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):

- debug everywhere
- debug in non-secure privileged user, and secure user.
- debug in non-secure.

Instruction cycle summary and interlocks Allows handling of cycle timing behavior.

Vector Floating-Point (VFP) Handles floating point arithmetic operations (*ddi0301h* 2009, pp. 1-19).

System control Controls the memory system and other functionality (*ddi0301h* 2009, pp. 1-21).

Interrupt handling The interrupt handling deal with the following areas (*ddi0301h* 2009, pp. 1-21):

- Vectors Interrupt Controller port
- Low interrupt latency configuration
- Configuration
- Exception processing enhancements

0.4 ARM1176JZF-S pipeline stages

0.4.1 The Common Decode Pipeline

The datapath for the ARM1176JZF-S consists of three pipelines. Each of these pipelines are composed of eight stages.

- ALU, Shift and Sat pipeline
- MAC pipeline, also called the Multiply pipeline.
- Load/Store pipeline

The ARM1176JZF-S processor overlaps operations to improve clock rate speed for each instruction. The following figure 11 illustrates the first 4 stages; Fe1, Fe2, De, Iss are considered the **common decode pipeline**. From the common decode pipeline the direction taken will be one of three,

four step pipelines; The ALU pipeline, Multiply pipeline, or a Load/Store pipeline.

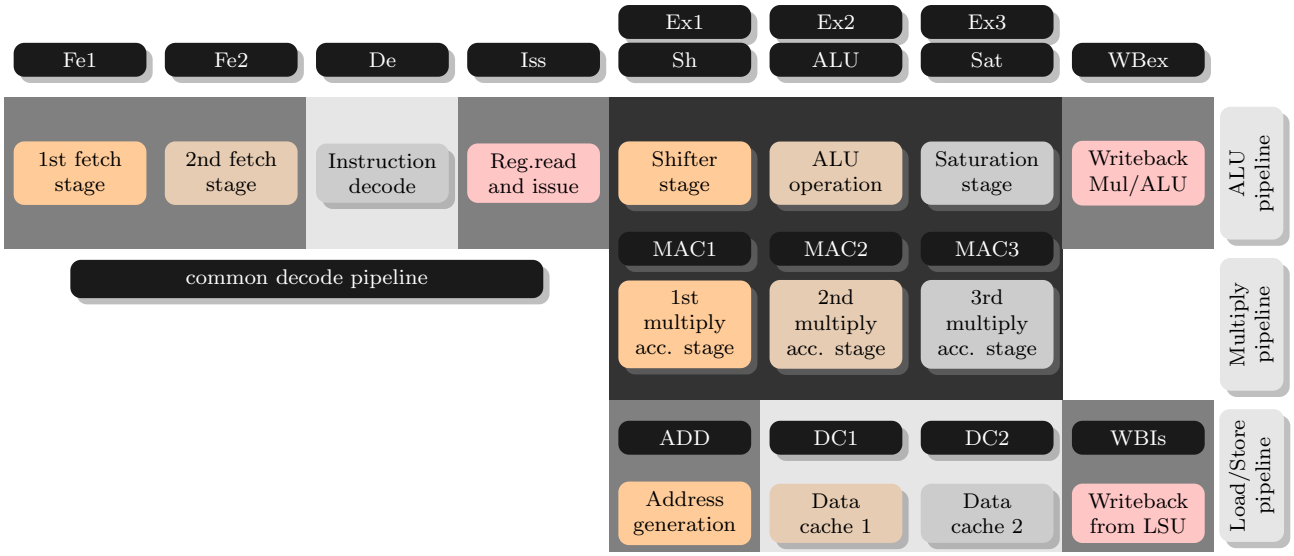

Figure 12: ARM1176JZF-S pipeline stages

0.4.2 Fetch stages 1 and 2

branch prediction

The first question you may ask is why two fetch stages? One reason is to provide time for conversion of a virtual address to a physical address. Beyond the integer core pipeline there is also a separate prefetch unit inside the core. The prefetch unit performs dynamic branch prediction. Branch prediction enables the detection of instructions before they enter the integer core (*ddi0301h* 2009, pp. 5-4).

