SPA - A Synthesisable Amulet Core for Smartcard Applications

L.A. Plana, P.A. Riocreux, W.J. Bainbridge, A. Bardsley, J.D. Garside, S. Temple
Dept. of Computer Science, The University of Manchester, Oxford Road, Manchester, M13 9PL, UK.

{lplana,priocreu,wbainbridge,bardsley,jgarside,stemple}@cs.man.ac.uk

Abstract

SPA is a synthesised, self-timed, ARM-compatible processor core. The use of synthesis was mandated by a need for rapid implementation. This has proved to be very effective, albeit with increased cost in terms of area and performance compared with earlier non-synthesised processors. SPA is employed in an experimental smartcard chip which is being designed to evaluate the applicability of self-timed logic in security-sensitive devices. The Balsa synthesis system is used to generate dual-rail logic with some enhancements to improve security against non-invasive attacks. A complete system-on-chip is being synthesised with a only small amount of hand design being employed to boost the throughput of the on-chip interconnection system.

1. Introduction

The Amulet group has developed a series of ARM-compatible microprocessor cores over the last ten years [1, 2, 3], all of which have followed the design style used in the early, synchronous ARM hard macrocells - full custom datapaths with standard cell control blocks [4].

In our latest collaborative project we are developing a self-timed smartcard chip with a specific focus on the ability of self-timed circuits to offer improved security through their resistance to non-invasive attacks and, in particular, attacks based on power analysis and electromagnetic analysis. At the outset we planned to re-use an existing Amulet core in this design but power analysis investigations into our existing chips showed that their security was poor, mainly as a result of their use of bundled-data communication. We needed a new processor core, based on a more secure self-timed technology, and there was insufficient time to design a new core in the (rather laborious) full custom style used previously.

The only satisfactory path open to us was to use synthesis to develop the new processor core. The Philips Tangram system [5] (from which Balsa takes its inspiration) has been used in a similar way to implement a clone of the 80C51 microcontroller [6] for use in pagers.

Balsa [7, 8] has proved its value in the DRACO chip [9], where it was used to synthesise the 32-channel DMA controller [10], but clearly a processor core is a significant step up in complexity. However, this presented an opportunity to drive Balsa forward and to demonstrate its capabilities in a new, challenging area. This paper describes the architecture of the new processor and presents some preliminary results.

In section 2 the general system architecture of the smartcard chip is presented; this is the context in which the processor must operate. The specific requirements for the processor are discussed in section 3. The design methodology is described in section 4, and the processor itself in section 5. An adjunct to the processor is the Memory Protection Unit (MPU) described in section 6, and the results and conclusions from the work are presented in sections 7 and 8.

2. System Architecture

A block diagram of the smartcard chip appears in figure 1. The smartcard chip is a small embedded system with the SPA processor at its heart. It has access to memory on the card which provides program and data storage and it communicates with a smartcard reader via a UART. The SPA CPU and the MPU are described in detail later in the paper; the other components are outlined below.

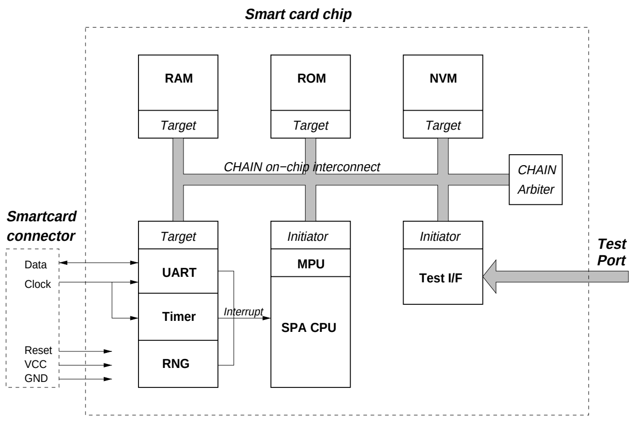

Figure 1: Smartcard system architecture

2.1. CHAIN self-timed on-chip interconnect

The various components on the smartcard chip are connected using the CHAIN on-chip interconnection system [11]. This is a development of the MARBLE system used in Amulet3i [12] and, like MARBLE, it is a multi-master system. Devices are connected to the bus with either initiator or target interfaces and the system provides arbitration between competing initiators. Having arbitrated for access to the bus, an initiator can then read or write a device connected to a target. The targets and initiators present a conventional address and data bus interface to their connected devices. CHAIN itself, however, is implemented with serial interconnect using a delay insensitive 1-of-4 code. This results in the signal paths of CHAIN transitioning at rates in excess of 1GHz which will make power and emission analysis of this part of the system very difficult for attackers.

The smartcard chip has two initiators, one for SPA and one for the test interface. All of the other blocks connect to CHAIN as targets. Because of a desire to keep the performance of CHAIN as high as possible, for security as well as throughput, it has been designed by hand throughout and is the only significant piece of hand design on the chip.

2.2. Smartcard UART and other peripherals

In common with the majority of smartcards the interface with the smartcard reader is by means of a UART. This is implemented in Balsa from the published specification [13]. The interface consists of a clock which is provided by the reader and a bidirectional data line. These signals are connected to the UART which is connected to CHAIN via a target interface. This target also hosts two other peripherals - a timer and a random number generator. The timer is incremented by the clock from the reader and is used to maintain a source of real-time information for the system. The random number generator supplies pseudo-random numbers using a feedback shift register. In a production smartcard a source of true randomness would be required.

