The ARM Architecture

Article · July 2006

CITATIONS
8

READS
8,845

1 author:

Leonid Ryzhyk

47 PUBLICATIONS **571** CITATIONS

SEE PROFILE

The ARM Architecture

Leonid Ryzhyk
<leonidr@cse.unsw.edu.au>

June 5, 2006

1 Introduction

ARM is a a 32-bit RISC processor architecture currently being developed by the ARM corporation. The business model behind ARM is based on licensing the ARM architecture to companies that want to manufacture ARM-based CPU's or system-on-a-chip products. The two main types of licenses are the *Implementation license* and the *Architecture license*. The Implementation license provides complete information required to design and manufacture integrated circuits containing an ARM processor core. ARM licenses two types of cores: *soft cores* and *hard cores*. A hard core is optimised for a specific manufacturing process, while a soft core can be used in any process but is less optimised. The architecture license enables the licensee to develop their own processors compliant with the ARM ISA.

ARM processors possess a unique combination of features that makes ARM the most popular embedded architecture today. First, ARM cores are very simple compared to most other general-purpose processors, which means that they can be manufactured using a comparatively small number of transistors, leaving plenty of space on the chip for application-specific macrocells. A typical ARM chip can contain several peripheral controllers, a digital signal processor, and some amount of on-chip memory, along with an ARM core.

Second, both ARM ISA and pipeline design are aimed at minimising energy consumption — a critical requirement in mobile embedded systems. Third, the ARM architecture is highly modular: the only mandatory component of an ARM processor is the integer pipeline; all other components, including caches, MMU, floating point and other co-processors are optional, which gives a lot of flexibility in building application-specific ARM-based processors.

Finally, while being small and low-power, ARM processors provide high performance for embedded applications. For example, the PXA255 XScale processor running at 400MHz provides performance comparable to Pentium 2 at 300MHz, while using fifty times less energy.

This report is largely based on material from Steve Furber's book about the ARM architecture [Fur00]. Other sources are referenced throughout the report.

2 History

The history of ARM started in 1983, when a company named Acorn Computers was looking for a 16-bit microprocessor for their next desktop machine. They quickly discovered that existing commercial microprocessors did not satisfy their requirements. These processors were slower than standard memory parts available at the time. They had complex instruction sets that included instructions taking hundreds of cycles to execute, leading to high interrupt latencies. The reason for these deficiencies was that early integrated microprocessors were modelled after processors of minicomputers, which consisted of many chips, were driven by microcode and had very complex instruction sets.

Therefore, Acorn engineers considered designing their own microprocessor. However, resources required for such a project were well beyond what the company could afford. In early 80's, microprocessor architectures became so complex that it took many years even for large companies with significant expertise in the area to design a new processor.

Fortunately, just two years before this, the Berkeley RISC 1 project had shown that it had actually been possible to build a very simple processor with performance comparable to the most advanced CISC processors of the time. Acorn decided to pick up the Berkeley approach, and two years later, in 1985, they released their first 26-bit Acorn RISC Machine (ARM) processor, which also became the first commercial RISC processor in the world. It used less than 25,000 transistors — a very small number even for 1985, but still performed comparably to or even outperformed the Intel 80286 processor that came out at about the same time. This first ARM architecture is now referred to as the ARM version 1 architecture. It was followed by the second processor in 1987. The main new feature introduced in ARM version 2 was coprocessor support. Later, the ARM v2 architecture was implemented with on-chip cache in the ARM3 processor.

In 1990 Apple made a strategic decision to use an ARM processor in their Newton PDA. A joint venture was co-founded by Acorn and Apple to design a new processor. The company was called ARM, standing for Advanced RISC Machines. The third version of the ARM architecture was developed by this company and featured 32-bit addressing, MMU support and 64-bit multiply-accumulate instructions. It was implemented in ARM 6 and ARM 7 cores. The release of these processors and the Apple Newton PDA in 1992 marked ARM's move to the embedded market.