instruction fetch

The first stage of instruction fetch deals with memory. Fe1 handles address issuing to memory and data returns from memory (*ddi0301h* 2009, pp. 1-26). If branch prediction handles instructions ahead of the fetch stages, the fetch

stage can handle up to four instructions. Instruction fetches are single-copy atomic (*ddi0406b* 2012):

- 32-bit granularity in ARM state.
- 16-bit granularity in Thumb and ThumbEE states.
- 8-bit granularity in Jazelle state.

0.4.3 Instruction Issue and Decode

The decode stage literally decodes data by extracting it from register addresses from the current instruction. The Issue and decode stages can contain any instruction in parallel with a predicted branch.

the coprocessor and the cores relationship

The coprocessor interface also has a decode stage which is passed directly from the core. Unlike, the core pipeline the coprocessor does not contain the fetch stages and instead begins with a instruction queue and decoder. The core does not discriminate between coprocessor related instructions or non-coprocessor instructions. Therefore, the coprocessor must react immediately with decoding (*ddi0301h* 2009, pp. 11-14). Other complexities are involved as the core may decide, after passing an instruction, to later cancel the previously passed instruction. For this reason the coprocessor maintains a cancel queue. The coprocessor is directed by the core to cancel the previous instruction by the signal **ACPCANCEL**. Upon the discovery of the ACPCANCEL signal the coprocessor removes the canceled instruction; the cancellation is handled during the Ex1 stage before moving to Ex2. This is only one small example of the many tasks the coprocessor handles.

0.5 Instruction Execution Pipeline

Past the Common Decode Pipeline is the Instruction Execution Pipeline. Each of these pipelines ALU, Multiply, and Load & Store are composed of

four stages, the same as the Common Decode Pipeline. Taking a look at the Program Counter register or R15 we can evaluate when a instruction has reached execution. The 32-bit ARM instructions can be read as the PC + 8, the 16-bit Thumb instructions as PC + 4 (*ddi0406b* 2012, A2-11). The following example of a GDB session demonstrates the registry **pc** position in relation to execution:

```
.text
.global main
main:
      MOV r0, #5
      MOV r1, #10
      MUL r2, r1, r0
finish:
      bx lr
```

Figure 13: 32-bit ARM assembly code

```
(gdb) info reg
r0             0x1        1
r1             0xbefff7d4         3204446164
r2             0xbefff7dc         3204446172
r3             0x8390         33680
r4             0x0        0
r5             0x0        0
r6             0x82e4         33508
r7             0x0        0
r8             0x0        0
r9             0x0        0
r10            0xb6fff000         3070226432
r11            0x0        0
r12            0xb6fc1000         3069972480
sp             0xbefff688         0xbefff688
lr             0xb6ead81c         -1226123236
pc             0x8390         0x8390 <main>
cpsr           0x60000010         1610612752
(gdb) disas
Dump of assembler code for function main:
=> 0x00008390 <+0>:     mov     r0, #5      EXECUTE
   0x00008394 <+4>:     mov     r1, #10     DECODE
   0x00008398 <+8>:     mul     r2, r1, r0  FETCH
End of assembler dump.
```

Figure 14: GDB

As you can see the **pc** registry is pointed to the current instruction for execution at address $0x00008390$. Here is another exciting fact, you are looking at the entire fetch, decode and execute cycle right now. As you may have noticed I have appended the text, EXECUTE, DECODE and FETCH to the assembler dump made by the disassemble command.

0.5.1 The Shift, ALU and Sat pipeline

The Shift, ALU and Sat pipeline handles the majority of ALU type operations (*ddi0301h* 2009, pp. 1-10). Additionally, the ARM1176JZF-S uses a

32-bit barrel shifter which works with a second operand before it enters the ALU.

