6 Architectural Support for High-level Languages

Summary of chapter contents

High-level languages allow a program to be expressed in terms of abstractions such as data types, structures, procedures, functions, and so on. Since the RISC approach represents a movement away from instruction sets that attempt to support these high-level concepts directly, we need to be satisfied that the more primitive RISC instruction set still offers building blocks that can be assembled to give the necessary support.

In this chapter we will look at the requirements that a high-level language imposes on an architecture and see how those requirements may be met. We will use C as the example high-level language (though some might debate its qualification for this role!) and the ARM instruction set as the architecture that the language is compiled onto.

In the course of this analysis, it will become apparent that a RISC architecture such as that of the ARM has a vanilla flavour and leaves a number of important decisions open for the compiler writer to take according to taste. Some of these decisions will affect the ease with which a program can be built up from routines generated from different source languages. Since this is an important issue, there is a defined *ARM Procedure Call Standard* that compiler writers should use to ensure the consistency of entry and exit conditions.

Another area that benefits from agreement across compilers is the support for floating-point operations, which use data types that are not defined in the ARM hardware instruction set.

6.1 Abstraction in software design

We have already met a level of abstraction that is important to software design. The essence of an ARM processor is captured in the ARM instruction set which was described in Chapter 5. We shall see in Chapter 9 that there are many ways to implement the ARM architecture, but the whole point of an architecture is to ensure that the programmer need not be concerned with particular implementation details. If a program works correctly on one implementation it should work correctly on them all (with certain provisos).

Assembly-level abstraction

A programmer who writes at the assembly programming level works (almost) directly with the raw machine instruction set, expressing the program in terms of instructions, addresses, registers, bytes and words.

A good programmer, faced with a non-trivial task, will begin by determining higher levels of abstraction that simplify the program design; for instance a graphics program may do a lot of line drawing, so a subroutine that will draw a line given the end coordinates will be useful. The rest of the program can then work just in terms of these end coordinates.

Abstraction is important, then, at the assembly programming level, but all the responsibility for supporting the abstraction and expressing it in terms of the machine primitives rests with the programmer, who must therefore have a good understanding of those primitives and be prepared to return frequently to think at the level of the machine.

High-level languages

A high-level language allows the programmer to think in terms of abstractions that are above the machine level; indeed, the programmer may not even know on which machine the program will ultimately run. Parameters such as the number of registers vary from architecture to architecture, so clearly these must not be reflected in the design of the language.

The job of supporting the abstractions used in the high-level language on the target architecture falls upon the *compiler*. Compilers are themselves extremely complex pieces of software, and the efficiency of the code they produce depends to a considerable extent on the support that the target architecture offers them to do their job.

At one time, the conventional wisdom was that the best way to support a compiler was to raise the complexity of the instruction set to implement the high-level operations of the language directly. The introduction of the RISC philosophy reversed that approach, focusing instruction set design on flexible primitive operations from which the compiler can build its high-level operations.

This chapter describes the requirements of high-level languages and shows how they are met by the ARM architecture, which is based on this RISC philosophy.

6.2 Data types

It is possible, though not very convenient, to express any computer program in terms of the basic Boolean logic variables 'true' (1) and 'false' (0). We can see that this is possible, since at the gate level that is all that the hardware can handle.

The definition of the ARM instruction set already introduces an abstraction away from logic variables when it expresses the functions of the processor in terms of instructions, bytes, words, addresses, and so on. Each of these terms describes a collection of logic variables viewed in a particular way. Note, for example, that an instruction, a data word and an address are all 32 bits long and a 32-bit memory location which contains one of these is indistinguishable from one that contains another of a different type. The difference is not in the way the information is stored but in the way it is used. A computer data type can therefore be characterized by:

- the number of bits it requires;
- the ordering of those bits;
- the uses to which the group of bits is put.

Some data types, such as addresses and instructions, exist principally to serve the purposes of the computer, whereas others exist to represent information in a way that is accessible to human users. The most basic of the latter category, in the context of computation, is the number.

Numbers

What started as a simple concept, presumably as a means of checking that no sheep had been stolen overnight, has evolved over the ages into a very complex mechanism capable of computing the behaviour of transistors a thousandth of a millimetre wide switching a hundred million times a second.

Roman numerals

Here is a number written by a human:

MCMXCV

Decimal numbers

The interpretation of this **Roman** numeral is complex; the value of a symbol depends on the symbol and its positional relationship to its neighbours. This way of writing a number has largely (but not entirely; see the page numbers in the front matter of this book) been replaced by the **decimal** scheme where the same number appears as:

1995

Here we understand that the right-hand digit represents the number of units, the digit to its left the number of tens, then hundreds, thousands, and so on. Each time we move left one place the value of the digit is increased by a factor of 10.

Binary coded decimal

To represent such a number as a group of Boolean variables, the simplest thing appears to be to find a representation for each digit and then use four of these to represent the whole number. We need four Boolean variables to be able to represent each digit from 0 to 9 differently, so the first form of this number that could easily be handled by logic gates is:

0001 1001 1001 0101

This is the binary coded decimal scheme which is supported by some computers and is commonly used in pocket calculators.

Binary notation

Most computers, most of the time, abandon the human-oriented decimal scheme altogether in favour of a pure binary notation where the same number becomes:

```
11111001011
```

Here the right-hand digit represents units, the next one 2, then 4, and so on. Each time we move left one place the value of the digit doubles. Since a value of 2 in one column can be represented by a value of 1 in the next column left, we never need a digit to have any value other than 0 or 1, and hence each binary digit (bit) can be represented by a single Boolean variable.

Hexadecimal notation

Although machines use binary numbers extensively internally, a typical 32-bit binary number is fairly unmemorable, but rather than convert it to the familiar decimal form (which is quite hard work and error-prone), computer users often describe the number in **hexadecimal** (base 16) notation. This is easy because the binary number can be split into groups of four binary digits and each group replaced by a hexadecimal number. Because, in base 16, we need symbols for numbers from 0 to 15, the early letters of the alphabet have been pressed into service where the decimal symbols run out: we use 0 to 9 as themselves and A to F to represent 10 to 15. Our number becomes:

7CB

(At one time it was common to use octal, base 8, notation in a similar role. This avoids the need to use alphabetic characters, but groups of three bits are less convenient to work with than groups of four, so the use of octal has largely been abandoned.)

Number ranges

When writing on paper, we use the number of decimal digits that are required to represent the number we want to write. A computer usually reserves a fixed number of bits for a number, so if the number gets too big it cannot be represented. The ARM deals efficiently with 32-bit quantities, so the first data type that the architecture supports is the 32-bit (unsigned) integer, which has a value in the range:

$$0 \text{ to } 4\,294\,967\,295_{10} = 0 \text{ to } \text{FFFFFFFF}_{16}$$

(The subscript indicates the number base, so the range above is expressed in decimal first and in hexadecimal second. Note how the hexadecimal value 'F' represents a binary number of all' 1 's.)

This looks like a large range, and indeed is adequate for most purposes, though the programmer must be aware of its limitations. Adding two unsigned numbers near the maximum value within the range will give the wrong answer, since the correct answer cannot be represented within 32 bits. The C flag in the program status register gives the only indication that something has gone wrong and the answer is not to be trusted.

If a large number is subtracted from a small number, the result will be negative and cannot be represented by an unsigned integer of any size.

Signed integers

In many cases it is useful to be able to represent negative numbers as well as positive ones. Here the ARM supports a 2's complement binary notation where the value of the top bit is made negative; in a 32-bit signed integer all the bits have the same value as they have in the unsigned case apart from bit 31, which has the value -2^{31} instead of$+2^{31}$. Now the range of numbers is:

$$-2\ 147\ 483\ 648_{10} \text{ to } +2\ 147\ 483\ 647_{10} = 80000000_{16} \text{ to } 7FFFFFFF,_{6}$$

Note that the sign of the number is determined by bit 31 alone, and the positive integer values have exactly the same representation as their unsigned equivalents.

The ARM, in common with most processors, uses the 2's complement notation for signed integers because adding or subtracting them requires exactly the same Boolean logic functions as are needed for unsigned integers, so there is no need to have separate instructions (the exception being multiplication with a full 64-bit result; here ARM *does* have separate signed and unsigned instructions).

The 'architectural support' for signed integers is the V flag in the program status registers which has no use when the operands are unsigned but indicates an **overflow** (out of range) error when signed operands are combined. The source operands cannot be out of range since they are represented as 32-bit values, but when two numbers near the extremes of the range are added or subtracted, the result could fall outside the range; its 32-bit representation will be an in-range number of the wrong value and sign.

Other number sizes

The natural representation of a number in the ARM is as a signed or unsigned 32-bit integer; indeed, internally to the processor that is all that there is. However, all ARM processors will perform unsigned 8-bit (byte) loads and stores, allowing small positive numbers to occupy a smaller space in memory than a 32-bit word, and all but the earliest versions of the processor also support signed byte and signed and unsigned 16-bit transfers, also principally to reduce the memory required to store small values.

Where a 32-bit integer is too small, larger numbers can be handled using multiple words and multiple registers. A 64-bit addition can be performed with two 32-bit

additions, using the C flag in the status register to propagate the carry from the lower word to the higher word:

```
; 64-bit addition of [r1,r0] to [r3,r2]
ADDS   r2, r2, r0       ; add low, save carry
ADC    r3, r3, r1       ; add high with carry
```

Real numbers

So far we have just considered integers, or whole numbers. 'Real' numbers are used to represent fractions and transcendental values that are useful when dealing with physical quantities.

The representation of real numbers in computers is a big issue that is deferred to the next section. An ARM core has no support for real data types, though ARM Limited has defined a set of types and instructions that operate on them. These instructions are either executed on a floating-point coprocessor or emulated in software.

Printable characters

After the number, the next most basic data type is the printable character. To control a standard printer we need a way to represent all the normal characters such as the upper and lower case alphabet, decimal digits from 0 to 9, punctuation marks and a number of special characters such as £, $, %, and so on.

ASCII

Counting all these different characters, the total rapidly approaches a hundred or so. Some time ago the binary representation of these characters was standardized in the 7-bit ASCII (American Standard for Computer Information Interchange) code, which includes these printable characters and a number of control codes whose names reflect the teletype origins of the code, such as 'carriage return', 'line feed', and 'bell'.

The normal way to store an ASCII character in a computer is to put the 7-bit binary code into an 8-bit byte. Many systems extend the code using, for example, the 8-bit ISO character set where the other 128 binary codes within the byte represent special characters (for instance, characters with accents). The most flexible way to represent characters is the 16-bit 'Unicode' which incorporates many such 8-bit character sets within a single encoding.

'1995' encoded as 8-bit printable characters is:

$$00110001\ 00111001\ 00111001\ 00110101 = 31\ 39\ 39\ 35_{16}$$

ARM support for characters

The support in the ARM architecture for handling characters is the unsigned byte load and store instructions; these have already been mentioned as being available to support small unsigned integers, but that role is rare compared with their frequency of use for transferring ASCII characters.

There is nothing in the ARM architecture that reflects the particular codes defined by ASCII; any other encoding is equally well supported provided it uses no more than eight bits. However, it would be perverse these days to choose a different code for characters without having a very good reason.

Byte ordering

The above ASCII example highlights an area of some potential difficulty. It is written to be read from left to right, but if it is read as a 32-bit word, the least significant byte is at the right. A character output routine might print characters at successive increasing byte addresses, in which case, with 'little-endian' addressing, it will print '5991'. We clearly need to be careful about the order of bytes within words in memory.

(ARMs can operate with either little- or big-endian addressing. See 'Memory organization'on page 106.)

High-level languages

A high-level language defines the data types that it needs in its specification, usually without reference to any particular architecture that it may run on. Sometimes the number of bits used to represent a particular data type is architecture-dependent in order to allow a machine to use its most efficient size.

ANSI C basic data types

The dialect of the 'C' language defined by the American National Standards Institute (ANSI), and therefore known as 'ANSI standard C' or simply 'ANSI C', defines the following basic data types:

- Signed and unsigned **characters** of at least eight bits.
- Signed and unsigned **short integers** of at least 16 bits.
- Signed and unsigned **integers** of at least 16 bits.
- Signed and unsigned **long integers** of at least 32 bits.
- **Floating-point, double** and **long double** floating-point numbers.
- **Enumerated** types.
- **Bitfields.**

The ARM C compiler adopts the minimum sizes for each of these types except the standard integer, where it uses 32-bit values since this is the most frequently used data type and the ARM supports 32-bit operations more efficiently than 16-bit operations.

Enumerated types (where variables have one value out of a specified set of possible values) are implemented as the smallest integer type with the necessary range of values. Bitfield types (sets of Boolean variables) are implemented within integers; several may share one integer, with the first declared holding the lowest bit position, but may not straddle word boundaries.

ANSI C derived data types

In addition, the ANSI C standard defines derived data types:

- **Arrays** of several objects of the same type.
- **Functions** which return an object of a given type.
- **Structures** containing a sequence of objects of various types.
- **Pointers** (which are usually machine addresses) to objects of a given type.

• Unions which allow objects of different types to occupy the same space at different times.

ARM pointers are 32 bits long (the size of ARM native addresses) and resemble unsigned integers, though note that they obey different arithmetic rules.

The ARM C compiler aligns characters on byte boundaries (that is, at the next available address), short integers at even addresses and all other types on word boundaries. A structure always starts on a word boundary with the first-named component, then subsequent components are packed as closely as possible consistent with these alignment rules. (Packed structures violate the alignment rules; use of memory is discussed more extensively in Section 6.9 on page 180.)

ARM architectural support for C data types

We have seen above that the ARM integer core provides native support for signed and unsigned 32-bit integers and for unsigned bytes, covering the C integer (given the decision to implement these as 32-bit values), long integer and unsigned character types. Pointers are implemented as native ARM addresses and are therefore supported directly.

The ARM addressing modes provide reasonable support for arrays and structures: base plus scaled index addressing allows an array of objects of a size which is 2" bytes to be scanned using a pointer to the start of the array and a loop variable as the index; base plus immediate offset addressing gives access to an object within a structure. However, additional address calculation instructions will be necessary for more complex accesses.

Current versions of the ARM include signed byte and signed and unsigned 16-bit loads and stores, providing some native support for short integer and signed character types.

Floating-point types are discussed in the next section, however here we can note that the basic ARM core offers little direct support for them. These types (and the instructions that manipulate them) are, in the absence of specific floating-point support hardware, handled by complex software emulation routines.

6.3 Floating-point data types

Floating-point numbers attempt to represent real numbers with uniform accuracy. A generic way to represent a real number is in the form:

$$R = a \times b^n \qquad \text{Equation 13}$$

where n is chosen so that a falls within a defined range of values; b is usually implicit in the data type and is often equal to 2.

IEEE 754

There are many complex issues to resolve with the handling of floating-point numbers in computers to ensure that the results are consistent when the same program is run on different machines. The consistency problem was greatly aided by the introduction in 1985 of the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 754-1985, sometimes referred to simply as IEEE 754) which defines in considerable detail how floating-point numbers should be represented, the accuracy with which calculations should be performed, how errors should be detected and returned, and so on.

The most compact representation of a floating-point number defined by IEEE 754 is the 32-bit 'single precision' format:

Single precision

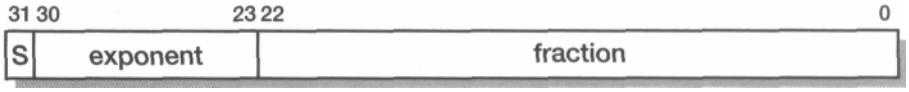


Figure 6.1 IEEE 754 single precision floating-point number format.

The number is made up from a **sign bit** ('S'), an **exponent** which is an unsigned integer value with a 'bias' of+127 (for normalized numbers) and a **fractional** component. A number of terms in the previous sentence may be unfamiliar. To explain them, let us look at how a number we recognize,' 1995', is converted into this format.

We start from the binary representation of 1995, which has already been presented:

11111001011 This is a positive number,

so the S bit will be zero.

Normalized numbers

The first step is to **normalize** the number, which means convert it into the form shown in Equation 13 on page 158 where $1 < a < 2$ and $b = 2$. Looking at the binary form of the number, a can be constrained within this range by inserting a 'binary point' (similar in interpretation to the more familiar decimal point) after the first' 1'. The implicit position of the binary point in the binary integer representation is to the right of the right-most digit, so here we have to move it left ten places. Hence the normalized representation of 1995 is:

$$1995 = 1.111100101\ \mathrm{Ix}2^{10} \qquad \text{Equation 14}$$

where a and n are both in binary notation.

When any number is normalized, the bit in front of the binary point in a will be a ' 1' (otherwise the number is not normalized). Therefore there is no need to store this bit.

Exponent bias

Finally, since the format is required to represent very small numbers as well as very large ones, some numbers may require negative exponents in their normalized form. Rather than use a signed exponent, the standard specifies a 'bias'. This bias (+127 for single precision normalized numbers) is added to the exponent value. Hence 1995 is represented as:

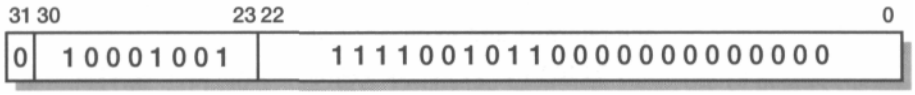


Figure 6.2 IEEE 754 single precision representation of '1995'.

The exponent is 127+10 = 137; the fraction is zero-extended to the right to fill the 23-bit field.

Normalized value

In general, the value of a 32-bit normalized number is given by:

$$\text{value (norm)} = (-1)^{S} \times 1.\text{fraction} \times 2^{(\text{exponent}-127)} \qquad \text{Equation 15}$$

Although this format represents a wide range of values efficiently, it has one rather glaring problem: there is no way to represent zero. Therefore the IEEE 754 standard reserves numbers where the exponent is either zero or 255 to represent special values:

- Zero is represented by a zero exponent and fraction (but either sign value, so pos itive and negative zeros can be represented).
- Plus or minus infinity are represented by the maximum exponent value with a zero fraction and the appropriate sign bit.
- NaN (Not a Number) is indicated by the maximum exponent and a non-zero fraction; 'quiet' NaNs have a T in the most significant fraction bit position and 'signalling' NaNs have a '0' in that bit (but a T somewhere else, otherwise they look like infinity).
- Denormalized numbers, which are numbers that are just too small to normalize within this format, have a zero exponent, a non-zero fraction and a value given by:

$$\text{value (denorm)} = (-1)^{s} \times 0.\text{fraction} \times 2^{(-126)} \qquad \text{Equation 16}$$

The 'NaN' format is used to represent the results of invalid floating-point operations, such as taking the logarithm of a negative number, and prevents a series of operations from producing an apparently valid result when an intermediate error condition was not checked.

Double precision

For many purposes the accuracy offered by the single precision format is inadequate. Greater accuracy may be achieved by using the double precision format which uses 64 bits to store each floating-point value.

The interpretation is similar to that for single precision values, but now the exponent bias for normalized numbers is +1023:

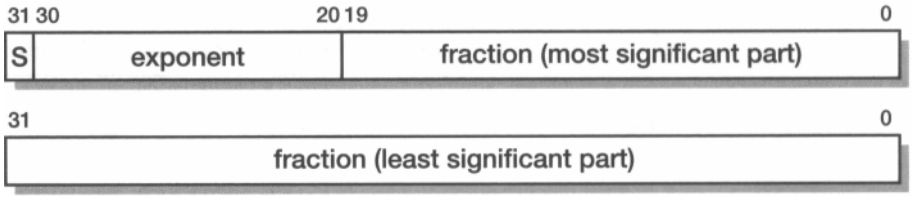


Figure 6.3 IEEE 754 double precision floating-point number format.

Double extended precision

Even greater accuracy is available from the double extended precision format, which uses 80 bits of information spread across three words. The exponent bias is 16383, and the J bit is the bit to the left of the binary point (and is a T for all normalized numbers):

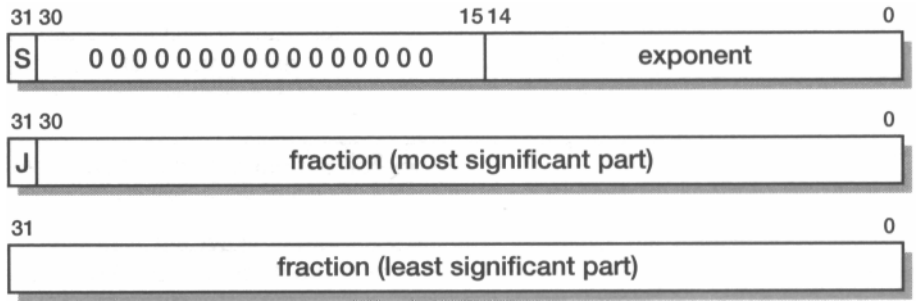


Figure 6.4 IEEE 754 double extended precision floating-point number format.

Packed decimal

In addition to the binary floating-point representations detailed above, the IEEE 754 standard also specifies packed decimal formats. Referring back to Equation 13 on page 158, in these packed formats b is 10 and a and n are stored in a binary coded decimal format as described in 'Binary coded decimal' on page 154. The number is normalized so that $1 < a < 10$. The packed decimal format is shown on page 162:

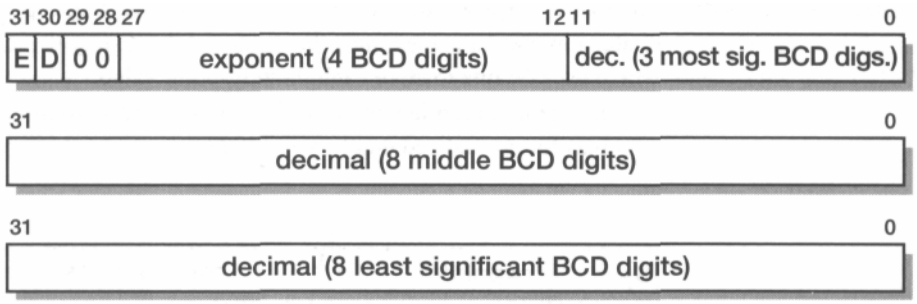


Figure 6.5 IEEE 754 packed decimal floating-point number format.

The sign of the exponent is held in bit 31 of the first word ('E') and the sign of the decimal in bit 30 ('D'). The value of the number is:

$$\text{value (packed)} = (-1)^{\circ} \times \text{decimal} \times 1\ o^{\prime\prime\prime 1} >^{*x\ \text{exponent})} \qquad \text{Equation 17}$$

Extended packed decimal

The extended packed decimal format occupies four words to give higher precision. The value is still as given by Equation 17 above and the format is:

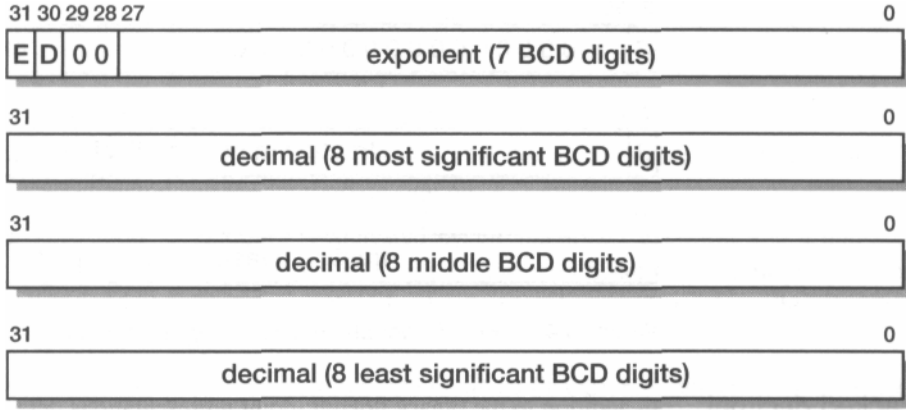


Figure 6.6 IEEE 754 extended packed decimal floating-point number format.

ARM floating-point instructions

Although there is no direct support for any of these floating-point data types in a standard ARM integer core, ARM Limited has defined a set of floating point instructions within the coprocessor instruction space. These instructions are normally implemented entirely in software through the undefined instruction trap (which collects any coprocessor instructions that are not accepted by a hardware coprocessor), but a subset may be handled in hardware by the FPA10 floating-point coprocessor.

ARM floating-point library

As an alternative to the ARM floating-point instruction set (and the only option for Thumb code), ARM Limited also supplies a C floating-point library which supports IEEE single and double precision formats. The C compiler has a flag to select this route which produces code that is both faster (by avoiding the need to intercept, decode and emulate the floating-point instructions) and more compact (since only the functions which are used need be included in the image) than software emulation.

6.4 The ARM floating-point architecture

Where full floating-point support is required, the ARM floating-point architecture provides extensive support for the data types described in the previous section either entirely in software or using a combined software/hardware solution based around the FPA10 floating-point accelerator.

The ARM floating-point architecture presents:

- An interpretation of the coprocessor instruction set when the coprocessor number is 1 or 2. (The floating-point system uses two logical coprocessor numbers.)
- Eight 80-bit floating-point registers in coprocessors 1 and 2 (the same physical registers appear in both logical coprocessors).
- A user-visible floating-point status register (FPSR) which controls various oper ating options and indicates error conditions.
- Optionally, a floating-point control register (FPCR) which is user-invisible and should be used only by the support software specific to the hardware accelerator.

Note that the ARM coprocessor architecture allows the floating-point emulator (FPE) software to be used interchangeably with the combination of the FPA10 and the floating-point accelerator support code (FPASC), or any other hardware-software combination that supports the same set of instructions. Application binaries will work with either support environment, though the compiler optimization strategy is different (the FPE software works best with grouped floating-point instructions whereas the FPA10/FPASC works best with distributed instructions).

FPAIOdata types

The ARM FPA10 hardware floating-point accelerator supports single, double and extended double precision formats. Packed decimal formats are supported only by software.

