ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**Mary Sheeran, Chalmers Technical University, Sweden, and Satnam Singh,
University of Glasgow, UK (Eds)**

Designing Correct Circuits

Proceedings of the 3rd Workshop on Designing Correct Circuits
(DCC96), Båstad, Sweden, 2-4 September 1996

ISBN: 3-540-76102-0

Paper:

Specifying and Property Checking the AMULET1 Address Interface

Y. Liu and G. Birtwistle

Published in collaboration with the
British Computer Society

Specifying and Property Checking the AMULET1 Address Interface

Ying Liu
Northern Telecomm, Ottawa, Canada

Graham Birtwistle
Computer Studies, Leeds, UK

Abstract

We describe work completed on the specification and verification of the address interface of AMULET1, an industrial strength asynchronous microprocessor designed, fabricated and tested by Manchester University. AMULET1 is an asynchronous version of ARM, the best selling RISC chip in the 1980's, and now a leading macro-cell. The address interface is a substantial floor plan element, shared by several processes, which acts as a filter into memory for pc-values, memory swap, and single and multiple load/store operations. The specification is at the register transfer level and is sufficiently detailed to detect the known deadlocks in earlier designs and verify that the final version is deadlock free. The specification work was carried out in Milner's Calculus of Communicating Systems (CCS) and property checked using the Concurrency Workbench (CWB). The work described is *post facto* — it was started after the AMULET1 chip had been sent for fabrication.

1 Introduction

AMULET1 [8, 10, 11, 12] is an asynchronous version of the ARM microprocessor [9, 25] developed by Acorn Computers Ltd., Cambridge, UK. With the full co-operation of the AMULET design team, the authors have developed and property checked specifications of AMULET1 at the instruction level and, for the four major floor elements, at the register transfer level (RTL). The instruction level specification shows how distinct instruction classes flow and handshake their way through the architecture as a whole and gives a good intuition for the roles of the floor plan elements and the interconnections amongst them. The individual floor plan element RTL specifications are sufficiently detailed to serve as implementation guides. [14] gives fairly complete specifications and property checks of both the instruction-level view of AMULET1 and of its floor plan elements.

In this paper, we present the RTL specification of the controller for the address interface, which accepts and ships out all memory access requests to the memory interface. The address interface hardware is shared by several processes. The major problem in its specification is to control their interactions whilst retaining mutual exclusion and liveness and ensuring that deadlock will not occur.

Major initial decisions lay in choosing a suitable and mechanised notation and deciding upon the level of abstraction at which to pitch the specification. Previous experience on specifying a small 4-phase micro [4, 5] showed that the CCS semantics match the asynchronous design style well and its object-oriented style of system description captures hardware design directly. In fact, using FPGA technology and the approach outlined in Brunvand's PhD thesis [7], Tom Borsodi (an EE graduate student at Calgary) quickly built a working chip from this RTL CCS description. Further, the CWB had already proved adequate for mid-sized specifications of [4, 5, 20]. Another lesson learned from this RTL work was to use a high level of abstraction on (regular) structures that are part and parcel of a designer's trade, for example, the alu, the memory interface, and fifos, thus leaving ourselves free to focus on the requests and acknowledgements amongst the remaining (irregular) register transfer level blocks.

Finding an appropriate level of abstraction was doubly important in this case, because at the time work started, the only useable account of AMULET1 was Paver's PhD thesis-to-be which described the operation of AMULET1 via its dataflows, but did not present its synchronisations. In order to understand the latter, we constructed models following our understanding and intuition, and argued their appropriateness with members of the AMULET1 team. Initially the discussion notation was that of figures 12–16 in this paper which had the advantage of being natural to engineers and directly translatable into CCS. As time went by, CCS itself became an acceptable notation for both sides.

The structure of the paper is as follows. We assume a basic understanding of the 2-phase asynchronous design style (see [24] or the first chapter in [2] by Davis and Nowick). In section 2 we introduce the CCS notation via examples and discuss its pros and cons. In section 3, we give an overview of the AMULET1 architecture and its major floor plan elements. We also present its various classes of instruction and how they flow through the design. Section 4 covers the role of the address interface and how it handles its various tasks (PC incrementation, single address transfer, multiple address transfer). Section 5 presents our RTL model of the address interface and its specification in CCS, and section 6 presents and explains some of the property checks that were made upon the initial and revised models.

2 Expressing asynchronous circuits in CCS

CCS builds from the agent (process or object) O which can do nothing. From there CCS permits three ways of building more interesting agents.

Prefixing: • The agent $a.\bar{b}.O$ can do an action a , then an action $\bar{b}$, and after that, nothing more. Recursive definitions are allowed. E.g. we may define a clock as: $Clock \stackrel{def}{=} \overline{tick.tock}.Clock$ This clock ticks and then tocks forever.

Figure 1: Toggle circuit

We now have enough notation to describe the toggle, T , pictured above: $T \stackrel{def}{=} a.\bar{b}.a.\bar{c}.T$. By convention, when describing asynchronous hardware, a transition on line q is written q if it is an input, and $\bar{q}$ if it is an output. After the first a input transition, the toggle outputs a transition on line b . After the second a input transition, the toggle outputs a transition on line c . This behaviour cycle then repeats. The circuit diagram for a toggle is usually decorated with a blob to denote which output fires first (here b).

Choice: +

Figure 2: C element

The C element accepts inputs on a and b in any order and then outputs on z . In CCS this is written

$$C \stackrel{def}{=} a.b.\bar{z}.C + b.a.\bar{z}.C \quad \text{or more compactly by} \quad C \stackrel{def}{=} (a.b + b.a).\bar{z}.C$$

If the C element receives a transition on a , it evolves into the agent $b.\bar{z}.C$. If the C element receives a transition on b , it evolves into the agent $a.\bar{z}.C$. If transitions on a and on b are both available, this C-element evolves non-deterministically and accepts both inputs in an arbitrary order.

This definition of a C-element is usually sufficient when describing circuits at the RTL. When describing low-level library components, it might be necessary to require the inputs to a C-element to cancel (the real circuit would permit this). Such behaviour is given by the “wobbly” C-element, WC :

$$\begin{aligned} WC &\stackrel{def}{=} a.(a.WC + b.\bar{z}.WC) + b.(a.\bar{z}.WC + b.WC) \\ WC' &\stackrel{def}{=} a.(a.\overline{error}.O + b.\bar{z}.WC') + b.(a.\bar{z}.WC' + b.\overline{error}.O) \end{aligned}$$

The variant WC' , which explicitly flags an $\overline{error}$ should one arise, and could be used in place of C . If so, the techniques of section 6 can be used to check whether an error could ever arise.

