AMULET3i – an Asynchronous System-on-Chip

J.D. Garside, W.J. Bainbridge, A. Bardsley, D.M. Clark, D.A. Edwards, S.B. Furber, J. Liu[1],
D.W. Lloyd, S. Mohammadi[1], J.S. Pepper, O. Petlin[2], S. Temple, J.V. Woods

Dept. of Computer Science, The University of Manchester, Oxford Road, Manchester M13 9PL, UK.
[1]Cogency Technology Inc. 144 Front St. West, Suite 580, Toronto, M5J 2L7, Canada
[2]ASIC Alliance Corporation, 78 Dragon Court, Woburn, MA 01801 USA.
jgarside@cs.man.ac.uk

Abstract

AMULET3i is the third generation asynchronous ARM-compatible microprocessor subsystem developed at the University of Manchester. It is internally modular, being based around the MARBLE asynchronous on-chip bus, and is also extensible through the addition of conventional clocked synthesizable peripherals via an on-chip synchronous peripheral bus. As such it is capable of forming the core of a wide range of system-on-chip applications, bringing asynchronous design into commercial use in a flexible and easy-to-use configuration. Its performance and area are comparable with clocked equivalents, and its low-power and electromagnetic emission characteristics give it unique capabilities in appropriate applications.

1. Introduction

AMULET3i is an asynchronous 'island' containing an AMULET3 microprocessor core and other asynchronous macrocells which can sit in a synchronous sea of gates. It provides a complete, self-contained microprocessor system with RAM, ROM, peripherals and an expansion bus; hardware debugging support and a production test interface are also included. The first AMULET3i based device is scheduled for commercial production in early 2000.

The expansion bus supports conventional, synthesizable synchronous peripherals, enabling AMULET3i to be used as a hard macro at the centre of a wide range of system-on-chip applications which can be developed using conventional design tools.

The asynchronous subsystem is itself modular, being based around the MARBLE asynchronous on-chip bus [1]. Different variations on the asynchronous island can therefore be developed with relative ease (although requiring asynchronous design skills which are outside the capabilities of a conventional design flow).

AMULET3i represents the culmination of almost a decade of research and development within the AMULET group and takes the technology forward through its most significant step to date, into full commercial use.

2. AMULET3i

- AMULET3 microprocessor
- 8 Kbytes RAM
- 16 Kbytes ROM
- Flexible multi-channel DMA controller
- Programmable external memory interface
- MARBLE, a fully asynchronous on-chip bus
- Bridge to on-chip synchronous bus
- Configuration registers
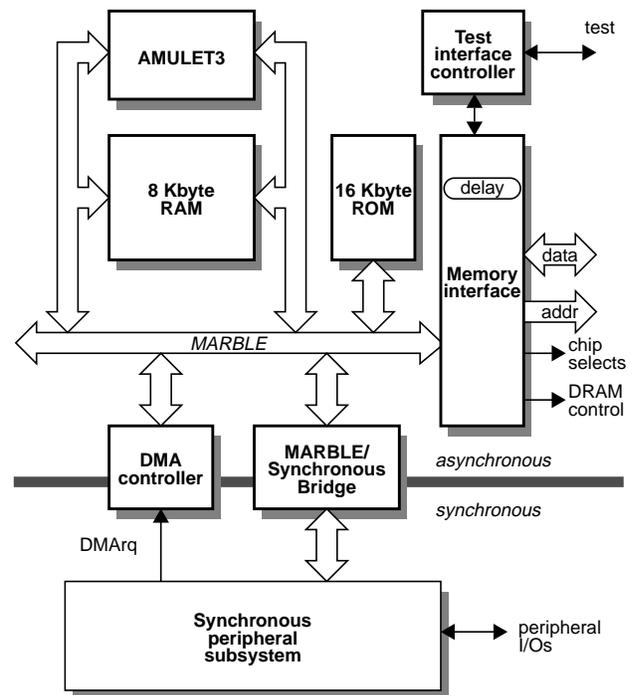- Software debug support
- Test interface

Figure 1: AMULET3i asynchronous subsystem

As shown in figure 1 the majority of the macrocells are interconnected by the MARBLE bus. However the asynchronous RAM is local to the processor to avoid the need for bus arbitration for each instruction fetch. This locality allows the use of separate instruction and data buses although, logically, the RAM is unified (see section 4). The 'local' buses each provide roughly twice the bandwidth available across MARBLE, the instruction fetch bus being somewhat faster than the data bus. The difference in the speeds of these buses is one of the more macroscopic of the system's asynchronous features.

The first use of AMULET3i is as the control system for DRACO (DECT RAdio COmmunication controller) a wireless communications device. DRACO is targeted at the DECT (Digital Enhanced Cordless Telecommunications) multimedia market and contains a large number of conventional, synthesized, synchronous peripheral subsystems (ISDN and DECT controllers, DSP etc.) which inhabit their own synchronous bus to which AMULET3i has access via a bus bridge.

3. AMULET3 microprocessor core

The AMULET3 microprocessor architecture (figure 2) has been described elsewhere [2], so only a brief overview of the more salient and unusual features will be given here.

AMULET3 is an ARM microprocessor. It is fully code compatible with version 4T of the ARM instruction set [3] as used in the ARM9 microprocessor [4]. This also includes full compliance with the 16-bit Thumb instruction set [5].

The processor has a Harvard-like memory architecture which interfaces to local RAM; one port is used for instruction fetches, the other is used for data operations with the exception of direct loads to the PC, which use the instruction fetch port. This facility allows PC loads to overtake some of the data operations which is useful in the standard ARM procedure return where processor context is popped from the stack. The PC, being aliased as R15, is nominally the last register to be loaded; speeding this up reduces the latency involved in starting the next code segment. The degree of overtaking is governed by numerous factors, such as the pipeline occupancy and the speed of the appropriate memory.

The instruction port normally fetches 32-bit words (the exceptions being when it is known that only one half-word is needed). This differs from earlier, synchronous ARM implementations which fetched Thumb instructions at one per clock cycle. AMULET3 therefore makes better use of available bus bandwidth – important when the memory speed is a performance limiting factor – and reduces the number of external accesses so reducing power consumption. This mechanism is assisted by the asynchronous nature of the processor in that the instruction packets may

be analysed and expanded as appropriate further down the pipeline without any need for global control.

The processor itself contains a number of asynchronous stages. Because of its elastic nature the pipeline depth is difficult to define, but there are identifiable Thumb expansion/ predecode, decode/register read and execution stages as well as an asynchronous reorder buffer [6] and register writeback. The data memory interface has its own pipelining and the memory system itself may also be pipelined. Not all these stages are invoked for every instruction – for example some instructions prefetched following a branch may be discarded before reaching the execution stage; contrariwise some instructions may be expanded into multiple cycles inside the pipeline, possibly "backing-up" and stalling the prefetch process.