The coprocessor registers are all extended double precision, and all internal calculations are carried out in this, the highest precision format, except that there are faster versions of some instructions which do not produce the full 80-bit accuracy. However loads and stores between memory and these registers can convert the precision as required.

This is similar to the treatment of integers in the ARM integer architecture, where all internal operations are on 32-bit quantities, but memory transfers can specify bytes and half-words.

Load and store floating instructions

Since there are only eight floating-point registers, the register specifier field in the coprocessor data transfer instruction (shown in Figure 5.16 on page 138) has a spare bit which is used here as an additional data size specifier:

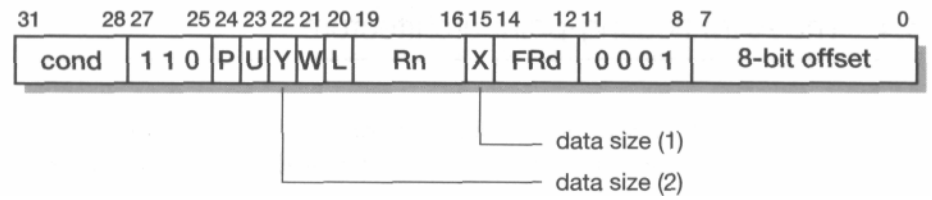


Figure 6.7 Load and store floating binary encoding.

The other fields in this format are described in Section 5.18 on page 138. The X and Y bits allow one of four precisions to be specified, choosing between single, double, double extended and packed decimal. (The choice between packed decimal and extended packed decimal is controlled by a bit in the FPSR.)

Load and store multiple floating

The load and store multiple floating-point registers instructions are used to save and restore the floating-point register state. Each register is saved using three memory words, and the precise format is not defined; it is intended that the only use for the saved values will be to reload them using the equivalent load multiple floating instruction to restore the context. 'FRd' specifies the first register to be transferred, and 'X' and 'Y' encode the number of registers transferred which can be from one to four. Note that these instructions use coprocessor number 2, whereas the other floating-point instructions use coprocessor number 1.

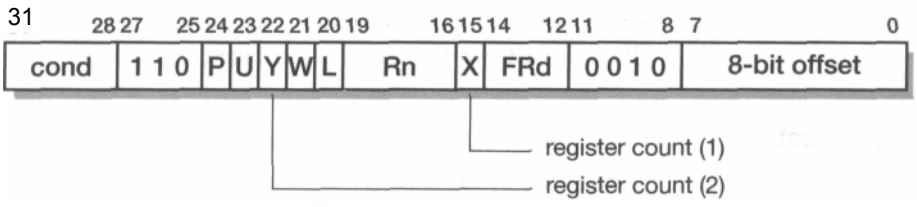


Figure 6.8 Load and store multiple floating binary encoding.

Floating-point data operations

The floating-point data operations perform arithmetic functions on values in the floating-point registers; their only interaction with the outside world is to confirm that they should complete through the ARM coprocessor handshake. (Indeed, a floating-point coprocessor may begin executing one of these instructions before the handshake begins so long as it waits for confirmation from the ARM before committing a state change.)

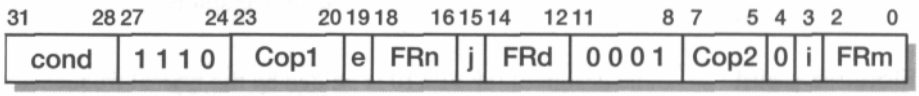


Figure 6.9 Floating-point data processing binary encoding.

The instruction format has a number of opcode bits, augmented by extra bits from each of the three register specifier fields since only three bits are required to specify one of the eight floating-point registers:

- 'i' selects between a register ('FRm') or one of eight constants for the second operand.
- 'e' and 'Cop2' control the destination size and the rounding mode.
- 'j' selects between monadic (single operand) and dyadic (two operand) operations.

The instructions include simple arithmetic operations (add, subtract, multiply, divide, remainder, power), transcendental functions (log, exponential, sin, cos, tan, arcsin, arccos, arctan) and assorted others (square root, move, absolute value, round).

Floating-point register transfers

The register transfer instructions accept a value from or return a value to an ARM register. This is generally combined with a floating-point processing function.

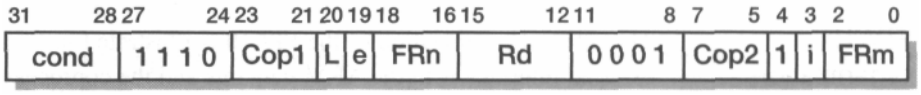


Figure 6.10 Floating-point register transfer binary encoding.

Transfers from ARM to the floating-point unit include 'float' (convert an integer in an ARM register to a real in a floating-point register) and writes to the floating-point status and control registers; going the other way there is 'fix' (convert a real in a floating-point register to an integer in an ARM register) and reads of the status and control registers.

The floating-point compare instructions are special cases of this instruction type, where Rd is r15. Two floating-registers are compared and the result of the comparison

is returned to the N, Z, C and V flags in the ARM CPSR where they can directly control the execution of conditional instructions:

- N indicates 'less than'.
- Z indicates equality.
- C and V indicate more complex conditions, including 'unordered' comparison results which can arise when an operand is a 'NaN' (not a number).

Floating-point instruction frequencies

The design of the FPAIO is guided by the typical frequencies of the various floating-point instructions. These were measured running compiled programs using the floating-point emulator software and are summarized in Table 6.1.

The statistics are dominated by the number of load and store operations that take place. As a result, the FPAIO has been designed to allow these to operate concurrently with internal arithmetic operations.

Table 6.1 Floating-point instruction frequencies.

Instruction	Frequency
Load/store	67%
Add	13%
Multiply	10.5%
Compare	3%
Fix and float	2%
Divide	1.5%
Others	3%

FPA10 organization

The internal organization of the FPAIO is illustrated in Figure 6.11 on page 167. Its external interface is to the ARM data bus and the coprocessor handshake signals, so it has a modest pin-count requirement. The major components are:

- The coprocessor pipeline follower (see Section 4.5 on page 101).
- The load/store unit that carries out format conversion on floating-point data types as they are loaded from and stored to memory.
- The register bank which stores eight 80-bit extended precision floating-point operands.
- The arithmetic unit which incorporates an adder, a multiplier and a divider, together with rounding and normalizing hardware.

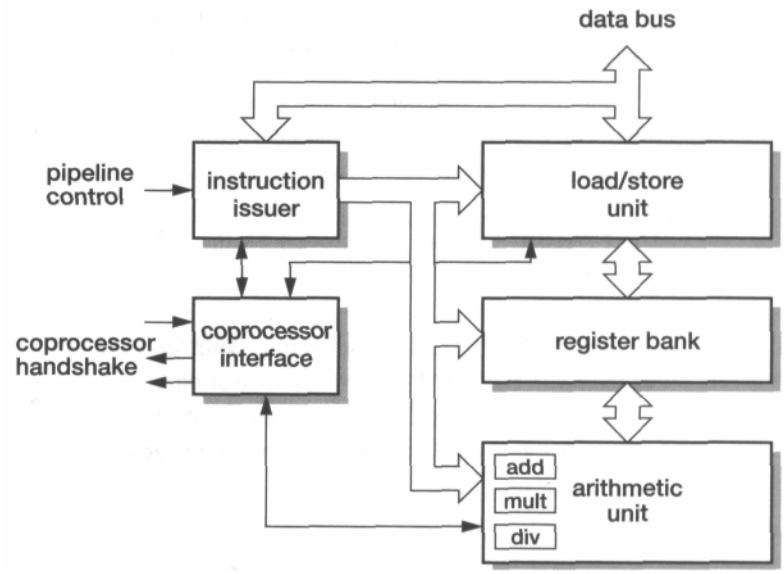


Figure 6.11 FPA10 internal organization.

The load/store unit operates concurrently with the arithmetic unit, enabling new operands to be loaded from memory while previously loaded operands are being processed. Hardware interlocks protect against data hazards.

FPA10 pipeline

The FPA10 arithmetic unit operates in four pipeline stages:

1. Prepare: align operands.
2. Calculate: add, multiply or divide.
3. Align: normalize the result.
4. Round: apply appropriate rounding to the result.

A floating-point operation can begin as soon as it is detected in the instruction pipeline (that is, before the ARM handshake has occurred), but the result write-back must await the handshake.

Floating-point context switches

The FPA registers represent additional process state which must be saved and restored across context switches. However, typically only a small number of active processes use floating-point instructions. Therefore saving and restoring the FPA registers on every switch is an unnecessary overhead. Instead, the software minimizes the number of saves and restores by the following algorithm:

- When a process using the FPA is switched out, the FPA registers are not saved but the FPA is turned off.

• If a subsequent process executes a floating-point instruction this will trap; the trap code will save the FPA state and enable the FPA.

Thus the FPA state saving and restoring overhead is only incurred for processes which use the FPA; in a typical system with only one process using the FPA, the overhead is avoided altogether.

FPA10 applications

The FPA10 is used as a macrocell on the ARM7500FE chip (see Section 13.5 on page 360).

TheVFPIO

A much higher performance floating-point unit, the VFP10, has been designed to operate with the ARM10TDMI processor core (see Section 12.6 on page 341). The VFP10 supports a different floating-point instruction set from the FPA 10 that includes support for vector floating-point operations.

6.5 Expressions

Unsigned arithmetic on an n-bit integer is defined in ANSI C to be modulo 2", so overflow cannot occur. Therefore the basic ARM integer data processing instructions implement most of the C integer arithmetic, bit-wise and shift primitives directly. Exceptions are division and remainder which require several ARM instructions.

Register use

Since all data processing instructions operate only on values in register, the key to the efficient evaluation of a complex expression is to get the required values into the registers in the right order and to ensure that frequently used values are normally resident in registers. There is clearly a trade-off between the number of values that can be held in registers and the number of registers remaining for intermediate results during expression evaluation (remembering that registers are also needed to hold the addresses of operands that must be fetched from memory). Optimizing this trade-off is a major task for the compiler, as is sorting out the right order to load and combine operands to achieve the result prescribed by the operator precedence defined in the language.

ARM support

The 3-address instruction format used by the ARM gives the compiler the maximum flexibility in how it preserves or re-uses registers during expression evaluation.

Thumb instructions are generally 2-address, which restricts the compiler's freedom to some extent, and the smaller number of general registers also makes its job harder (and will result in less efficient code).

Accessing operands

A procedure will normally work with operands that are presented in one of the following ways, and can be accessed as indicated:

Pointer arithmetic

1. As an argument passed through a register.

   The value is already in a register, so no further work is necessary.
2. As an argument passed on the stack.

   Stack pointer (r13) relative addressing with an immediate offset known at compile-time allows the operand to be collected with a single LDR.
3. As a constant in the procedure's literal pool.

   PC-relative addressing, again with an immediate offset known at compile-time, gives access with a single LDR.
4. As a local variable.

   Local variables are allocated space on the stack and are accessed by a stack pointer relative LDR.
5. As a global variable.

   Global (and static) variables are allocated space in the static area and are accessed by static base relative addressing. The static base is usually in r9 (see the 'ARM Procedure Call Standard' on page 176).

If the value that is passed is a pointer, an additional LDR (with an immediate offset) may be required to access an operand within the structure that it points to.

Arithmetic on pointers depends on the size of the data type that the pointers are pointing to. If a pointer is incremented it changes in units of the size of the data item in bytes. Thus:

```
int    *p;
P  =  P+l;
```

will increase the value of p by 4 bytes. Since the size of a data type is known at compile-time, the compiler can scale constants by an appropriate amount. If a variable is used as an offset it must be scaled at run-time:

```
int  i  =  4 ;
p  =  p  +
i;
```

If p is held in r0 and i in r1, the change to p may be compiled as:

```
ADD   r0, r0, r1, LSL #2 ; scale r1 to int
```

Where the data type is a structure with a size which is not a power of 2 bytes, a multiplication by a small constant is required. The shift and add instructions can usually produce the desired product in a small number of operations, using a temporary register where necessary.

Arrays

Arrays in C are little more than a shorthand notation for pointer operations, so the above comments apply here too. The declaration:

```
int    a[10];
```

establishes a name, a, for the array which is just a pointer to the first element, and a reference to a [i] is equivalent to the pointer-plus-offset form * (a+ i); the two may be used interchangeably.

6.6 Conditional statements

Conditional statements are executed if the Boolean result of a test is true (or false); in C these include if...else statements and switches (C 'case' statements).

if...else

The ARM architecture offers unusually efficient support for conditional expressions when the conditionally executed statement is small.

For example, here is a C statement to find the maximum of two integers:

```
if (a>b) c=a; else c=b;
```

If the variables a, b and c are in registers r0, r1 and r2, the compiled code could be as simple as:

```
CMP     r0, r1          ;  if (a>b)... ;
MOVGT   r2, r0          ;    ..c=a..
MOVLE   r2, rl          ;    ..else c=b
```

(In this particular case the C compiler can produce a rather more obvious sequence:

```
MOV CMP r2, r0 r0,      ;  c=a
MOVLE   r1 r2,          ;  if (a>b)...
        r1              ;     ...c=b
```

However, this doesn't scale well to more complex 'if statements so we will ignore it in this general discussion.)

The 'if and 'else' sequences may be a few instructions long with the same condition on each instruction (provided none of the conditionally executed instructions changes the condition codes), but beyond two or three instructions it is generally better to fall back on the more conventional solution:

```
        CMP     r0, rl          ; if (a>b)...
        BLE     ELSE            ; skip clause if false
        MOV     r2, r0          ; ..c=a..
        B       ENDIF           ; skip else clause
ELSE    MOV     r2, rl          ; ...else c=b
ENDIF
```

Here the 'if and 'else' sequences may be any length and may use the condition codes freely (including, for example, for nested if statements) since they are not required beyond the branch immediately following the compare instruction.

Note, however, that whichever branch is taken, this second code sequence will take approximately twice as long to execute as the first for this simple example. Branches are expensive on the ARM, so the absence of them from the first sequence makes it very efficient.

switches

A switch, or case, statement extends the two-way decision of an if...else statement to many ways. The standard C form of a switch statement is:

```
switch (expression) {
  case constant-expression]: statements; case
  constant-expressio^: statementS2
  ...
  case constant-expression^: statements^
  default: statements^
}
```

Normally each group of statements ends with a 'break' (or a 'return') to cause the switch statement to terminate, otherwise the C semantics cause execution to fall through to the next group of statements. A switch statement with 'breaks' can always be translated into an equivalent series of if..else statements:

```
temp = expression;
if (temp==constant-expressionj) {statements]}
else ...
else if (temp==constant-expressionN) {statements^}
else {statementsj)}
```

However this can result in slow code if the switch statement has many cases. An alternative is to use *a jump table.* In its simplest form a jump table contains a target address for each possible value of the switch expression:

```
        ; r0 contains value of expression
        ADR     r1, JUMPTABLE       ; get base of jump table
        CMP     r0, #TABLEMAX       ; check for overrun..
        LDRLS   pc, [r1,r0,LSL #2]; .. if OK get pc
        ; statements_D             ; .. otherwise default
        B       EXIT                ; break
L1      ..      ; statements_1
        B       EXIT                ; break
        ..
LN      ..      ; statements_N
EXIT    ..
```

Clearly it is not possible for the jump table to contain an address for every possible value of a 32-bit integer, and equally clearly it is vital that the jump table look-up does not fall past the end of the table, so some checking is necessary.

Another way of compiling a switch statement is illustrated by a procedure in the 'Dhrystone' benchmark program which ends (effectively) as follows:

```
  switch (a) {
    case 0: *b = 0; break;
    case 1: if (c>100) *b = 0; else *b = 3; break;
    case 2: *b = 1; break
    case 3: break;
  case 4: *b = 2; break; } /*
end of switch */ } /* end of
procedure */
```

The code which is generated highlights a number of aspects of the ARM instruction set. The switch statement is implemented by adding the value of the expression (in v2; the register naming convention follows the ARM Procedure Call Standard which will be described in Section 6.8 on page 176) to the PC after scaling it to a word offset. The overrun case falls through the add instruction into the slot left free by the PC offset. Any cases which require a single instruction (such as case 3), and the last case whatever its length, can be completed in line; others require a branch. This example also illustrates an if...then...else implemented using conditional instructions.

```
          ; on entry a1 = 0, a2 = 3, v2 = switch expression
          CMP     v2,#4                  check value for overrun..
          ADDLS   pc,pc,v2,LSL #2      ..if OK, add to pc (+8)
          LDMDB   fp,{v1,v2,fp,sp,pc} ;  ..if not OK, return
          B       L0                     case 0
          B       L1                     case 1
          B       L2                     case 2
          LDMDB   fp,{v1,v2,fp,sp,pc} ; case 3 (return)
          MOV     a1,#2                  case 4
          STR     a1,[v1]
          LDMDB   fp,{v1,v2,fp,sp,pc} ; return
L0      STR       a1,[v1]
          LDMDB   fp,{v1,v2,fp,sp,pc} ; return
L1      LDR       a3,c ADDR        ; get address of c
          LDR     a3, [a3]              ; get c
          CMP     a3,#&64               ; c>100?..
          STRLE   a2, [v1]              ;  .. No:  *b = 3
          STRGT   a1, [v1]              ;  .. Yes: *b = 0
          LDMDB   fp,{v1,v2,fp,sp,pc) ; return
c_ADDR  DCD       <address of c>
L2      MOV       a1,ttl
          STR     a1,[v1]               ; *b = 1
          LDMDB   fp,{v1,v2,fp,sp,pc} ; return
```

6.7 Loops

The C language supports three forms of loop control structure:

- for(el;e2;e3){..}
- while (el) {..}
- do {..} while (el)

Here el, e2 and e3 are expressions which evaluate to 'true' or 'false' and {..} is the body of the loop which is executed a number of times determined by the control structure. Often the body of a loop is very simple, which puts the onus on the compiler to minimize the overhead of the control structure. Each loop must have at least one branch instruction, but any more would be wasteful.

for loops

A typical 'for' loop uses the control expressions to manage an index:

```
for (i=0; i<10; i++) {a[i] = 0}
```

The first control expression is executed once before the loop begins, the second is tested each time the loop is entered and controls whether the loop is executed, and the third is executed at the end of each pass through the loop to prepare for the next pass. This loop could be compiled as:

```
        MOV     r1, #0              ; value to store in a[i]
        ADR     r2, a[0]            ; r2 points to a[0]
        MOV     r0, #0              ; i=0
LOOP    CMP     r0, #10             ; i<10 ?
        BGE     EXIT                ; if i >= 10 finish
        STR     r1, [r2,r0,LSL #2]; a[i] = 0
        ADD     r0, r0, #1          ; i++
        B       LOOP
EXIT    ..
```

(This example illustrates the optimization technique of *strength reduction,* which means moving fixed operations outside the loop. The first two instructions are logically inside the loop in the C program, but have been moved outside since they are initializing registers to the same constants each time round the loop.)

This code may be improved by omitting the conditional branch to EXIT and applying the opposite condition to the following instructions, causing the program to fall through rather than branch round them when the terminating condition is met. However, it can be further improved by moving the test to the bottom of the loop (which is possible here since the initialization and test are with constants, so the compiler can be sure that the loop will be executed at least once).

while loops

A 'while' loop has a simpler structure and generally controls loops where the number of iterations is not a constant or clearly denned by a variable at run-time. The standard conceptual arrangement of a 'while' loop is as follows:

```
LOOP   ..              ; evaluate expression
       BEQ    EXIT
        ..             ; loop body
       B      LOOP
EXIT
```

The two branches appear to be necessary since the code must allow for the case where the body is not executed at all. A little reordering produces more efficient code:

```
       B      TEST
LOOP  . .              ; loop body
TEST   ..              ; evaluate expression
       BNE    LOOP
EXIT
```

This second code sequence executes the loop body and evaluates the control expression exactly the same way as the original version and has the same code size, but it executes one fewer (untaken) branch per iteration, so the second branch has been removed from the loop overhead. An even more efficient code sequence can be produced by the compiler:

```
        ..             ; evaluate expression
       BEQ      EXIT   ; skip loop if necessary
LOOP  . .              ; loop body
TEST  ..               ; evaluate expression
       BNE          LOOP
EXIT
```

The saving here is that one fewer branch is executed each time the complete 'while' structure is encountered (assuming that the body is executed at least once). This is a modest gain and costs extra instructions, so it is worthwhile only if performance matters much more than code size.

do..while loops

The conceptual arrangement of a 'do..while' loop is similar to the improved 'while' loop above, but without the initial branch since the loop body is executed before the test (and is therefore always executed at least once):

```
LOOP  . .              ; loop body
                ; evaluate expression BNE
LOOP EXIT
```

6.8 Functions and procedures

Program design

Good programming practice requires that large programs are broken down into components that are small enough to be thoroughly tested; a large, monolithic program is too complex to test fully and is likely to have 'bugs' in hidden corners that do not emerge early enough in the program's life to be fixed before the program is shipped to users.

Each small software component should perform a specified operation using a well-defined interface. How it performs this operation should be of no significance to the rest of the program (this is the principle **of abstraction;** see Section 6.1 on page 152).

Program hierarchy

Furthermore, the full program should be designed as a **hierarchy** of components, not simply a flat list.

A typical hierarchy is illustrated in Figure 6.12. The top of the hierarchy is the program called *main.* The remaining hierarchy is fairly informal; lower-level routines may be shared by higher-level routines, calls may skip levels, and the depth may vary across the hierarchy.

Leaf routines

At the lowest level of the hierarchy there are **leaf** routines; these are routines which do not themselves call any lower-level routines. In a typical program some of the bottom-level routines will be **library or system** functions; these are predefined

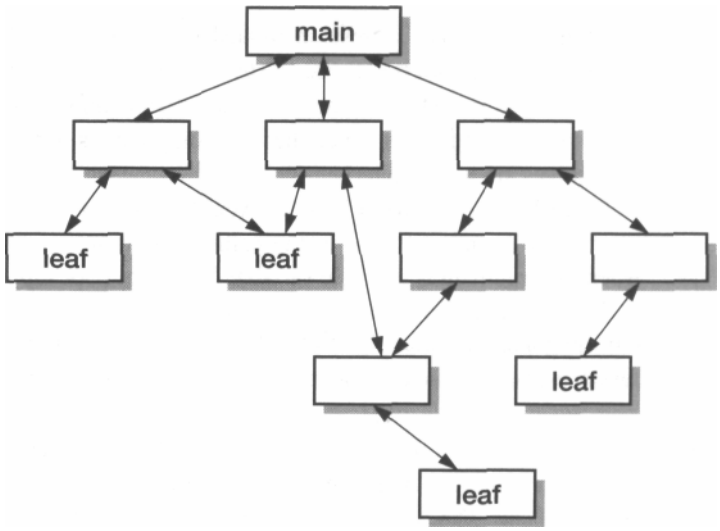


Figure 6.12 Typical hierarchical program structure.

operations which may or may not be leaf routines (that is, they may or may not have internal structure).

Terminology

There are several terms that are used to describe components of this program structure, often imprecisely. We shall attempt to apply terms as follows:

- **Subroutine:** a generic term for a routine that is called by a higher-level routine, particularly when viewing a program at the assembly language level.
- **Function:** a subroutine which returns a value through its name. A typical invocation looks like:

```
c = max (a, b);
```

- **Procedure:** a subroutine which is called to carry out some operation on specified data item(s). A typical invocation looks like:

```
printf ("Hello WorldXn");
```

C functions

Some programming languages make a clear distinction between functions and procedures, but C does not. In C all subroutines are functions, but they can have side-effects in addition to returning a value, and when the returned value is of type 'void' it is effectively suppressed and only the side-effects remain, giving a behaviour which looks just like a procedure.

Arguments and parameters

An **argument** is an expression passed to a function call; a value received by the function is a **parameter.** C uses a strict 'call by value' semantics, so a copy is made of each argument when a function is called and, though the function may change the values of its parameters, since these are only copies of the arguments the arguments themselves are not affected.

(A **call by reference** semantics would cause any change to a parameter within a function to be passed back to the calling program, which clearly only makes sense when the argument is a simple variable, but C does not support this.)

The way a C function can change data within the calling program, other than by returning a single value, is when it is passed a pointer to the data as an argument. The function can then use the pointer to access and modify the data structure.

ARM Procedure Call Standard

In order to support flexible mixing of routines generated by different compilers and written in assembly language, ARM Limited has defined a set of rules for procedure entry and exit. The ARM Procedure Call Standard (APCS) is employed by the ARM C compiler, though this is of significance to the C programmer only when the assembly-level output must be understood in detail.

The APCS imposes a number of conventions on the otherwise 'vanilla' flavour of the ARM architecture:

- It defines particular uses for the 'general-purpose' registers.
- It defines which form of stack is used from the full/empty, ascending/descending choices supported by the ARM instruction set.
- It defines the format of a stack-based data structure used for back-tracing when debugging programs.
- It defines the function argument and result passing mechanism to be used by all externally visible functions and procedures. ('Externally visible' means that the procedure interface is offered outside the current programming module. A func tion which is used only within the current module may be optimized by deviating from this convention.)
- It supports the ARM shared library mechanism, which means it supports a standard way for shared (re-entrant) code to access static data.

ARCS register usage

The convention for the use of the 16 currently visible ARM registers is summarized in Table 6.2. The registers are divided into three sets:

Table 6.2 APCS register use convention.

Register	APCS name	APCS role
0	al	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	vl	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	Ir	Link address / scratch register
15	pc	Program counter

1. Four argument registers which pass values into the function.

   The function need not preserve these so it can use them as scratch registers once it has used or saved its parameter values. Since they will not be preserved across any calls this function makes to other functions, they must be saved across such calls if they contain values that are needed again. Hence, they are **caller-saved** register variables when so used.

2. Five (to seven) register variables which the function must return with unchanged values.

   These are **callee-saved** register variables. This function must save them if it wishes to use the registers, but it can rely on functions it calls not changing them.