As a second example of choice, here is a 0 - n counter ($n \geq 1$) which can be incremented when not full and decremented when not empty. States 1 - n are flagged $\overline{nE}$ (not empty) and states 0 -($n-1$) are flagged $\overline{nF}$ (not full).

$$\begin{aligned} CNT &\stackrel{def}{=} CNT_0 \\ CNT_0 &\stackrel{def}{=} \overline{inc}.CNT_1 + \overline{nF}.CNT_0 \\ CNT_1 &\stackrel{def}{=} \overline{inc}.CNT_2 + \overline{dec}.CNT_0 + \overline{nE}.CNT_1 + \overline{nF}.CNT_1 \\ &\vdots \\ CNT_k &\stackrel{def}{=} \overline{inc}.CNT_{k+1} + \overline{dec}.CNT_{k-1} + \overline{nE}.CNT_k + \overline{nF}.CNT_k \\ &\vdots \\ CNT_{n-1} &\stackrel{def}{=} \overline{inc}.CNT_n + \overline{dec}.CNT_{n-2} + \overline{nE}.CNT_{n-1} + \overline{nF}.CNT_{n-1} \\ CNT_n &\stackrel{def}{=} \overline{dec}.CNT_{n-1} + \overline{nE}.CNT_n \end{aligned}$$

Parallel composition: |

Figure 3: 2-phase register

Suppose we wish to specify a pipeline register REG which co-operates with a sender downstream (left in figure 3) and a receiver upstream (right in figure 3). After its input data has been loaded on dIn , the register receives a signal on its request line rin . REG then latches this data, “shows” it strongly on $dOut$, and then signals both on $\overline{ain}$ (acknowledging to the sender that the data is secure) and on $\overline{rout}$ (the data in REG is now stable and can be used whenever the receiver thinks fit). The sender may now update the value on dIn , and send a fresh signal on rin , in which case REG must accept this signal, but may not read the data value until it is certain that the previous value has been safely used upstream, which is signalled on out .

$$\begin{aligned} S_0 &\stackrel{def}{=} rin.S_8 & S_{12} &\stackrel{def}{=} \overline{ain}.S_9 + out.S_{14} \\ S_8 &\stackrel{def}{=} \overline{ain}.S_{15} + \overline{rout}.S_{12} & S_9 &\stackrel{def}{=} rin.S_7 + out.S_0 \\ S_{15} &\stackrel{def}{=} rin.S_{16} + \overline{rout}.S_9 & S_7 &\stackrel{def}{=} out.S_8 \\ S_{16} &\stackrel{def}{=} \overline{rout}.S_7 & S_{14} &\stackrel{def}{=} \overline{ain}.S_0 \end{aligned}$$

Table 1: REG: state by state specification

Figure 4: REG: corresponding state diagram

The specification of REG is quite dense and is rendered non-trivial by this overlapping behaviour. Another way of specifying the behaviour of *REG* (which generalises nicely) is to consider it as the composition of a left interface (handling the downstream interaction) and a right interface (handling the upstream interaction) together with a local state machine which keeps track of whether the register is currently empty or full, for which *CNT* with $n=1$ will suffice.

On receiving a signal on *rin*, the left interface waits until *REG* has a free slot ($\overline{nF}$), increments the local counter (to state CNT_1 which causes $\overline{nE}$ to be raised), sends a signal $\overline{ain}$, and is then ready to do it again. Expressed in CCS this is: $L \stackrel{def}{=} rin.\overline{nF}.inc.\overline{ain}.L$.

The right interface waits until *REG* is not empty (nE), sends a prompt signal upstream on $\overline{rout}$, waits until the data value has been accepted upstream (*aout*), and then decrements the counter (to state CNT_0 which causes $\overline{nF}$ to be raised), and is then ready to do these actions again, i.e. $R \stackrel{def}{=} nE.\overline{rout}.aout.dec.R$.

REG is defined by composing its constituent objects which synchronize by communicating. A communication can occur when an action is an input to one agent and an output to another, and both are enabled (e.g. *L* offers *inc* and *CNT* (in state CNT_0) offers $\overline{inc}$). When a signal is hidden, the communication becomes local. Thus, *REG* may now be specified as:

$$REG \stackrel{def}{=} (L \mid CNT \mid R) \setminus \{inc, dec, nE, nF\}$$

Table 2: Compositional specification of REG

The above composition, but with *CNT* initialised to $n = k$, gives us the specification of a fifo of size k in a form considerably easier to understand and write than its monolithic equivalent using only prefixing and choice. The technique can be applied to many regular structures (stacks, queues, fifos).

CCS is a very compact system-description notation which permits hierarchic specifications. Being object-based, it maps well onto hardware structures. [13, 14, 20] give many examples of 4-phase and 2-phase circuit specifications. CCS is mechanised by the Concurrency Workbench [19] so that specifications can be entered and tested before acceptance. Amongst the pros of CCS are that the same notation and tools can be used to describe software, hardware, and mixed systems. It is rigorously defined with formal and clear semantics and equivalencing.

However CCS has its limitations, the most important of which are: (i) vanilla CCS does not handle data well¹ so that the choice of CCS meant that we had made a conscious decision not to model functional behaviour, but to con-

¹It can be done in its value-passing variant. Bruns [6] maps the value-passing calculus into vanilla CCS, but the resulting state explosion renders the technique impractical.

centrate upon sub-system decomposition and interaction. Our view is that when attempting the description of a large system (especially the first time), the most important issues are capturing the right model structure (decomposition and interaction) and the right level of abstraction. Further, the finer the grain of the model the larger the number of states in the resulting specification. It turned out that the coarse grained CCS model was already pushing the limits of the CWB, and it is unlikely that we would have been able to complete our work using anything with much finer grain. Although finer grain process algebras do exist, they are not as well mechanised as CCS on the CWB. (ii) when an internal communication is offered (signal sent on a line) in the CCS model, the recipient may (a) choose to ignore it for the time being, or (b) not be ready to accept it. (a) corresponds fairly well to an arbitrary delay on a line, but (b) is undesirable since it might result in a missed signal and result in hardware failure. Such checking is not built into our CCS model, but Stevens [20] explains how to check designs expressed in CCS for receptivity, and of course, the AMULET1 designers had made sure receptivity was not a problem in the real hardware.