The 4th generation of ARM cores came out in 1996. The main innovation in this version of the architecture was support for Thumb 16-bit compressed instruction set. Thumb code takes 40% less space compared to regular 32-bit ARM code but is slightly less efficient. The most prominent representative of the 4th generation of ARM's is the ARM7TDMI core, which still remains the most popular ARM product, in particular, it is used in most Apple iPod players, including the video iPod. Curiously, ARM7TDMI is based on essentially the same 3-stage pipeline as the very first ARM designed in 1985, and contains only 30000 transistors. Another popular implementation of the ARM v4 architecture

Version	Year	Features	Implementations
v1	1985	The first commercial RISC (26-bit)	ARM1
v2	1987	Coprocessor support	ARM2, ARM3
v3	1992	32-bit, MMU, 64-bit MAC	ARM6, ARM7
v4	1996	Thumb	ARM7TDMI, ARM8, ARM9TDMI, StrongARM
v5	1999	DSP and Jazelle extensions	ARM10, XScale
v6	2001	SIMD, Thumb-2, TrustZone, multiprocessing	ARM11, ARM11 MPCore
v7	?	VFP-3	?

Table 1: History of the ARM architecture

is the Intel StrongARM processor.

In 1999, the 5th generation of the ARM architecture introduced digital signal processing and Java byte code extensions to the ARM instruction set. The most popular implementation of this architecture is the Intel XScale processor. It is used in a variety of high-end embedded devices, including network processors, smart-phones, and PDA's.

The 6th generation of the ARM architecture was released in 2001 introducing SIMD instruction set extension, improved Thumb instruction set, the TrustZone virtualisation technology, and multiprocessor support.

The recently released ARMv7 architecture features extended SIMD instruction set and improved floating point support.

Table 1 summarises the history of the ARM architecture.

References: [Lev05].

3 ARM ISA overview

In most respects, ARM is a RISC architecture. Like all RISC architectures, the ARM ISA is a load-store one, that is, instructions that process data operate only on registers and are separate from instructions that access memory. All ARM instructions are 32-bit long and most of them have a regular three-operand encoding. Finally, the ARM architecture features a large register file with 16 general-purpose registers. All of the above features facilitate pipelining of the ARM architecture.

However, the desire to keep the architecture and its implementation as simple as possible prompted several design decisions that deviated from the original RISC architecture. First, the original Berkeley RISC design used register windows to speedup procedure invocations. ARM designers rejected this feature as one that would increase size and complexity of the processor. In retrospect, this appears to be a wise decision, as register windows did not prove to be a

particularly useful feature and are not used in the majority of modern RISC processors.

Second, the classical RISC approach requires the execution stage of any instruction to complete in one cycle. This is essential for building an efficient 3-stage fetch-decode-execute pipeline. While most ARM data processing instructions do complete in one cycle, data transfer instructions are one important exception. Completing a simple store or load instruction in one cycle would require performing two memory accesses in a single cycle: one — to fetch the next instruction from memory, and the other — to perform the actual data transfer. Performing two memory accesses in one cycle would, in its turn, requires a Harvard architecture with separate instruction and data memories, which was considered too expensive by the designers of the first ARM processor. However, in order to achieve better utilisation of the pipeline during 2-cycle instruction executions, they introduced an auto-indexing addressing mode, where the value of an index register is incremented or decremented while a load or store is in progress. While all modern ARM implementations have separate instruction and data caches and can complete a memory transfer in one cycle, they still support the auto-indexing mode that proved to improve performance and code size of ARM programs.

Third, ARM supports multiple-register-transfer instructions that allow to load or store up to 16 registers at once. While violating the one cycle per instruction principle, they significantly speedup performance-critical operations, such as procedure invocation and bulk data transfers, and lead to more compact code.

In summary, the ARM architecture offers all the benefits of the RISC approach, such as pipeline-friendlyness and simplicity, while deviating from it in few aspects, which makes it even more appealing to embedded systems developers.

3.1 Registers

The ARM ISA provides 16 general-purpose registers in the user mode (see Figure 1). Register 15 is the program counter, but can be manipulated as a general-purpose register. The general-purpose register number 14 has is used as a link register by the branch-and-link instruction (see Section 3.4). Register 13 is typically used as stack pointer, although this is not mandated by the architecture.

The *current program status register* (CPSR) contains four 1-bit condition flags ('Negative', 'Zero', 'Carry', and 'oVerflow') and four fields reflecting the execution state of the processor. The 'T' field is used to switch between ARM and Thumb (Section 8.1) instruction sets. The 'I' and 'F' flags enable normal and fast interrupts respectively. Finally, the 'mode' field selects one of seven execution modes:

- *User mode* is the main execution mode. By running application software in user mode, the operating system can achieve protection and isolation.