Internally the pipeline stages interact by asynchronous
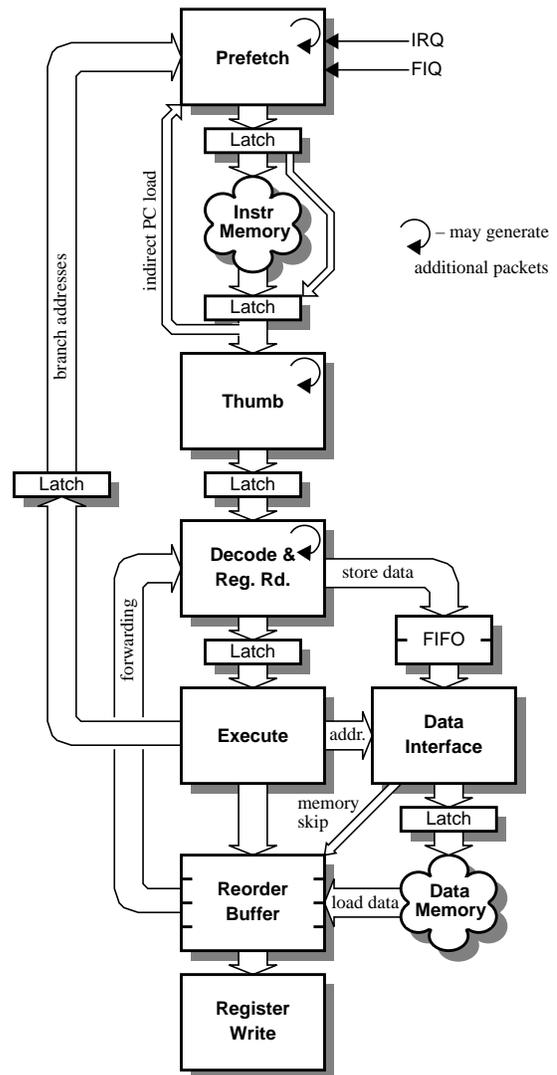

Figure 2: AMULET3 core organisation

handshakes and – whenever feasible – systems are not invoked unless needed. Thus the barrel shifter and multiplier (located within the execution stage) are bypassed most of the time thus speeding up execution and minimising power consumption. Register operand fields are only read onto buses when they are required, again lowering power demands and removing any artificial register dependencies. The register read processes are also mutually asynchronous and tend to become slightly staggered as a consequence of the forwarding mechanism; this is an attempt to spread the power demand over time and consequently reduce electromagnetic interference.

One final subsystem worth mentioning in an asynchronous context is the instruction prefetch unit. This has several novel features. Perhaps the most significant is the "HALT" instruction which has been retrofitted into the instruction set by reinterpreting an instruction which branches to itself. On execution this causes the prefetch unit to stop cycling immediately, thus dropping the power consumption of the processor to near-zero. As the processor instigates most of the activity within AMULET3i this, in turn, effectively turns off the asynchronous subsystem. Full speed operation is restored instantly in response to an interrupt with the first instruction fetched being the beginning of the interrupt service code. Experience with a similar feature in AMULET2e [7, 8] has shown that this leads to very simple yet efficient power management.

The other notable feature of the prefetch unit is its, albeit limited, ability to predict branches. A sixteen entry Branch Target Buffer (BTB) [9,10] is used which reduces the number of instructions prefetched unnecessarily by about half. In addition however *any* branch prediction (whether correct or not) supplies *all* the information contained in the branch instruction. The fetch is therefore suppressed, thus delivering the instruction faster than can be done by the memory system and at a lower energy cost. Even without memory bypass, branch prediction showed a power saving in AMULET2e [8] and with the suppression of perhaps 10% or more of instruction fetches the power reduction is expected to be significant. The BTB can perform this function when running ARM, Thumb or a mixture of code. In Thumb code it is also capable of identifying that only one half word is required and adjusting the fetch size accordingly. This is not significant when running code from the internal memory but will have some effect when using narrower, external store.

4. Local RAM

The AMULET3 processor core has 'local' address and data buses for instruction and data memory accesses (see figure 1). This would normally imply split instruction and data memories; RISC systems frequently employ a 'modi-
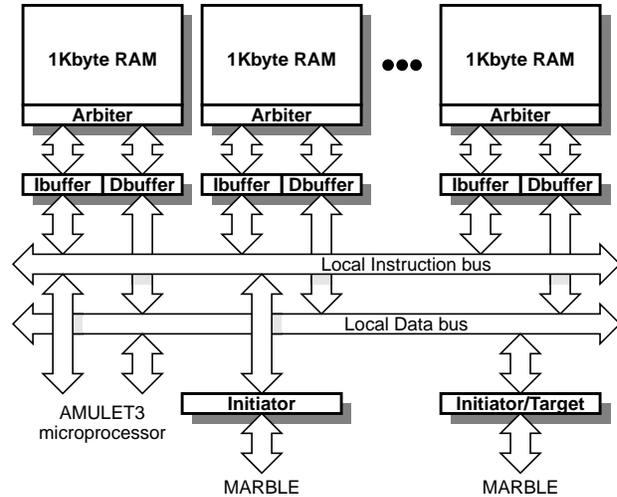

Figure 3: Memory block organisation

fied Harvard' architecture where there are separate instruction and data caches with a unified main memory.

The AMULET3i controller employs memory-mapped RAM rather than cache memory as this is more cost-effective and has more deterministic behaviour for real-time applications. Instead of separate instruction and data memories, a dual-ported, unified memory structure is used (see figure 3). Dual-porting the memory at the individual bit level would be too costly, so the memory is divided into eight 1 Kbyte blocks, each of which has two ports which are arbitrated internally. (Dividing the memory into small blocks also reduces the energy cost of a memory access.) Internally each RAM block is 4 words wide and the blocks are interleaved above this level so that it is unlikely that simultaneous references require the same block. When concurrent data and instruction accesses *are* to different RAM blocks each can proceed unimpeded by the other; when they conflict in the same block, one access may suffer a delay while it waits for the other to complete.

Conflicts (and average memory access time and energy cost) are further reduced by including separate quad-word instruction and data buffers in each RAM block. Each access to a block first checks to see whether the required data is in the buffer. Only if it is not must the RAM be interrogated, with a risk of conflict. Simulations suggest that about 60% of instruction fetches may be satisfied from within these buffers and many short, time-critical loops will run entirely from here. A smaller but still significant proportion of data references also show sufficient locality to benefit from this mechanism.

These buffers, in effect, form simple 128-byte direct mapped first-level caches inside the RAM blocks. This is a particularly apt analogy when it is observed that the avoidance of the RAM array results in a faster read cycle, an
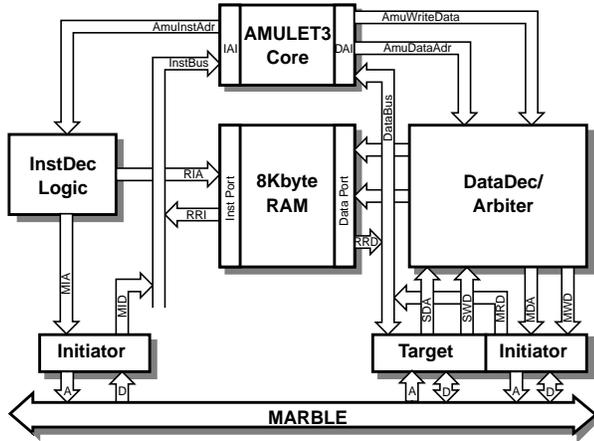
Figure 4: Memory port organisation

occurrence which is exploited automatically by the asynchronous pipeline. Note that cycle speed varies on a word-by-word basis so that sequential reads will induce one 'slow' followed by three 'fast' cycles.