3 AMULET1 architecture

Figure 5: AMULET1 internal organization

The ARM instruction set is based upon a load/store architecture and features 16 visible registers each of thirty-two bits. In common with other RISC processors, the instructions that perform data processing functions are separated from those that move data between the memory and the registers. Amongst the more unusual features of the ARM instruction set are: conditional execution of all instructions, pre- and post- indexing of load and store operations, general use of program counter (PC), and multiple register load/store operations. In practice, conditional execution reduces the number of branch operations from about 1 in 5 to about 1 in 7, pre- and post- indexing speed up array access and renders their coding easier, the PC can be assigned to in both load and data operations, and load/store multiple improves the programming efficiency of procedure entry and exit. More details of the ARM processor are given in [9, 25].

The major functional units of our AMULET1 model are: the register bank **RB**, the address interface **AI**, the memory interface **MI** (an abstraction of the memory management unit and the memory), the data interface **DI**, the decode unit **DEC**, and the execution unit **XC**. The bus interconnections amongst these units (excepting **MI**) are shown in figure 5. The main task of the **AI** is to generate sequential instruction addresses and forward them to the memory interface. The **AI** also ships addresses of load (right hand sides) and store operations (left hand sides only), and generates the addresses for load/store multiples. Each time the **AI** generates a next sequential instruction address, it inserts a copy into a local fifo (PCpipe), the head of which acts as R15, the PC register. Thus another task for the **AI** is to “update” PCpipe when a jump is taken.

The instruction set of AMULET1 is a direct mapping of the ARM instruction set [9, 25] onto the asynchronous organization of AMULET1. Instructions may be grouped into eight classes according to their datapath activities.

- **Multiply operation** ($w := a \times b$). The multiplier accepts two input operands from the **RB**. The partial sum and partial carry produced by the multiplier are added together in the ALU. The final result is written back to the **RB** via the W bus.
- **Data operation** ($w := a \text{ op } b$). A data operation takes two operands and performs one of the 16 arithmetic or logic operations supported by the ALU. The final result may be dispatched either to the register bank or, when w is R15 (the PC), to the **AI**.
- **Branch operation** ($R15 := R15 + k \{ R14 := R15 + 4 \}$). The simple branch operation is a PC relative jump. It adds the current PC value and the offset together in the ALU and sends the result back to the **AI** as the new next instruction address. Branch and link requires two phases. The first is the same as branch. In addition, it saves the address of the next instruction in the **RB** as the subroutine return address.
- **Load operation (pre-index: $w := MEM[a + b] \{ a := a + b \}$:: post-index: $w := MEM[a]; a := a + b$)**. A load operation is split into an address calculation phase and a data transfer phase which follows the calculation phase immediately down the execution pipeline. The first phase calculates the load data address in the execution unit, and then sends it to the memory and via the W bus and the address interface. The second phase transfers the data read from the memory via the **DI** and the W bus either to the **RB** or to the **DI**. The load operation may additionally allow the pre-indexing or post-indexing of a register. The calculated address may be written back to the **RB** in a pre-indexing load operation, and will always be written back to the **RB** in a post-indexing load operation. In the latter case, transfer of a to the **AI** during phase 1 takes place on Apipe since the arithmetic operation requires W.
- **Store operation (pre-index: $MEM[a + b] := w \{ a := a + b \}$:: post-index: $MEM[a] := w; a := a + b$)**. A store operation is split into an address calculation phase and a data dispatch phase. The store data address is calculated in the execution unit and sent to the memory via the W bus and the **AI**. The datum to be stored is dispatched to the memory via the Dpipe and the **DI**. The address and the data rendezvous at the memory to complete the store operation. The store operation also allows pre-indexing or post-indexing of a register with the same provisos.
- **Load/store multiple operation**. The load/store multiple operation supports the load or store operation of any subset of the 16 visible registers. It differs from individual load/store operations in that, once the base address is calculated, the remaining addresses are produced autonomously by the address interface. Since the registers are operated on from low index to high index, if the PC is loaded or stored, it is always taken last. The load/store multiple operation is by far the most complex instruction in ARM’s repertoire.
- **Swap operation** ($w := MEM[b], MEM[b] := a$). The swap operation swaps the contents of a register and a memory location. It carries out two memory accesses atomically without the possibility of being interrupted.

- **Software interrupt operation** ($R15 := k$, $R14 := R15 + 4$). The software interrupt operation causes the processor to break off from the current program by introducing an exception vector k to the memory via the execution unit, the W bus and the **AI**. The return address of the instruction after the interruption happens is saved in the **RB**. During the operation, the processor is switched to its supervisor mode. Thus, this operation is similar to the branch and link operation described above, but with mode switching.

4 Address Interface

Figure 6: Address interface

Figure 6 shows the bus connections of just the **AI**. Its internal organization is shown in Figure 7.

Figure 7: Internal organization of the address interface

Inputs to the address interface are regular addresses and PC values that come from the execution pipeline via either

the W bus or the Apipe. Regular addresses and PC values are destined for the Memory Interface (MI), but PC values sent to the PC register are also forwarded to the PCpipe (so as to be accessible later to the execution unit). The associated req/ack interface never causes system problems and so the PCpipe is omitted in the sequel.

The main data path modules inside the address interface include a PC register that stores the current PC value for fetching the corresponding instruction from the memory, an LSM register that stores the current address during a load/store multiple operation, and a memory address register MAR that contains the next address to be dispatched to the memory. The contents of the PC register and the LSM register are generated by incrementing the current MAR contents through the incrementer, INC. Mutually exclusive requests from the execution unit to the MAR are filtered by a regular multiplexor; competing requests (initialisation, instruction loop, and branch and load/store multiple requests) requiring access to the MAR are filtered by an arbitrating multiplexor. Exclusive access to the MAR is thus guaranteed at any time.

Accesses to the address interface use the data path modules inside the address interface in one of three ways:

- PC incrementing loop (PCL) for instruction fetch;
- Single address transfer (SAT) for transferring a regular address in LDR, STR, and SWP operations, or a new PC value in data, B and BL, LDR, and SWI operations;
- Multiple address transfer (MAT) for transferring multiple address in LSM operation.

