Building safe SoCs

Functional safety from an SoC designer’s perspective and the importance of finding faults at the earliest opportunity.

By Neil Howard.

Watchdog
A watchdog timer that merely checks if the watchdog has been patted ‘not later than time tmax’ is inadequate.
Better practice is to:
• Break firmware down into tasks, each with a particular ID and minimum and maximum execution durations (tmin and tmax). Before starting a task, the scheduler programs the watchdog with this information.
• On completion, the task ‘pats’ the watchdog with the required ID.
• A watchdog reset occurs if the watchdog is patted before tmin as well as not before tmax, or if the data does not match the required ID.
Ideally, the watchdog should use an independent oscillator and local power supply.

Data transfers
DMA transfers upset normal processor operation just as interrupts do and hence should be avoided where possible.
In ‘large’ SoCs, DMA must transfer chunks of data in order to maximise data throughput by using burst accesses. However, these bursts intrude on processor execution. In ‘small’ SoCs, data throughput is less significant than getting data into/out of peripherals quickly enough to minimise the amount of buffering within each peripheral.
Techniques to assist functional safety include:
• Firmware sets up a single multi channel controller with DMA transfers cycling around all DMA channels for all peripherals, including diagnostic memory to memory transfers.

Functional safety is becoming increasingly important in industrial and automotive applications, as is adherence to the relevant standards (IEC61508 and ISO26262 respectively).
Functional safety is about building systems that are dependable; it is important that any faults are detected at the earliest opportunity so problems can be fixed, rather than the fault only becoming apparent when the system fails.
Mixed signal SoCs for industrial and automotive applications typically comprise sensors and/or actuators, with wired or wireless communications to a host. The processor within the SoC must continually perform diagnostic checks on the system to ensure that it is fault free so the results provided are valid and can be relied upon.
Techniques described elsewhere for improving functional safety in SoCs typically focus on the processor. This article focuses on the practical issues of building a functional safety SoC from its various components.

Firmware
The presence of firmware complicates functional safety significantly. In abstract terms, the problem with a processor executing firmware is that the state space rapidly becomes too large, making it impossible to verify every possible state. We want to minimise the number of states not covered in simulation.
In particular, interrupts could arrive on any processor clock cycle, exploding the state space and should therefore be avoided. Instead, for example:
• Firmware should be split into tasks of fairly well determined execution duration
• Wait, polling for the next 1ms boundary after the end of each task.
• Poll status to determine the next task to perform.
• Separate out control from data paths wherever possible.
Consequently, there are only two bus masters in the SoC:

- Accesses are arbitrated at the slaves, rather than at the masters.
- Limit the DMA controller’s access to the ‘DMA-able’ peripherals and memory – ideally, just some banks within memory.
- Deterministic memory access; DMA transfers are completely transparent to the CPU. For example: DMA accesses take precedence over CPU and are never back-to-back; DMA accesses are always zero wait-state; and CPU accesses to ‘DMA-able’ SRAM may be possible in one cycle, but are forced to always be with one wait state.

**Unwrapping IP**

An existing serial communications controller IP block will typically contain a transmitter, a receiver, FIFOs and control/status registers and a processor bus interface. Usually, this IP will either contain features that are not needed or it may lack features, such as a receiver for diagnostic checking.

We want to be able to peel the outer layer of the IP away to reveal the basic IP components, which is where the IP’s real value is. For example:

- Instantiate the basic IP components directly; for example, one transmitter and two receivers;
- Aggregate control and status registers into a single bank of SoC level registers;
- Tie unused control signals into the basic IP components off rather than using a register;
- Separate the control from data by only using the standard bus interface for data.

**Memory protection**

Even if the processor in the SoC has a memory protection unit, it is desirable to also apply memory protection mechanisms to the DMA controller and control registers.

SRAM interfaces can include memory protection to block DMA access to certain regions, but this assumes the bus to which it is connected identifies the master (CPU or DMA) from which bus transactions originate. Instead, programmable ‘watermark addresses’ can determine the region(s) of memory which the DMA bus can write to and read from, before arbitration with the CPU bus.

Control registers of all peripherals should be aggregated into a SoC-level control block which can:

- Permit specific registers to be locked down. Writing to locked registers causes exceptions, so this lock-down should be through a lock sequence rather than single data value.
- Permit register addresses to be allocated so there is a Hamming distance of at least two between each control register.
- Allow ECC to be applied to these control registers.

And, obviously, ensure that exceptions are generated for all invalid addresses.

All internal volatile and non volatile memory is normally protected by Error Correction Codes. Rather than having one per memory, it may be possible to move ECC closer to the bus masters and to use the ECC encode/decode logic for the SRAMs for the centralised control registers as well. In any case, there is the need to be able to bypass any ECC in order to write invalid codes so that diagnostics can check ECC’s correct operation.

**Implementation**

Reusing a proven implementation flow is also very important for functional safety.

A standard design flow that is fully automated – including through layout (place and route) – strongly encourages fixing issues properly (at source RTL) and helps to reduce the number of tool warnings to a sensible number that can be reviewed properly.

Unfortunately, third-party IP generally creates difficulties here. Adherence to suitably strict coding and implementation rules is still the exception, rather than the rule, with IP whose authors generally rely too much on ‘downstream’ synthesis optimisation. Unfortunately, there is still too much reliance on faith with IP and warnings at the interfaces of the IP are particularly problematic.

Generally speaking, third-party IP should be avoided if possible.

**Conclusion**

It is not usually possible to design the entire system from scratch and it is equally impossible to design the system first and then retrofit functional safety features.

However, a top down approach in which functional safety is taken into account when selecting and integrating IP and where there is careful planning of data transfers around the SoC and hardware allows firmware to be written in such a way that reduces risk and ensures the design of the SoC fulfils all functional safety requirements.

**Author profile:**

Neil Howard is digital design manager for Swindon Silicon Systems.