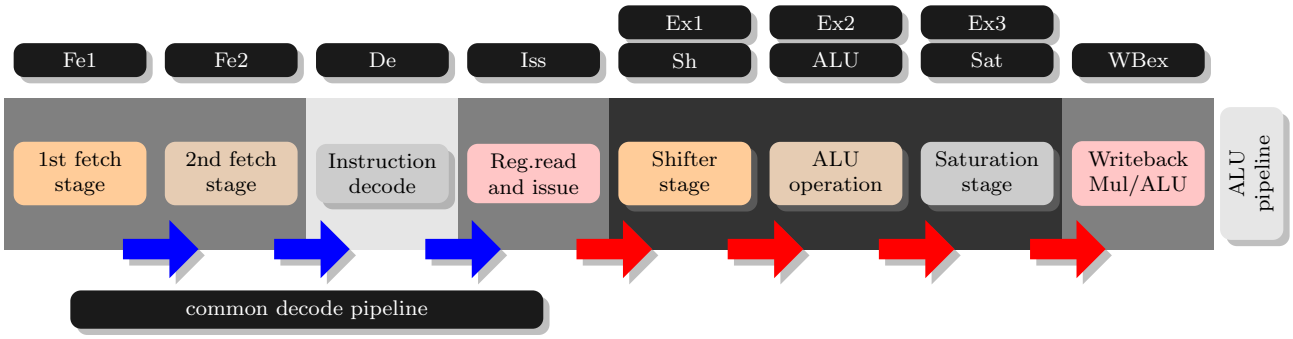

Figure 15: ARM1176JZF-S ALU Operation

Shift

The shift stage pertains to the full barrel shifter (Thomas, 2012) performing all shifts needed. The barrel shifter is connected to the ALU allowing it to directly pass the second operand after doing its job. A good example of the barrel shift in action is the logical shift left illustrated below:

```
.text
.global main
main:
      movs r0, #2
      mov r0, r0, LSL #1
finish:
      bx lr
```

Figure 16: Logical Shift Left ARM 32-bit

In the above example you might notice the LSL is on the tail end of the instruction. This is actually a MOV shifted register. The canonical form or equivalent instruction would be:

```
LSL{S} <Rd>, <Rm>,#<n>
```

In fact if you disassemble the above code clip with GDB, the instruction will be seen as the LSL instruction, and not a MOV instruction at all:

```
(gdb) disas
Dump of assembler code for function main:
=> 0x00008390 <+0>:     movs    r0, #2
   0x00008394 <+4>:     lsl     r0, r0, #1
End of assembler dump.
(gdb) x/t 0x00008394
0x8394 <main+4>:    11100001101000000000000010000000
```

Figure 17: Logical Shift Left Assembler Dump

ALU

The ALU handles all arithmetic and logic operations. The ALU also deals with the conditional codes for instructions. The ALU stage is composed of:

- logic unit
- arithmetic unit
- flag generator

Sat

The Sat stage implements saturation logic needed by various classes of DSP instructions.

WBex

To complete the ALU pipeline there is a write back to a base register during the WBex stage.

