# Design of 16 bit ARM based Instruction Set Architecture

**Apurva Tayade, Chinmay Gadgil, Janaki Thakkar,**

**Mohnish Bhatia, Sagar Bachwani, Saurabh Nair**

**B.Tech(Electronics and Telecommunication),**

**Veermata Jijabai Technological Institute(VJTI), Mumbai, India.**

**Abstract:**

The sales pitch goes something like this, ”The ARM architecture has the best MIPS to Watts ratio as well as best MIPS to dollars ratio in the industry; the smallest CPU die size; all the necessary computing capability coupled with low power consumption , all at a low cost.” This success of ARM has made it the best choice for Embedded Low-Power Applications. An important factor that contributes to making this claim true is the 16 bit compressed instruction set of ARM called Thumb Instruction Set due to which programs can be coded much more densely thus cutting down the hardware size. Our paper aims to make use of this optimized subset of ARM and also strike a difference between 16 bit ARM and 16 bit MIPS in terms of their speed and performance.

**Index Terms**: ARM, ISA, RISC, Code density

## I. Introduction:

ARM is a family of instruction set architectures for computer processors based on a reduced instruction set[1] computing (RISC) architecture developed by British company ARM Holdings. The first ARM processor was developed at Acorn Computers Limited, of Cambridge, England, between October 1983 and April 1985. At that time, and until the formation of Advanced RISC Machines Limited(which later was renamed simply ARM Limited) in 1990, ARM stood for Acorn RISC Machine. Later, after a judicious modification of the acronym expansion to Advanced RISC Machine, it lent its name to the company formed to broaden its market
beyond Acorn's product range. Despite the change of name, the architecture still remains close to the original Acorn design.

## II. 16 bit ARM Instruction Set Description

### A. Motivation

Even though, RISC processors provide the advantages of a smaller die size, low cost, shorter development cycle and a higher performance, they also suffer from the drawback of having a poor code density as compared to CISC processors.[6][9] The poor code density is the result of having a fixed length instruction set. ARM ,being made on the RISC principles, also has poor code density. This disadvantage is overcome by using the 16 bit compressed subset of ARM instruction set called the Thumb Instruction Set. Thumb Instructions are 16 bit long and encode the functionality of ARM instruction in half the number of bits. The implementation of such an instruction set would take up less space on FPGA and will also serve as a means of reducing the die size and hardware thus leading to simple and small design and also reducing the power consumption.[5][7]

**B. The 16 bit ARM programmer's model**

The Thumb instruction set is a condensed state of the ARM instruction set with total flexibility in accessing eight 'Lo' general purpose registers r0 to r7. r13 to r15 are special purpose registers. r13 is used as a stack pointer, r14 is used as link register and r15 as program counter. r8 to r12 have restricted access. [3]

| |
|---|
| r0 |
| r1 |
| r2 |
| r3 |
| r4 |
| r5 |
| r6 |
| r7 |
| r8 |
| r9 |
| r10 |
| r11 |
| r12 |
| SP(r13) |
| LR(r14) |
| PC(r15) |

## C. 16 bit ARM and 32 bit ARM differences

| 16-bit ARM | 32-bit ARM |
|---|---|
| Instructions are executed unconditionally. | Instructions are executed conditionally. |
| Instructions use 2-address format. | Instructions use 3-address format. |
| 16-bit ARM code uses 40% more instructions than 32-bit ARM code. | It uses fewer instructions than 16-bit ARM. |
| With 16-bit memory, 16-bit ARM code is 45% faster than 32-bit ARM code. | With 32-bit memory, 32-bit ARM code is 40% faster than 16-bit ARM code. |
| 16-bit instruction formats are less regular than 32-bit instruction format. | 32-bit instruction formats are more regular. |
| 16-bit ARM code uses 30% less external memory power than 32-bit ARM code. | 32-bit ARM code consumes more external power. |

