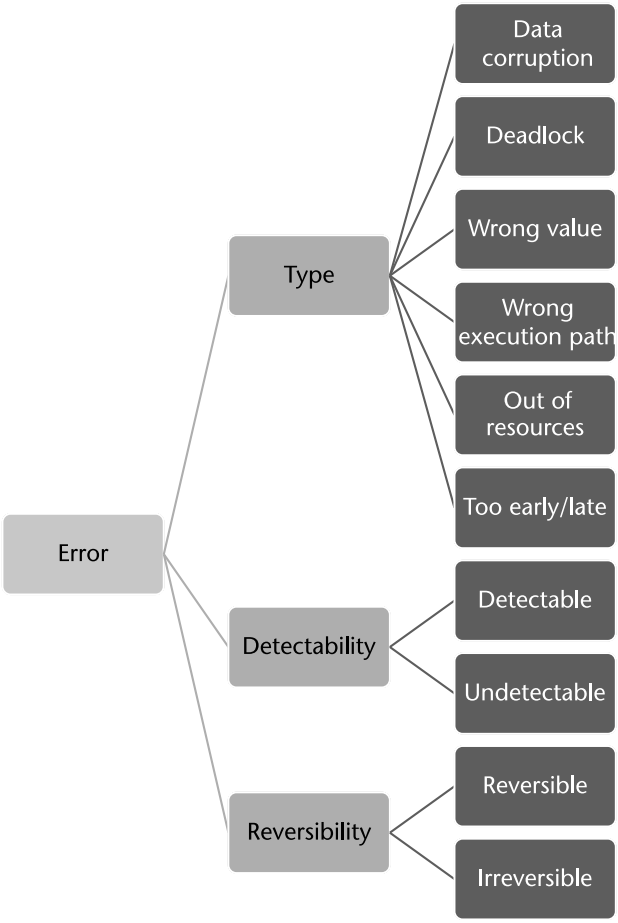



**Figure 4.10** Sources and types of errors.

Since fault prevention attempts to eliminate the possibility of a fault occurring in a system while it is operational, there are two stages in which it can be implemented: (1) fault avoidance, and (2) fault removal. Fault avoidance aims to minimize the introduction of any faults during system design and manufacturing phase using good design practices as highlighted in Figure 4.12. Fault removal involves procedures for finding and removing causes of errors (e.g., design reviews, program verification, code inspections, and system testing). However, unless there is an unlimited budget and time available, system testing can never be exhaustive. This means that there is always a probability that some of the errors were not caught during the testing phase. Moreover, system testing suffers with some shortcomings. For example, a test can only be used to show the presence of faults, not their absence. Moreover, it is sometimes impossible to test under realistic conditions (e.g., if a design would need to be installed in space). Since most of the testing is performed using software simulations, the efficiency of test is limited to the accuracy of simulation software. Finally, many errors that were inserted during the system's development phase may not manifest themselves until the system goes operational and hence may not get caught during the testing phase. There is no system in this world that can be tested rigorously to be 100% fault proof as believe it or not, for

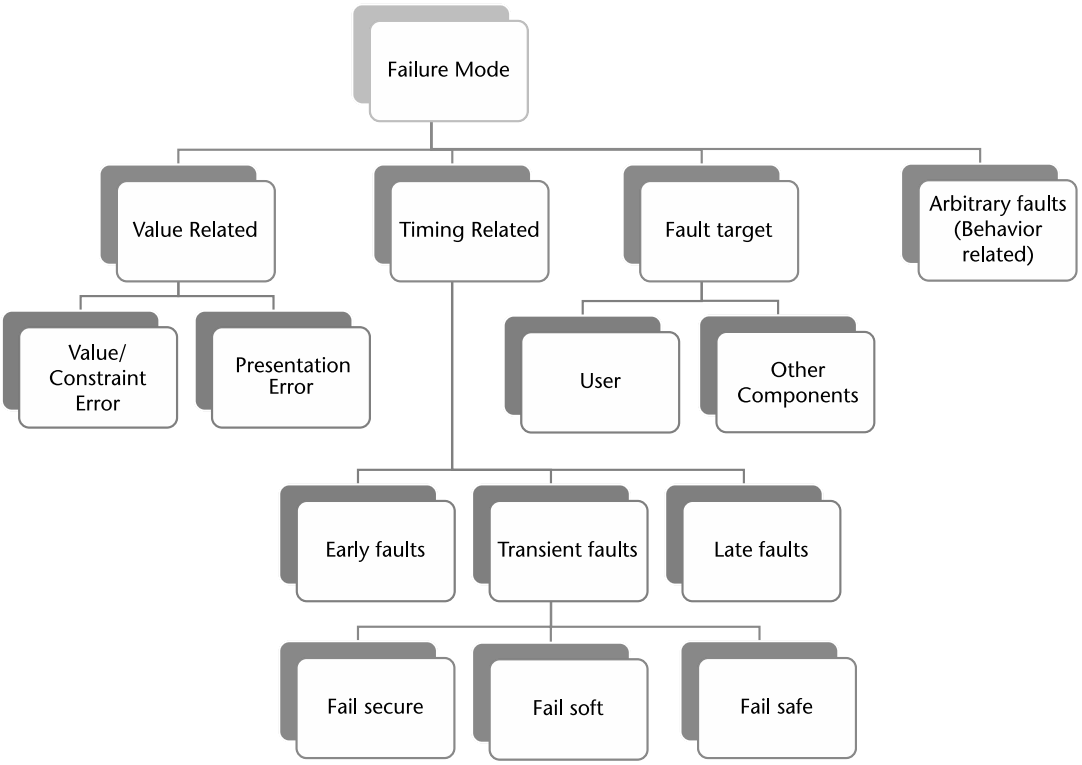



**Figure 4.11** Failure mode analysis depending on type of faults.

some reason or another, hardware components will fail (with some probability of failure). Moreover, the fault prevention approach is considered unsuccessful when either the frequency or duration of repair times too high or if the system is inaccessible for maintenance and repair activities (e.g., a satellite in space or a mission to another planet).

## 4.5 Fault Tolerance Techniques

Since fault prevention cannot be guaranteed, despite taking up a significant amount of budget and resources, it is prudent to design a system keeping in mind that the system will develop faults and some key components will fail over time. If these assumptions (regarding inherent fault rates) are used as design criteria and production criteria, it would ultimately be a lot cheaper and quicker to deliver a product that can provide the expected delivery of service.

As mentioned earlier, reliability of a system is a function of its failure rate, which, in turn, is dependent on the failure rate of various key components that make the system. The importance of reliability consideration, while specifying the design specifications of a system, depends on the application for which the system is being designed. Therefore, it is obvious that a system that needs high reliability will require a very high level of fault tolerance techniques included in the design and development stage, while other applications that may not need such high reliability can continue to operate with lower fault tolerance design requirements. To

