The Amulet chips: Architectural Development for Asynchronous Microprocessors

J.D. Garside, S.B. Furber, S. Temple, J.V. Woods
School of Computer Science, The University of Manchester, Oxford Road, Manchester, M13 9PL, UK.

{jgarside, sfurber, stemple, jvwoods}@cs.man.ac.uk

Abstract

During the 1990s a series of asynchronous microprocessors based on the ARM architecture was developed at the University of Manchester. The objective was to demonstrate that it was feasible to implement a commercial architecture with asynchronous logic and that certain advantages could be gained from a self-timed processor. By carrying these designs through to silicon it was demonstrated that processors, caches and whole systems-on-chip could be built without clocks and could perform competitively with 'conventional', synchronous systems.

These processors, named Amulet1-3, exhibited some useful characteristics, particularly their ability to stop and start almost instantaneously - with consequent energy savings in embedded applications - and their low-energy and wide spectrum electromagnetic emission - useful in wireless applications. As might be expected they also adapt automatically to supply voltage variation so that their speed and energy demands may be regulated simply by altering their power supply. There were disadvantages too, particularly in the design effort which was not well supported by conventional tool flows. Thus, at that time, the advantages were not great enough for any widespread commercial acceptance.

This paper summarises the architectural features which were developed during the Amulet series of microprocessors and systems with emphasis on mechanisms which are non-specific to the ARM architecture. These include register forwarding and cache-line fetching which typically rely on non-local synchronisation provided by a clock signal. In addition deadlocks, an ever-present danger in asynchronous systems, are discussed and means of preempting them presented.

1. Introduction

Many early processors were clockless but, towards the end of the last century, the convenience of using a clocked model for logic design had become overwhelmingly dominant. Continuous rapid development of electronic technology means that this paradigm has become entrenched as *the* method of digital design. Attempts to develop alternative logic styles must therefore struggle to catch a fast-moving target propelled by considerable financial incentive.

The Amulet chips were asynchronous ARM microprocessors and systems on chip developed to investigate the feasibility and commercial potential of asynchronous technology. This paper summarises the major, transferrable contributions developed for these devices.

Two notable, relevant works preceded the Amulet projects: work at Caltech had demonstrated the feasibility of implementing asynchronous circuits on silicon [1] and Sutherland's Turing-award presentation [2] set the initial style. However there remained considerable development work to demonstrate complete, contemporary, commercial ISA compatibility and, possibly, some unique advantages of the technology. At the time power dissipation in processors (such as Alpha [3]) was beginning to become a serious issue, especially with a demand high-performance portable devices. The initial objective was therefore to exploit the promise of power saving by removing of the clock and halting whenever possible.

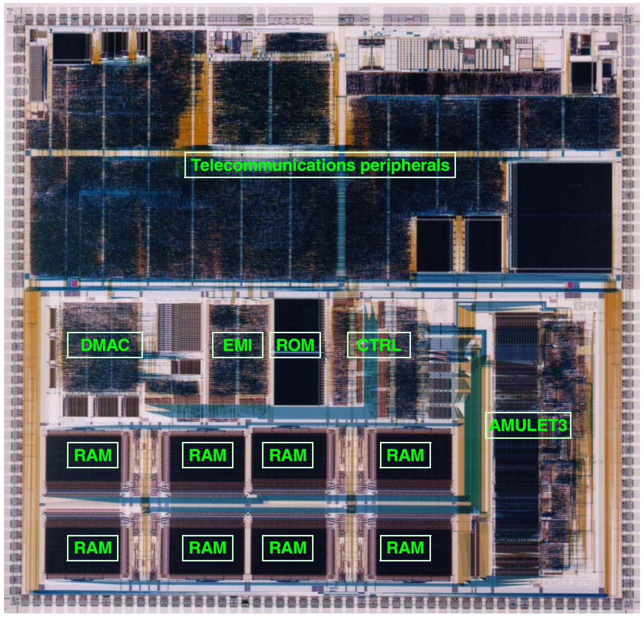

Figure 1: DRACO layout

2. Amulet chips

Before outlining any architectural details, a brief overview of the Amulet chips may be in order. Amulet1, manufactured 1993, was a 1.0 µm device, similar to ARM 6 and delivering around 16 MIPS. Amulet2e was an improved processor core, similar to ARM7, delivering nearly 40 MIPS on a 0.5 µm process, integrated with an asynchronous cache, a few peripherals and an easy-to-use external interface. It was manufactured in 1996.

