

# A micro-architectural enhancements study and comparison for latest ARM families

Ashraf El-Antably, *MSc. 1<sup>st</sup> year, ALaRI*  
 Pranav Tendulkar, *MSc. 1<sup>st</sup> year, ALaRI*

**Abstract**—Microprocessors nowadays have become the dominant technology used for embedded system. Due to increasing need of shortening time-to-market and of re-use, programmable architectures are used extensively. Due to also the wide spread of pervasive computing and the emerging new applications, microprocessors are getting more and more complex from architectural and micro-architectural point of view to cope with applications. Micro-architecture is every detail about implementation of the processor and because of the recent race for the highest achievable performance, micro-architecture share of responsibility has been being bigger and bigger. In this work different ARM families are going to be investigated spotting the light on the enhancements and improvements.

## I. INTRODUCTION

This work aims at studying micro-architectural enhancements in ARM Cortex, ARM11, ARM9 and ARM7 families through somehow detailed comparison between them. This work is managed such that the microarchitecture implementation for these various families are explained trying to spot the light on why these enhancements have taken place with necessary explanation of how they contribute to better performance in the application field to which every family has been tuned. The micro-architectural study is carried out in this report focusing mainly on the topics such as Instruction Pipelining, Cache Organization, Branch Prediction, and the Processor Bus Architecture.

## II. PIPELINING

There are quite large differences between different ARM families in their pipelines and this is, of course, depending upon each family targeted performance and efficiency. This section briefly spots the light on important differences between pipelines of different processor families and sometimes between different variants pipelines within the same family. ARM7, ARM11 and ARM Cortex are the families of interest for this section. This section expands every stage of the pipeline spotting the light on how every family implements every stage and how it deals with instructions. An overview of the pipelines of every family is expanded first so as to give notion about the enhancements and modifications for every family.

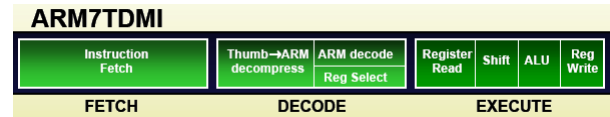


Fig. 1. ARM7TDMI pipeline

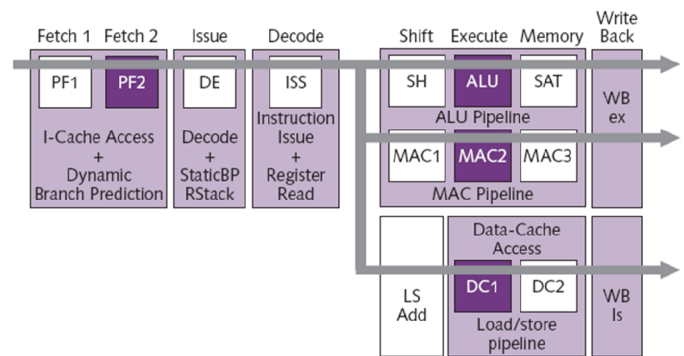


Fig. 2. The ARM11 pipeline has eight stages. Stages highlighted in the darker color were added to permit the cores to achieve higher clock speeds.

ARM7 family of processors implements the old ARMv4 architecture and some variants implement ARMv4T architecture. ARM7TDMI is one core of ARM7 family; its pipeline is composed of just 3 stages (IF, ID, IE) as shown in figure 1. While ARM11 family which implement ARMv6 architecture, has got more complex pipeline than that of ARM7 composed of 8 stages as shown in figure 2. Finally Cortex family implements different pipelines according to the variant. ARM Cortex-A8 processor which is a variant of Cortex-A series of Cortex family, has got a complex pipeline with 13 stages forming 3 decoupled parts which are (fetch, decode, execute)[1]. ARM Cortex-M3 processor which is a variant of Cortex-M series of Cortex family, has got 3-stage pipeline based on Harvard architecture[2]; its pipeline is shown in figure 3.

### A. Instruction Fetching Stage

In ARM7TDMI, the instruction is fetched trivially from memory and placed in the instruction pipeline. In ARM11 fetch stage is split into 2 sub-stages. The first one is for