These accesses are now detailed and numbered 1–6 for later reference.

PCL. The PC incrementing loop generates the next sequential PC value for instruction fetch by circulating the contents of PC register around an incrementing loop. Each PC value generated by the loop is also copied to the PCpipe associated with the address interface as it is dispatched to the memory interface. Figure 8 illustrates the data flow.

Figure 8: Data flow of PC incrementing loop

The PC incrementing loop has two phases: the initialization phase and the continuation phase.

1. The *initialization phase* begins with the MAR being reset to zero. This is the first value to be dispatched to the

MI. This first value is also simultaneously incremented and saved in the PC register².

2. The *continuation phase* begins with the contents in the PC register being sent to the MAR via the arbitrating multiplexors. Once latched in the MAR, this PC value is then dispatched to the MI via the MAR, and to the PC register after being incremented (for the incrementing loop to continue on).

SAT. A single address transfer interrupts the PC incrementing loop in order to transfer either a regular address or a new PC value to MI, the memory interface. A regular address may be generated by a load, store, and swap operations. A new PC value may be generated by a branch instruction and by any add, load, or software interrupt operation that assigns to R15 (PC). Figure 9 illustrates the possible data flows.

Figure 9: Data flow of single address transfer

The single address transfer begins with an address or a new PC value being supplied on either the W bus or the APIPE. When it wins the arbitrating multiplexor, the PC incrementing loop is temporarily interrupted and the regular address or the new PC value enters the MAR.

3. If a *new PC value* is to be transferred, it is dispatched to the MI and also to the PC register after being incremented.
4. If a *regular address* is to be transferred, it is dispatched to the MI only.

MAT. Multiple address transfer generates and transfers a set of sequential instruction addresses based upon a base value supplied to the address interface while executing the load/store multiple operation. Figure 10 illustrates the data flow. The multiple address transfer has an initialization phase and a continuation phase:

5. The *initialization phase* begins with the base address arriving on the W bus. When it wins the arbitrating multiplexor, the PC incrementing loop is temporarily interrupted and the base address enters the MAR. The address interface then uses the information from the decode unit to detect whether this is the end of the transfer. If this is the last address to be transferred (5a), it is dispatched to the MI only and the multiple address transfer completes.

²This first PC value will not be placed in the PCpipe.

Figure 10: Data flow of multiple address transfer

If not (5b), it is dispatched to both the MI and to the LSM register after being incremented. The multiple address transfer then enters its continuation phase.

6. The *continuation phase* of the multiple address transfer begins with the contents in the LSM register being dispatched to the MAR directly without any contention from other resources (the multiple address transfer is the sole user of the address interface until the end of the transfer is detected and completed). The address interface again uses the information from the decode unit to detect whether this is the end of the transfer. If the end is detected (6a), the address is dispatched to the MI only and the multiple address transfer completes. If not (6b), the address is dispatched to both the MI and the LSM register after being incremented, and the multiple address transfer continues.

Only one of the PCL, SAT, and MAT types of access can be granted at a time. The general operating sequence of the AI is:

1. The address interface is invoked by the initialization phase of the PCL during which the MAR is reset to zero.
2. Once the address interface is initialized, it enters the continuation phase of the PCL, which is the main function of the address interface.
3. After the initialization phase is completed, both the SAT and the MAT requests may compete with the continuation phase of the PCL for access to the address interface. If a SAT or a MAT requests gains access, the PCL is temporarily interrupted.

5 Model

Figure 11 gives an initial view of our model of the address interface hardware in which we treat the six distinct requests as independent agents. Our model is partitioned into three major units, the arithmetic unit, the memory access unit, and a “decision unit”.

Figure 11: AI accesses

- The *decision unit* guarantees that at any time, only one of the six distinct access requests to the address interface will be granted. It generates the control signals for accessing the arithmetic unit and the memory access unit:
 - inputs of all the access requests, cases 1–6, are dispatched to the memory interface³;
 - inputs of cases 1, 2, 3 are dispatched to the MAR and then to the PC register;
 - inputs of cases 5b, 6b are dispatched to the MAR and then to the LSM register;
 - inputs of cases 4, 5a, 6a are dispatched to the memory interface only, they have no access to the arithmetic unit.
- The *arithmetic unit* accepts access requests from the decision unit:
 - inputs of cases 1, 2, 3 are incremented and then dispatched to the PC register;
 - inputs of cases 5a, 6a are incremented and then dispatched to the LSM register;
 - inputs of cases 4, 5b, 6b bypass the INC, the PC register, and the LSM register.

These access requests are routed through an XOR module after their corresponding accesses have been carried out.

- The *memory access unit* accepts access requests from the decision unit: inputs of cases 1–6 are dispatched to the memory interface via the MAR.

Acknowledgements to the MAR are collected by a C-element after the corresponding access needs have been carried out in both the arithmetic unit and the memory interface.

Decision Unit. Some of the access requests to the address interface are naturally mutually exclusive: e.g. initialisation, and the two varieties of SAT request. The initialisation phase of PCL is called only once and may share an arbitration port with the continuation phase. A two-way call module (CALL) is used here.

³except the old PC value residing in the PC register after a new PC value has been transferred.

Figure 12: Control flow of AI accesses: decision unit

SAT and MAT both contend for the address interface with the PCL. Even though SAT and MAT are exclusive operations, how they operate is quite tricky and we give them each a port to a 3-user arbiter to make the presentation more comprehensible. For extra clarity, our arbiter uses the RGDA protocol rather than the RGD protocol of the actual implementation.

Once the access to the address interface is granted to the initialization phase of the MAT, the arbiter will not be released until the continuation phase of the MAT is completed, and thus there is no need for the continuation phase of the MAT to compete for the address interface.