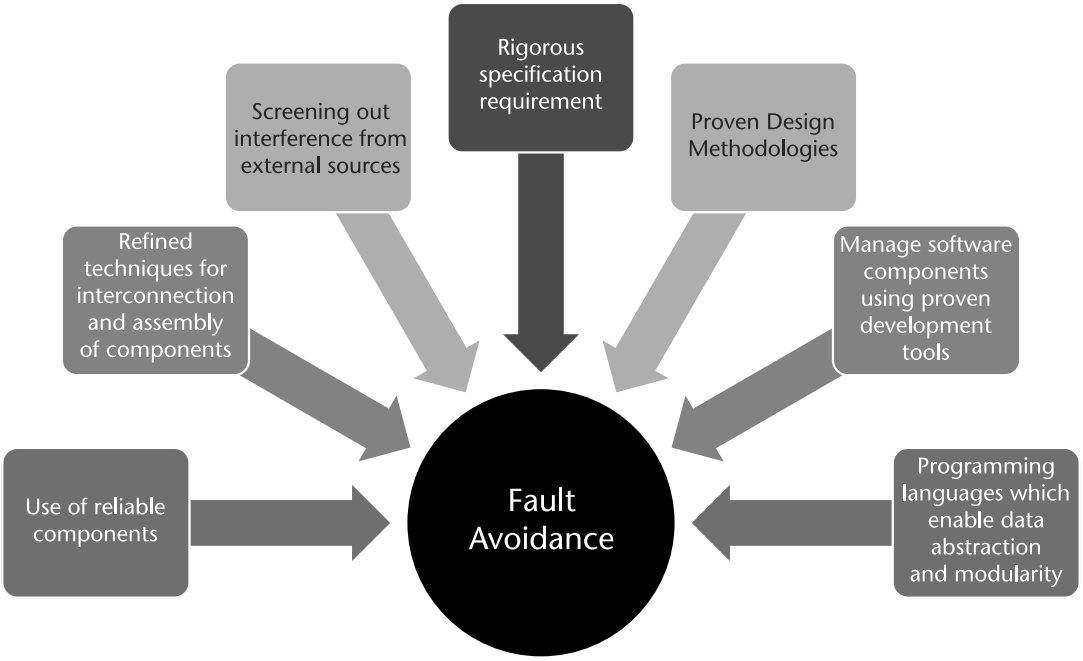



**Figure 4.12** Fault avoidance techniques.

make an informed decision on the level of fault tolerance that needs to be incorporated into a design, we need to answer these three questions (Figure 4.13):

- How critical is the component? For example, will the sensor design platform ordered by SensorsThink be considered a highly critical system if it were to be used as one of the control systems in a nuclear power plant or in a fighter jet?
- How likely is the component to fail? Some components, the other casing of the sensor design platform, are not likely to fail, so no fault tolerance is needed.
- How expensive is it to make the component fault tolerant? Requiring a redundant Zynq processor in the sensor design platform, for example, would

| Fault Tolerance Level | Fail Secure | Fail Soft | Fail Safe |
|---|---|---|---|
| Description | System operates in the presence of faults without significant loss in performance | Allows partial degradation in performance while maintaining operation | Allows temporary halt in operation |
| Example Applications | Critical systems such as nuclear power stations, space vehicles etc. | Most systems fall under this category. (e.g. ageing phone battery or memory) | Noncritical systems that can go offline for repairs or replacement. |

**Figure 4.13** Levels of fault tolerance.

likely be too expensive both economically and in terms of weight, space, and power consumption to be considered.

Fault tolerance techniques for embedded systems can broadly be classified into two categories: (1) hardware fault tolerance techniques, and (2) software fault tolerance techniques. It needs to be understood that both these fault tolerance categories have been well studied and researched over the past few decades. However, embedded system design is a special case where both techniques are implemented together and with equal importance. This is because reliability of an embedded system is a function of reliable design of both its hardware and its software.

### 4.5.1 Redundancy Technique for Hardware Fault Tolerance

The concept of fault tolerance techniques through redundant hardware components was conceived in the early 1950s and it has been studied and evolved over the decades since. The most common approach towards providing fault tolerance for key hardware components is the redundancy technique, which relies on extra elements introduced into the system to detect and recover from faults. The extra hardware components inserted in the design are considered redundant as they are not required in a perfect (fault-free) system. However, the addition of redundant components inevitably increases the complexity (and costs) of the overall system, which in itself can lead to less reliable systems. Two of the most common redundancy techniques used in hardware design are: (1) static redundancy, also known as masking redundancy, and (2) dynamic redundancy.

#### 4.5.1.1 Static Redundancy

Static redundancy uses extra components to mask the effects of faulty critical components. It is called static redundancy as once the redundant components are installed, their interconnection remains fixed. A common assumption used in static redundancy is that the fault is not due to design error, but rather due to transient fault or component degradation. While there are several types of redundancy design techniques available (such as triple modular redundancy, N-modular redundancy, ECC), it would be out of the scope of this book to study all of them. Triple modular redundancy (TMR), as shown in Figure 4.14, is one of the most common techniques used in fault tolerant hardware design. It makes use of three identical

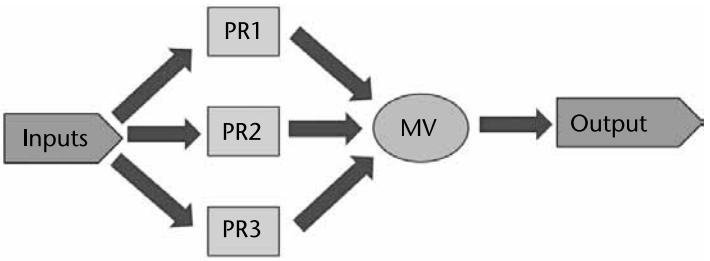



**Figure 4.14** The TMR technique for hardware fault tolerance. All three processor units (PR1, PR2, and PR3) process the same input and the majority voter (MV) compares the results and outputs the majority vote (two of three).

subcomponents and the majority voting circuit. All three subcomponents process the same input, and the majority voter compares the output from all three subcomponents. If any one of the output is different from other two, that component is masked out and is considered faulty.

#### 4.5.1.2 Dynamic Redundancy

A system designed with dynamic redundancy is capable of reconfiguring of its components when a fault is detected. However, the success of dynamic redundancy is dependent on the effectiveness of fault detection capability. Dynamic redundancy is supplied within a component to detect an error in the output and indicate the same. Once a faulty component is flagged, the circuit should be able to recover the output using another component. A good example of dynamic redundancy is in circuits used for communication systems that utilize checksums to detect and flag faulty transmission.

### 4.5.2 Software Fault Tolerance

The most prominent feature of embedded systems is the emphasis on hardware-software codesign. While it is easier to develop systems with built-in fault tolerance for hardware components that provides a highly reliable service throughout the useful life of the product, it is virtually impossible to create a flawless software. It is safe to assume that no matter how well we design the system, there will be some bugs in the software design. Software fault detection is a bigger challenge than hardware fault tolerance as most of the software faults are of a latent type that shows up while the system is operational. Therefore, we do require software fault tolerance techniques to be incorporated to tolerate these bugs. However, while hardware fault tolerance can keep a system fully operational even in the presence of faults, software fault tolerance mostly involves a change in programming specification to run the system at a lower performance, while the bug is being fixed. Software fault tolerance can be broadly classified into two groups: (1) single version techniques, and (2) multiversion techniques.

#### 4.5.2.1 Single-Version Techniques

Single-version techniques are generally used to improve the fault tolerance of a single piece of software. These techniques improve the quality of software by including some extra mechanisms into the design, which help in error detection, containment, and handling. Figure 4.15 shows some of the key attributes of single-version techniques such as modularity, system closure, atomicity of actions, and exception handling.

#### 4.5.2.2 Multiversion Techniques

Multiversion software fault tolerance makes use of two or more versions of a piece of software executed either in sequence or in parallel. Like the single-version techniques, key attributes such as modularity, system closure, atomicity of actions, and

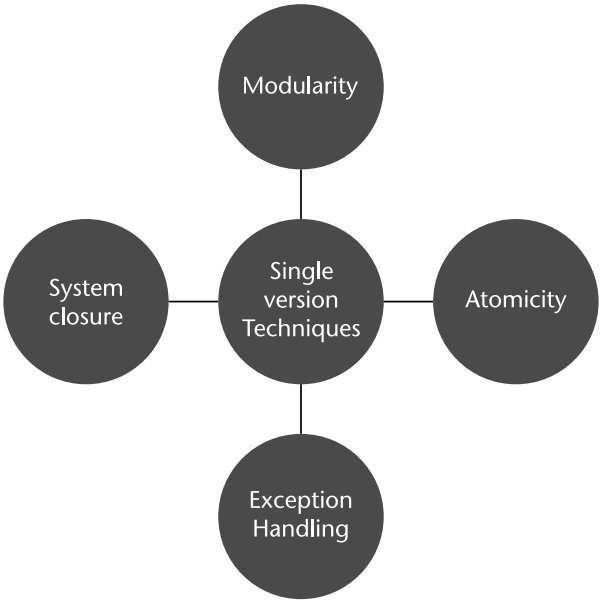



**Figure 4.15** Key attributes of single-version software fault tolerance techniques.

exception handling are desirable and advantageous in each version of the software being used in the multiversion scheme. Two of the most widely used techniques of multiversion software fault tolerance are recovery block (RB) scheme and N-version programming (NVP) scheme. Of these two, NVP is a static method, like the triple modular redundancy used in hardware fault tolerance. However, the RB scheme is a dynamic redundancy approach. Both NVP and RB schemes require design diversity (for various alternate versions of the software) as well as hardware redundancy to implement the scheme. Design diversity is a software design approach in which different versions of the software are built independently but deliver the same service. The fundamental assumption of design diversity is that components built differently will fail differently.

