ASIC Design Practices from a Firmware Perspective

by
Gary Stringham
gary.stringham@hp.com

#466 at the Embedded Systems Conference – San Francisco 2004

Table of Contents

1 Introduction

ASIC engineers design their ASICs with varying levels of understanding to the impact of the design on the firmware writer who has yet to write the code to set up and control the ASIC. Months later when the firmware writer has an ASIC and is ready to start coding, the ASIC designer has basically forgotten the ASIC and is on to the next project.

Discussed here are tips and techniques that can be used across several areas of ASIC design that will help make the firmware writer's job easier when that time arrives. First, *Assumptions* and *Definitions* are mentioned to establish the context of the paper. The paper will then go over the several areas of ASIC design, such as *Firmware Involvement*, *Documentation*, *Register Layout*, *Interrupts*, and *Errors*.

1.1 Assumptions

It is assumed that the reader is familiar with ASICs, their operation and design. This includes registers, how firmware interfaces with it, and interrupt handling.

While most of the ideas mentioned here may seem like basic practices that all designers employ, in practice, almost all of these have been found lacking in one or more ASIC designs. It is the intent that this will help spread some best practices and generate new ideas for most readers.

1.2 Definitions

The following terms are defined in the context of this paper:

- ***Application*** – This is software that runs on top of an operating system. Several of them are running in an embedded system. Technically, it could range from one *Application* doing everything to many *Applications* each doing one thing. For this paper, many *Applications* each doing one thing will be used. Some *Applications* interact directly with the *Drivers*. For this paper, one *Application* will interact with only one *Driver*. Applications typically run in protected mode which means they have limited privileges and cannot directly access hardware. Applications can be written in C, C++, assembly, or other languages.
- ***ASIC*** – Application Specific Integrated Circuit. It is a chip that has been specifically designed to serve a specific purpose. Most embedded systems contain one or more custom *ASIC*s. (a.k.a – Hardware.)
- ***Block*** – A block is one portion of an *ASIC* doing a dedicated function, such as a UART. An *ASIC* can contain more than one block, such as a UART, a display controller, and a JPEG compressor.
- ***Driver*** – The software that directly interacts with the *ASIC*. The *Driver* is the interface between the *Application* and the *ASIC*. It writes to the registers to configure and invoke actions. It reads from the registers to get results and status. It responds to interrupts generated by the *ASIC*. Typically one *Driver* interacts with one *Block* on the *ASIC*. *Drivers* run in privileged mode which means it operates down in the kernel of the operating systems and can directly access hardware. This can be written in C, assembly, or other languages.

- ***Engineer*** – The person who designs, programs, owns, and debugs. There are *ASIC Engineers* and *Firmware Engineers*, each owning their respective *ASIC Block*, *Driver*, or *Application*. (a.k.a – Designer, Writer.)
- ***Firmware*** – The software that runs on the embedded system controlling the device. It includes the *Applications*, *Drivers*, and operating systems. A *Firmware Engineer* could own one or more *Applications*, one or more *Drivers*, or both.
- ***Sub-block*** – This refers to sections of a *Block*, such as a DMA controller inside a compressor *Block*.
- ***Thread*** – A *Thread* of execution is a portion of firmware that is working on one task. It could be an *Application*, a *Driver*, or part of the operating system. Many *Threads* may be active but only one *Thread* executes on the CPU at a time. *Threads* are swapped in and out based on their priority, when it is their turn, and if they have work to do.

2 Firmware Involvement

The key to designing an ASIC is to get the firmware engineers involved early. They are the customers that will be using the ASIC. Unfortunately, getting them involved early is often difficult to do because the ASIC design has to start several months before the firmware engineers will get parts. The firmware engineers are busy with something else so it is difficult to get some bandwidth from them this early in the development cycle. But if it can be done, it really helps. This up-front investment pays off months later when parts arrive; the firmware engineers are already somewhat familiar with the new ASIC.

When the parts do arrive, the roles are reversed. The firmware engineers are trying to work with it while the ASIC engineers have mainly forgotten it and have moved on to new projects.

A good way to get firmware involved is to include them in all design reviews and checkpoints. This includes getting their signature at the checkpoints. This starts at the beginning when the overall ASIC design is being worked on, including what blocks will be in the ASIC, whether they are new blocks or blocks leveraged from prior ASICs, modified or unchanged.

Eventually each block is designed. The first thing that should happen at the block level is the writing of the documentation for that block, such as an ERS (External Reference Specification.) This should contain all the details that the firmware engineers need to know to write the application and driver for that block. More details are discussed below in *Documentation*. It is at this step that the firmware engineers will probably have the most influence in the design. This is the chance for them to comment on register layout, block functionality, firmware interactions, etc. This introduction is also a good start to get them acquainted with the block, which will make their job easier months later when they start to write the firmware for it.

As the design progresses the ASIC designer will come across problems to be solved or changes that could be made. By seeking the opinion of the firmware engineers as appropriate, smarter decisions could be made.

When the parts arrive and the firmware engineers are writing code for it, the ASIC engineers are there to answer questions and to help solve problems. Problems can be put into three generalized buckets.

- The firmware engineers' incorrect assumption on how the block works.
- Inadequate or erroneous documentation.
- Defect in the ASIC.

