Splitting Domains and the Construction of Bridges

David Whipp

GEC Plessey Semiconductors

email: David.Whipp@gpsemi.com

Several years ago we started our first Shlaer-Mellor project. Its purpose was to model the functionality of complex integrated circuits. To work within tool limitations, and after taking advice, the functionality was modelled in a single domain. This domain was partitioned into subsystems based on identifiable functional blocks. As time progressed, the domain got bigger as more subsystems were added. As the domain complexity increased, it became clear that the partitioning should be carried out at the domain level, not subsystem. This paper describes how this big domain is split into many smaller domains.

Introduction

Microcontrollers are integrated circuits that contain a processor core and a selection of peripheral components, such as serial/parallel port controllers and memory interfaces. To provide a customisable product we have developed processes that allow us to easily build chips with different configurations of these peripheral cells.

To help our customers to embed these customised chips into their products, we have developed a capability that allows us to provide models of microcontrollers. These models can be used for software and systems development before actual silicon is available; and at simulation speeds that are unattainable through traditional hardware simulation techniques.

After evaluating various methods, Shlaer-Mellor was chosen for developing the models. When the project was started, it was important to work within the limitations of our CASE tools. One consequence of this was that it was found necessary to model the entire microcontroller capability as a single domain.

The model emulates behaviour using cycle accurate timing. The external behaviour of the model must occur during the correct system clock period. This allows the system to be

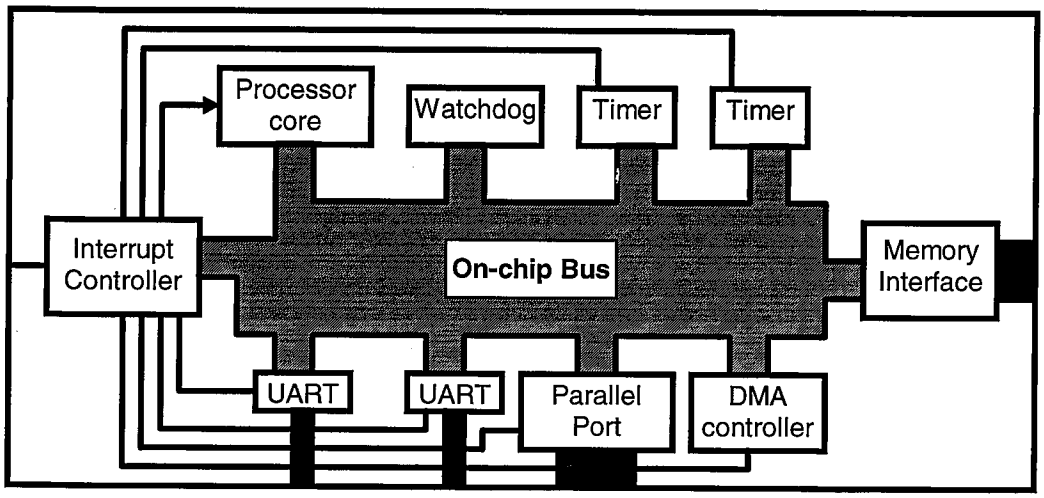

Figure 1 - A simple microcontroller

developed with realistic performance relative to other parts of the simulated system. However, it does impose restrictions on model.

A Single Domain Abstraction

Modelling in a single domain requires a common semantic context, and thus an abstraction of the problem that is common across the chip - both the processor core and all its peripherals. Fortunately we already had a software model of the processor, as an implementation domain, so our chip-model domain need not include the processor model. Figure 1 shows a block diagram of a typical microcontroller.

So what is a reasonable abstraction for the chip? All the peripherals communicate with the rest of the system using one of two mechanisms: discrete signal wires and the on-chip bus. It seemed reasonable to base the domain on these concepts.

Each component within the chip provides a set of registers that can be accessed using the on-chip bus. Registers may be written to and read from; and may have interesting behaviour when these actions occur. Thus the register was seen as an object, with subtypes that describe the specific behaviour within the component. Read and Write events are delivered polymorphically to the appropriate register subtype.

A similar argument was made for a "Port" object, which represents a component's communication over discrete signals. Some additional objects are required for components that can act as bus masters, allowing them to take control of the bus to access a slave component's registers. Figure 2 shows these objects on an outline information model.

Synchronisation of the model onto cycle accurate behaviour is achieved through handshaking events. Registers and Ports can be closely mapped onto blocks within the hardware, so the mappings to clock cycles can easily be identified.

Problems of the Single Domain

There are many problems with this simple model. Some are practical, whilst others are more philosophical — though possibly more important.

