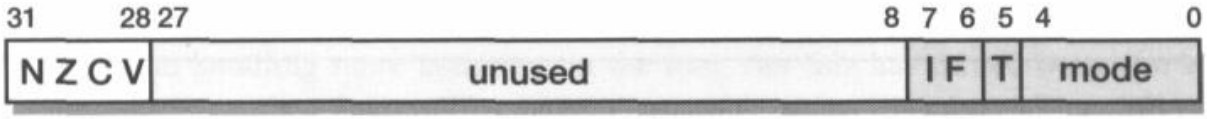
Module-5

Thumb Instruction Set: The Thumb bit in the CPSR, The Thumb programmer's model ,Thumb branch instructions, Thumb software interrupt instruction , Thumb data processing instructions , Thumb single register data transfer instructions, Thumb multiple register data transfer instructions, Thumb breakpoint instruction, Thumb implementation ,Thumb applications . Architectural Support for System Development: The ARM memory interface, The Advanced Microcontroller Bus Architecture (AMBA),The ARM reference peripheral specification, Hardware system prototyping tools, The ARMulator.

Thumb Instruction Set

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. 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.



Thumb entry

The normal way to switch to execute Thumb instructions is by executing a Branch and Exchange instruction (BX). 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 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.

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 below. 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.

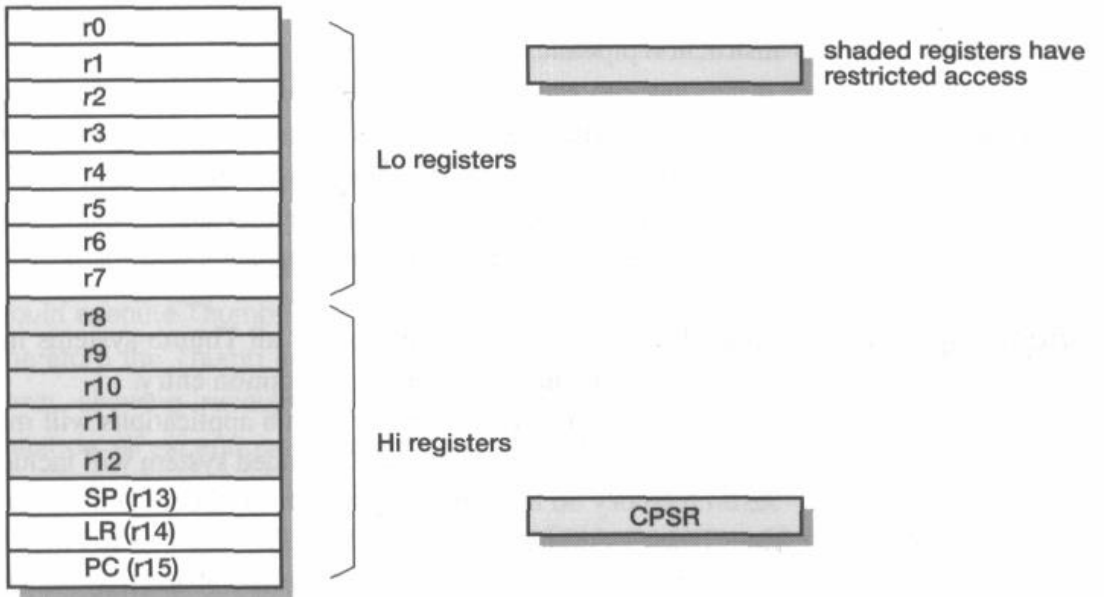


Figure 7.1 Thumb accessible registers.

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:
• 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
• 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. 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.

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.
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.

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).

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.
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.

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.

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>
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 instructtion).
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 condition 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.

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).