After the arbitration via the arbiter, a decision call unit (DCALL) is used to accept the mutually exclusive access requests from the PCL, the SAT, the initialization phase of the MAT, and the continuation phase of the MAT. The DCALL is a small state machine which generates control signals for accessing the arithmetic unit and the memory address unit according to the corresponding access needs. Its operation sequences in CCS are

$$\begin{aligned}
 DCALL &\stackrel{def}{=} r1.'rinM.ainM.'isP.'a1.DCALL \\
 &+ r2.'rinM.ainM.('isP.'a2.DCALL' + 'ntP.'a2.DCALL) \\
 &+ r3.'rinM.ainM.('isE.'a3.DCALL + 'ntE.LOOP) \\
 LOOP &\stackrel{def}{=} r4.'rinM.ainM.('isE.'a4.'a3.DCALL + 'ntE.'a4.LOOP) \\
 DCALL' &\stackrel{def}{=} r1.'a1.DCALL \\
 &+ r2.'rinM.ainM.'ntP.'a2.DCALL' \\
 &+ r3.'rinM.ainM.('isE.'a3.DCALL' + 'ntE.LOOP') \\
 LOOP' &\stackrel{def}{=} r4.'rinM.ainM.('isE.'a4.'a3.DCALL' + 'ntE.'a4.LOOP')
 \end{aligned}$$

with interpretation:

- Access requests from the PCL ($r1$), the SAT ($r2$), and the initialization phase of the MAT ($r3$) are accepted by the DCALL; access request from the continuation phase of the MAT ($r4$) is accepted by the LOOP. If a PC value is transferred (isP) during the SAT, the DCALL unit enters a new state expressed by the DCALL' and the LOOP'.
- The DCALL' may accept access request from the PCL ($r1$), but will throw away the PC value and return to the state expressed by the DCALL and the LOOP. It may also transfer a regular address (ntP) during the SAT, but not a PC value (isP). This new state has no effect on the MAT which is carried out via $r3$ in DCALL' and $r4$ in LOOP'.

Arithmetic Unit. The register transfer level implementation of the arithmetic unit is illustrated in Figure 13. The arithmetic unit accepts mutually exclusive access requests from the decision unit. They are: isP , ntP , isE and ntE .

Figure 13: Control flow of AI accesses: arithmetic unit

- if a PC value is transferred (isP), it is dispatched to the PC register after being incremented by the INC;
- if a regular address is transferred (ntP), it bypasses the INC and the PC register;
- if a LSM address (not the last one) is transferred (ntE), it is dispatched to the LSM register after being INCed;
- if the last LSM address is transferred (ntE), it bypasses the INC and the LSM register.

The mutually exclusive accesses to the INC from isP and ntE are guaranteed by a two-way call module (CALL). The corresponding acknowledgments ($ainP$, $ainL$) to the INC ($aoutI$) are collected by an XOR module. These acknowledgments, together with ntP and isE , are collected by a three-input XOR module for acknowledging the MAR in the memory access unit.

Memory Access Unit. The register transfer level implementation of the memory access unit illustrated in Figure 14 remains the same as that illustrated in Figure 11. Input to the memory interface is first latched in the MAR upon receiving $rinM$ from the decision unit. It is then dispatched to the memory interface via $routM$. Acknowledgments to the MAR ($aoutM$): $ackMI$ from the memory interface and the output of the three-input XOR module in the arithmetic unit, are collected by a C-element.

Figure 14: Control flow of AI accesses: memory access unit

Putting it all together. The register transfer level implementation of the complete control flow of accesses to the address interface is illustrated in Figure 15.

Figure 15: Complete control flow of AI accesses

To access the address interface, the access requests for the initialization phase of the PCL, the SAT, and the initialization phase of the MAT come from the environment; the access requests for the continuation phase of the SAT and the continuation phase of the MAT come from the PC register (*rouP*, *ainP*) and the LSM register (*rouL*, *ainL*) respectively.

6 Property checking

Although we can never be sure that a specification is correct, we can increase our confidence in its appropriateness by testing its properties, e.g. does the **AI** deadlock; once a request is made, will it get served; once a request is accepted, will it complete; will the **AI** only serve one request at a time? Notice that since they must be valid over all timing sequences (something that comes naturally with the interleaving semantics of CCS), these properties are nigh impossible to verify by simulation.

The processes of CCS generate labelled transition systems of the form $(\mathcal{P}, \mathcal{A}, \mathcal{T})$ where $\mathcal{P}$ is a non-empty set of agents, $\mathcal{A}$ is a set of input actions (α) and output actions ($\bar{\alpha}$), and $\mathcal{T}$ are the transition relations for each α (or $\bar{\alpha}$) $\in \mathcal{A}$. E.g. for the state-by-state specification of *REG* of table 1 (or figure 4)

$$\begin{aligned}
 \mathcal{P} &= \{S_0, S_8, S_{15}, S_{16}, S_{12}, S_9, S_7, S_{14}\} \\
 \mathcal{A} &= \{rin, rout, ain, aout\} \\
 \mathcal{T} &= \{S_0 \xrightarrow{rin} S_8, S_8 \xrightarrow{ain} S_{15}, S_8 \xrightarrow{rout} S_{12}, \dots, S_{14} \xrightarrow{ain} S_0\}
 \end{aligned}$$

In all, there are 12 transitions in $\mathcal{T}$.

6.1 Expressing properties in modal temporal logic

HML – Hennessy-Milner Logic. The syntax of HML formulae is:

$$A ::= T \mid \neg A \mid A \wedge A \mid \langle a \rangle A \mid [a]A$$

with interpretation⁴: T is the constant true formula; $\neg A$ is a negated formula; $A \wedge A$ is a conjunction; $\langle a \rangle A$ is read as “ A holds after some a action”; $[a]A$ is read as “ A holds after all a actions”. Since the formulae are parameterised by the action set, each transition system has its own associated HML.

For a given fixed transition system, we now define when a process $E \in \mathcal{P}$ satisfies the property A (written $E \models A$):

1	$E \models T$	$\forall E$
2	$E \models \neg A$	iff $E \not\models A$
3	$E \models A \wedge B$	iff $E \models A$ and $E \models B$
4	$E \models \langle a \rangle A$	iff $\exists E' \in \mathcal{P}, a \in \mathcal{A}. E \xrightarrow{a} E' \text{ and } E' \models A$
5	$E \models [a]A$	iff $\forall E' \in \mathcal{P}, a \in \mathcal{A}. E \xrightarrow{a} E' \supset E' \models A$

with interpretation

1. Every process in $\mathcal{P}$ satisfies property T
2. A process has property $\neg A$ when it fails to have property A
3. A process has property $A \wedge B$ when it has property A and it has property B
4. A process satisfies $\langle a \rangle A$ if there exists one a action which the resulting process has property A
5. A process satisfies $[a]A$ if after every performance of an a action all the resulting processes have property A

Example: REG.