The most obvious practical problem is the size of the information model. It is dominated by the register and port subtype trees. Each component has many subtypes in each of these trees and, additionally, may be represented in the bus master subtype tree. It soon proves impossible to get a usable graphical layout of this information model.

A simple solution to this problem is to partition the model into multiple subsystems. All the objects associated with a given component are placed within a subsystem. The small number of remaining objects are placed in a chip-core subsystem. Each subsystem is of a manageable size, and a sensible graphical layout is

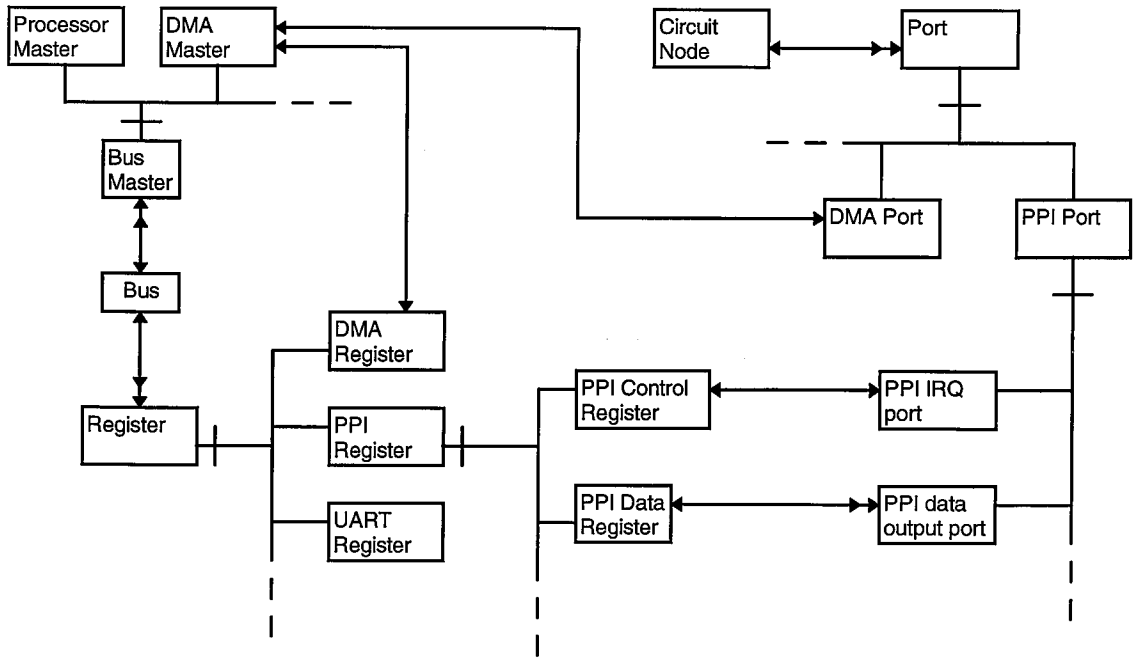

Figure 2 - Extract from the Chip-Model OIM

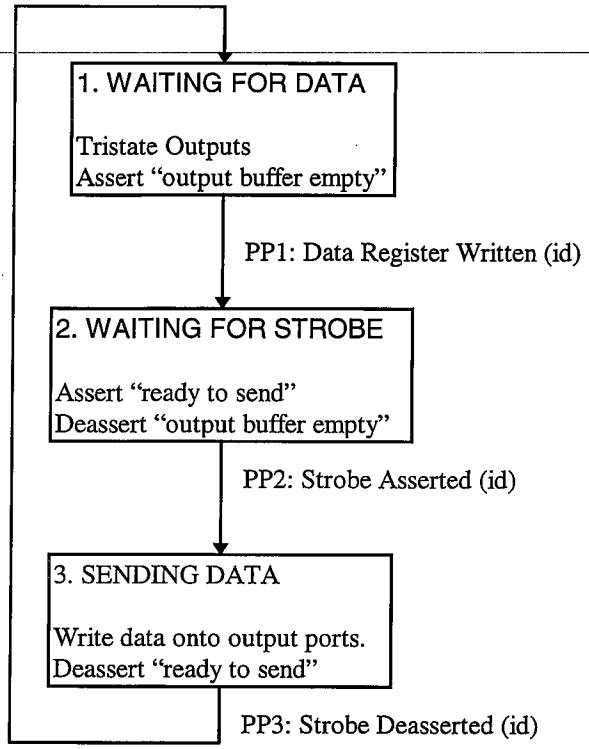

Figure 3 - Possible state model for a Parallel Port

achievable.

