CS440

The ARM1176JZF-S Architecture

Author:
Mr. Jason Mansfield

Instructor:
Prof. Pamela Smallwood

Contents

0.1 In the beginning there was Acorn

In 1985 the first ARM processor, the Acorn RISC Machine was introduced to the world (Levy and Promotions, 2005). Later in 1990 the Advanced RISC Machines Ltd.(ARM) would be launched. Unlike other RISC processor vendors of their time ARM began creating small scale processors. A whitepaper (Kamath and Kaundin, 2001) from 2001 Strategy made this statement:

> At Wipro, significant focus has been on the ARM processor technology, since we believe that will drive the evolving market for embedded applications, mobile devices and next generation information appliances.

Although this insight was probably not difficult to gauge by 2001, the scale at which embedded mobile devices has exploded onto the market has been impressive. Larger corporations, which have not been know for ingenuity, such as Microsoft, have been dealt a massive blow by new mobile devices such as Apples IOS based iPhone and iPad, or the fleets of Android based devices. The need for a smaller architecture has never been greater and ARM is sitting center stage.

0.1.1 The need for a smaller silicon area

The ARM architecture a **Reduced Instruction Set Computer** or RISC based architecture is now considered a dominant choice for developers and manufacturers. The ARM architecture incorporates standard RISC features (*ddi0406b* 2011, A1-2):

- a large uniform register file.
- a load/store architecture, where data-processing operations only operate on register contents, not directly on memory contents.

- simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only.

The ARM architecture has proven to be a better choice for smaller devices due to the low power consumption along with good performance. The **RM Architecture Reference Manual** listed the following additional reasons ARM is designed for smaller devices (*ddi0406b* 2011, A1-2):

- instructions that combine a shift with an arithmetic or logical operation.
- auto-increment and auto-decrement addressing modes to optimize program loops.
- Load and Store Multiple instructions to maximize data throughput.
- conditional execution of almost all instructions to maximize execution throughput.

0.2 The Instruction Set Architecture

Currently ARMv6 has ISA support for the following (*Specifications* 2013):

- ARM
- Thumb®
- Jazelle DBX®
- DSP extension
- Floating Point Unit

0.2.1 state switching

The ARM processor allows switching of states using the BX and BLX instructions. The ARM state is 32-bit word-aligned, the Thumb a 16-bit halfword-aligned, and the Jazelle state is variable length, byte aligned for instructions (*ddi0301h* 2009, pp. 2-12).

0.2.2 Conditionalised

ARM instructions can be whats called conditionalised. If the needs of the condition code are not met the instruction will simply become a NOP:

Condition Code	Meaning
N	Negative condition code, set to 1 if result is negative
Z	Zero condition code, set to 1 if the result of the instruction is 0
C	Carry condition code, set to 1 if the instruction result in a carry condition
V	Overflow condition code, set to 1 if the instruction results in an overflow condition.

Figure 1: Conditions

0.2.3 Conditional codes

Conditional Codes mean Conditional execution. The bit [31:28] determine the condition or lack there of:

cond	mnemonic ext	meaning int	meaning floating point	cond flag
0000	EQ	Equal	Equal	Z==1
0001	NE	Not Equal	Not Equal, or unordered	Z==0
0010	CS^b	Carry set	Greater than, equal, or unordered	C==1
0011	CC^c	Carry clear	Less than	C==0
0100	MI	Minus, negative	Less than	N==0
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N==0
0110	VS	Overflow	Unordered	V==1
0111	VC	No overflow	Not unordered	V==0
1000	HI	Unsigned higher	Greater than, or unordered	C==1 and Z==0
1001	LS	Unsigned lower or same	Less than or equal	C==0 and Z==0
1010	GE	Signed greater than or equal	Greater than or equal	N==V
1011	LT	Signed less than	Less than, or unordered	N!=V
1100	GT	Signed greater than	Greater than	Z==0 and N==V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z==1 or N!=V
1110	None $(AL)^d$	Always(unconditional)	Always(unconditional)	Any

Figure 2: Conditional Codes

0.2.4 A32

ARM is also known as A32 (*A32(ARM)* 2013). ARMv6 architecture is amongst a few others which use A32 such as ARMv5TEJ and ARMv4T.

Instruction length and format

ARM instructions are 32-bits wide and have a 4-byte boundary (*A32(ARM)* 2013). The subdivisions of the ARM instruction set can be seen in the below figure 1 (*ddi0406b* 2011, A5-2). As you can see each ARM instruction is composed of a 32-bit word. The 32-bit word's subdivisions are determined by bits [31:25,4]. Additionally, the conditional subdivision can be see between bits [31:28]. The conditional field allows for more optimizations.