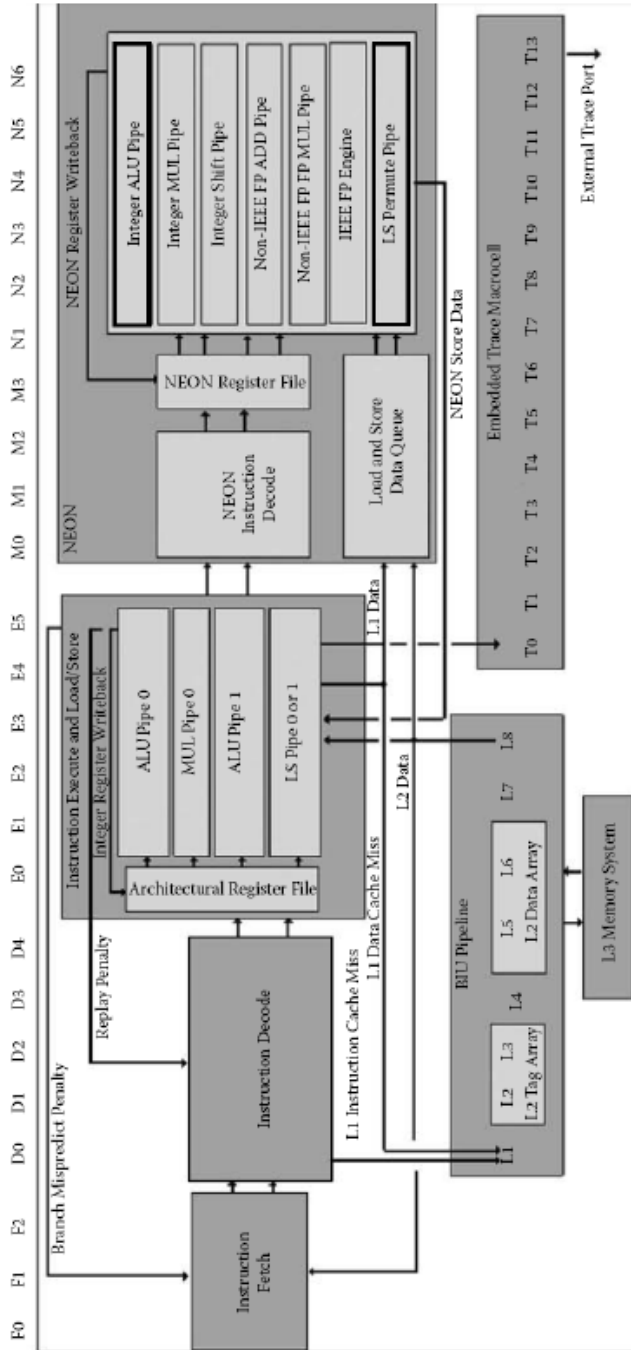


Fig. 3. ARM Cortex-A8 full pipeline

fetching the instruction from the cache and accessing BTB (Branch Target Buffer) for branch prediction. The other stage is used for storing the fetched instruction from the cache (assuming hit) in instruction queue for future consumption.

In ARM Cortex-A series of Cortex family, Fetch stage is split into 3 sub-stages unlike to ARM11 and ARM7, This

is because of need for one sub-stage for virtual address generation and translation. The fetch pipeline begins with the F0 stage where a new virtual address is generated; this stage has AGU (Address Generation Unit). This address can either be a branch target address provided by a branch prediction for a previous instruction, or if there is no prediction made this cycle, the next address will be calculated sequentially from the fetch address used in the previous cycle. Note that the F0 fetch stage is not counted as an official stage in the 13-stage main integer pipeline. This is because ARM processor pipelines have always counted stages beginning with the instruction cache access as the first stage[1]. Cortex also can fetch 4 instructions per cycle, unlike to ARM11 which can fetch only 2 instructions per cycle.

### B. Instruction Decoding Stage

Trivially in ARM7 the instruction is decoded and the datapath control signals prepared for the next cycle. In this stage the instruction 'owns' the decode logic but not the datapath. Decompression for Thumb instructions takes place in decode stage. In ARM11 implements the decode operation in one stage just prior to ISS (Instruction Issue) stage. In decode stage if the instruction is branch its branch is predicted statically through accessing static Branch Prediction Register Stack, then the instruction is issued in ISS. ARM11 issues only single instruction to ISS stage unlike ARM Cortex which is superscalar dual issue processor. In Cortex, The logic of the decode unit occupies the D0D4 stages of the pipeline; unlike ARM11 and ARM7, decode operation in Cortex is quite complex. The decoding is performed as follows. Early instruction decoding is done in pipeline stages D0 and D1. Multi-cycle instructions are broken down by the sequencer into multiple single-cycle operations (micro-ops) in the D1 stage. This is to increase performance. Unlike to ARM Cortex family, other families like ARM11 and ARM7 families still deal with multi-cycle instructions. These early stages generate all the decode information that is needed by the issue logic which is located in D3 stage of the pipeline. The D2 stage is used to write instructions into and to read from the pending/replay queue structure. The instruction scheduling logic operates in D3. The scoreboard is read in this stage for all the operands of the next two instructions that could issue. These instructions are read from the pending queue or directly from the D2 stage if the pending queue is empty. The two instructions are also cross-checked against each other to check for any other dependency hazards that would not be detected by the scoreboard. The cross-checks and scoreboard results are combined to determine whether 0, 1, or 2 instructions will be issued. Once this issue decision is made and the next set of instructions is issued across the D3/D4 boundary, these instructions cannot be stalled. After this point, instructions advance one pipeline stage per cycle and the replay mechanism will be used to handle any unpredictable hazards from the

memory system (cache miss, store buffer full, etc.). The D4 stage performs the final decode for all the control signals required by the I-execute and load-store units. These are then registered and sent to the I-execute and load-store units in E0[1].