2.3. Memory

A smartcard generally requires two or three types of memory. RAM is required for working storage and, in the system described here, 16KB of RAM are provided organised as 4K 32-bit words. The RAM is of a conventional design and produced using a memory generator program. This presents a synchronous interface and a simple self-timed wrapper is then used to present a dual-rail interface to the rest of the system. The reading and writing of data to the RAM can be a major security hole in a smartcard because of the relatively high currents which are involved. It is common to encrypt data stored in the RAM to overcome this but this is not attempted here.

In addition to RAM, some sort of non-volatile memory is required to hold programs and to maintain data when the card is unpowered. Traditionally ROM was used for program storage and EEPROM for persistent data. Recently some cards have dispensed with ROM and used EEPROM or Flash for both purposes. In the interest of reducing risk in a prototype design, we are providing both ROM (32KB) and Flash (2 blocks of 64KB). Of the limited number of fabrication processes available to us, none provided EEPROM so we have used Flash in this design. Having two blocks of Flash memory allows code to run from one block while the other is being programmed. Like the RAM, these blocks are automatically generated and have self-timed wrappers. The ROM will contain a bootstrap loader program to allow the card to be programmed via the UART.

2.4. Test interface

A test interface which is similar to the one which was successfully employed on Amulet3i [9] is implemented here. The provision of a test interface on a chip designed for security is something which must be treated very carefully in a production design as it represents a major security hole. In this experimental context, however, it is desirable from both a test and a debugging point of view. The test interface consists of a 32-bit bidirectional bus, a 3-bit control bus and a clock input. The external test interface is synchronous as this style is best suited to current chip test machinery. The basic operations provided by the test interface are reading and writing at arbitrary addresses in the on-chip address space. Internally, the test interface connects to the CHAIN interconnect via an initiator interface. Test code can be downloaded into on-chip memory and subsequently executed by the SPA processor and this mechanism forms the basis for the post-manufacture testing.

3. Processor Requirements

The performance requirements for a smartcard application are, fortunately, not too demanding. A performance in the 10 to 20 MIPS range is adequate for many applications, and is certainly sufficient to address the research goals of this work - to demonstrate the security benefits of asynchronous operation.

Amulet3 [14] is a 100 MIPS core on a 0.35µm CMOS technology. It was clearly not going to be possible to get close to this performance with a core synthesised using a tool that had never been applied to a 32-bit processor before and which contained no optimisations for datapaths. However, we plan to use a more advanced process technology (0.18µm) which would give a factor of two performance improvement, so we could afford to lose an order of magnitude of performance in the synthesis process. This seemed achievable even with a synthesis tool using a relatively immature technology description. It also allows us to work with a relatively simple microarchitecture, using a straightforward 3-stage pipeline as described in section 5.

The second requirement was that the processor should offer full 32-bit ARM compatibility [15]. This is a challenge to the designer, but one well understood by the team.

The third requirement, most important to the research goals, was to minimise side-channel information leakage.

At the lowest levels, side-channel information leakage can be minimised by balancing circuits. A dual-rail circuit has a symmetry between ‘0’ and ‘1’ actions that reveals little through its power consumption and electromagnetic emissions. Even so, a standard dual-rail register is unsafe because, though it is symmetric with respect to its two values, it will still reveal whether a new value is the same as or different from its predecessor. Reconstructing data streams from transition information is little harder than doing so from level information. The solution here is to propagate ‘spacers’ through registers, removing all traces of the old value before the new one is stored. These and similar techniques were used to develop a new ‘secure’ Balsa back-end.

The back-end takes care of low-level issues, but at the higher level of design it is still necessary to ensure that gross activity is as constant as possible. This has been an interesting culture change for the group, since almost everything we had learnt in the past about low-power design [16] had to be unlearnt. Any technique that saves power by dynamically eliminating unnecessary activity leaks information in the power signature! The objective, therefore, is to develop a processor that does pretty much the same all of the time and, in particular, has data-independent activity characteristics.

4. Design Methodology

SPA is intended to be a secure implementation of the ARM v5 instruction set architecture (ISA). Security here is security from non-invasive attack by observing information leakage through variations in power supply current or electromagnetic radiation [17]. This form of attack relies on two features of the circuit under attack:

- a timing reference with which to synchronise repeated observations of circuit behaviour
- the presence of information-bearing changes in power consumption by the circuit.

It is anticipated that the use of asynchronously communicating modules will make correlating behaviour from cycle to cycle more difficult. The use of 1-of-n codes (dual-rail, 1-of-4) should reduce variations in energy dissipation due to differences of Hamming weights between data values. Four problems associated with this simple view present themselves.