The use of these buffers does not detract from the unified model of the RAM. A (comparatively rare) data write operation is synchronised with any instruction fetch within the same block and, if it addresses the buffered instruction line, the buffer is invalidated. This averts any memory coherency problems and retains a totally unified RAM model. As in a synchronous processor this could be broken by modifying code which has already been prefetched, but the prefetch depth, while non-deterministic, is bounded (at 4 instructions in this implementation).

The two RAM ports are not symmetrical. The processor fetches approximately two instructions for every data access, so it is the instruction local bus that is performance critical and thus is made as simple and as fast as possible. This is assisted by the fact that this is a read-only bus.

The local data bus must provide read, write and atomic read-write transfers (for semaphore operations) and is thus somewhat more complex. Access to the RAM is also required from MARBLE (for DMA transfers), so a MARBLE target interface is provided on the data side. This bus therefore must arbitrate between accesses to the RAM from MARBLE and to the RAM or MARBLE from the processor (see figure 4). The fact that the required data bandwidth is lower than the required instruction bandwidth allows this additional complexity to be supported here with less adverse effect on performance than if the instruction local bus supported the MARBLE accesses to the RAM.

All these effects cause subtle differences in the timings of local RAM cycles. Whilst the asynchronous pipeline accommodates these variations automatically, the 'dithering' of the cycles on the local buses – which contain some of the longest wires on the device – should assist in reducing electromagnetic emissions.

5. MARBLE

Whilst a local memory bus can provide the high bandwidth required by the processor instruction port, its simplicity does not allow for the connection of many peripheral devices. This is the domain of MARBLE, a general purpose multimaster, split transfer, asynchronous system-on-chip bus [1].

MARBLE comprises two multipoint asynchronous channels, the command channel and the response channel. Each channel uses a four-phase signalling protocol with tristate data lines.

The split-transfer nature of MARBLE relates activity on the two channels through the causal dependency that the start of a response cycle will always occur after the start of the corresponding command cycle and provides a throttle on the number of outstanding commands that any one initiator can issue. The control flow for a MARBLE transfer is shown in figure 5 as a token-flow around a loop involving the command and response controllers of one initiator and one target. MARBLE can support multiple outstanding commands but this requires the introduction of a reorder-buffer at the initiator; to avoid this extra complexity the AMULET3i subsystem only permits a maximum of one outstanding command per initiator (i.e. only one token in the control loop). Transfers from different initiators can still interleave at will however.

Figure 5: MARBLE control flow

The first cycle of a MARBLE transfer is the transmission of a command from the initiator. Control information carried on the command channel includes:

- a 32-bit address
- a read/write signal
- a 2-bit initiator identification tag
- a lock signal indicating atomicity of transfer
- a 2-bit transfer size (byte, halfword or word)
- a 3-bit code showing the sequential address relationship of this command with previous/subsequent commands

The address is partially decoded to allow the command to be routed to the appropriate target. The target can then process the command and return a response. The MARBLE

Figure 6: MARBLE interface locations

response channel carries:

- the initiator identification tag (for routeing)
- a copy of the read/write signal for this transfer
- an exception status bit
- 32-bit read data returning to the initiator or write data delivered to the target

Each channel has its own dedicated arbitration network using tree-arbiters to provide the multiway arbitration. MARBLE supports locking of the command channel by an initiator, thus allowing atomic read-modify-write actions upon memory-mapped peripherals and memory, as are required for the implementation of semaphores. Support for deferred transfers allowing bridging of the bus to other multimaster buses is provided as a part of the MARBLE specification, but is not required in the AMULET3i subsystem.
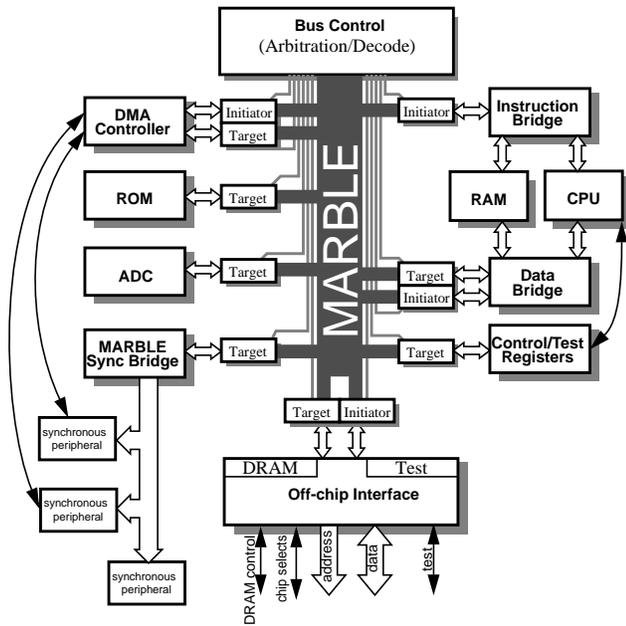
In general, a target's arbitration for the response channel does not occur until the payload (exception status, and any read data) is available, thus providing maximum bus availability so that the bus can be used by other transfers. This behaviour means that the full arbitration delay is incurred as additional latency (this can be up to 2 ns in the AMULET3i subsystem which uses a balanced arbitration tree). This additional penalty is avoided for some of the performance critical targets by starting the arbitration before the payload is available (using a matched delay, 2 ns shorter than the target's response time to indicate when to start the arbitration.

There are a total of 4 initiators and 7 targets connected

to MARBLE in this system. As illustrated in figure 6, transactions may be initiated by the processor reading instructions *or* transferring data, a DMA transfer or from off-chip via the test port; the other ports in this figure are all target interfaces. The split transfer protocol employed in MARBLE allows (for example) the fine-grained interleaving of DMA transfers between the synchronous peripherals and the on-chip memory while the CPU fetches instructions from its local memory and reads data from off-chip. In such a case MARBLE would allow an arbitrary length CPU stall waiting for data without impeding DMA activity.

The MARBLE interfaces use around 2000 transistors each, the majority of these being contained in latches used to store the address and write-data at the target, and the read and write data at the initiator.

The MARBLE interfaces and bus control are compiled from standard cells, except for the drivers and arbiters for which were assembled manually. All control circuits have been designed manually with verification/guidance provided through Petri-net synthesis using Petrify [11].

