ARM Assembly Language Programming: Data processing instructions, Data transfer instructions, Control flow instructions, writing simple assembly language programs. ARM Organization and Implementation: Pipeline, Types, 3-stage pipeline ARM organization, 5-stage pipeline ARM organization, ARM instruction execution, ARM implementation, The ARM coprocessor interface.

ARM Assembly Language Programming

Data processing instructions

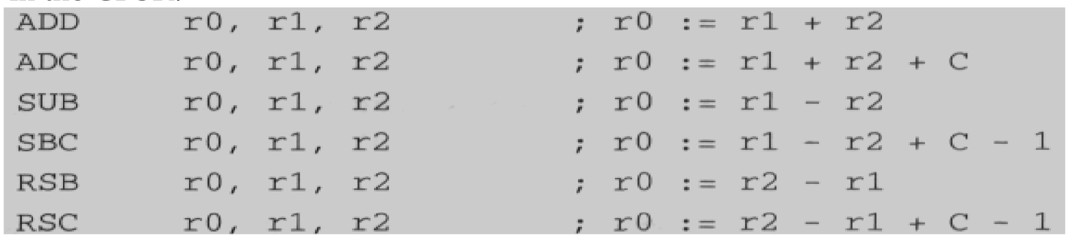
ARM data processing instructions enable the programmer to perform arithmetic and logical operations on data values in registers. So the data processing instructions are the only instructions which modify data values. These instructions typically require two operands and produce a single result, though there are exceptions to both of these rules. A characteristic operation is to add two values together to produce a single result which is the sum.
Here are some rules which apply to ARM data processing instructions:
• All operands are 32 bits wide and come from registers or are specified as literals in the instruction itself.
• The result, if there is one, is 32 bits wide and is placed in a register.
• Each of the operand registers and the result register are independently specified in the instruction.

Simple register operands

```
ADD    r0, r1,r2          ; r0 : = r1 + r2
```

The semicolon in this line indicates that everything to the right of it is a comment and should be ignored by the assembler. Comments are put into the assembly source code to make reading and understanding it easier.
Note that in writing the assembly language source code, care must be taken to write the operands in the correct order, which is result register first, then the first operand and lastly the second operand. When this instruction is executed the only change to the system state is the value of the destination register r0.

• Arithmetic operations.
These instructions perform binary arithmetic (addition, subtraction and reverse subtraction, which is subtraction with the operand order reversed) on two 32-bit operands. The operands may be unsigned or 2's-complement signed integers; the carry-in, when used, is the current value of the C bit in the CPSR.



```
ADD     r0, r1, r2        ; r0 := r1 + r2
ADC     r0, r1, r2        ; r0 := r1 + r2 + C
SUB     r0, r1, r2        ; r0 := r1 - r2
SBC     r0, r1, r2        ; r0 := r1 - r2 + C - 1
RSB     r0, r1, r2        ; r0 := r2 - r1
RSC     r0, r1, r2        ; r0 := r2 - r1 + C - 1
```

'ADD' is simple addition, 'ADC' is add with carry, 'SUB' is subtract, 'SBC' is subtract with carry, 'RSB' is reverse subtraction and 'RSC' reverse subtract with carry.

• Bit-wise logical operations.
These instructions perform the specified Boolean logic operation on each bit pair of the input operands, so in the first case *r0[i]:= r1[i]* AND *r2[i]* for each value of *i* from 0 to 31 inclusive, where *r0[i]* is the *i*th bit of r0.

```
AND     r0, r1, r2          ; r0 := r1 and r2
ORR     r0, r1, r2          ; r0 := r1 or r2
EOR     r0, r1, r2          ; r0 := r1 xor r2
BIC     r0, r1, r2          ; r0 := r1 and not r2
```

The final mnemonic, BIC, stands for 'bit clear'.

• Register movement operations.
These instructions ignore the first operand, which is omitted from the assembly language format, and simply move the second operand to the destination.

```
MOV     r0, r2              ; r0 := r2
MVN     r0, r2              ; r0 := not r2
```

The 'MVN' mnemonic stands for 'move negated'; it leaves the result register set to the value obtained by inverting every bit in the source operand.

• Comparison operations.
These instructions do not produce a result (which is therefore omitted from the assembly language format) but just set the condition code bits (N, Z, C and V) in the CPSR according to the selected operation.

```
CMP    r1,r2     ; set cc on r1 - r2

CMN    r1,r2     ; set cc on r1+ r2


TST             r1, r2   ; set cc on r1 and r2

TEQ             r1,r2    ; set cc on r1 xor r2
```

The mnemonics stand for 'compare' (CMP), 'compare negated' (CMN), '(bit) test' (TST) and 'test equal' (TEQ).

