Performance of the ARM9TDMI™ and ARM9E-S™ cores compared to the ARM7TDMI™ core

1. Introduction

By using more transistors to implement a more sophisticated design, the ARM9TDMI and ARM9E-S cores provide over twice the performance than the ARM7TDMI and ARM7TDMI-S cores, when compared on the same silicon process. The performance improvements come from a combination of increased clock frequency and decreased number of clock cycles to execute some frequently occurring instructions[1].

For brevity this document will use the term “the seven cores” to refer to the ARM7TDMI and ARM7TDMI-S cores collectively, and the term “the nine cores” to refer to the ARM9TDMI and ARM9E-S collectively.

2. Clock Frequency Improvement

The increased clock frequency of the nine cores comes from a 5-stage pipeline design, compared to a 3-stage pipeline design in the seven cores (see figures 1 to 3). Increasing the number of pipeline stages increases the amount of parallelism in the design, and reduces the amount of logic which must be evaluated within a single clock period. With a five stage pipeline design, the processing of each instruction is spread across 5 (or more) clock cycles and up to 5 instructions are being worked on during any one clock cycle. The maximum clock frequency of the ARM9TDMI core is generally in the range 1.8 to 2.2 times the clock frequency of the ARM7TDMI core when compared on the same silicon process. The range is due to differences in silicon processes.

3. Cycle Count Improvement

Cycle count improvements give increased performance, independent of clock frequency. The amount of improvement depends on the mix of instructions in the code being executed, which is affected by the nature of the program, and for high level languages, by the compiler used. Programs studied by ARM typically show a performance improvement of around 30%, although this can vary significantly.

3.1 Loads and stores

The most significant improvement in instruction cycle count moving from the seven cores to the nine cores is the performance of load and store instructions. Reducing the number of cycles for loads and stores gives a significant improvement in program execution time as typically around 30% of instructions are loads or stores. The reduction is cycles for loads and stores is achieved by the two fundamental micro-architectural differences between the designs:

- The nine cores have separate instruction and data memory interfaces, allowing the CPU to simultaneously fetch and instruction and read or write a data item. This is called a modified-Harvard architecture. The seven cores have a single memory

[1] The ARM9E-S core reduces the number of cycles even further by introducing new instructions designed to allow efficient coding of DSP algorithms which use 16-bit fixed point data, and those which use saturating arithmetic. This document concentrates on common features of the ARM9TDMI and ARM9E-S cores, and does not describe the new DSP instructions which are described elsewhere.

interface which is used for both instruction fetches and data accesses.

- The five-stage pipeline introduces separate “Memory” and “Write Back” stages. These are used to access memory for loads or stores, and to write results back to the register file.

Together, these allows load and store instructions to complete in a single cycle, as is explained below the pipeline diagrams (figures 1 to 3).

Table 1 summarises the cycles taken to execute various load and store instructions. The table shows that all store instructions take one cycle less on the nine cores than on the seven cores. It also shows that load instructions generally take two less cycles on the nine cores, if there are no interlocks.

Table 1 Load and Store Cycle Counts (simple cases)

Instruction[2] type	ARM7TDMI and ARM7TDMI-S		ARM9TDMI and ARM9E-S	
	Execute Cycles	Interlock cycles	Execute Cycles	Interlock cycles
LDR load one word	3	0	1	0 or 1
LDRH (one halfword) LDRB (one byte) LDRSB (one signed byte) LDRSH (one signed halfword)	3	0	1	0 to 2 *An extra interlock cycle can occur because these instructions must rotate the data to the correct position after it is loaded.*
LDM of *n* registers (load multiple words)	*n*+2	0	*n* if loading > 1 register.	0 or 1
STR (store one word)	2	0	1	0
STRH (store one halfword) STRB (store one byte)	2	0	1	0
STM (store multiple words)	*n*+1	0	*n*	0

[2] For simplicity we ignore loads into the program counter, accesses which abort, etc., and we assume zero wait state memory, or equivalently, the cycle counts for a cache hit. For cycle counts for the more complex scenarios, see the technical reference manual for the ARM processor in question.