6. MARBLE/Synchronous bus Bridge (MSB)

Synchronous peripherals can be connected to the AMULET3i subsystem using a simple, synchronous, strobed-access bus. This synchronous bus only hosts peripherals which act as bus slaves; the bridge is the only bus master, which simplifies its design. Initiators on MARBLE can address targets on the synchronous bus through the MARBLE/synchronous bus bridge.

Synchronisation of an asynchronous event with a periodic clock carries a risk of failure. There is only one synchronisation point in the MSB which uses an edge-triggered flip-flop with well understood metastable failure characteristics. The same approach was used in the synchronous peripheral interface of the DMA controller (see section 7.2)

7. DMA controller

The design of the DMA controller (figure 7) was influenced greatly by the structure of the peripherals outside the AMULET3i subsystem. The peripherals are synchronous, dumb (they present/take register values to/from the synchronous bus but have very simple control) and may each be mapped to a number of addresses from which data should be transferred on each DMA request. This is a special requirement in the telecommunications device – which has heavy DMA traffic – and results in the controller containing an unusually large number of transfer channels (32) many of which are used entirely to perform clusters of transfers from the same peripheral. To reduce the physical size of the registers required to hold all this control, count

Figure 7: DMA controller structure

and address information the channels are partitioned into two types: 'long' channels with full 32-bit addresses and count registers and 'short' channels with 16-bit registers.

The DMA controller was built using full custom design for the regular blocks – particularly the register files – with the control logic being synthesized. The use of the asynchronous circuit synthesis language Balsa [12] in the description of the non-regular parts of the controller allowed the design to be re-engineered as the customer's demands changed with the minimum of designer effort and time. During the implementation of the custom-made parts of the controller the specification for the DMA controller underwent significant change at least twice. Each time large sections of the 900 line Balsa functional description changed without incurring large delays in the implementation of the controller.

The completed DMA controller consists of four main parts: the MARBLE bus interfaces, the synchronous peripheral interface (SPI), a large block of standard cells and a number of small, custom-made register blocks (logically part of the DMA registers unit in figure 7).

7.1. DMA/MARBLE Interfaces

The DMA controller presents two interfaces to MARBLE: a target interface through which the controller is programmed and interrogated by the processor and an initiator interface through which the controller performs its peripheral data and memory transfers. The data being transferred between devices is held in the MARBLE initiator interface while in transit giving 'store and forward' DMA operation. For each DMA transfer the transfer engine must perform two bus transactions, a read followed by a write.

7.2. The DMA Synchronous Peripheral Interface

In a completely asynchronous environment arbiters are needed to select between unsynchronised incoming DMA requests. In AMULET3i the peripherals are synchronous and so provide clock-synchronised DMA requests. Where requests arrive synchronised to a clock, arbitration is not only unnecessary, it is unwise; the likelihood of an arbiter signal becoming metastable requiring a possible lengthy resolution is increased if all the input signals are presented shortly after synchronisation. For this reason it was decided to implement the SPI using synchronous techniques.

The SPI controls the mapping of 16 incoming synchronous peripheral requests onto the DMA controller channels and the filtering out of requests for disabled channels. For this reason the channel enable and request number to channel number mappings for all channels are stored in the SPI along with a global 'fake request' register which allows the SPI to be tested by introducing software generated requests at the front of the request to channel mapping block. The SPI's registers can be programmed by the asynchronous control across the bidirectional bundle shown in figure 7.

The SPI contains not only the channel mapping hardware but also request state machines for each channel. These state machines allow incoming requests to be latched, for the asynchronous control in the DMA registers unit to be able to set and reset requests (setting requests is used to trigger channels which are free running, resetting to clear requests between transfers). The state machines effectively act as modulo-three saturating counters which are incremented each time a new request is received and decremented by the DMA registers control each time a new transfer is issued. This counter allows a new DMA request to be issued by a peripheral as soon as a previous request has begun to be acted upon. In this way a peripheral which is read by a DMA transfer need not wait for the whole transfer to be completed before requesting a second transfer and, combined with the essentially synchronous nature of the SPI, allows the use of DMA requests without explicit acknowledgements (the memory access to the requesting peripheral can be used as an implicit acknowledgement) simplifying the design of the synchronous peripherals.

The processed requests from the request state machines are bundled together and presented to the transfer engine as a single word on which a static prioritisation scheme is applied. Only 22 of the 32 DMA channels have request-to-channel mappings present in the SPI. The remaining 10 channels provide 'chain only' transfer functionality, transfers on these being initiated by a completed transfer on another channel. The SPI is implemented as a block of 22 stripes (one per request capable channel) of both custom and standard cells and occupies around 15% of the total area of the DMA controller.

7.3. The DMA Standard Cell Blocks

The standard cell block dominates the DMA controller occupying nearly 50% of its total area. This block contains the initial decoding and control for the register blocks, some of the MARBLE interfacing glue, the SPI top level control, a request prioritisation system based on a synchronously presented vector of channel requests from the SPI, the DMA registers asynchronous control and the transfer engine. The DMA registers' control and the transfer engine make up the largest part of the standard cell block as they are made from compiled Balsa descriptions of the overall DMA controller's main operations: the coordination of DMA registers access, register value incrementers, end-of-transfer test logic for transfer addresses and the transfer engine which actually initiates DMA transfers on the bus. The transfer engine is responsible for reacting to channel requests received from the SPI, requesting register values from the DMA registers' control for transfers and controlling the initiator interface through which transfers are performed.

7.4. The DMA Register Blocks

The DMA register blocks contain channel address, count and control data totalling around 2000 bits. They can be accessed via the MARBLE target interface for programming purposes and by the transfer engine by arbitrated access through the asynchronous standard cell control block coded in Balsa. The blocks were constructed from full custom register and decoder cells in a similar manner to the register bank in the processor core. Each register block corresponds to a particular channel register type and is indexed by channel number. They each provide a single read/write interface to the standard cell block allowing all the registers for a single channel to be read or written in one operation. A typical transfer operation would access a channel's registers twice, once to read addresses and count values for a channel and a second time to update the registers with those incremented address and count values.

There are 7 separate register blocks in the DMA controller. These can be seen in figure 10 as the 3 slim blocks on the left of the DMA controller, the 3 larger blocks at the bottom of it and the similarly sized block to the right of the controller. Each block corresponds to different set of registers for those channels which include that register type. The slim blocks contain the upper 16 bits of the source/destination addresses and count registers for the 4 'long' (full memory range). The larger 3 blocks contain the lower 16 bits of the addresses and the common control register bits. The remaining block contains the count registers and remaining control bits for the 22 channels which are capably of receiving requests from the SPI.