This allowed our model to continue to grow until we hit a second problem: configuration management. As time progressed, many new hardware components were produced. These were often modified versions of existing components. However, because our model represents many different chips (through population files), we had to maintain multiple versions of each component within the model. It because apparent that the multiple subsystem approach would soon become unworkable. To implement configuration management we had to move beyond the limitations of our CASE tool and use an external configuration management tool to manage multiple OOA databases. This move required each component to be modelled in different domain because our tool does not allow domains to be split across multiple databases.

A Naïve Multi-Domain Model

It is surprisingly easy to move from a multi-subsystem model to a multi-domain model. The domain interfaces can be extracted by examining the objects that are used to connect the subsystems. One subsystem "owns" the object, and the other one references it. The subsystem interfaces can therefore be described in terms of the processes that are used across subsystem boundaries. Accessors require a synchronous return whilst event generators are asynchronous.

The Real Problems

The configuration issues may force us to move to a multi-domain model, but the real reasons for the shift are more fundamental. The chosen abstraction uses registers and ports as the primary vehicle for describing the component's behaviour. This was necessary to avoid semantic shifts within the single domain. But this abstraction does not necessarily lead to understandable and maintainable models.

Consider the state model for strobing data out of a parallel port. Figure 3 shows a possible state model: data must be written to the data output register; it then becomes available to be strobed off the chip.

When the behaviour is packed into register and port objects, this state model is not explicitly given in any single object. The actions of states 1 and 3 are given within the strobe-port object; and the action of state 2 is provided in the data register object.

The impact of this dispersion of the state model is increased still further if a register contains many bitfields. It is common for many control signals to be packaged into a single register. A bitfield mapping describes which bits in the register represent each signal. This is a problem because it means that a single "register value" attribute is not atomic within the domain.

When many control signals are packaged into a single register, so many state machines may be multiplexed into the state actions. This demonstrates the weakness of using a non-atomic attribute in the register object. Maintenance of these distributed, multiplexed, state models becomes a nearly impossible task.

A Better Multi-Domain Model

The naïve multi-domain model maintained the same semantic context for all domains: they were all modelled in terms of registers and ports. This is fine when our aim is to migrate from one modelling style to another with the minimum of re-work. However, when modelling a new component, it becomes very tedious.

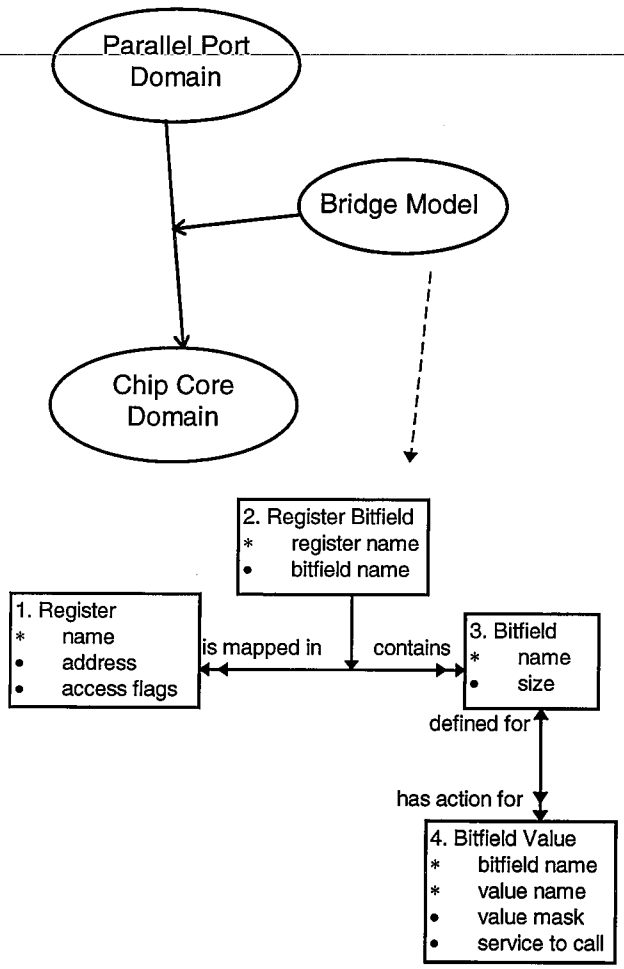

Figure 4 - Describing the bridge

A domain should be modelled using concepts appropriate to that domain. Its interfaces should be defined in terms that are meaningful to the domain. The functionality of a component does not depend in its implementation-interface, so its analysis shouldn't be dependent on that interface.