NVP is a static scheme in which output is computed using $N$ independently designed software modules or versions and their results are sent to a decision algorithm that determines a single decision result. The primary belief behind the NVP scheme is that the "independence of programming efforts will greatly reduce the probability of identical software faults occurring in two or more versions of the program." The first step in the NVP scheme involves the generation of $N \geq 2$ functionally equivalent programs (also called alternate versions) from the same initial specification. As the goal of the NVP scheme is to minimize the probability of similar errors in each version of the software component, it encourages the use of different algorithms, programming languages, environments, and tools wherever possible. With such a big diversity in the design of different versions of the same software component, NVP requires considerable developmental effort and increase in problem complexity. According to some studies, a 25% rise in the problem complexity can lead to a 100% rise in program complexity. However, in case of NVP, the increase in complexity is not greater than the inherent complexity of building a single version.

The different version of program execute concurrently with the same inputs and their results are compared by a decision algorithm (driver) as shown in Figure 4.16. The decision algorithm should be capable of detecting the software versions with erroneous outputs and prevent the propagation of bad values to main output. The decision algorithm should be developed keeping in mind the security and reliability of the application. In the case of applications that require embedded system design using SoCs, the NVP scheme is quite common. Significant advancements in chip design industry are leading to a greater complexity in design; therefore, the probability of design fault is greater because a complete verification of the design is very difficult to achieve within a given timeframe. The use of N-versions of components allows the continued use of chips with design faults as long as their errors at the decision points are not similar.

The efficiency of NVP scheme depends on a number of factors, primarily the initial design specification. If the specification is incomplete or incorrect, the same error will manifest in all different versions of the software, rendering the scheme useless. Design diversity is also a key parameter of with regard to success of NVP scheme. Finally, the development budget for the NVP scheme limits the number of independent versions of the software that can be used. A 3-version scheme will require 3 times the budget of a single version, which raises another question. Would the resources and time needed to implement an *N*-version scheme may be better utilized by producing a more efficient single-version scheme with the same amount of resources and time? This question probably depends on the industry for which the product is intended. Most companies or industries will probably opt for a robust single-version scheme with a highly robust software system rather than an *N*-version scheme.

The recovery block scheme, shown in Figure 4.17, uses alternate versions of the software to perform a run-time error detection using an acceptance test performed on the results delivered by the first (primary) version. If the system fails the acceptance test, the state is restored to what existed prior to the execution of that algorithm and an alternative module is executed. If the alternative module also fails the acceptance test, the program is restored to the recovery point and yet another module is executed. Recovery is considered complete when the acceptance test is passed. A sample test script for a recovery block scheme is shown in Figure 4.18. Checkpoint memory is needed to recover the state after a version fails to provide a valid starting operational point for the next version.

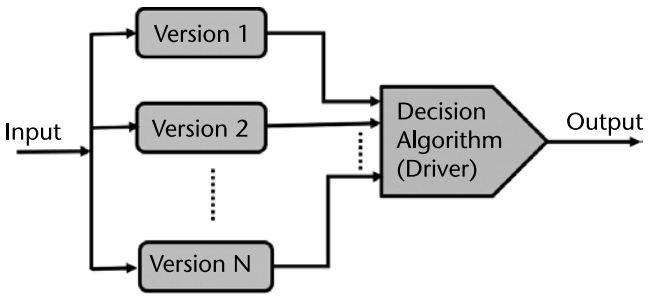



**Figure 4.16** N-version programming scheme.

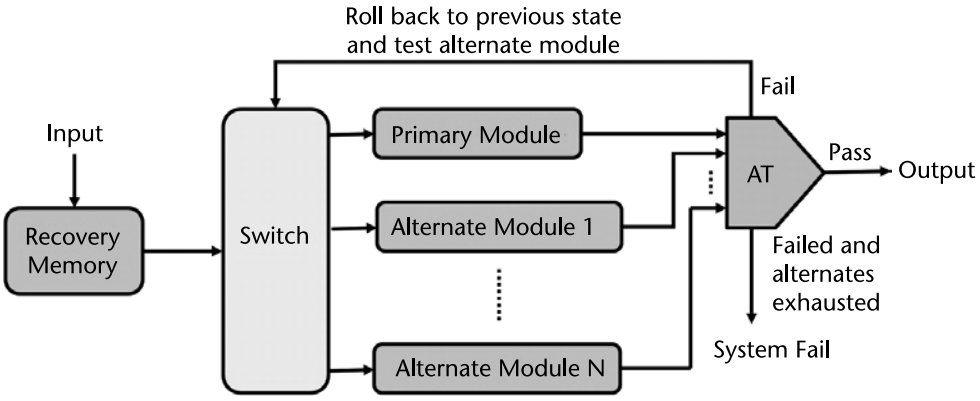



**Figure 4.17** The recovery block scheme consists of three key elements: a primary module to execute critical software functions, an acceptance test (AT) for the output of the primary module, and alternate modules, which perform the same functions as the primary module and deliver output in the event of a fault with the primary module.

```
ensure <acceptance test>
by
      <primary module>
else by
      <alternative module>
else by
      <alternative module>
      ...
else by
      <alternative module>
else error
```

**Figure 4.18** Recovery block test script.

If all modules fail, then the system itself fails and recovery must take place at a higher level.

While both NVP and RB schemes are quite useful in providing software fault tolerance, there are some key differences in both schemes. These differences have been summarized in Table 4.4.

With the ever-increasing complexity and omnipresence of embedded systems and different aspects of our everyday lives, the comparison between NVP and RB schemes needs to be studied in far greater depth and understanding than just a section in this book. So we only take a cursory view of most used hardware and software fault tolerance techniques with which the readers should be acquainted. For a deeper understanding of this topic, there are some outstanding texts available in the market.

## 4.6 Worst-Case Circuit Analysis

Worst-case circuit analysis (WCCA) is a commonly used technique used in the industry to specify the boundary parameters for the operating conditions during the

**Table 4.4** Comparison Between NVP and RB Schemes

| | *N-Version Programming (NVP)* | *Recovery Block (RB)* |
|---|---|---|
| *Type of Scheme* | Static | Dynamic |
| *Design Overheads* | Requires a driver | Requires Acceptance test |
| *Runtime Overheads* | Requires N*resources | Requires memory to store recovery points |
| *Diversity of Design* | Susceptible to errors in requirements | Susceptible to errors in requirements |
| *Error Detection* | Majority vote comparison algorithm | Acceptance test |
| *Atomicity* | Votes before output is delivered | Only outputs if acceptance test passed |

useful life of a product. WCCA examines the effects of potentially large magnitudes of variations of components used in the embedded system design beyond their initial tolerance level. While it may not seem that obvious, the variation in component parameters by just a few percentage points can lead to a big difference in system reliability under the worst-case scenario. The variations can be the result of aging or environmental influences, which can cause circuit outputs to drift out of specification.