## D. Difference between 16 bit MIPS and 16 bit ARM

| 16-bit MIPS | 16-bit ARM |
|---|---|
| ALU is simpler as it doesn't support operations with bulky logic such as multiplication and division. | ALU is very complicated as it supports operations like multiplication and division. |
| It does not consist of the Barrel Shifter unit for shifting operations | It consists of Barrel Shifter which provides shifting operations. |
| Pipelining is not essential and its implementation depends on applications. | Pipelining is the main characteristic of 16-bit ARM processor. |
| MIPS is not power efficient. | ARM is power efficient. |

## E. List of Instructions

The 16 bit ARM which we have designed supports 33 instructions which can be divided into 5 categories:

1. Data Processing Instruction: It includes all the arithmetic ,logical and shifting instructions
2. Data Transfer Instructions: It includes move, load and store instructions
3. Branch Instructions: It includes conditional and unconditional branch instructions
4. Stack Related Instructions: It includes push and pop instructions
5. Miscellanous: It includes instructions like breakpoint, halt and no operation instructions

A complete list of 33 instructions is provided in the table with a short description of each.

| Sr. No | Mnemonic | Instruction Format | Description |
|---|---|---|---|
| 1 | ADD | ADD Rm,Rn,Rd | Adds the contents of Rm and Rn and stores the result in Rd |
| 2 | SUB | SUB Rm,Rn,Rd | Subtracts the contents Rn from Rm and stores the result in Rd |
| 3 | ADC | ADC Rn,Rd | Add Rm and Rn with carry flag and store the result in Rd |
| 4 | SBC | SBC Rn,Rd | Subtracts Rn and borrow from Rm and result is stored in Rd |
| 5 | MOV | MOV Rn,Rd | Copies the contents of the register Rn to Rd |
| 6 | MUL | MUL Rn,Rd | Multiplies the contents of Rn and Rd and the result is stored in Rd |
| 7 | AND | AND Rn,Rd | Bitwise ANDs the contents of Rn and Rd and stores the result in Rd |
| 8 | OR | OR Rn,Rd | Bitwise ORs the contents of Rn and Rd and stores the result in Rd |
| 9 | XOR | XOR Rn,Rd | Bitwise XORs the contents of Rn and Rd and stores the result in Rd |
| 10 | MVN | MVN Rn,Rd | Copies the complemented contents of register Rn to Rd |
| 11 | LDR | LDR [Rm,Rn],Rd | Loads Rd with contents of memory address formed by Rm+Rn |
| 12 | STR | STR [Rm,Rn],Rd | Store contents of Rd to the memory address formed by Rm+Rn |
| 13 | ADI | ADI #<immd_8>,Rd | Adds Rd and a 8-bit immediate data and stores the result in Rd |
| 14 | SBI | SBI #<immd_8>,Rd | Subtracts 8-bit immediate data from Rd and result is stored in Rd |
| 15 | MVI | MVI #<immd_8>,Rd | Copies the 8-bit immediate content to the destination register Rd |
| 16 | CMP | CMP #<immd_8>,Rd | Compares the contents of Rd with 8 bit immediate data |
| 17 | ADDH | ADDH Rn,Rd | Adds Rn and Rd where one or both of these are high registers |
| 18 | MOVH | MOVH Rn,Rd | Copies Rn to Rd where one or both of these are high registers |
| 19 | LSL | LSL #<immd_4>,Rn,Rd | Logically shifts Rn to left by 4-bit data and stores result in Rd |
| 20 | LSR | LSR #<immd_4>,Rn,Rd | Logically shifts Rn to right by 4-bit data and stores result in Rd |
| 21 | ASL | ASL #<immd_4>,Rn,Rd | Provides the signed value of Rn multiplied by variable power of 2 |
| 22 | ASR | ASR #<immd_4>,Rn,Rd | Provides the signed value of Rn divided by a variable power of 2 |
| 23 | PUSH | PUSH <registers> | Used to store contents of the mentioned registers on the stack |
| 24 | POP | POP<registers> | Used to load contents of the mentioned registers from the stack |
| 25 | BL | BL <target_addr> | Provides an unconditional subroutine call to another routine |
| 26 | BUC | BUC,target adress | Branch unconditionally to the target address |
| 27 | BC | BC,target adress | Branch to the target address when carry is set |
| 28 | BNC | BNC,target adress | Branch to the target address when carry is not set |
| 29 | BZ | BZ,target adress | Branch to the target address when zero flag is set |
| 30 | BNZ | BNZ,target adress | Branch to the target address when result of operation is non zero |
| 31 | BKPT | BKPT | Causes a software breakpoint to occur |
| 32 | NOP | NOP | No operation. An idle machine cycle is executed |
| 33 | HLT | HLT | Halts the processor |