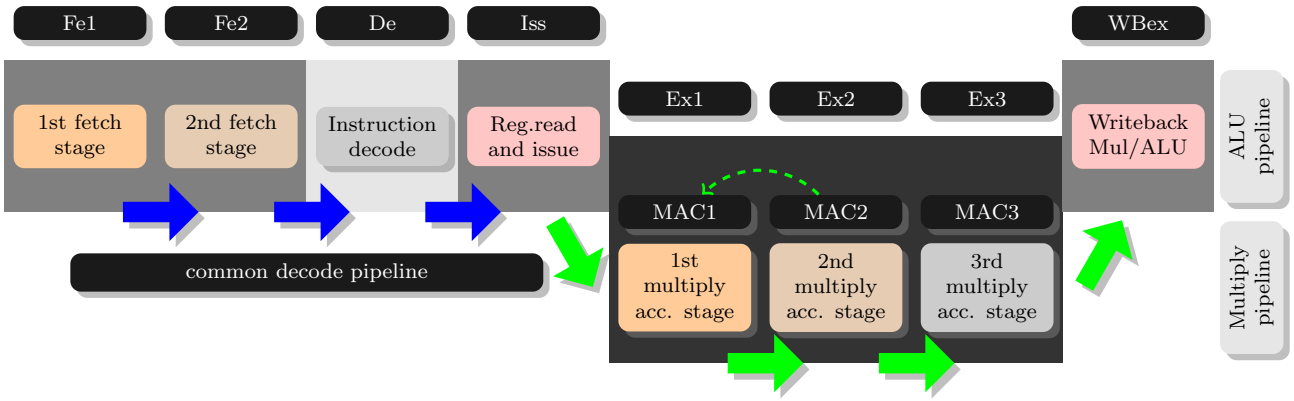

Figure 18: ARM1176JZF-S Multiply Operation

0.5.2 Multiply pipeline

Unlike MAC2 and MAC3, MAC1, the first stage in the multiply pipeline, can loop until it has passed through the first part of the multiplier array enough times. The second half of the array is dealt with by MAC2 and MAC3 until completion (*ddi0301h* 2009, pp. 1-28). To complete the pipeline a write back is done to the base register. The MAC pipeline executes all of the enhanced multiplication instructions (*ddi0301h* 2009, pp. 1-11).

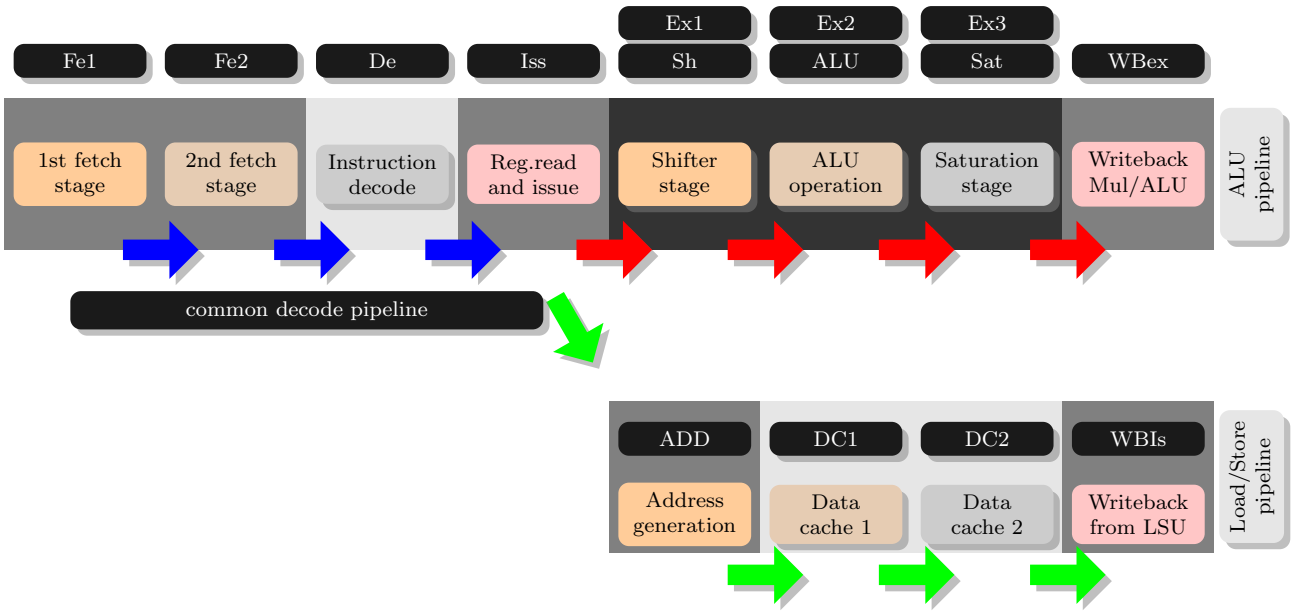

Figure 19: ARM1176JZF-S LDR/STR operation

0.5.3 The Load/Store pipeline

The Load/Store pipeline consists of the Data address calculation stage followed by two data cache stages. The pipeline completes with a write back of data from the LSU or Load Store Unit. When the processor issues LDM or STM instructions to the LSU, other instructions from the MAC and ALU pipelines run concurrently (*ddi0301h* 2009, pp. 1-11).