WCCA combined with failure mode and error analysis and MTBF analyses are essential to the design of any reliable system; however, there is one key difference. WCCA is a quantitative analysis technique as opposed to a part-based approach for stress testing, FMEA, and MTBF (Figure 4.19). WCCA determines the mathematical sensitivity of circuit performance to parameter variations (for components and subsystems) and provides both statistical and nonstatistical methods for handling the variables that affect system performance. In a nutshell, WCCA is used to predict the most likely scenarios under which there is a high probability of system failure.

WCCA technique usually involves considerable amount of planning and data collection before it can be performed on a circuit. Before commencing, we should first document the theory of operation and develop a functional breakdown of the circuit (block diagrams and outlines of critical subcomponents). We should also have access to technical documentation of most parts used in the design (see step C below). The flowchart shown in Figure 4.20 presents the general overview of WCCA technique. It demonstrates the interrelationship of the various tasks required for completion of a WCCA.

- *Step A—Critical system attributes:* Before we begin WCCA, we must have a complete understanding of the system requirements and specifications. We should then decide whether we wish to perform a complete circuit analysis or implement circuit partitioning to analyze critical functions. Further on, we need to identify the circuitry to be analyzed (analog, digital, software) and determine the availability of component data. Finally, we need to identify the tools available to support the analysis. Failure to review this information could result in the failure to examine relevant system parameters.
- *Step B—Comparison of performance specification with WCCA results:* WCCA generally includes two steps: (1) worst-case stress analysis, which

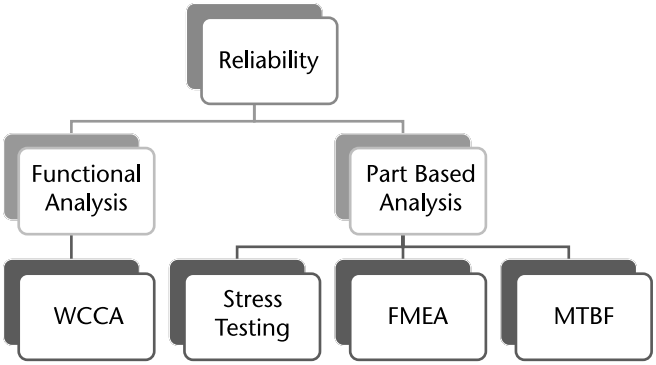



**Figure 4.19** Most used reliability techniques in embedded system design.

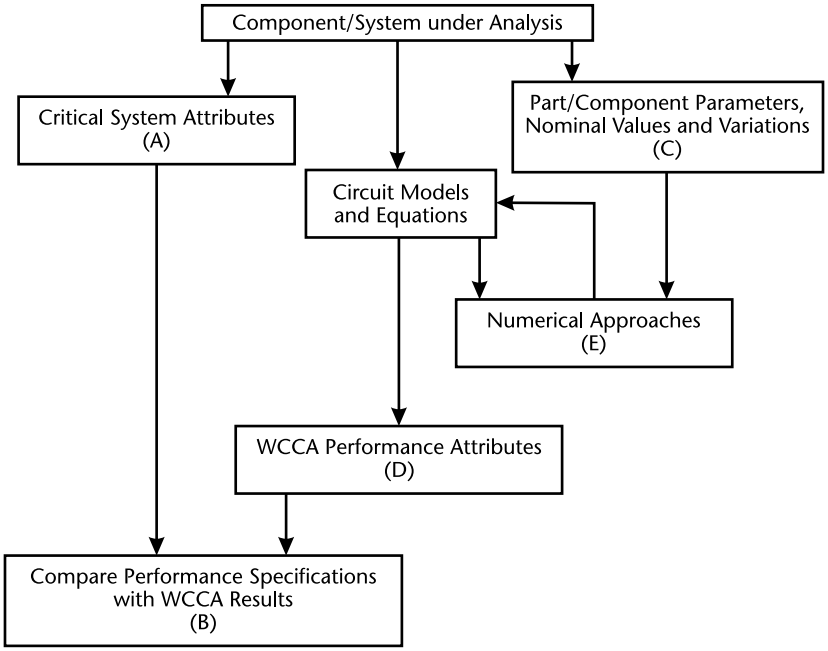



**Figure 4.20** General overview of the steps involved in WCCA.

identifies the components that are overstressed under worst-case conditions, and (2) worst-case circuit performance analysis, which analyzes the circuit to determine if it will perform as specified under worst-case conditions. All environmental conditions and part parameters are simultaneously changed to their worst realizable extremes. The circuit is then analyzed to determine if the circuit performs correctly. Analysis results should be verifiable (document or reference all information used to develop the analysis: circuit equations, part data sources, and circuit simulation tools used). If problems are found during the analysis, develop and propose potential fixes and alternative solutions.

- *Step C—Part/component parameters, nominal values, and variations:* One of the most critical steps involved in completing a WCCA is the development of

a part database. A part database is a composite of information for quantifying sources of component variation. The database typically contains of performance requirements and specifications, schematics and block diagrams, interconnection lists and wiring diagrams, part lists, theory of operation, operations environments, and operational configurations. Critical part parameters also include environmental (ambient and self- heating temperature, vibration, humidity, radiation) and component aging characteristics. Once the most probable sources of component parameter variation have been identified, the database can be used to calculate the worst-case component drift for critical parameters. The establishment of a component database is recommended for all analysis complexity levels and the analyst must be knowledgeable of the circuit and parameters being examined, including timing diagrams where appropriate. Table 4.5 provides a list of parameter variations for different types of components used in circuit design.

### 4.6.1 Sources of Variation

As evident from Table 4.5, variations in component parameters are of two types: random and bias. Random variation is not predictable in direction. However, bias is predictable given known inputs. All sources of component variation can be grouped into one of these effects. The effects are subsequently combined to give an overall indication of part variability. The addition of individual random and biased variables is as follows:

- *Bias effects:* Added algebraically;
- *Random effects:* Root sum squared- ($\pm 3\sigma$ limits of a normally distributed population).

The determination of the minimum and maximum limits of component value due to drift is as follows:

$$Worst-Case\,Minimum = Nominal\,value - \left(Nominal\,value \times \sum |Negative\,biases|\right)$$
$$- \left(Nominal\,value \times \sqrt{\sum (Random\,effects)^2}\right)$$
$$Worst-Case\,Maximum = Nominal\,value$$
$$+ \left(Nominal\,value \times \sum |Negative\,biases|\right) + \left(Nominal\,value \times \sqrt{\sum (Random\,effects)^2}\right)$$

Using the table below, determine the worst-case minimum and maximum values for a 1,200-$\mu$F CLR capacitor. These parameters are used to determine the potential resultant effect of the CLR capacitor drift on circuit applications.

**Table 4.5** Affected Parameters Versus Source of Variation

| *Component Type* | *Environmental Source of Variation* | *Parameter Affected* |
|---|---|---|
| Bipolar/field effect transistors | Temperature, radiation | $H_{FE}$ (bias), $V_{BE}$ (bias), $I_{CBO}$ (bias), $RDS_{ON}$ (bias), $V_{TH}$ (bias), $H_{FE}$ (bias), $I_{CBO}$ (bias), $V_{CE}$ (saturation) (random and bias), $V_{TH}$ (bias) |
| Rectifiers/switching diodes | Temperature, radiation | $V_F$ (bias), $T_S$ (bias), $I_R$ (bias), $I_R$ (bias), $V_F$ (bias) |
| Zener diodes | Temperature | $V_Z$ (bias, sometimes random), $Z_Z$ (bias) |
| Resistors | Temperature, humidity, aging (powered), life (unpowered), vacuum mechanical | Resistance (bias and random, random), resistance (bias), carbon composition resistance (bias and random), resistance (bias and random), resistance (bias), resistance (bias and random) |
| Capacitors | Temperature, aging, mechanical electrical, vacuum humidity | Capacitance (bias and/or random), ESR (bias), DF (bias, nonlinear), ESR (bias), capacitance (bias and/or random), capacitance, voltage coefficient, capacitance (bias, nonhermetic), capacitance (bias) |
| Linear ICs | Radiation temperature | Voltage, current offset (random), $A_{OL}$ (bias), voltage, current offset (bias and random), random, $A_{OL}$ (bias) |
| Digital ICs | Temperature, rise/fall; time, propagation, delay radiation | Propagation delay (bias) |
| Magnetics (strongly dependent on materials) | Temperature, aging mechanical | Saturation flux density (bias), permeability (bias), core loss (bias, nonlinear, nonmonotonic), saturation flux density (bias, very small), permeability (bias), saturation flux density (bias) |
| Relays | Temperature | Pull-in/dropout current/volts (bias), contact resistance (bias, secondary effect), mechanical contact resistance (bias) |