### C. Instruction Execution Stage

After decode stage the pipelines either is split to more than one pipeline or remains as it is like in ARM7 family. In ARM7 family, the execution stage which comes directly after the decode stage, includes the ALU part in addition to the Write Back part in addition to memory access part also and since ARM7 execution is in-order execution, there is no need for Commit stage; just Write Back is sufficient. In ARM11 the pipeline is split into 3 pipelines after the decode stage unlike ARM7 although it is superscalar; one is for Load-Store (LS) instructions executions, other is for multiply-accumulate (MAC) and the last is for ALU. This leads to that although the pipeline of ARM11 is single issue scalar core; parallelism is exploited in the back end of the pipeline architecture[3]. In ARM Cortex-A series, the pipeline is split into 5 pipelines; 2 are for ALU pipe 0 or 1, another 2 are for LS (Load/Store) pipe0 or 1 and the last is for multiplication. The integer execution unit is pipelined across the E0 to E5 stages. It is responsible for doing the full execution of all ARM data processing, multiply, and traditional branch instructions. It is also responsible for maintaining the program counter, resolving condition codes for conditional instructions, generating addresses for load/store instructions, and prioritizing all potential pipeline flushes due to exceptions, branch miss-prediction, or a memory replay. NEON2 data processing instructions pass straight through the execution pipeline and are passed into the NEON instruction queue after the E5 stage.

### D. Load Store Stage

Both ARM Cortex and ARM11 families have got nearly the same pipeline structure for LS pipeline. Both have DC1 and DC2 sub-stages in their pipelines ARM7 family unlike those late families has got the memory access part in the execution stage.

### E. NEON media processing engine

This part is related to the NEON pipelines which is a new feature supported only by ARM Cortex. In the following a brief idea is given for this new technology.

1) *Pipeline Overview*: The NEON unit processes the advanced SIMD instruction set that consists of 32-bit SIMD integer and floating-point instructions that can operate on 128-bit, 64-bit, 32-bit, 16-bit, or 8-bit data values. The NEON media processing engines pipeline starts at the end of the main integer pipeline. As a result, all instruction speculation

is resolved before instructions reach the NEON pipeline. This reduces the complexity of the NEON Unit and also allows for a zero-cycle load-use penalty in most cases, even when data is returned from the L2 cache, due to the decoupling buffers used to hold pending instructions and data. NEON has four decode stages, M0M3, and six execute stages, N1N6. The four decode stages in M0M3 are very similar in structure and design to the four decode stages D0 D4 seen in the main pipeline. The first two stages are used to decode the instruction resource and operand requirements and the later two stages are used for instruction scheduling. A static scoreboard with a fire-and-forget issue mechanism is used for the NEON pipeline in a similar way to what is seen in the ARM integer pipeline with the key difference being no requirement for a replay queue because there are no conditions under which a pipeline flush can occur. NEON has three SIMD integer pipelines, a load store/permute unit, two SIMD single-precision floating-point pipelines, and a non-pipelined IEEE compliant floating-point engine. All NEON and floating-point instructions are processed by one or more of these execution pipelines which are NEON Integer Pipelines, NEON Load-Store/Permute Pipeline, NEON Floating-Point Pipelines and IEEE-Compliant Floating-Point Engine. Just for the sake of not making the report so lengthy, it was enough to mention the pipelines without going to further details for each.

## III. BRANCH PREDICTION

The throughput and performance gain added by applying the idea of pipelining instruction execution are sometimes threatened by possibilities of pipeline flushing due to control hazards. Here branch prediction comes into play with considerable importance of avoiding such case from happening frequently. Various branch predictions schemes are available trying to give as low miss-predictions as possible. This section displays how branch prediction is implemented in various ARM families so as to avoid processor stalling times which are not just undesirable for the sake of performance and throughput but also undesirable from energy point of view because of the power consumed for executing useless instructions. Here branch prediction techniques adopted by ARM families are explained as follows starting from former to later ones.

### A. ARM7 and ARM9 families

ARM7 family of processors does not use any branch prediction scheme. Like ARM7, neither ARM9 nor ARM9E family implements branch prediction [4].

### B. ARM11 family

The ARM11 micro-architecture uses two techniques to predict branches. First, the dynamic branch predictor uses a historical record to see whether the branch has been seen before, and whether it was most-frequently taken, or most

frequently not taken. A 64-entry, 4-state branch target address cache (BTAC) is maintained. This table is sufficient to hold the majority of most recent branches those that are most likely to be seen again. If the branch prediction has been encountered before, a prediction is made based on the previous outcome. If the dynamic branch predictor cannot find a record of the branch instruction, a static branch prediction procedure takes over. Very simply, the static prediction looks at the branch to see whether it is going backwards or forwards. If backwards, it assumes its a loop, and takes the branch. If its a forward branch, it doesnt take the branch. The return stack manages branch prediction for returns from up to three procedure calls. Similarly, these are treated as conditional branches. As well as predicting branches, the ARM pipeline will also fold3 the branches. This means that if the result of the branch prediction is that the branch wont be taken; the original branch instruction is removed or folded from the pipeline, as there is no point in executing it. This saves a clock cycle for a correctly predicted branch[3].