- Symmetry in processing
- Persistent storage
- Timing information leakage
- Keeping some self-timed advantages

4.1. Symmetry in processing

Although all data values in a 1-of-n code have the same weight, so making communications ‘balanced’, processing of 1-of-n data values may involve circuits in which there is no symmetry of logic between the bits making up each code group. Consider the DIMS implementation of a dual-rail 2-input AND gate shown in figure 2 [18]. The ‘1’output is

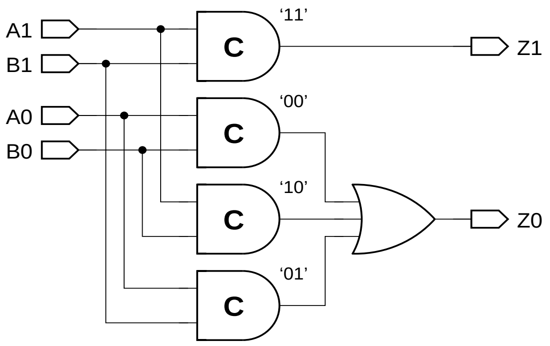

Figure 2: Dual-rail 2-input AND gate

asserted by the single minterm corresponding to both inputs being 1. All three other input combinations result in the ‘0’ output being asserted. It is obvious then that although the codes involved are balanced, operations on these codes need not have balanced logic.

Luckily, a number of useful operations (XOR gates, adders, inversions) can be constructed using balanced circuits. For other functions, pseudo-balanced versions can be constructed by placing dummy loads on chosen signals.

4.2. Persistent storage

Balancing processing and communications is possible using the above techniques. Balanced storage can also easily be constructed by storing 1-of-n coded data. However, it may still be possible to determine the difference in weight between a word loaded into a storage element and the word previously occupying that place. Conventional latches and flip-flops dissipate energy in proportion to the number of bits changed during a load. By explicitly resetting a piece of storage before loading it we can hide the differences between successive values albeit at the cost of slower, more expensive latches. ‘Secure’ latches can also be constructed by building a short Muller pipeline, using the return-to-zero phase to propagate a reset through the latch [19].

4.3. Timing information leakage

Self-timed circuits are often built to exhibit ‘average-case performance’. Circuit area can be traded off against worst-case performance to allow (frequent) average-case operations to be performed more quickly than (rarer) operation on worst-case data. Unfortunately, this desirable property can lead to information leakage. Even with balanced codes, variations in timing between operations may still be visible. These timing variations are well known to authors of encryption software for secure applications. The cycle time of software loops is the software analogue of this hardware timing problem. The software solution of making each loop take the same amount of time to execute can also be applied to the hardware. For example, a constant propagation delay adder can be made by forcing the carry ripple to propagate all the way up the carry chain. This can be achieved by constructing full-adders which wait for their full complement of inputs to arrive before asserting their outputs. An adder built this way will not operate more quickly in the case that one or more adder stages generate rather than propagate a carry. This principle can be applied to many circuits to make their propagation delays constant.

4.4. Keeping some self-timed advantages

Making all operations take the same amount of time effectively results in a synchronous implementation. We believe, however, that the careful use of asynchronous features in such an implementation can result in a circuit in which it is harder to make the cycle-to-cycle timing correlation required to perform power analysis. The designers must choose from a continuum of such implementations varying from those with fixed-duration operations to those with varying and overlapping operation delays (either data dependent but hard to extract or genuinely random). For SPA we take an approach somewhere in the middle of this continuum to attempt to test whether a processor with units with fixed but unequal latencies will result in a more secure implementation. Some features included to achieve this objective are:

- Different classes of instructions will still take different amounts of time to execute, quite independent of the data processed.
- The self-timed pipeline will make the energy dissipation signatures of different parts of the processor overlap in continuously varying ways.
- The CHAIN interconnect system will be operating concurrently with the processor and at a wildly different operating speed (as it is serial and implemented in a different circuit style).

The security features of the Balsa synthesised portions of this first SPA system therefore include:

- Use of dual-rail encoding throughout, requiring the introduction of a new Balsa back-end (Balsa had previously been entirely bundled-data based).
- Reduced choice in the Balsa description used to synthesise SPA leading to more monotonous operation. This choice reduction is explained, unit by unit, in section 5.
- Balanced implementation of the ALU and other processing units.

There are many other ways to obfuscate the energy dissipation and timing of a circuit. These include:

- Inserting random, fixed delays into a circuit to result in less predictable patterns of operation.
- Inserting dynamically varying delays, based on either the pattern of data (possibly leaking information) or by using some physical process to generate genuinely random delays.
- Reproduction of hardware. A second encryption processor, running a different program, could work in parallel with the first introducing noise into power supply.
- Connection of intentionally noisy structures to the power supply in order to mask the processor power signature.

These are all possible future directions for our research, but it was felt that a ‘balanced’ solution was the most appropriate choice to be able to draw conclusions about the use of 1-of-n codes and self-timed technology in implementing secure solutions.

4.5. Using Balsa

The use of synthesis makes it easier to apply consistency of implementation style across the whole processor. The same Balsa circuit description can also easily be synthesised into a number of different circuits with different combinations of security features with very little effort on the part of the designer. This was not possible with previous Amulet processor designs and will almost certainly lead to SPA being implemented in a number of different guises (should the security benefits become clear).