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
4:    <op>Rd, [PC, #<#off8>]
5:    <op>Rd, [SP, #<#off8>]    ; <op> = LDRISTR
```

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

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.

The block copy forms of the instruction use the LDMIA and STMIA addressing modes. 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.
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}}

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.

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

Assembler format BKPT

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

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 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 to the corresponding ARM 'ADDS Rd,Rd, #imm8' instruction is shown in Figure below. Note that:

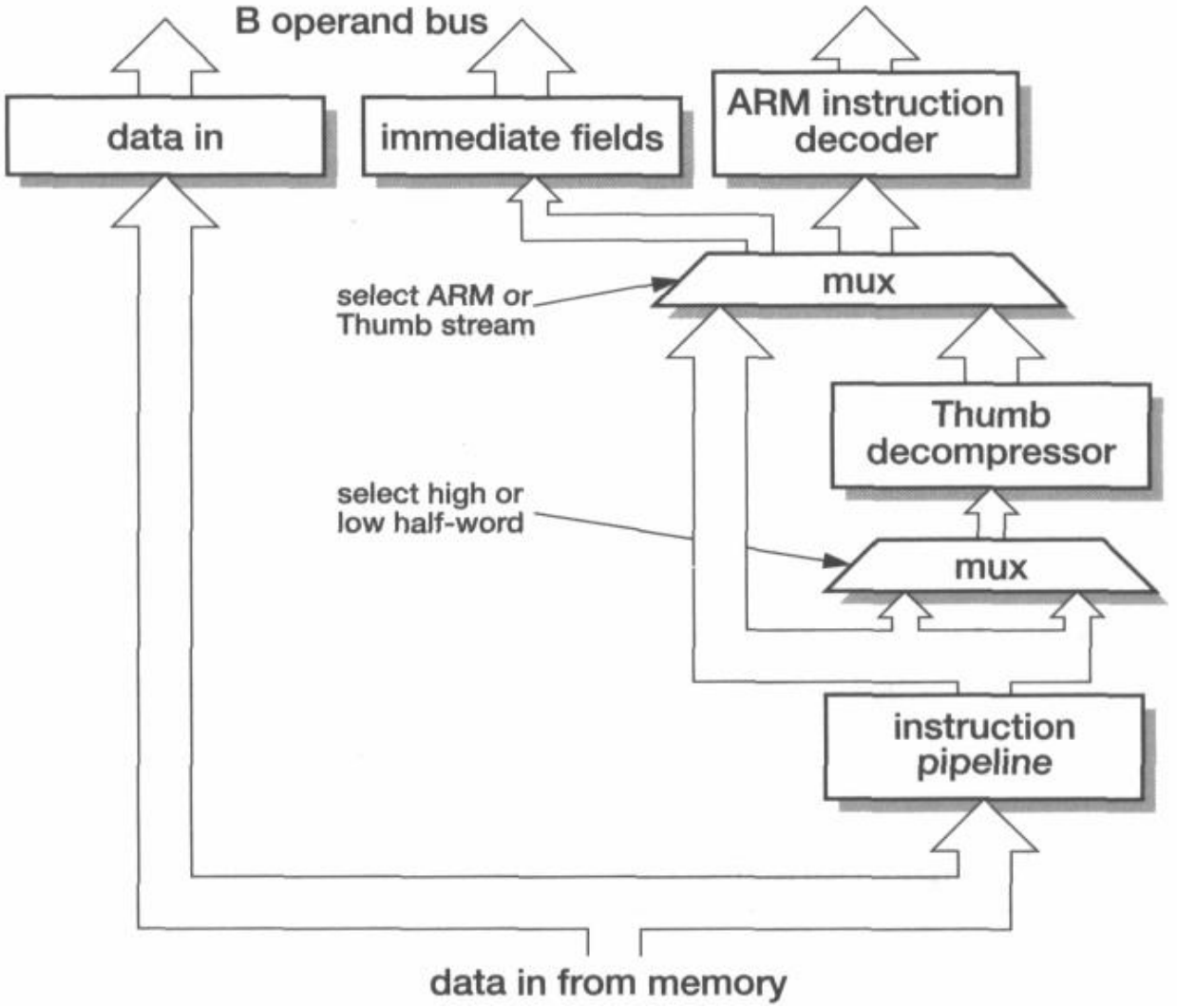


Figure 7.8 The Thumb instruction decompressor organization.

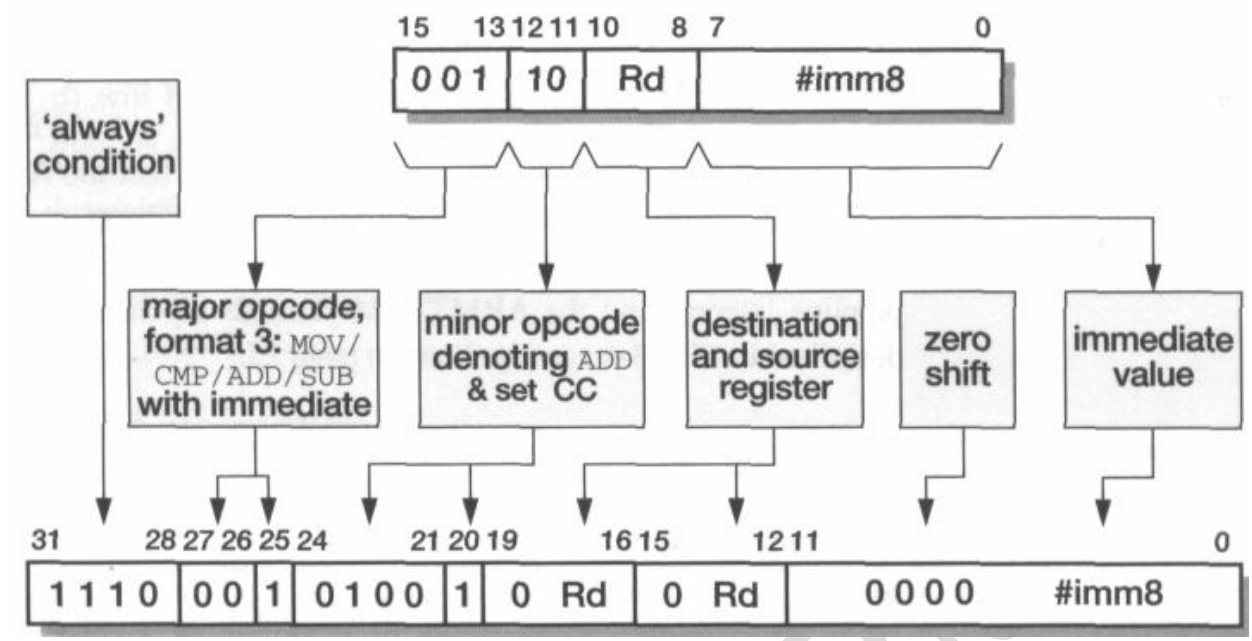


Figure 7.9 Thumb to ARM instruction mapping.

• 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. The simplicity of the decompression logic is crucial to the efficiency of the Thumb instruction set.