### C. ARM Cortex family

1) **ARM Cortex-A8 processors:** The 13-stage pipeline was selected to enable significantly higher operating frequencies than precious generations of ARM micro-architectures. Note as stated in previous section that stage F0 is not counted because it is only address generation. As stated in the introduction of this section, to minimize the branch penalties typically associated with a deeper pipeline, the Cortex-A8 processor implements a twolevel global history branch predictor. It consists of two different structures: the Branch Target Buffer (BTB) and the Global History Buffer (GHB) which are accessed in parallel with instruction fetches. The BTB indicates whether or not the current fetch address will return a branch instruction and its branch target address. It contains 512 entries. On a hit in the BTB a branch is predicted and the GHB is accessed. The GHB consists of 4096 2-bit saturating counters that encode the strength and direction information of branches. The GHB is indexed by 10-bit history of the direction of the last ten branches encountered and 4 bits of the PC [5]. The only flaw to global history type of prediction is that it is possible to alias on two similar histories that differ in the nth branch where n is the number of history bits + 1. To help prevent this type of aliasing, low-order instruction address bits are also used to index the GHB. The GHB has 4096 entries, but is organized as a 256 entry by 32-bit array. So, only the upper eight bits of history are used to access the array and the final indexing based on remaining history and low-order PC bits is done after the array is accessed. Each access to the GHB reads out 16 two-bit prediction values, each of which indicates whether the next branch should be predicted taken or not taken. The 16 values are multiplexed down to a single prediction using an XOR combination of the remaining history bits and the low-order PC bits. GHB accesses always return a valid value and

therefore there is no concept of a GHB miss. Instead, the GHB prediction is qualified by a hit from the BTB. To save power, the GHB is only accessed when the global history has changed. Because branch history is only updated on the prediction of a branch, the GHB array is not accessed any more often than necessary[1]. In addition to the dynamic branch predictor, a return stack is used to predict subroutine return addresses. The return stack has eight 32-bit entries that store the link register value in r14 (register 14) and the ARM or Thumb state of the calling function. When a return-type instruction is predicted taken, the return stack provides the last pushed address and state[5].

2) **ARM Cortex-R4(F) processor:** This processor use dynamic branch prediction enabled by branch target, an eight bit global branch history scheme is used and a return stack to allow the correct prediction of function return addresses. This scheme provides good accuracy without the need for a large cache of previous branch outcomes[6].

### D. Analysis of branch prediction design choices

It is noticed that for simple processor families like ARM7, ARM9 and ARM9-E which are dedicated for low end applications, branch prediction is not even used at all. This is because branches on these processors 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]. This, of course, results in a pipeline flush each time a branch is taken[6]. This also might be seen that the performance lose due to this is not that much as the pipeline in the case of ARM7 is just composed of 3 stages, for instance. On the other hand, ARM11 and Cortex families which are dedicated to more complex applications covering wide range of fields, implement branch prediction schemes. For ARM11, the net benefit of the dynamic and static branch prediction employed in the ARM11 micro-architecture is that around 85processor clock cycles for every correctly-predicted branch[3]. Cortex family, like ARM11 family implements branch prediction using branch history. This is make prediction works quite well as heuristically instruction traces tend to take similar paths through a program creating different histories that predict the outcome on the next subsequent branch. Finally for Cortex family, the net benefit of Cortex-A8 and Cortex-R4 (F) processors branch prediction techniques is reflected in their achievements, 95% [7] and 90% [6] of the branches are predicted correctly in case of Cortex-A8 processor and in case of Cortex-R4 (F) processor respectively.

## IV. CACHE ARCHITECTURE

### A. ARM7 family

ARM7TDMI processor core is based mainly on ARMv4 architecture which is Von Neumann architecture with a single 32-bit data bus carrying both instructions and data. In

ARM7TDMI processor, the cache and MMU functions are managed by System Coprocessor interface CP15[8]. There are few CPU cores that are built on ARM7TDMI processor. ARM710T and ARM720T have unified instruction and data L1 virtually addressed cache, while ARM740T implements 4KB or 8KB physically addressed cache. The unified cache automatically adjusts the proportion of the cache memory used by instructions according to the current program requirements, giving a better performance than a fixed partitioning. To permit tasks to have their own virtual memory map and lessen the eviction, the MMU hardware performs address relocation, translating the memory address output by the processor core before it reaches main memory. It is done with the help of Process Identifier Register which is unique for every task. The cache can be enabled with the registers in MMU and hence, MMU has to be enabled in order to use the Cache. This processor family supports a special read-lock-write instruction for which the cache has a special behavior and is used to implement the semaphores. ARM740T supports partial locking of instructions or data in the cache in order to ensure high performance. The cache also can be operated in split instruction data mode. This forces instructions and data to be cached in separate banks of the cache. This can be used to improve performance where a small code set is processing a large data set. The split nature of the cache prevents the data from replacing the cached instructions. The cache uses a write-through strategy as the target clock rate is only a few times higher than the rate at which standard off-chip memory devices can cycle and the write-back mode is introduced in future architectures. The processor is stalled for the read accesses, and can proceed using the Write Buffer in case of Write accesses. The Cache replacement algorithm is random.