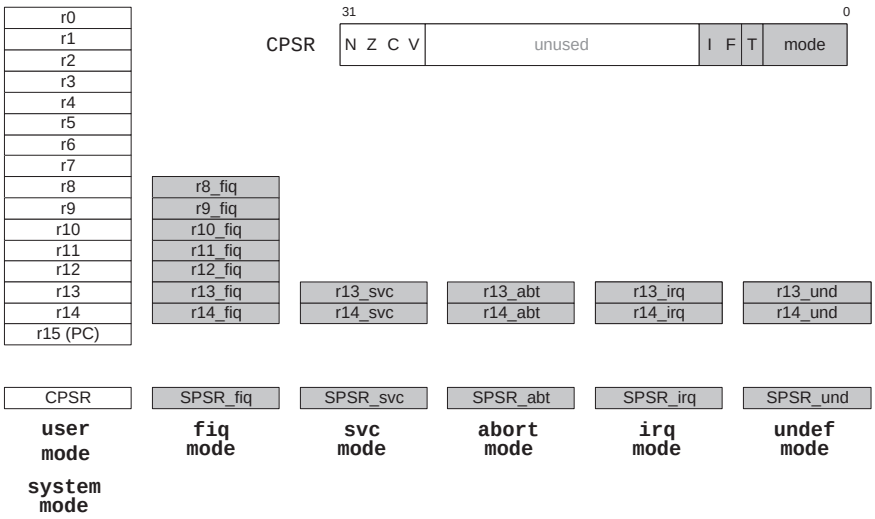


Figure 1: ARM registers

All other execution modes are privileged and are therefore only used to run system software.

- *Fast interrupt processing mode* is entered whenever the processor receives an interrupt signal from the designated fast interrupt source.
- *Normal interrupt processing mode* is entered whenever the processor receives an interrupt signal from any other interrupt source.
- *Software interrupt mode* is entered when the processor encounters a software interrupt instruction. Software interrupts are a standard way to invoke operating system services on ARM.
- *Undefined instruction mode* is entered when the processor attempts to execute an instruction that is supported neither by the main integer core nor by one of the coprocessors. This mode can be used to implement coprocessor emulation.
- *System mode* is used for running privileged operating system tasks.
- *Abort mode* is entered in response to memory faults.

In addition to user-visible registers, ARM provides several registers available in privileged modes only (shaded registers in Figure 1). SPSR registers are used to store a copy of the value of the CPSR register before an exception was raised. Those privileged modes that are activated in response to exceptions have their own R13 and R14 registers, which allows to avoid saving the corresponding user registers on every exception. In order to further reduce the amount of state that

has to be saved during handling of fast interrupts, ARM provides 5 additional registers available in the fast interrupt processing mode only.

3.2 Data processing instructions

The ARM architecture provides a range of addition, subtraction, and bit-wise logical operations that take two 32-bit operands and return a 32-bit result that can be independently specified in the 3-address instruction format. The first operand and the result should be stored in registers, while the second operand can be either register or immediate. In the former case, the second operand can be shifted or rotated before being sent to the ALU. Due to the limited space available for operand specification inside the 32-bit instruction, an immediate operand should be a 32-bit binary number where all the binary ones fall within a group of eight adjacent bit positions on a 2-bit boundary.

ARM also supports several multiply and multiply-accumulate instructions that take two 32-bit register operands and return a 32- or 64-bit result. Finally, ARM supports binary comparison operations that do not return any values but only modify condition flags in the CPSR register.

One interesting feature of the ARM architecture is that modification of condition flags by arithmetic instructions is optional, which means that flags do not necessarily have to be read right after the instruction that set them, but it can be done later in the instruction stream provided that other intermediate instructions do not change the flags.

3.3 Data transfer instructions

ARM supports two types of data transfer instructions: *single-register transfers* and *multiple-register transfers*. Single-register transfer instructions provide a flexible way to move 1, 2, or 4-byte blocks between registers and memory, while multiple-register transfer instructions provide an efficient but less flexible way to move larger amounts of data. The main addressing mode is base-plus-offset addressing. Value in the base register is added to the offset stored in a register or passed as an immediate value to form the memory address for load or store.

