Behavioural Modelling of Asynchronous Systems for Power and Performance Analysis

Philip Endecott, Stephen Furber,

Department of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL U.K.

pbe@cs.man.ac.uk, sbf@cs.man.ac.uk

Abstract

Conventional hardware description languages do not provide all the facilities required for efficient behavioural modelling of asynchronous systems. This paper presents a new HDL incorporating CSP-like channel communication and other features making it more suitable for this task. A model of a simple microprocessor is used to illustrate how the language and tools can be applied to real design problems. We use the tool to investigate the complex relationships that can exist between the speed of individual blocks and the system's overall performance, and look at power modelling based on channel activity.

1. Introduction

Despite the widespread success of globally clocked synchronous design it seems likely that in the future asynchronous design styles will be more widely adopted. Reasons for this shift include:

- As chips become larger, wire delays grow and operating frequencies rise, it will not be possible for a signal to go from one part of a chip to another in a single clock period. The SIA projects that by 2012, chips will have up to 10 000 distinct "time zones", between which some sort of asynchronous signalling will be required [3].
- Increasing clock frequencies will make electromagnetic compatibility a more serious problem as radiated energy increases with the square of the frequency. In globally clocked systems many signal transitions are highly correlated, further increasing the radiated energy. In contrast asynchronous systems have less correlated switching activity and so generate less interference.
- In a globally clocked system, all logic is activated on all cycles whether there is useful work to do or not. In contrast an asynchronous block is activated only when data arrives. This can make asynchronous systems more power efficient than synchronous ones.

The AMULET group at the University of Manchester has developed significant experience in the field of asynchronous design. In particular, we have made two asynchronous microprocessors implementing the ARM instruction set. AMULET1 [4] was the first asynchronous implementation of a commercial instruction set and illustrated that asynchronous logic could be applied to a problem of this size. In the AMULET2 design [1] we added a cache and a flexible external memory interface. The electromagnetic compatibility of this chip has been measured and compared with an equivalent synchronous design, and was found to be significantly better. AMULET2 has been evaluated in real-world applications where its electromagnetic compatibility, power efficiency and ease of use have been useful.

At present we are working on our third processor, AMULET3. This will comprise a complete embedded system including the processor core, RAM, ROM, a DMA controller, an asynchronous on-chip bus, external memory interface, and a bridge to a synchronous peripheral subsystem. The system has been designed with a telecommunications application in mind and we hope that it will reach commercial exploitation.

For designs of this complexity suitable support from CAD tools is essential. Although conventional tools are useful for the back end of the design flow (i.e. from schematic entry onwards) we found that behavioural modelling of our systems using conventional languages was difficult. In particular we found that the semantics of VHDL and Verilog did not suit our asynchronous design style.

In view of this we decided to implement our own hardware description language, which is called LARD, Language for Asynchronous Research and Development. The new language was first used for the behavioural modelling of AMULET3, where it has proved highly successful. For example we have found that LARD gives an order of magnitude improvement in designer productivity compared to our previous modelling language.

This paper starts by outlining the features of the LARD language and the facilities provided by its toolkit. We then go on to show how LARD can be used during the development of a microprocessor, focusing on areas where the asynchronous behaviour of the design is important. These concepts are illustrated using a simple microprocessor model.

2. The LARD language and toolkit

The key difference between LARD and conventional hardware description languages is its provision of CSP style channels for inter-block communication. In an asynchronous system timing information for each communication is provided by local request and acknowledge signals. When a conventional hardware description language such as VHDL is used to model the communication these timing signals have to be described explicitly. In contrast LARD allows us to model an abstract atomic communication without giving details of the protocol used; we have adopted the CSP notation of **!** for sending and **?** for receiving. For example, the following listings show VHDL and LARD code to communicate a value between two blocks:

	VHDL	LARD

Sender process:	`data <= x;`
	`req <= 1;`
	`wait until ack=1;`
	`req <= 0;`
	`wait until ack=0;`
		`C!x`

Receiver process:	`wait until req=1;`
	`y:=data;`
	`ack <= 1;`
	`wait until req=0;`
	`ack <= 0;`
		`C?(`
		`  y:=?C`
		`)`

LARD also provides fine grained concurrency; statements can either be composed sequentially using **;** or concurrently using **|**. There is a VHDL-like model of time with explicit delays introduced by **`wait_for`** statements. The other features of the language are common to other languages; LARD provides the expected facilities for structured programming and has a comprehensive type system.