### B. ARM11 family

In comparison, ARM11 processor family is based on ARMv6 architecture which has four variants. The ARM11 family has a memory hierarchy of L1 and L2 Cache which was introduced in ARMv5 architecture[8]. They are virtually indexed and physically addressed. Based on Harvard architecture this family has separate instruction and data caches which allow load and store instructions to execute in a single clock cycle. The cache sizes are configurable with sizes in the range of 4 to 64KB. Both the Instruction Cache and the Data Cache are capable of providing two words per cycle for all requesting sources. This combined with pipeline features leads to efficient operation of LDM and STM instructions, which loads or stores all the registers, can load 2 registers in one cycle as contrast to  $(n+2)$  cycles for  $n$  registers in ARM7TDMI. The advantage of virtual indexing is that, the address is available before the physical translation takes place, which gives an improvement in speed to check if the data exists in Cache. The tag is obtained from the MicroTLB after a translation. With the higher clock frequency in ARM11, to avoid a critical

path from the Tag RAM comparison to the enable signals for the data RAMs, there is a minimum of one cycle of latency between the determination of a hit to a particular way, and the start of writing to the data RAM of that way. This requires the cache write buffer to be able to hold three entries, for back-to-back writes. ARM11 employs customized hardware to reduce the power consumption during the sequential reads, especially in the instruction cache. On a cache read that is sequential to the previous cache read, only the data RAM Set that was previously read is accessed, if the read is within the same cache line. The Tag RAM is not accessed at all during this sequential operation. Further, only the addressed words within a cache line are read at any time, which reduces power consumption. With the required 64-bit read interface, this is achieved by disabling half of the RAMs on occasions when only a 32-bit value is required. The L2 cache in ARM11 is coupled with Prefetch Unit through Instruction Fetch Interface. This interface is optimized for cache linefills rather than individual requests. The L2 cache controller has capability to accept multiple outstanding requests which increases the efficiency. Apart from the memory transfers, the controller implements peripheral interface which enables the data transfer to or from the private peripherals like Watchdog timer, Interrupt Controller.

### C. ARM Cortex family

Finally the Cortex core which is the latest ARM processor family has a similar organization for Cache with subtle changes for performance improvement. The Cortex-M variant targets the low power microcontroller application and does not implement on-chip Cache[9], [10]. The cache is included in Cortex-A series CPUs which are targeted for application domain. Each instruction and data cache is configurable to 16KB or 32KB size and is capable of providing two words per cycle for all requesting sources. The number of cycles required to execute an LDM or STM instruction is  $(\text{number of registers} / 2)$  or 2 cycles, and is same as ARM11. Data cache can provide four words per cycle for NEON or (Vector Floating Point) VFP memory accesses. For data/instruction cache, Preload (PLD/PLI) instruction, introduced in ARMv5 architecture, is provided which is used to preload the cache before the actual access. The Preload instructions are hints, and so implementations can treat them as NOPs without affecting the functional behavior of the device. The L2 Cache in Cortex A8 processor is a unified cache which has configurable cache size of 0KB, 128KB, 256KB, 512KB, and 1MB. The L2 cache is partitioned into multiple banks to enable parallel operations. There are two levels of banking:

- the tag array is partitioned into multiple banks to enable up to two requests to access different tag banks of the L2 cache simultaneously
- Each tag bank is partitioned into multiple data banks to enable streaming accesses to the data banks

L2 cache memory has parity checking implemented on tag arrays. There is an option of having ECC or parity checking on data arrays. If ECC support is implemented, two extra cycles are added to the L2 pipeline to perform the checking and correction functionality. In addition, ECC introduces extra cycles to support read-modified-write conditions when a subset of the data covered by the ECC logic is updated. The ECC supports single-bit correction and double-bit detection. Cortex A8 introduces new hardware, logic PLE (Pre-Load Engine) which permits movement between L2 Cache and RAM. This PLE is not the same Dynamic Memory Allocation (DMA) engine, used in previous ARM family of processors but has a similar programming interface. It has two channels for upstream and downstream. The preload engine is coupled with the address translation mechanism to perform the conversion of virtual address to physical. During transfers to or from the L2 cache RAM, if the PLE crosses a page boundary, a hardware translation table walk is performed to obtain a new physical address for that new page. All standard fault checks are also performed. If a fault occurs, the PLE signals an interrupt on error. To save the power, when processor executes WFI (Wait for Interrupt) instruction, the Preload Engine stops the transfers on all the channels and resumes them when processor is wake up by the interrupt.