As was mentioned above, in the original ARM pipeline loads and stores took two cycles to execute. Therefore, *auto-indexed* addressing was introduced to keep the pipeline busy while the processor is reading or writing memory. An auto-indexed addressing mode writes the value of the base register incremented by the offset back to the base register, so it is possible to access the value in the next memory location in the following instruction, without wasting an additional instruction to increment the register. Two auto-indexed addressing modes are supported: the *pre-indexed* mode uses the computed address for the load or store operation, and then updates the base register to the computed value, while the *post-indexed* mode uses the unmodified base register for the transfer, and then updates the base register to the computed address.

Multiple-register transfer instructions allow to load or store any subset of the

sixteen general-purpose registers from/to sequential memory addresses. Auto-indexed addressing modes are also supported for multiple-register transfers.

3.4 Control flow instructions

In addition to usual conditional and unconditional branch instructions, the ARM architecture provides support for *conditional execution* of arbitrary instructions: any instruction can be predicated using values of CPSR condition flags. ARM supports efficient procedure invocations using a *branch-and-link* instruction that saves the address of the instruction following the branch to R14.

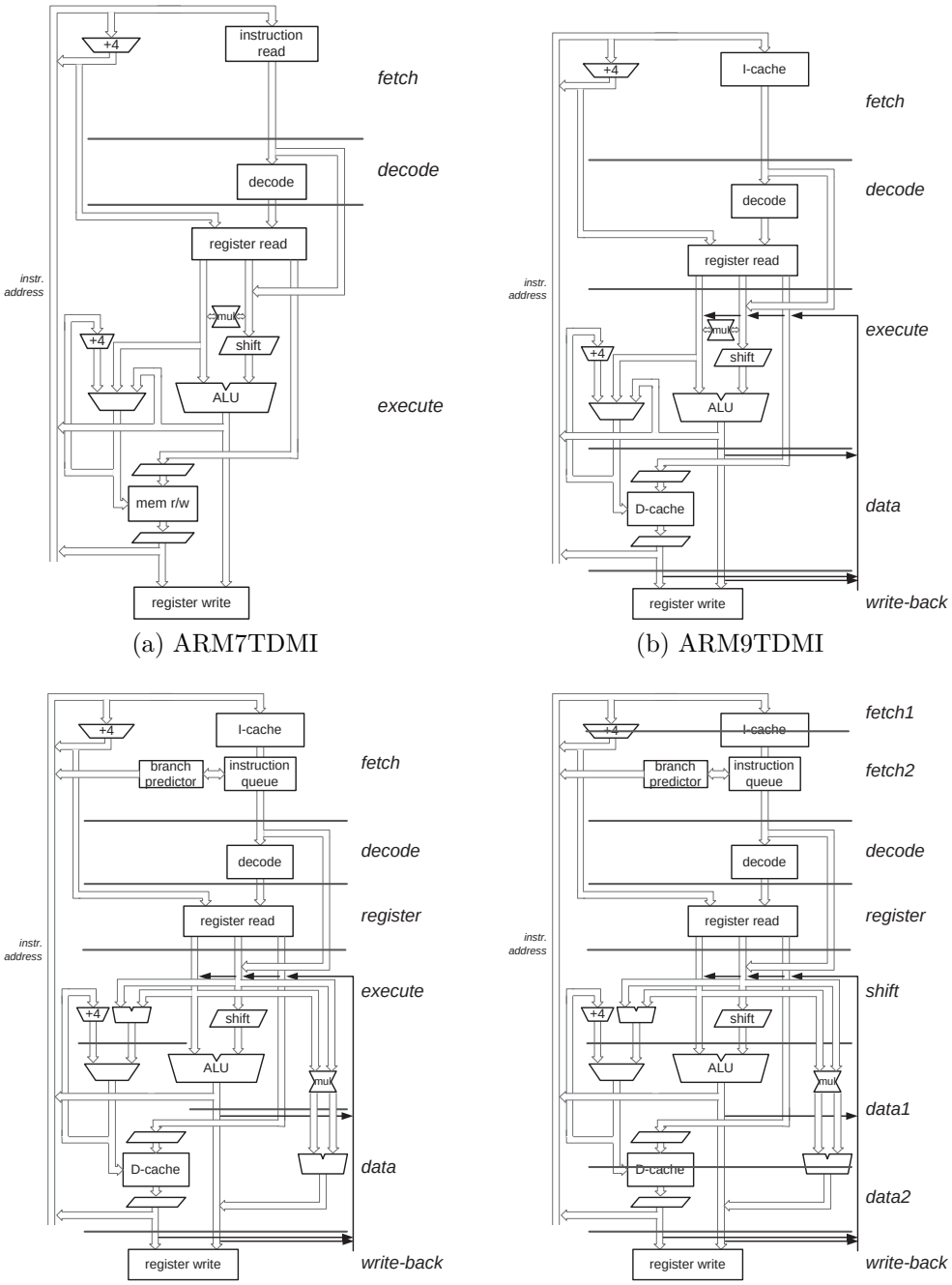
4 Evolution of the ARM pipeline

One important property of the ARM architecture mentioned above is its simplicity. Without register windows, register renaming, out-of-order execution, multiple instruction issue, speculation, and other complex optimisations found in modern microprocessors, even the latest ARM pipelines remain quite simple. This section traces the evolution of the ARM integer pipeline between ARM1 and ARM11 processors.

4.1 The 3-stage pipeline

Figure 2a shows the original 3-stage ARM pipeline that remained essentially unchanged from the first ARM processor to the ARM7TDMI core. It is a classical fetch-decode-execute pipeline, which, in the absence of pipeline hazards and memory accesses, completes one instruction per cycle. The first pipeline stage reads an instruction from memory and increments the value of the instruction address register, which stores the value of the next instruction to be fetched. This value is also stored in the PC register. The next stage decodes the instruction and prepares control signals required to execute it on. The third stage does all the actual work: it reads operands from the register file, performs ALU operations, reads or writes memory, if necessary, and finally writes back modified register values. In case the instruction being executed is a data processing instruction, the result generated by the ALU is written directly to the register file and the execution stage completes in one cycle. If it is a load or store instruction, the memory address computed by the ALU is placed on the address bus and the actual memory access is performed during the second cycle of the execute stage.

