Appendix A The Co-processor Instructions

In Chapter Seven, we talked about the undefined instruction trap. This occurs when the ARM tries to execute an instruction which does not have a valid interpretation. There are over four billion possible combinations of the 32 bits which form an instruction code, so it is not really surprising that some of these are not defined to do anything meaningful.

There are two types of undefined instruction. The first set are totally illegal, and cause the undefined instruction vector to be called whenever they are encountered. The second set are only classed as illegal if there is no co-processor in the system which can execute them. Unsurprisingly, these are called the co-processor instructions. Note that the ARM itself treats all undefined instructions as possible co-processor ones. The only thing which distinguishes the first class from the second is that no co-processor will ever indicate that it can execute the first type.

Note also that if an instruction is not executed because its condition codes cause it to be ignored by the ARM, it will never be 'offered' to a co-processor, or trapped as an undefined instruction. This means that if an instruction is a 'no-operation' due to its condition code being 'never', the rest of instruction can be anything - it will never cause a trap to occur.

A.1 ARM/co-processor handshaking

An ARM co-processor is an external chip (or chips), connected to the ARM data and control buses. When the ARM fails to recognise an instruction, it initiates a co-processor handshake sequence, using three special signals which connect the two chips together. The three signals are:

CPI Co-processor instruction.

The ARM asserts this when it encounters any undefined instruction. All co-processors in the system monitor it, ready to leap into action when they see it become active.

CPA Co-processor absent.

When a co-processor sees that an undefined instruction has been fetched (by reading the CPI signal), it uses CPA to tell the ARM if it can execute it. There are two parts to the instruction which determine whether it is 'co-processable'. The first is a single bit which indicates whether a particular undefined instruction is a proper co-processor one (the bit is set) or an undefined one (the bit is clear).

The second part is the co-processor id. This is a four-bit field which determines which of 16 possible co-processors the instruction is aimed at. If this field matches the co-processor's id, *and* the instruction is a true co-processor one, the co-processor lets the ARM know by setting the CPA signal low. If none of the co-processors recognises the id, or there aren't any, the CPA line will remain in its normal high state, and the ARM will initiate an undefined instruction trap.

CPB Co-processor busy

Once a co-processor claims an instruction, the ARM must wait for it to become available to actually execute it. This is necessary because the co-processor might still be in the middle of executing a previous undefined instruction. The ARM waits for the co-processor to become ready by monitoring CPB. This is high while the co-processor is tied up performing some internal operation, and is taken low when it is ready to accept the new instruction.

Note that while the ARM is waiting for CPB to go low, the program which executed the co-processor instruction is effectively halted. However, interrupts are still enabled, so these may be serviced as usual. When the interrupt routine returns to the co-processor instruction, it is effectively re-executed from the beginning, with the handshake starting again from the ARM asserting CPI. Note also that in a multi-tasking system, where several programs share the processor in turn, switching between tasks is performed under interrupts, so only the program which executed the co-processor instruction will be held up by the ARM waiting for the co-processor to become available.

Once the ARM gets the co-processor's undivided attention, they can execute what happens once the co-processor is ready depends on which class the instruction belongs to. Simplest are the internal co-processor operations. These require no further intervention from the ARM, which continues executing the next instruction, while the co-processor performs the required operation.

The second class is ARM to (or from) co-processor data transfer. This is where data is transferred between ARM registers and those on the co-processor. Thirdly, co-processor to (or from) memory operations may be executed. In these, the ARM provides the addresses, but data is transferred into or out of the co-processor. In the sections below, each class of co-processor instruction is described in detail.

Like the ARM, a co-processor can restrict instruction execution to supervisor mode only. It does this by monitoring the state of the SPVMD signal, described in Chapter Seven. If a privileged instruction is attempted from user mode, the co-processor could signal this using the ABORT line, just as the MEMC does. The abort handler code would have to be able to determine that it was a co-processor rather than the MEMC which created the abort, so that it could deal with it appropriately. However, there is no way in which an aborted co-processor instruction could be re-started, because the ARM cannot tell where the instruction originally came from.