1. $S_0 \models \langle rin \rangle T$	REG can accept an rin in state S_0
2. $S_0 \models \neg(\langle rout \rangle T)$	REG cannot signal on $rout$ in state S_0
3. $S_8 \models \langle ain \rangle T \wedge \langle rout \rangle T$	REG in state S_8 can send a signal on ain and on $rout$
4. $S_{15} \models \langle rin \rangle T \wedge \overline{\langle rout \rangle T} \wedge \neg(\langle ain \rangle T)$	REG in state S_{15} can accept rin , and send on $rout$ but not on ain

We make HML more convenient to use by allowing a notational extension illustrated by examples: $[-] \stackrel{def}{=} \text{‘all actions } \mathcal{A}\text{’}$, and $[-k,l,m] \stackrel{def}{=} \text{‘all actions except } k,l,m \text{ in } \mathcal{A}\text{’}$. In particular

$E \models [a] F$	E cannot do an a move
$E \models [-] F$	E cannot make any further move, i.e. is deadlocked
$E \models \langle - \rangle T$	E can make some move, i.e. is live
$E \models \langle a \rangle T \wedge [-a] F$	E can only do an a move

The counter $CNT(n = 1)$ has the property that it can be repeatedly incremented and then decremented. An obvious notation in which to express this infinite behaviour is: $CNT(n = 1) = \langle \overline{inc} \rangle \langle \overline{dec} \rangle CNT(n = 1)$. Such a fix point equation may have no solutions (e.g. $X = \neg X$) or several solutions. There will always be at least one solution provided that each fix point variable is within the scope of an even number of negations. From now on, we assume that all our modal formulae pass this simple syntactic test.

We may associate with a property the set of states satisfying it:

⁴We derive $F, \vee, \supset, \equiv, \dots$ from the basic operators. Notice that $[-]$ and $\langle \rangle$ are duals of each other since $\langle a \rangle A = \neg[a]\neg A$ — only one need be defined as a primitive.

$\ A \ $	$\stackrel{def}{=}$	set of all states satisfying A
$\ T \ $	$\stackrel{def}{=}$	true of all states = $\mathcal{P}$
$\ F \ $	$\stackrel{def}{=}$	true of no state = $\emptyset$
$\ \neg A \ $	$\stackrel{def}{=}$	$\mathcal{P} - \ A \ $
$\ A \wedge B \ $	$\stackrel{def}{=}$	$\ A \ \cap \ B \ $
$\ \langle a \rangle A \ $	$\stackrel{def}{=}$	true iff it is possible to do an a and move to a state enjoying A
$\ [a] A \ $	$\stackrel{def}{=}$	true iff however we do an a we move to a state enjoying A. Note that this is TRUE if we cannot do an a

Here are some satisfaction relations over the $CNT(n = 2)$:

Property p	Set of states with p	Set description
$\ T \ $	$= \{ CNT_0, CNT_1, CNT_2 \}$	all states
$\ \langle - \rangle T \ $	$= \{ CNT_0, CNT_1, CNT_2 \}$	all non-deadlocked states
$\ \langle \overline{inc} \rangle T \ $	$= \{ CNT_0, CNT_1 \}$	states that permit an $\overline{inc}$ action
$\ \langle \overline{dec} \rangle T \ $	$= \{ CNT_1, CNT_2 \}$	states that permit an $\overline{dec}$ action
$\ \langle \overline{inc} \rangle T \wedge [\overline{dec}] F \ $	$= \{ CNT_0 \}$	states that permit $\overline{inc}$ but not $\overline{dec}$
$\ \langle \overline{inc} \rangle T \wedge \langle \overline{dec} \rangle T \ $	$= \{ CNT_1 \}$	states that permit $\overline{inc}$ and $\overline{dec}$ actions
$\ [\overline{inc}] F \wedge \langle \overline{dec} \rangle T \ $	$= \{ CNT_2 \}$	states that permit $\overline{dec}$ but not $\overline{inc}$

In general when we look for the fix points of a formula several sets of states might be solutions. If we wish to find all the fix points, we have to test all the possible combinations from the empty set, the sets of singletons, two states at a time, ..., all the way up to $\mathcal{P}$. It turns out that the fix points form a lattice and that the “least” and “largest” of solutions are not only unique, but they also have interesting physical interpretations and fast algorithms. The minimum (least) fixpoint includes only what is necessarily true. It expresses *liveness*: e.g. **a** must eventually happen. It is found by iteration: we start from the empty set of states and include what must be there. The maximum (largest) fix point includes everything except that which is necessarily false. It expresses *safety*: e.g. **a** holds everywhere. It is found by iteration: start with all possible states and pare out those found wanting.

The modal μ calculus extends HML with fix points: $A ::= HML \mid \min(X.A) \mid \max(X.A)$ ⁵ where X is a fix point variable. As an example, a test for the absence of deadlock is: $\max(X. \langle - \rangle T \wedge [-]X)$. It expresses the set of states S which can themselves make a move ($\langle - \rangle T$) and from which all moves ($[-]$) takes us to a member of the set of states S which can ... Thus, no member of this set of states is incapable of making a move. Properties expressed in raw modal μ are rather hard to read. However it is easy describe a small set of expressive and composable operators in modal μ :

ALWAYS P	$\max(X.P \wedge [-]X)$	needs all states on all paths as witnesses
PATH P	$\max(X.P \wedge (\langle - \rangle X \vee [-]F))$	needs all states on a single path to be witnesses
EVENTUALLY P	$\min(X.P \vee ([-]X \wedge \langle - \rangle T))$	needs a single witness on all paths
POSSIBLE P	$\min(X.P \vee \langle - \rangle X)$	needs only a single state on a single path as a witness

ALWAYS and POSSIBLE are duals; so are EVENTUALLY and PATH. PATH may end in a deadlocked state; EVENTUALLY cannot. Here are some common properties in terms of these operators:

⁵Since min and max are dual operations, only one need be defined as a primitive.

DEADLOCK means that a system may reach a state in which it cannot make a move (is stuck). For any system SYS , absence of deadlock may be expressed as:

$$SYS \models ALWAYS \langle \rightarrow \rangle T$$

read as “in every state (ALWAYS), it is possible to make a move ($\langle \rightarrow \rangle T$)”.

LIVELOCK means that a system may loop on internal handshakes (τ moves):

$$SYS \models PATH \langle \tau \rangle T$$

informally understood as there exists a path along which one can move by τ moves alone. If the system is deadlock free, the path will loop. Otherwise it might consist of a number of τ moves into a deadlocked state. That a path is reachable from SYS is a property of max fix points, i.e. there is no need to write POSS (PATH $\langle \tau \rangle T$).