The first two are easy to fix by improving the documentation. The third one is a challenge because the engineers first have to determine that it is not one of the first two. Once it is determined that it is the third, then two things need to happen. First, the defect in the ASIC must be understood. Secondly, a firmware workaround can be attempted and hopefully, there is a firmware workaround. Otherwise, that block on the ASIC may be useless. Sometimes, the workaround is fairly simple and straight forward. Other times, it is fraught with problems and the solution cannot guarantee 100% correct behavior under all conditions. Many firmware workarounds are thought up by the ASIC engineers because they are most familiar with their block and its capabilities, and know what possible solutions might exist.

When problems are encountered, a fix in the ASIC should be worked on. There might not be a next revision of the ASIC to roll in the fix, but it should be fixed for the next generation of the ASIC. A correctly operating function in the ASIC is much better than a workaround in firmware from a long-term maintenance and support point of view. The same applies with design problems. If something can be done easier in the block, then roll that into the next generation of that block.

In one ASIC, we had a particularly complex block with several problems, each of which required understanding the root cause and then coming up with a satisfactory workaround. An analysis of the driver code showed that more than 10% of the lines in the driver were for workarounds. Because of the time required to solve those problems and come up with workarounds, it is estimated that about half of the time spent on the driver was spent on workarounds. That is significant for a driver that took several months to complete.

3 Firmware Architecture

Good design practices for both firmware and ASICs are similar. They involve writing isolated tasks with well-defined interfaces. These individual tasks are tested thoroughly before putting them together to test them as a system. Information is on a need-to-know basis; other tasks do not have access to one task's data unless necessary. This means that information that one task has is not necessarily known by another.

3.1 Division of Firmware

Figure 1 is a general illustration of how the firmware is divided. For a given block in the ASIC, there is typically a corresponding driver and application. The application communicates with the other firmware applications which control the overall system.

Drivers deal with low-level access to the ASIC block. They handle the register reads and writes, pass data back and forth, and deal with interrupts. Applications deal with what to pass back and forth and when. For example, suppose Block A is a UART for RS232 communication. Driver A would handle passing the characters back and forth but it does not look at the characters. It is

just the middleman. Application A looks at the incoming characters and generates what the outgoing characters should be.

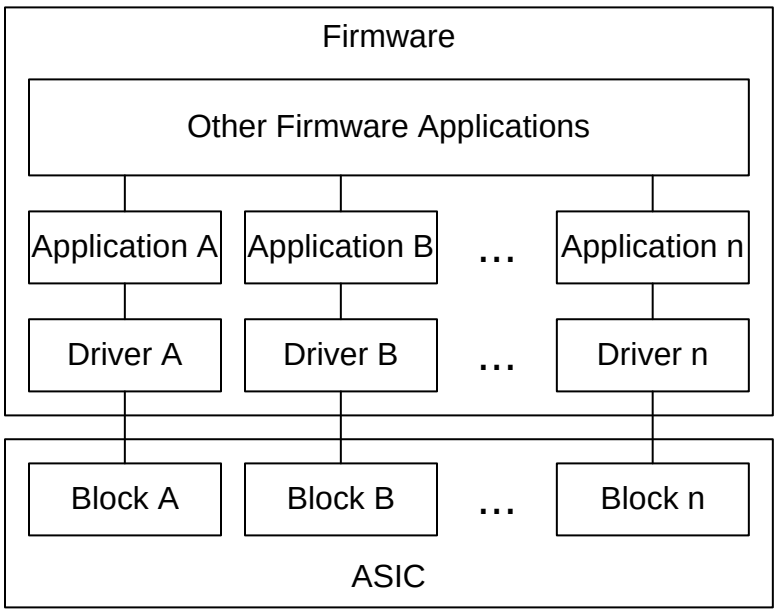

Figure 1 - A General Illustration of ASIC and Firmware

Firmware designers divide up the drivers and applications based on the design of the ASIC. Sometimes when ASIC designers make changes to the block layout, it affects the architecture of the firmware. We had a case where the ASIC designer added a feature to a block. Supporting this feature required making major changes to the driver and three applications. Generally new features are good. However, in this case, this feature was not usable in all operating modes. So both the old and the new methods had to be supported by the driver and applications.

3.2 Threads

The CPU executing the firmware can only work on one thread at a time, be it an application, a driver, or some operating system thread. The operating system handles the switching of the CPU back and forth between several threads. A thread is kicked off of the CPU either because it has run out of things to do at the moment or because it has used up its allotted time. In the case of the latter, it is interrupted in the middle of whatever it was doing. Its context is saved away for resuming later. This works well for each thread with its own data. If data is shared across more than one thread, care must be taken to make sure one thread is done with the data before another thread uses it. Threads do not know if they have been kicked off or not, nor does it typically need to know that.

Hardware and firmware designers have to think differently. For hardware designers, everything is happening in parallel, every state machine and memory cell operates together on the same clock edge. Combinatorial logic is racing down many paths until it settles down. Firmware operates in serial. Firmware engineers know that when their thread has the CPU, it has it exclusively. In cases of a thread working in critical code, it can disable interrupts to be sure that no other thread interrupts it. After that section of code is completed, it re-enables interrupts, allowing other threads to interrupt as needed.

Threads have priorities. Higher-priority threads will take over CPU from lower-priority threads. In some cases, a lower-priority thread might not be able to respond to its block's interrupt for a while because something more important is happening. Threads of equal priority take turns, running during its time slice before being swapped out to allow another one to execute.