3.2 Interlocks

Pipeline interlocks occur when the data required for an instruction is not available due to the incomplete execution of an earlier instruction. When an interlock occurs, the hardware stalls the execution of an instruction until the data is ready. This provides complete binary compatibility with earlier ARM processor designs, however it increases the execution time of the code sequence by a number of *interlock* cycles. Compilers and assembler-code programmers can in many cases reduce the number of interlock cycles by re-arranging the order of instructions and other techniques. For example, consider the code sequences A, B, and C

```
A:
      LDR   R0, [R1]   ; load R0 from the address contained in R1
      ADD   R2, R3, R4 ; add R3 and R4 and put the result in R2
      SUB   R5, R6, R7 ; R5 = R6 – R7

B:
      LDR   R0, [R1]   ; load R0 from the address contained in R1
      ADD   R2, R3, R0 ; add R3 and R0 and put the result in R2
      SUB   R5, R6, R7 ; R5 = R6 – R7

C:
      LDR   R0, [R1]   ; load R0 from the address contained in R1
      SUB   R5, R6, R7 ; R5 = R6 – R7
      ADD   R2, R3, R0 ; add R3 and R0 and put the result in R2
```

In code sequence A the three instructions all use different registers. There are not data dependencies between them, and they all execute in a single cycle with no interlocks, giving a execution time for the sequence of 3 cycles. The LDR instruction spends one cycle in the execute stage of the pipeline. In the second cycle, the LDR instruction moves to the Memory stage of the pipeline to load the data from the memory system (which in most systems is a data cache). Also in the second cycle, the ADD instruction moves into the execute stage. In the third cycle the SUB instruction enters the execute stage. The code sequence takes three cycles to execute.

In code sequence B the ADD instruction uses R0 which is the register being loaded by the LDR instruction. This introduces a dependency between the two instructions – the ADD instruction cannot execute until the data has returned from the load. During cycle 1, the LDR instruction is in the execute stage, where it calculates the address to load the data from. In cycle two the LDR instruction moves to at the end of cycle 2, where it reads the data from the memory system. The data is returned at the end of cycle 2. Since the ADD instruction cannot execute until the data is returned, it cannot execute in cycle 2, so it is stalled for one interlock cycle. The ADD instruction enters the execute stage in cycle 3, and the SUB instruction enters the execute stage in cycle 4. The code sequence takes four cycles to execute.

Code sequence C shows how instructions can be re-arranged to avoid interlock cycles. In this case there is a useful instruction which can be moved between the LDR and the ADD instructions without modifying the resulting program behaviour. This means that the LDR can execute in cycle 1, the SUB in cycle 2, and the ADD in cycle 3, executing the complete sequence in 3 cycles.

ARM's compilers implement *code scheduling* optimisations to reduce the number of interlock cycles. It is often possible to find a useful instruction to move between the load and the subsequent use, but not always. This means that the average number of cycles to execute a LDR is a number between 1 and 2. The exact number depends on the code being compiled, and the sophistication of the compiler. Some examples are presented at the end of this document.

3.3 Branches

Many people ask whether the number of cycles for a branch instruction executed on the nine cores is larger than the number of cycles on the seven cores. The answer is no, they take the same number of cycles. This is because the pipelines have the same number of stages up to the end of the execute stages, and branches are implemented in the same way on all of the cores discussed in this document. The cycle counts are:

	ARM7TDMI and ARM7TDMI-S	ARM9TDMI and ARM9E-S
Branch Taken (passes its condition code check)	3	3
Branch Taken (fails its condition code check)	1	1

The ARM9TDMI and ARM9E-S cores do not implement branch prediction, because branches on these CPUs are fairly inexpensive in terms of lost opportunity to execute other instructions. This means that the cost of logic to implement branch prediction, and the resulting die size increase, is not justified by the performance improvement that would be gained.