The final device in the series was DRACO (fig. 1), a DECT base station produced in collaboration with Hagenuk GmbH who designed the synchronous peripherals. This device contained Amulet3i, an asynchronous subsystem comprising an upgraded and re-engineered processor core, similar to ARM9 and including Thumb support, tightly coupled 'dual-port' RAM, ROM and an asynchronous DMA controller. Delivered in 2000 on a 0.35 µm process the processor provided over 100 MIPS – the same as a contemporary ARM9. DRACO was a commercial prototype which was functional but not developed further due to unrelated, financial difficulties.

3. Problems in asynchronous architectures

What is different about an asynchronous microarchitecture? Fundamentally, there is no clock to coordinate disparate parts of the system so that any parts which must communicate have to synchronise locally and temporarily. This is typically done with a Muller C-element [4] which acts as an AND gate for logic transitions although, as illustrated later, there are other ways of synchronising if it is not certain that a complete set of inputs must be awaited.

Another widely employed specialist element is an arbiter or mutual exclusion element [5], which prevents more than one of its inputs being output at any time. Such units are used specifically to avoid synchronisation but introduce non-determinism into the circuit's operation since they are only necessary when inputs compete for a service and it is not possible to ordain which will be serviced first.

One characteristic which complex asynchronous systems share with synchronous ones is pipelining. Pipelines are a 'natural' match for circuits which repeatedly part-process data then pass on results. Indeed there is a temptation to over-pipeline functions which can increase latency and thus reduce performance. Asynchronous pipeline stages synchronise with their neighbours via handshakes when communicating but are not coordinated with more distant circuits, thus arbitrary passing of information around the system is not possible.

Because data flow through the pipeline at their own rate the occupancy of the pipeline is *elastic* – i.e. it is uncertain at any given time. Although data remain in order – at least in a simple pipeline structure – operations such as flushing following a branch may be awkward due to this uncertainty.

Probably the most challenging problem is avoiding deadlocks. A synchronous system tends to be pushed forwards by its clock but an asynchronous system waits until all expected processes meet. If a process stalls, other processes will wait for it. It is essential that there is never a 'circle of dependency' which can cause a deadlock. This problem is exacerbated if the asynchronous machine is allowed to behave non-deterministically because the number of reachable states expands rapidly and conventional logic simulation is unlikely to explore them all.

Synchronous logic is deterministic; each processing step can be timed by counting a number of clock cycles. Asynchronous logic may be non-deterministic if a function is granted 'on demand' because the time is not quantised. Whilst this *need* not feature greatly in an asynchronous design it may be desirable in the interest of performance; unpredicted behaviour may then emerge.

For example, like most processors Amulet1 accessed the memory from both the instruction fetch and data processes (fig. 2). As these are at different stages in an asynchronous pipeline they compete for the memory's attention. Because not every instruction needs a data access there was no centralised control: requests were made as necessary via an asynchronous arbiter and serviced as soon as possible.

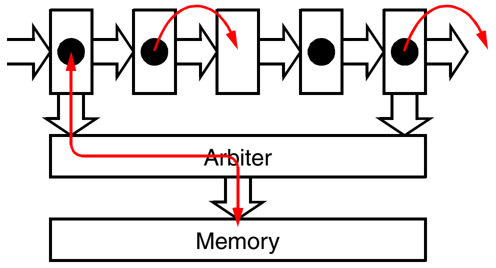

Figure 2: Async. pipeline with shared resource

A deadlock may occur as follows. Slow instruction execution has filled the (elastic) prefetch buffer as a data load reaches is execution point. If a new instruction fetch wins the arbitration for the memory it stalls the load. The fetch returns data waits until space is freed in the subsequent pipeline, which is stalled until the load can complete.

4. Example microarchitectural solutions

4.1. Deadlock avoidance

To prevent the deadlock described in the previous section it is necessary to delay the last instruction fetch from beginning. This was done using another pipeline in parallel with the memory but one stage shorter, which 'backs up' first and stalls the fetch stage. This pipeline was already present because – in the ARM architecture – the PC is available to the execution unit but is resident in the prefetch unit. Simply reading the PC when desired, as in contemporary synchronous ARMs – was not feasible as the units are desynchronised.

In Amulet3 a Harvard bus structure was employed for the processor which means this problem is exported to the memory system. A similar mechanism is used to delay incoming accesses until they can complete.

4.2. Distant 'synchronisation'

Choosing one of a set of asynchronous requests risks metastability if requests arrive (nearly) simultaneously. Arbiters can resolve this safely but lead to non-deterministic operation making system verification more complex.

Metastability is only risked when an opportunity is withdrawn. As an analogy, consider a set of traffic signals: the only time when there should be any indecision is if a light changes from green to red whilst it is in view. If the light is set red sometime before it is in view – and only changes once – then there may be delays but there is no ambiguity in function. Unlike a pre-planned rendezvous this operates regardless of the volume of traffic.