7.5. Designing with Balsa

Balsa was used to implement the register bank control, transfer engine and initiator interface. Each block was described by a single Balsa process communicating with its neighbours using handshake channels. For example, the main loop in the transfer engine is:

```
loop
    ChannelReq := {0, false} || CountEqZero := false;
    – read DMA request vector
        select PReq then
        – priority encode, chan 0 has highest priority
        if PReq[0] then ChannelReq := {0, true}
        else if PReq[1] then ChannelReq := {1, true}
        else if PReq[2] then ChannelReq := {2, true}
        …
    if ChannelReq.dotfr then
        PerformTransfer ();
        – while we are asked to chain
        while RegReadData.usechain
            and RegReadData.genable then
        ChannelReq := {RegReadData.nextchan, true};
        PerformTransfer ()
        end end – while and if
    end – select
end – loop
```

The initialisation of the variables ChannelReq and CountEqZero can be seen in the above example. They are followed by a *select PReq then … end* statement within which values on the channel PReq (Peripheral Request) are visible. The PReq channel is the channel on which the SPI request vector is received by the transfer engine. The cascading *if* statement involving the value on PReq is used as a priority encoder on incoming requests with the channel number of the chosen request ending up in ChannelReq.

The transfer is performed (or rather communicated to the initiator interface) by the sub-process PerformTransfer. The *while* loop is used to perform the tail transfers in a chain of channels. It is easy to see that chains are composed of 'linked lists' of channels from the way that they are processed.

The initiator interface portion of the transfer engine is the simplest of the Balsa blocks as it simply performs transfers on behalf of the transfer engine. The complete definition of the initiator interface is:

```
procedure DMA_II (
    output II_Addr : MARBLEAddr;
    input DI : IIData;
    – inform control of end of run
    output EndOfRun : ChannelNo
```

```
) is local variable RegReadData : IIData
begin loop
    DI → RegReadData;
    – read from Source Device
    II_Addr ← {RegReadData.src, Read,
       RegReadData.size};
    – write to Destination Device
    II_Addr ← {RegReadData.dst, Write,
        RegReadData.size};
    if RegReadData.endofrun then
       EndOfRun ← RegReadData.channelno
    end
end end – loop and procedure
```

The channel interface that the process (DMA_II) presents to other processes can be clearly identified along with the locally defined variable RegReadData and the loop of 4 operations. The EndOfRun indication is attached to the register bank arbitration as a third input (register requests for the transfer engine and accesses from the target interface being the other two) and is used to signal the end of a run of transfers by the channel transfer count becoming zero.

The AMULET3i DMA controller is the first substantial design to be attempted with Balsa. It is a block with modest performance requirements constructed in the midst of a changing specification without occupying designer time which could be more constructively spent working on the processor core and memory system. The use of a synthesis system to meet this need has turned the DMA controller from a potential millstone into a promising example of an asynchronous design approach applicable to peripheral construction.

The required performance of the controller is limited by the speed of the MARBLE/Synchronous bus Bridge (MSB). The delivered controller can perform a single transfer in approximately 90 ns (simulated), which is sufficient to saturate this bridge.

8. External Memory and Test Interfaces

The External Memory Interface (EMI) is based around the design which was used successfully in AMULET2e [7]. This has been designed to allow direct access to a range of standard memory devices without the need for any glue logic. The pin level interface consists of an address bus, a bidirectional data bus and a range of programmable control signals to provide strobes and enables to external memory devices. The supported devices include static RAM and ROM devices and also conventional DRAM. There is no support for synchronous DRAM devices and, unlike AMULET2e, there is no mechanism for hardware refresh of the DRAM. The EMI contains a simple write buffer which allows up to five data items to be buffered before they are written to external memory. Coherency is maintained by delaying read operations until the write buffer is empty.

As AMULET3i is targeted at embedded applications the EMI provides a simple memory map which allows a relatively small number of memory and peripheral devices to be connected. The processor's 32-bit address space is divided into eight 512Mbyte regions. Timing and other characteristics (such as databus width) for the memory devices associated with each region are controlled by configuration registers which are set under program control. With the exception of the bottom two megabytes of address space (reserved for on-chip RAM and ROM) and the top 16 megabytes (used for on-chip peripherals and configuration registers) the whole of the address space is available to external devices.

The external data bus supports memory devices of 32, 16 and 8 bits and the EMI will automatically pack and unpack data from the processor so that it is the correct width for the external memory. In the first application of AMULET3i the requirement is for a system with most of the peripherals on chip and only memory devices off chip. Because of pin count limitations in this application a 16-bit external data bus and 20-bit address bus has been implemented and only 4 of the 8 possible regions may be used.

8.1. EMI Timing

In order to control the timing of the EMI a calibrated timing reference is required. In AMULET2e this was provided by allocating two pins on the device and sending a transition out on one pin to be delayed by an external reference delay and then fed back to the other pin to indicate that a certain time period had passed. This basic delay can be called a number of times to wait for longer periods. This is a reasonable scheme as it minimises power consumption but is somewhat imprecise and, in practice, there was some difficulty in finding suitable reference delay devices. It is impractical to use a crystal oscillator in this sort of application as such circuits require a substantial time to start up and – if left running continuously – require the interface to be at least partially synchronous, thus negating some of the advantages gained from the device's asynchronous nature – notably the electromagnetic emission characteristics.

AMULET3i addresses the timing reference problem by implementing the delay on chip. This introduces a calibration problem in that the absolute delay of a chain of gates will vary according to a number of factors, including (for example) the manufacturing conditions and the operating temperature of the device. To combat this a delay line built from a chain of simple gates is used, the length of which can be controlled by the processor (figure 8). A latch which may be written by the processor drives the control signals
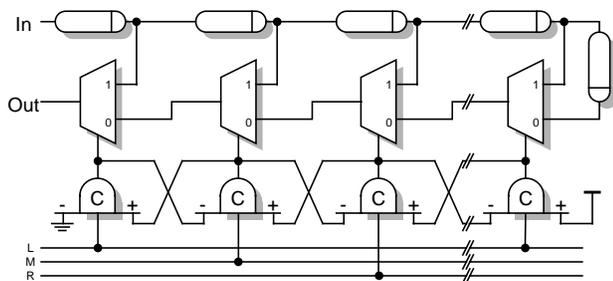
Figure 8: Controllable delay circuit

L, M and R. Setting these three signals high causes the outputs of the C gates to go high and the delay line then presents its minimum delay. Setting the control signals low drives the C gates low and the delay is them maximised. From either of these states writing a sequence of walking zeros or ones onto L, M and R allows the length of the delay to be incremented or decremented one stage at a a time. The delay line has 54 stages each of which has a delay of approximately 250 ps (typical silicon). Thus the basic delay can range up to 31.5 ns. To extend the range of the delay line a prescaler can multiply the delay by 1, 2, 4 or 8 to give the basic timing unit (*Tref*) which is used by the EMI. Timing parameters for memory devices in each region are given in terms of this timing unit. For example, the write strobe width for a static RAM can be specified to be 1, 2, 3 or 4 *Tref*. The logic which implements the *Tref* timing imposes various overheads so that the minimum *Tref* which can be specified is around 5.0 ns and the maximum around 120 ns. There are testability issues arising from this circuit and additional hardware is incorporated to address these problems.