3.3 Information Passing

When information coming from one block needs to be conveyed to another, it typically has to pass through the driver, through the application, through other firmware applications, through the destination application, then the destination driver, and eventually to the destination block. Since each one of these are threads, it can take some time for each one to get CPU time to process the message and pass it on. In other words, a piece of information in one block can take a considerable and indeterminate amount of time to get it to another block.

There are under-handed ways of quickly getting the message across when urgency is required, but those are used sparingly and considered exceptions to proper communication channels.

4 Documentation

One of the best things that can be done to help the firmware engineers is to have complete and accurate documentation for the ASIC. Having accuracy is obvious. The firmware engineer cannot write correct code if the register addresses, the bit locations, or the functions listed are incorrect. Here are some reasons why inaccuracy creeps into a document.

- The document is written before the block is and the engineer does not refer to the document when writing the block.
- As the engineer is writing the block and discovers that the design needs to change, the document does not get updated.
- The block is "highly" leveraged from a previous generation ASIC and the document is not updated (or completely updated) to reflect the few changes that are made.

4.1 What to Include

Completeness can seem overwhelming. But here are a few things to include.

- Include two different types of documentation, reference and tutorial. A reference lists all the registers in some logical order and contains a description of each of the bits in that register. This is useful when the firmware engineer needs a reminder of little details, such as the position of a particular bit. A tutorial explains how to do something, including sequence order and register settings. "To invoke this function, the following registers need to be configured as specified before initiating the function by setting that bit in that register." A few examples usually suffice to give the firmware engineer an idea of how it is done. The engineer can then make variations as required. Also include tables that illustrate the settings of several interrelated bits for each type of work.
- List all the registers that firmware needs to access. Also include registers that firmware should not need to access for normal operation but might be needed to help solve problems.

- Document interactions. "If this bit is set, that bit in that register is ignored." "The contents of that register are read when this function is invoked, so it should be configured before invoking this function."
- Document externally-visible state machines. This might include having the flowchart in the document too. If firmware needs to wait or do something different under certain conditions, the flowchart will help illustrate those conditions, how it got in there, and what will get it out of there. Flowcharts of state machines are especially useful when trying to debug why the firmware or the ASIC does not work as expected.
- When describing a bit, do not simply state something like, "The XYZ bit enables the XYZ mode." What the XYZ mode is may not be understood by the firmware engineer. Also include what the XYZ mode is and when it should be enabled or disabled. Include any interactions with other modes.

As the firmware engineer reviews the document, there will be questions. Those questions are indications of vague and insufficient information. Update the document with the answers to those questions. This will help future engineers get their questions answered without having to go back to the ASIC engineer.

4.2 Changes Section

One of the most-useful sections of a document is at the beginning where revision changes are listed from the previous generation of the block. This really helps those who are familiar with the previous generations so that they do not have to read the whole document to find the few differences. Best practices is to not only include the differences from the previous generation but about three or four generations prior to that, starting with the current ASIC and going back.

The ABC block in the Labrador ASIC is different than the Poodle ASIC in the following ways.

- …

The ABC block in the Poodle ASIC is different than the Chihuahua ASIC in the following ways.

- …

The ABC block in the Chihuahua ASIC is identical to the Wolf Hound ASIC.
The ABC block in the Wolf Hound ASIC is different than the Collie ASIC in the following ways.

- …

4.3 Good Documentation

One time my vacation had to be altered because my driver would not work correctly due to repeatedly finding additional problems with the ASIC, and yet my driver was needed by others. My manager asked why there was no backup engineer to take over so I could go on my vacation. I said that the block was too complex and the documentation inadequate. A few minutes later, another engineer owning the driver for a similar block walked up and the manger asked him the same question. He gave the same answer.

By having better documentation, especially for a complex block, it makes it easier for a firmware engineer to come up to speed on it, especially a backup engineer when needed.

There will be more comments about documentation in other sections below as it applies to those sections.

5 Register Layout

Firmware's interface to the ASIC is via registers. Those registers are accessed via addresses. The ASIC is assigned an address range and all registers are within that range.

5.1 Address Space

Typically a custom ASIC will have more than one block, such as a USB device and a Compact Flash memory controller. Each block should also have its own address range that does not overlap with other blocks. Each block should have extra address space within its range to be able to add new registers in the future. The base address for each block is an offset from the ASIC's base address. Within each block each register is assigned an address, which is an offset from the block's base address. When the block is leveraged to the next generation ASIC, the block's offset can move within the ASIC range as necessary. However, the register offsets from the block's base address should stay the same.

Having a base address for the block (which is the ASIC's base address plus the block's offset) is easy for the driver writer. One line of code specifies the block's base address. When the block is moved onto a different ASIC, it is a one-line change in the driver to specify the new base address for the block. No further register address changes are needed as long as the registers' offsets did not change.

Within a block, there may be sub-blocks, each with their groupings of functions within the block, (such as a DMA within a compressor.) Likewise, the registers should be grouped together with some space in between the groups to allow for new and future registers in to that sub-block.

5.2 Bits in a Register

In the following discussion on bits within a register, 32-bit registers are used but it also applies to 16- and 8-bit registers or any other size. Bit 0 is the LSB (Least Significant Bit) and bit 31 (or 15 or 7) is the MSB (Most Significant Bit.)

