Chapter 3

Architecture of various ARM core and Performance Enhancement of Selection Parameters

3.1 History and different version of ARM architecture
3.2 Architecture version and features description
3.3 Commonly used ARM core and resources
3.4 Advanced Microcontroller Bus Architecture (AMBA) Bus
3.5 Power consumption parameter
3.6 Stages of pipeline
3.7 Branch Prediction Logic
3.8 Summary and Discussion
3.1 History And Different Version Of Arm Architecture:

The term ARM derived initially from Acorn RISC Machine in 1980s developed by Acorn Computer Limited[2] at Cambridge. Later on the term Acorn replaced by Advanced and it is said as Advanced RISC Machine. Since inception of 32-bit ARM architecture different versions along with different core[2][3] were made available which is listed in Table 3.1[2].

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv1</td>
<td>ARM1</td>
</tr>
<tr>
<td>ARMv2</td>
<td>ARM 2, ARM 3, AMBER</td>
</tr>
<tr>
<td>ARMv3</td>
<td>ARM 6, ARM7</td>
</tr>
<tr>
<td>ARMv4</td>
<td>StrongARM, ARM7TDI, ARM8, ARM9TDI, FA526</td>
</tr>
<tr>
<td>ARMv5</td>
<td>ARM7EJ, ARM9E, ARM10E, XScale</td>
</tr>
<tr>
<td>ARMv6</td>
<td>ARM 11</td>
</tr>
<tr>
<td>ARMv6-M</td>
<td>ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1</td>
</tr>
<tr>
<td>ARMv7-A</td>
<td>ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15</td>
</tr>
<tr>
<td>ARMv7-M</td>
<td>ARM Cortex – M3</td>
</tr>
<tr>
<td>ARMv7-R</td>
<td>ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7</td>
</tr>
<tr>
<td>ARMv7E-M</td>
<td>ARM Cortex-M4</td>
</tr>
<tr>
<td>ARMv8-A</td>
<td>ARM Cortex-A53, ARM Cortex-A57</td>
</tr>
</tbody>
</table>

Table 3.1 ARM versions and its family cores

3.2 Architecture Version And Features Description:

- **Version 1** – (obsolete) having following features[2][3]
  - Basic data processing
  - Byte, word and multi-word load/store
  - Software interrupt
  - 26 bit address bus
- **Version 2** – (obsolete) having following features[2][3]
  - Multiply & Multiply-accumulate (MAC unit)
- Coprocessor support
- Atomic instruction for thread synchronization
- 26 bit address bus

- **Version 3** – with following improved features\[2\][3]
  - 32 bit address bus
  - Additional CPSR, SPSR
  - Addition of MRS, MSR. Modify exception handler
  - Addition of ‘Data abort mode’ and ‘undef mode’

- **Version 4** – with following improved features\[2\][3]
  - Half word transfer
  - Introduce THUMB processor state
  - Addition of ‘Privileged mode’ for operating system
  - 2 word distance of PC from current instruction
    - ‘PC+8’ behavior (at ARM state)
  - First fully formalized architecture

- **Version 5** – with following improved features\[2\][3]
  - Improve ARM/THUMB inter-working
  - Addition of CLZ instruction for efficient integer divides
  - Addition of software breakpoint
  - Addition of more coprocessor support
  - More tight definition of arithmetic flags

- **Version 6** – with following improved features\[2\][3]
  - Improved memory management
  - Multiprocessing
    - Added new synchronization instruction (LDREX, STREX)
  - Improved exception handling
    - New bit in PSR
  - Mixed endian support
  - Media extension
    - ARM SIMD (16bit 2 way and 8 bit 4 way)
    - FFT, MPEG4
    - Saturation, Selection
### 3.3 Commonly Used ARM Core And Its Resources:

Stages of pipeline, Branch Prediction, Latency because of pipeline, architecture based on memory, operating speed, Throughput and data forwarding is shown in Table 3.2[2][3].