3. Seven (to five) registers which have a dedicated role, at least some of the time.

   The link register (Ir), for example, carries the return address on function entry, but if it is saved (as it must be if the function calls subfunctions) it may then be used as a scratch register.

APCS variants

There are several (16) different variants of the APCS which are used to generate code for a range of different systems. They support:

- 32-or 26-bit PCs.

  Older ARM processors operated in a 26-bit address space and some later versions continue to support this for backwards compatibility reasons.

- Implicit or explicit stack-limit checking.

  Stack overflow must be detected if code is to operate reliably. The compiler can insert instructions to perform explicit checks for overflow.

  Where memory management hardware is available, an ARM system can allocate memory to the stack in units of a page. If the next logical page is mapped out, a stack overflow will cause a data abort and be detected. Therefore the memory management unit can perform stack-limit checking and there is no need for the compiler to insert instructions to perform explicit checks.

- Two ways to pass floating-point arguments.

  The ARM floating-point architecture (see Section 6.4 on page 163) specifies a set of eight floating-point registers. The APCS can use these to pass floating-point arguments into functions, and this is the most efficient solution when the system makes extensive use of floating-point variables. If, however, the system makes little or no use of floating-point types (and many ARM systems do not) this approach incurs a small overhead which is avoided by passing floating-point arguments in the integer registers and/or on the stack.

- Re-entrant or non-re-entrant code.

  Code specified as re-entrant is position-independent and addresses all data indirectly through the static base register (sb). This code can be placed in a ROM and

can be shared by several client processes. Generally, code to be placed in a ROM or a shared library should be re-entrant whereas application code will not be.

Argument passing

A C function may have many (or even a variable number of) arguments. The APCS organizes the arguments as follows:

1. If floating-point values are passed through floating-point registers, the first four floating-point arguments are loaded into the first four floating-point registers.
2. All remaining arguments are organized into a list of words; the first four words are loaded into al to a4, then the remaining words are pushed onto the stack in reverse order.

Note that multi-word arguments, including double precision floating-point values, may be passed in integer registers, on the stack, or even split across the registers and the stack.

Result return

A simple result (such as an integer) is returned through al. A more complex result is returned in memory to a location specified by an address which is effectively passed as an additional first argument to the function through al.

Function entry and exit

A simple leaf function which can perform all its functions using only al to a4 can be compiled into code with a minimal calling overhead:

```
        BL      leaf1
        ..
leaf1   ..
        MOV     pc, lr  ; return
```

In typical programs somewhere around 50% of all function calls are to leaf functions, and these are often quite simple.

Where registers must be saved, the function must create a stack frame. This can be compiled efficiently using ARM's load and store multiple instructions:

```
        BL      leaf2
        ..
leaf2   STMFD   sp!, {regs, lr}    ; save registers
        ..
        LDMFD   sp!, {regs, pc}    ; restore and return
```

Here the number of registers which are saved and restored will be the minimum required to implement the function. Note the value saved from the link register (Ir) is returned directly to the program counter (pc), and therefore Ir is available as a scratch register in the body of the function (which it was not in the simple case above).

More complex function entry sequences are used where needed to create stack backtrace data structures, handle floating-point arguments passed in floating-point registers, check for stack overflow, and so on.

Tail continued functions

Simple functions that call another function immediately before returning often do not incur any significant call overhead; the compiler will cause the code to return directly from the continuing function. This makes veneer functions (functions that simply reorder arguments, change their type or add an extra argument) particularly efficient.

ARM efficiency

Overall the ARM supports functions and procedures efficiently and flexibly. The various flavours of the procedure call standard match different application requirements well, and all result in efficient code. The load and store multiple register instructions are exploited to good effect by the compiler in this context; without them calls to non-leaf functions would be much more costly.

The compiler is also able to make effective optimizations for leaf and tail continued functions, thereby encouraging good programming styles. Programmers can use better structured design when leaf function calls are efficient and can exploit abstraction better when veneer functions are efficient.

6.9 Use of memory

An ARM system, like most computer systems, has its memory arranged as a linear set of logical addresses. A C program expects to have access to a fixed area of program memory (where the application image resides) and to memory to support two data areas that grow dynamically and where the compiler often cannot work out a maximum size. These dynamic data areas are:

- The stack.

  Whenever a (non-trivial) function is called, a new activation frame is created on the stack containing a backtrace record, local (non-static) variables, and so on. When a function returns its stack space is automatically recovered and will be reused for the next function call.

- The heap.

  The heap is an area of memory used to satisfy program requests (malloc ()) for more memory for new data structures. A program which continues to request memory over a long period of time should be careful to free up *all* sections that are no longer needed, otherwise the heap will grow until memory runs out.

Address space model

The normal use of memory is illustrated in Figure 6.13 on page 181. Where an application can use the entire memory space (or where a memory management unit can

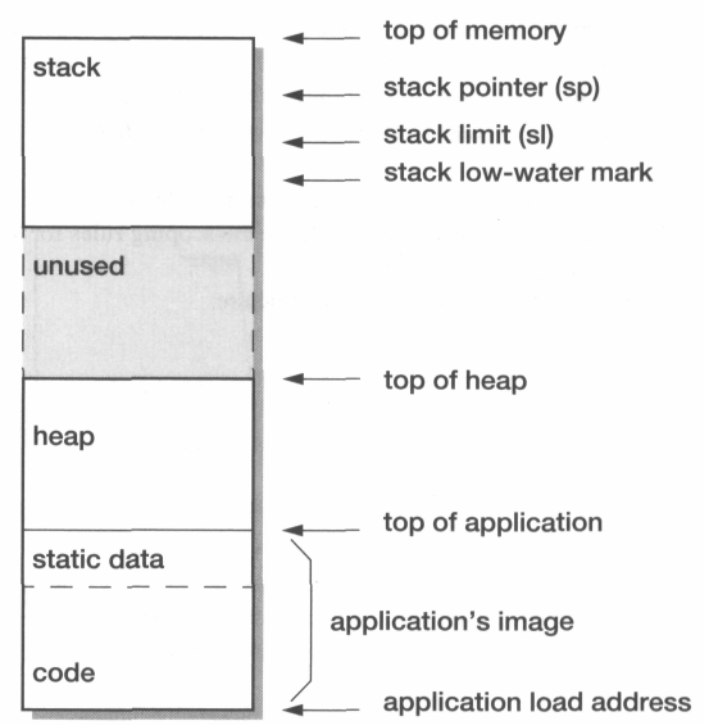


Figure 6.13 The standard ARM C program address space model.

allow an application to think it has the entire memory space), the application image is loaded into the lowest address, the heap grows upwards from the top of the application and the stack grows downwards from the top of memory. The unused memory between the top of the heap and the bottom of the stack is allocated on demand to the heap or the stack, and if it runs out the program stops due to lack of memory.

In a typical memory managed ARM system the logical space allocated to a single application will be very large, in the range of 1 to 4 Gbytes. The memory management unit will allocate additional pages, on demand, to the heap or the stack, until it runs out of pages to allocate (either due to having allocated all the physical memory pages, or, in a system with virtual memory, due to running out of swap space on the hard disk). This will usually be a long time before the top of the heap meets the bottom of the stack.

In a system with no memory management support the application will be allocated all (if it is the only application to run at the time) or part (if more than one application is to run) of the physical memory address space remaining once the operating system has had its requirements met, and then the application runs out of memory precisely when the top of the heap meets the bottom of the stack.

Chunked stack model

Other address space models are possible, including implementing a 'chunked' stack where the stack is a series of chained chunks within the heap. This causes the application to occupy a single contiguous area of memory, which grows in one direction as required, and may be more convenient where memory is very tight.

Stack behaviour

It is important to understand the dynamic behaviour of the stack while a program is running, since it sheds some light on the scoping rules for local variables (which are allocated space on the stack).

Consider this simple program structure:

```
main () {
  ..                    /* t1 */
  func1 ();
  ..                    /* t5 */
  func2 ();
  ..                    /* t7 */
} /* end of main */
func1 () {
  ..                    /* t2 */
  func2 ();
  ..                    /* t4 */
} /* end of func1 */

func2 () {
  ..                    /* t3, t6 */
} /* end of func2 */
```

Assuming that the compiler allocates stack space for each function call, the stack behaviour will be as shown in Figure 6.14. At each function call, stack space is allocated for arguments (if they cannot all be passed in registers), to save registers for use within the function, to save the return address and the old stack pointer, and to allocate memory on the stack for local variables.

Note how all the stack space is recovered on function exit and reused for subsequent calls, and how the two calls to func2 () are allocated different memory areas (at times t3 and t6 in Figure 6.14 on page 183), so even if the memory used for a local variable in the first call had not been overwritten in an intervening call to another function, the old value cannot be accessed in the second call since its address is not known. Once a procedure has exited, local variable values are lost forever.

Data storage

The various data types supported in C require differing amounts of memory to store their binary representations. The basic data types occupy a byte (chars), a half-word (short ints), a word (ints, single precision float) or multiple words (double precision floats). Derived data types (structs, arrays, unions, and so on) are defined in terms of multiple basic data types.

The ARM instruction set, in common with many other RISC processors, is most efficient at loading and storing data items when they are appropriately aligned in

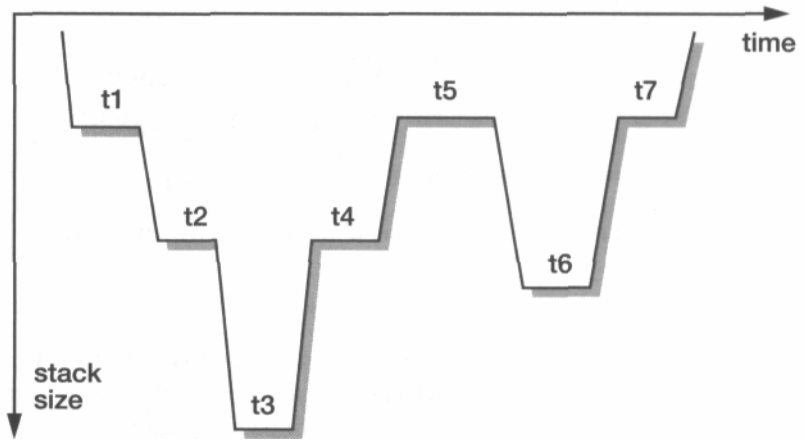


Figure 6.14 Example stack behaviour.

memory. A byte access can be made to any byte address with equal efficiency, but storing a word to a non-word-aligned address is very inefficient, taking up to seven ARM instructions and requiring temporary work registers.

Data alignment

Therefore the ARM C compiler generally aligns data items on appropriate boundaries:

- Bytes are stored at any byte address.
- Half-words are stored at even byte addresses.
- Words are stored on four-byte boundaries.

Where several data items of different types are declared at the same time, the compiler will introduce **padding** where necessary to achieve this alignment:

```
struct SI {char c; int x; short s;} example1;
```

This structure will occupy three words of memory as shown in Figure 6.15 on page 184. (Note that structures are also padded to end on a word boundary.)

Arrays are laid out in memory by repeating the appropriate basic data item, obeying the alignment rules for each item.

Memory efficiency

Given the data alignment rules outlined above, the programmer can help the compiler to minimize memory wastage by organizing structures appropriately. A structure with the same contents as above, but reordered as below, occupies only two memory words instead of the original three:

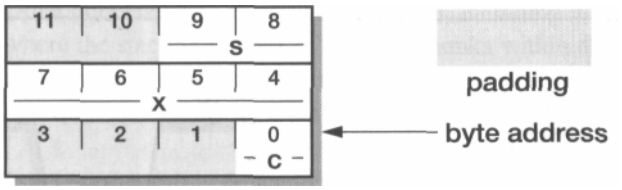


Figure 6.15 An example of normal struct memory allocation.

```
struct S2 {char c; short s; int x;} example2;
```

This will result in the memory occupancy illustrated in Figure 6.16. In general, ordering structure elements so that types smaller than a word can be grouped within a word will minimize the amount of padding that the compiler has to insert to maintain efficient alignment.

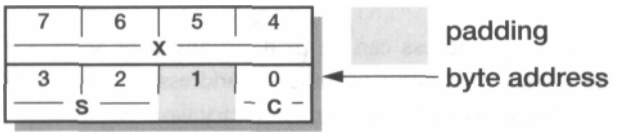


Figure 6.16 An example of more efficient struct memory allocation.

Packed structs

Sometimes it is necessary to exchange data with other computers that follow different alignment conventions, or to pack data tightly to minimize memory use even though this will reduce performance. For such purposes the ARM C compiler can produce code that works with packed data structures where all the padding is removed:

```
..packed struct S3 {char c; int x; short s;}
example3;
```

A packed struct gives precise control of the alignment of all the fields but incurs the overhead of the ARM's relatively inefficient access to non-aligned operands, and therefore should only be used when strictly necessary. The memory occupancy of the packed structure declared above is illustrated in Figure 6.17.

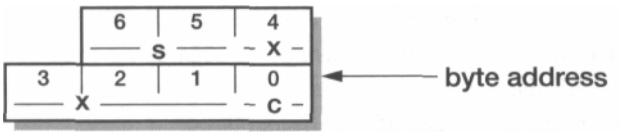


Figure 6.17 An example of packed struct memory allocation.

6.10 Run-time environment

A C program requires an environment in which to operate; this is usually provided through a library of functions that the C program can call. In a PC or workstation a C programmer can expect to find the full ANSI C library, giving access to a broad range of functions such as file management, input and output (print f ()), the real-time clock, and so on.

Minimal run-time library

In a small embedded system such as a mobile telephone, most of these functions are irrelevant. ARM Limited supplies a minimal stand-alone run-time library which, once ported to the target environment, allows basic C programs to run. This library therefore reflects the minimal requirements of a C program. It comprises:

- Division and remainder functions.

  Since the ARM instruction set does not include divide instructions, these are implemented as library functions.

- Stack-limit checking functions.

  A minimal embedded system is unlikely to have memory management hardware available for stack overflow detection; therefore these library functions are needed to ensure programs operate safely.

- Stack and heap management.

  All C programs use the stack for (many) function calls, and all but the most trivial create data structures on the heap.

- Program start up.

  Once the stack and heap are initialized, the program starts with a call to main ().

- Program termination.

  Most programs terminate by calling _exit (); even a program which runs forever should terminate if an error is detected.

The total size of the code generated for this minimal library is 736 bytes, and it is implemented in a way that allows the linker to omit any unreferenced sections to reduce the library image to around half a kilobyte in many cases. This is a great deal smaller than the full ANSI C library.

6.11 Examples and exercises

Example 6.1 **Write, compile and run a 'Hello World' program written in C.**

The following program has the required function:

```
/* Hello World in C */
#include <stdio.h>

int main() {
  printf( "Hello World\n" );
  return
(    0    ); }
```

The principal things to note from this example are:

- The '#include' directive which allows this program to use all the standard input and output functions available in C.
- The declaration of the 'main' procedure. Every C program must have exactly one of these as the program is run by calling it.
- The 'printf (. .}' statement calls a function provided in stdio which sends output to the standard output device. By default this is the display terminal.

As with the assembly programming exercises, the major challenge is to establish the flow through the tools from editing the text to compiling, linking and running the program. Once this program is working, generating more complex programs is fairly straightforward (at least until the complexity reaches the point where the design of the program becomes a challenge in itself).

Using the ARM software development tools, the above program should be saved as 'HelloW.c'. Then a new project should be created using the Project Manager and this file added (as the only file in the project). A click on the 'Build' button will cause the program to be compiled and linked, then the 'Go' button will run it on the ARMulator, hopefully giving the expected output in the terminal window.

Exercise 6.1.1 Generate an assembly listing from the compiler (using the '-s' option) and look at the code which is produced.

Exercise 6.1.2 Run the program under the debugger, using single-stepping to observe the progress of the processor through the code.

Example 6.2 **Write the number 2001 in 32-bit binary, binary-coded decimal, ASCII and single-precision floating-point notation.**

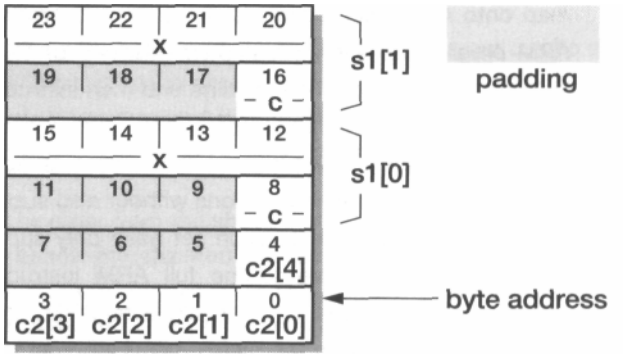
```
Binary: 2001    = 1024 + 512 + 256 + 128 + 64 + 16 + 1
                = 00000000000000000000011111010001₂
BCD:    2001    = 0010 0000 0000 0001
ASCII:  2001    = 00110010 00110000 00110000 00110001
F-P:    2001    = 1.1111010001 x 2^1010
                = 01001001 11110100 01000000 00000000
```

Exercise 6.2.1 Write a C program to convert a date presented in Roman numerals into decimal form.

Example 6.3 **Show how the following data is organized in memory:**

```
struct SI {char c; int x;}; struct
S2 {
  char c2[5];
  SI  si [2]; }
example;
```

The first structure statement only declares a type, so no memory is allocated. The second establishes a structure called 'example' comprising an array of five characters followed by an array of two structures of type SI. The structures must start on a word boundary, so the character array will be padded out to fill two words and each structure will also occupy two words. The memory organization is therefore as shown below:



Exercise 6.3.1 Show how the same structure would be organized in memory if it were packed.

7 The Thumb Instruction Set

Summary of chapter contents

The Thumb instruction set addresses the issue of code density. It may be viewed as a compressed form of a subset of the ARM instruction set. Thumb instructions map onto ARM instructions, and the Thumb programmer's model maps onto the ARM programmer's model. Implementations of Thumb use dynamic decompression in an ARM instruction pipeline and then instructions execute as standard ARM instructions within the processor.

Thumb is not a complete architecture; it is not anticipated that a processor would execute Thumb instructions without also supporting the ARM instruction set. Therefore the Thumb instruction set need only support common application functions, allowing recourse to the full ARM instruction set where necessary (for instance, all exceptions automatically enter ARM mode).

Thumb is fully supported by ARM development tools, and an application can mix ARM and Thumb subroutines flexibly to optimize performance or code density on a routine-by-routine basis.

This chapter covers the Thumb architecture and implementation, and suggests the characteristics of applications that are likely to benefit from using Thumb. In the right application, use of the Thumb instruction set can improve power-efficiency, save cost and enhance performance all at once.

7.1 The Thumb bit in the CPSR

ARM processors which support the Thumb instruction set can also execute the standard 32-bit ARM instruction set, and the interpretation of the instruction stream at any particular time is determined by bit 5 of the CPSR, the T bit (see Figure 2.2 on page 40). If T is set the processor interprets the instruction stream as 16-bit Thumb instructions, otherwise it interprets it as standard ARM instructions.

Not all ARM processors are capable of executing Thumb instructions; those that are have a T in their name, such as the ARM7TDMI described in Section 9.1 on page 248.

Thumb entry

ARM cores start up, after reset, executing ARM instructions. The normal way they switch to execute Thumb instructions is by executing a Branch and Exchange instruction (BX, see Section 5.5 on page 115). This instruction sets the T bit if the bottom bit of the specified register was set, and switches the program counter to the address given in the remainder of the register. Note that since the instruction causes a branch it flushes the instruction pipeline, removing any ambiguity over the interpretation of any instructions already in the pipeline (they are simply not executed).

Other instructions which change from ARM to Thumb code include exception returns, either using a special form of data processing instruction or a special form of load multiple register instruction (see 'Exception return' on page 109). Both of these instructions are generally used to return to whatever instruction stream was being executed before the exception was entered and are not intended for a deliberate switch to Thumb mode. Like BX, they also change the program counter and therefore flush the instruction pipeline.

Thumb exit

An explicit switch back to an ARM instruction stream can be caused by executing a Thumb BX instruction as described in Section 7.3 on page 191.

An implicit return to an ARM instruction stream takes place whenever an exception is taken, since exception entry is always handled in ARM code.

Thumb systems

It should be clear from the above that all Thumb systems include some ARM code, if only to handle initialization and exception entry.

It is likely, however, that most Thumb applications will make more than this minimal use of ARM code. A typical embedded system will include a small amount of fast 32-bit memory on the same chip as the ARM core and will execute speed-critical routines (such as digital signal processing algorithms) in ARM code from this memory. The bulk of the code will not be speed critical and may execute from a 16-bit off-chip ROM. This is discussed further at the end of the chapter.

7.2 The Thumb programmer's model

The Thumb instruction set is a subset of the ARM instruction set and the instructions operate on a restricted view of the ARM registers. The programmer's model is illustrated in Figure 7.1. The instruction set gives full access to the eight 'Lo' general purpose registers r0 to r7, and makes extensive use of r13 to r15 for special purposes:

- r13 is used as a stack pointer.
- r14 is used as the link register.
- r15 is the program counter (PC).

These uses follow very closely the way these registers are used by the ARM instruction set, though the use of r13 as a stack pointer in ARM code is purely a software convention, whereas in Thumb code it is somewhat hard-wired. The remaining registers (r8 to r!2 and the CPSR) have only restricted access:

- A few instructions allow the 'Hi' registers (r8 to r15) to be specified.
- The CPSR condition code flags are set by arithmetic and logical operations and control conditional branching.

Thumb-ARM similarities

All Thumb instructions are 16 bits long. They map onto ARM instructions so they inherit many properties of the ARM instruction set:

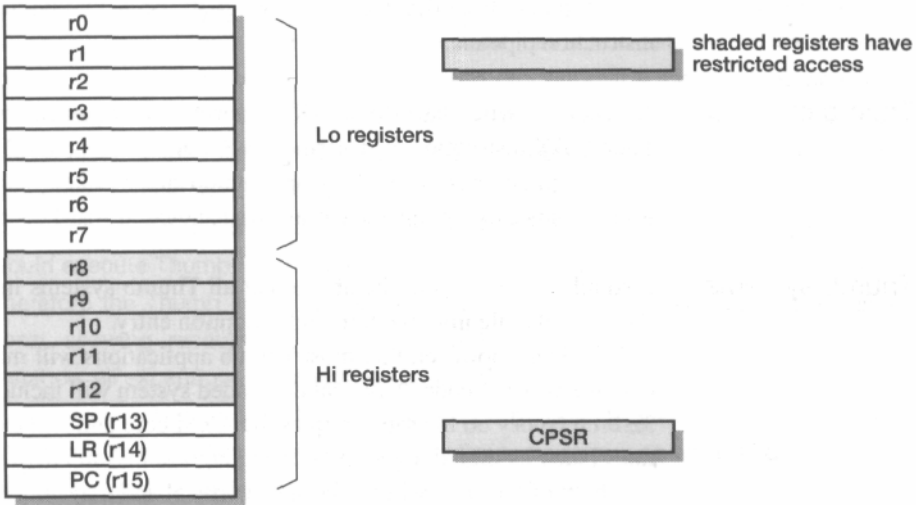


Figure 7.1 Thumb accessible registers.

- The load-store architecture with data processing, data transfer and control flow instructions.
- Support for 8-bit byte, 16-bit half-word and 32-bit word data types where half-words are aligned on 2-byte boundaries and words are aligned on 4-byte boundaries.
- A 32-bit unsegmented memory.

Thumb-ARM differences

However, in order to achieve a 16-bit instruction length a number of characteristic features of the ARM instruction set have been abandoned:

- Most Thumb instructions are executed unconditionally.

  (All ARM instructions are executed conditionally.)

- Many Thumb data processing instructions use a 2-address format (the destination register is the same as one of the source registers).

  (ARM data processing instructions, with the exception of the 64-bit multiplies, use a 3-address format.)

- Thumb instruction formats are less regular than ARM instruction formats, as a result of the dense encoding.

Thumb exceptions

All exceptions return the processor to ARM execution and are handled within the ARM programmer's model. Since the T bit resides in the CPSR, it is saved on exception entry in the appropriate SPSR, and the same return from exception instruction will restore the state of the processor and leave it executing ARM or Thumb instructions according to the state when the exception arose.

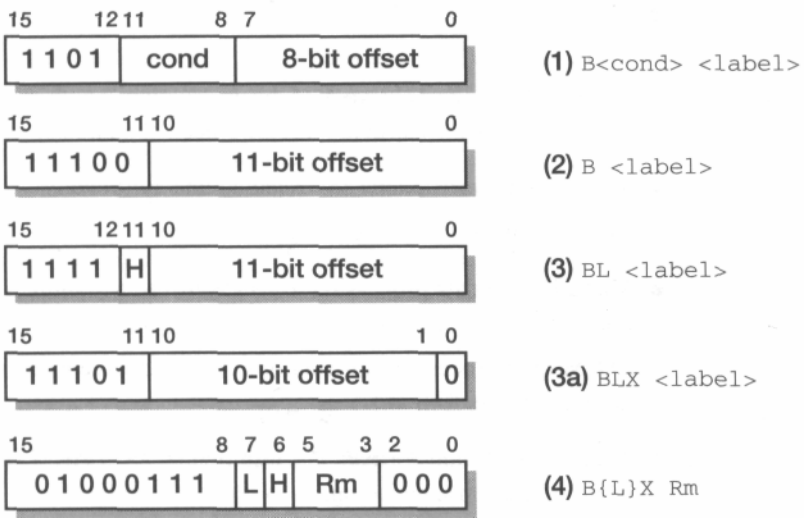
Note that the ARM exception return instructions, described in 'Exception return' on page 109, involve return address adjustments to compensate for the ARM pipeline behaviour. Since Thumb instructions are two bytes rather than four bytes long, the natural offset should be different when an exception is entered from Thumb execution, since the PC value copied into the exception-mode link register will have incremented by a multiple of two rather than four bytes. However, the Thumb architecture requires that the link register value be automatically adjusted to match the ARM return offset, allowing the same return instruction to work in both cases, rather than have the return sequence made more complex.