The functionality of the parallel port is described in the parallel port domain whilst its registers are defined in the chip core domain. Figure 4 shows the domain chart for this. The bridge between the two domains must define how a value written to a register is translated to a service invocation within the parallel port domain.

This mapping must break down a register into its constituent bitfields; and then determine which service to call depending on the value of each bitfield whose value has changed (many services may therefore be invoked).

I have found that this is most easily described by constructing an information model of the mapping. Note, however, that all the objects are specification-objects. Bridges have no state nor dynamic behaviour. It should also be realised that the complete bridge requires a more complex information model because registers are not the only concept involved; and because communication is two-way.

Evaluating the Improved Model

Moving the register mapping into the bridge allows the analyst to concentrate on the functionality of a component, rather than its implementation-interface. The models are therefore easier to understand and maintain.

However, the world is still not perfect. Objects within the peripheral component's domain have attributes. Many of these attributes are counterparts of bitfields within registers. Therefore we have a consistency problem. Whenever the domain changes the value of an attribute, this must be reflected in the corresponding bitfield, and hence in a register within the chip core domain.

In practice, this has resulted in cluttered state-actions. Every accessor must be paired a wormhole invocation allow propagation of the value change. This is tedious for the analyst, and error prone: it would not be immediately apparent if a wormhole was inadvertently omitted.

A similar situation arises with event generators. Tight synchronisation is required to maintain the cycle accurate behaviour of the model. A special synchronisation domain is used to keep track of a thread's progress by monitoring the number of times it splits and terminates. An asynchronous return is made to chip-core when the counter reaches zero. For this system to work, each state action must inform the synchronisation domain of the number of events that were generated. This has an added inconvenience of prohibiting the use of "ignored" events in the state transition table.

Architectural support

It is easy to see how these two problems could be solved with simple architectural enhancements. Attributes could be coloured to describe how their values map onto wormhole invocations; and communication with the synchronisation domain does not even require coloration.

Unfortunately, we have wanted to maintain the ability to simulate our models using a CASE tool

simulator. Once we start relying on non-standard architectural features, we lose this ability. Thus our architecture requires additional enhancements to ensure that an equivalent level of instrumentation can be achieved.

Thus by abandoning multi-domain simulation within our CASE tool we can provide a capability for clean modelling of the chip's components. But can we do better than architectural tweaks?

An Architectural Layer?

The chip-core domain defines mechanisms for storing data and for passing messages. By partitioning this domain from the rest of the model we have been forced to build bridges to it.

The purpose of an architecture is to define mechanisms for data storage and passing messages. Clearly the chip core domain meets this definition. An architecture is applied as a set of transforms over the constituent domains of a model. This leads to the conclusion that the correct way to extract the chip core domain from the original domain is through a process of factorisation, not partitioning.

The process of factorisation extracts information from one or more entities such that the extracted factors can be applied as a global transform over the residual entities to recreate the originals. The application of an architecture to a set of domains is an example of such a transform.

If we view the chip-core as an architecture then there is no need to explicitly communicate with it through wormholes. Architectures are applied as transforms of the whole model, not just specific processes. The mapping of attribute-instances to register-bitfields must still be explicitly provided as part of the model, but its application to the component domains is non-intrusive.

Unfortunately, adopting this strategy is not as simple as it might appear. Components that act as bus masters must explicitly communicate with registers via the system bus. Thus the architecture is not truly orthogonal to the component's functionality. Our architectural domain must therefore be linked to the chip model using both wormholes and transforms.

Summary

I have shown how we set out to develop a microcontroller modelling capability. Initially, to keep within the limitations of our CASE tool, it was necessary to aim for a single-domain model. This entailed finding a level of abstraction that could be applied globally across the chip.

As the model grew, it became necessary to partition the model into multiple subsystems. However, this was only a temporary solution. It was soon apparent that configuration control would soon become an issue if we continued to insist on a single domain model.

So the model was split into multiple domains. Each of the new domains had a single subsystem, which was copied directly from the original, big, domain. The only changes that were required were to replace references to shared objects with wormholes. These were easily identified as accessor and event-generator processes.

But separation into multiple domains provides a new landscape of opportunity. We are no longer constrained to model all components in the same semantic context.

Thus we are able to separate a component's functionality from its interface mechanisms. These interfaces are moved into a third domain, which appears to be associated with the bridge between the component's domain and the chip-core domain.

This separation results in a model that has one fact in two places: the values of registers are stored in both the chip-core domain and the component domains. We are currently maintaining the mapping manually but, if we are willing to drop the ability to simulate the model, then we can hide the problem in the architecture. Indeed, we may be able to view the chip core domain as an architectural layer in its own right - a domain that has been factored out of the earlier component domains.