<table>
<thead>
<tr>
<th>ARM core Parameters</th>
<th>ARM 7/ARM 7TDMI</th>
<th>ARM 9TDMI</th>
<th>STRONGARM</th>
<th>ARM 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Von-Neumann</td>
<td>Harward</td>
<td>Harward</td>
<td>Harward</td>
</tr>
<tr>
<td>Stages of pipeline</td>
<td>3-stages</td>
<td>5-stages</td>
<td>5-stages same as ARM9</td>
<td>6-stages</td>
</tr>
<tr>
<td>Throughput</td>
<td>Single cycle for almost all simple data processing instruction and multi cycle for Multiply multiple Load/Store.</td>
<td>Almost more than one instruction per cycle.</td>
<td>More than one instruction per cycle.</td>
<td></td>
</tr>
<tr>
<td>Data forwarding</td>
<td>Requires multiple cycles</td>
<td>In same cycle for most of the instructions</td>
<td>Fast compare to ARM9</td>
<td>Very Fast compare to ARM9</td>
</tr>
<tr>
<td>Branch prediction</td>
<td>More than one cycle in case of branch penalty</td>
<td>Branch target available – reduce branch penalty to 1 cycle</td>
<td>Dedicated branch predictor available</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2 showing important parameters for different cores

### 3.4 Advanced Microcontroller Bus Architecture (AMBA) Bus:

Most of the ARM core uses AMBA[2][3] bus for communication with system modules and peripherals. Important features of AMBA bus are listed below.

- Standard of on-chip communication between different macrocells for high performance embedded system design
Hierarchical Bus architecture

Figure 3.1 AMBA Bus Structure

Figure 3.1 shows how AMBA bus structure[2][3] is working for system and peripherals resources. AMBA buses are of three categories as discuss below.

3.4.1 AMBA Buses:

- **AHB (Advanced High Performance Bus)**
  - Connect between high-performance system modules
- **ASB (Advanced System Bus)**
  - Subset of AHB
- **APB (Advanced Peripheral Bus)**
  - Simple interface for low-performance peripherals

Functions[2][3] of all three buses are shown in Table 3.3

<table>
<thead>
<tr>
<th>AHB</th>
<th>ASB</th>
<th>APB</th>
</tr>
</thead>
<tbody>
<tr>
<td>burst transfers</td>
<td>burst transfers</td>
<td>low power</td>
</tr>
<tr>
<td>split transactions</td>
<td>pipelined operation</td>
<td>latched address and control</td>
</tr>
<tr>
<td>single-cycle bus master handover</td>
<td>multiple bus masters</td>
<td>simple interface</td>
</tr>
<tr>
<td>single-clock edge operation</td>
<td></td>
<td>suitable for many peripherals</td>
</tr>
<tr>
<td>wider data bus configurations (64/128 bits)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>multiple bus masters (up to 16)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipelined operation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3 Explaining the functions of three buses
3.4.2 AMBA AHB Component:

- **Master**: Initiate read and write operations by providing an address and control information. Only one bus master is allowed to actively use the bus at any one time.

- **Slave**: Responds to a read or write operation within a given address-space range. The bus slave signals back to the active master the success, failure or waiting of the data transfer.

- **Arbiter**: Ensures that only one bus master at a time is allowed to initiate data transfers. Can use the priority

- **Decoder**: Decode the address of each transfer and provide a select signal for the slave that is involved in the transfer

![Figure 3.2 Working of AHB in AMBA Bus](image)

Figure 3.2 shows the working[2][3] of AHB in AMBA bus, which comprises of Master, Slave, Decoder and Arbiter modules.

As far as performance is concern one needs to look at power consumption, Pipeline stages, Branch Prediction logic, Code Density, Compactness of the design and Support of the peripherals. To enhance the performance of the system these parameters should be optimized. Subsequent sections talks about review and enhancement of these parameters in detail.
3.5 Power Consumption Parameter:

Many embedded systems have substantially different design constraints than desktop computing applications. Power consumption and battery life is among the most critical concerns in embedded systems. No single characterization applies to the diverse spectrum of embedded systems. A less pervasive system-level issue, but one that is still common, is a need for power management to either minimize heat production or conserve battery power. While the push to laptop computing has produced "low-power" variants of popular CPUs, significantly lower power is needed in order to run from inexpensive batteries for 30 days in some applications, and up to 5 years in others. So, one can say that power consumption has become very essential parameter for enhancing the performance of any architecture. I have gone through the minute details of the power consumption sources and did survey[1] on different mechanism on minimizing the power consumption in embedded systems.

The total power consumed can broadly be divided into two categories:

- Dynamic power and
- Static power.

- Dynamic power[7] dissipation refers to the power consumed by a circuit/system due to some activities within it. For a static CMOS realization, at steady-state, either the pull-up or the pull-down network is OFF. Thus, when there are no activities in the system, power consumption is expected to be very low. During circuit activities, the power consumed depend upon the short circuit power and switching power dissipation mechanisms.