## V. MEMORY MANAGEMENT

ARM provides several processors equipped with hardware that actively protects system resources, either through a memory protection unit (MPU) or a memory management unit (MMU). In ARM7 family, ARM710T and ARM720T implement MMU (Memory Management Unit)[9]. The MMU in this processor family carries out two basic functions of MMU, translation of virtual address to physical and control the memory access permissions. These functions are done with the help of TLB and table-walk hardware. The MMU supports sections of 1MB size and pages of 4KB (Small) and 64 KB (Large). The MMU supports concept of domain, where 16 regions are defined with different access control rights. ARM720T supports 1KB (Tiny) pages also in order to have higher granularity in pages. ARM740T however uses a simpler MPU (Memory Protection Unit) which is targeted for Embedded Domain[11]. The protection unit does not support virtual to physical memory address translation, but provides basic protection and cache control functionality in a lower cost form. There are also performance and power-efficiency advantages in omitting address translation hardware. The protection unit allows the ARM 4 Gbyte address space to be mapped into eight regions (possible to overlap the regions), each with a programmable start address and size and with programmable protection and cache properties. A region is a set of attributes associated with an area of memory. The minimum region size is 4 Kilobytes, the maximum 4 Gigabytes. ARM11 family implements MMU for memory

management except the ARM1156T2 variant in the family implements MPU for the same reasons for ARM740T. A new hardware logic called as TrustZone is introduced in ARM1176JZ CPU. TrustZone adds a new security domain to the existing user mode and privileged mode and has higher privileges than the Operating System. The secure zone can be entered only from the privileged mode which gives OS a chance to check for any malicious attack. The resources like peripherals, on-chip memory to be protected, are accessed in this mode. Keeping the trend of embedded processors alive, in the Cortex Family, the Cortex M3 core implements optional MPU which is same as the MPU in ARM1156T2 CPU. Cortex A-Series CPU implements MMU which is same as MMU in the ARM11 family.

## VI. TRANSLATION LOOKASIDE BUFFER

There is a difference within the TLB organization within ARM7 family. The ARM720T based on Von Neumann architecture has a single, while the other architectures[12] have two TLBs because they use Harvard bus architecture: one TLB for instruction translation and one TLB for data translation[13]. In ARM710T, TLB is a 64-entry associative cache of recently used translations which accelerates the translation process by removing the need for the 2-stage table look-up in a high proportion of accesses. It is coupled with table-walking hardware in case of a miss in TLB. ARM11 processor family introduces MicroTLB of ten entries that is implemented on each of the instruction and data sides in order to perform fast Translation. TLB incorporates two more identifiers namely ASID (Application Space Identifiers) and NSTID (Non-secure Table Identifier), which are absent in ARM7 family. For Virtual cache, TLB entries can be global, or can be associated with particular processes or applications using ASIDs. ASIDs enable TLB entries to remain resident during context switches to avoid subsequent reload of TLB entries and also enable task-aware debugging. TrustZone extensions enable the system to mark each entry in the TLB as Secure or Non-secure with the NSTID. ASID and a NSTID are used to distinguish different address mappings that might be in use. ARM Cortex A8 processor has separate instruction and data Translation Look-aside Buffers (TLBs) of 32 entries each. The translation mechanism is similar to the ARM11 processor, however this core enables a facility to enable / disable table walk hardware. If translation table walks are disabled, the processor returns a Section Translation fault for a miss in TLB.

## VII. BUS ARCHITECTURE

AMBA (Advanced Microprocessor Bus Architecture) is an open standard, master-slave architecture. The architecture has a central bus arbiter, which depending on the priorities resolves the conflicting requests and assigns the bus to the requesting

master. The ARM7TDMI processor core uses Native bus architecture which can be interfaced with AHB (Advanced High-performance Bus) architecture using a wrapper. ARM720T however uses AHB bus architecture. The ARM11 processor family uses supports AHB-Lite or AMBA AXI bus protocol. AXI is a bus protocol is a next version of AHB which supports separate address/control and data phases, unaligned data transfers using byte strobes, burstbased transactions with only start address issued, separate read and write data channels to enable low-cost DMA, ability to issue multiple outstanding addresses, out-of-order transaction completion, and easy addition of register stages to provide timing closure. The AXI protocol also includes optional extensions to cover signaling for low-power operation. AXI is targeted at high performance; high clock frequency system designs and includes a number of features that make it very suitable for high speed sub-micron interconnects. AHB-Lite is a subset of AHB bus protocol, which is required for IP development. The APB (Advanced Peripheral Bus) is a simpler, non-pipelined bus protocol than AXI and AHB and is designed for use with general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. Connection to the main system bus is through a system-to-peripheral bus bridge that helps to reduce system power consumption. The ARM Cortex Family uses Advanced High-performance Bus-Lite (AHB-Lite) for ICode, DCode and System bus interfaces. And it uses Private Peripheral Bus (PPB) based on Advanced Peripheral Bus (APB) interface to connect processor core with the peripherals.

### VIII. POWER MANAGEMENT

ARM7 does not have explicit power manager. The ARM7 core cannot be shut off; however the clock of the processor can be stretched indefinitely in either phase to allow access to slow peripherals or memory or to put the system into a lowpower state. Also EmbeddedICE-RT Debug logic can be disabled to keep power consumption to a minimum. ARM11 processor family supports four modes of power management.