A number of language-level changes to a Balsa description are possible which affect security. These are principally related to avoiding choice in order to prevent data-dependent timing in the synthesised implementation. Balsa allows descriptions to be parameterised and the structure of a circuit to be influenced by this parameterisation. This feature was heavily used to allow flexibility while taking design decisions relating to the developing Balsa SPA code and to allow such choice-reducing modifications to be turned on or off prior to synthesis. The Balsa system compiles descriptions into circuits in two stages. Firstly, the Balsa compiler processes a Balsa description producing a ‘handshake circuit’ which directly implements that description. Secondly, the Balsa back-end takes that handshake circuit and produces standard-cell netlists of the circuit and its constituent handshake components. The choice of data encoding, handshake protocols and security features in this netlist synthesis can be chosen by the user prior to running the back-end tools. The back-end uses a repository of handshake component synthesis instructions written in a simple technology-independent pseudo-netlist language in order to build parameter specific versions of each component.

These building instructions are grouped into ‘technologies’. A core technology, (‘common’), contains the majority of component descriptions for all the implementation options. Target cell library specific handshake component descriptions are only provided where there is a clear technology-specific advantage to be gained from implementing those components in a non-standard manner. In this way a new protocol or data-encoding can be ported to a new fabrication process with very little effort.

5. Processor Design

As mentioned previously, the SPA core is an implementation of the ARM v5 ISA. There is full support for interrupts, precise exceptions and coprocessors. It is a Harvard design although a wrapper allows it to be used in a von Neumann manner if desired. The core is implemented as an ARM-style, 3-stage Fetch-Decode-Execute design as illustrated in figure 3 and described in more detail below.

Because of the limited time available for its design, simplicity was desired, so in general all of the implementation uses the solutions which were the *simplest to implement in Balsa* whilst still allowing the desired security features. This does not always provide the optimal solution as on occasion the simplest conceptual solution is either difficult or impractical to implement in Balsa, or results in an unacceptable amount of hardware. On occasion, Balsa’s peculiarities suggest an unusual and efficient implementation that would probably not have been arrived at if the design were captured by schematic entry.

5.1. Fetch

The PC (register 15 in the ARM ISA) exists solely in the fetch unit, and as such the fetch unit implements all possible changes to instruction address flow - reset, explicit branches, interrupts, undefined instruction traps and instruction and data fetch memory exceptions (handled differently in the ARM ISA). Some of these changes in flow originate from within the fetch unit (interrupts and instruction fetch aborts), while those originating outside the fetch unit are sent to it by the execute unit

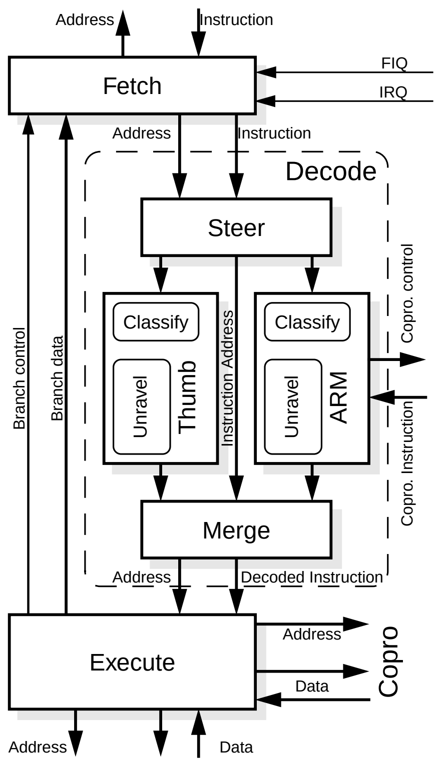

Figure 3: SPA 3-stage pipeline

In common with previous Amulet cores[1, 2, 3], the processor implements a halting mechanism: any instruction which explicitly causes a branch to its own address causes the fetch unit to stop supplying instructions. This is implemented by means of a branch type, one of a small number of bits communicated to the fetch unit besides the target address of the branch.

The origin of the fetch address must be arbitrated between a sequential address, a branch or an interrupt - this is the only place in the Harvard core that arbitration is required. Interrupts cause a branch to a fixed location and are taken in preference to the unbranched instruction stream. They must defer to a branch for a single instruction fetch as failure to do so would result in either the loss of the branch or a possible deadlock, depending on the implementation.

5.2. Decode

In order to decode the Thumb instruction set as well as the ARM v5 set, two decoders are required. Originally it was intended to decompress Thumb instructions into ARM instructions if necessary and then decode the ARM instructions in ignorance of their origin. There is a 1:1 mapping from Thumb to ARM instructions, but it is not necessarily easy to perform this mapping on an instruction, especially in the case of branches relative to the instruction address. It became apparent that the circuit cost of decompressing the Thumb instructions was about the same as that of decoding them directly, but increased the latency for Thumb instructions. As a result of this, it was decided to implement parallel decoders and steer instructions to the appropriate decoder, collecting the decoded instruction from the other end of the relevant decoder. The two parts of each decoder, classification and unravelling, are described in more detail below.

In this architecture every instruction becomes a register-to-register instruction, with the register numbering extended to allow current and saved status registers, coprocessor registers, immediate values, a zero, etc. to be addressed.

The decoder is an elastic pipeline stage as it always receives 32-bit words from the fetch unit and, if the instructions are Thumb instructions, this 32-bit word becomes two decoded instructions. In addition, both ARM and Thumb instructions can specify multiple register loads or stores. This can result in a single 32-bit word theoretically becoming as many as 20 decoded instructions.