This property was exploited in several ways in the Amulet systems. One example was in Amulet3's register forwarding mechanism (fig. 3) [6]. It is impossible to predict if a register will be read in a given interval but it cannot be read until at least the next instruction. During the dispatch of one instruction a destination is assigned in a reorder buffer; a subsequent instruction can then be directed to read this location and, if necessary, will stall until it has arrived. The result may already be present or may arrive at any later time and this process flow is not impeded by the garnering instruction

A similar example is found in the cache controller in Amulet2e [7]: here a cache miss may initiate a line fill which proceeds, asynchronously, in parallel with processor operation (fig. 4). Rather than stalling whilst the line fills the words are invalidated when the memory is first invoked and then revalidated as they arrive. The processor stalls only if a subsequent read is from the line being fetched and the desired word is not yet present. Interestingly, having been designed simply to work reliably, this mechanism provided full hit-under-miss capability which was uncommon in comparable processors at that time.

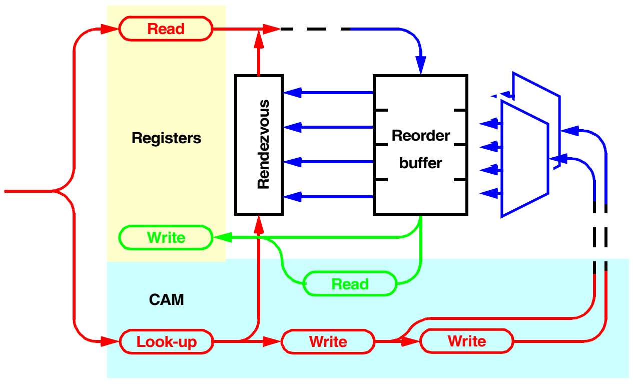

Figure 3: Register forwarding processes

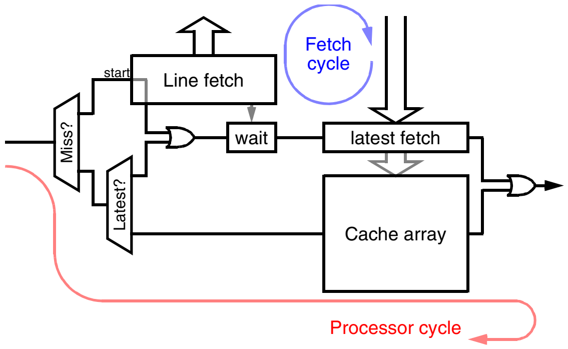

Figure 4: Amulet2e cache dataflows

4.3. Pipeline flushing

All the Amulet processors had a non-deterministic prefetch depth. A taken branch was arbitrated into the prefetch unit. Because operation is pipelined this meant that there were an unknown number of instructions to discard in the pipeline. To solve this difficulty instructions were 'coloured' with the colour changing on each branch. At the execution point a branch caused all subsequent instructions of the branch's (old) colour to be discarded, thus execution continued only when the target instruction had been delivered. Two colours sufficed until Amulet3 – which delayed aborts to reduce the memory latency penalty – where three colours were needed because a branch could be sent before being preempted by the failure of a preceding load.

5. Other microarchitectural techniques

Some examples have already been given; this section summarises other features where asynchronous implementations were produced/adapted in the Amulet processors.

Reasonably deep pipelines were employed: it is difficult to quote exact depths because this can vary dynamically. As a subsystem interacts with handshaking it is possible that data may enter but never leave (e.g. a CMP will not produce a register result) or cause many output handshakes (e.g. a LDM 'LoaD Multiple' instruction spawns 1-16 sub instructions). Units can be constructed hierarchically and assembled without global control signals.

Amulet3 issued instructions in order but allowed them to complete at any time. This allowed speculation on internal operations whilst loads were pending. It is possible to avoid contention by designating destinations for different instruction streams and reordering them subsequently. The reorder buffer allows register state to be preserved in case the memory aborts which is a necessary facility in many computer systems.

Other asynchronous systems were constructed outside the microprocessor core. The most significant in Amulet was MARBLE [8], an AMBA-like clock-free multimaster bus. This was used to interconnect the asynchronous subsystems in Amulet3i and also to bridge onto the synchronous subsystems. It could be controlled from outside the chip for test purposes. Ironically, one of the more difficult problems here was providing a reliable on-chip delay for access to the off-chip *asynchronous* DRAM; this is because commodity memories, whilst self-timed internally, do not have handshake interfaces!