Keeping the delay line internal reduces electromagnetic emissions and having it adjustable in software allows more flexibility than was afforded with AMULET2e. To support this the output of the delay line can be routed to a 16-bit binary counter which is readable by the processor. The delay line can be configured to free run so the counter can then be used in conjunction with a reliable timing reference (such as a crystal) to calibrate it. Variations in operating conditions of the device will have an effect on the timing the delay line and so any calibration software will have to be written with this in mind, although effects such as changes in temperature happen relatively slowly and should be easy to compensate for.

8.2. Test interface port

Because AMULET3i is a relatively complex device, production testing is a considerable task. The fact that all of the major units on the device are connected via the MAR-BLE bus affords an opportunity to rationalise the testing

procedure. This is achieved by having a test mode in which the pins allocated to the EMI are used to create a test port. The data bus and 16 bits of the address bus are used to form a 32-bit bidirectional test port and three other address bits are used to specify a test function. The remaining address input forms a test clock input which is used to initiate various test functions. An asynchronous interface would be inappropriate here as current chip testers are not able to interface in this manner. Instead a clock is used and the tester run at an appropriate rate.

The test port connects internally to a MARBLE initiator which allows it to have read/write access to all MARBLE peripherals. This allows many functions of these devices to be tested directly. Of particular interest is the fact that the on-chip RAM may be loaded via the test interface. When in test mode the normal bootstrap mechanism is ignored and, following reset, the processor fetches instructions from the on-chip RAM. This allows very powerful self-test mechanisms whereby a test program of up to 8 Kbytes is loaded via the test port and the processor is then allowed to execute that program. The processor is independent from the tester (including clocking!) at this point and will run at full speed. This greatly reduces tester time for testing on-chip devices (particularly the RAM and ROM) and allows much more complex test algorithms to be contemplated than would have been possible with just a basic interface to the tester. Even more time may be saved by filling any unused parts of the ROM with test routines which could be called from a short, downloaded program.

Most parts of the system are accessible and relatively easy to test. However the CAM-RAM structure required for the Branch Target Buffer (BTB) is particularly hard to test in normal operation. Access for test is provided by two additional test buses and a number of test registers. This makes the BTB accessible to the test program, but isolated from the remainder of the prefetch unit which will be required for the execution of the test program. The design is such that tests can be conducted whilst the processor is running (albeit with branch prediction disabled) so that this can be included in self-test software.

9. On-chip debug support

The AMULET3i subsystem includes breakpoint and watchpoint hardware which can cause an interrupt or abort to be generated whenever a particular instruction or data access occurs. The debug registers specify values for the address, data and control signals on each of the AMULET3 instruction memory and data memory interfaces and also specify mask values which cause any subset of these signals to be ignored in the matching logic. This allows, for example, ranges of addresses and particular data fields to be tested.

The debug registers are implemented as serial scan chains, one for the instruction memory interface and one for the data memory interface. They are loaded by software via 8-bit control registers: the BreakPoint Control register (BPC) and the WatchPoint Control register (WPC). Each debug chain is organized in three sections, as illustrated in figure 9.

Once the appropriate scan chain has been loaded and enabled, execution of application code can continue unimpeded until the interface state matches (under its mask) that loaded into the scan chain. When a match is detected either an (imprecise) interrupt or a (precise) memory fault can be generated. In either case control is returned to the system software and the processor and system state can be examined by software.

This does not give the debug capability of the EmbeddedICE macrocell used by clocked ARM cores [13] which is fully independent of any resident code, but it provides a basic debug facility at low cost; this form of debugging has been used on DRACO's (synchronous) predecessors with positive results. The AMULET3 processor core has been designed to include the full ARM debug functionality and it is possible that this will be included in a future device.

10. Performance figures

The current AMULET3i macrocell occupies approximately one half of the DRACO chip (figure 10) which is implemented in a VLSI Technology Inc. 0.35 μm 3-layer metal ASIC process; it measures about 7.0 x 3.5 mm. The transistor counts of the major blocks are:

- AMULET3 –   113 000
- RAM (total) –   504 000
- DMA controller –   70 000
- EMI –   26 000

These, together with the smaller logic elements, give a total of about 800 000 transistors in the asynchronous subsystem; the synchronous peripherals contribute about another million transistors.
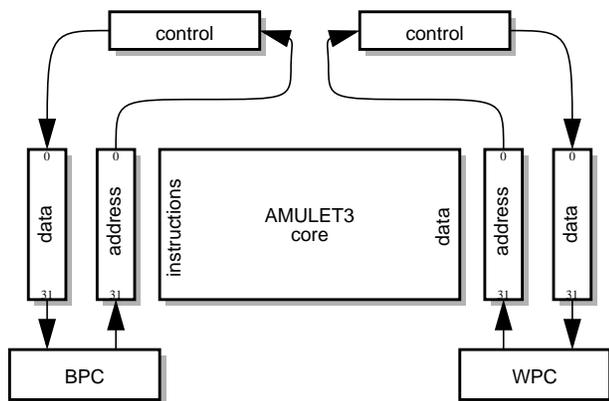

Figure 9: Debug hardware structure

Extensive simulation has been employed on extracted layout. The simulator most used was TimeMill from EPIC Design Technology, Inc. which was used to simulate subsystems as they were commissioned and is able to simulate extracted layout for the whole AMULET3i system. All the simulations in the following sections were performed assuming 'typical' silicon parameters at 3.3 V, 25°C.

10.1. Processor

Simulation suggests that the processor's execution path will consume most data processing instructions at about 120 MIPS. There is some variation in this due to instruction classes, so that MOV instructions are somewhat faster and ADD instructions a little slower. Including a series shift with these instructions stretches the cycle by around 35%. Multiplication, which uses 32-bit operands and can return a 64-bit result, takes about 30 ns although an optional accumulation can be included with no time penalty. Data dependencies between instructions are resolved by forwarding from the reorder buffer and – apparently – incur no additional cost.

Branches execute at a similar rate, although they must interrupt the (asynchronous) prefetch unit so there is some inherent uncertainty in their timings. Branch prediction alleviates some of this (although, unfortunately, not significantly in the Dhrystone benchmark).

The longest processor stalls are caused by external data references. This is due to the longer latency and limited bandwidth available on the local data bus and is described below.

10.2. Local Bus

The quad word instruction and data buffers contained in the on-chip RAM have a significant impact on the performance of instruction and data accesses across the local bus. An instruction access that 'hits' the instruction buffer can be returned in around 9.5 ns, whereas an access that 'misses' will take 12 ns (with some variation as physically distant blocks are fractionally slower than nearby ones due to the length of the handshake wires). This averages to about 95 Mwords/s although it will be faster in code with much locality and slower if many jumps are taken.