4. Pipeline designs

This section explains how loads and stores are implemented on the pipelines of the ARM7TDMI, ARM7TDMI-S, ARM9TDMI, and ARM9TDMI-S cores.

4.1 ARM7TDMI and ARM7TDMI-S

The ARM7TDMI and ARM7TDMI-S CPU cores implement the 3-stage pipeline design show in figure 1 below. In a single cycle, the Execute stage can read operands from the register bank, pass them through the Shift Register, pass them though the Arithmetic and Logic unit (ALU) and write the results back to the Register bank.

Doing all of these operations in a single clock cycle simplifies the design, leading to the low power, transistor count, and die size of the ARM7TDMI and ARM7TDMI-S CPU cores. It also limits the maximum clock frequency of the design.

Data reads from the memory system and writes to the memory system are also performed in the execute stage. To do this the instruction stays in the execute stage of the pipeline for multiple cycles as follows.

LDR	execute stage cycle 1	Calculate the load address + fetch an instruction to be executed later.
	Execute stage cycle 2	Read the data from the memory system
	Execute stage cycle 3	Rotate the data if necessary, and write to register

LDM is like LDR but with cycle 3 repeated for each additional register loaded.

STR	execute stage cycle 1	Calculate the store address + fetch an instruction to be executed later.
	Execute stage cycle 2	Write the data to the memory system

STM is like STR but with cycle 2 repeated for each additional register stored.

4.2 ARM9TDMI and ARM9E-S

The ARM9TDMI and ARM9E-S CPU cores implement the 5-stage pipeline designs show in figures 2 and 3 below. The pipeline designs are the same, except that the ARM9E-S core implements a more sophisticated pipelined multiplier-accumulate unit to execute the new DSP enhancements present in the ARMv5TE instruction set. They also have a Harvard architecture, so that data accesses do not have to compete with instruction fetches for the use of one bus. “Result forwarding” is also implemented, so that results from the ALU and data loaded from memory to be fed back immediately to be used by the following instructions – this avoids having to wait for results to be written back to register bank and read from the register bank.

In these pipeline designs, dedicated pipeline stages have been added for Memory access and for writing results back to the register bank. Also, register read has been moved back into the decode stage. These changes allow higher clock frequencies by reducing the maximum amount of logic which must operate in a single clock cycle.

LDR the clock frequency pipeline In a single cycle, the Execute stage can read operands from the register bank, pass them through the Shift Register, pass them though the Arithmetic and Logic unit (ALU) and write the results back to the Register bank.

Doing all of these operations in a single clock cycle simplifies the design, leading to the low power, transistor count, and die size of the ARM7TDMI and ARM7TDMI-S CPU cores. It also limits the maximum clock frequency of the design.

Loads from the memory system and stores to the memory system are also performed in the execute stage. To do this the instruction stays in the execute stage of the pipeline for multiple cycles as follows.

LDR	execute stage cycle 1	Calculate the load address
	memory stage cycle 2	Read the data from the memory system
	writeback stage cycle 3	Rotate the data if necessary, and write to register

LDR uses the execute stage for only one cycle, allowing other instructions to use the execute stage in the following cycles (unless there are interlocks, which are explained above). This means LDR is a single cycle instruction.

STR	execute stage cycle 1	Calculate the store address
	memory stage cycle 2	write the data to the memory system

Again, as STR only uses the execute stage for a single cycle it is a single cycle instruction.