Note that since PC is accessible as a general-purpose register, there are several places in the pipeline where the next instruction address can be issued. Under normal conditions, it is incremented on every cycle during the fetch stage. If a data processing instruction specifies R15 as its destination operand, then the result of the ALU operation is used as the next instruction address. Finally, a load to R15 has a similar effect.



(a) ARM7TDMI (b) ARM9TDMI

(c) ARM10TDMI (d) ARM11

Figure 2: Evolution of the ARM pipeline

4.2 The 5-stage pipeline

The pipeline structure shown in Figure 2a assumes availability of only one memory port, which means that every data transfer instruction causes a pipeline stall, because the next instruction cannot be fetched while memory is being read or written. One way around this problem, which was used in ARM9TDMI (Figure 2b) and later microarchitectures, is to use separate instruction and data caches. This allows to modify the pipeline to avoid stalls on data transfer instructions.

First, to make the pipeline more balanced, ARM9TDMI moved the register read step to the decode stage, since instruction decode stage was much shorter than the execute stage. Second, the execute stage was split into 3 stages. The first stage performs arithmetic computations, the second stage performs memory accesses (this stage remains idle when executing data processing instructions) and the third stage writes the results back to the register file.

This results in a much better balanced pipeline, which can run at faster clock rate, but there is one new complication — the need to forward data among pipeline stages to resolve data dependencies between stages without stalling the pipeline. The forwarding paths are shown with black arrows in Figure 2b.

4.3 The 6-stage pipeline

The ARM10 core made further improvements to the pipeline structure (see Figure 2c). Designers realised that the pipeline performance was mainly constrained by memory bandwidth. Therefore, they made both instruction and data buses 64-bit-wide. In the fetch stage, this allowed to fetch two instructions on each cycle, which enabled the introduction of a static branch prediction unit. The branch predictor tries to look ahead the instruction stream and predicts backward pointing branches as taken and forward pointing branches as not taken. This trivial algorithm eliminates the penalty of branches for loops that execute many times. In the execution section of the pipeline, 64-bit data bus allowed to improve the performance of multiple-register transfer instructions by transferring two registers at a time.