Registers have bits or fields that are written to or read from. Zero or more bits of the register are used, but not necessarily all of the bits. When first populating a register with bits, it should start at the LSB, bit 0. If only two bits in a register are defined, then starting with the LSB allows firmware to use as masks, 0x1 and 0x2, instead of 0x80000000 and 0x40000000. As more bits need to be added, add them behind the others. Having all the bits next to each other towards the LSB end is a general guideline. Reasons for doing something different are discussed next.

While laying out a register, think about possible future needs and make room for them. For example, suppose a register is going to use five bits to show the levels of five signals coming into the ASIC. The register might also have bits showing other status regarding the block. Maybe it would be useful to allocate eight bits, making three of them "reserved for future use," then allocating other bits as needed. We did this recently and the next ASIC is now going to use all eight positions.

Once an ASIC has been released, then leave the bit positions where they are on the next ASIC. If some bits are no longer needed, take them out and leave holes there. New bits are to be added

on the end. Do not shift bits around to make them look nice without consulting with the firmware engineer. Drivers for the blocks tend to be used on more than one generation of ASIC. If bits are shifted around, then the driver has to add logic to first determine which generation is being used, then figure out which mask to use. "If (Asic = B), then mask=0x040, else mask=0x100."

Unused bit positions should ignore any data that firmware writes to those positions. This allows firmware some flexibility in data formats and improves CPU performance if they don't have to mask off the extra bits every time. When reading a register, unused bit positions should always return 0. Again, this allows more flexibility in data formats and improvements in CPU performance for the firmware. If there are "hidden" bits that contain some debugging or testing function, they must be documented so that the firmware engineer knows to make sure they are 0. Better yet, those bits belong in a different register, such as a test register.

5.3 Types of Bits

Ideally, the bits of a register should be of the same type. For example, one register could contain read-only bits with various status information about the block. The block is what is changing the value of those bits. Another register could contain configuration values that only the firmware can change; it is "read-only" from the block's point of view. Interrupt status should be in one register, interrupt enables in another. An example of a register that could contain mixed types is status bits (read-only) and interrupt status bits (read status, write a 1 to clear the interrupt.) The writing to acknowledge the interrupt does not affect the read-only bits. An example of a bad mix is putting interrupt status bits with read-write configuration bits. There is a potential of inadvertent acknowledging of interrupts or changing configuration values. If the firmware engineer is not taking that into consideration when coding, then it could lead to bugs that would be difficult to isolate and track down.

Consider what registers need to be accessed only at bootup versus those that need to be accessed regularly during the operation and divide them up that way.

Firmware initiates an action in the block with a register write. It is typically done by setting a bit in a register. It is preferred that for those types of bits, that firmware can only write a 1 and the block can only write a 0. If firmware has to write a 1 followed by a 0, there is the risk that, depending on how the block works, the 0 might be written before the block notices it. We have one case where firmware has to write a 1, another 1, and another 1, then finally a 0, to make sure a 1 is in the register long enough for the worse-case delay of the block noticing it. So the preferred way is to have firmware write the 1 and only the block can set it back to 0 when done. Firmware can read the register to determine if the task is done or not.

Several bits in a register could be used to launch different actions in the block, allowing the firmware to launch more than one of them at a time. However, if there are actions that should not be run at the same time, put them in different registers. A way of doing this is to have a register with zero bits. The simple act of writing to the register (regardless of what the data is) invokes the action. A different address then must be used to invoke a different action. In this case, there should be a status bit in another register that indicates that the action has been invoked.

5.4 Read-Modify-Write

In registers that contain several configuration bits, firmware often needs to change one while leaving the others alone. One way firmware does this is by keeping a shadow copy of that register in its own data structures. It modifies the desired bit in its copy, then writes the new value out. Another way is to first read the register, modify the desired bit, then write the new value out. If the register is only used by the one driver, then a shadow copy can be used. However, for registers potentially used by more than one driver (such as the ASIC-level interrupt enable or GPIO ports,) then each driver must use the read-modify-write method.

The read-modify-write is a potential dangerous area for firmware. For example, suppose one driver does the read, then modify, but then is interrupted by a higher-priority driver. It wants to modify the same register. It reads, then modifies, then writes out its change. It is done and the first driver resumes. The first driver would not know that it has been temporarily kicked off, nor would it know what other task was running, nor would it know that the other task just modified that register. It just simply resumes and writes its modified copy of that register out, thereby wiping out the changes that the higher-priority task made.

This rarely happens, but when it does happen, it can hang the system and it is very hard to isolate and track down. To prevent this, each and every driver must disable interrupts when doing a read-modify-write operation of a register that another driver can access. If one driver does not disable interrupts, then it could clobber other drivers that manage to interrupt it at the right time. It is for this reason that where possible, registers need to contain bits that only one driver would access.

There are ways that hardware can fix this problem for firmware. One way is to break out the register into three. One is a read-only register that contains the current setting of all the bits. Another is a write-only set register where a 1 in certain bit positions sets the specified bits. The third is a write-only clear register where a 1 in certain bit positions clears the specified bits. This allows different drivers to set and clear their respective bits without first having to read the current contents of the register and without having to disable interrupt. If address space is tight, the read-only register could share the same address as one of the two write-only registers.

5.5 Write-only Registers

There should be no write-only registers where the firmware cannot see if the write worked or not. Firmware should be able to read that register back (or some other register) to determine if the write was successful.