- Run mode - Run mode is the normal mode of operation when all of the functionality of the core is available.
- Standby mode - Standby mode disables most of the clocks of the device, while keeping the design powered up. This reduces the power drawn to the static leakage current, plus a tiny clock power overhead required to enable the device to wake up from the standby state.
- Shutdown mode - Shutdown mode has the entire device powered down, and you must externally save all state, including cache and TCM state. The processor is returned to Run mode by the assertion of Reset.
- Dormant mode - Dormant mode (partially supported in the core) enables the core to be powered down, leaving the caches and the Tightly-Coupled Memory (TCM) powered up and maintaining their state. Transition from Dormant state to Run state is triggered by the external

power controller asserting Reset to the processor until the power to the processor is restored. When power has been restored the core leaves reset and, by interrogating the external power controller, can determine that the saved state must be restored.

ARM1176JZF processors implements Intelligent Energy Management block. This piece of logic is responsible for balancing performance and power consumption by dynamic alteration of the processor clock frequency and supply voltage. To build IEM capable chip we need extra hardware on chip like Dynamic Clock Generator, Intelligent Energy Controller, Dynamic Voltage Controller etc. ARM also provides with IEM software which can be integrated with the Operating System, puts the system in different states depending on the policies that are enforced. The Cortex Family processor extensively uses gated clocks to disable unused functionality, and disables inputs to unused functional blocks, so that only actively used logic consumes any dynamic power. The design has separated clocks for essential blocks, so clocking circuits for most parts of the processor can be stopped during sleep. The ARMv7-M architecture supports system multiple sleep modes (SLEEPING and SLEEPDEEP) that can stop the processor core and system clocks for greater power reductions. The core has a separate free running clock, which is used to run a small amount of logic to detect the interrupt in sleep state. Frequency of this clock can be reduced in the sleep state in order to maximize the power saving. The cortex core supports Wait For Interrupt (WFI) or the Wait For Event (WFE) instructions request the sleep-now model. These instructions cause the Nested Vectored Interrupt Controller (NVIC) to put the processor into the low-power state pending another exception. A Wakeup Interrupt controller (WIC) is implemented in core, which gets the processor out of the sleep state on the arrival of an unmasked interrupt. The small size of the WIC ensures that its power requirements fit the budget available while in very-deep-sleep mode enabling it to always remain powered. Another feature of sleep mode is that it can be programmed to go back to sleep automatically after the interrupt routine exit. In this way we can make the core sleep all the time unless an interrupt needs to be served.

### IX. MISCELLANEOUS

ARM processor families are always equipped with new features in order to provide an increase in the performance as compared with its predecessor family. There are some important features that have gone unmentioned in the wide arm architecture. To name them a few the features are as follows.

#### A. Jazelle

Jazelle is a java byte code acceleration hardware, which was introduced in ARMv4 architecture. By utilizing the underlying Jazelle technology architecture extensions, the ARM MVM

(Multitasking Virtual Machine) software solution delivers high performance applications and games, fast start-up and application switching with a very low memory and power budget. This is done with the help of introducing a Jazelle stage in the pipeline to decode the Java Byte Code, which can be activated in Jazelle State. Later, the technology evolved into two more variants - ARM Jazelle DBX (Direct Bytecode eXecution) technology for direct bytecode execution and ARM Jazelle RCT (Runtime Compilation Target) which supports efficient ahead-of-time (AOT) and just-in-time (JIT) compilation with Java and other execution environments.

### ***B. Unaligned and Mixed-Endian Data Access Support***

This architectural feature was added in ARMv6. It adds unaligned word and half-word load and store data access support[8]. When enabled, one or more memory accesses are used to generate the required transfer of adjacent bytes transparently, apart from a potentially greater access time where the transaction crosses a word-boundary. The original ARM architecture was designed as little-endian. ARMv6 architecture adds up the support for both endian data but the instructions are still in little-endian format. Data accesses can be either little-endian or big-endian as controlled by a bit in the Program Status Register. In addition the processor has introduced new instructions reverse the bytes in the registers.

### ***C. ARM CoreSight***

CoreSight is a debug technology which is capable of instruction and data tracing inside a core and adds a wider range of functionality and systems including the tracing of multiple ARM cores, ARM and DSP cores, complex peripherals and busses. This technology enables the real-time cycle accurate trace of entire CPU without halting the processor core.

### ***D. Vector Floating Point Coprocessor***

ARM Vector Floating-point Architecture (VFPv2) is low-cost floating-point computation coprocessor that is fully compliant with the IEEE Standard for Binary Floating-Point Arithmetic and supports single-precision and double-precision add, subtract, multiply, divide, multiply and accumulate, and square root operations[8]. This hardware provides very fast hardware execution of floating-point operations to avoid long latencies. It supports parallel hardware divide and square root operations with other arithmetic operations to reduce the impact of long-latency operations. The VFP11 coprocessor provides a performance-power-area solution for embedded applications and high performance for general-purpose applications. This co-processor has three internal pipelines a. the Multiply and Accumulate (FMAC) pipeline b. the Divide and Square root (DS) pipeline and c. the Load/Store (LS) pipeline. In the Cortex family, this technology evolved into VFPv3 architectures.