## III. Pipeline Implementation

In a pipeline, instruction cycle is broken up into fixed steps. Rather than processing each instruction sequentially just like a single cycle execution design, each instruction is split up into a sequence of steps which are executed with steps of several instructions being executed parallel. This makes a productive design allowing the processor deal with several instructions at the same time increasing efficiency. With many more than one instruction being executed in parallel, we can achieve more through-put proportional to number of stages in the pipeline. The latency remains same compared to non-pipelined systems but maybe even more due to stalls and pipeline hazards. But with separate systems working together, we have the liberty to increase the frequency of the system making it faster.

In our model, we used the classic RISC pipeline with 5 stages.[2]

The 5 stages were fetch, decode, execute, memory operation and write back.

1) **Fetch**: The fetch block is the program counter and incrementer itself. This block keeps incrementer the program counter register at every rising edge of the clock. Overwriting the register for a new value supplies the method for branching instructions. Fetching keeps the instruction queue filled.

2) **Decode**: This block of code takes in the instruction from the instruction queue and assigns correct signals at the next clock cycle. The decoder doesn't deal with timing of all the signals. This cycle decides the next state of all the blocks in the system.

3) **Execute**: The signals from the decoder pass select signals for the execute stage for e.g. add select lines for ALU.

4) **Memory read/write**: This stage is especially designed for load store instructions for the memory data transactions. The ALU result is passed to this stage as address or value to be written back.

5) **Write back**: This stage is the final stage of the pipeline where the final result is stored back into the register file.

Implementation of states: The decoder simply analyses the whole opcode and decides on all the pertaining signals to all units. But just assigning signals directly will lead to a single cycle execution only. Hence to design a pipeline out of these signals, we designed a control unit with different delays to synchronize the timing for all the 5 stages. The control unit is a 5 process design which is 5 states which work simultaneously making the pipeline possible.

## IV. Conclusion:

With the reviews of various type of controllers and processors in the market, we thus have come to a conclusion that it is time to have a 16 bit efficient controller with higher processing capabilities like arm v9 processors. Hence to keep the costs low and size small, we have designed and simulated the 16 bit arm design with 33 instructions successfully on VHDL and iSim simulator respectively. The design isn't using all features of Thumb mode and arm but has the powerful instruction set architecture that ARM possesses.

## V. References


[1] Patterson & Hennessy, Computer Organization and Design (ARM Edition)

[2] Steve Furber, ARM system On-chip architecture

[3] ARM v7-M Architecture, Application Level Reference Manual

[4] vhdlguru.blogspot.com

[5] Pong Chu, FPGA prototyping by VHDL examples

[6] ARM architecture reference manual

[7] John F Wakerly, Digital design

[8] MIPS Technologies, "MIPS32 Architecture for Programmers Volume IV-a:

The MIPS16 Application Specific Extension to the MIPS32 Architecture"

[9]D. Seal, Editor, "ARM Architecture Reference Manual," Second Addition, Addison-Wesley

[10]http://en.wikipedia.org/wiki/ARM_Holdings3Sales_and_market_share

[11] Infocenter.arm.com – home>key features of ARM Architecture Versions

# Acceptance Letter

Dear Authors,

Warm Greetings from ISRJ

Thank you for valuable contribution in the research work. Your research paper titled "**Design of 16 bit ARM based Instruction Set Architecture**" has been accepted and it will be published in "Indian Streams Research Journal" in June Issue.

Regards,

ISRJ Team