7.3 Thumb branch instructions

These control flow instructions include the various forms of PC-relative branch and branch-and-link instruction seen in the ARM instruction set, and the branch-and-exchange instruction for switching between the ARM and Thumb instruction sets.

The ARM instructions have a large (24-bit) offset field which clearly will not fit in a 16-bit instruction format. Therefore the Thumb instruction set includes various ways of subsetting the functionality.

Binary encodings



Description

Figure 7.2 Thumb branch instruction binary encodings.

Typical uses of branch instructions include:

1. short conditional branches to control (for example) loop exit;
2. medium-range unconditional branches to 'goto' sections of code;
3. long-range subroutine calls.

ARM handles all these with the same instruction, typically wasting many bits of the 24-bit offset in the first two cases. Thumb has to be more efficient, using different formats for each of these cases, numbered respectively in Figure 7.2.

The first two formats show how the condition field is traded off against the offset length. The condition field in the first format is the same as that in all ARM instructions (see Section5.3 on page 111); in both cases the offset is shifted left one bit (to give half-word alignment) and sign-extended to 32 bits.

The third format is more subtle. The branch and link subroutine mechanism often needs to have a long range, which is difficult within a 16-bit instruction format. Therefore Thumb uses two instructions, both with this format, to give a combined 22-bit half-word offset (which is sign-extended to 32 bits). The range of the instruction is therefore +/- 4 Mbytes. In order to make the two instructions independent, so that, for example, an interrupt can be taken between them, the link register is used as temper-

ary storage. It will be overwritten at the end of the instruction pair anyway, so it can't contain anything useful. The operation of the instruction pair is:

1. (H=0) LR := PC + (sign-extended offset shifted left 12 places);
2. (H=l) PC := LR + (offset shifted left 1 place);
   LR := oldPC + 3.

Here 'oldPC' is the address of the second instruction; the return address has two bytes added to point to the next instruction and the bottom bit set to indicate that the caller is a Thumb routine.

Format 3a is an alternative second step which gives the BLX variant; it is only available in architecture v5T. It uses the same first step as BL above:

1. (BL, H=0) LR := PC + (sign-extended offset shifted left 12 places);
2. (BLX) PC := LR + (offset shifted left 1 place) & Oxfffffffc;
   LR := oldPC + 3;
   the Thumb bit is cleared.

Note that as the target is an ARM instruction, the offset only requires 10 bits and the computed PC value may still have bit [1] set, so it is cleared implicitly.

The fourth format maps directly onto the ARM B{L}X instructions (see Section 5.5 on page 115, except that with BLX (available in architecture v5T only) r14 is set to the address of the following instruction plus 1, indicating that the caller was Thumb code). Here 'H' can be set to select a 'Hi' register (r8 to r15).

Assembler format

```
B<cond> <label> ;  format 1  - Thumb target
B       <label> ;  format 2  - Thumb target
BL      <label> ;  format 3  - Thumb target
BLX     <label> ;  format 3a - ARM target
B{L}X   Rm      ;  format 4  - ARM or Thumb target
```

A branch and link generates both format 3 instructions. It is not intended that format 3 instructions are used individually; they should always appear in pairs. Likewise, BLX generates a format 3 instruction and a format 3a instruction.

The assembler will compute the relevant offset to insert into the instruction from the current instruction address, the address of the target label, and a small correction for the pipeline behaviour. If the target is out of range an error message will be output.

Equivalent ARM instruction

Although formats 1 to 3 are very similar to the ARM branch and branch-with-link instructions, the ARM instructions support only word (4-byte) offsets whereas the Thumb instructions require half-word (2-byte) offsets. Therefore there is no direct mapping from these Thumb instructions into the ARM instruction set. The ARM cores that support Thumb are slightly modified to support half-word branch offsets, with ARM branch instructions being mapped to even half-word offsets.

Format 4 is equivalent to the ARM instruction with the same assembler syntax. The BLX variant is supported only by ARM processors that implement architecture v5T.

Subroutine call and return

The above instructions, and the equivalent ARM instructions, allow for subroutine calls to functions written in an instruction set the same as, or opposite to, the caller.

Functions that are called only from the same instruction set can use the conventional BL call and MOV pc, r14 or LDMFD sp!, {. . ,pc) (in Thumb code, POP {. . ., pc}) return sequences.

Functions that can be called from the opposite instruction set or from either instruction set can return with BX lr or LDMFD sp!, {. . . ,rN}; BX rN (in Thumb code, POP { . . . , rN} ; BX rN).

ARM processors that support architecture v5T can also return with LDMFD sp!, {. . ., pc} (in Thumb code, POP {. . ., pc}) as these instructions use the bottom bit of the loaded PC value to update the Thumb bit, but this is not supported in architectures earlier than v5T.

7.4 Thumb software interrupt instruction

The Thumb software interrupt instruction behaves exactly like the ARM equivalent and the exception entry sequence causes the processor to switch to ARM execution.

Binary encoding

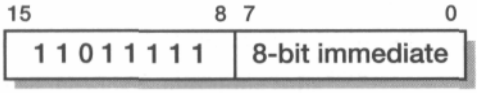


Figure 7.3 Thumb software interrupt binary encoding.

Description

This instruction causes the following actions:

- The address of the next Thumb instruction is saved in r14_svc.
- The CPSR is saved in SPSR_svc.
- The processor disables IRQ, clears the Thumb bit and enters supervisor mode by modifying the relevant bits in the CPSR.
- The PC is forced to address 0x08.

The ARM instruction SWI handler is then entered. The normal return instruction restores the Thumb execution state.

Assembler format

```
SWI <8-bit immediate>
```

Equivalent ARM instruction

The equivalent ARM instruction has an identical assembler syntax; the 8-bit immediate is zero-extended to fill the 24-bit field in the ARM instruction.

Clearly, this limits the SWIs available to Thumb code to the first 256 of the 16 million potential ARM SWIs.

7.5 Thumb data processing instructions

Thumb data processing instructions comprise a highly optimized set of fairly complex formats covering the operations most commonly required by a compiler.

The functions of these instructions are clear enough. The selection of those to include and those to leave out is far less obvious, but is based on a detailed understanding of the needs of typical application programs.

Binary encodings

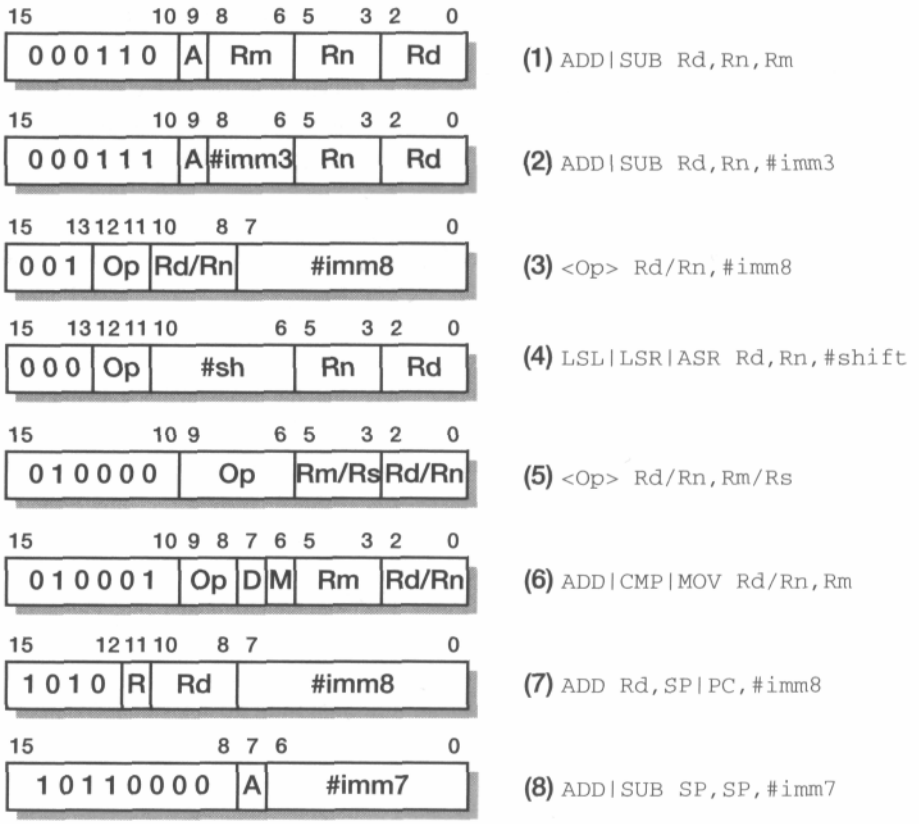


Figure 7.4 Thumb data processing instruction binary encodings.

Description

These instructions all map onto ARM data processing (including multiply) instructions. Although ARM supports a generalized shift on one operand together with an ALU operation in a single instruction, the Thumb instruction set separates shift and ALU operations into separate instructions, so here the shift operation is presented as an opcode rather than as an operand modifier.

Assembler format

The various instruction formats are:

```
1:      <op>    Rd, Rn, Rm          ; <op> = ADD|SUB
2:      <op>    Rd, Rn, #<#imm3>    ; <op> = ADD|SUB
3:      <op>    Rd|Rn, #<#imm8>     ; <op> = ADD|SUB|MOV|CMP
4:      <op>    Rd, Rn, #<#sh>      ; <op> = LSL|LSR|ASR
5:      <op>    Rd|Rn, Rm|Rs        ; <op> = MVN|CMP|CMN|..
    ; ..TST|ADC|SBC|NEG|MUL|LSL|LSR|ASR|ROR|AND|EOR|ORR|BIC
6:      <op>    Rd|Rn, Rm           ; <op> = ADD|CMP|MOV
    ;                                        (Hi regs)
7:      ADD     Rd, SP|PC, #<#imm8>
8:      <op>    SP, SP, #<#imm7>    ; <op> = ADD|SUB
```

Equivalent ARM instructions

The ARM data processing instructions that have equivalents in the Thumb instruction set are listed below, with their Thumb equivalents in the comment field. Instructions that use the 'Lo', general-purpose registers (r0 to r7):

```
ARM instruction                 Thumb  instruction

MOVS    Rd,  #<#imm8>           ;MOV   Rd,  #<#imm8>
MVNS    Rd,  Rm                 ;MVN   Rd,  Rm
CMP     Rn,  #<#imm8>           ;CMP   Rn,  #<#imm8>
CMP     Rn,  Rm                 ;CMP   Rn,  Rm
CMN     Rn,  Rm                 ;CMN   Rn,  Rm
TST     Rn,  Rm                 ;TST   Rn,  Rm
ADDS    Rd,  Rn, #<#imm3>       ;ADD   Rd,  Rn, #<#imm3>
ADDS    Rd,  Rd, #<#imm8>       ;ADD   Rd,  #<#imm8>
ADDS    Rd,  Rn, Rm             ;ADD   Rd,  Rn, Rm
ADCS    Rd,  Rd, Rm             ;ADC   Rd,  Rm
SUBS    Rd,  Rn, #<#imm3>       ;SUB   Rd,  Rn, #<#imm3>
SUBS    Rd,  Rd, #<#imm8>       ;SUB   Rd,  #<#imm8>
SUBS    Rd,  Rn, Rm             ;SUB   Rd,  Rn, Rm
SBCS    Rd,  Rd, Rm             ;SBC   Rd,  Rm
RSBS    Rd,  Rn, #0             ;NEC   Rd,  Rn
MOVS    Rd,  Rm, LSL #<#sh>     ;LSL   Rd,  Rm, #<#sh>
```

```
;  ARM instruction              Thumb instruction

   MOVS    Rd, Rd, LSL Rs       ;LSL  Rd, Rs
   MOVS    Rd, Rm, LSR #<#sh>   ;LSR  Rd, Rm, #<#sh>
   MOVS    Rd, Rd, LSR Rs       ;LSR  Rd, Rs
   MOVS    Rd, Rm, ASR #<#sh>   ;ASR  Rd, Rm, #<#sh>
   MOVS    Rd, Rd, ASR Rs       ;ASR  Rd, Rs
   MOVS    Rd, Rd, ROR Rs       ;ROR  Rd, Rs
   ANDS    Rd, Rd, Rm           ;AND  Rd, Rm
   EORS    Rd, Rd, Rm           ;EOR  Rd, Rm
   ORRS    Rd, Rd, Rm           ;ORR  Rd, Rm
   BICS    Rd, Rd, Rm           ;BIG  Rd, Rm
   MULS    Rd, Rm, Rd           ;MUL  Rd, Rm
```

Instructions that operate with or on the 'Hi' registers (r8 to r15), in some cases in combination with a 'Lo' register:

```
;  ARM instruction              Thumb instruction

    ADD    Rd, Rd, Rm            ;ADD  Rd, Rm (1/2 Hi regs)
    CMP    Rn, Rm                ;CMP  Rn, Rm (1/2 Hi regs)
    MOV    Rd, Rm                ;MOV  Rd, Rm (1/2 Hi regs)
   ADD    Rd, PC, #<#imm8>      ;ADD  Rd, PC, #<#imm8>
   ADD    Rd, SP, #<#imm8>      ;ADD  Rd, SP, #<#imm8>
   ADD    SP, SP, #<#imm7>      ;ADD  SP, SP, #<#imm7>
   SUB    SP, SP, #<#imm7>      ;SUB  SP, SP, #<#imm7>
```

Notes

1. *All* the data processing instructions that operate with and on the 'Lo' registers update the condition code bits (the S bit is set in the equivalent ARM instruc tion).
2. The instructions that operate with and on the 'Hi' registers do *not* change the condition code bits, with the exception of CMP which only changes the condi tion codes.
3. The instructions that are indicated above as requiring '1 or 2 Hi regs' must have one or both register operands specified in the 'Hi' register area.
4. #imm3, #imm7 and #imm8 denote 3-, 7- and 8-bit immediate fields respectively. #sh denotes a 5-bit shift amount field.

7.6 Thumb single register data transfer instructions

Again the choice of ARM instructions which are represented in the Thumb instruction set appears complex, but is based on the sort of things that compilers like to do frequently.

Note the larger offsets for accesses to the literal pool (PC-relative) and to the stack (SP-relative), and the restricted support given to signed operands (base plus register addressing only) compared with unsigned operands (base plus offset or register).

Binary encodings

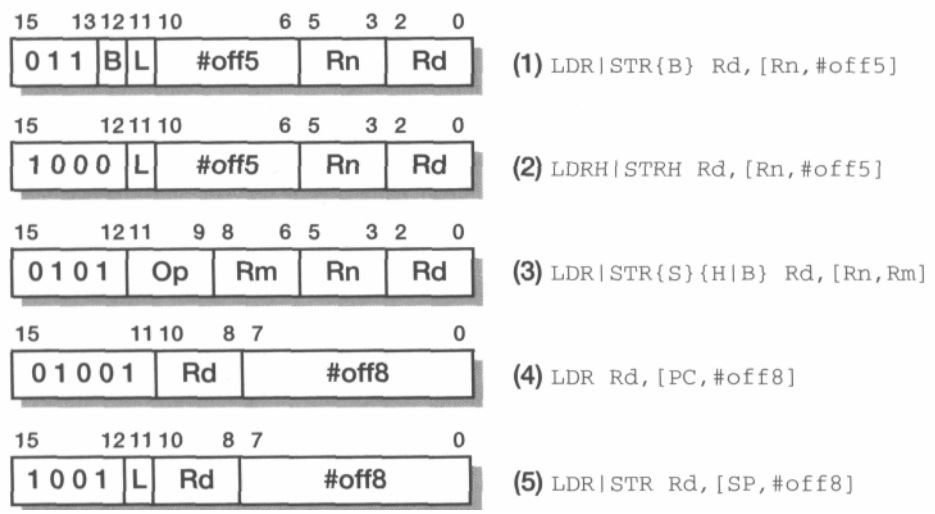


Figure 7.5 Thumb single register data transfer binary encodings.

Description

These instructions are a carefully derived subset of the ARM single register transfer instructions, and have exactly the same semantics as the ARM equivalent.

In all cases the offset is scaled to the size of the data type, so, for instance, the range of the 5-bit offset is 32 bytes in a load or store byte instruction, 64 bytes in a load or store half-word instruction and 128 bytes in a load or store word instruction.

Assembler format

The various assembler formats are:

```
1:      <op>    Rd, [Rn, #<#off5>] ; <Op> = LDRILDRB|STRISTRB
2:      <op>    Rd, [Rn, #<#off5>] ; <op> = LDRHISTRH
3:      <op>    Rd, [Rn, Rm]       ; <op> = ..
                ; .. LDRILDRHILDRSHILDRBILDRSBISTRISTRHISTRB
A:      LDR     Rd, [PC, #<#off8>]
5:      <op>    Rd, [SP, #<#off8>] ; <op> = LDRISTR
```

Equivalent ARM instruction

The ARM equivalents to these Thumb instructions have identical assembler formats.

Notes

1. #of f 5 and toff 8 denote 5- and 8-bit immediate offsets respectively. The assem bler format specifies the offset in bytes in all cases. The 5- or 8-bit offset in the instruction binary is scaled by the size of the data type.
2. As with the ARM instructions, the signed variants are only supported by the load instructions since store signed and store unsigned have exactly the same effect.

7.7 Thumb multiple register data transfer instructions

As in the ARM instruction set, the Thumb multiple register transfer instructions are useful both for procedure entry and return and for memory block copy. Here, however, the tighter encoding means that the two uses must be separated and the number of addressing modes restricted. Otherwise these instructions very much follow the spirit of their ARM equivalents.

Binary encodings

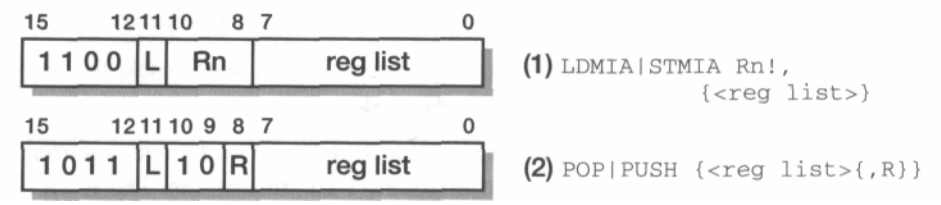


Figure 7.6 Thumb multiple register data transfer binary encodings.

Description

The block copy forms of the instruction use the LDMIA and STMIA addressing modes (see Figure 3.2 on page 62). The base register may be any of the 'Lo' registers (r0 to r7), and the register list may include any subset of these registers but should not include the base register itself since write-back is always selected (and the result of a load or store multiple register with the base register in the list and write-back selected is unpredictable).

The stack forms use SP (r13) as the base register and again always use write-back. The stack model is fixed as full-descending. In addition to the eight registers which may be specified in the register list, the link register (LR, or r14) may be included in the 'PUSH' instruction and the PC (r15) may be included in the 'POP' form, optimizing procedure entry and exit sequences as is often done in ARM code.

Assembler format

<reg list> is a list of registers and register ranges from r0 to r7.

```
LDMIA   Rn!, {<reg list>)
STMIA   Rn!, {<reg list>}
POP     {<reg list>{, pc}}
PUSH    {<reg list>{, lr}}
```

Equivalent ARM instruction

The equivalent ARM instructions have the same assembler format in the first two cases, and replace POP and PUSH with the appropriate addressing mode in the second two cases. Block copy:

```
LDMIA   Rn!,  {<reg list>}
STMIA   Rn!,  {<reg list>}
```

Pop:

```
LDMFD   SP!,  {<reg list>{, pc}}
```

Push:

```
STMFD   SP!,  {<reg list>{, lr}}
```

Notes

1. The base register should be word-aligned. If it is not, some systems will ignore the bottom two address bits but others may generate an alignment exception.
2. Since all these instructions use base write-back, the base register should not be included in the register list.
3. The register list is encoded with one bit for each register; bit 0 indicates whether r0 will be transferred, bit 1 controls r1, etc. The R bit controls the PC and LR options in the POP and PUSH instructions.
4. In architecture v5T only, the bottom bit of a loaded PC updates the Thumb bit, enabling a direct return to a Thumb or ARM caller.

7.8 Thumb breakpoint instruction

The Thumb breakpoint instruction behaves exactly like the ARM equivalent. Breakpoint instructions are used for software debugging purposes; they cause the processor to break from normal instruction execution and enter appropriate debugging procedures.

Binary encoding

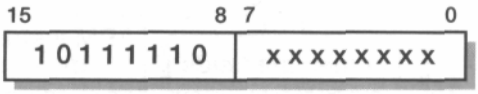


Figure 7.7 Thumb breakpoint binary encoding.

Description

This instruction causes the processor to take a prefetch abort when the debug hardware unit is configured appropriately.

Assembler format

BKPT

Equivalent ARM instruction

The equivalent ARM instruction has an identical assembler syntax. The BRK instruction is supported only by ARM processors that implement ARM architecture v5T.

7.9 Thumb implementation

The Thumb instruction set can be incorporated into a 3-stage pipeline ARM processor macrocell with relatively minor changes to most of the processor logic (the 5-stage pipeline implementations are trickier). The biggest addition is the Thumb instruction decompressor in the instruction pipeline; this logic translates a Thumb instruction into its equivalent ARM instruction. The organization of this logic is shown in Figure 7.8 on page 202.

The addition of the decompressor logic in series with the instruction decoder might be expected to increase the decode latency, but in fact the ARM? pipeline does relatively little work in phase 1 of the decode cycle. Therefore the decompression logic can be accommodated here without compromising the cycle time or increasing the pipeline latency, and the ARM7TDMI Thumb pipeline operates in exactly the way described in 'The 3-stage pipeline' on page 75.

Instruction mapping

The Thumb decompressor performs a static translation from the 16-bit Thumb instruction into the equivalent 32-bit ARM instruction. This involves performing a look-up to translate the major and minor opcodes, zero-extending the 3-bit register specifiers to give 4-bit specifiers and mapping other fields across as required.

As an example, the mapping of a Thumb 'ADD Rd, #imm8' instruction (see Section 7.5 on page 195) to the corresponding ARM 'ADDS Rd,Rd, #imm8' instruction (described in Section 5.7 on page 119) is shown in Figure 7.9 on page 202. Note that:

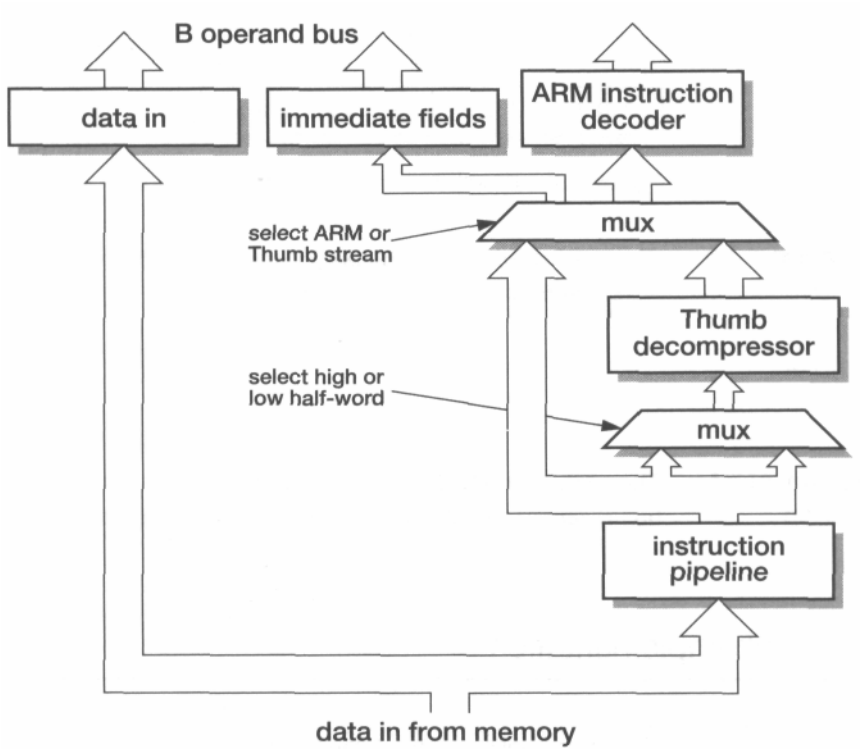


Figure 7.8 The Thumb instruction decompressor organization.

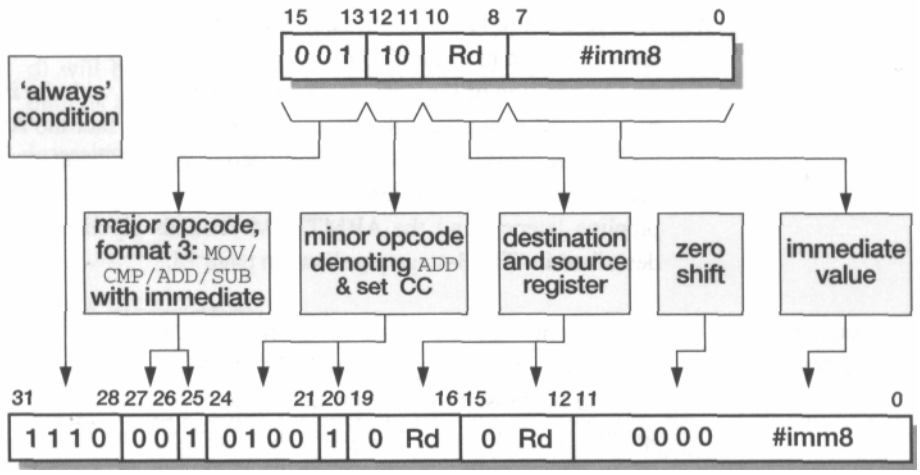


Figure 7.9 Thumb to ARM instruction mapping.

- Since the only conditional Thumb instructions are branches, the condition 'always' is used in translating all other Thumb instructions.
- Whether or not a Thumb data processing instruction should modify the condition codes in the CPSR is implicit in the Thumb opcode; this must be made explicit in the ARM instruction.
- The Thumb 2-address format can always be mapped into the ARM 3-address format by replicating a register specifier. (Going the other way is not, in general, possible.)