5.2.1. Classification

Instruction classification superficially appears trivial, as the instructions are all of a fixed-length. However, the many years of development of the instruction set and complete binary backward-compatibility have resulted in a very disjoint instruction coding. Many data processing operations use exactly the same arrangements of bits and so are classified together and, conversely, some instruction mnemonics can, depending on the addressing mode, result in wildly different bit patterns.

Instruction classification does not modify the instruction in any way, but simply deduces what type of instruction it is and passes this information, along with the unmodified instruction, to the next stage. This classification is not purely according to the assembler mnemonic that would have been assembled into the instruction in question, but rather it is a classification that reflects the arrangement of bits in the instruction word.

The ARM path of this stage must also communicate with the coprocessor system. If a coprocessor can execute the instruction, it latches it and starts to carry out whatever processing it can without committing any irreversible changes. If no coprocessor can execute it the instruction is classified as undefined.

5.2.2. Unravelling

This stage takes the 28 bits (all ARM instructions have a 4-bit condition associated with them which is orthogonal to the instruction itself) or 16 bits (Thumb) of the instruction and expands them to 69 bits of control and 36 bits of register specification, as used in the execution unit.

This stage also unrolls any instructions which require more than one cycle through the execute unit. These instructions are coprocessor memory transfers (where the number of cycles is determined by the coprocessor), the swap instructions (atomic exchange of a register and a memory location) and load or store of multiple registers. For these instructions the decode unit issues multiple decoded instructions to the execute unit which is largely unaware of the multi-cycle nature of the original instruction.

All of the multi-cycle instructions involve (possibly multiple) memory accesses. If a memory abort occurs during a multi-cycle instruction, there is currently no mechanism for the execute unit to communicate this to the decoder and it must consume and discard the unrolled instruction cycles without executing them. This would be a major problem in some systems, but in the context of a smartcard, memory exceptions should be very rare.

5.3. Execute

The execute unit may be conveniently divided into three concurrent parts, the memory access unit, the execution datapath, and the register bank as shown in figure 4. The latter two are described in more detail below - the memory access unit simply takes an address, write data if necessary, and a few bits specifying the nature of the access and returns read data if necessary and a positive or negative abort - an abort is returned every time the memory access unit is used.

5.3.1. Execution Datapath

The execution datapath was designed to mask the nature of its behaviour in order to enhance security. It does this by using all of the channels in the datapath for every instruction that is executed The exceptions to this are channels associated solely with memory accesses. In this design we make no significant effort to hide the identity, parameters or data associated with a memory access in the memory itself, so it was not considered important to hide the nature of the access in the core - the presence of a memory access of some type will be obvious.

The datapath has three fundamental modes of behaviour: memory accesses, multiplications and other instructions. Unless a multiplication is being performed, the multiplier is transparent and control of the rest of the datapath is determined by the value on channel Rs, which is passed through the multiplier to the datapath control unit. When multiplication is being performed and the channel Rs is required as a multiplier argument, this lack of any specific control data is communicated to the control unit which uses a fixed control state for the datapath - performing no memory accesses and using the ALU to resolve the carries for the upper word of a long multiply.

For a memory access, the control data on Rs indicates that a memory access is required, its parameters and whether the addressing should be pre- or post-indexed. It also controls whether the data on Rd is sent to the memory access unit or not, based on whether the memory access is a read or a write. As indicated above, the channel through the CLZ unit is used whether or not the access is a read or a write. If the access is a write then no data will be returned by the memory fetch unit and the data read from Rd will be echoed on Wm.

The use of the Rd and Wm channels for the memory data allows the ALU and shifter to be used for address computation, and for Ws to be used for writing back the resultant address to the base register if necessary.

For instructions that involve neither multiplication nor memory access large parts of the control data have the same value as the only parts of the datapath whose behaviour is not fixed in these circumstances are the ALU (4 bits of control), the CLZ unit (1 bit) and the shifter (10 bits including the size of the shift).

5.3.2. Shifter

If the shift distance is specified as an immediate value in the instruction, and that value is zero, all types of shifts (logical left and right, arithmetic right and rotate right) will yield the same result. This means that three of the possible encodings are redundant and are used for other purposes. This, and other reuse of the encoding space, make for a very complex behaviour by the shifter.

In another effort to hide the behaviour of an instruction, the shifter in this architecture re-encodes this behaviour so that it always performs a shift of some kind, with the 0-bit shift becoming the identical 32-bit rotate right.

5.3.3. ALU

As arithmetic operations are expensive to realise, it was desired to minimise the implementation as far as possible. In Balsa, if an addition and a subtraction of two numbers is described, two arithmetic blocks will be implemented. It is therefore necessary to explicitly describe the transformation of A-B to A+$\overline{B}$+1. Using this method, the five different arithmetic operations all use the same circuit. This is not only more efficient in area, but also makes identification of the arithmetic operation harder for an attacker.

5.3.4. Count Leading Zeroes unit

The Count Leading Zeroes (CLZ) unit is very expensive for its usage, but is required to implement the ARM v5 ISA completely and it would be prohibitively slow if microcoded in the shifter. In this design it is implemented by a recursive function, which is very easy to build in Balsa.