LIVENESS tests to see that good things may happen, e.g. each request may be accepted. For any system SYS , and a particular action a , liveness of action a may be expressed as:

$$SYS \models ALWAYS POSSIBLE \langle a \rangle T$$

read as “from every state (ALWAYS) there exists a path (POSSIBLE) to some state where action a is enabled ($\langle a \rangle T$)”. EVENTUALLY is stronger than POSSIBLE, e.g. for any system SYS , and for a particular action a

$$SYS \models ALWAYS EVENTUALLY \langle a \rangle T$$

reads “from every state (ALWAYS) for each path (EVENTUALLY) a state will be reached where action a is enabled”.

SAFETY tests to see that bad things cannot happen. Safety tests must be tailored to the system at hand. For $CNT(n=2)$, we may want to check that it is never possible to increment three times without doing a decrement.

$$CNT_0 \models ALWAYS [\overline{inc}][\overline{inc}][\overline{inc}]F$$

read as “in every state (ALWAYS) it is not possible to perform three consecutive $\overline{inc}$ actions”.

6.2 Property checking the AI

Although the PCL, SAT, and MAT may require accesses to the **AI** at unpredictable times, they have to obey the following environmental constraints:

1. The first access request to the address interface has to be the initialization phase of the PCL ($reqENV_PCL$);
2. Once the address interface has been initialized ($ackENV_PCL$), both the single address transfer ($reqENV_SAT$) and the multiple address transfer ($reqENV_MAT$) may arbitrate with the continuation phase of the PC incrementing loop. However, ($reqENV_SAT$) and ($reqENV_SAT$) are mutually exclusive. The winner has the sole access to the address interface.

To incorporate these constraints, we augment our compositional specification of the **AI** with

$$\begin{aligned} INIT &\stackrel{def}{=} reqENV_PCL.\overline{reqPCL}.\overline{ackPCL}.\overline{ackENV_PCL}.INIT \\ ENV &\stackrel{def}{=} reqENV_SAT.\overline{reqSAT}.\overline{ackSAT}.\overline{ackENV_SAT}.ENV \\ &\quad + reqENV_MAT.\overline{reqMAT}.\overline{ackMAT}.\overline{ackENV_MAT}.ENV \\ AI1 &\stackrel{def}{=} (AI \mid INIT \mid ENV) \setminus \{ackMAT, ackPCL, ackSAT, reqMAT, reqPCL, reqSAT\} \end{aligned}$$

Deadlock. The specification $AI1$ was submitted to the CWB for minimisation (which takes the human-readable compositional specification and automatically transforms it into the equivalent monolithic state machine with the least number of states, cf. the two specifications of REG in table 2 (compositional) and table 1 (minimised). After minimisation, the deadlock detection check failed. min is the CWB command to minimise an agent. cp is the CWB command to check a proposition from a specified initial state. Since it uses teletype, the CWB writes ‘ a ’ for $\bar{a}$.

```
...
min AI1
AI1'
AI1' has 90 states.
```

```

cp AI1'
ALWAYS <->T
false

fd AI1'
--- reqENV_PCL mar' mem' sinc' pc' 'ackENV_PCL reqENV_SAT
    t<rl> mar' t<routP> mem' sinc' 'ackENV_SAT reqENV_MAT t<routP>
---> AI1'minState335

```

The check using ALWAYS fails (there is deadlock) but offers no help in telling why. The trace sequence generated by the CWB's built-in find deadlock operation (fd) led us to the following chain of events causing the deadlock:

1. After the continuation phase of a PCL is interrupted by a SAT during which a PC value is transferred (*isP*), the old PC value generated by the PCL remains in the PC register while the new PC transferred by the SAT enters the MAR.
2. The new PC value in the PC register cannot circulate any further until the PC register is free: the old PC value wins the ARB and is then discarded immediately.
3. At this point, if another SAT wins the ARB instead of the old PC value in the PC register, deadlock results:
 - the address to be transferred by the SAT cannot enter the MAR because the new PC value is still occupying it and will not be free until the PC register is empty;
 - the PC register will not be free until the old PC value wins the ARB;
 - the ARB will not be free until the address to be transferred enters the MAR.

This potential deadlock was known to the AMULET design team. Their simple solution was to replace the single PC register by a pair of PC registers coupled together in FIFO fashion. All we have to do in CCS, is change the definition of *PC* as a *REG* with its local *CNT* initialised to 1, to a *REG* with its local *CNT* initialised to 2. The CWB shows this improved design to be deadlock free.

```

...

min AI2
AI2'
AI2' has 162 states.

fd AI2'
**No such agents.

```

Livelock. As with deadlock, this check does not vary from one specification to the next.

```

cp AI2
PATH <t>T
false

```

Because *PATH* is a maxfix point, there is no need to write *POSSIBLE*(*PATH* < τ >T) with the intent of explicitly searching the reachable states to find a start state for a cycle. We check the proposition over *AI2* since the minimised version *AI2'* would have its cyclic τ 's removed.

Liveness. A typical liveness check on the **AI** would be from all reachable states it is always possible to reach a state in which another PCL (req_PCL) is requested. Similarly for SAT and MAT.

```

cp AI2'
ALWAYS ( POSSIBLE <reqENV_PCL> T & POSSIBLE <reqENV_SAT> T & POSSIBLE <reqENV_MAT> T )
true

```

Safety. Typical safety tests on the **AI** might include a check that once a PCL has been initiated it must complete.

```

cp AI2'
ALWAYS [reqENV_PCL](EVENTUALLY <'ackENV_PCL> T)
true
    
```

There are symmetric versions for req_SAT and req_MAT.