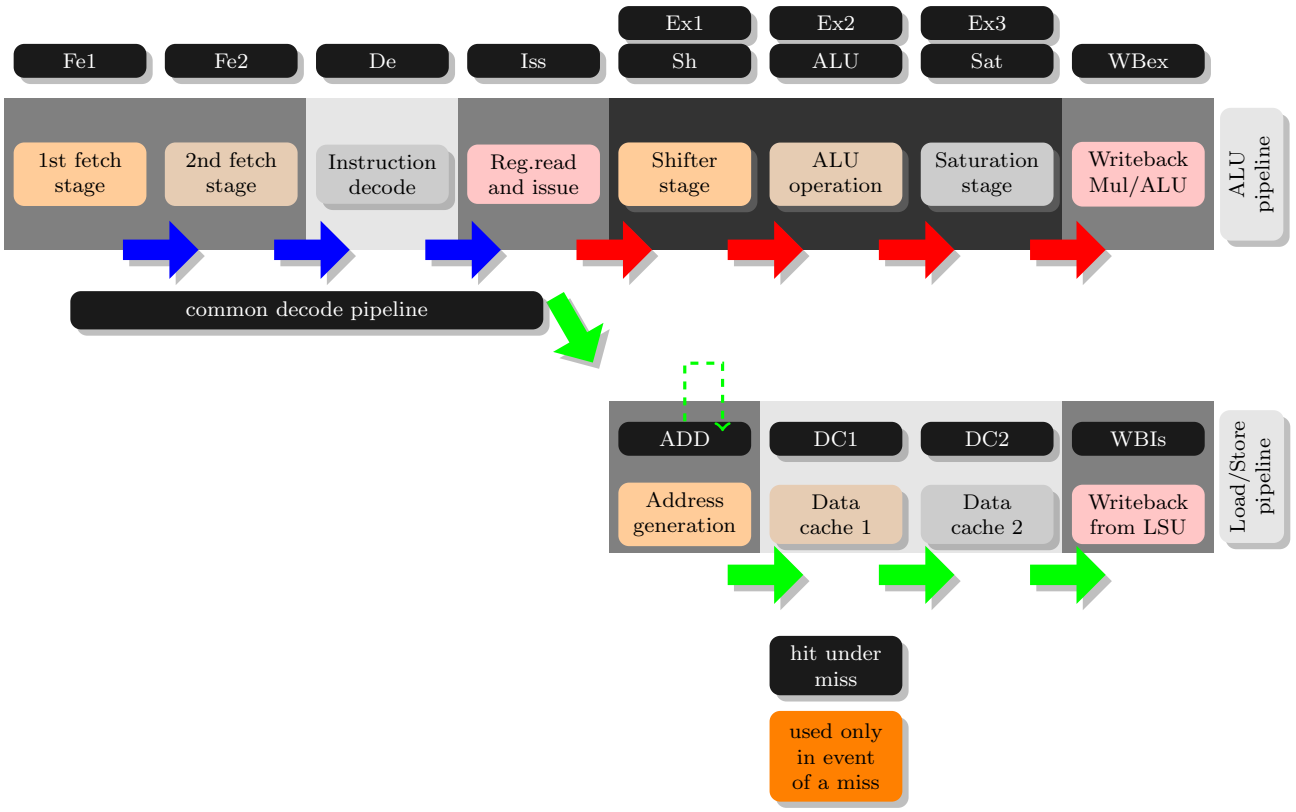

Figure 20: ARM1176JZF-S LDM/STM operation

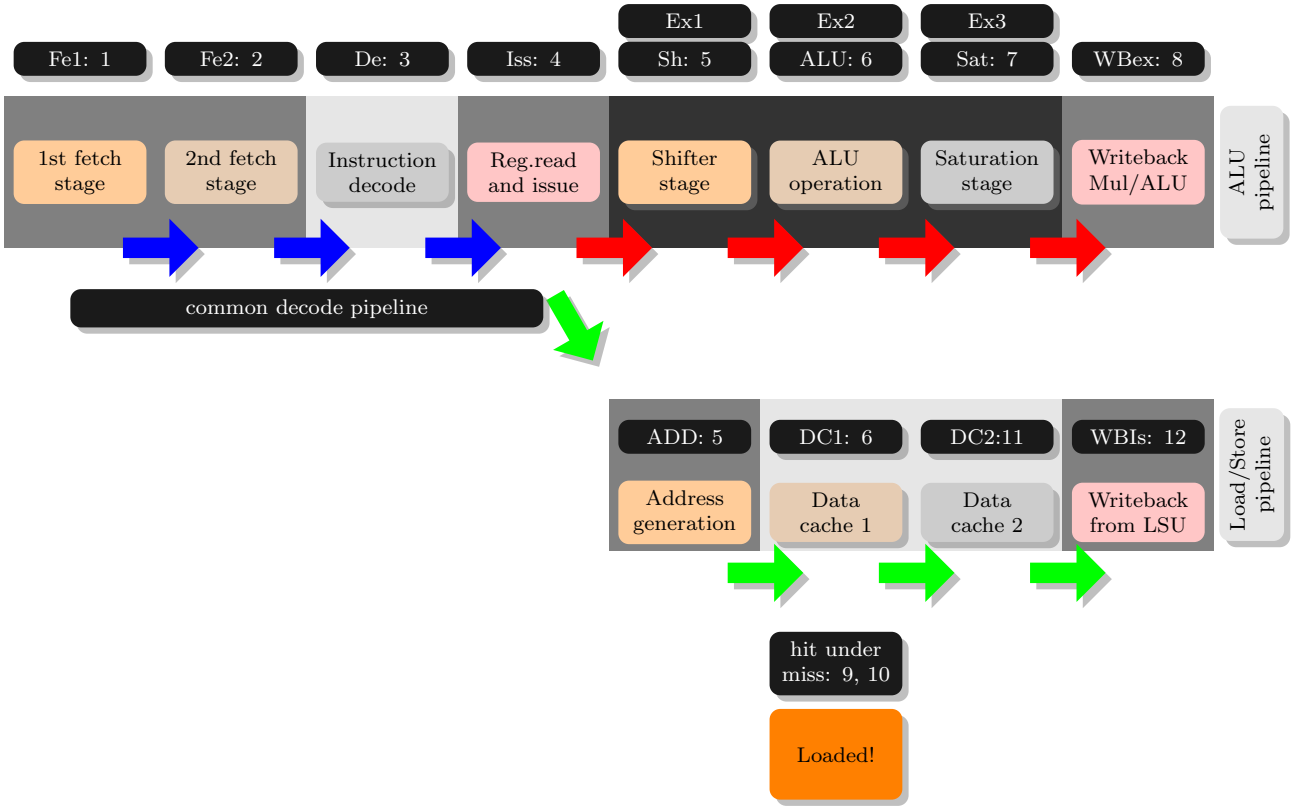

Figure 21: ARM1176JZF-S Example LDR miss

0.5.4 Cycle Timing and Instruction Execution

Both the ALU and Multiply pipelines move together so all instructions finalize at the same moment. Unlike the ALU and Multiply pipelines the load/store pipeline is decoupled and allows the other pipelines to continue afterwards. For example to avoid an instruction from writing to a register before a store multiple has stored that particular register, the register list has a lock latency which determines how many cycles before a write may occur. The ARM1176JZF-S Reference Manual gives this description regarding the use of forwarding (*ddi0301h* 2009, pp. 16-21):

> Extensive forwarding to the Sh, MAC1, ADD, ALU, MAC2, and DC1 stages enables many dependent instruction sequences to run without pipeline stalls. General forwarding occurs from the ALU, Sat, WBex and WBls pipeline stages. In addition, the multiplier contains an internal multiply accumulate forwarding path. Most instructions do not require a register until the ALU stage. All result latencies are given as the number of cycles until the register is required by a following instruction in the ALU stage