A.2 Types of co-processor

Currently, only one co-processor's instruction set has been fully defined. This is the floating point unit (FPU). As the chips have not been made yet, even these instructions are treated as undefined ones. However, Acorn has written a program, the floating point emulator package, which intercepts the undefined instruction trap and emulates in software the floating point instruction set. The only difference between a system which has an FPU fitted and one which uses the emulator is the speed in which they execute the instructions (the emulator is anything between 10 and 100 times slower, depending on the operation).

Because of the way in which the co-processor protocol works, it is possible to design chips which recognise successively more of the instruction set. Thus a first generation FPU might just execute the four simple arithmetic operations in hardware, leaving the emulator to do the difficult stuff (possible using the FPU instructions to do it). A final version could implement the whole of the instruction set, leaving the emulator with nothing to do.

The FPU instruction set is described in Appendix B.

Further co-processors might concentrate on other aspects of a system. For example, a graphics co-processor might execute line-drawing and shape filling instructions, performing the required calculations and data transfers without placing any burden on the ARM. Such a chip would probably have its own screen memory so that it would not have to compete with the ARM for access to RAM.

A.3 Co-processor data instructions

This class of instruction requires no further action from the ARM once it has ascertained that there is a co-processor capable of dealing with it. However, as mentioned above, the ARM may still have to wait for the co-processor to become available before it can carry on with the next instruction.

Because there is no further communication between the ARM and the co-processor, the latter could incorporate an internal buffer where it stores the instructions for later execution. As long as there is room in the buffer (commonly called a FIFO for 'first-in, first-out queue'), the co-processor can accept an instruction immediately. Thus while the ARM is fetching and executing (defined) instructions from memory, the co-processor could meanwhile be executing its internal operations in parallel. This is similar to the well known Acorn Second Processor concept, where an I/O processor accepts commands from a FIFO while the language processor works independently on some other task.

In practice the amount of parallel processing which can be achieved is limited by the fact that eventually the ARM will want a result from the co-processor; so it will hang around waiting for the queue of instructions to be executed before it can ask for the required values. In situations where results are not required, e.g. a graphics co-processor which just takes commands and executes them without passing information back, the benefits can be great.

The general mnemonic for this type of instruction is **CDP**, for co-processor data processing. There are five operands which define which co-processor is required, which operation is required, which are the source and destination registers, and one for 'optional extra' information. Of course, the condition code mnemonic may also be present.

Here is the general form of a **CDP** instruction:

```
CDP{cond}  <cp#>,<op>,<dest>,<lhs>,<rhs>,{info}
```

Where:

{cond}	is the optional condition code
<cp#>	is the co-processor number (0-15)
<op>	is the desired operation code (0-15)
<dest>	is the co-processor destination register (0-15)
<lhs> and **<rhs>**	are the co-processor source registers (0-15)
{info}	is the optional additional information field (0-7)

The last five are all stored as four-bit (or three-bit in the case of **info**) fields in the instruction code. Note that except for the co-processor number field, which must be in a fixed place in the instruction for the system to work, none of the items has to be interpreted as above. A co-processor can place whatever meaning on the remaining 19 bits that it wants, but this standard defined by Acorn should at least be used as a starting point.

A particular type of co-processor would have its co-processor field value fixed, and the operation code could be given a more meaningful name. For example, the FPU has a **<cp#>** of 1 and uses the **<op>** field to describe one of sixteen possible instructions.

There are only eight registers in the FPU, so the top bit of the **<dest>**, **<lhs>** and **<rhs>** fields can be used for other purposes. For example, the top bit of the **<dest>** field is used to indicate which class of operation is required (dyadic, operating on two values, or monadic, operating on one).

So, although the FPU conforms to the spirit of the standard, it uses the fields to maximise the use of the bits available in the instruction. A typical FPU instruction is:

```
ADF {cond}<P>{R}  <dest>,<lhs>,<rhs>
```

where:
<P> is the precision of the operation
{R} is the optional rounding mode and the other fields are as above.

The **<P>** and **{R}** fields are encoded into the 'optional extra' information field and the top bit of the **<lhs>** field. For more details about the FPU instruction set, see Appendix B.