The simplicity of the decompression logic is crucial to the efficiency of the Thumb instruction set. There would be little merit in the Thumb architecture if it resulted in complex, slow and power-hungry decompression logic.

7.10 Thumb applications

To see where Thumb offers a benefit we need to review its properties. Thumb instructions are 16 bits long and encode the functionality of an ARM instruction in half the number of bits, but since a Thumb instruction typically has less semantic content than an ARM instruction, a particular program will require more Thumb instructions than it would have needed ARM instructions. The ratio will vary from program to program, but in a typical example Thumb code may require 70% of the space of ARM code. Therefore if we compare the Thumb solution with the pure ARM code solution, the following characteristics emerge:

Thumb properties

- The Thumb code requires 70% of the space of the ARM code.
- The Thumb code uses 40% more instructions than the ARM code.
- With 32-bit memory, the ARM code is 40% faster than the Thumb code.
- With 16-bit memory, the Thumb code is 45% faster than the ARM code.
- Thumb code uses 30% less external memory power than ARM code.

So where performance is all-important, a system should use 32-bit memory and run ARM code. Where cost and power consumption are more important, a 16-bit memory system and Thumb code may be a better choice. However, there are intermediate positions which may give the best of both worlds:

Thumb systems

- A high-end 32-bit ARM system may use Thumb code for certain non-critical routines to save power or memory requirements.

• A low-end 16-bit system may have a small amount of on-chip 32-bit RAM for critical routines running ARM code, but use off-chip Thumb code for all non-critical routines.

The second of these examples is perhaps closer to the sort of application for which Thumb was developed. Mobile telephone and pager applications incorporate real-time digital signal processing (DSP) functions that may require the full power of the ARM, but these are tightly coded routines that can fit in a small amount of on-chip memory. The more complex and much larger code that controls the user interface, battery management system, and so on, is less time-critical, and the use of Thumb code will enable off-chip ROMs to give good performance on an 8- or 16-bit bus, saving cost and improving battery life.

7.11 Example and exercises

Example 7.1 **Rewrite the 'Hello World' program in Section 3.4 on page 69 to use Thumb instructions. How do the ARM and Thumb code sizes compare?**

Here is the original ARM program:

```
        AREA    HelloW,CODE,READONLY
SWI_WriteC      EQU     &0          ; output character in r0
SWI_Exit        EQU     &11         ; finish program
        ENTRY                       ; code entry point
START   ADR     r1, TEXT            ; r1 -> "Hello World"
LOOP    LDRB    r0, [r1], #1        ; get the next byte
        CMP     r0, #0              ; check for text end
        SWINE   SWI_WriteC          ; if not end print ..
        BNE     LOOP                ;  .. and loop back
        SWI     SWI_Exit            ; end of execution
TEXT    =       "Hello World",&0a,&0d,0
        END                         ; end of program source
```

Most of these instructions have direct Thumb equivalents; however, some do not. The load byte instruction does not support auto-indexing and the supervisor call cannot be conditionally executed. Hence the Thumb code needs to be slightly modified:

```
        AREA    HelloW_Thumb,CODE,READONLY
SWI_WriteC      EQU    &0       ; output character in r0
SWI_Exit        EQU    &11       ; finish program
        ENTRY                      ; code entry point ;
        CODES2                     enter in ARM state
```

```
        ADR     r0, START+1          ;get Thumb entry address
        BX      r0                   ;enter Thumb area
        CODE16                       ;Thumb code follows..
START   ADR     r1, TEXT             ;r1 -> "Hello World"
LOOP    LDRB    r0, [r1]             ;get the next byte
        ADD     r1, r1, #1           ;increment pointer  **T
        CMP     r0, #0               ;check for text end
        BEQ     DONE                 ;finished?           **T
        SWI     SWI_WriteC           ;if not end print . .
        B       LOOP                  ;.. and loop back
DONE    SWI     SWI_Exit             ;end of execution
        ALIGN                        ;to ensure ADR works
TEXT    DATA
                "Hello World",&0a,&0d,&00
        END
```

The two additional instructions required to compensate for the features absent from the Thumb instruction set are marked with '**T' in the above listing. The ARM code size is six instructions plus 14 bytes of data, 38 bytes in all. The Thumb code size is eight instructions plus 14 bytes of data (ignoring the preamble required to switch the processor to executing Thumb instructions), making 30 bytes in all.

This example illustrates a number of important points to bear in mind when writing Thumb code:

- The assembler needs to know when to produce ARM code and when to produce Thumb code. The 'CODES 2' and 'CODE16' directives provide this information. (These are instructions to the assembler and do not themselves cause any code to be generated.)
- Since the processor is executing ARM instructions when it calls the code, explicit provision must be made to instruct it to execute the Thumb instructions. The 'BX rO' instruction achieves this, provided that r0 has been initialized appropri ately. Note particularly that the bottom bit of r0 is set to cause the processor to execute Thumb instructions at the branch target.
- In Thumb code 'ADR' can only generate word-aligned addresses. As Thumb instructions are half-words, there is no guarantee that a location following an arbitrary number of Thumb instructions will be word-aligned. Therefore the example program has an explicit 'ALIGN' before the text string.

In order to assemble and run this program on the ARM software development toolkit, an assembler that can generate Thumb code must be invoked and the ARMula-tor must emulate a 'Thumb-aware' processor core. The default setting of the Project Manager targets an ARM6 core and generates only 32-bit ARM code. This may be changed by choosing 'Project' from the 'Options' menu within the Project Manager

and selecting 'TCC/TASM' in the 'Tools' dialogue box before generating the code. The 'Target Processor' will automatically switch to the Thumb-aware ARM7t when this is done.

Otherwise, assembling and running Thumb code is just like using ARM code.

Exercise 7.1.1 Convert the other programs in Sections 3.4 and 3.5 on pages 69 and 72 into Thumb code and compare their sizes with the original ARM code.

Exercise 7.1.2 Use TCC to generate Thumb code from C source programs (starting, as usual, with a 'Hello World' program). Look at the assembly code generated by the C compiler (using the '-s' option). Compare the code size and the execution time with the same programs compiled into ARM code.

8 Architectural Support for System Development

Summary of chapter contents

Designing any computer system is a complex task; designing an embedded 'system-on-chip' is a daunting one. The development often takes place entirely within a CAD environment, and the first silicon must not only function but it must deliver the necessary performance and be manufacturable. The only scope for fixing design flaws is in the software; to modify the chip in order to correct errors takes too long and usually incurs considerable cost.

For the past two decades the principal approach to microprocessor system development has been based upon the *In-Circuit Emulator* (ICE). The system itself was a printed circuit board incorporating a microprocessor chip and various memory and peripheral devices. To use the ICE, the microprocessor was removed from its socket and replaced by a header plug with an umbilical connection to the ICE equipment. The ICE emulated the function of the microprocessor and gave the user an inside view on the internal state of the system, giving access to read and modify processor registers and memory values, to set breakpoints, and so on.

Now that the microprocessor itself has become just a cell on a larger chip, this whole approach has collapsed. It is not possible to unplug part of a chip! There is, as yet, no approach that has replaced the ICE in all its roles, but there are several techniques that contribute, some of which require explicit support in the processor's architecture. This chapter covers the techniques available to support the development of system chips based on ARM cores and the architectural features built into the cores to assist in this process.

8.1 The ARM memory interface

In this section we look at the general principles involved in connecting an ARM processor to a memory system built from standard memory parts. The efficiency of the memory interface is an important determinant of the system performance, so these principles must be well understood by the designer who wishes to develop a high-performance system.

More recent ARM cores have direct AMBA interfaces (see Section 8.2 on page 216), but many of the basic issues discussed here still apply.

ARM bus signals

ARM processor chips vary in the details of their bus interfaces, but they are generally similar in nature. The memory bus interface signals include the following:

- A 32-bit address bus, *A[31:0],* which gives the byte address of the data to be accessed.
- A 32-bit bidirectional data bus, *D[3J:OJ,* along which the data is transferred.
- Signals that specify whether the memory is needed *(mreq)* and whether the address is sequential *(seq);* these are issued in the previous cycle so that the memory control logic can prepare appropriately.
- Signals that specify the direction (r/w) and size *(b/w* on earlier processors; *mas[1:0]* on later processors) of the transfer.
- Bus timing and control signals *(abe, ale, ape, dbe, lock, bl[3:0]).*

Simple memory interface

The simplest form of memory interface is suitable for operation with ROM and static RAM (SRAM). These devices require the address to be stable until the end of the cycle, which may be achieved by disabling the address pipeline (tying *ape* low) on later processors or retiming the address bus (connecting *ale* to *mclk)* on earlier processors. The address and data buses may then be connected directly to the memory parts as shown in Figure 8.1 on page 209 which also shows the output enable signals *(RAMoe* and *ROMoe)* and the write enables *(RAMwe).*

This figure illustrates the connection of 8-bit memory parts, which are a standard configuration for SRAMs and ROMs. Four parts of each type are required to form a 32-bit memory and an individual device is connected to a single byte section of the bus. The notation on the figure shows the device's bus numbering inside the device and the bus wires to which it is connected outside the device, so, for example, the SRAM shown nearest the ARM has its pins *D[7:0]* connected to bus pins *D[31:24]* which are connected to the ARM's pins *D[31:24].*

Since the bottom two address lines, *A [1:0],* are used for byte selection, they are used by the control logic and not connected to the memory. Therefore the address lines on the memory devices are connected to *A [2]* and upwards, the precise number used depending on the size of the memory part (for example, a 128 Kbyte ROM part will *useA[18:2J).*

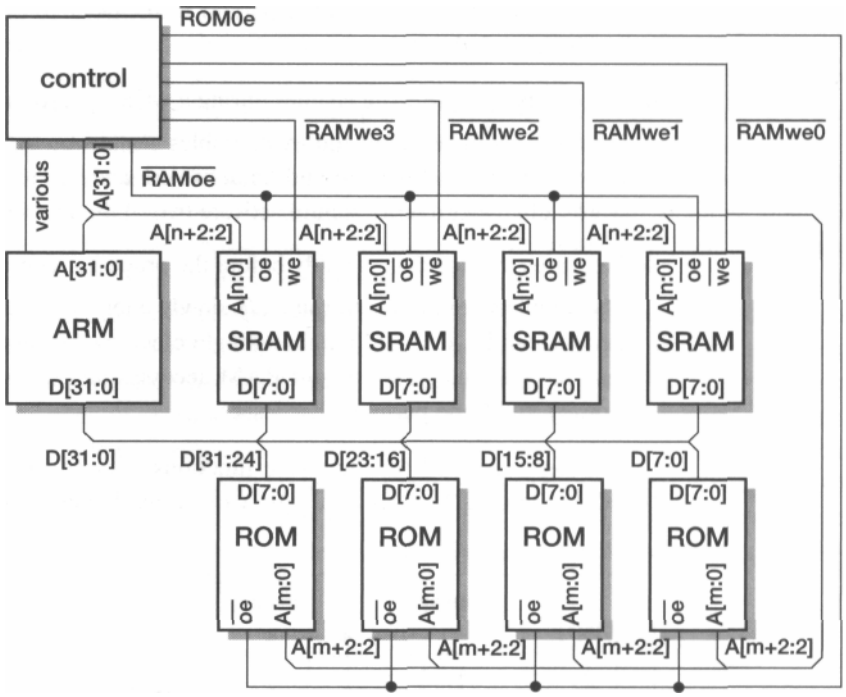


Figure 8.1 A basic ARM memory system.

Although the ARM performs reads of both bytes and words, the memory system can ignore the difference (at the cost of some power wastage) and always simply supply a word quantity. The ARM will extract the addressed byte and ignore the remainder of the word. Therefore the ROMs do not need individual enables and using 16-bit devices causes no problems. Byte *writes,* however, do require individual byte enables, so the control logic must generate four byte write enable controls. This makes the use of wider RAMs difficult (and inefficient) unless they incorporate separate byte enables, since writing an individual byte would require a read-modify-write memory operation. Since many processors require support for writing bytes, it is likely that if RAMs do become available with a data width greater than a byte, they will incorporate individual byte enables.

Control logic

The control logic performs the following functions:

- It decides when to activate the RAM and when to activate the ROM.

  This logic determines the system memory map. The processor starts from location zero after a reset, so it must find ROM there since the RAM is uninitialized. The simplest memory map therefore enables the ROM *ifAf31J* is low and the RAM if it is high. (Most ARM systems change the memory map shortly after

start-up to put the RAM at the bottom of memory so that the exception vectors can be modified.)

- It controls the byte write enables during a write operation.

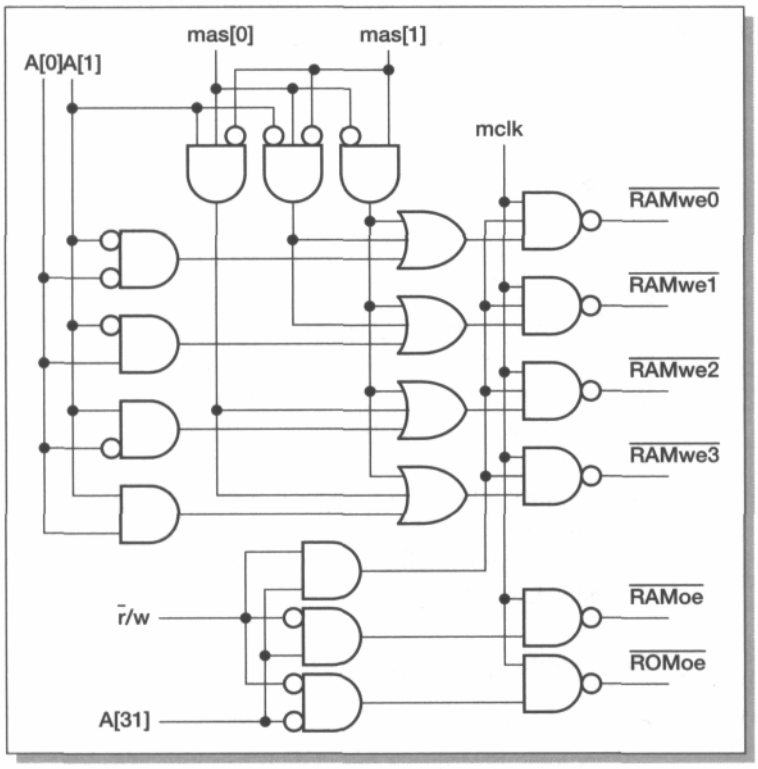
  During a word write all the byte enables should be active, during a byte write only the addressed byte should be activated, and where the ARM supports half-words a half-word write should activate two of the four enables.

- It ensures that the data is ready before the processor continues.

  The simplest solution is to run *mclk* slowly enough to ensure that all the memory devices can be accessed within a single clock cycle. More sophisticated systems may have the clock set to suit RAM accesses and use wait states for (typically slower) ROM and peripheral accesses.

The logic required for the above functions is quite straightforward and is illustrated in Figure 8.2. (All this logic can be implemented using a single program-

Figure 8.2 Simple ARM memory system control logic.


mas[0]
mas[1]
A[0]A[1]
mclk
RAMwe0
RAMwe1
RAMwe2
RAMwe3
r/w
RAMoe
ROMoe
A[31]

mable logic device.) Perhaps the trickiest aspect of the design relates to the bidirectional data bus. Here it is very important to ensure that only one device drives the bus at any time, so care is needed when turning the bus around for a write cycle, or when switching between reading from the ROM and reading from the RAM. The solution illustrated in the figure activates the appropriate data source when *mclk* is high and turns all sources off when *mclk* is low, so *dbe,* the processor's data bus enable, should also be connected to *mclk.* This is a very conservative solution which will often compromise the performance of the system by limiting the maximum clock frequency that can be used.

Note that this design assumes that the ARM outputs are stable to the end of the clock cycle, which will be the case on newer processors with the address pipeline enable *(ape)* control input tied low. Older processors should use *ale = mclk* to retime the address outputs, but will need an external transparent latch which is open when *mclk* is low to retime *r/w* and *~b/w* (which replaces *mas[l],* and *mas[0]* is tied low).

This simple memory system makes no use of *mreq* (or *seq)* it simply activates the memory on every cycle. This is safe since the ARM will only request a write cycle on a genuine memory access. The *r/w* control remains low during all internal and coprocessor register transfer cycles.

Wait states

If we try to speed up the clock in this system it will stop working when the slowest path fails. This will normally be the ROM access. We can get a lot more performance from the system if the clock is tuned to the RAM access time and wait states are introduced to allow the ROM more access time. Usually the ROM will be allowed a fixed number of clock cycles per access, the exact number being determined by the clock rate and the ROM data sheet. We will assume an access time of four clock cycles.

The memory control logic must now incorporate a simple finite state machine to control the ROM access. A suitable state transition diagram is shown in Figure 8.3.

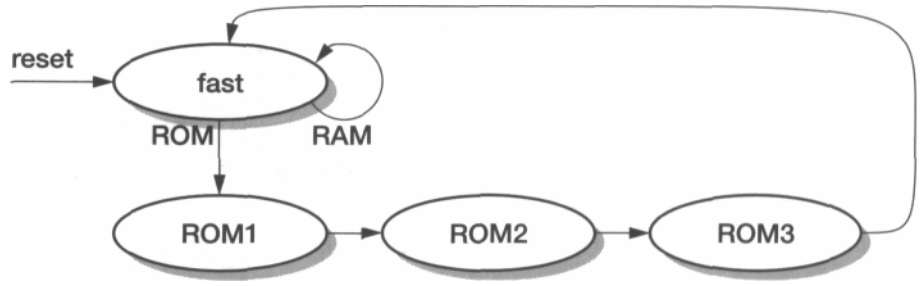


Figure 8.3 ROM wait control state transition diagram.

The three ROM states are used to stretch the ROM access time to four cycles by asserting the ARM's *wait* input. A design problem here is that since the addresses have been retimed to become valid early in the current cycle and *wait* must be asserted before *mclk* rises, *wait* cannot be generated as a simple state machine output since there is no clock edge that can be used to generate it. Another problem is to generate a stretched *ROMoe* signal that is glitch-free.

A possible circuit is shown in Figure 8.4. The state machine is a synchronous counter which uses the two edge-triggered flip-flops. Only the state ROM3 is of significance, since it de-activates *wait* which is otherwise active whenever a ROM access is detected. The two level-sensitive latches are used to generate a clean, stretched *ROMoe* using *wait* as the starting point. A timing diagram for this circuit is shown in Figure 8.5 on page 213, which should clarify the operation of the logic.

Sequential accesses

If the system is to operate even faster it may not be possible to decode a new address and perform a RAM access in a single clock cycle. Here an extra cycle can be inserted whenever an unknown address is issued to allow time for address decoding. The only addresses that are not unknown are sequential ones, but these represent around 75% of all addresses in a typical program.

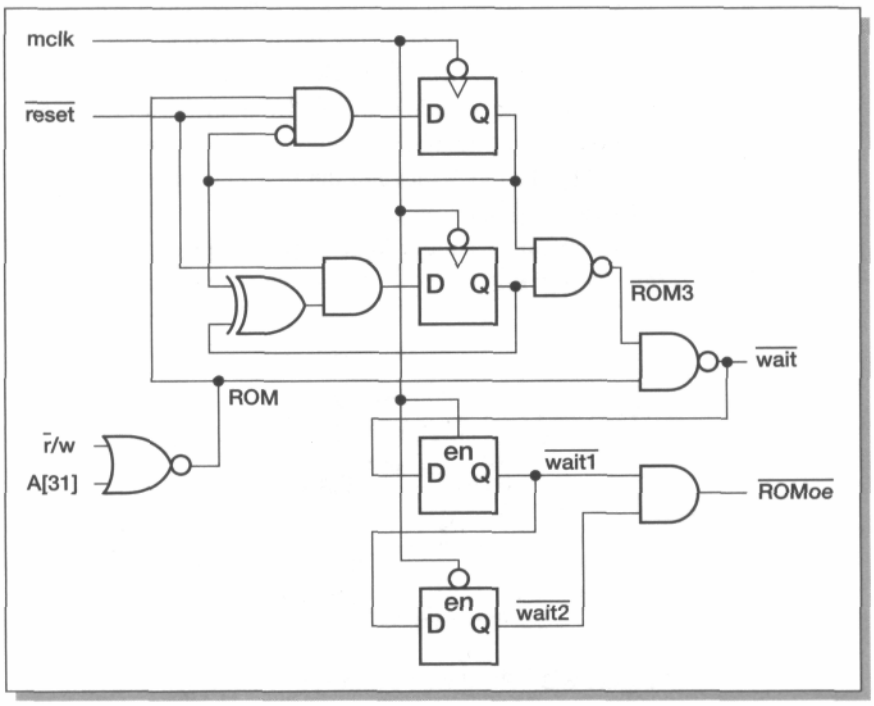


Figure 8.4 ROM wait state generator circuit.

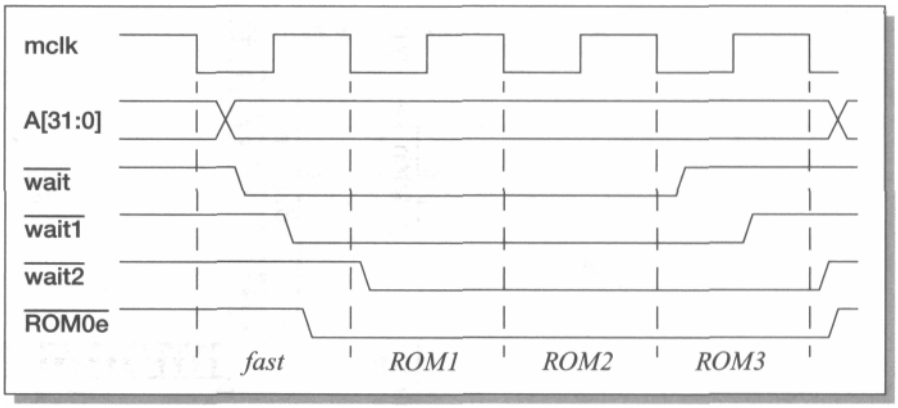


Figure 8.5 The timing diagram for the ROM wait state logic.

We should also now begin to recognize cycles that do not use the memory, since there is no reason why they should not operate at the full clock rate. A suitable state transition diagram is shown in Figure 8.6.

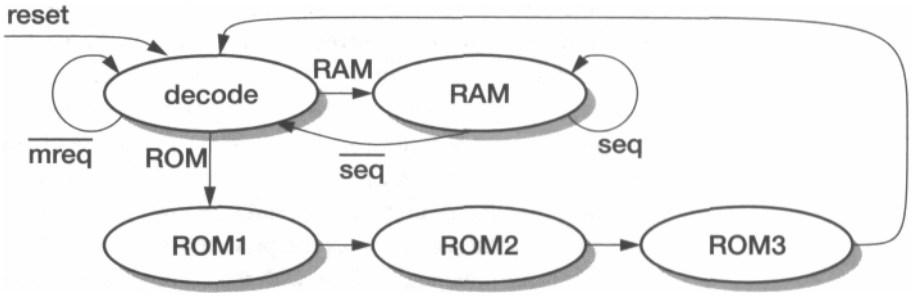


Figure 8.6 State transition diagram with a wait state for address decoding.

DRAM

The cheapest memory technology (in terms of price per bit) is dynamic random access memory (DRAM). 'Dynamic' memory stores information as electrical charge on a capacitor where it gradually leaks away (over a millisecond or so). The memory data must be read and rewritten ('refreshed') before it leaks away. The responsibility for refreshing the memory usually lies with the memory control logic, not the processor, so it is not of immediate concern to us here. What is of concern is the internal organization of the memory which is shown in Figure 8.7 on page 214.

Like most memory devices, the storage cells in a DRAM are arranged in a matrix which is approximately square. Unlike most other memory devices, this organization is exposed to the user. The matrix is addressed by *row* and by *column,* and a DRAM accepts the row and column addresses separately down the same multiplexed address bus. First the row address is presented and latched using the active-low row

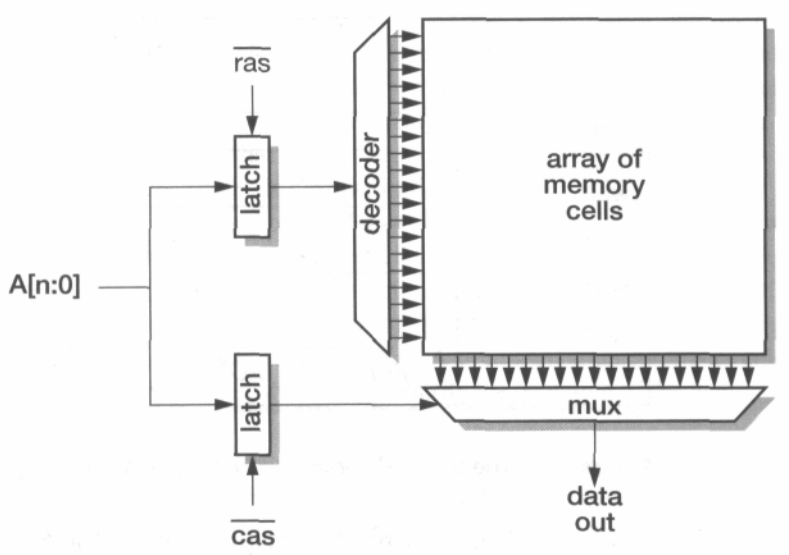


Figure 8.7 DRAM memory organization.

address strobe signal *(ras),* then the column address is presented and latched using the active-low **column address strobe** *(cas).* If the next access is within the same row, a new column address may be presented without first supplying a new row address. Since a cos-only access does not activate the cell matrix it can deliver its data two to three times faster than a full *ras-cas* access and consumes considerably less power. It is therefore very advantageous to use *cas-on\y* accesses whenever possible.