We had a case of an ASIC that had three write-only configuration registers. When the driver wrote to them, the desired behavior did not occur. Since what we read back was not what we wrote out, we assumed the document had a bad address listed. Eventually we were able to determine that it was a write-only register but the underlying block function was defective.

5.6 Documenting Registers

The following is an example of how registers can be documented.

XYZ Configuration Register – 0x0064

Position	*31*	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Bit	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*T*	*T*	*T*	*T*	-	-	*R*	*A*	*r*	*r*	*r*	*r*	*r*	*r*	*S*	*E*
Default	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*X*	*X*

E – Eating: True if the Eating signal is asserted.
S – Sleeping: True if the Sleeping signal is asserted.
r – Reserved for future signals.
A – Active: True if the block is actively processing.
R – Reset: True if the block is performing a reset.
T – Timeout: Contains the timeout value that the block should use. It is in units of milliseconds. Default is 2. Valid range is 1-10. Invalid values written could cause undefined behavior.

Be consistent in how the registers are documented, even across all the blocks in the ASIC. Put the LSB on the right side; it makes it easier to look at it and write its corresponding hex value. For example, a mask for the Reset bit shown above is 0x200. As shown in the example the power-on defaults are shown for each one. The *X* indicates that it is not known. In this example, it is not known because it reflects whatever level the signal is coming in to the ASIC.

5.7 ID Codes

ASICs should have (most do) a read-only register that contains an ID code identifying the ASIC. ASICs should also have a revision register, identifying which version of the ASIC it is. This allows the same firmware to operate on multiple ASICs. This is especially useful when more than one version of the ASIC is in use.

Taking this concept farther, each block within an ASIC could contain its own ID and revision codes. This allows firmware to base things on the block itself, and not have to decode which ASIC ID maps into what block ID, especially when the same block is used across many ASICs.

6 Interrupts

Briefly, interrupts are conditions that the ASIC detects that it needs to alert firmware that the condition has occurred. The ASIC provides a means for firmware to control whether a particular condition will actually propagate to firmware or not. Once firmware has been interrupted, it then has to acknowledge to the ASIC that it received it, by writing something to some register.

There have been enough differences, problems, and confusion with regards to interrupts that it merits its own section. The biggest problem is a lack of consistency and clarity with regards to interrupts. What one ASIC calls Enable, another ASIC calls Mask. On one ASIC a 1 is written to ack (acknowledge) the interrupt whereas on another, a 0 is written. This leads to confusion and problems since the firmware writer could easily write the code the wrong way.

6.1 Hierarchical Interrupt Structure

From an ASIC point of view, interrupts should be hierarchical. There should be one interrupt module at the ASIC level (top level.) Each bit in that module corresponds to each block on the ASIC. Each block should have its own interrupt module, which contains one or more interrupts from that block. Figure 2 illustrates the ASIC- and block-level interrupt modules.

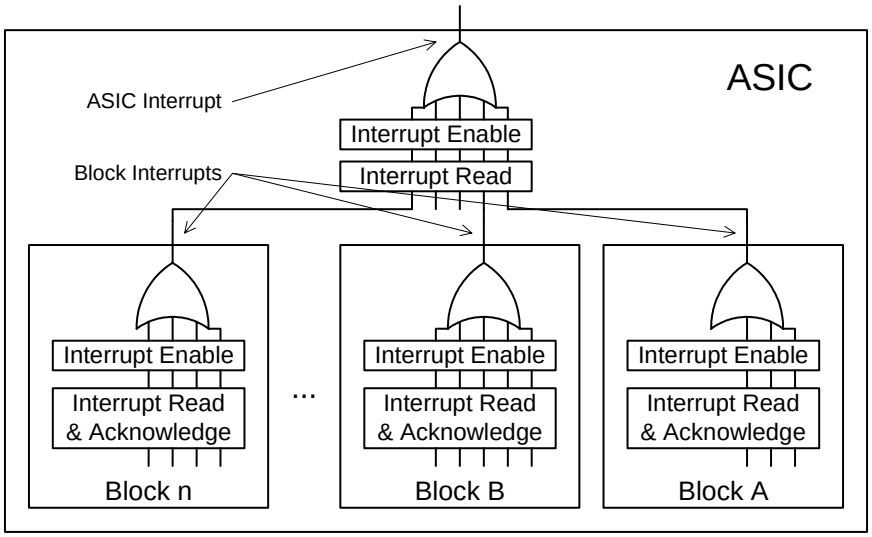

Figure 2 - Hierarchical Interrupt Structure

This hierarchical approach allows a driver to control its own interrupts. It can control which of all the block's interrupts are allowed to propagate. It can read which of the block's interrupts did occur. If any one of that block's interrupts is pending, then the block's bit in the top-level interrupt module is pending. If any of the top-level bits are pending, then the ASIC's interrupt line to the CPU will be asserted.

This hierarchical approach helps the firmware's interrupt architecture. A front-line interrupt handler in the firmware will be invoked when the ASIC generates an interrupt. The front-line handler will read the top-level register. For each bit that is set, the top-level handler will call the corresponding interrupt handler for that block. That block's interrupt handler will then read the block's interrupt register and respond accordingly.

The top-level interrupt module would not necessarily be a regular interrupt module in that the individual bits would not be an interrupt that needs to be acked, but rather a status that reports when a particular block has an interrupt pending. Figure 2 illustrates the difference. When the block handler acks all the interrupts at the block level, the block's interrupt line is de-asserted and the corresponding bit in the top-level is back to 0. If none of the blocks have interrupts pending, then the ASIC's own interrupt line is 0.