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:

• 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.
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.

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
        CODE32              ;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 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.

• 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 appropriately. 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.

Architectural support for System Development)

The ARM memory interface

ARM bus signals

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.

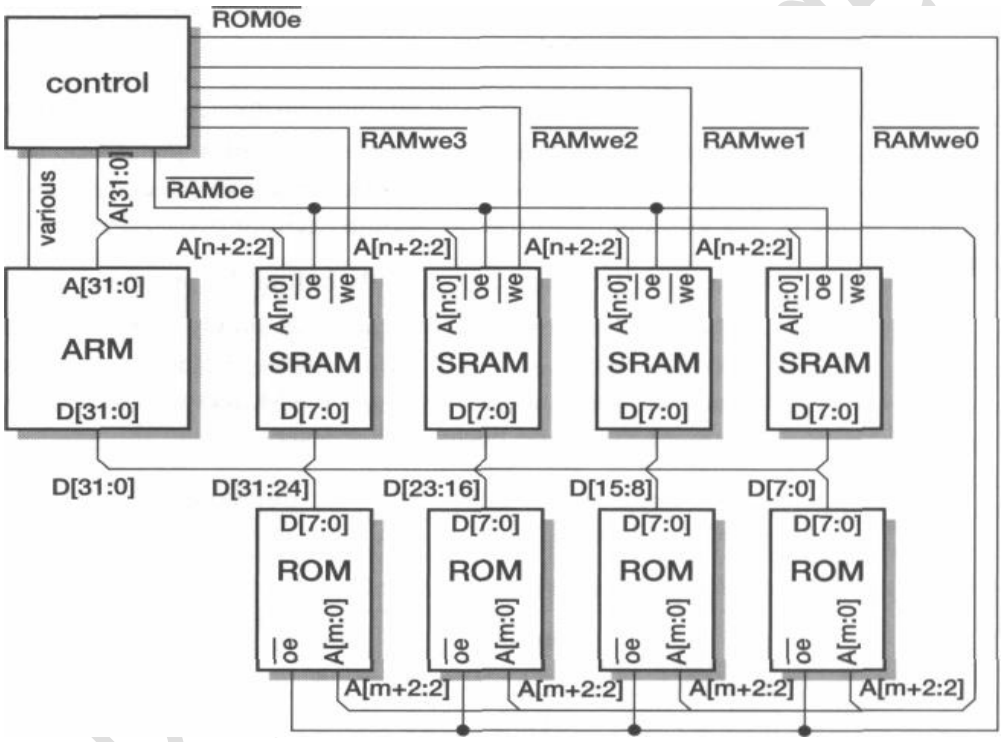
Simple memory interface

• Signals that specify whether the memory is needed *(mreq)* and whether the address is sequential *(seq);*

• Signals that specify the direction (r/w) and size *(b/w byte write* 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]).*

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 used depending on the size of the memory part.



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.