The difficulty is in detecting early enough in the memory access that the new address is in the same row as the previous address. Performing a comparison of the relevant bits of the new address with the corresponding bits of the previous address is almost always too slow.

ARM address incrementer

The solution adopted on the ARM exploits the fact that most addresses (typically 75%) are generated in the address incrementer. The ARM address selection logic (shown in Figure 8.8 on page 215) picks the address for the next cycle from one of four sources. One of these sources is the incrementer. The ARM indicates to the outside world whenever the next address is coming from the incrementer by asserting the *seq* output. External logic can then look at the previous address to check for row boundaries; if the previous address is *not* at the end of a row and the *seq* signal is asserted then a *cas-on\y* memory access can be performed.

Although this mechanism will not capture all accesses which fall within the same DRAM row, it does find most of them and is very simple to implement and exploit. The *seq* signal and the previous address are all available over half a clock cycle before the cycle in question, giving the memory control logic plenty of time.

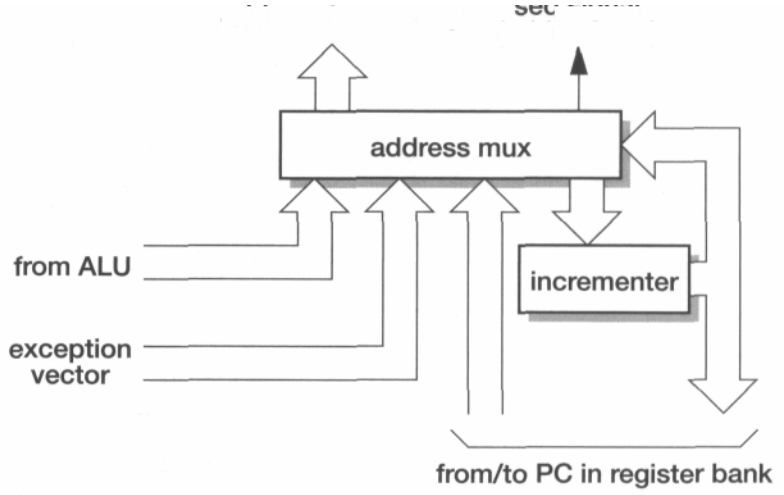


Figure 8.8 ARM address register structure.

A typical DRAM timing diagram is shown in Figure 8.9. The first, non-sequential access takes two clock cycles as the row address is strobed in, but subsequent sequential addresses use a cos-only access and operate in a single clock cycle. (Note that early address timing is now used, with *ape* or *ale* high.)

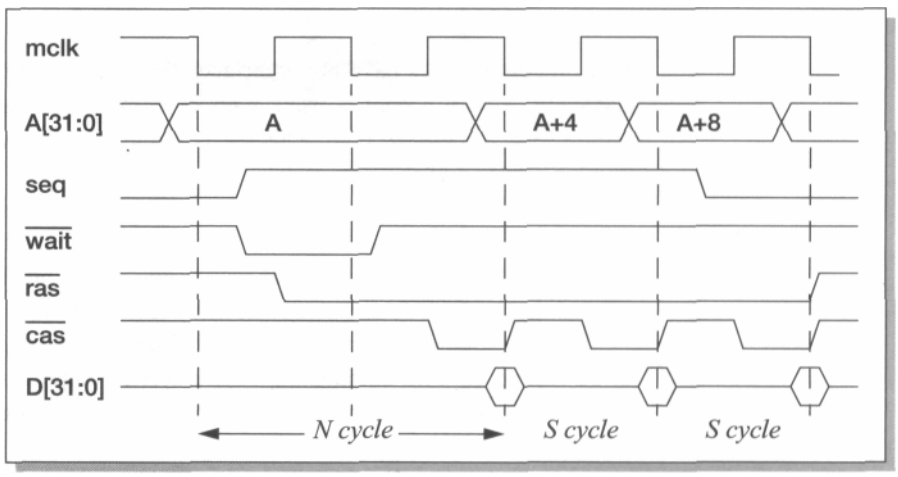


Figure 8.9 DRAM timing illustration.

The other use of the *seq* signal, to indicate a cycle which will use the same address as the preceding internal or coprocessor register transfer cycle, can also be exploited to improve DRAM access times. The DRAM access is started in the preceding cycle, which is only possible because *seq* is available so early. Typical timing is illustrated in Figure 8.10. *(wait* is inactive during this sequence.)

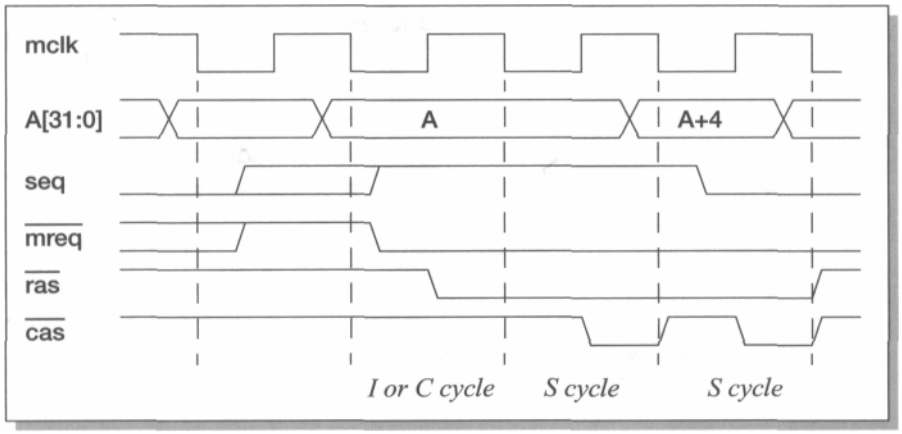


Figure 8.10 DRAM timing after an internal cycle.

Peripheral access

Most systems incorporate peripheral devices in addition to the memory components described so far. These often have slow access speeds, but can be interfaced using techniques similar to those described above for ROM access.

8.2 The Advanced Microcontroller Bus Architecture (AMBA)

ARM processor cores have bus interfaces that are optimized for high-speed cache interfacing. Where a core is used, with or without a cache, as a component on a complex system chip, some interfacing is required to allow the ARM to communicate with other on-chip macrocells.

Although this interfacing is not particularly difficult to design, there are many potential solutions. Making an *ad hoc* choice in every case consumes design resource and inhibits the reuse of peripheral macrocells. To avoid this waste, ARM Limited specified the Advanced Microcontroller Bus Architecture, AMBA, to standardize the on-chip connection of different macrocells. Macrocells designed to this bus interface can be viewed as a kit of parts for future system chips, and ultimately designing a complex system on a chip based on a new combination of existing macrocells could become a straightforward task.

AMBA buses

Three buses are denned within the AMBA specification:

- The *Advanced High-performance Bus* (AHB) is used to connect high-performance system modules. It supports burst mode data transfers and split transactions, and all timing is reference to a single clock edge.

- The *Advanced System Bus* (ASB) is used to connect high-performance system modules. It supports burst mode data transfers.
- *The Advanced Peripheral Bus* **(APB)** offers a simpler interface for low-performance peripherals.

A typical AMBA-based microcontroller will incorporate either an AHB or an ASB together with an APB as illustrated in Figure 8.11. The ASB is the older form of system bus, with AHB being introduced later to improve support for higher performance, synthesis and timing verification.

The APB is generally used as a local secondary bus which appears as a single slave module on the AHB or ASB.

In the following sections we assume the system bus is an ASB. More details on the AHB are provided at the end of the section.

Arbitration

A bus transaction is initiated by a bus master which requests access from a central arbiter. The arbiter decides priorities when there are conflicting requests, and its design is a system specific issue. The ASB only specifies the protocol which must be followed:

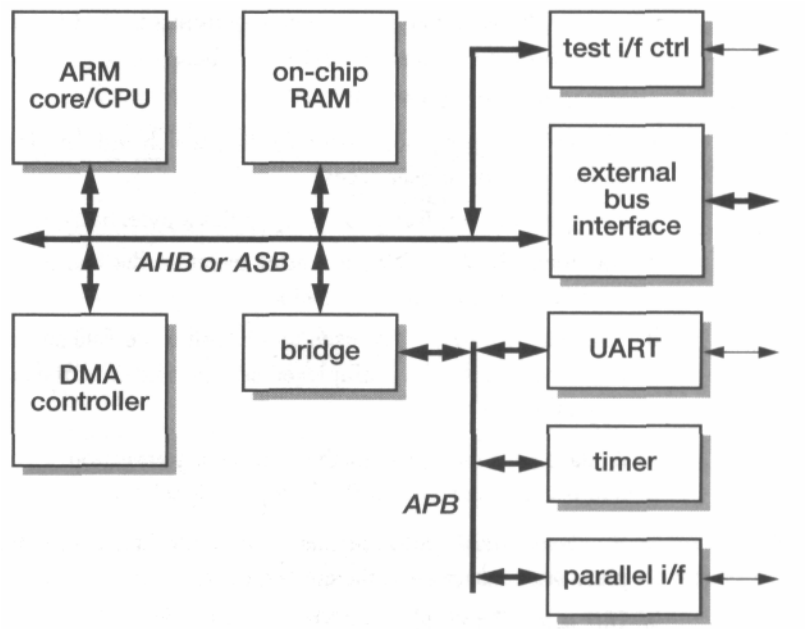


Figure 8.11 A typical AMBA-based system.

- The master, *x,* issues a request *(AREQx)* to the central arbiter.
- When the bus is available, the arbiter issues a grant *(AGNTx)* to the master. (The arbitration must take account of the bus lock signal *(BLOK)* when deciding which grant to issue to ensure that atomic bus transactions are not violated.)

Bus transfers

When a master has been granted access to the bus, it issues address and control information to indicate the type of the transfer and the slave device which should respond. The following signal is used to define the transaction timing:

- The bus clock, *BCLK.* This will usually be the same as *mclk,* the ARM processor clock.

The bus master which holds the grant then proceeds with the bus transaction using the following signals:

- Bus transaction, *BTRAN[1:0],* indicates whether the next bus cycle will be address-only, sequential or non-sequential. It is enabled by the grant signal and is ahead of the bus cycle to which it refers.
- The address bus, *BA[31:OJ.* (Not all address lines need be implemented in sys tems with modest address-space requirements, and in a multiplexed implementa tion the address is sent down the data bus.)
- Bus transfer direction, *BWRITE.*
- Bus protection signals, *BPROT[1:0],* which indicate instruction or data fetches and supervisor or user access.
- The transfer size, *BSIZE[1:0],* specifies a byte, half-word or word transfer.
- Bus lock, *BLOK,* allows a master to retain the bus to complete an atomic read-modify-write transaction.
- The data bus, *BD[31:0],* used to transmit write data and to receive read data. In an implementation with multiplexed address and data, the address is also transmitted down this bus.

A slave unit may process the requested transaction immediately, accepting write data or issuing read data on *ED[31:0],* or signal one of the following responses:

- Bus wait, *BWAIT,* allows a slave module to insert wait states when it cannot com plete the transaction in the current cycle.
- Bus last, *BLAST,* allows a slave to terminate a sequential burst to force the bus master to issue a new bus transaction request to continue.
- Bus error, *BERROR,* indicates a transaction that cannot be completed. If the master is a processor it should abort the transfer.

Bus reset

The ASB supports a number of independent on-chip modules, many of which may be able to drive the data bus (and some control lines). Provided all the modules obey the bus protocols, there will only be one module driving any bus line at any time. Immediately after power-on, however, all the modules come up in unknown states. It takes some time for a clock oscillator to stabilize after power-up, so there may be no reliable clock available to sequence all the modules into a known state. In any case, if two or more modules power-up trying to drive bus lines in opposite directions, the output drive clashes may cause power supply crow-bar problems which may prevent the chip from powering up properly at all.

Correct ASB power-up is ensured by imposing an asynchronous reset mode that forces all drivers off the bus independently of the clock.

Test interface

A possible use of the AMBA is to provide support for a modular testing methodology through the *Test Interface Controller.* This approach allows each module on the AMBA to be tested independently by allowing an external tester to appear as a bus master on the ASB.

The only requirement for test mode to be supported is that the tester has access to the ASB through a 32-bit bidirectional port. Where a 32-bit bidirectional data bus interface to external memory or peripheral devices exists, this suffices. Where the off-chip data interface is only 16 or 8 bits wide other signals such as address lines are required to give 32 lines for test access.

The test interface allows control of the ASB address and data buses using protocols defined on the two test request inputs *{TREQA* and *TREQB)* and an address latch and incrementer in the controller. A suitably designed macrocell module can then allow access to all of its interface signals in groups of up to 32 bits. For example, the ARM? macrocell has a 13-bit control and configuration input, a 32-bit data input, a 15-bit status output and 32-bit address and data outputs. The test vectors are applied and the responses sensed following a sequence defined by a finite state automata whose state transitions are controlled by *TREQA* and *TREQB.*

The AMBA macrocell test methodology may be compared with the JTAG-based methodology proposed in 'Macrocell testing' on page 230. Although perhaps less general, the AMBA approach will reduce test cost due its parallel tester interface.

Advanced Peripheral Bus

The ASB offers a relatively high-performance on-chip interconnect which suits processor, memory and peripheral macrocells with some built-in interface sophistication. For very simple, low-performance peripherals, the overhead of the interface is too high. The **Advanced Peripheral Bus** is a simple, static bus which operates as a stub on an ASB to offer a minimalist interface to very simple peripheral macrocells.

The bus includes address *(PADDR[n:0]*\ the full 32 bits are not usually required) and read and write data *(PRDATA[m:0]* and *PWDATA[m:0],* where *m* is 7, 15 or 31) buses which are no wider than necessary for the connected peripherals, a read/write direction indicator *(PWRITE),* individual peripheral select strobes *(PSELx)* and a

peripheral timing strobe *(PENABLE).* APB transfers are timed to *PCLK,* and all APB devices are reset with *PRESETn.*

The address and control signals are all set up and held with respect to the timing strobe to allow time for local decoding with the select acting as the local enable. Peripherals which are slave devices based around simple register mapping may be interfaced directly with minimal logic overhead.

Advanced High-performance BUS

The AHB is intended to replace the ASB in very high performance systems, for example those based on the ARM1020E (described in Section 12.6 on page 341).

The following features differentiate the AHB from the ASB:

- It supports split transactions, where a slave with a long response latency can free up the bus for other transfers while it prepares its data for transmission.
- It uses a single clock edge to control all of its operations, aiding synthesis and design verification (through the use of static timing analysis and similar tools).
- It uses a centrally multiplexed bus scheme rather than a bidirectional bus with tristate drivers (see Figure 8.12 on page 221).
- It supports wider data bus configurations of 64 or 128 bits.

The multiplexed bus scheme may appear to introduce a lot of excess wiring, but bidirectional buses create a number of problems for designers and even more for synthesis systems. For example, as chip feature sizes shrink wire delays begin to dominate performance issues, and unidirectional buses can benefit from the insertion of repeater drivers that are very hard to add to a bidirectional bus.

8.3 The ARM reference peripheral specification

The support for system development described so far in this chapter is principally aimed at testing and providing low-level access to processor and system state. AMBA offers a systematic way to connect hardware components together on a chip, but software development must still start from first principles on each new chip.

If a system developer wishes to start from a higher baseline, for example to base the software on a particular real-time operating system, then a number of components must be available to support the basic operating system functions. The **ARM reference peripheral specification** defines such a basic set of components, providing a framework within which an operating system can run but leaving full scope for application-specific system extensions.

The objective of the reference peripheral specification is to ease the porting of software between compliant implementations and thereby raise the level from which software development begins on a new system.

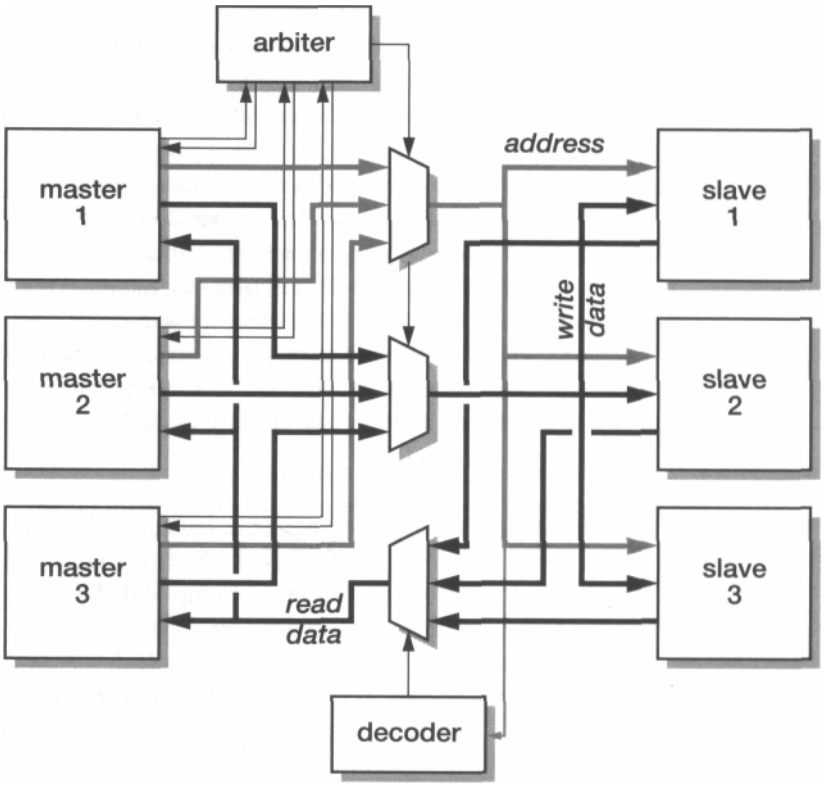


Figure 8.12 AHB multiplexed bus scheme.

Base components

The reference peripheral specification defines the following components:

- A memory map which allows the base address of the interrupt controller, the counter timers and the reset controller to vary but defines the offsets of the vari ous registers from these base addresses.
- An interrupt controller with a defined set of functions, including a defined inter rupt mechanism for a transmit and receive communications channel (though the mechanism of the channel itself is not defined).
- A counter timer with various defined functions.
- A reset controller with defined boot behaviour, power-on reset detection, a 'wait for interrupt' pause mode and an identification register.

The particular ARM core used with these components is not specified since this does not affect the system programmer's model.

Memory map

The system must define the base addresses of the interrupt controller **(ICBase),** the counter-timer **(CTBase)** and the reset and pause controller **(RPCBase).**

These addresses are not defined by the reference peripheral specification, but all the addresses of the registers are defined relative to one or other of these base addresses.

Interrupt controller

The interrupt controller provides a uniform way of enabling, disabling and examining the status of up to 32 level-sensitive IRQ sources and one FIQ source. Each interrupt source has a mask bit which enables that source. Memory locations are defined with fixed offsets from *ICBase* to examine the unmasked, mask and masked interrupt status and to set or clear interrupt sources.

Five IRQ sources are defined by the reference peripheral specification, corresponding to the communication receive and transmit functions, one for each counter-timer and one which can be generated directly by software (principally to enable an FIQ handler to generate an IRQ).

Counter-timers

Two 16-bit counter-timers are required, though more may be added. These are controlled by registers with fixed offsets relative to *CTBase.* The counters operate from the system clock with selectable pre-scaling of 0, 4 or 8 bits (so the input frequency is the system clock frequency divided by 1, 16 or 256).

Each counter-timer has a control register which selects the pre-scaling, enables or disables the counter and specifies the mode of operation as free-running or periodic, and a load register which specifies the value that the count starts from. A write to the 'load' register initializes the count value, which is then decremented to zero when an interrupt is generated. A write to the 'clear' register clears the interrupt. In free-running mode the counter continues to decrement past zero, whereas in periodic mode it is reloaded with the value in the 'load' register and decrements from there.

The current count value may be read from the 'value' register at any time.

Reset and pause controller

The reset and pause controller includes registers which are addressed at fixed offsets from *RPCBase.* The readable registers give identification and reset status information, including whether or not a power-on reset has occurred. The writeable registers can set or clear the reset status (though not the power-on reset status bit; this can only be set by a hardware power-on reset), clear the reset map (for instance to switch the ROM from location zero, where it is needed after power-on for the ARM reset vector, to the normal memory map), and put the system into pause mode where it uses minimal power until an interrupt wakes it up again.

System design

Any ARM system which incorporates this basic set of components can support a suitably configured operating system kernel. System design then consists of adding further application-specific peripherals and software, building upwards from a functional base.

Since most applications require these components there is little overhead incurred in using the reference peripheral specification as the starting point for system development, and there is considerable benefit in starting from a functional system.

8.4 Hardware system prototyping tools

The task facing today's system-on-chip designer is daunting. The number of gates on a chip continues to grow at an exponential rate, and is already in the millions. With the best software design tools available on the market, designers cannot produce fully tested systems of this complexity within the time-to-market constraints.

The first step towards addressing this problem, as has already been indicated, is to base a significant proportion of the design on pre-existing design components. Design reuse can reduce the amount of new design work to a small fraction of the total number of gates on the chip. A systematic approach to on-chip interconnect through the use of a bus such as AMBA further reduces the design task. However, there are still difficult problems to be solved, such as:

- How can the designer be sure that all of the selected re-usable blocks, which may come from various sources, will really work together correctly?
- How can the designer be sure that the specified system meets the performance requirements, which often include complex real-time issues?
- How can the software designers progress their work before the chip is available?

Simulating the system using software tools usually results in a performance that is several orders of magnitude lower than that of the final system, rendering software development and full system verification impractical.

A solution that goes a considerable way towards addressing all these issues is the use of hardware prototyping: building a hardware system that combines all of the required components in a form that makes no attempt to meet the power and size constraints of the final system, but does provide a platform for system verification and software development. The ARM 'Integrator' is one such system; another is the 'Rapid Silicon Prototyping' system from VLSI Technology, Inc.

Rapid Silicon Prototyping

VLSI Technology, Inc., have introduced a development system called 'Rapid Silicon Prototyping'. The basis of this system is to use specially developed reference chips, each offering a particular plentiful set of on-chip components and support for off-chip extensions, which can be used to prototype a system-on-chip design. The target system is modelled in two steps:

1. The selected reference chip is 'deconfigured' to render those on-chip blocks that are not required in the target system inactive.
2. The blocks required in the target system that are not available on the reference chip are implemented as off-chip extensions. These may be either existing integrated circuits with the necessary functionality or FPGAs configured to the required function (usu ally by synthesis from a high-level language such as VHDL).

The use of pre-existing blocks, interconnected using standard buses such as AMBA, minimizes the technical risk in producing the final chip. All of the blocks in the reference chip exist as synthesizable components. The high-level language descriptions of the required functions are taken together with the high-level language source used to configure the FPGAs, the deconfigured functions are discarded, and the result resynthesized to give the target chip. This process is illustrated in Figure 8.13.

It is clear that this approach depends on the reference chip containing appropriate key components, such as the CPU core and possibly a signal processing system (where this is demanded by the target application). Although in principle it might be possible to build a single reference chip that contains every different ARM processor core (including all the different cache and MMU configurations), in practice such a chip would not be economic even for prototyping purposes. The success of the approach therefore depends on a careful choice of the components on the reference chip to ensure that it can cover a wide spread of systems, and different reference chips with different CPU and other key cores being built for different application domains.

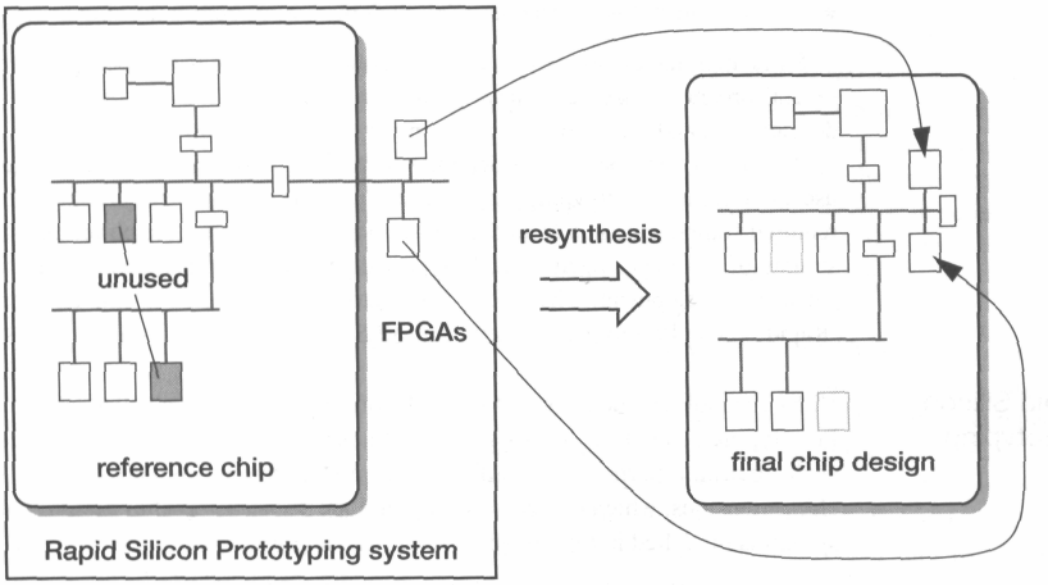


Figure 8.13 Rapid Silicon Prototyping principle.

8.5 The ARMulator

The ARMulator is part of the cross-development toolkit described in Section 2.4 on page 43. It is a software emulator of the ARM processor which supports the debugging and evaluation of ARM code without requiring an ARM processor chip.

The ARMulator has a role in embedded system design. It supports the high-level prototyping of various parts of the system to support the development of software and the evaluation of architectural alternatives. It is made up of four components:

- The processor core model, which can emulate any current ARM core, including the Thumb instruction set.
- A memory interface which allows the characteristics of the target memory system to be modelled. Various models are supplied to support rapid prototyping, but the interface is fully customizable to incorporate the level of detail required.
- A coprocessor interface that supports custom coprocessor models.
- An operating system interface that allows individual system calls to be handled by the host or emulated on the ARM model.