| | | | |
|---|---|---|---|
| $h_{FE}$ | Gain | $A_{OL}$ | Open loop gain |
| $I_{CBO}$ | Collector-base output current | DF | Dissipation Factor |
| $RDS_{ON}$ | On-drain source resistance | ESR | Equivalent series resistance |
| $V_{CE}$ | Collector-emitter voltage | $Z_Z$ | Zener Impedance |
| $V_F$ | Forward voltage | $V_Z$ | Zener voltage |
| $T_S$ | Storage time | $I_R$ | Reverse current |
| $V_{TH}$ | Threshold voltage | | |

*Example 4.10*

| | *Bias (%)* | | |
|---|---|---|---|
| *Capacitance Parameters* | *Negative* | *Positive* | *Random (%)* |
| Initial Tolerance at 25°C | — | — | 18 |
| Low Temperature (–20°C) | 25 | — | — |
| High Temperature (+80°C) | — | 15 | — |
| Other-Environments (Hard Vacuum) | 18 | — | — |
| Radiation (10KR, 1,013 N/cm$^2$) | — | 9 | — |
| Aging | — | — | 12 |
| TOTAL VARIATION | 43 | 24 | $(20)^2 + (10)^2 = 22.4$ |

*Solution:*

Worst-case minimum and maximum occur when the variation due to bias is of the same sign to the random variation, hence adding the effect of both:

Worst-case minimum = - |total negative bias + total random bias|

= –43% – 22.4% = –65.4%

Worst-case maximum = + |total positive bias + total random bias|

= 24% + 22.4% = +46.4%

Using the equations given above:

Worst-case minimum capacitance = 1,200 $\mu$F – 1,200 $\mu$F (|-0.43|+ 0.224) = 415.2 $\mu$F

Worst-case minimum capacitance = 1,200 $\mu$F + 1,200 $\mu$F (|0.24|+ 0.224) = 1,756.8 $\mu$F

---

- *Step D—WCCA performance attributes:* When conducting a worst-case circuit performance analysis, the key elements examined within the system depend on the functionality of the circuit or the subcircuit. Therefore, while partitioning the circuit, it is a good practice to minimize the active components to as few logical blocks as possible. It is also a good practice to include circuit and timing diagrams, as appropriate, in the analysis. Critical timing of digital circuits, transfer functions of filtering networks, and characteristics of amplifiers are examples of circuit performance elements. Table 4.6 provides a list of most important circuit parameters which should be analyzed in a worst-case performance analysis for digital and analog circuits. Finally, it will be evident that not all parts and circuits perform as specified in the technical documentation, therefore it is extremely important to compare WCCA results with the technical specifications of each part/component and document all inconsistencies.
- *Step E—Numerical analysis techniques:* Worst-case analysis can be performed using three different numerical approaches: extreme value analysis (EVA), root-sum-square (RSS) analysis, and Monte Carlo analysis (MCA). These three methods are the most accepted industry standards for WCCA. EVA is generally the easiest to apply. It involves the analysis of a given circuit/product under simultaneous worst-case parts limits and produces conservative results. RSS is a statistical approach that is more labor-intensive. However, RSS results are more realistic than EVA. Finally, MCA involves randomly selecting part parameters and analyzing the resulting system performance. MCA requires a large number of simulations (typically 1,000 to 50,000 runs) and the data for piece-part parameter statistical distributions must be known (or assumed normal). A comparative study of the advantages and disadvantages for all three approaches is presented in Table 4.7.

While the designers can choose any worst-case design technique listed in Table 4.7, the most common industry practice is to perform MCA due to the availability of software tools that can help accomplish that task. LTSPICE is an open-source SPICE modeling tool used widely and, although its user interface and features do

**Table 4.6** Important Circuit Parameters for WCCA

| *Digital Circuit Parameters* | |
|---|---|
| • Circuit Logic • Pulse Widths • Compatibility | |
| • Circuit Timing • Current Draw | |
| *Analog Circuit Parameters* | |
| • Comparator | • Oscillator |
| → Threshold Precision → Hysteresis<br>→ Switching Speed/Time Constant<br>→ Offset Stability | → Frequency, Accuracy, Stability<br>→ Output Power Level Stability<br>→ Output Impedance → Load Impedance<br>→ Phase Stability<br>→ Noise and Spurious Output |
| • Filter | • Detector |
| → Insertion Loss → Phase Response<br>→ Frequency Response → Linearity<br>→ Input/Output Impedance → VSWR<br>→ Spurious or Out-of-Band Feedthrough | → Bias Voltage → Input Impedance<br>→ Frequency Range → Output Impedance<br>→ VSWR → Input |
| • Modulator | • RF Switch (Solid State/Mechanical) |
| → Frequency Response → Phase Response<br>→ Input/Output Impedance → Linearity<br>→ Insertion Loss → Output Level<br>→ Deviation- VSWR | → Drive Requirements→ Insertion Loss<br>→ Power Dissipation → Frequency Response<br>→ Power Handling → Video Feedthrough<br>→ Switching Speed → Switch Duty Cycles<br>→ Input/Output Impedance → VSWR |
| • Multiplier | • Coupler, Circulator |
| → Output Power → Input Drive<br>→ Frequency Response<br>→ Input/Output Impedance | → Insertion Loss → Magnetic Leakage<br>→ Frequency Response → VSWR<br>→ Power Handling- Directivity<br>→ Input/Output Impedance |
| • Mixer (Converter) | • Stripline, Waveguide, Cavity |
| → Noise Figure →Group Delay<br>→ Frequency → Power Dissipation<br>→ Output Spectrum → Conversion Loss<br>→ Intercept Points → Driver Requirement<br>→ Compression Points<br>→ Terminating Impedance | → Mode Suppression → Insertion Loss<br>→ Adjustment Range, Resolution<br>→ Dimensional Stability → VSWR<br>→ Input/Output Impedance<br>→ Material Stability |

not match up to the standards of licensed software such as Altium Designer, one can easily perform MCA on smaller circuits in no time.

There are several circuit simulators available for engineering analysis and verification of electrical circuits based on SPICE technology. The reason why these simulators use SPICE technology is because each element of an electric circuit can be represented in the form of a mathematical model, where the set of mathematical models of elements and their correlations form the electric circuit model. In this manner, the circuit simulator can perform mathematical calculations on electric circuits, which allow it to perform a number of computed experiments (such as WCCA using the Monte Carlo method), with high reliability.