For data reads both the hit and miss cycle are slower (13 ns and 16 ns respectively) due to the increased complexity of the local data bus. In the case of single load operations the added latency is the most important consideration; this imposes about 8.75 ns/10.5 ns depending on the buffer hit/miss status. In block transfers bandwidth is more important; a typical LDM (LoaD Multiple) instruction can achieve around 70-75 Mwords/s, STM (STore Multiple) is 3-4 Mwords/s slower. The asynchro-
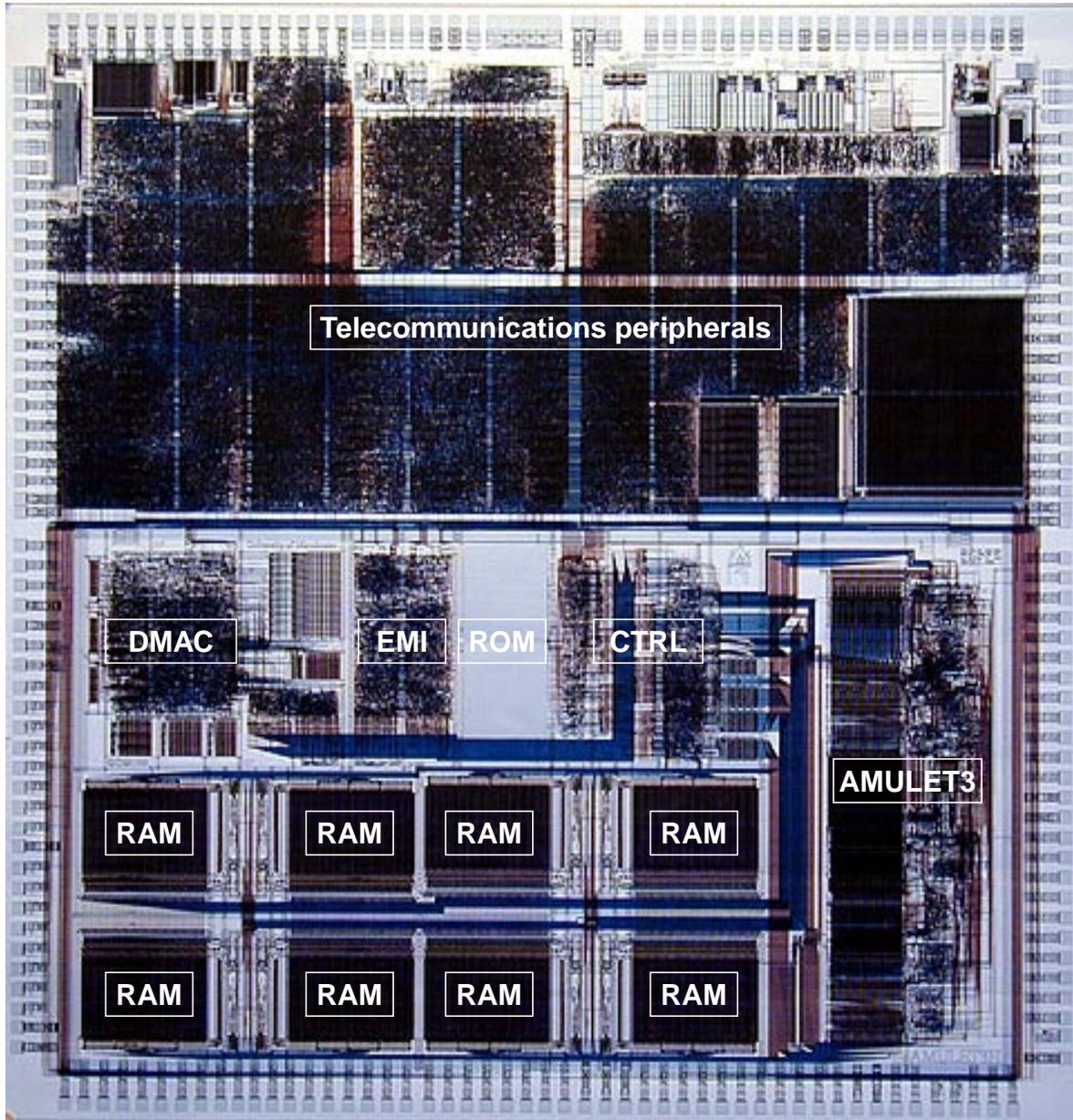
Figure 10: DRACO layout

nous operation of the local buses allows these performance variations to be exploited by the typically faster AMULET3 processor.

10.3. MARBLE

Simulation results show MARBLE cycling at up to 85 MHz when more than one initiator is active. Saturation cannot be achieved by one initiator alone because of the single-outstanding command constraint, the implementation of which means that there is a delay (for arbitration) between the start of a response cycle and the start of the subsequent command cycle. Even so, the maximum throughput for one initiator has been measured as 55 MHz (CPU data-port performing a series of ROM accesses), including the access time for the ROM of 6 ns. Read latency starting from an empty bus is 15 ns (or 13 ns if arbitration is commenced early) plus the access time of the target peripheral.

Bus arbitration takes approximately 0.5 ns per stage of the arbiter tree (two stages for command channel arbitration, three stages for data channel arbitration).

10.4. System

To assess the performance of the whole system some metric which averages out many of the interactions and dependencies is required. The Dhrystone 2.1 benchmark has been used because it is a familiar standard which allows comparison with other devices. This does not measure MARBLE performance, but it does exhibit a 'typical' instruction mix and takes into account interactions between instructions and the instruction and data local bus speeds. This program can also be compiled for both the ARM and Thumb instruction sets so the performance of these can be compared. It is already known that Dhrystone exhibits atypical branch behaviour and so the BTB gives less benefit than in code which iterates more.

The system achieves about 176 kDhrystones/s – which translates to 100 Dhrystone MIPS – when running code compiled into the ARM instruction set. During this process the system power averages 215 mW within the AMULET3i system (of which 130 mW is within the processor core); this yields about 465 MIPS/W for the system (or 780 MIPS/W for the processor alone).

Running Thumb code is somewhat slower as there are more instructions to execute. This gives around 125 kDhrystones/s (71 MIPS), which is around 30% slower than the ARM code. This is according to expectation; ARM Ltd. estimate that the Thumb instruction overhead as 40% but the AMULET3 processor is not throttled by instruction fetch bandwidth (although the pipeline/prefetch depth is increased).

For comparison an ARM9 using the same instruction set and manufactured on the same process may be clocked at up to 120 MHz, at which speed it yields 133 Dhrystone MIPS. Although this number will be derated to account for elevated temperature (and, possibly, voltage fluctuations) the comparison is fair because AMULET3i will accommodate such fluctuations; the typical operating conditions of ARM-based products are those of devices such as cellular telephones, which do not operate at 70°C! The ARM9 processor core achieves 800 MIPS/W. Both processor cores occupy just over 3mm^2 on the same 0.35μm process; it is believed that AMULET3 is slightly smaller but an exact like-for-like comparison is difficult.

This compiler used has not been optimised for AMULET3 although it has been adjusted to allow for a 5-stage instruction pipeline. Using an earlier version of the compiler intended for a 3-stage pipeline gives a system performance about 2% lower; it is believed that this is chiefly due to the increased load latency allowance.