If the most significant half of bits being considered are all zero, then a one is appended to the result and the least significant half of the bits are multiplexed onto the output. If they are not all zero, a zero is appended to the result and the most significant half of the bits are multiplexed onto the output. This is repeated until just a single bit remains whose

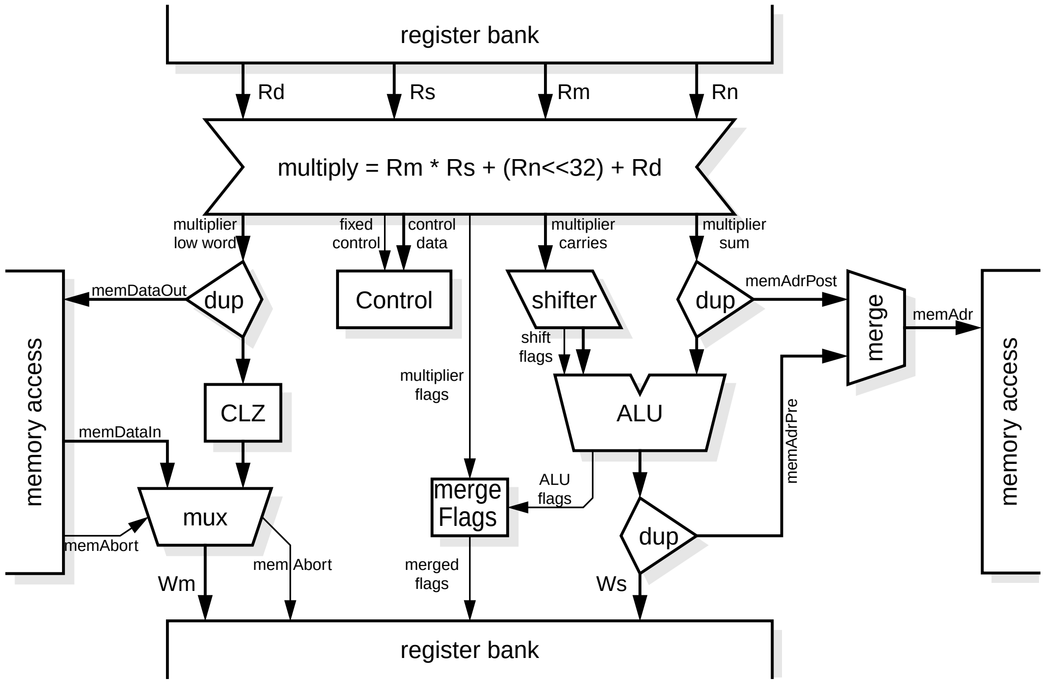

Figure 4: SPA Execute Unit

value is added to both ends of the result; the last bit will be 1 unless all 32 bits are zero, in which case the result would have been 31 were it not for this modification.

This results in an implementation with 31 bits of AND function and 31 bits of multiplexing, which is a relatively cheap and relatively fast implementation.

5.3.5. Multiplier

The multiplier is a 32 by 32 multiplier with 64-bit accumulation. It is implemented by shift-and-add with the sum being shifted right a single bit each cycle rather than one of the multiplication operands being shifted left. This means that only a shift of 1 is needed and only 33 bits of adder. The addition does not explicitly resolve any carries, but saves them for resolution in the next cycle.

All carries from the low half of the 64-bit result are implicitly resolved during the multiplication process, but the sums and carries from the addition in the last cycle represent the most significant 32 bits and are added together in the ALU to resolve them.

No early termination of the multiply is provided as this would allow deduction of operands more easily from behavioural information. Similarly, the addition is performed in each cycle, regardless of the value of the relevant multiplier bit, to prevent behavioural based deduction of arguments. Additionally, all multiplies are considered to be long and to have accumulation – the accumulator values supplied to the multiplier are zero if no accumulation is really required. Again, this adds slightly to the difficulty of attack.

5.3.6. Register Bank

The register bank is the most complex part of the core as it must implement most of the unusual corners of the ISA as well as the more regular parts. It operates in a loop where it performs the following operations in logical sequence:

- Decoded instruction reading
- Condition-code/abort check
- Register reading
- Dispatch of registers and control to datapath
- Reading of datapath results
- Register writing
- Resolution of effects of a memory abort if necessary
- Updating of status registers

While these are a logical sequence there are only three physically sequential steps at the top level. The first two operations are easily parallelisable, as are the last three. The remaining three are implicitly sequenced by the passage of the register contents through the datapath, and are thus allowed to share the same sequencing slot at this level.

When a source or destination register is a real register (logically numbered 0-14) or a dummy register (reads as zero and discards writes) the effects of reading from or writing to a register are easy to implement. When the destination of a write is the PC (R15) then a branch results and much more must be done than simply updating a real register. Similarly, if the current status register is the target of a write then a mode change and a branch may result.

The condition-code check is required on every cycle as any ARM instruction may be conditionally executed. The abort check is required over and above a branch colour check as, if a multi-cycle instruction has suffered a memory abort in an earlier cycle, it must not continue to actually execute, but should instead be discarded.

If an instruction is a coprocessor instruction then at least part of the functionality must execute even if the condition code fails, as the execute unit must communicate to the coprocessor whether it should execute or discard the instruction. It will be recalled that the coprocessor will already have performed some or all of the computation required and this signal to the coprocessor is an indication that it should commit or flush those results.