The processor core model incorporates the remote debug interface, so the processor and system state are visible from ARMsd, the ARM symbolic debugger. Programs can be loaded, run and debugged through this interface.

System modelling

Using the ARMulator it is possible to build a complete, clock-cycle accurate software model of a system including a cache, MMU, physical memory, peripheral devices, operating system and software. Since this is likely to be the highest-level model of the system, it is the best place to perform the initial evaluation of design alternatives.

Once the design is reasonably stable, hardware development will probably move into a timing-accurate CAD environment, but software development can continue using the ARMulator-based model (probably moving up from cycle- to instruction-accurate timing for higher performance).

In the course of the detailed hardware design it is likely that some of the timing assumptions built into the original software model prove impossible to meet. As the design evolves it is important to keep the software model in step so that the software development is based on the most accurate estimates of timing that are available.

It is now common for complex systems development to be supported by multiple computer models of the target system built upon different levels of abstraction. Unless the lower-level models are synthesized automatically from the more abstract models, maintaining consistency between the models always takes considerable care and effort.

8.6 The JTAG boundary scan test architecture

Two difficult areas in the development of a product based around an application specific embedded system chip are the production testing of the VLSI component and the production testing of the assembled printed circuit board.

The second of these is addressed by the IEEE standard number 1149, 'Standard Test Access Port and Boundary-Scan Architecture'. This standard describes a 5-pin serial protocol for accessing and controlling the signal levels on the pins of a digital circuit, and has extensions for testing the circuitry on the chip itself. The standard was developed by the *Joint Test Action Group* (hence JTAG), and the architecture described by the standard is known either as 'JTAG boundary scan' or as 'IEEE 1149'.

The general structure of the JTAG boundary scan test interface is shown in Figure 8.14. All the signals between the core logic and the pins are intercepted by the serial scan path which can connect the core logic to the pins in normal operating mode, or can read out the original value and replace it with a new value in test mode.

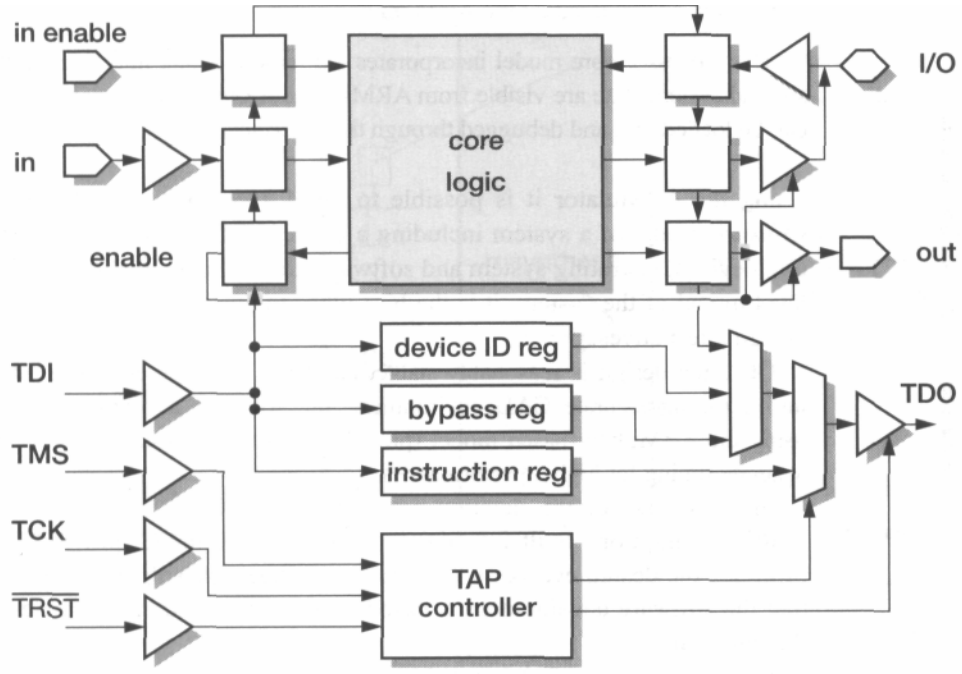


Figure 8.14 JTAG boundary scan organization.

Test signals

The interface works with five dedicated signals which must be provided on each chip that supports the test standard:

- *TRST* is a test reset input which initializes the test interface.
- *TCK* is the test clock which controls the timing of the test interface independ ently from any system clocks.
- *TMS* is the test mode select which controls the operation of the test interface state machine.
- *TDI* is the test data input line which supplies the data to the boundary scan or instruction registers.
- *TDO* is the test data output line which carries the sampled values from the boundary scan chain and propagates data to the next chip in the serial test circuit.

The normal organization of the test circuit on a board that incorporates several chips with JTAG support is to connect *TRST, TCK* and *TMS* to every chip in parallel and to connect *TDO* from one chip to *TDI* of the next in a single loop, so the board test interface has the same five signals listed above.

TAP controller

The operation of the test interface is controlled by the Test Access Port (TAP) controller. This is a state machine whose state transitions are controlled by *TMS;* the state transition diagram is shown in Figure 8.15 on page 228. All the states have two exits so the transitions can be controlled by one signal, *TMS.* The two main paths in the state transition diagram control the operation of a data register (DR) and the instruction register (IR).

Data registers

The behaviour of a particular chip is determined by the contents of the test instruction register, which can select between various different data registers:

- The device ID register reads out an identification number which is hard-wired into the chip.
- The bypass register connects *TDI to TDO* with a 1-clock delay to give the tester rapid access to another device in the test loop on the same board.
- The boundary scan register intercepts all the signals between the core logic and the pins and comprises the individual register bits which are shown as the squares connected to the core logic in Figure 8.14 on page 226.
- Other registers may be employed on the chip to test other functions as required.

Instructions

The normal operation of a JTAG test system is to enter an instruction which specifies the sort of test to be carried out next and the data register to be used for that test into the *instruction* register, and then to use the data register to carry out the test.

Instructions may be public or private. Public instructions are declared and available for general test use, and the standard specifies a minimum set of public instructions

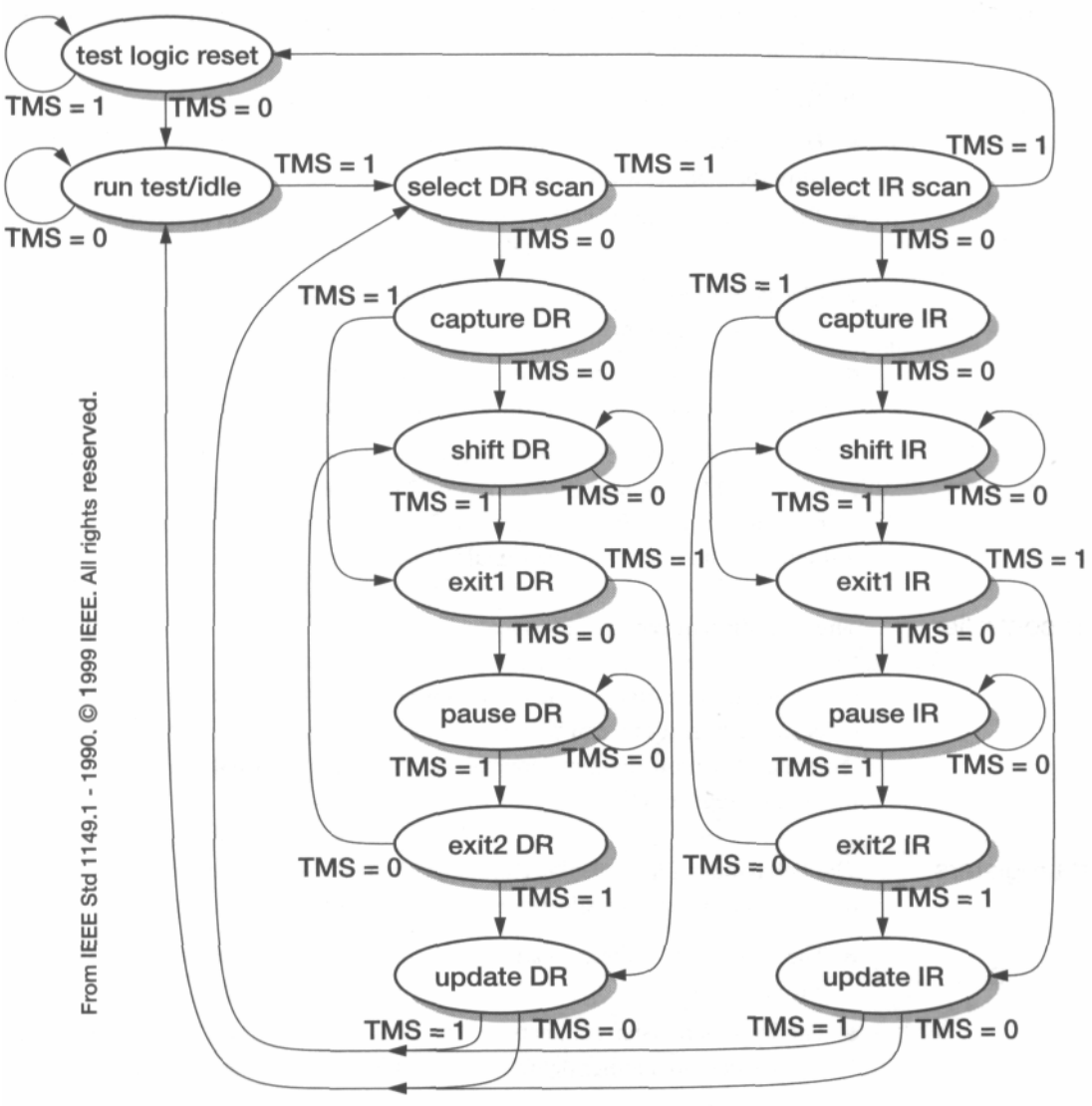


Figure 8.15 Test Access Port (TAP) controller state transition diagram.

that must be supported by all devices that comply with the standard. Private instructions are for specialized on-chip test purposes and the standard does not specify how these should operate or how they should be used.

Public instructions

The minimum set of public instructions that all compliant devices must support is:

- BYPASS: here the device connects *TDIto TOO* though a single clock delay. This instruction exists to facilitate the testing of other devices in the same test loop.

- EXTEST: here the boundary scan register is connected between *TDI* and *TOO* and the pin states are captured and controlled by the register. Referring to the state transition diagram in Figure 8.15 on page 228, the pin states are captured in the *Capture DR* state and shifted out of the register via the *TDO* pin in the *Shift DR* state. As the captured data is shifted out, new data is shifted in via the *TDI* pin, and this data is applied to the boundary scan register outputs (and hence the output pins) in the *Update DR* state. This instruction exists to support the testing of board-level connectivity.
- IDCODE: here the ID register is connected between *TDI* and *TDO*. In the *Capture DR* state the device ID (a hard-wired identification number giving the manufacturer, part number and version of the part) is copied into the register which is then shifted out in the *Shift DR* state.

Other public instructions may include:

- INTEST: here the boundary scan register is connected between *TDI* and *TDO* and the core logic input and output states are captured and controlled by the register. Note that the inputs are driven to the complement of the values supplied. Other wise the operation is similar to EXTEST. This instruction exists to support the testing of the core logic.

PCB testing

The principal goal of the JTAG test circuit is to enable the continuity of tracks and the connectivity of solder connections on printed circuit boards *(PCBs)* to be tested. This has become difficult since the introduction of surface mount packages which do not require through-holes in the circuit board. Previously 'bed of nails' testers could contact all the pins on each 1C package from the back of the PCB to check continuity; surface mount packages typically have the pins closer together and the tracks accessible only on the component side of the board, making the 'bed of nails' approach inapplicable.

When the surface mount components have a JTAG test interface, this can be used (using the EXTEST instruction) to control outputs and observe inputs independently of the normal function of the chip, so board-level connectivity can readily be checked from chip to chip. If the board also contains components which do not have a JTAG interface these will require 'bed of nails' contacts, but these can be used together with the JTAG interfaces where available to minimize the cost and difficulty of building the production test equipment.

VLSI testing

High-complexity integrated circuits require extensive production testing to identify faulty devices before they get built into product. A production 1C tester is a very expensive piece of equipment, and the time each device spends on the tester is an important factor in its manufacturing cost. Since the JTAG test circuitry operates through serial access, it is not a high-speed way to apply test vectors to the core

logic. Furthermore, it is not possible to apply test vectors through the JTAG port at the normal operating speed of the device to check its performance.

Therefore the JTAG architecture is not a generic solution to all the problems of VLSI production testing. However, it can solve a number of problems:

- The JTAG port can be used for in-circuit functional testing of an 1C (provided that the INTEST instruction is supported).
- It gives good control of the 1C pins for parametric testing (checking the drive of the output buffers, leakage, input thresholds, and so on). This uses only the EXTEST instruction which is required on all JTAG compliant devices.
- It can be used to access internal scan paths to improve the controllability and observability of internal nodes that are hard to access from the pins.
- It can be used to give access to on-chip debug functions with no additional pins and without interfering with the system functions. This is exploited by the ARM EmbeddedlCE debug architecture described briefly below and in detail in Section 8.7 on page 232.
- It offers an approach to the functional testing of macrocell-based designs as described below.

These uses are all in addition to its principal purpose, which is in the production testing of printed circuit boards.

Embedded-ICE

The ARM debug architecture, described in Section 8.7 on page 232, is based on an extension of the JTAG test port. The EmbeddedlCE module introduces breakpoint and watchpoint registers which are accessed as additional data registers using special JTAG instructions, and a trace buffer which is similarly accessed. The scan path around the ARM core macrocell is used to introduce instructions into the ARM pipeline without interfering with other parts of the system and these instructions can be used to access and modify the ARM and system state.

The debug architecture gives most of the functionality of a conventional In-Circuit Emulation system to debug an ARM macrocell on a complex system chip, and since the JTAG test access port is used to control the debug hardware, no additional pins are required on the chip.

Macrocell testing

A growing trend in the design of complex system chips is to incorporate a number of complex, pre-designed macrocells, together with some application-specific custom logic. The ARM processor core is itself one such macrocell; the others may come from ARM Limited, a semiconductor partner or some third party supplier. In such cases the designer of the system chip will have limited knowledge of the macrocells and will depend on the macrocell suppliers for the production test patterns for each macrocell.

Since the macrocells are buried within the system chip, the designer is faced with the problem of devising a way to apply the supplied test vectors to each of the macro-cells in turn. Test patterns must also be generated for the custom logic part of the design, but the designer is assumed to understand that part of the logic.

There are various approaches to getting the test patterns onto the edges of the macrocells:

- Test modes may be provided which multiplex the signals from each macrocell in turn onto the pins of the system chip.
- An on-chip bus may support direct test access to each macrocell which is attached to it (see Section 8.2 on page 216).
- Each macrocell may have a boundary scan path through which the test patterns may be applied using an extension of the JTAG architecture.

This last approach is illustrated in Figure 8.16. The chip has a peripheral boundary scan path to support the public EXTEST operation and additional paths around each macrocell, designed into the macrocell, for applying functional tests as supplied. The custom logic designed specifically for this chip may have its own scan path or, as shown in the figure, rely on the fact that all its interface signals must intercept one of the existing scan paths.

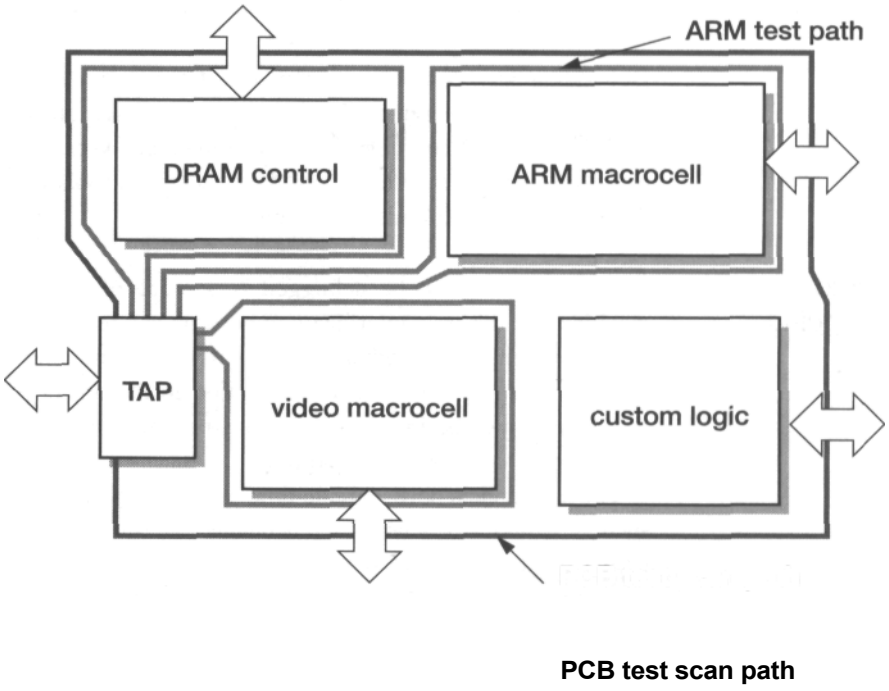


Figure 8.16 A possible JTAG extension for macrocell testing.

It should be recognized that although perfectly feasible for functional testing, the scan path approach to macrocell testing has the same drawbacks as using the JTAG boundary scan path to test the core logic on a chip. The serial access is much slower than parallel access through the pins and performance testing at speed is not possible.

The most promising production test methodology for macrocell based system chips appears to be to exploit the on-chip bus to give parallel access to the macrocell's periphery (especially where the macrocell has been designed specifically to give good access through this route). Multiplexing is used to give external access to peripheral macrocell signals that are important for performance testing and cannot conveniently be accessed via the on-chip bus, and the JTAG port is used to access other signals and internal state where necessary via scan chains. We have seen this approach as it is supported by ARM's 'Advanced Microcontroller Bus Architecture' (AMBA) which is described in Section 8.2 on page 216; the AMBA testing methodology is described on page 219.

The JTAG system continues to be very important for board-level testing, and can also be used for in-circuit testing of the core logic and to access on-chip debug facilities. It is incorporated into most ARM designs and is an important component of their test and debug methodologies.

8.7 The ARM debug architecture

Debugging any computer system can be a complex task. There are two basic approaches to debugging, the simplest being based on watching a system from the outside using test equipment such as a logic analyser, and the more powerful being based on viewing a system from the inside with tools that support single stepping, the setting of breakpoints, and so on.

Desktop debugging

When the system to be debugged is a piece of software running on a desktop machine, all the user interface components are readily available and the debugger may itself simply be another piece of software running on the same machine. Breakpoints are set by replacing an instruction in the object program with a call to the debugger, remembering the original instruction so that it can be replaced when execution continues past the breakpoint.

Often compilers have compile-time options to generate extensive debug information such as symbol tables which the debugger can use to allow the user to debug the program from a source-level view, addressing variables by their source names rather than by their memory address. This 'source-level debugging' is more powerful and requires less detailed knowledge of the machine environment than object-level debugging.

A common weakness in software debuggers is the lack of a 'watchpoint' facility. A watchpoint is a memory address which halts execution if it is accessed as a data transfer address. Since many processors have no support for trapping on a particular

address (apart, perhaps, from a memory management page fault, which is rather coarse for this purpose) this functionality is often omitted. This is a pity, since a very common source of error in a C program is corrupted data caused by an errant pointer in some unrelated part of the program, and this can be very hard to track down without a watchpoint facility.

Embedded debugging

Debugging becomes significantly more difficult when the target system is embedded. Now there is probably no user interface in the system, so the debugger must run on a remote host through some sort of communication link to the target. If the code is in ROM, instructions cannot simply be replaced by calls to the debugger since the locations are not writeable.

The standard solution here is the *In-Circuit Emulator* (ICE). The processor in the target system is removed and replaced by a connection to an emulator. The emulator may be based around the same processor chip, or a variant with more pins (and more visibility of its internal state), but it will also incorporate buffers to copy the bus activity off to a 'trace buffer' (which stores the signals on all the pins in each clock cycle for some number of cycles) and various hardware resources which can watch for particular events, such as execution passing through a breakpoint. The trace buffer and hardware resources are managed by software running on a host desktop system.

When a 'trigger' event occurs, the trace buffer is frozen so the user can observe activity around the point of interest. The host software will present the trace buffer data, give a view on the processor and system state and allow it to be modified, and generally attempt to appear as similar to a debugger in a desktop system as possible.

Debugging processor cores

The ICE approach depends on the system having an identifiable processor chip which can be removed and replaced by the ICE. Clearly, once the processor has become just one macrocell of many on a complex system chip, this is no longer possible.

Although simulation using software models such as the ARMulator should remove many of the design bugs before the physical system is built, it is often impossible to run the full software system under emulation, and it can be difficult to model all the real-time constraints accurately. Therefore it is likely that some debugging of the complete hardware and software system will be necessary. How can this be achieved? This is still an area of active research to identify the best overall strategy with acceptable hardware overhead costs, but considerable progress has been over the last few years in developing practical approaches, and the rest of this section presents the approach developed by ARM Limited.

ARM debug hardware

To provide debug facilities comparable with those offered by a typical ICE, the user must be able to set breakpoints and watchpoints (for code running in ROM as well as RAM), to inspect and modify the processor and system state, to see a trace of processor activity around the point of interest, and to do all this from the comfort of a desk-top system with a good user interface. The trace mechanism used in ARM

systems is separate from the other debug facilities, and is discussed in Section 8.8 on page 237. In this section we restrict our attention to the breakpoint, watchpoint and state inspection resources.

The communication between the target system and the host is achieved by extending the functionality of the JTAG test port. Since the JTAG test pins are included in most designs for board testing, accessing the debug hardware through this port requires no additional dedicated pins, sparing the most precious resource on the chip from further pressure. JTAG scan chains are used to access the breakpoint and watch-point registers and also to force instructions into the processor to access processor and system state.

The breakpoint and watchpoint registers represent a fairly small hardware overhead that can often be accepted on production parts. The host system runs the standard ARM development tools, communicating with the target through a serial port and/or a parallel port. Special protocol conversion hardware sits between the host serial line and the target JTAG port.

In addition to the breakpoint and watchpoint events, it may also be desirable to halt the processor when a system-level event occurs. The debug architecture includes external inputs for this purpose.

The on-chip cell containing these facilities is called the *EmbeddedlCE* module.

Embedded-ICE

The EmbeddedlCE module consists of two watchpoint registers and control and status registers. The watchpoint registers can halt the ARM core when the address, data and control signals match the value programmed into the watchpoint register. Since the comparison is performed under a mask, either watchpoint register can be configured to operate as a breakpoint register capable of halting the processor when an instruction in either ROM or RAM is executed.

The comparison and mask logic is illustrated in Figure 8.17.

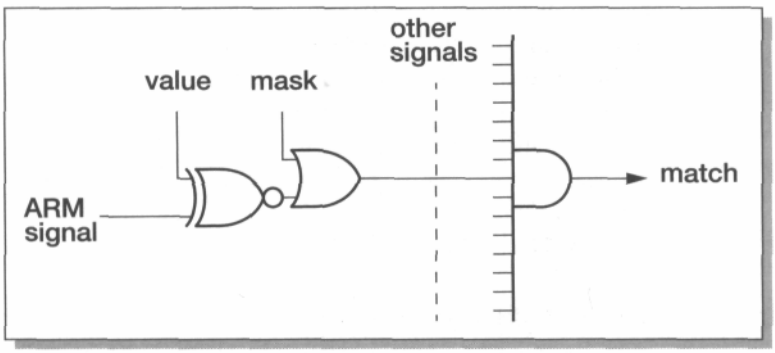


Figure 8.17 EmbeddedlCE signal comparison logic.

Chaining

Each watchpoint can look for a particular combination of values on the ARM address bus, data bus, *trans, ope,* mo?[l:0] and *r/w* control signals, and if either combination is matched the processor is stopped. Alternatively, the two watchpoints may be chained to halt the processor when the second watchpoint is matched only after the first has previously been matched.

Registers

EmbeddedlCE registers are programmed via the JTAG test port, using a dedicated scan chain. The scan chain is 38 bits long, with 32 data bits, *5* address bits and a *r/w* bit which controls whether the register is read or written. The address bits specify the particular register following the mapping detailed in Table 8.1.

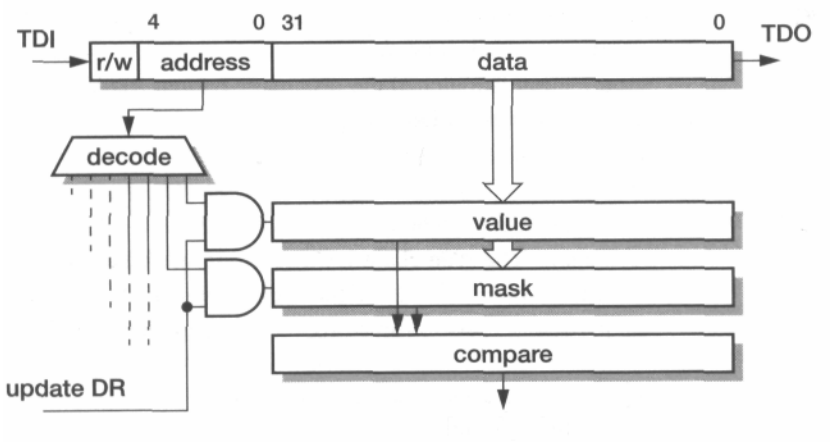
Table 8.1 EmbeddedlCE register mapping.