In comparison with other asynchronous microprocessors, AMULET2e achieved about 42 Dhrystone MIPS and TITAC-2 [14] about 52 Dhrystone MIPS. The Caltech asynchronous MIPS R3000 [15] was reported as achieving 170 native MIPS but we are unsure how this translates into Dhrystone MIPS. Note that all these processors were built on different (older) 0.5 μm or 0.6 μm technologies.

The peak MIPS numbers for AMULET3i are still somewhat uncertain. However it has been observed to run ARM instructions at a rate of around 105 MIPS (limited by the prefetch bandwidth) and Thumb instructions at 110 MIPS (limited by the Thumb expansion stage, which slows down when instruction translation is required).

11. Summary of novel features

AMULET3i is wholly asynchronous. This means that its different subsystems run at different rates. In the processor pipeline this simply means the traffic flow is speed limited by the slowest stage (and the pipeline elasticity), but outside the simple pipeline structure traffic is passing and overtaking according to less rigorous constraints.

The speeds of the buses are quite markedly different; the local RAM instruction bus has the highest required bandwidth and is the fastest of the buses. Trade-offs were made which reduced the speed of local data accesses and the bandwidth on this bus is about 25% lower. This is not relevant to the functionality of the system, although it would probably have forced an overall speed reduction in a synchronous environment. The speed of local RAM accesses also varies by about 20% due to the limited 'cacheing' implemented. The access speed drops further if MARBLE is used.

Another vaunted benefit of asynchronous systems is their modularity. Within AMULET3i several modules (processor module, ROM, DMA controller, EMI, etc.) were designed independently and 'bolted on' to MARBLE; each has its own design style and performance criteria.

The entire design has been produced with the intent of minimising power consumption. This includes many detailed features: for example tristate enables onto shared buses are non-overlapping, a simple and familiar example of self-timing. However there are also a few novel mechanisms which have been introduced in order to reduce unnecessary bus activity.

Because of its asynchronous nature the pipeline cannot discard erroneously prefetched instructions en masse. Instead it uses a 'colouring' mechanism [16] as in earlier AMULET processors. The discard takes place at the execution stage, which means that instructions must flow to this point first, in the process being decoded and acquiring their register operands. To alleviate much of the associated power wastage AMULET3 counterflows the 'branch colour' back to the decode stage (one stage earlier) and further discarding takes place here. This counterflow is arbitration

free and uses the local synchronisation between stages, which means that it 'leapfrogs' the instruction following the branch, but is effective after that. Simulation reveals that this has no great speed benefit – other pipeline stages still limit performance – but it removes typically one and sometimes two instructions before their register values are read, with a consequent power reduction. However the real significance is apparent when running Thumb code, which effectively doubles the prefetch buffer size, typically discarding at least three instructions early.

Other notable features to reduce bus activity act at a more global level. The halt mechanism and the division of RAM into sub-blocks have been employed before, including in AMULET2e; the predicted branch fetch suppression described in section 3 is new to this design and should prove of significant power (and some performance) benefit.

Lastly, whilst not a specific asynchronous advantage, the inclusion of a flexible DMA controller provides a major efficiency gain when moving data amongst peripherals. By removing the need for the processor to perform many simple data transfers the inclusion of this unit should provide a significant power reduction at system level.

12. Conclusions

AMULET3i is one of the most complex asynchronous devices yet produced. It demonstrates that an asynchronous implementation can compete directly with a synchronous implementation of the same instruction sets in performance, area and power consumption. Although AMULET3 does not demonstrably beat ARM9 in performance or power consumption it must be remembered that the development effort and experience available within a university is significantly less than in a world leading microprocessor design company. The asynchronous implementation also has unique advantages in terms of power management – especially in embedded systems – and, judging from previous experience with AMULET2 [8], electromagnetic emission.

As the basis of a commercial device the issue of production test has been addressed; indeed it may be that self-timed self-test can even reduce the time-on-tester in production as a single attempt at running the code will indicate if the device is functional and give a speed measurement.

Modularity has often been cited as an advantage of asynchronous circuits. Within this design this has been exploited to include units which have used different design processes and styles. It is intended to exploit this further in the future to allow incremental change to the system and the development of more functional modules. AMULET3i is the first commercial asynchronous ARM; it is not intended to be the last.

14. References

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

[2] Garside, J.D., Furber, S.B., Chung, S-H. "AMULET3 Revealed" Proc. Async '99, Barcelona, April 1999 pp. 51-59.

[3] Jaggar, D., "Advanced RISC Machines Architecture Reference Manual", Prentice Hall, 1996. ISBN 0-13-736299-4

[4] Segars, S., "The ARM9 Family - High Performance Microprocessors for Embedded Applications", Proc. ICCD'98, Austin, October 1998, pp. 230-235.

[5] Segars, S., Clarke and Goudge, "Embedded Control Problems, Thumb, and the ARM7TDMI", IEEE Micro, 15 (5), October 1995, pp. 22-30.

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

[7] Furber, S.B., Garside, J.D., Temple, S., Liu, J., Day, P. and Paver, N.C., "AMULET2e: An Asynchronous Embedded Controller", Proc. Async'97, Eindhoven, April 1997, pp. 290-299.

[8] Furber, S.B., Garside, J.D., Riocreux, P., Temple, S., Day, P. Liu, J., Paver, N.C. "AMULET2e: An Asynchronous Embedded Controller" Proceedings of the IEEE, volume 87, number 2 (February 1999), pp. 243-256 ISSN 0018-9219

[9] York, R. "Branch Prediction Strategies for Low Power Microprocessor Design" M.Sc. Thesis, University of Manchester 1994

[10] Chung, S-H., "The Design of a Branch Target Cache for an Asynchronous Microprocessor", MPhil Thesis, University of Manchester, 1998 (http://www.cs.man.ac.uk/amulet/publications/thesis/chung98_mphil.html)

[11] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A. "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers" IEICE Transactions on Information and Systems, Vol. E80-D, No. 3, March 1997, pp. 315-325.

[12] Bardsley, A. "Balsa: An Asynchronous Circuit Synthesis System", MPhil Thesis, University of Manchester, 1998 (http://www.cs.man.ac.uk/amulet/publications/thesis/bardsley98_mphil.html)

[13] Furber, S.B., "ARM System Architecture", Addison Wesley Longman, 1996. ISBN 0-201-40352-8.

[14] Takamura, A.,Kuwako M., Imai, M., Fujii, T., Ozaw, M., Fukasaku, I., Ueno, Y. Nanya, T. "TITAC-2: An Asynchronous 32-Bit Microprocessor Based on Scalable-Delay Insensitive Model" Proc. ICCD'97, 288-294, October 1997.

[15] Martin, A.J., Lines, A., Manohar, R., Nystrom, M., Penzes, P., Southworth, R., Cummings, U., Lee, T.K. "The Design of an Asynchronous MIPS R3000 Microprocessor" Proc. 17th Conf. on Advanced Research in VLSI, September 1997.

[16] Paver, N.C., "The Design and Implementation of an Asynchronous Microprocessor", PhD Thesis, University of Manchester, June 1994.