### 4.6.2 Numerical Analysis Using SPICE Modeling

Ideally speaking, worst-case analysis of any design should be performed on the entire circuit once the design is complete. However, in most real-life applications, this is not possible due to several reasons. First, SPICE models are not readily available

**Table 4.7** WCCA Numerical Analysis Techniques

| | *Advantages* | *Disadvantages* |
|---|---|---|
| *EVA* | • Most readily obtainable estimate of worst-case performance (best initial WCCA approach)<br>• Does not require statistical inputs for circuit parameters (easiest to apply)<br>• Database needs only supply part parameter variation extremes (easiest to apply)<br>• If circuit passes EVA, it will always function properly (high confidence for critical production applications) | • Pessimistic estimate of circuit worst-case performance<br>• If circuit fails, there is insufficient data to assess risk (modify circuit to meet EVA requirements, or apply RSS or MCA for less conservatism) |
| *RSS* | • More realistic estimate of worst case performance than EVA<br>• Knowledge of part parameter probability density function (pdf) is not required<br>• Provides a limited degree of risk assessment (percentage of units to pass or fail) | • Standard deviation of piece part parameter probability distribution is required<br>• Assumes circuit sensitivities remain constant over range of parameter variability<br>• Assumes circuit performance variability follows a normal distribution |
| *MCA* | • Provides the most realistic estimate of true worst-case performance<br>• Provides additional information in support of circuit/product risk assessment | • Requires use of computer<br>• Consumes a large amount of CPU time<br>• Requires knowledge of part parameter pdf |

for all the components used in the design, so it may be a tedious task to find one for the component that you want to use in your design. While most leading manufacturers do provide SPICE model libraries, many others do not, so one needs to be aware of this factor while performing MCA. Second, it takes a long time to assign SPICE model parameters to all circuit components and then use the SPICE model to vary the parameters and observe and record outputs for worst-case scenario. Also, performing the analysis on the entire circuit together also pushes the WCCA towards the end of the project cycle, which may not be an ideal scenario if the analysis shows results that do not match the specifications. For example, the example used in this book has approximately 941 components. Assigning tolerance values to each of them and running a SPICE simulation for the entire circuit would be an onerous task with so many component variations happening simultaneously.

Therefore, another standard industry practice is to perform worst-case analysis on smaller circuit components as they are being designed and tests which limit any potential surprises down the line. Figure 4.21 shows the Altium Designer schematic of the LDO voltage regulator circuit used in the design of SoC Platform design project for SensorsThink. Let us use this circuit as an example to estimate the worst-case voltage output range for this circuit. In our design, we have used input voltage tolerance of 2% and component tolerance of 1% as shown in Table 4.8.

In order to perform WCCA using MCA, we used LTSPICE to run a dc sweep of the input voltage from 2V to 3.3V, as shown in Figure 4.22.

Ideally, the dc sweep should be run from 0 to 3.3V, but since the ideal output voltage is 2.8V, we are only interested in input values of 2.8V and higher, as lower input values in the dc sweep analysis only give a zero output. We are interested in checking the maximum and minimum output swing around the ideal output voltage of 2.8V. Moreover, to keep it simple to understand, we have assumed that the

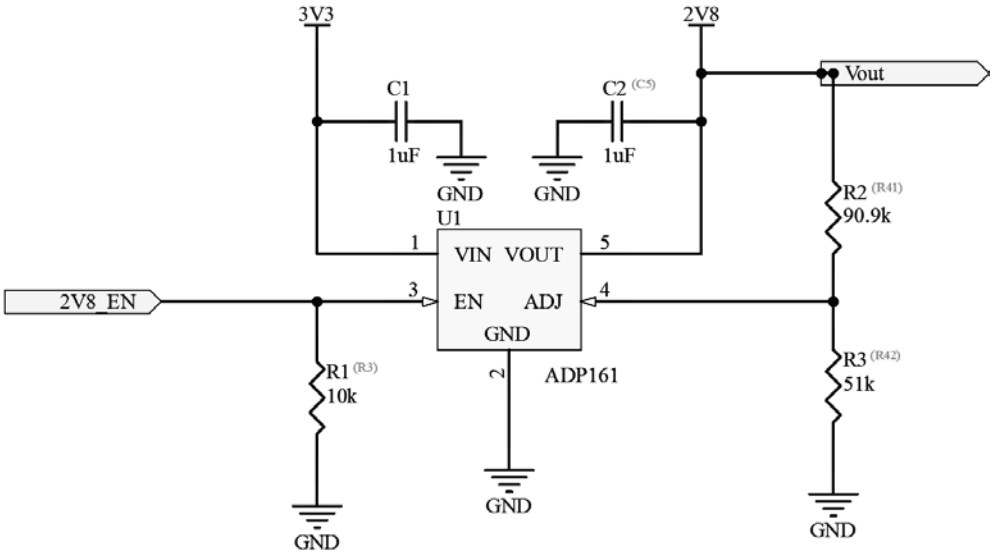



**Figure 4.21** Schematic of an LDO voltage regulator circuit used in SoC Platform project.

**Table 4.8** Component Tolerance Values for LDO Voltage Regulator Circuit Shown in Figure 4.21

| *Component Name* | *Ideal Value* | *Tolerance* |
|---|---|---|
| Supply Voltage | 3.3V | ±2% |
| R1 | 10 kΩ | ±1% |
| R2 | 90.9 kΩ | ±1% |
| R3 | 51 kΩ | ±1% |
| C1 | 1 $\mu$F | ±1% |
| C2 | 1 $\mu$F | ±1% |
| U1 | Voltage regulator | 0% (ideal) |

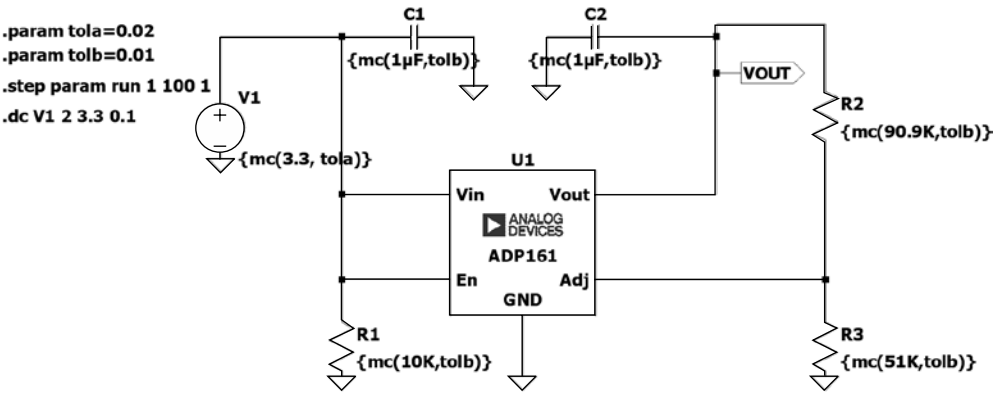



**Figure 4.22** LTSPICE schematic of an LDO voltage regulator circuit with SPICE modeling components.

ADP161 regulator IC has 0% tolerance (in other words, it is assumed to be ideal). Figure 4.23 shows the plot of MCA simulation results in this circuit. It is evident from the plot that the variation in output voltage of the regulator for the given tolerance levels is between 2.75V and 2.81V, which is quite good for most sensor-based applications.

Let us now simulate the same circuit with higher component tolerances and evaluate the impact on the worst-case output of this voltage regulator. The new values of tolerances are given in Table 4.9. For the sake of simplicity, we have again assumed that component U1 is ideal (0% tolerance) and increased the tolerance of voltage supply and resistors/capacitors to 3% and 5%, respectively.

The LTSPICE schematic and simulation results for the dc sweep analysis are shown in Figures 4.24 and 4.25, respectively.

It is quite evident from the simulation results that higher tolerance values of the components result in much greater fluctuations at the output in the worst-case scenario. For the increased tolerance levels, the variation in worst-case voltage output

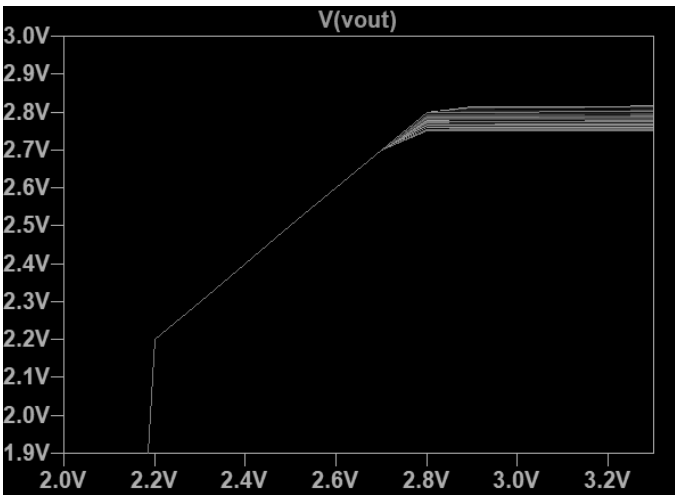



**Figure 4.23** LTSPICE dc sweep simulation of an LDO voltage regulator circuit with voltage supply tolerance = 2% and resistance/capacitance tolerance = 1%.