6. Memory Protection Unit

As smartcards become more complex and the requirement to store and run several applications has emerged, it has become desirable to employ some form of memory protection unit. An MPU allows the operating system and each application to protect its memory space from other applications which may not be trusted. It also affords some protection from glitch attacks which disrupt the normal flow of execution in the hope of obtaining information from the card.

The MPU provides a relatively lightweight address space access control system and is used to partition the memory address space into regions and set individual protection attributes for each region. Up to eight protected memory regions of variable size can be defined and the MPU will allow or deny access to these regions, depending on the mode (user/privileged) and type (read/write) of the attempted access. If the access is allowed then the request is passed to the memory system and any data or abort that results is passed back to the processor. If the permissions do not allow the access or if no protected region is hit then the MPU generates an abort and the access is abandoned. The MPU does not perform address translation.

The MPU can operate in two modes and, by default, it operates as an ARM standard MPU [15]. However, it can be switched into secure mode, which provides finer-grained control of memory regions and additional resources to track successful read and write accesses to regions. Figure 5 shows how the MPU is organized. The MPU is disabled on

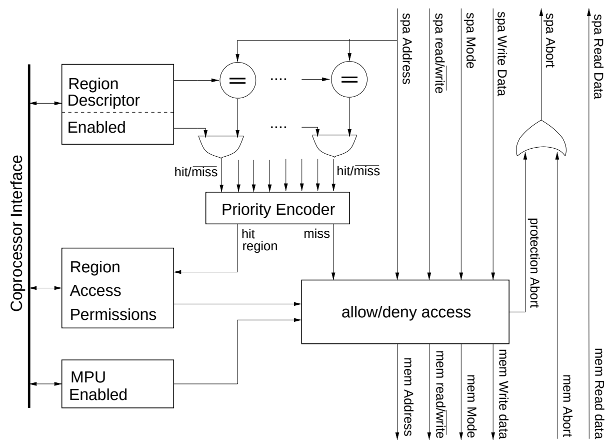

Figure 5: SPA MPU organisation

reset and all accesses are allowed. The operating system can dynamically enable/disable the MPU. Each region can be enabled individually and is defined through a region descriptor and a region access permits register.

The MPU operates as a coprocessor and is accessed by the processor through its normal coprocessor interface: all region descriptors, access permission registers and read/write tracking registers appear as coprocessor registers. In default mode, the region descriptor contains the base address and size of the region. Regions can have sizes of 4KB to 4GB. In secure mode, the descriptor contains the start and end addresses of the region and the sizes can vary from 1 byte to 4GB. Access permissions depend on the mode and type of access. In privileged mode, attributes are ‘no access’ and ‘read/write access’. In user mode, regions can be defined as ‘no access’, ‘read-only access’ and ‘read/write access’.

The incoming address is compared in parallel with all enabled region descriptors to determine the hit region. Since regions can overlap, they are prioritized and the attributes of the highest-numbered hit region are used. If none of the regions is hit the access is denied. In secure mode, the MPU keeps track of which regions are successfully accessed, both on reads and writes. This is provided as a means for the operating system to determine which protection region to change when the need arises.

The self-timed nature of the MPU allows it to handle denied, allowed and aborted memory accesses as soon as they are identified and to respond in the shortest possible time in each situation.

7. Results

At the time of writing the design work is complete and the design is being validated. A fabrication route has been identified and it is expected that the device will be fabricated in a 0.18μm process and that samples will be available in 2Q 2002.

It was expected from the outset that the use of the secure dual-rail technology would be expensive in terms of transistors and that the use of synthesis would increase this cost still further over hand designed logic. The preliminary transistor counts obtained thus far confirm these expectations. For the SPA processor the counts in table 1 have been obtained (the register bank is a component of the execute stage but it is shown separately as it contributes significantly to the total count). The table also shows a breakdown of the Amulet3 core and the relative cost of dual-rail against bundled data.

Block	Secure Dual-Rail	Bundled Data	Cost Factor	Amulet3 core
Fetch	33,752	12,428	2.7	23,474
Decode	172,342	80,460	2.1	18,087
Register Bank	321,482	58,886	5.5	24,426
Execute	198,366	55,714	3.6	37,806
Total	736,878	205,070	3.6	103,793

Table 1: SPA transistor counts

The total transistor count for the SPA processor is around 740,000. For comparison, the Amulet3 core contains around 104,000 transistors. Amulet3 includes several features not provided in SPA, such as a Branch Target Cache (in the Fetch unit) and a reorder buffer (in the Register Bank). A more relevant comparison, which gives some idea of the efficiency of Balsa, is to compare Amulet3, which was hand designed using a bundle-data approach, with Balsa configured to generate bundled-data circuits (a total of 205,000 transistors). Here the SPA core contains almost twice as many transistors as the more complex Amulet3. Comparing the synthesised secure dual-rail implementation with the synthesised bundled-data version shows an overall cost factor of 3.6 in terms of transistors.

The register bank is particularly costly in the dual-rail implementation. This is because there are four read ports and, unlike a bundled data implementation where the port can simply be bussed to each input that it drives, a handshake circuit is required for each bit of each input fed by that port.

Simple optimisations in the Balsa back-end which translates the Balsa intermediate format to the target technology netlist have already yielded significant reductions in transistor count and it is expected that further optimisations will be possible before the device is sent for fabrication.