As another example, a graphics co-processor might use a **CDP** instruction to set its colour palette entries. For example,

```
CDP    2,<palette>,<entry>,<value>,<component>
```

where:
`2` is the co-processor number
`<palette>` is the op-code for setting the palette
`<entry>` is the logical colour number (0-15) (the **`<dest>`** field)
`<component>` is the red, green or blue component (0-2) (the **`info`** field)
`<value>` is the intensity for that component (0-65535) (the **`<lhs>`** and **`<rhs>`**) field.

As long as the desired operation can be expressed in the number of bits available, any operation which requires no further communication between the ARM and co-processor can use a **`CDP`**-type instruction.

A.4 Co-processor register transfer operations

This class of co-processor instruction is used to transfer a single 32-bit value from the ARM to the co-processor, or from the co-processor to the ARM. The standard mnemonics are **`MRC`** for Move from ARM to Co-processor, and **`MCR`** for Move from Co-processor to ARM. The general forms are:

```
MRC{cond}  <cp#>,<op>,<ARM  srce>,<lhs>,<rhs>,{info}
MCR{cond}  <cp#>,<op>,<ARM  dest>,<lhs>,<rhs>,{info}
```

where:
`<cp#>` is the co-processor number (0-15)
`<op>` is the operation code required (0-7)
`<ARM srce>`/**`<ARM dest>`** is the ARM source/destination register (0-15)
`<lhs>` and **`<rhs>`** are co-processor register numbers (0-15)
`{info}` is optional extra information (0-7)

Notice that the number of possible opcodes has been halved compared to **`CDP`**, as there are now two directions between which to share the sixteen possible codes.

The **`MCR`** instruction does not necessarily have to be a simple transfer from a co-processor register to an ARM one. A complex internal operation could be performed on **`<lhs>`** and **`<rhs>`** before the transfer takes place. The co-processor only signals its readiness to the ARM when it has performed all the necessary internal work and is prepared to transfer the result to the ARM.

Again, the exact interpretation of the fields depends on the type of co-processor. An example of an FPU **`MRC`** operation is the 'float' instruction, to convert a 32-bit two's complement integer (in an ARM register) to a floating point number of some specified precision and rounding type (in an FPU register):

```
FLT{cond}<P>{R}   <ARM  srce>,<lhs>
```

Similarly, an FPU **MCR** instruction is the 'fix' operation, which converts a floating point number (in the FPU) to an integer (in the ARM):

```
FIX{cond}<P>{R}   <ARM  dest>,<rhs>
```

Notice that a different field is used in each case to specify the required FPU register. This does not concern the programmer, since only a register number (or name) is specified. It is up to the assembler to generate the appropriate binary op-code.

A graphics co-processor might use this instruction to read or write the current pixel location. If expressed as two sixteen bit co-ordinates, this could be encoded into a single 32-bit register. The operations might be:

```
MCR    2,<cursor>,<ARM  dest>
MRC    2,<cursor>,<ARM  srce>
```

In this case, only the opcode is set to **<cursor>** to indicate a cursor operation. The **<lhs>**, **<rhs>** and **{info}** fields are not used.

Using R15 in MCR/MRC instructions

When R15 in the ARM is specified as the **<ARM dest>** register, only four bits are ever affected: the N, Z, V and C flags. The PC part of R15 and the mode and interrupt bits are never changed, even in supervisor mode. This makes the **MCR** instruction useful for transferring data from the co-processor directly into the ARM result flags. The FPU's compare instructions use this facility.

If R15 is given as the **<srce>** register, all 32-bits are transferred to the co-processor. It is not envisaged that many co-processors will have a use for the ARM's PC/status register, but the ability to access it is there if required.

A.5 Co-processor data transfer operations

This group of instructions provides a similar function to the **LDR**/**STR** instructions, but transfers data between a co-processor and memory instead of between the ARM and memory. The address of the word(s) to be transferred is expressed in a similar way to **LDR**/**STR**, that is to say you can registers as a base. However, only immediate offsets are allowed, in the range -255 to +255 words (-1020 to +1020 bytes).

Like **LDR**/**STR**, write-back to the base register may be performed optionally after a pre-indexed instruction and is always performed after a post-indexed instruction.