Even though each block has its own interrupt enables that defaults to disabled, the top-level block should still have a top-level interrupt enable that defaults to enabled. Typically, the top-level enable will remain enabled and the driver controls the interrupt enabling at the block level. We had a case where one interrupt in the block level did not have a block-level disable. That ASIC did not have a disable at the top-level either. So the only way to disable that interrupt was by disabling the interrupts for the whole ASIC. Having a top-level enable control provides a safety net for defects on the block level.

The exact details discussed here are not the only way interrupts can be architected. But many of the pitfalls and concepts mentioned can be used to help architect interrupts in an ASIC.

6.2 Interrupt Enable vs. Interrupt Mask

The name of the function that controls whether or not an interrupt is allowed to propagate has been called "Enable" in some ASICs and "Mask" in others. The term, "Mask," is unclear to the reader. Does a 1 mask off an interrupt, not allowing it to propagate? Or does a 1 provide a mask which is OR'd with interrupts, allowing it to propagate? We have seen both cases implemented in ASICs, which means the firmware writer has to read the document very carefully to determine whether a 1 or a 0 is needed. A better term to use is "Enable." To "Enable an interrupt" is clear in the reader's mind that a 1 in that position will allow the interrupt to propagate. So it sould be called the Interrupt Enable register.

6.3 How to Acknowledge an Interrupt

The Interrupt Status register reports which interrupts has occurred with a 1 in the appropriate bit position. The firmware acks the interrupt by causing the bit to go back to 0. How the firmware is supposed to do it has varied on different ASICs, some by writing a 1, others by writing a 0. We had a case where some code was leveraged from one ASIC to another which was different. The code inadvertently acked a different interrupt if it happened to occur at the same time. It was rare that both interrupts occurred at the same time but when it did, it resulted in a system hang waiting for the already-acked interrupt to be serviced. This was very difficult to diagnose.

From a firmware perspective, writing a 1 is better. What a firmware interrupt handler typically does is read the Interrupt Status register to get a list of pending interrupts, then it writes that very same value back out to ack the interrupts. If a zero is required to ack an interrupt, firmware has to invert (1's compliment) the whole word to make it all 1's except the bit to be acked. Also, firmware will use masks (such as 0x200) to specify a bit position. Firmware would have to remember to invert it as well, if used to ack interrupts. Be sure to clearly specify in the document how to ack an interrupt so the firmware writer can know without guessing.

We had a case where on the same ASIC, some registers containing interrupts had to be acked with a 1 and others had to be acked with a 0. An engineer tried to explain to me that it was a difference of "interrupt" versus "status," where you write a 1 to clear an "interrupt" but write a 0 to clear a "status." I said that it did not make a difference to me why they were different but that in my interrupt handler, I had to remember to write a 0 in one case (remembering to invert the other bits) and write a 1 in another before I could exit the handler.

6.4 Alignment of Status and Interrupt Bits

For some interrupt conditions, there is also a status register that shows the current state of the condition. The Status register shows the current condition, the Interrupt Status register shows whether or not a change has occurred, and the Interrupt Enable register controls if the interrupt will be propagated upward.

For each condition, that bit position should be the same in all three registers. This allows firmware to use the same bit mask to access that position in all three registers. For some conditions, an interrupt is not needed when the condition changes. In that case, those bit positions in the Interrupt Status and Interrupt Enable registers would be left unused. In other cases, an interrupt position is needed for which it does not make sense to report on the condition,

such as an error occurred. In that case, those bit positions would be left unused in the Status register. The following illustrates this concept.

Position	*31*	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Status	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*F*	*E*	*D*	-	-	*A*
Interrupt	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*F*	-	*D*	*C*	*B*	*A*
Enable	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*F*	-	*D*	*C*	*B*	*A*

Bits A, D, and F appear as status in the Status register as well as interrupts in the Interrupt Status and Interrupt Enable registers. Bits B and C are interrupts with no corresponding bit in the Status register. And bit E is in the Status register only and is not an interrupt. This format allows firmware to make the following constants that will apply to all three registers.

```
#define BIT_A  0x01
#define BIT_B  0x02
#define BIT_C  0x04
#define BIT_D  0x08
#define BIT_E  0x10
#define BIT_F  0x20
```

6.5 Delay of Acknowledgment

Under certain conditions, the interrupt handler can ack the interrupt, re-enable interrupts, and get out, only to be re-invoked. The reason is that the acking requires writing a value out to the ASIC, which requires getting the bus arbitrated, then getting the data across, which takes several CPU cycles. When the driver does the ack, the CPU puts it in a buffer to be written out and then moves on to the next instruction before the write has completed. The next instruction enables interrupts and the interrupt from the ASIC is still pending because the write has not gotten there yet. So the CPU re-invokes the handler. The handler reads the top-level status register to figure out which block generated the interrupt and gets confused because none of them are reporting it. When the read of the status register is launched, the CPU gets blocked waiting for the read to come back. It is behind the write which is doing the acking. The write (acking) completes before the read is allowed to happen. To avoid this problem, whenever firmware writes to ack an interrupt, it follows it with a read, which will block the CPU until the write completes. Then when interrupts are re-enabled, the ASIC's interrupt will be gone.