We are awaiting simulation models from the semiconductor vendor which would allow us to simulate the design as a transistor netlist. Until these are available it is not possible to give an estimate of performance or the security benefits.

8. Conclusions

We have described the design of a self-timed, ARM-compatible core using the Balsa synthesis system. Balsa has shown itself to be capable of dealing with the complexity of a 32-bit processor and the design has progressed very rapidly with major changes being readily accommodated at all stages. This is in contrast to the hand-designed implementation of Amulet3 where much greater thought went into the design at an early stage as major changes were very difficult once implementation was underway. In addition, synthesis gives the flexibility to explore a range of different implementations as part of the design process.

We estimate that the SPA design will have consumed under 2 man-years of effort by the time it is complete. The Amulet3 core required about 8 man-years. One price of this rapid development using synthesis is the size of the result - at least a factor 2 in terms of transistor count seems likely for processors with similar features and design style.

9. Acknowledgments

The development of SPA has been supported primarily within the EU-funded G3CARD project, and authors are grateful to the European Commission for their support for this work. The support of our project partners in G3CARD, is also acknowledged. Work on Balsa is supported by a grant from the EPSRC and funding for the work on CHAIN has been provided by Theseus Logic Inc.

10. References

[1] Woods, J.V., Day, P., Furber, S.B., Garside, J.D., Paver, N.C. and Temple, S., “AMULET1: An Asynchronous Microprocessor”, IEEE Trans. Computers, 46 (4), April 1997, pp. 385-398. ISSN 0018-9340.

[2] 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

[3] Furber, S.B., Edwards, D.A. and Garside, J.D., “AMULET3: a 100 MIPS Asynchronous Embedded Processor”, Proc. ICCD’00, Austin, Texas, September 17-20 2000, pp. 329-334, ISSN 1063-6404, ISBN 0-7695-0801-4.

[4] Furber, S.B., “ARM System-on-Chip Architecture”, Addison Wesley, 2000. ISBN 0-201-67519-6.

[5] van Berkel, K. “Handshake Circuits - An asynchronous architecture for VLSI programming”, Cambridge International Series on Parallel Computers 5, Cambridge University Press, 1993.

[6] van Gageldonk, H., Baumann, D., van Berkel, K., Gloor, D., Peeters, A., Stegmann, G., “An asynchronous low-power 80C51 microcontroller”. Proc. Async’98, San Diego, March 1998, pp 96-107.

[7] Bardsley, A. “Implementing Balsa Handshake Circuits”, PhD Thesis, University of Manchester, 2000, (http://www.cs.man.ac.uk/amulet/publications/thesis/bardsley00_phd.html)

[8] http://www.cs.man.ac.uk/pub/amulet/projects/balsa

[9] Garside, J.D., Bainbridge, W.J., Bardsley, A., Clark, D.M., Edwards, D.A., Furber, S.B., Liu, J., Lloyd, D.W., Mohammadi, S., Pepper, J.S., Petlin, O., Temple, S., Woods J.V., “AMULET3i - an Asynchronous System-on-Chip”, Proc. Async 2000, April 2000, pp. 162-175, IEEE Computer Society Press, ISSN 1522-8681, ISBN 0-7695-0586-4

[10] Bardsley, A., Edwards, D. A., “Synthesising an asynchronous DMA controller with Balsa”, Journal of Systems Architecture 46 (2000) pp 1309-1319.

[11] Bainbridge, W.J., Furber, S. “Delay Insensitive System-on-Chip Interconnect Using 1-of-4 Data Encoding”, Proc. Async 2001, March 2001, pp. 118-126. IEEE Computer Society Press, ISSN 1522-8681, ISBN 0-7695-1034-4

[12] Bainbridge, W.J., “Asynchronous System-on-Chip Interconnect”, PhD Thesis, Dept. of Computer Science, University of Manchester, 2000. (http://www.cs.man.ac.uk/amulet/publications/thesis/bainbridge00_phd.html)

[13] ISO/IEC 7816-3: 1997, “Identification Cards - Integrated Circuit(s) with Contacts - Part 3: Electronic Signals and Transmission Protocols”, International Standards Organisation.

[14] Garside, J.D., Furber, S.B., Chung, S-H. “AMULET3 Revealed” Proc. Async’99, Barcelona, April 1999, pp. 51-59.

[15] Jaggar, D., Seal, D., “ARM Architecture Reference Manual”, Addison Wesley Publishing Company. 2000, ISBN: 020173719

[16] Furber, S.B., Efthymiou, A., Garside, J.D., Lloyd, D.W., Lewis, M.J.G. and Temple, S., “Power Management in the Amulet Microprocessors”, IEEE Design and Test of Computers **18** (2) special issue on Dynamic Power Management of Electronic Systems (Ed. E. Macii), March-April 2001, pp. 42-52. ISSN 0740-7475.

[17] Kocher, P., Jaffe, J., Jun, B., “Introduction to Differential Power Analysis and Related Attacks”, Technical Report, Cryptography Research, 1998.

[18] Sparsø, J., Staunstrup, J. “Delay Insensitive Multi Ring Structures. Integration”, The VLSI Journal. Vol. 15. 1993.

[19] Yu, Z. C., Furber, S.B., Defeating Power Analysis, Proceedings of the 9th UK Asynchronous Forum, Cambridge, 18-19 Dec 2000