The mnemonics are **LDC** to load data into a co-processor, and **STC** to store data. The full syntax is:

```
LDC{cond}{L}   <cp#>,<dest>,<address>
STC{cond}{L}   <cp#>,<srce>,<address>
```

where:
{L} is an optional bit meaning 'long transfer'; see below
<cp#> is the co-processor number (0-15)
<srce>,<dest> is the co-processor register number (0-15)
<address> specifies the address at which to start transferring data

Address modes

There are three forms that **<address>** can take: a simple expression, a pre-indexed address, and a post-indexed address.

If a simple expression is given for the address, the assembler will try to generate a PC-relative, pre-indexed instruction, without write-back. Thus if you say:

```
LDC     1,F1,label
```

(F1 being a co-processor register name) the assembler will generate this equivalent instruction:

```
LDC     1,F1,[R15,#label-(P%+8)]
```

assuming the label is after the instruction. Note that as with **LDR**/**STR**, the assembler takes pipe-lining into account automatically when R15 is used implicitly (as above), or explicitly (as below). Note also that the maximum offset in each direction is only 1020 bytes for **LDC**/**STC**, instead of 4095, which **LDR**/**STR** give you. If the address that the instruction tries to access is outside the available range, the assembler gives an error.

The remaining forms of **<address>** are:

[<base>]	address in ARM register **<base>**
[<base>,#<offset>]{!}	**<base>+<offset>**, {with write-back}

The offsets are specified in bytes, in the range -1020 to +1020. However, they are scaled to words (by dividing by four) when stored in the instruction, so only word-boundary offsets should be used.

These instructions differ from **LDR**/**STR** in that more than one register may be transferred. The optional **L** part of the syntax is designed to enable a choice between short (single word) and long (multiple word) transfers to take place. It is up to the co-processor to control how many words are transferred. The ARM, having calculated the start address from the base register, offset, and address mode will automatically increment the address after each operation.

Long transfers

Remember that **LDM**/**STM** always loads/stores registers in ascending memory locations. A similar thing happens with multiple register **LDC**/**STC** transfers. Suppose an **STCL** instruction for a given co-processor stores its internal registers from the one given in the instruction, up to the highest one, F7. The instruction:

```
STCL   1,F0,[R1,#-32]!
```

works as follows. The start address of the transfer is calculated from the contents of R1 minus 32. This is written back into R1. Then the eight data transfers take place, starting from the calculated address and increasing by four bytes after each one. Similarly,

```
LDCL   1,F4,[R1],#16
```

will load four registers, starting from the address in R1 and adding four after each transfer. Finally, 16 is added to the base register R1.

The FPU uses this group of instructions to load and store numbers from its floating point registers. There are four possible lengths allowed (single, double, extended and packed) which take one, two, three and three words of memory respectively. The **L** option only allows two possible lengths. However, the FPU only has eight internal registers, so the top bit of the **<dest>**/**<srce>** field is used to encode the extra two length options.

Apart from this minor difference, the FPU load and store data instructions obey exactly the general form of the co-processor data transfer, as Appendix B describes.

Aborts and address exceptions

Since this class of instruction uses the ARM address bus, instruction aborts exception (the start address has one or more of bits 26-31 set), exactly the same action occurs as for a native ARM exception, i.e. the instruction is stopped and the address exception vector called. In the case of multiple word transfers, only an illegal start address will be recognised; if it subsequently gets incremented to an illegal value, it will 'wrap-around' into the first few words of memory.

Data aborts also behave in a similar way to native ARM instructions. As usual, if write-back was specified, the base register will be updated, but no other changes will take place in the ARM or co-processor registers. This enables the abort handler code to re-start the instruction after undoing any indexing. Note that it is up to the co-processor to monitor the ABORT signal and stop processing when it is activated.

A.6 Co-processor instruction timings

In addition to the n, i and i-cycles mentioned at the end of Chapter Three, co-processor instructions also use c (for co-processor) cycles. These are the same period as s and i-cycles.

CDP timing

1 s + B i-cycles.

MRC timing

1 s + B i + 1 c-cycles.

MCR timing

1 s + (B+1) i + 1 c- cycles.

LDC/STC timing

(N-1) s + B i + 1 c-cycles.

where:
B is the number of cycles the ARM spends waiting for the co-processor to become ready (which has a minimum of zero).
N is the number of words transferred.