- In static power[7] dissipation the power consumed when the circuit is not in active mode of operation. In such a situation, there is still some power dissipation due to various leakage mechanisms. The situation is aggravated with the scaling of supply voltages. As the supply voltage is reduced to keep the delay of a gate unchanged, the transistors need to be turned ON early by reducing their threshold voltages. There are three major leakage mechanisms-Sub-threshold leakage, gate direct tunneling and junction band to band tunneling.
Various sources of power consumption are shown in Figure 3.3. These mainly include the dynamic and static power consumption and its sub-parts.

![Figure 3.3 Sources Of Power Consumption](image)

3.5.1 Selection Of Proper Hardware And Software For Implementation:
In a view to optimize the power consumption[4] one can look at the following criterion for implementation of hardware and software.

- **Cache memory**: Onchip cache saves the memory access time, as well as bus energy consumption.
- **On/Off chip memory**: The access pattern of memory should be sequential because only the least sequential bits of the address bus change, but in random access, most of the address bus switch, thus creating higher power dissipation.
- **Use of effective algorithm**: Number of operation, operators, variables should be reduced.
- **compiler/assembler optimization**: The design[5] of compiler/assembler include strength reduction, common subexpression elimination, minimization of memory traffic can be effectively analysed.
- **Number Representation**: Fixed point operations are much simpler than floating point as it saves power. Selection of sign-magnitude representation may have significant power saving over 2’s complement.
Selection of the lower precision allows to reduce the size of the space needed to store values, which thereby reduces the power requirements.

### 3.5.2 Selection Of Proper Architecture:

Architecture should be parallelism and pipelining. As all devices are operating simultaneously, total power saving is very higher.

Communication on buses within a chip as well inter-device wireless communication has been always a major source for power consumption. On chip bus communication becomes even more challenging in tiny-processors that does not feature advance architecture. A considerable amount of energy is consumed in onchip interconnect. The main source of power dissipation occurs when the voltage swings in communication lines. Bus coding and encoding techniques [7] can be used to reduce power consumption along with increasing the performance in terms of throughput and latency. These technique reduce voltage swings along interconnect lines, which can result in large power savings.

### 3.5.3 Hardware Implementation:

- **Selection of logic families:** CMOS logic can be realized as static or dynamic CMOS. In a dynamic CMOS the power consumption is more than that of Static CMOS [7] because of its pre-change value of the output, where as in static CMOS because of the difference in the arrival of the input glitches and hazards produced which consumes more power. So, there is trade-off between the use of Dynamic CMOS and Static CMOS. To remove the glitches and hazards in Static CMOS one can go for modification of the circuit design by adding one or two redundant gate to make the arrival of the input at the same instance.

- **PCB DESIGN:** As we know size of electronic product also depends of Printed Circuit Board (PCB), so, one can optimize the size of PCB such that we can minimize the power consumption of electronic circuits. In the PCB design to reduce power consumption we can place the component in such a way that signal track length can be minimized and as track length is reduce power consumption can be minimized.
- **Mapping methods:** In logic design basically there is tradeoff between area, propagation delay & minimum power. So, one can go for power optimization by managing delay in the circuit design. Instead of using less number of gates one can go for more number of gates to optimize power and trade-off with delay.

3.5.4 Control Logic Strategy:
- **Storage element design:** Must have minimum propagation delay. There is always a trade-off between area and power.
- **State assignment:** It should minimize the number of transitions in the next state and output combinational logic.
- **FSM (Finite state machine) division:** FSM can be partitioned in two or more sub-FSM. At any point of time, only one sub-FSM[1] is active, and clock and power gating is such that there is not dynamic power consumption in sub FSM

3.5.5 Scheduling For Power Management:
Task scheduling on single or multiple processing elements is considered as one of the most common methods to achieve lower power consumption. Scheduling saves power by shutting down devices when they are not operating. Processing elements in embedded system design serve different requests at different times. Ordering task execution adjusts the lengths of the idle periods and exploits the opportunities for power management. A suitable power management policy[1][3] is needed.