6.6 Level vs. Edge Triggered

Another thing to put in the document is to specify whether an interrupt is level or edge triggered. This helps the firmware writer to know if a condition needs to be cleared first or not before re-enabling interrupts.

7 Aborts

Aborting is a very complex task. It requires going from any condition back to default, going from any state back to home, and resetting all counters, registers, and buffers back to default or empty. It also requires getting everything ready for the next task. However, sometimes the abort process is not thought through very well because it is not "normal" operation. But it is the exceptions and error conditions that require the most thought.

In one case, a question was asked of an engineer about how to abort a transaction. He responded with, "Why would you want to abort it?" It was clear that the abort case had not been designed in. Aborts are needed for several reasons. Here are a few.

- Something else in the system had an error so this block will not be getting any more data or will not have anything consume any more of its data.
- There was not enough or too much data or memory to work on.
- Something was wrong with the data.
- A timeout occurred, something was taking too long.
- The user decided to abort for whatever reason.

In one particular case in one of our products, an abort is needed 60% of the time. Aborts need to be considered part of "normal" operation.

7.1 Proper Abort Behavior

Since firmware cannot know at the exact instance whether a task in the block is active or not, the block needs to respond to an abort from firmware under every condition, even idle. The response to an abort request should be identical under every condition. If the abort is not always immediate, it should have one bit show that it is pending and an interrupt to show when it is done. Even in the idle state, an interrupt should occur. If the abort process takes time, document best and worst time it will take to process it. This will help the firmware writer know if a few simple polling reads is sufficient to wait for completion or if it should go off and do something else until the interrupt occurs.

Always provide an abort function for any action that takes time, including things such as DMA transfers. Look at any state machines very carefully to make sure that the abort request will always be seen, at least within a few clock cycles. We had a case where there were a few states in the state machine that looped waiting on some event without looking at the abort bit. It would get stuck in that state because the event would never occur which was why an abort was in process. Fortunately, because of some test and debugging functions, we could artificially generate those events so that the state machine could get to a state where it would see the abort bit.

Ideally, there would only be one abort bit and one abort done interrupt in a block. But some blocks have more than one sub-block, each with their own abort method, such as a DMA and a compressor. Sometimes a sequence is needed to abort, requiring firmware to abort one before the other. A good way to handle it would be to have a bit for each one of those in same register. This allows firmware to generate aborts in any order, sequence, and even together. Be sure to document the steps necessary to abort the block and get it ready to resume.

We had one case where a block was so complicated in its abort process that it took 200 lines of code to accomplish it. Not only that, it took a 10-page Word document complete with three diagrams and a table to clearly explain those 200 lines of code. An abort should only take two lines of code, write the abort bit and wait for the abort interrupt.

8 Errors

Errors! Errors cause the most problems. Someone said that only 20% of our code is used to handle the normal operation. The other 80% is for error handling. Errors do occur and they must be taken into consideration.

The ASIC must deal with two types of errors, ones that it detects in its operation and the other when firmware is doing something wrong. In the latter case, firmware can be corrected to prevent it from occurring again. In the former, firmware must be able to handle all the errors detected by the ASIC.

A good way to handle errors is to provide too much information. With too much information, there is usually some clue that would indicate why the error occurred so that the engineers can try to fix it and prevent it from occurring again.

We had a case where all that was known was that there was a decompression error. But why? Was it a bad setup by firmware? An unexpected size? Not enough data? Corrupted data? Or what? Some of these possible reasons are firmware errors, others are operation errors. If the engineers could know the answer to why, then appropriate action can take place. By providing as much information that is known by the block to the firmware, better action can take place.

In the documentation for each error listed, explain the following:

- Under what specific conditions would the block generate that error?
- What is the current state of the block? Did it reset itself? Did it stay put in the bad state?
- What should firmware do to reset and continue? Ignore it? Log it? Abort the block? Or power cycle the device?
- Did something happen or not?

It would be better if the block left itself as is when the error is detected. This allows firmware to query the addresses, counters, status registers, and other registers to try to determine why an error occurred. Then the firmware can abort to get it back to normal to resume.

In one case, we have a state machine that detects various error conditions along the way. For any error detected, it goes to the error state to generate an error interrupt, then back to idle. We added some status bits that indicate which state detected the error. That is very useful in determining why the error occurred.

In the documentation, explain what would happen if the firmware tried to do something wrong. Would it be ignored? Would it potentially clobber something? Would an error bit be set?

9 Debugging Hooks

Firmware has the advantage in that the time between writing a line of code, compiling it, and running it is very short. If a bug was found, it is relatively painless to make a change in the code, recompile it, and try that to see if it works. If there is a bug that is difficult to find, it is easy to throw in some debugging print statements until the root cause problem is found.

ASIC designing does not have that luxury because it takes months to make silicon. (Software simulations and FPGA's help but they do have their limitations.) ASIC designers only get very few chances to get it right, sometimes as few as one. Once the silicon gets back, hopefully it works. However, there will always be some problems. In order to debug those problems, hooks are needed to aid in finding the root cause. Ideally, there are hooks in the block that would aid in debugging when problems occur.

Bugs can be in both the ASIC and in the firmware. However, when a problem first appears, it is not always known if it is in the ASIC or firmware. Test and debugging hooks in the ASIC help to make that determination.

Since the intention is that they not be used at all, the level of detailed support or documentation is not as demanding. The documentation should at least call out these test and debug registers but with brief explanations, so that the firmware engineer can look through there for something that might help a problem. If a problem arises, then the ASIC engineer can be consulted for more details.