**Table 4.9** Modified Component Tolerance Values for the LDO Voltage Regulator Circuit Shown in Figure 4.21

| *Component Name* | *Ideal Value* | *Tolerance* |
|---|---|---|
| Supply Voltage | 3.3V | ±3% |
| R1 | 10 kΩ | ±5% |
| R2 | 90.9 kΩ | ±5% |
| R3 | 51 kΩ | ±5% |
| C1 | 1 $\mu$F | ±5% |
| C2 | 1 $\mu$F | ±5% |
| U1 | Voltage regulator | 0% (ideal) |

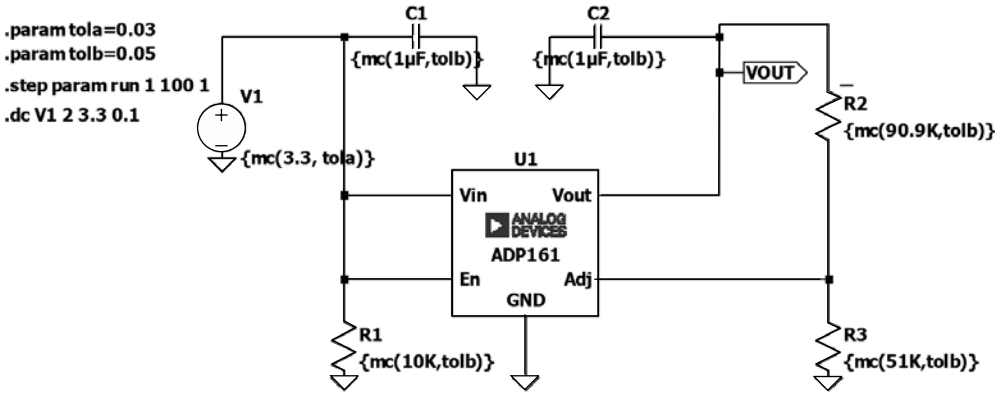



**Figure 4.24** LTSPICE schematic of an LDO voltage regulator circuit with higher tolerance values for SPICE modeling components.

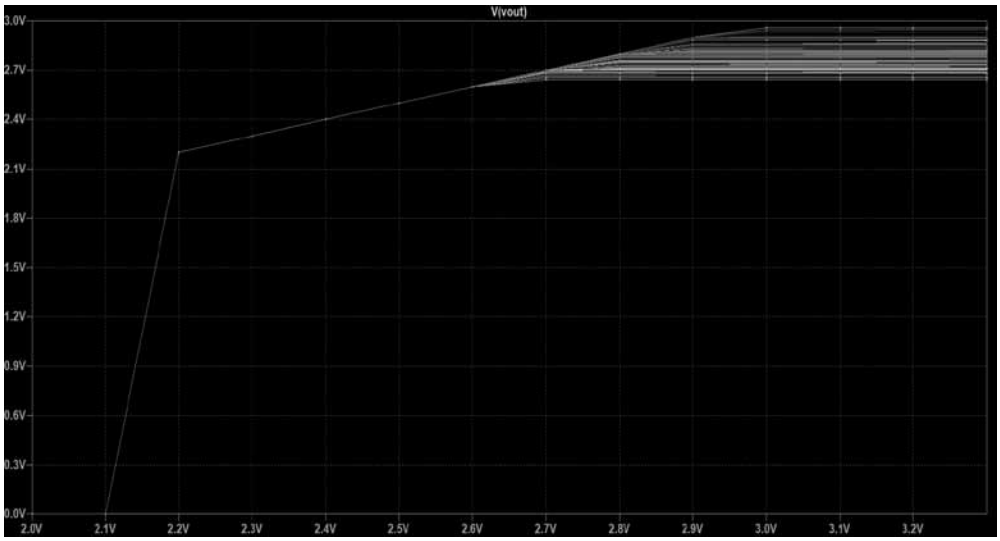
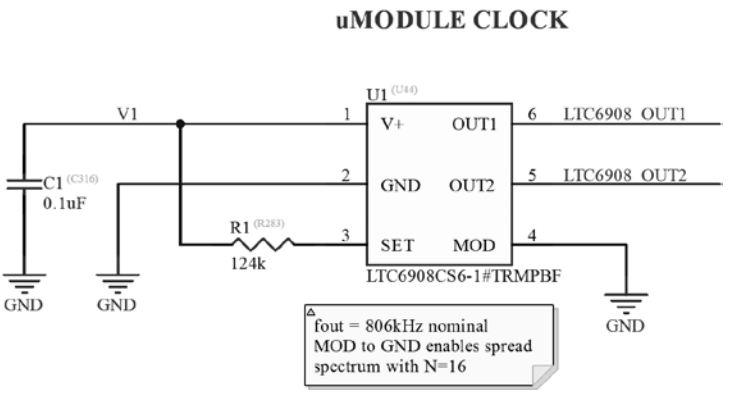



**Figure 4.25** LTSPICE dc sweep simulation of an LDO voltage regulator circuit with voltage supply tolerance = 3% and resistance/capacitance tolerance = 5%.

is between 2.64V and 2.96V, which is a significant variation from the ideal 2.8-V output level.

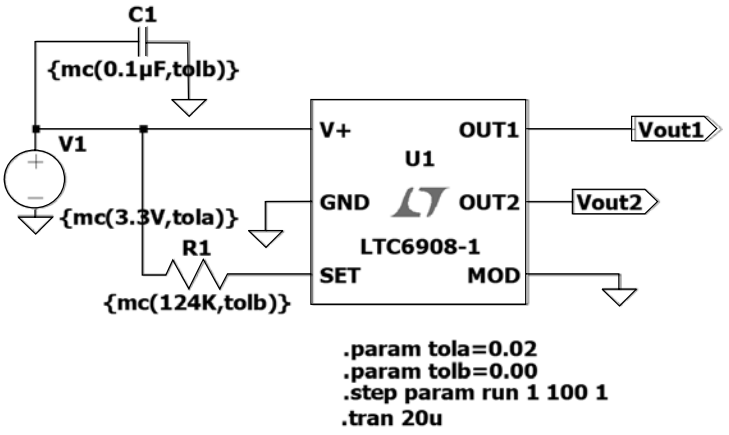
*Example 4.11*

A schematic diagram of a clock generator circuit used in the design of the SoC Platform design project for SensorsThink is shown below. We will perform Monte Carlo analysis for the worst-case performance of this circuit if the supply voltage tolerance is 2% and resistance or capacitance tolerance is 2%. Assume that the LTC6908 oscillator has 0% tolerance (ideal).