• 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 logic required for the above functions is quite straightforward and can be implemented using a single program-mable logic device.

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.

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.

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. Like most memory devices, the storage cells in a DRAM are arranged in a matrix which is approximately square. 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 **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.

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 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.

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.

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 found 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 below. 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.

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:

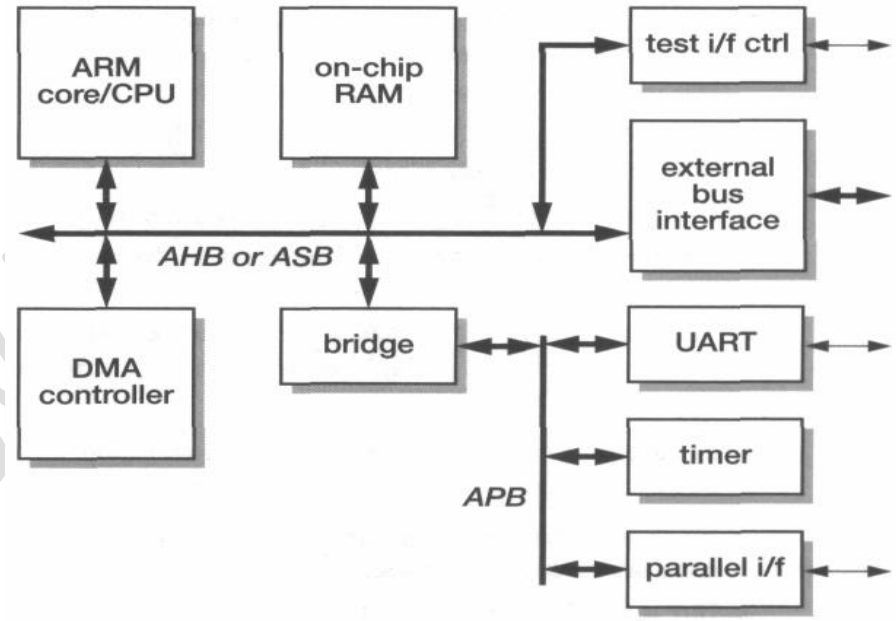


Figure 8.11 A typical AMBA-based system.