LARD source code is compiled into a low level bytecode which is then interpreted. The interpreter has been embedded in an execution environment written using the tcl language and the tk graphical user interface package [2]. The facilities provided by the execution environment include:

- Source level debugging with single stepping and breakpoints.
- Traces of selected channel and signal activity.
- Animated displays based on model behaviour, with for example controls allowing delays within the model to be adjusted and the effects observed.
- Structured output of information messages from the model

Various ways in which these tools can be used are illustrated in the following sections.

3. The STUMP processor

To illustrate some of the ways in which LARD can be used in a microprocessor project, a simple processor called STUMP is presented here. The STUMP architecture was developed for undergraduate teaching in Manchester: students implement a synchronous version of the processor for a second year engineering lab exercise. It has been chosen for this paper because it has the right level of complexity: it can illustrate the important points without undue complexity.

STUMP has a RISC-like instruction set with fixed-length 16-bit instructions. Only explicit load and store instructions access memory; all other instructions operate register to register. The instruction set has many features similar to ARM, which was the origin of its name - it is a “cut down ARM”.

The implementation modelled here has a three stage pipeline, with fetch, decode and execute stages. The interconnection of these blocks and of the processor core to the memory is show in figure 1.

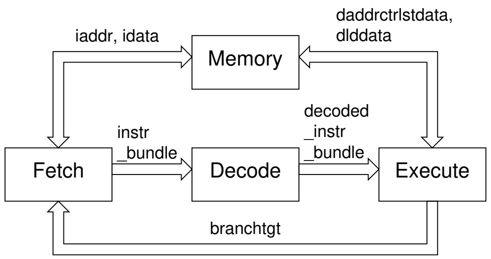

Figure 1: STUMP processor organisation

The LARD model of the processor core instances the three pipeline stages, interconnected by various channels:

```
"fetch"   is  fetch(iaddr,idata,branchtgt,instr_bundle) |
"decode"  is  decode(instr_bundle,decoded_instr_bundle) |
"execute" is  execute(decoded_instr_bundle,branchtgt,
              daddrctrlstdata,dlddata)
```

The fetch unit contains a loop which fetches instructions from memory, sends them to the decoder and increments the program counter. On each cycle around the loop it also checks whether a branch target has arrived from the execute unit, and if so it replaces the current program counter value.

```
forever(
  if (chan_ready(branchtgt)) then (
    branchtgt?(
      pc:=?branchtgt;
      fetch_colour:=!fetch_colour
    )
  );
  ( iaddr!pc | idata?(instr:=?idata) );
  instr_bundle!{instr,fetch_colour};
  pc:=pc+1
)
```

The decoder stage is modelled by a simple loop that receives an instruction from the fetch unit, decodes it, and sends the decoded instruction to the execute unit.

After a branch instruction has been executed, a number of instructions from the original instruction stream may still be present in the pipeline. To avoid erroneously executing these instructions the fetch unit tags instructions with a colour bit which is propagated down the pipeline. This colour bit is inverted when a branch target arrives at the fetch unit. The execute unit then discards instructions that don't have the expected colour. So the execute unit consists of a loop which receives a decoded instruction, checks its colour and if it is as expected executes the instruction.

```
forever(
  decoded_instr_bundle?(
    {dec_instr,colour}:=?decoded_instr_bundle
  );
  if (colour=expected_colour) then (
    case dec_instr->itype
      when ADD =>
        arith_op(dec_instr->dest,srcA,srcB,false)
      /* etc. */
))
```

The code shown here represents a highly behavioural style of description. As the design is refined, this behavioural code is replaced by a more structural description. We want to be able to find out as much as possible about the properties of the model, such as its correct functionality and performance, at as early a stage as possible, i.e. with this style of behavioural description.