Next, ARM10 unloaded the execute stage by introducing a separate adder for multiply-accumulate instructions instead of using the main ALU to do addition. Since multiply instructions do not read or write memory, this adder could be placed to the data stage, which lead to a better balanced pipeline and enabled it to run at a higher clock rate.

At this point, the memory access stage became the longest running pipeline stage, hindering further growth of clock frequency. In order to remove this performance bottleneck, another adder dedicated to address computation was introduced. Since address computation is always just a simple addition, this adder could complete its computation in less than 1 cycle, effectively leaving one and a half cycles for the memory access.

The last improvement introduced in the ARM10 pipeline was separation of instruction decoding into a separate stage.

4.4 The 8-stage pipeline

The ARM11 core introduced two main changes to the pipeline architecture (see Figure 2d). First, the shift operation was separated into a separate pipeline stage. Second, both instruction and data cache accesses are now distributed across 2 pipeline stages.

Note that the instruction execution section of the resulting 8-stage pipeline is split into three separate pipelines that can operate concurrently in some situations and can commit instructions out-of-order. Still, fetch and decode stages are executed in order.

5 Exceptions

The ARM architecture defines the following types of exceptions (listed in the order of decreasing priority):

- *Reset* starts the processor from a known state and renders all other pending exceptions irrelevant.
- *Data abort* exception is raised by memory management hardware when a load or store instruction violates memory access permissions.
- *Fast interrupt* exception is raised whenever the processor receives an interrupt signal from the designated fast interrupt source.
- *Normal interrupt* exception is raised whenever the processor receives an interrupt signal from any non-fast interrupt source.
- *Prefetch abort* exception is raised by memory management hardware when memory access permissions are violated during instruction fetch.
- *Software interrupt* exception is raised by a special instruction, typically to request an operating system service.
- *Undefined instruction* exception is generated when trying to decode an instruction that is supported neither by the main integer core nor by one of the coprocessors.

Except for the reset exception, all exceptions are handled in a similar way: the processor switches to the corresponding execution mode (see Section 3.1), saves the address of the instruction following the exception entry instruction in R14 of the new mode, saves the old value of CPSR to SPSR of the new mode, disables IRQ (in case of a fast interrupt, FIQ is also disabled), and starts execution from the relevant exception vector.

6 Coprocessors

The ARM architecture supports a general mechanism for extending the instruction set through the addition of coprocessors. For example, the ARM floating point unit is implemented as a coprocessor. Another example of a coprocessor is the system control coprocessor that manages MMU, caches, TLB, and the write buffer.

The ARM architecture defines a protocol for interaction between the main ARM core and coprocessors and instructions for data transfer between ARM and coprocessors. Coprocessors are required to expose a load-store architecture. Each coprocessor can have up to sixteen registers of any size. There are three types of coprocessor-related instructions recognised by the main ARM core. First, there are data processing instructions. These are completely internal to the coprocessor. Whenever the ARM core fetches such an instruction, it executes a simple handshake protocol to make sure that one of coprocessors accepts this instruction.

The second type of coprocessor instructions are load-store instructions that transfer data between coprocessor registers and memory. The ARM core initiates execution of such instructions by computing a memory address and sending it to the address bus. However, since the ARM core does not know the size of coprocessor registers, the coprocessor controls the number of transferred words itself.

Finally, the ARM architecture supports register transfer instructions that transfer data between the integer pipeline and coprocessor registers. No matter what the native coprocessor register size is, these operations transfer 32-bit values only.

7 Memory hierarchy

7.1 Cache

All modern ARM-based processors are equipped with either on-chip L1 cache or on-chip memory. In most implementations, cache is split into separate instruction and data caches. Both caches are typically virtually addressed and set-associative with high degree of associativity (up to 64-way). The instruction cache is read-only, while the data cache is read/write with copy-back write strategy. One less standard feature present in most ARM processors is cache lock-down capability that allows to lock critical sections of code or data in cache.

7.2 Memory protection

ARM-based chips come with one of two types of memory protection hardware. Simple systems that run a predefined set of applications and do not require full-featured protection are equipped with simple *protection units*. A protection unit does not provide address translation, but simply defines eight regions within the

4-gigabyte physical address space and allows assigning access permissions and cacheability attributes to individual regions.