3.5.6 Low Power Data Management:
Power optimization in data management is very important. Limited supply of energy, inadequate processing power, small memory size, power hungry communication and a low bandwidth communication impose various challenges[1][7] for data management in systems which are closely linked with physical environment.
3.6 Stages Of Pipeline:

Pipeline architecture is always preferable when we want to achieve higher throughput. ARM architecture offers various cores which comprise 3 – stage pipeline to 7 – stage pipeline architecture[2][3]. It is but obvious that more the stages of pipeline gives much more rise in the throughput. Pipeline architecture creates overhead when there is branch instruction, interrupts or any other activities by which control of execution moves from current location to some different location. This situation is undesirable for better throughput. Apart from that in some situation pipeline could not moves smoothly because of pipeline stall, which is usually called as pipeline hazards. In those conditions pipelines needs to be flushed and to be reloaded with fresh set of instructions. So, one can say that more the stages of pipeline will create more overhead in terms of wastage of time & underutilizing various other resources. To avoid the pipeline hazards ideally architecture should have a mechanism by which it anticipate branch and take corrective measures in advance. This mechanism is called as Branch Prediction. Branch prediction logic is usually implemented based on history (by analyzing the history of grouping of instructions). Several types of branch prediction logic can be implemented to minimize overhead of the system[9].

3.7 Branch Prediction Logic:

For pipeline architecture it is as good as essential to have a Branch Prediction mechanism available. I have used SimpleScalar Simulator for developing Branch Prediction logic. Generally HDL is used to test the processor design which requires the lots of time and resources to design whole processor and test benches. SimpleScalar is the simulator which simulates the processor and gives performance parameters. SimpleScalar[14][15] consists of different tools to simulate the processor with different perspectives. This tool set consists of basic compiler, cross compiler, assembler, linker, simulator, benchmarks (test bench programs) and visual tools. This tool also enable user to simulate C codes which gives more flexibility than the conventional HDL design flow. SimpleScalar supports PISA, APLHA, x86, SPARC and ARM Instruction Set Architectures.
As it is open source one can design your own architecture and simulate it. New branch prediction technique (BTFNT) is developed & verified.

3.7.1 Introduction Of SimpleScalar Toolset:
The SimpleScalar tool set is a system software infrastructure used to build modeling applications for program performance analysis, detailed micro architectural modeling, and hardware-software co-verification. Using the SimpleScalar tools[14][15], users can build modeling applications that simulate real programs running on a range of modern processors and systems. The tool set includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. In addition to simulators, the SimpleScalar tool set includes performance visualization tools, statistical analysis resources, and debug and verification infrastructure. In simple words SimpleScalar is a toolset which compares two processors and give us their performance parameters without designing processors from the scratch.

3.7.2 Architecture Of SimpleScalar Toolset:
SimpleScalar tool set consists of basic compiler, cross compiler, assembler, linker, simulator, benchmarks (test bench programs) and visual tools[10]. Detail description is explained in Figure 3.4. We can simulate both C and FORTRAN source code in this tool set. For FORTRAN source code it is converted into C through f2c convertor. After that it assembled and compiled which gives you the executable. Now using different SimpleScalar Tool Set we can simulate and get performance parameters of the processor. SimpleScalar consists of following Programs to simulate: sim-fast, sim-safe, sim-cache, sim-cheetah, sim-bpred, sim-profile, sim-outorder.

- **sim-safe:** Same as sim-fast but it checks for memory exceptions. Slightly slower instruction interpreter, as it checks for memory alignment and memory access permission on all memory operations. This simulator can be used if simulated program causes sim-fast to crash without explanation.
**Figure 3.4 SimpleScalar Toolset**

- **sim-fast**: Fast instruction interpreter, optimized for speed. This simulator does not account for the behaviour of pipelines, caches, or any other part of the micro architecture. It performs only functional simulation using in-order execution of the instructions (i.e. they are executed in the order they appear in the program).

- **sim-cache**: It is used to simulate the cache stat of the processor. Cache parameters like cache size, block size, no of ways of cache, no of sets in cache given as input and hit-miss ratio we get as performance parameter. Separate instruction, data and TLB caches can be configured.

- **sim-chetaah**: Same as sim-cache but it can configured for multiple caches structure. sim-bpred: It is branch prediction logic for the simulation. It supports taken, not taken, bimod branch prediction logics.

- **sim-profile**: Instruction interpreter and profiler. This simulator keeps track of and reports dynamic instruction counts, instruction class counts, usage of address modes, and profiles of the text and data segments. Different types of profiling like instruction profiling, branch profiling, address mode profiling, segment profiling is supported in SimpleScalar.

- **sim-Outorder**: Detailed micro architectural simulator. This tool models in detail and out-of-order microprocessor with all of the bells and whistles, including branch prediction, caches, and external memory. This simulator is highly parameterized and can emulate machines of varying numbers of execution units.
Figure 3.5 shows the comparison of different simulator supported by simplescalar toolset.