31 30 29 28	27 26 25	24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4	3 2 1 0
cond	op1		op	

Figure 3: ARM subdivisions

General Instruction Categories

The following figure 4, shown below, illustrates the encoding which defines the various classes of instructions used with ARMv6 (*ddi0406b* 2011, A5-2):

cond	op1	op	Instruction classes
not 1111	00x	-	Data-processing and misc instructions
not 1111	010	-	Load/Store word and unsigned byte
not 1111	011	0	Load/Store word and unsigned byte
not 1111	011	1	Media instructions
not 1111	10x	-	Branch, branch with link, and block data transfer
not 1111	11x	-	Supervisor Call and coprocessor instructions
1111	-	-	Unconditionally executed

Figure 4: ARM Instruction encoding

The Branch Instruction

As can be seen in figure 4 op1 determines the instruction class. For example when op1 = 10x one of the various branching or block data transfer instructions is being used. If branch is the instruction specifically being used then 10xxxx will be found between [25:20] as shown in figure 3 below (*ddi0406b* 2011, A5-27):

31 30 29 28	27 26	25 24 23 22 21 20	19 18 17 16	15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
cond	1 0	op			
not 1111	1 0	10xxxx			

Figure 5: Branch equals 10xxxx

Going a step further, the following figure 5 shows the branch instructions details in Encoding A1 with no conditions (*ddi0406b* 2011, A8-44).

31 30 29 28	27 26 25 24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0	1 0 1 0	imm24

Figure 6: imm32 = SignExtend(imm24:'00', 32);

Encoding A1 indicates multiples of 4 in the range −33554432 to 33554428. Other encodings such as T1, T2, T3, and T4 have smaller ranges with T1 being the smallest of the permitted offsets. The T1 range is −256 to 254. All other offset ranges besides Encoding A1 are in even numbers, while Encoding A1, shown in figure 4, is in Multiples of 4. ARM encodings are labeled as A1, A2, A3 and so forth, while Thumb encodings are listed as T1, T2, T3 and so forth. Additionally, there are also encodings for ThumbEE which are listed as E1, E2, E3 and so forth (*ddi0406b* 2011, A8-282).

The MOV Instruction

Looking back at figure 2 to op1 = 00x you can see the 27th and 26th bit is determined to be both 0 for all instructions defined as data processing and misc.

31 30 29 28	27 26	25	24 23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0	op	op1		op2	

Figure 7: Data-processing and misc

One instruction which falls under the category of data processing is the MOV instruction. The following figure demonstrates the use of the MOV instruction in encoding A1 (*ddi0406b* 2011, A8-194).

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	1 1 0 1	S	(0)(0)(0)(0)	Rd	imm12

Figure 8: MOV instruction

0.2.5 Syntax for using the branch and mov instructions

The aforementioned instructions mov and branch are used in the following manor:

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0	0 0	1	1 1 0 1	0	0 0 0 0	0 0 0 0	0 0 0 0 0 0 0 0 1 0 1 1

Figure 10: MOV r0, #11

```
/*an assembly code example using instruction mov*/
.global main
.func main

main:
     mov r0, #11 /* Put the number eleven in register r0*/
     bx lr
```

Figure 9: mov instruction

The below code clip shows a unconditional branch being used:

```
/*an assembly code example using the unconditional branch instruction*/
.text
.global main
main:
      mov r0, #11
      b finish /*branch to finish*/
      mov r0, #22
finish:
      bx lr
```

Figure 11: branch instruction

When the above code is run the second mov instruction will be skipped due to the branch instruction pointing to *finish*.

0.3 The main components of the ARM1176JZF-S

The following components are considered the main components for the ARM1176JZF-S processor (*ddi0301h* 2009, pp. 1-8):

0.3.1 List of components

Integer Core The ARM1176JZF-S processor is built around the ARM11 integer core. Therefore, it is a implementation of the ARMv6 architecture. This architecture handles the following critical items (*ddi0301h* 2009, pp. 1-9):

- Instruction sets
- Conditional execution
- Registers
- Modes and exceptions
- Thumb instruction set
- DSP instructions
- Media extensions
- Datapath
- Branch prediction
- Return Stack

Load Store Unit (LSU) The load-store pipeline decouples loads and stores from the MAC and ALU (*ddi0301h* 2009, pp. 1-11).

Prefetch unit Fetches instructions from the instruction cache, external memory and instruction TCM to predict branch outcomes (*ddi0301h* 2009, pp. 1-11).

Memory system The memory system provides the core with features such as virtual indexing, export of memory, memory access control and many other capabilities (*ddi0301h* 2009, pp. 1-12).

AMBA AXI interface This bus interface allows high bandwidth connectivity between the processor, second level caches, on-chip RAM, peripherals, and interfaces to external memory (*ddi0301h* 2009, pp. 1-16).

Coprocessor interface This is a external coprocessor which interfaces with the ARM1176JZF-S to handle ARM coprocessor instructions (*ddi0301h* 2009, pp. 1-17).

Debug Using the ARMv6 debug architecture the following levels of debugging are allowed (*ddi0301h* 2009, pp. 1-18):