Questions that might want to answer include:

- Is the model functionally correct? Does it get the right answer? Does it have any potential deadlocks?
- How fast is it? What limits its performance? How sensitive is the overall performance to the speed of individual blocks?
- What is the effect of changes to the architecture?
- Which blocks are the most important in terms of power dissipation?

The following sections illustrate how we can answer questions of this sort using the LARD toolkit for the STUMP processor.

4. Measuring effects on performance

When working with behavioural models early in the design cycle, measuring absolute performance is less important than establishing the underlying influences on the system's performance. For example, if the designer can identify which blocks have the greatest effect on overall performance, design effort can be focused in those areas.

In a synchronous design this is a relatively simple task. Each block is either on the system's critical path or it isn't. Increasing the performance of a block that isn't on the critical path will have no effect on overall performance. If a block is on the critical path, reducing its delay by δ will reduce the system cycle time by δ, until it isn't on the critical path any more.

For asynchronous systems we find that the relationship between block delays and system performance is much more complex. Dependencies between components mean that some delays can propagate to affect performance in other parts of the system so that their effects are magnified, whereas others are masked and have no overall effect. To further complicate things delays can have one effect at one time and another at another time, depending on the tasks being carried out.

Our approach is to measure the contribution of each delay to overall performance by simulation. We run a benchmark simulation with nominal delay values, and then repeat it with each delay adjusted by a small δ in turn. Table 1 shows results for the STUMP processor, measuring the change in overall execution time (Δ) relative to the change in block delay (δ) for each block. [1]

Block	Δ / δ
Memory	56
Decode	2
Execute (all instructions)	19
Execute (executed instructions)	19

Table 1: Influence of each block's delay on overall performance

From this analysis we can conclude that the speed of the memory makes the greatest contribution to overall performance, followed by the speed of the execute unit. An increase in speed of the decoder would make only a small increase in overall performance.

Thinking of the performance of the system as a function of many variables, each representing one of the delays, we can see that the numbers reported above are the gradients of the graph of the function in each of its dimensions at the point representing the nominal delay values. It can also be useful to look at the characteristics of this function further from the nominal delays.

Since we have established that the memory delay is the most significant let us look at the effect of varying the memory delay while keeping the other variables constant. Figure 2 shows the result of this experiment.

As we would expect, when the memory is very fast it has no effect on the overall performance. Some other block is limiting performance and small changes to the speed of the memory have no overall effect.

At a delay of around 40 the memory becomes the processor's bottleneck, and above that point there is a general upward trend in overall execution time with increas-

1. The delay in the execute block differs for instructions that are cancelled for having the wrong colour than for normal executed instructions.

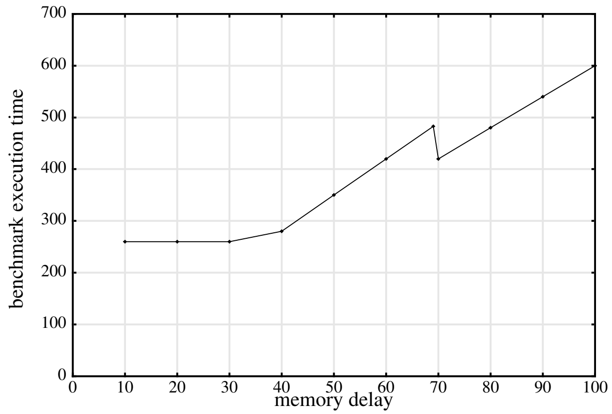

Figure 2: Influence of memory delay on overall performance

ing memory delay. There is however a surprising discontinuity in the graph at memory delay 70. This effect is typical of the sort of behaviour seen in complex asynchronous systems and illustrates why it is important to be able to carry out this sort of analysis. The explanation for this discontinuity is that above this point the pipeline contains one fewer instructions as preceding instructions have progressed further by the time the next is fetched. The consequence of this is that when a branch occurs fewer instructions have to be flushed from the pipeline, giving a shorter branch latency and hence higher performance.