Figure 1 : The ARM7TDMI core and ARM7TDMI-S core pipeline

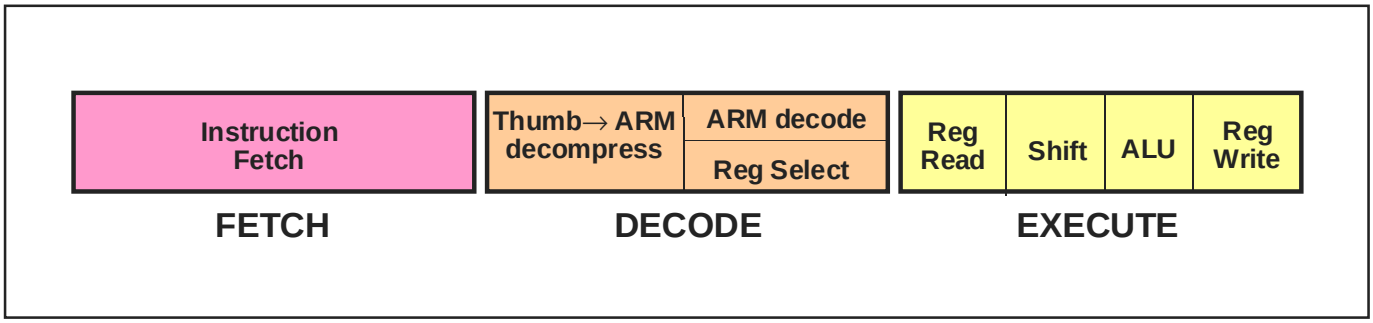


Figure 2 : The ARM9TDMI core pipeline

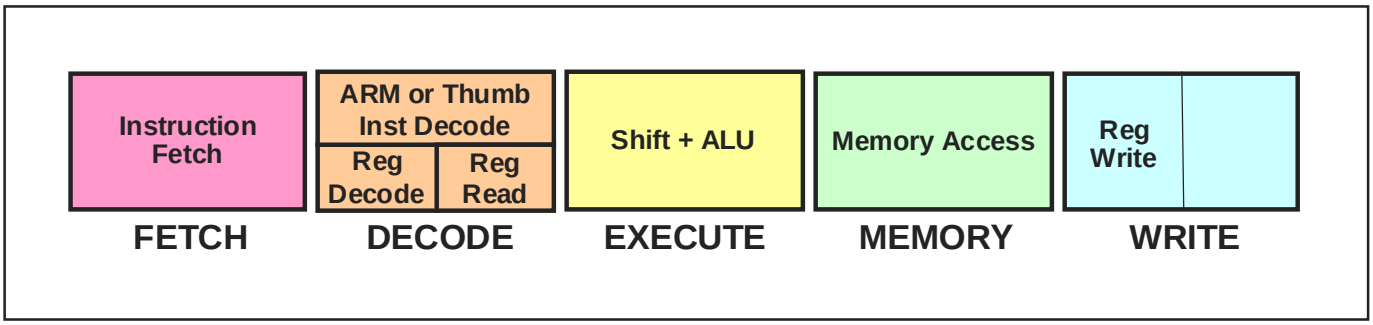
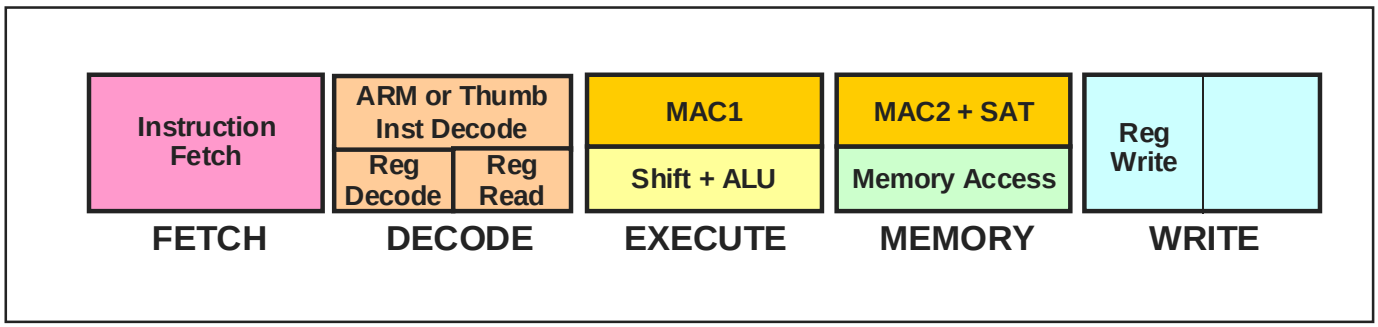


Figure 3 : The ARM9E-S core pipline



5. Performance Improvement Examples

The performance improvement moving from the seven cores to the nine cores depends on the way the software uses the ARM instruction set. Code with a large percentage of loads and stores will see a much larger improvement than code with a small percentage of loads and stores. Analysis of two pieces of code are given below as examples.