• 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 systems with modest address-space requirements, and in a multiplexed implementation 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 readmodify-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 complete 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. T*he AMBA approach will reduce test cost due its parallel tester interface.

Advanced High 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.

The AHB is intended to replace the ASB in very high performance systems, for example those based on the ARM1020E.

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 .
• It uses a centrally multiplexed bus scheme rather than a bidirectional bus with tristate drivers.
• 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.

The ARM reference peripheral specification

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. 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.**

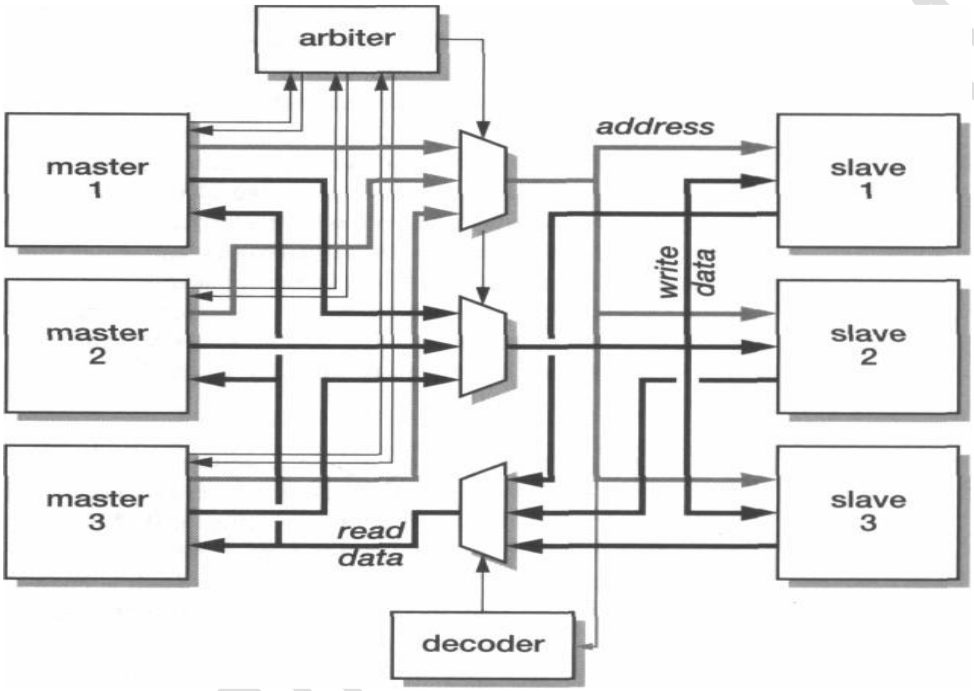


Figure 8.12 AHB multiplexed bus scheme.

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 various registers from these base addresses.