6. Asynchronous advantages

Several advantages have been claimed for asynchronous circuits by various researchers. The two which were apparent from the Amulet processors were power saving and electromagnetic compatibility (EMC).

The energy per operation of the Amulet processors were close to equivalent to their synchronous counterparts. This is a good comparison because equivalents running the same code and manufactured on the same process were available. The style of design used a similar, custom datapath and the removal of the clock was offset by the need for asynchronous control circuits. However when running embedded code with real-time constraints the Amulets exploited the ease by which an asynchronous processor can be halted and restarted and, when halted, power consumption dropped to almost zero as there was no switching activity. Whilst conventional processors increasingly employ power-saving 'sleep' modes asynchronous systems adapt easily and automatically. In addition an asynchronous system is able to adapt its speed to its supply voltage making dynamic voltage scaling (DVS) [9] very simple. A rough approximation is that switching speed is proportional to supply voltage but energy is proportional to its square, thus the supply voltage can be used to trade-off speed and power. An asynchronous processor is more convenient in that altering the clock is not a concern: performance tracks the supply automatically.

There is significant evidence that electromagnetic emissions are considerably lessened in asynchronous circuits This is to be expected as switching activity is not correlated across the system so the AC components of the power demand are smaller. Synchronous circuits tend to emit significant energy at the clock frequency and its harmonics. This is evidenced, for example, by a comparison of similar Amulet and ARM processors (fig. 5).

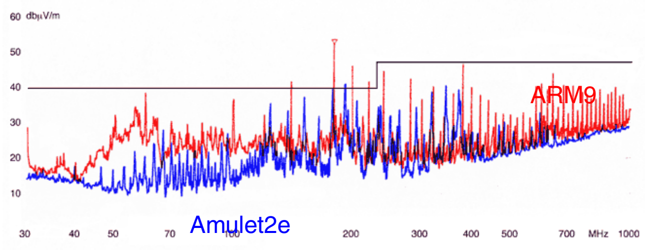

Figure 5: Amulet2e vs ARM7 EMC

7. Conclusions

This paper has sought to illustrate some general techniques which can be applied to asynchronous systems, particularly microprocessors. A summary of some general principles could include:

- Think of a software/system flow, not in terms of cycles
- Pipelining is easy but beware of dependencies
- With non-deterministic access to a shared resource, don't begin until certain that the operation will complete

There were and are some follow-up asynchronous devices in Manchester and those developed by other researchers are too numerous to list here. A decade ago, the complexity of asynchronous development and the limited advantages of the devices failed to make a major mainstream impression.Tool development has since advanced and silicon designers are facing increasing challenges in system timing closure and in managing device manufacturing variation. Asynchronous techniques offer some powerful advantages in overcoming some of these new problems. Thus whilst the Amulet devices are now a historical curiosity it is probable that some of the lessons will emerge again within the next decade.

8. Acknowledgements

The majority of the Amulet development work was funded by the European Commission with various grants as part of the Open Microprocessor Systems Initiative. The authors wish to acknowledge this support.

9. References

[1] A.J. Martin, S. Burns, T.K. Lee, D. Borkovie, P.J. Hazewindus, *The Design of an Asynchronous Microprocessor*, Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI, (1989) MIT Press, pp 351-373.

[2] I.E. Sutherland, *Micropipelines*, Communications of the ACM, Vol. 32, Number 6, June 1989, pp 720-738.

[3] Digital Technical Journal, Volume 4, Number 4, Special Issue 1992 Alpha AXP Architecture and Systems

[4] D.E. Muller, W.S. Bartky, *A Theory of Asynchronous Circuits*, Proc. Int'l Symp. Theory of Switching, Part 1, Harvard Univ. Press, 1959, pp. 204-243.

[5] C.L. Seitz, *System Timing* in *Introduction to VLSI Systems*, C.A. Mead, L.A. Conway (eds.) Addison-Wesley 1980.

[6] D.A. Gilbert, J.D. Garside, *A Result Forwarding Mechanism for Asynchronous Pipelined Systems*, Proc. Async'97, Eindhoven, April 1997, pp. 2-11.

[7] J.D. Garside, S. Temple, R. Mehra, *The AMULET2e Cache System*, Proceedings: Async'96, Aizu-Wakamatsu, Japan, March 18-21 1996.

[8] Bainbridge, W.J., Furber, S.B., *Asynchronous Macrocell Interconnect using MARBLE* Proc. Async'98, San Diego, April 1998 pp. 122-132.

[9] A. Wang, S. Naffziger (Eds.), *Adaptive Techniques for Dynamic Processor Optimization, Theory and Practice*, Springer 2008 ISBN 978-0-387-76471-9