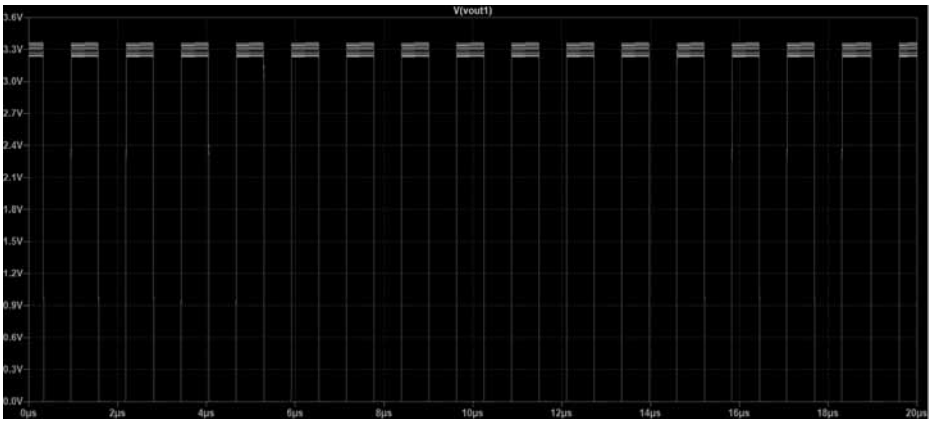



*Solution:*

In order to understand this better, let us run the Monte Carlo simulation for this circuit in two steps. In the first case, we run a transient analysis of the circuit and observe the impact of voltage supply variation while keeping the other components ideal. The LTSPICE schematic of this circuit with component parameters and tolerances is shown below.

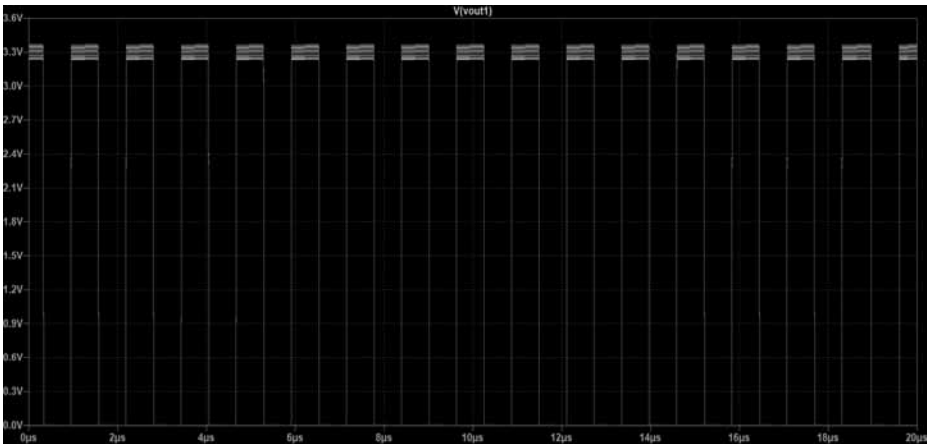



*Running transient analysis on this circuit analysis for 20 µs with* voltage tolerance 2% and 0% (ideal) for all other components gives the following output.



It can be observed that 2% supply voltage variation only impacts the output voltage level and it varies between 3.2V to 3.4V. These values are approximately equal to the voltage variation calculated theoretically using $V_{OUT} = V_{IN}*(1 \pm 0.02)$.

Running transient analysis on this circuit analysis for 20 $\mu$s with voltage tolerance 2% and 0% (ideal) for all other components gives the following output.




It can be observed that 2% supply voltage variation only impacts the output voltage level and it varies between 3.2V and 3.4V. These values are approximately equal to the voltage variation calculated theoretically using $V_{OUT} = V_{IN}*(1 \pm 0.02)$.

---

It is important to note that the LTSPICE simulation examples given above assume that the designer has good knowledge of SPICE modeling and circuit analysis. It is out of scope for this book to present a tutorial on SPICE modeling, however, there are some very reference materials in print and online that can get one started on using SPICE modeling.

Finally, it is important to understand that reliability and worst-case analysis for a given circuit or system are only as good as the robustness of the system specifications and the reliability of the individual components. So while ideally we would like everything to be 100% reliable and fault tolerant, it is never going to happen due to all the factors affecting the design cycle as discussed earlier. All we can do is to manage the reliability of a circuit within a reasonable range that takes into account the application and the operating environment. If the design itself cannot help in bringing down the reliability numbers within a reasonable range, then the only other method (more expensive) is to provide redundancy to improve the fault tolerance levels of the circuit or system.

## Selected Bibliography

Avizienis, A., "Fault Tolerance and Fault Intolerance. Complimentary Approaches to Reliable Computing," *Proc. 1975 Int. Conf. Reliable Software*, Los Angeles, CA, April 21-27, 1975, pp. 458–464.

Avizienis, A., "N-Version Approach to Fault Tolerant Software," *IEEE Software*, Vol. SE11, No. 12, December 1985, pp. 1491-1501.

Ireson, W. G., C. F. Coombs, and R. Y. Moss, *Handbook of Reliability Engineering and Management,* 2nd ed., New York: McGraw-Hill, 1996.

Laprie, J. C., "Dependable Computing and Fault Tolerance: Concepts and Terminology," *Proceedings of 15th International Symposium on Fault-Tolerant Computing (FTSC-15)*, 1985, pp. 2–11.

LTspice, Analog Devices.

O'Connor, P. D. T., and A. Kleyner, *Practical Reliability Engineering,* New York: Wiley, 2012.

RAC Publication, CRTAWCCA, *Worst Case Circuit Analysis Application Guidelines,* 1993.

# About the Authors

**Adam Taylor** is a world-recognized expert in design and development of embedded systems and field programmable gate arrays (FPGAs) for several end applications. Throughout his career, Mr. Taylor has used FPGAs to implement a wide variety of solutions from radar to safety-critical control systems (SIL4) and satellite systems. He has also worked in image processing and cryptography. Mr. Taylor has held executive positions, leading large developments for several major multinational companies. For many years, he held significant roles in the space industry, including as a design authority at Astrium Satellites (now Airbus Space) in the payload processing group for 6 years and as the chief engineer of e2v Space Imaging, where he was responsible for several game-changing projects, for 3 years.

Mr. Taylor is the author of numerous articles and papers on electronic design and FPGA design, including over 400 blog entries and more than 25 million views on how to use the Zynq and Zynq MPSoC for Xilinx. He is a Chartered Engineer, a Senior Member of the IEEE, a Fellow of the Institute of Engineering and Technology, a visiting professor of embedded systems at the University of Lincoln, and an Arm Innovator. Mr. Taylor is also the owner of the engineering and consultancy company Adiuvo Engineering and Training (www.adiuvoengineering.com).

**Dan Binnun** has been involved in product development for nearly his entire career, spanning companies as small as 12 people to large, multinational, publicly traded corporations. He was exposed early in his career to product development and research and development processes and has a wide range of experiences that has given him an engineering process and development perspective.

Mr. Binnun has expertise in electrical system design, embedded system architecture, and printed circuit board design. He founded E3 Designers to aid his growth as an engineer. He graduated from the University of Hartford in 2007 with a bachelor's degree in electrical engineering. Since graduating, Mr. Binnun has continued his education in engineering by taking courses in EMI/EMC and high-speed digital design. You can connect with him on LinkedIn at www.linkedin.com/in/dbinnun/ or follow E3 Designers at https://www.linkedin.com/company/e3-designers.

**Saket Srivastava** earned a Ph.D. in electrical engineering from the University of South Florida, Tampa, in 2008. Dr. Srivastava began working at the University of Lincoln in 2013 and is currently a senior lecturer and program leader in School of Engineering. He was a postdoctoral research fellow in the School of Electronics and

Computer Science at the University of Southampton, United Kingdom, from 2008 to 2010 and was an assistant professor at the Indraprastha Institute of Information Technology, Delhi, India, from 2010 to 2013. His research interests include probabilistic modeling of emerging nanoelectronics devices, reconfigurable hardware design, embedded systems, and machine learning. Dr. Srivastava has published more than 30 papers in leading IEEE/ACM journals and conferences. He has also led several U.K. and multinational collaborative research projects funded by various RCUK funding agencies and other industry partners (http://staff.lincoln.ac.uk/ssrivastava).

# Index

## F

## G



## H

## I



## J



## L



## M

## N



## O



## P

## R

## S



## T

## U



## V



## W



## X

## Z