These testing and debugging registers should be separate from the normal-use registers.

The following are some ideas of test and debugging hooks that could be implemented.

- The current value of address registers and counters, such as in a DMA block. This would help give an indication where a data corruption might exist in memory.
- The ability to read the current state of a state machine. Normally, the current state is changing rapidly during operation. But if the state machine is hung somewhere, reading this will indicate where, giving the engineers more clues to understand what went wrong.
- The ability to simulate that an external or internal event occurred, causing the block to progress in its task. As mentioned above in the Abort section, this ability was used to get the state machine to move through its states so that the abort can be processed.
- Enable or disable parts of the block. We had a case where we discovered that one part of the block interfered with the operation of another part. By shutting off that block, causing things to be bypassed, the other part was able to function properly. In fact, enabling and disabling became part of the normal operation because we needed both parts at one time or another.
- Another set of interrupts could be provided that will generate interrupts (when enabled) based on changes of various internal signals.
- Registers that show the current level of various internal signals. Knowing those levels allows both the ASIC and firmware engineers to debug why the block is having problems. And it may become normal operation if it contains needed information.
- Pull data out of or insert into the middle of a pipeline. We had a case where we used this testing feature to empty out some buffers that had been inadvertently left unconnected to the abort signal.

There is no precise list of what should be included as hooks. But experience with problems in the past will help identify possible areas where hooks would be useful.

However, testing and debugging hooks should not be used as normal operation in future generations of ASICs. The defect should be fixed or the function should be moved into normal register space with normal support and documentation.

As illustrated, testing and debugging hooks proved to be very valuable in getting the block to function. We have had cases where the block would have been rendered useless had it not been for these hooks. We would have had to roll the ASIC to get those fixed. Instead, we were able to come up with firmware workarounds to get the block to be functional. However, those workarounds are not always reliable. So getting rid of them in future ASICs should be a goal.

10 Other Items

The following are other items to note.

10.1 Test Cases Should Model Actual Firmware Usage

Typically a block can be used in many different ways. Find out how firmware intends to use it and write the block tests accordingly. We had a defect in a block that was due to the fact that the block test used only one chunk of memory when the firmware's fundamental design was to use several chunks of memory chained together. The chaining did not work and it was not caught in the block's simulation testing.

10.2 Leverage Functions

Be consistent throughout the whole ASIC. Leverage sub-blocks. The same DMA controller can be used by all blocks needing to transfer data. The same interrupt module can be used by all blocks.

10.3 Make the Block a Superset

Make the block be the superset of all the functionality that it might need in various ASICs used on various product lines.

On some ASICs, some of it might not be used so tie incoming signals high or low and do not connect outgoing signals to anything. Some signals lines are tied to ASIC pins. But for some versions, not all signals lines would be connected to ASIC pins so that the package could be smaller (and less expensive.)

Leave all the internal registers the same and do not change the bit mapping. This allows the same firmware driver to be used on several ASICs. It just so happens that certain interrupts might never occur because certain pins are not connected. Or there might be certain status bits that will never change. Some actions invoked by firmware might be ignored. But that is okay.

We had a case where the behavior of an external device had changed. Had the block in the ASIC been a superset of all known functionality, we could have turned on a part of that block that was not originally to be used and have solved the problem in a matter of an hour. Instead, it took several weeks with several redesigns to get a firmware workaround working.

We had another case where an ASIC was designed with limited features to go into a limited product line. However, later on after the ASIC was done, the question was asked if it could be put into a product line with expanded features. So we had to do lots of investigation to see what it would take to get that accomplished. Had the superset been included, the answer would have been arrived at much sooner.

10.4 Flexible Control

Where it makes sense, do not hardcode modes or values. Use default values but allow firmware to override them if so desired. For example, suppose some specification requires that a timeout be 10ms. Make a register for the timeout and have it default to 10ms. But if the firmware wants to override it, to 5ms or 20ms for example, then it could do so.

When adding a new function to replace an old one, leave in the old one (if there is room for it) in case the new one is broken. Firmware could then set a bit to switch out the new for the old and the ASIC would still be functional.

10.5 Rare Problems

Always look for problems that might occur, no matter how rare they may seem. A classic case is when two asynchronous events happen to occur at the same time, causing a collision. Just because such an event is improbable does not make it impossible. In fact, we find many of those types of problems while doing extended test runs near the end of the development cycle. Typically those collisions are very difficult to isolate and understand. So we have to run many devices for many hours to get them to occur.

One example previously mentioned above where firmware had to write a 1, another 1, then a 1 again, and finally a 0, to guarantee that the block will always see it. In 99% of the time, writing a 1 once was sufficient. But there was an external event that, when it occurred, distracted the block briefly to process it, before it could go back to see if that bit was set. The ASIC engineer had to analyze various timing parameters before a determination could be made that writing a 1 three times was sufficient 100% of the time.

Strive towards making something work 100% of the time, not 99%. That 1% will always occur. Looking for and eliminating or at least documenting such conditions will help save many hours of debugging time.

11 Conclusion

Many points were given that can be used when designing ASICs. Some of them will apply and others will not in the various ASIC designs. Some of these will generate ideas for other improvements. But it is hoped that some of these concepts will be worked into designs. More importantly, it is hoped that a dialog will be established with the firmware engineers that will eventually be using the block.