Another check might be to ensure that all inputs to the PC register and the LSM register are incremented.

```

cp AI2'
(NEC inc' pc') & (NEC inc' lsm')
true
    
```

where $NEC\ a\ z$ is another useful macro: $NEC\ a\ z \stackrel{def}{=} \max(X.[z]F \wedge [-a]X)$ which ensures that each z action is preceded by an a action. We now use it to ensure that once a PCL request has been initiated, it must complete before another request (from PCL or SAT or MAT) can be initiated.

```

cp AI2'
ALWAYS [reqENV_PCL]( NEC 'ackENV_PCL reqENV_PCL
                        & NEC 'ackENV_PCL reqENV_SAT
                        & NEC 'ackENV_PCL reqENV_MAT
                        )
true
    
```

7 Summary

This paper has presented a RTL specification of the address interface, one of the major floor plan modules of AMULET1. The address interface were classified and their individual control flows were developed, specified, combined, and tested. Its main contribution lies in providing a description of the architecture which captures the designers' intent and can be formally property checked. The description consists mainly of standard building blocks or regular structures (e.g. incrementers, FIFOs), whose implementation techniques are well known. All the irregular logic has been concentrated into DCALL and Steven's PhD [20] describes techniques for converting such state machines into efficient logic. Thus this level of specification serves as a design guide for lower level implementations. Major initial decisions were choosing a suitable and mechanised notation and deciding upon the level of abstraction at which to pitch the specification. [14] gives fairly complete specifications and property checks of an "instruction-level" view of AMULET1 and of its other floor plan elements. The address interface was subjected to a range of property checks resolving such issues as deadlock, liveness, and safety properties by static testing. [18, section 2, pages 177–387] is a very readable and practical account of property checking. [1] gives the intuition and background to minimum and maximum fix points. Stirling's [21, 22, 23] are all excellent accounts of process logics and CCS.

Acknowledgements

This work is a condensation of material contained in Ms. Liu's PhD thesis [14] and other project reports [3, 15, 16, 17]. We would like to thank Ken Stevens for continuous and insightful help with the intricacies of asynchronous design in general and microprocessor design in particular — he is always ready to help and discuss. The AMULET team at Manchester gave us a great deal of help and guidance over a two year period. In particular, Doug Edwards gave us our initial understanding of the AMULET architecture and planted the possibility of its specification in our minds; and Jim Garside and Nigel Paver patiently guided us through the intricacies of the design and were never stumped or flustered by our queries whatever their technical depth or angle. The research was partially supported by an Operating Grant from Natural Science and Engineering Research Council of Canada; and by workstations from the Canadian Microelectronics Centre. Dr. Liu was supported by scholarships from the Alberta Microelectronic Corporation and the Killam Foundation.

References

- [1] J. Aldwinckle, R. Nagarajan, and G. Birtwistle. An Introduction to Modal Logic and its Applications on the Concurrency Workbench. Technical Report, Computer Science Department, University of Calgary, 1991.
- [2] G. Birtwistle and A. Davis, editors. *Proceedings VII Banff Workshop: Asynchronous Digital Circuit Design*. Springer Verlag, Workshops in Computing Series, London, 1995.
- [3] G. Birtwistle and Y. Liu. Specification of the Manchester AMULET1: Top level. Computer Science Department Technical Report, University of Calgary, January, 1995.
- [4] G. Birtwistle, Y. Liu, D. Spooner, J. Aldwinckle, and K. Stevens. Case Studies in Asynchronous Design. Part I: AMM — an Asynchronous MOVE Machine. Computer Science Technical Report, University of Calgary, 1994.
- [5] G. Birtwistle, Y. Liu, D. Spooner, J. Aldwinckle, and K. Stevens. Case Studies in Asynchronous Design. Part II: a 4 stroke AMM. Computer Science Technical Report, University of Calgary, 1994.
- [6] G. Bruns. A Language for value-passing CCS. Technical Report LFCS ECS-LFCS-91-175, Edinburgh University, 1991.
- [7] E. Brunvand. Translating Concurrent Communicating Programs into Asynchronous Circuits. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.
- [8] Ed. S. Furber. *AMULET1 Modelling Workshop*. Computer Science Department, University of Manchester, Windermere, Cumbria, England, July 18-22, 1994.
- [9] S. Furber. *VLSI RISC Architecture and Organisation*. Marcel Dekker, Amsterdam, 1989.
- [10] S. Furber. Computing without Clocks: Micropipelining the ARM Processor. In G. Birtwistle and A. Davis, editor, *Proceedings VII Banff Workshop: Asynchronous Digital Circuit Design*, pages 211–262. Springer Verlag, Workshops in Computing Series, 1995.
- [11] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V. Woods. A Micropipelined ARM. In T. Yanagawa and P.A. Ivey, editor, *Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration (VLSI'93)*. North Holland, 1993.
- [12] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V. Woods. AMULET1: A Micropipelined ARM. In *Proceedings of the IEEE Computer Conference*, March 1994.
- [13] Y. Liu. Reasoning about Asynchronous Designs in CCS. MSc Thesis, Department of Electrical and Computer Engineering, University of Calgary, 1992.
- [14] Y. Liu. AMULET1: specification and verification. PhD Thesis, Department of Computer Science, University of Calgary, 1995.
- [15] Y. Liu, G. Birtwistle, and N. Paver. Specification of the Manchester AMULET1: Address Interface. Computer Science Department Technical Report, University of Calgary, April, 1994.
- [16] Y. Liu, G. Birtwistle, and N. Paver. Specification of the Manchester AMULET1: Execution Pipeline. Computer Science Department Technical Report, University of Calgary, June, 1994.
- [17] Y. Liu, G. Birtwistle, and N. Paver. Specification of the Manchester AMULET1: Register Bank. Computer Science Department Technical Report, University of Calgary, June, 1994.
- [18] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: specification*. Springer-Verlag, New York, 1992.
- [19] F. G. Moller. The Edinburgh Concurrency Workbench, Version 6.0. Tech Report, Computer Science Department, University of Edinburgh, 1991.
- [20] K. Stevens. Practical Verification and Synthesis of Low Latency Asynchronous Systems. PhD Thesis, Computer Science Department, University of Calgary, 1994.
- [21] C. Stirling. Modal and Temporal Logics. Tech Report ECS-LFCS-91-157, Laboratory for the Foundations of Computer Science, Computer Science, University of Edinburgh, 1991.
- [22] C. Stirling. Modal and Temporal Logics for Processes. Tech Report ECS-LFCS-92-221, Laboratory for the Foundations of Computer Science, Computer Science, University of Edinburgh, 1992.
- [23] Colin Stirling. An Introduction to Modal and Temporal Logics for CCS. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, number 491 in LNCS, pages 2–20. Springer-Verlag, 1991.
- [24] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.
- [25] A. van Someren and C. Attack. *The ARM RISC Chip: A Programmer's Guide*. Addison Wesley, 1994.