5. Looking for Deadlocks

One bug that commonly occurs during asynchronous design is the introduction of deadlocks. One possible deadlock that can occur in the STUMP processor implementation is as follows: The memory is being accessed for an instruction fetch and will not be freed until the decoder is able to accept the new instruction. The decoder in turn will not become free until the execute unit is able to accept its next instruction. The execute unit is executing a load or store instruction which cannot complete until it is able to access the memory. The result is a deadlock.

In the future we hope that formal analysis will be able to detect deadlocks like this one, but at the present time the tools available to us a limited to small problems and cannot cope with data-dependent behaviour.

Often deadlocks will be detected during normal simulations. Deadlocks are surprisingly easy to understand when they occur; the simulation simply grinds to a halt, and the execution environment's source code viewer will show where each thread got stuck.

Other deadlocks can be harder to detect because they will only occur under a limited set of timing conditions. For example, the deadlock described above will only occur when the pipeline is running mostly full, which will happen when the execute stage is slower than the fetch and decode stages.

To detect timing dependent deadlocks we run a modified simulation with randomised delays throughout the model, and hope that a sufficiently long simulation run will eventually detect any problems. At present we have not developed a probabalistic model of our chances of detecting any faults, but in all the cases we have looked at the random delay testing has quickly identified the potential deadlock.

6. Measuring Power Efficiency

Our concern with power efficiency at the behavioural level is primarily to know which parts of the system will benefit most from efforts to improve power efficiency. To address that question we are able to monitor activity on the communication channels. The activity information includes the number of communications that have occurred and the number of bits that have toggled during all the communications. This information could be back-annotated with extracted signal capacitances to obtain a more accurate measure of power distribution, but we find that it is sufficient to look at the raw data for the initial behavioural model. This may show that some channels have orders of magnitude more activity than others, leading us to focus our low power design activities in their direction. Figure 3 shows how channel activity data is presented by the tool for the STUMP model.

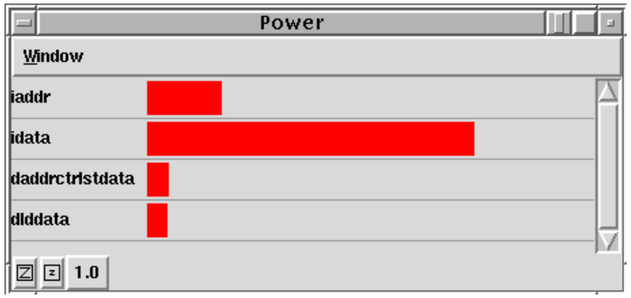

Figure 3: Channel activity for the STUMP model

7. Conclusions

By implementing a language whose semantics match our design style more closely than the alternatives we have seen a substantial improvement in designer productivity for behavioural modelling. The extra time created as a result could be used to explore the area of performance modelling where some systems have surprising behaviours.

LARD is freely available and runs on most Unix/X platforms. Source code and comprehensive documentation can be found on the web at **`http://www.cs.man.ac.uk/amulet/projects/lard`**. You can also find source code for the STUMP processor example at this address.

8. References

[1] S. B. Furber, J. D. Garside, S. Temple, J. Liu, "AMULET2e: An Asynchronous Embedded Controller", Proc. Async'97, Eindhoven, pages 290-299. http://www.cs.man.ac.uk/amulet/publications/papers/async97_A2e.html

[2] J. K. Ousterhout, "Tcl and the Tk Toolkit", Addison Wesley, 1994, ISBN 0-201-63337-X.

[3] "The National Technology Roadmap for Semiconductors", Semiconductor Industry Association, 4300 Stevens Creek Boulevard, Suite 271, San Jose, CA 95129, 1997.

[4] J. V. Woods, P. Day, S. B. Furber, J. D. Garside, N. C. Paver, S. Temple, "AMULET1: An Asynchronous ARM Microprocessor", IEEE Transactions on Computers, April 1997, Vol. 46, No. 4, pages 385-398.