### ***E. THUMB-2 Instruction Set***

THUMB instruction set is a 16-bit instruction set that was introduced in ARMv4 architecture. The goal of this feature was to increase the code density without sacrificing the performance. In THUMB instruction set, there was a code efficiency gain at the cost of performance. This technology evolved in ARMv6 architecture. The Thumb-2 instruction set uses a combination of 16- and 32-bit instructions to deliver improved code density and performance. On average, using code benchmarked to date, the performance of Thumb-2 is about 98 percent of that benchmarked with code based on 32-bit ARM instructions. However, the memory footprint is significantly smaller, with many applications needing just 75 percent of the memory footprint previously taken up by 32-bit ARM instructions.

### ***F. Fast Context Switch Extension***

The Fast Context Switch Extension (FCSE) is additional hardware in the MMU that is considered an enhancement feature, which can improve system performance in an ARM embedded system. The FCSE enables multiple independent tasks to run in a fixed overlapping area of memory without the need to clean or flush the cache, or flush the TLB during a context switch. With the FCSE there is an additional address translation when managing virtual memory. The FCSE modifies virtual addresses before it reaches the cache and TLB using a special relocation register that contains a value known as the process ID. ARM refers to the addresses in virtual memory before the first translation as a virtual address (VA), and those addresses after the first translation as a modified virtual address (MVA). When using the FCSE, all modified virtual addresses are active. Tasks are protected by using the domain access facilities to block access to dormant tasks.

### ***G. Clock-less Processors***

The AMULET processor cores are fully asynchronous implementations of the ARM architecture, which means that they are self-timed and operate without any externally supplied clock. These processors eliminate problems regarding clock like clock skew and are highly power efficient. Their application domain is unclear at the moment since the entire industry is clock-guided. Similarly, ARM996HS is a first synthesizable clock-less processor. Brief comparison figures are shown in Table 2.

## **X. APPENDIX**

### ***A. NEON Technology***

ARM NEON technology is a 128 bit SIMD (Single Instruction, Multiple Data) architecture extension for the ARM Cortex-A series processors, designed to provide flexible and powerful acceleration for intensive multimedia applications,

thereby delivering a significantly enhanced user experience. NEON technology accelerates multimedia and signal processing algorithms such as video encode/decode, 2D/3D graphics, audio, voice and speech processing, image processing, telephony, and sound synthesis by at least 3x the performance of ARMv5 and at least 2x the performance of ARMv6 SIMD. The architecture is optimally defined and works seamlessly with its own independent pipeline and register file. The ARM Cortex-A8 processor with NEON technology is becoming widely accepted as the leading processor in multimedia applications ranging from smart-phones and mobile computing devices to HDTV[14].

## XI. ACKNOWLEDGEMENTS

The authors would like to thank Prof. Antonio Gonzalez for giving us necessary guidance through his explanation of the future architectures course for making this report and proposing it in the way how it looks.

## REFERENCES

- [1] John, Eugene, et al. Unique chips and systems. s.l. : CRC press Taylor and Francis group.
- [2] An Introduction to the ARM Cortex-M3 Processor. arm.com. [Online] ARM, October 2006. [Cited: June 11, 2009.] <http://arm.com/pdfs/IntroToCortex-M3.pdf>
- [3] Cormie, David. The ARM11 Microarchitecture. 2002.
- [4] Performance of the ARM9TDMI and ARM9E-S cores compared to the ARM7TDMI core. s.l. : ARM, 2000.
- [5] Architecture and Implementation of the ARM Cortex-A8 Microprocessor. design and reuse. [Online] [Cited: June 13, 2009.]<http://www.designreuse.com/articles/11580/architecture-and-implementation-ofthe-arm-cortex-a8-microprocessor.html>
- [6] ARM Cortex-R4(F). s.l. : Arrow Electronics ARM Solutions.
- [7] ARM Cortex-A8. arm.com. [Online] ARM, June 11, 2009. [http://www.arm.com/products/CPUs/ARM\\_Cortex-A8.html](http://www.arm.com/products/CPUs/ARM_Cortex-A8.html)
- [8] ARM1156T2F-S Technical Reference Manual. infocenter.arm.com. [Online] ARM, 2007. [Cited: June 13, 2009.]<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0290g/ch04s02s03.html>
- [9] ARM7TDMI-S Technical Reference Manual, Revision: r4p3. infocenter.arm.com. [Online] ARM. [Cited: June 13, 2009.] <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0234b/i1010871.html>
- [10] Cortex-M3 Technical Reference Manual, Revision:r2p0. infocenter.arm.com. [Online] ARM. [Cited: June 13, 2009.]<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337g/index.html>
- [11] Furber, Steve. ARM System-on-Chip Architecture.
- [12] Cortex-A8 Technical Reference Manual, Revision: r3p0. infocenter.arm.com. [Online] ARM. [Cited: June 13, 2009.]<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344i/ch01s09s12.html>
- [13] Yiu, Joseph. The Definitive Guide to the ARM Cortex-M3.
- [14] NEON Technology. arm.com. [Online] ARM. [Cited: June 13, 2009.]<http://arm.com/products/multimedia/neon/index.html>
- [15] [http://en.wikipedia.org/wiki/ARM\\_architecture](http://en.wikipedia.org/wiki/ARM_architecture)
- [16] <http://www.handshakesolutions.com>
- [17] Prof. A. Gonzalez lectures notes ALaRI 2009