• An interrupt controller with a defined set of functions, including a defined interrupt 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.

Hardware system prototyping tools

The number of gates on a chip continues to grow at an exponential rate. 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 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 (usually 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 shown in figure below.

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.

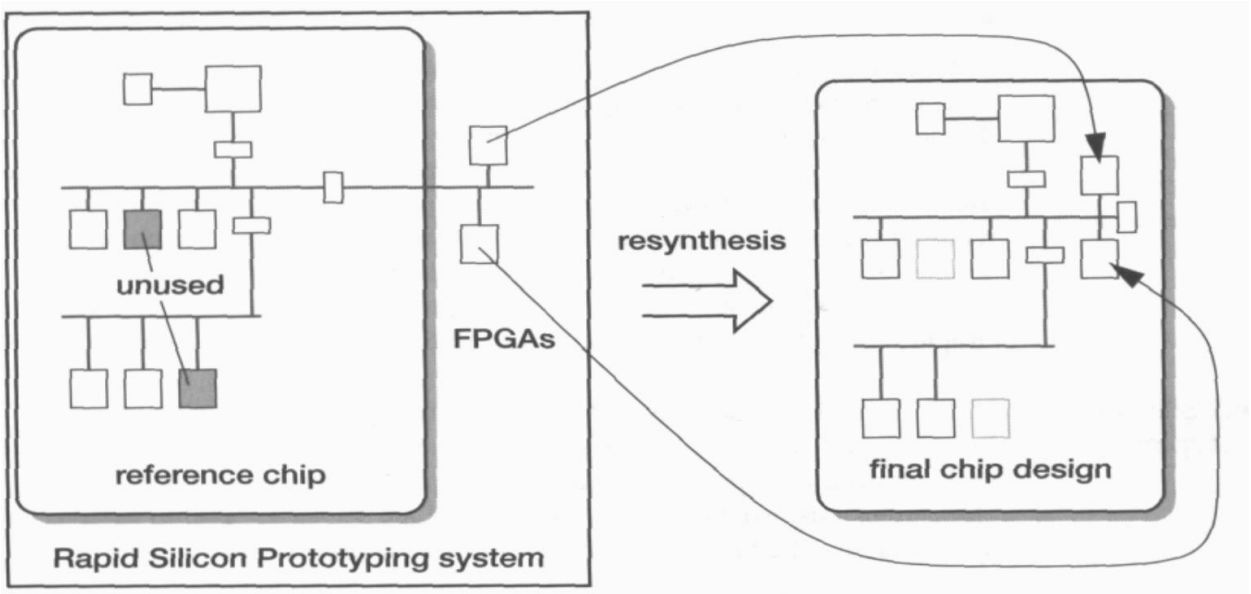
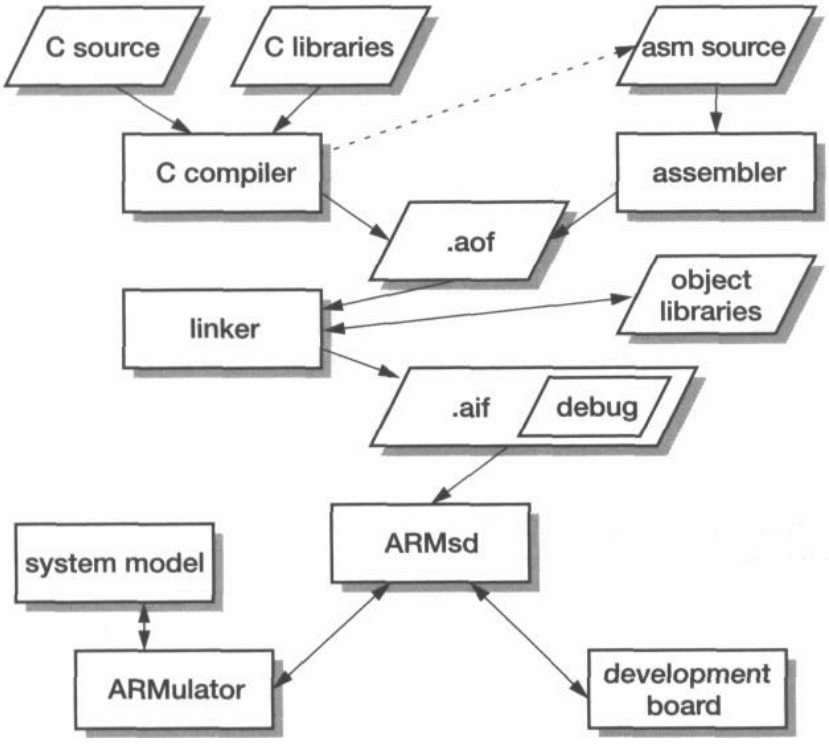


Figure 8.13 Rapid Silicon Prototyping principle

The ARMulator



The ARMulator is part of the cross-development toolkit. 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 Using the ARMulator it is possible to build a complete, clock-cycle accurate modelling software model of a system including a cache, MMU, physical memory, peripheral devices, operating system and software.

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**. 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.

The lower-level models are synthesized automatically maintaining consistency between the models.