### 3.7.3 Branch Prediction Technique In Simplescalar:

Branch Prediction is essential part of the pipelined based architecture. In pipeline based architecture, processor is executing an instruction and at a same time it is pre-fetching next instructions. Now when a branch comes in the execution stage if it is taken the whole instructions which are pre-fetch before must be removed from queue and start a new queue which disturbs the flow of the pipeline. Now if we able to predict the branch immediately after the fetch we can save the time and increase the performance of the processor.

SimpleScalar supports various branch prediction techniques like taken, not taken, bimod, 2level etc. Here we have implemented this technique in sim-bpred.c, bpred.c and bpred.h source codes. **New technique of Backward Taken and Forward Not Taken technique is implemented and verified in the simplescalar[9].**

As a result of Branch Prediction Technique simulator gives following performance parameters. Number of Branches, Number of Branches whose direction is predicted correctly, Number of Branches whose direction and
address both predicted correctly and Prediction parameter for the register relative jump.

A. NotTaken Branch Prediction:
In this technique we always assume that the branch which is occurred during program is not taken. In the sense we fetch the immediate next instruction of the running program. Now if we find that at the execution state of the branch operation that the branch is taken then we flush the all the prefetch data and fill it with the next jump address which is pointed by the branch. The main drawback of this technique[9] is that the all of the assembly level programs which we are designing are jump if not zero kind of programs where most of the time branch is taken instead of not taken. Due to this problem Not Taken Branch Prediction will not give the desired branch prediction for the program.

B. Taken Branch Prediction:
This technique assumes that the branch which is occurring always going to be taken. For taking up the branch we must require an addition hardware which will fetch the target address of branch and fetch the new instruction from that point only. This gives us the advantage for the assembly level programs and gives much better result than the Not Taken Prediction Technique.

C. Backward Taken and Forward Not Taken Branch Prediction:
This technique is the combination of the above two techniques. Now according to the survey, most of the loops are backward jump loops and which will be taken most of the time. Same way Forward branches are normally not taken. This will enhance the performance[9] of the taken prediction to certain extend. But for the programs which has more if else ladders will not give proper prediction for this technique. This BTFNT technique is not implemented in the SimpleScalar. By changing the source code of the SimpleScalar We have implemented this technique in Simplescalar.

D. Dynamic Branch Prediction:
The techniques which we have seen up till now is static techniques in which predict the branch compile time not run time. Now some new techniques are developed based on history table which stores the previous branch status and used it for branch prediction. 2level, bimod and gskew are the example of this type of technique.
E. Two–bit Branch Prediction:
In this technique k bit counter is runs in the table to maintain the branch history and prediction logic of the branch. From the value of that counter we can predict that whether the branch is taken branch or not taken branch. We have assumed it as two bit counter. If branch is taken it increases its value else decrease the value. At prediction time if the value of the prediction table is more than 2 than we predict the branch as taken branch else take it as not taken branch. Figure 3.6 shows 2-bit branch prediction.

F. Two–level Branch Prediction:
In this technique two levels of table are used to predict the branch. Sometimes instead of first table single register is used which is known as the Global History Register and the table is known as Branch History Table. An example diagram of the 2level branch prediction is as follow. First Table selects the PHT (Pattern History Table) using PC and previous branch predictions. After branch BHT is rotated and according to taken and
not taken of the branch it is updated. In second table we use same fundamental we are using in the 2 bit prediction. Figure 3.7 shows the same.

3.7.4 Simulation Results:

![Branch Prediction Chart]

Figure 3.7 Two – level Branch Prediction

Figure 3.8 Result Chart
A BTFNT technique is verified on different binaries of the spec95 benchmark programs. Result graph is shown above in figure 3.8. Here comparison between BTFNT, taken and not taken is shown. From the result graph it is clear that the result of the branch prediction is increased in the BTFNT technique than the not taken and taken techniques.

3.8 Summary And Discussion:

In this chapter I have discussed various version of ARM family and its cores. The parameters which are critical for optimization are discussed.

- Power consumption is very critical issue and it should be handled with care for effective power optimized embedded systems. For designing embedded system, designer should look for the application specific power optimization techniques. Mainly there is trade-off between area, delay and power consumption, so one should have design approach where there is little-bit of compromise of any one of the three factors.

- In this chapter I have shown overview of the simplescalar branch prediction technique along with that new implemented technique is also described with the results. These techniques can be easily implemented on any architecture which consists of pipelining. By implementing the BTFNT one can easily avoid the pipeline hazards and remove pipeline stalls from the architecture.

Architecture version 7 and higher versions are explained in detailed in Chapter number 5.