General-purpose ARM-based systems are equipped with *memory management units* (MMU) that provide a virtual memory model similar to conventional desktop and server processors. The MMU supports two-level page tables with page sizes of 1M, 64K, 4K, and 1K. 1M pages are called sections and are described by first-level page table entries. Other types of pages are described by second-level page table entries. Address translations are cached in separate address and data *translation lookaside buffers* (TLB) typically implemented as fully associative caches.

One major concern associated with memory protection is the cost of address space switching. On ARM a context switch requires switching page tables. The complete cost of page table switch includes the cost of flushing page tables, purging TLBs and caches and then refilling them. Two mechanisms were introduced to enable operating system designers eliminate this cost in some cases. The first mechanism is *protection domains*. Every virtual memory page or section belongs to one of sixteen protection domains. At any point in time, the running process can be either a manager of a domain, which means that it can access all pages belonging to this domain bypassing access permissions, a client of the domain, which means that is can access pages belonging to the domain according to their page table access permission bits, or can have no access to the domain at all. In some situations, it is possible to do context switch by simply changing domain access permissions, which means simply writing a new value to the *domain access register* of coprocessor 15.

The second mechanism present in newer ARM cores is the *fast context switch extension* (FCSE) that allows multiple processes to use identical address ranges, while ensuring that the addresses they present to the rest of the memory system differ. To that end, virtual addresses issued by a program within the first 32 megabytes of the address space are effectively augmented by the value of the *process identifier* (PID) register. FCSE allows to avoid the overhead of purging caches when performing a context switch; however it is still necessary to flush TLBs.

References: [WH00], [Sea00].

8 ARM ISA extensions

8.1 Thumb

The Thumb instruction set was introduced in the fourth version of the ARM architecture in order to achieve higher code density for embedded applications. Thumb provides a subset of the most commonly used 32-bit ARM instructions which have been compressed into 16-bit wide opcodes. On execution, these 16-bit instructions can be either decompressed to full 32-bit ARM instructions or executed directly using a dedicated Thumb decoding unit. Although Thumb code uses 40% more instructions than equivalent 32-bit ARM code, it typically

requires 30% less space. Thumb code is 40% slower than ARM code; therefore Thumb is usually used only in non-performance-critical routines in order to reduce memory and power consumption of the system.

8.2 Jazelle

Jazelle is a hardware implementation of the Java virtual machine that allows ARM processors to execute Java bytecode. The hardware implementation avoids the overhead of software approaches, such as software emulation of the virtual machine or JIT compilation.

8.3 DSP extensions

ARM-based systems typically perform signal processing tasks using dedicated DSP coprocessors. However, in some cases it is convenient to have some DSP support provided by the main ARM core. A corresponding ISA extension was introduced in the fifth version of the architecture. It features support for saturating addition and subtraction operations and 16-bit multiplication.

In the sixth version of the ARM architecture, DSP support evolved into the SIMD instruction set extension that allows simultaneous execution of two 16-bit or four 8-bit arithmetic instructions.

8.4 TrustZone

The TrustZone extension enables creation of a trusted computer base (TCB) within a system-on-chip. Such TCB can include a small set of trusted applications running in a special privileged execution mode outside the main operating system, on-chip or off-chip secure memory and peripherals. The TCB can be used to perform security-related tasks, such as system software integrity checking, secure software upgrades, cryptographic key management, etc.

References: [GS05], [AF04].

References

[AF04] Tiago Alves and Don Felton. TrustZone: integrated hardware and software security. ARM whitepaper, 2004.

[Fur00] Steve Furber. *ARM System-on-Chip Architecture*. Addison-Wesley, 2nd edition, 2000.

[GS05] John Goodacre and Andrew N. Sloss. Parallelism and the ARM instruction set architecture. *IEEE Computer*, 38(7):42–50, 2005.

[Lev05] Markus Levy. The history of the ARM architecture: From inception to IPO. *ARM IQ*, 4(1), 2005.

[Sea00] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley, 2nd edition, 2000.

[WH00] Adam Wiggins and Gernot Heiser. Fast address-space switching on the StrongARM SA-1100 processor. In *Proceedings of the 5th Australasian Computer Architecture Conference*, pages 97–104, Canberra, Australia, January 2000. IEEE CS Press.