A spreadsheet showing the data and calculations to achieve these results is show on the next page.

5.1 EPOC version 4 boot sequence

Trace analysis of booting version 4 of Symbian's EPOC operating system shows that ARM9TDMI takes only 79% as many clock cycles as ARM7TDMI to execute the code sequence. This is equivalent to a 27% performance increase, assuming that both cores were running at the same clock frequency.

This version of EPOC was compiled with the gnu C++ compiler for a machine with an ARM7 processor core, and does not have optimizations to avoid interlocks.

5.2 The Dhrystone 2.1 Benchmark

The Dhrystone 2.1 benchmark is a synthetic benchmark program written in C. The version used has ANSI C function declarations and was compiled with ARM's ADS 1.0 C compiler, which performs code scheduling optimizations. Analysis of a trace of executed instructions shows that ARM9TDMI gives a 27% performance increase over ARM7TDMI at the same clock frequency. The spreadsheet shown on the next page shows that the contributions of different instructions to this performance improvement are quite different to the previous example.

CPI Comparison of ARM7TDMI and ARM9TDMI

EPOC Version 4 boot sequence
EPOC is compiled with the GNU compiler.

Information from trace analysis on ARM9TDMI

	% of instructions	average length	ARM7TDMI cycles	ARM9TDMI cycles	
Single Loads	24.7%		3	1.66	average including interlocks
Single Stores	3.5%		2	1	
LDM	3.8%	3.1	*x + 2*	*x*	
STM	3.7%		*y + 1*	*y*	
			Overall ARM9TDMI CPI	1.76	

So on average 100 instructions on ARM9TDMI take 1.76 * 100 =				176	cycles
For single loads we saved	1.34	cycles x	24.7 instructions =	33.1	cycles
For single stores we saved	1	cycle x	3.5 instructions =	3.5	cycles
For LDMs we saved	2	cycles x	3.8 instructions =	7.6	cycles
For STMs we saved	1	cycle x	3.7 instructions =	3.7	cycles

So result is	223.9	cycles in ARM7TDMI
Which is a CPI of	2.239	
So ARM9TDMI takes	79%	of the cycles ARM7TDMI takes
So ARM9TDMI performance is	127%	of ARM7TDMI performance at the same clock frequency

Dhrystone 2.1 Synthetic Benchmark
Dhrystone 2.1 compiled with the ARM ADS 1.1 compiler which performs code scheduling optimizations to minimize interlocks.

Information from trace analysis on ARM9TDMI

	% of instructions	average length	ARM7TDMI cycles	ARM9TDMI cycles	
Single Loads	17.6%		3	1.24	average including interlocks
Single Stores	10.5%		2	1	
LDM	2.3%	3.8	*x+2*	*x*	
STM	2.0%		*y+1*	*y*	
			Overall ARM9TDMI CPI	1.76	

So on average 100 instructions on ARM9TDMI take 1.76 * 100 =				176	cycles
For single loads we saved	1.76	cycles x	17.6 instructions =	31.0	cycles
For single stores we saved	1	cycle x	10.5 instructions =	10.5	cycles
For LDMs we saved	2	cycles x	2.3 instructions =	4.6	cycles
For STMs we saved	1	cycle x	2 instructions =	2	cycles

So result is	224.1	cycles in ARM7TDMI
Which is a CPI of	2.241	
So ARM9TDMI takes	79%	of the cycles ARM7TDMI takes
So ARM9TDMI performance is	127%	of ARM7TDMI performance at the same clock frequency