Address	Width	Function
00000	3	Debug control
00001	*5*	Debug status
00100	*6*	Debug comms control register
00101	32	Debug comms data register
01000	32	Watchpoint 0 address value
01001	32	Watchpoint 0 address mask
01010	32	Watchpoint 0 data value
01011	32	Watchpoint 0 data mask
01100	9	Watchpoint 0 control value
01101	8	Watchpoint 0 control mask
10000	32	Watchpoint 1 address value
10001	32	Watchpoint 1 address mask
10010	32	Watchpoint 1 data value
10011	32	Watchpoint 1 data mask
10100	9	Watchpoint 1 control value
10101	8	Watchpoint 1 control mask

The use of the JTAG scan chain is illustrated in Figure 8.18 on page 236. The read or write takes place when the TAP controller enters the 'update DR' state (see Figure 8.15 on page 228).

Accessing state

The EmbeddedlCE module allows a program to be halted at specific points, but it does not directly allow the processor or system state to be inspected or modified. This is achieved via further scan paths which are also accessed through the JTAG port.



breakpoint Figure 8.18

EmbeddedICE register read and write structure.

The mechanism employed to access the processor state is to halt the processor, then to force an instruction such as a store multiple of all the registers into the processor's instruction queue.Then clocks are applied to the processor, again via the scan chain, causing it to write the registers out through its data port. Each register is collected by the scan chain and shifted out.

System state is harder to glean, since there may be system locations that cannot be read at the very low speeds that the scan path can generate. Here the processor is pre-loaded with a suitable instruction, then allowed to access the system location at system speed. This transfers the required system state into a processor register, whereupon it may be passed to the external debugger through the JTAG port as described above.

Debug comms

In additions to the breakpoint and watchpoint registers, the EmbeddedICE module also includes a **debug comms** port whereby the software running on the target system can communicate with the host. The software in the target system sees the comms port as a 6-bit control register and 32-bit data read and write registers which are accessed using MRC and MCR instructions to coprocessor 14. The host sees these registers in the EmbeddedICE register map as shown in Table 8.1 on page 235.

Debugging

An ARM-based system chip which includes the EmbeddedICE module connects to a host computer through the JTAG port and a protocol converter. This configuration supports the normal breakpoint, watchpoint and processor and system state access that the programmer is accustomed to using for native or ICE-based debugging (in addition to the comms port described above), and with suitable host software gives a full source-level debugging capability with low hardware overhead.

The only facility that is missing is the ability to trace code in real time. This is the function of the *Embedded Trace Macrocell* described in the next section.

8.8 Embedded Trace

When debugging real-time systems it is often difficult to debug the application software without the capability of observing its operation in real time. The breakpoint and watchpoint facilities offered by the EmbeddedlCE macrocell are insufficient for this purpose as using them causes the processor to deviate from its normal execution sequence, destroying the temporal behaviour of the software.

What is required is an ability to observe the processor operating at full speed by generating a trace of its address, data and control bus activity as the program executes. The problem is that this represents a huge data bandwidth - an ARM processor running at 100 MHz generates over 1 Gbyte/s of interface information. Getting this information off the chip would require so many pins that it would be uneconomic to include the capability on production devices, so special development devices would be needed, adversely affecting the costs of developing a new system-on-chip application.

Trace compression

The solution adopted by ARM Limited is to reduce the interface bandwidth using intelligent trace compression techniques. For example:

- Most ARM addresses are sequential, so it is not necessary to send every address off chip. Instead, the sequential information can be sent on most cycles and the full address only when a branch is taken.
- If there is off-chip logic which has access to the code which is running on the processor, it will know when the processor is executing a branch instruction and where the target of the branch is. The only information which must be sent off chip is whether or not the branch is taken.
- Fuller address information is now required only when the branch target is not known, such as in a subroutine return or jump table instruction. Even here, only those low-order address bits that change need be issued.
- The address information is now very bursty. A first-in-first-out (FIFO) buffer can be used to smooth the data rate so that the necessary address information can be transmitted in 4-, 8- or 16-bit packets at a steadier rate.

Using a number of techniques similar along these lines, the ARM Embedded Trace Macrocell can compress the trace information to the extent necessary to allow it to be communicated off chip through 9, 13 or 21 pins depending on the configuration. These pins could be used for other purposes when trace output is not required.

Real-time debug

A complete real-time debug solution is as shown in Figure 8.19 on page 238. The EmbeddedlCE unit supports breakpoint and watchpoint functionality, and communication channels between the host and target software. The Embedded Trace Macrocell compresses the processor's interface information and sends it off chip through the Trace port. The JTAG port is used to control both units. The external EmbeddedlCE

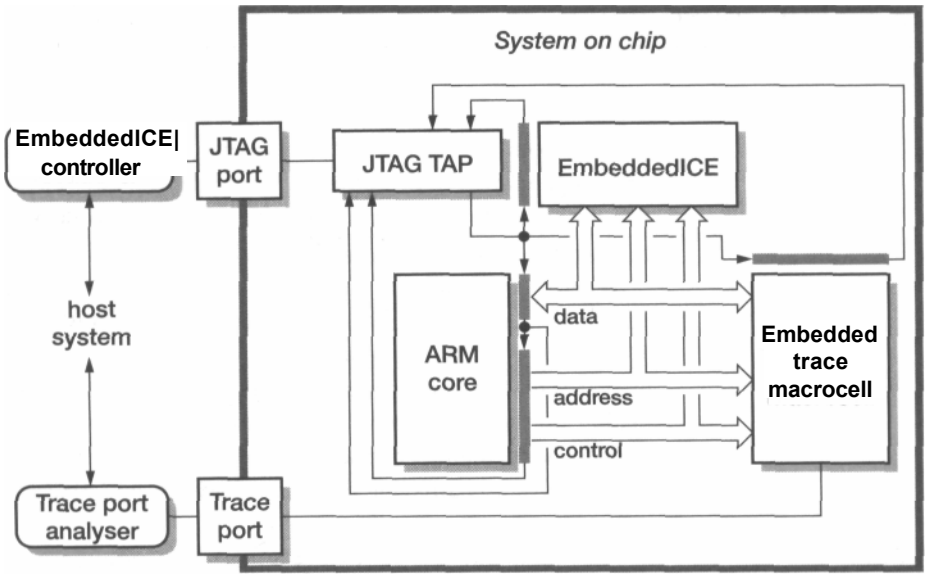


Figure 8.19 Real-time debug system organization.

controller is used to connect the host system to the JTAG port, and the external Trace Port Analyser interfaces the host system to the Trace port. The host may connect to both the trace port analyser and the EmbeddedlCE controller via a network.

The user has control of the breakpoint and watchpoint settings and various trace functions. All the application software may be traced, or just particular routines. Trigger conditions can be specified, and the trace can be collected before or after the trigger or with the trigger in the centre of the trace. Data accesses can be selected to be included in the trace or not, and the trace may collect just the address of the data access, or just the data itself, or both.

Embedded trace options

As noted above, the Embedded Trace Macrocell may be synthesized in several different configurations allowing the functionality of the unit to be traded off against cost (measured in terms of the numbers of gates and pins used).

- The minimal system requires 5 pins to issue pipeline information and 4 pins to issue data (in addition to the 5-pin JTAG interface). This is sufficient for execution tracing, but will only support very limited data tracing and has restricted triggering and filter ing capabilities. A 9-byte FIFO is used to smooth the data transfer rate, and the hard ware cost of this implementation is approximately 15 Kgates.
- A maximal system uses 5 pins to issue pipeline information and 16 pins to issue data (again, in addition to the 5-pin JTAG interface). It is capable of tracing the flow of execution and all but the very-worst-case data activity. A 40-byte FIFO is used to smooth the data flow, and the hardware cost is approximately 50 Kgates.

Between these two extremes several intermediate configurations are possible. All allow for external inputs (that is, inputs from other logic on the chip) to control the trace triggering, and for triggering from the EmbeddedlCE breakpoint logic.

Trace overflow

With all of the implementations of the Embedded Trace Macrocell there are some circumstances where the trace FIFO buffer can overflow. The unit can be configured either to stall the processor or to discontinue tracing when this happens. In either case, real-time tracing is lost, although only temporarily while the FIFO drains.

Fundamentally what has happened is the information bandwidth has exceeded the capability of the compression algorithm to reduce it to match the bandwidth capacity of the trace port. If this happens, the filtering setup must be modified to reduce the amount of data that is being traced.

N-Trace

The real-time trace technology was developed through a collaboration between ARM Limited, VLSI Technology, Inc., and Agilent Technologies (formerly part of Hewlett-Packard). VLSI Technology, Inc., offers a synthesizable version of the Embedded Trace Macrocell under the 'N-Trace' product name.

Trace port analyser

The trace port analyser may be a conventional logic analyser, but an ARM-specific low-cost trace port analyser has been developed by Agilent Technologies and similar systems will become available from other vendors.

Trace software tools

The Embedded Trace Macrocell is configured via the JTAG port using software that is an extension to the ARM software development tools. The trace data is downloaded from the trace port analyser and decompressed using source code information. It is then presented as an assembly listing with interspersed data accesses, and has links back to the source code.

With EmbeddedlCE and the Embedded Trace facility the ARM system-on-chip designer has all the facilities offered by traditional in-circuit emulation (ICE) tools. These technologies give full visibility of the real-time behaviour of the application code with the ability to set breakpoints and inspect and change processor registers and memory locations, always with firm links back to the high-level language source code.

8.9 Signal processing support

Many applications that use an ARM processor as a controller also require significant digital signal processing performance. A typical GSM mobile telephone handset is a case in point; first-generation ARM-based designs typically incorporate a DSP core on the same chip as the ARM core, and the system designer has to make careful

choices regarding which system functions are best implemented on the DSP core and which on the ARM core.

DSP cores have programmers' models that are very different from the ARM's model. They employ several separate data memories, and often require the programmer to schedule their internal pipeline very carefully if maximum throughput is to be obtained. Synchronizing the DSP code with the ARM code that is running concurrently is a complex task.

ARM Limited has introduced two different extensions to the ARM architecture in attempting to simplify the system design task in applications which require both controller and signal processing functions: the Piccolo coprocessor, and the signal processing instruction set extensions in ARM architecture v5TE.

Piccolo

The Piccolo coprocessor is a sophisticated 16-bit signal processing engine that uses the ARM coprocessor interface to cooperate with the ARM core in the transfer of operands and results from and to memory, but also executes its own instruction set.

The organization of Piccolo is illustrated in Figure 8.20 on page 241. Operands are loaded and results stored via the ARM coprocessor interface, so suitable addresses must be generated by the ARM core. The input and output buffers allow these transfers to move many 16-bit values in a single instruction, and values are transferred in pairs, making full use of the ARM's 32-bit bus width. The input buffer stores values until they are called upon by the signal processing code, and they may be accessed out of order from the buffer.

The Piccolo register set holds operands that may be 16 bits, 32 bits or 48 bits wide. The four 48-bit registers are used to accumulate results such as inner products without risk of overflow. The processing logic can compute a 16x16 product and add the result to one of these 48-bit accumulator registers in a single cycle. It also offers good support for fixed point operations and supports saturating arithmetic.

The signal processing operations that use values held in the register file are specified in a separate instruction set which Piccolo loads from memory via the AMBA bus into a local instruction cache.

An objective of the Piccolo architecture is to provide sufficient local storage, in the form of registers, the instruction cache and the input and output buffers, that a single AMBA bus can support good throughput. This contrasts with conventional signal processor designs that use two independent data memories and a separate instruction memory. As a result, Piccolo offers a much more straightforward programming model than does a system that combines an ARM core with a conventional signal processing core. However, some care is still required to synchronize the ARM code with the Piccolo code, and in an intensive signal processing application the ARM core is kept quite busy with the operand and result transfers, and is therefore unable to carry out extensive control functions at the same time. An even more straightforward programming model is offered by the ARM architecture v5TE instruction set extensions.

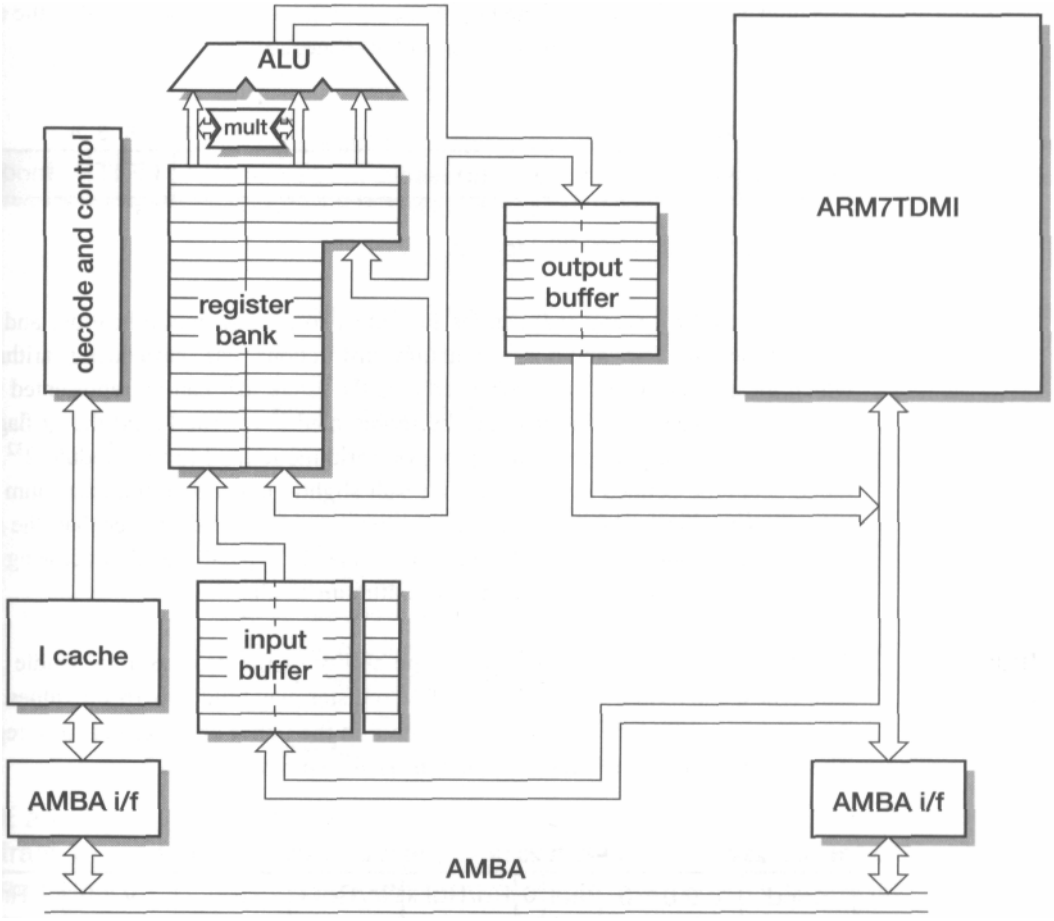


Figure 8.20 Piccolo organization.

V5TE signal processing instructions

The signal processing instruction set extension defined by ARM architecture v5TE, first implemented on the ARM9E-S synthesizable core, represents a very different approach to the problem from that used in the design of Piccolo. All that is used here is a carefully chosen addition to the native ARM instruction set to provide much better intrinsic support for the data types used in signal processing applications.

The ARM programmers' model is extended in architecture v5TE as shown in Figure 8.21 on page 242. The 'Q' flag is added in bit 27 of the CPSR, and all SPSRs also have a Q flag. Q is a 'sticky' overflow flag which may be set by certain v5TE instructions, and then reset by a suitable MSR instruction (see Section 5.14 on page 133). The term 'sticky' describes the fact that once set, the Q flag remains set until explicitly reset by an MSR instruction, so a series of instructions may be

executed and the Q flag inspected only once (using an MRS instruction) at the end to see if an overflow occurred at any point in the sequence.

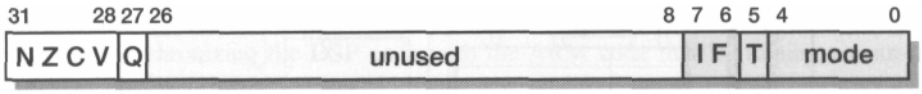


Figure 8.21 ARM v5TE PSR format.

The signal processing instructions fall into two groups: multiplication, and addition/subtraction. The addition/subtraction instructions use **saturating** arithmetic, which means that when the result overflows the range that can be represented in the data type the nearest value that can be represented is returned (and the Q flag set). This is in contrast to conventional processor arithmetic (and to the modulo 2^{32} arithmetic data type defined by C), where a result slightly larger than the maximum value has a value close to the *minimum* value. Here a result slightly larger than the maximum value simply returns the maximum value. In typical signal processing algorithms this minimizes the error and gives optimum results.

v5TE multiply instructions

The multiply instructions all improve the ability of the processor to handle 16-bit data types, and assume that a 32-bit ARM register may hold two 16-bit values. They therefore give efficient access to values held in the upper or lower half of a register. The binary encoding of these instructions is shown in Figure 8.22.

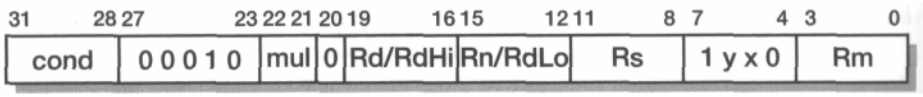


Figure 8.22 Architecture v5TE multiply instruction binary encoding.

The instructions supported by this format are:

```
SMLAxy{cond}    Rd,Rm,Rs,Rn        ;   mul   =   00
```

This instruction computes the 16x16 product of the two signed 16-bit values from the lower (x = 0) or upper (x = 1) half of Rm and the lower (y = 0) or upper (y = 1) half of Rs. The 32-bit product is added to the 32-bit value in Rn, and the result placed in Rd. The assembly format replaces x and y with 'B' for the lower (bottom) halfword or 'T' for the upper (top) halfword.

```
SMLAWy{cond} Rd,Rm,Rs,Rn      ;  mul   =0 1 ,
                                 x  =   0
```

This instruction computes the 32 x 16 product of the 32-bit value in Rm and the 16-bit value in the lower (y = 0) or upper (y = 1) half of Rs. The most signif-

icant 32 bits of the 48-bit product are added to the 32-bit value in Rn, and the result placed in Rd.

```
SMULWy{cond}    Rd,Rm,Rs        ;   mul   =   01,   x   =   1,
Rn   =   0
```

This instruction computes the 32 x 16 product of the 32-bit value in Rm and the 16-bit value in the lower (y = 0) or upper (y = 1) half of Rs. The most significant 32 bits of the 48-bit product are placed in Rd.

```
SMLALxy{cond} RdLo,RdHi,Rm,Rs; mul = 10
```

This instruction computes the 16x16 product of the two signed 16-bit values from the lower (x = 0) or upper (x = 1) half of Rm and the lower (y = 0) or upper (y = 1) half of Rs. The 32-bit product is added to the 64-bit value in RdHi:RdLo, and the result placed in RdHi:RdLo.

```
SMULxy{cond}    Rd,Rm,Rs        ;   mul   ••••   11,   Rn   =   0
```

This instruction computes the 16 x 16 product of the two signed 16-bit values from the lower (x = 0) or upper (x = 1) half of Rm and the lower (y = 0) or upper (y = 1) half of Rs. The 32-bit product is placed in Rd.

In all the above instructions the CPSR flags N, Z, C and V are *not* affected by the instruction, and the PC (r15) should not be used for any of the operand or result registers. If the addition in the accumulation (in SMLA and SMLAW) overflows the Q bit in the CPSR is set, but the addition uses conventional modulo 2^{32} rather than saturating arithmetic.

V5TE add/ SUbtract instructions

The other group of instructions in the architecture v5TE extensions are 32-bit addition and subtraction instructions that use saturating arithmetic. In each case there is an additional instruction which also doubles one of the operands before performing the addition and subtraction, which improves the efficiency of certain signal processing algorithms. The binary encoding of these instructions is shown in Figure 8.23.

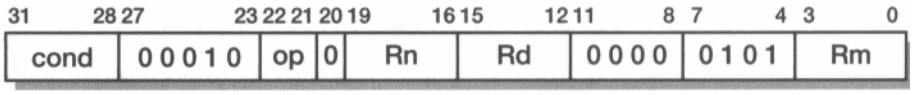


Figure 8.23 Architecture v5TE add/subtract instruction binary encoding.

The instructions supported by this format are:

```
QADD{cond}    Rd,Rm,Rn        ;   op   =   00
```

This instruction performs a 32-bit saturating addition of Rm and Rn, placing the result in Rd.

```
QSUB{cond}          ;       op
Rd,Rm,Rn            =    01
```

This instruction performs a 32-bit saturating subtraction of Rn from Rm, placing the result in Rd.

```
QDADD{cond}         ;    op    =
Rd,Rm,Rn            10
```

This instruction doubles Rn (with saturation) and then performs a 32-bit saturating addition of the result and Rm, placing the result in Rd.

```
QDSUB{cond) Rd,Rm,Rn          ;       op
                              =    11
```

This instruction doubles Rn (with saturation) and then performs a 32-bit saturating subtraction of the result from Rm, placing the result in Rd.

Again, in all the above instructions the CPSR flags N, Z, C and V are *not* affected by the instruction, and the PC (r!5) should not be used for any of the operand or result registers. If the saturating addition or subtraction overflows, or doubling Rn (where specified) causes overflow, the Q bit in the CPSR is set.

v5TE code example

To illustrate the use of these instructions, consider the problem of generating the inner ('dot') product of two vectors of 16-bit signed numbers held memory on an ARM9E-S core, which supports the architecture v5TE extensions, and an ARM9TDMI core, which does not. Computing an inner product is a very common procedure in signal processing applications. To minimize errors, saturating arithmetic should be used. The v5TE code for the central loop is as follows:

```
loop    SMULBB  r3,rl,r2 r4,  ; 16x16 multiply
        SUBS    r4,r2         ; decrement loop counter
        QDADD   r5,r5,r3      ; saturating x2 & accumulate
        SMULTT  r3,rl,r2      ; 16x16 multiply
        LDR     rl,[r6],#4    ; get next two multipliers
        QDADD   r5,r5,r3      ; saturating x2 & accumulate
        LDR     r2,[r7],#4    ; get next two multiplicands
        BNE     loop
```

This code example illustrates several important points:

The instructions are 'scheduled' to avoid pipeline stalls. On an ARM9E-S this means that the result of a load or 16-bit multiply should not be used in the following cycle.

Although the operands are 16-bit halfwords, they are loaded in pairs as 32-bit words. This is a more efficient way to use ARM's 32-bit memory interface than using halfword loads, and the v5TE multiply instructions can access the individual 16-bit operands directly from the registers.

The saturating 'double and accumulate' instructions are used to scale the product before accumulation. This is useful because the fixed point arithmetic used in

signal processing generally assumes operands in the range -1 to +1 but certain algorithms need coefficients greater than 1. The doubling operation gives an effective range from —2 to +2, which is sufficient for most algorithms.

Performance comparison

The single-cycle 32 x 16 multiplier on the ARM9E-S enables it to complete the above loop in 10 clock cycles, of which 4 are loop overhead (decrementing the loop counter and branching back at the end of the loop). Each loop computes two products, so each product requires 5 cycles. With loop unrolling (replicating the code to compute many more products in a single loop iteration) the cost for each product reduces towards 3 cycles. The best an ARM9TDMI can achieve is one product in 10 cycles, over three times slower. The difference is accounted for partly by the slower multiplier on the ARM9TDMI, partly by its less efficient handling of 16-bit operands, and the remainder by the extra instructions required to test and correct for saturation.

8.10 Example and exercises

Example 8.1 **Estimate the proportion of the number of test vectors required to test an ARM core via the JTAG and AMBA interfaces.**

An ARM core has in the region of 100 interface connections (32 data, 32 address, control, clock, bus, mode, and so on). The JTAG interface is serial. If the tester allows one vector to specify a pulse on *TCK* it will take 100 vectors to apply a parallel pattern to the ARM core. The AMBA test interface accesses the ARM periphery in five sections (see 'Test interface' on page 219), requiring five vectors on a standard tester.

The JTAG interface therefore appears to require 20 times the number of vectors. (Remember that JTAG is intended for PCB testing, not production VLSI testing.) In fact both the JTAG-based EmbeddedlCE and AMBA interfaces include optimizations to improve the efficiency of getting instructions into the ARM core, which is the dominant requirement in testing it, but a very detailed analysis would be required to take these into account in the estimate.

Exercise 8.1.1 Summarize the problem areas in the production VLSI testing of complex macrocell-based system chips and discuss the relative merits of the various solutions.

Exercise 8.1.2 Describe and differentiate between production VLSI testing, printed circuit board testing and system debugging, and describe how a JTAG test port may be used to address each of these. Where is the JTAG approach most effective and where is it least effective?

Exercise 8.1.3 What problem does the Advanced Microprocessor Bus Architecture address and what problem does the ARM reference peripheral specification address? How might they be related?

Exercise 8.1.4 Sketch a system development plan for an embedded system chip showing at which stage the ARMulator, AMBA, the reference peripheral specification, Embed-dedlCE and JTAG are (i) designed into the chip, and/or (ii) used to assist in the development process.

9 ARM Processor Cores

Summary of chapter contents

An ARM processor core is the engine within a system that fetches ARM (and possibly Thumb) instructions from memory and executes them. ARM cores are very small, typically occupying just a few square millimetres of chip area. Modern VLSI technology allows a large number of additional system components to be incorporated on the same chip. These may be closely related to the processor core, such as cache memory and memory management hardware, or they may be unrelated system components such as signal processing hardware. They may even include further ARM processor cores. Among all of these components the processor cores stand out as being the most densely complex components which place the greatest demand on software development and debugging tools. The correct choice of processor core is one of the most critical decisions in the specification of a new system.

In this chapter the principal current ARM processor core products are described. They offer a choice of cost, complexity and performance points from which the most effective solution can be selected.

Many applications require the processor core to be supported by closely coupled cache and memory management subsystems. A number of standard configurations combining these components are described in Chapter 12, 'ARM CPU Cores', on page 317.

Further ARM-compatible processor cores are described in Chapter 14, The AMULET Asynchronous ARM Processors', on page 374. These processor cores are research prototypes and not yet commercial products.