Use of Flags and Condition codes

According to the ARM1176JZF-S Reference Manual most instructions will execute within one or two cycles unless failing their condition codes. A specific algorithm is given by the manual for determining the number of cycles for multicycle instructions that have a failed condition code (*ddi0301h* 2009, pp. 16-4):

$$Min(NonFailingCycleCount, Max(5 - FlagCycleDistance, 3))$$

The **NonFailingCycleCount** variable shown above is the number of cycles the same instruction will take when it passes. The **FlagCycleDistance** is the distance between two instructions. The first instruction is the flag setter and the second the conditional instruction which relies on it. A good example of a FlagCycleDistance of zero might be (*ddi0301h* 2009, pp. 16-4):

```
ADDS R0, R1, R2
MULEQ R3,R4, R5
```

Notice the operand ADD is ADDS, the S is an opptional suffix, which is a condition code that asks to update all the ALU status flags in the CPSR. Also notice that the MUL operand has the suffix EQ, this is a conditional code for Equal which uses a Z flag. In contrast the below example has a FlagCycleDictance of one, as there is a single instruction between the two aforementioned instructions:

```
ADDS R0, R1, R2
MOV R6, R6
MULEQ R3,R4, R5
```

Register Interlocks

Interlocks are used to avoid instructions writing to the same register at the same moment and creating a pipeline hazard. The following example takes four cycles (*ddi0301h* 2009, pp. 16-6) to complete due to the resultant latency of **R0**.

```
LDR R0, [R1]
ADD R4, R3, R0
```

However, if the ADD instruction was not waiting on a register from the previous command the instruction sequence would only take two cycles:

```
LDR R0, [R1]
ADD R4, R3, R2
```

0.6 A note about the ARM11 design

The ARM design is primarily driven towards low energy consumption. In the past ARM would not have been considered a good solution for mid-sized to large computers such as a web server. The majority of desktop sized computers run on x86 or x86_64 based processors of one sort or another. The architecture reviewed today is 32-bit and is unlikely to be used in future desktop sized computers, but other ARM based creations are being considered by high-tech companies such as Apple (Duncan, 2012) and AMD (Shah, 2013). The ARM11 is primarily for smaller hand held devices such as the iPhone, iPad, Linux or Android based phones and Tablets and other smaller devices such as the Raspberry Pi or your Samsung TV. The ARM11 is small enough, energy efficient enough, and designed well enough to run on a small

battery for hours. Very few architectures can claim the same capabilities at this time.

Bibliography

A32(ARM) (2013). ARM Limited. URL: http://www.arm.com/products/processors/instruction-set-architectures/index.php.

ddi0301h (2009). ARM1176JZF-S Technical Reference Manual. ARM Limited. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/index.html.

ddi0406b (2011). ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. ARM Limited. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b/index.html.

ddi0406b (2012). ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. ARM Limited. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b/index.html.

Duncan, Geoff (Nov. 9, 2012). *Why would Apple move Macs to ARM processors?* URL: http://www.digitaltrends.com/computing/why-might-apple-move-macs-to-arm-processors/.

Kamath, Udaya and Rajita Kaundin (2001). "System-on-Chip Designs". In: *White Paper, June*.

Levy, Markus and Convergence Promotions (2005). "The history of the ARM architecture: From inception to IPO". In: *ARM IQ* 4.1.

Shah, Agam (June 18, 2013). *AMD reboots server strategy with first ARM chips*. URL: http://www.infoworld.com/d/computer-hardware/amd-reboots-server-strategy-first-arm-chips-220919.

Specifications (2013). ARM Limited. URL: http://www.arm.com/products/processors/classic/arm11/arm1176.php.

Thomas, David (July 13, 2012). *Introduction to ARM*. URL: http://www.davespace.co.uk/arm/introduction-to-arm/instruction-sets.html.