Several Possibilities in Data Processing Instructions:
Immediate operands
If, instead of adding two registers, we simply wish to add a constant to a register we can replace the second source operand with an immediate value, which is a literal constant, preceded by '#':

```
ADD  r3,  r3, #1          ; r3 := r3+ 1

ADD  r8,  r7, #&ff        ; r8 :=r7+255
```

The first example also illustrates that although the 3-address format allows source and destination operands to be specified separately, they are not required to be distinct registers. The second example shows that the immediate value may be specified in hexadecimal (base 16) notation by putting '&' after the '#'.

Shifted register operands

A third way to specify a data operation is similar to the first, but allows the second register operand to be subjected to a shift operation before it is combined with the first operand. For example:

```
        ADD r3, r2, r1, LSL #3      ; r3 := r2 + 8 x r1
```

Note that this is still a single ARM instruction, executed in a single clock cycle.
Most processors offer shift operations as separate instructions, but the ARM combines them with a general ALU operation in a single instruction.
Here 'LSL' indicates 'logical shift left by the specified number of bits', which in this example is 3. Any number from 0 to 31 may be specified, though using 0 is equivalent to omitting the shift altogether. As before, '#' indicates an immediate quantity.
The available shift operations are:

• LSL: logical shift left by 0 to 31 places; fill the vacated bits at the least significant end of the word with zeros.

• LSR: logical shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros.

• ASL: arithmetic shift left; this is a synonym for LSL.

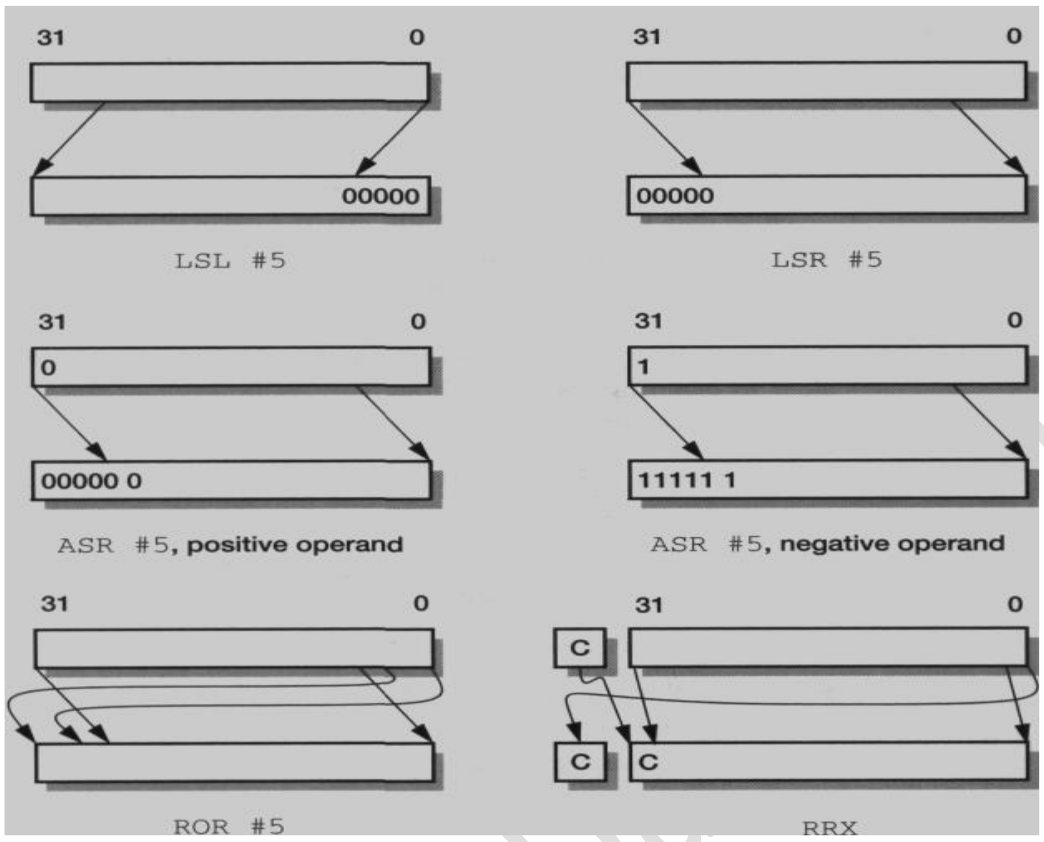
• ASR: arithmetic shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros if the source operand was positive, or with ones if the source operand was negative.

• ROR: rotate right by 0 to 32 places; the bits which fall off the least significant end of the word are used, in order, to fill the vacated bits at the most significant end of the word.

• RRX: rotate right extended by 1 place; the vacated bit (bit 31) is filled with the old value of the C flag and the operand is shifted one place to the right. With appropriate use of the condition codes (see below) a 33-bit rotate of the operand and the C flag is performed.

It is also possible to use a register value to specify the number of bits the second operand should be shifted by:

ADD r5, r5, r3, LSL r2 ; r5 : = r5 + r3 x 2^{r2}



Setting the condition codes

Any data processing instruction can set the condition codes (N, Z, C and V) if the programmer wishes it to. The comparison operations only set the condition codes, so there is no option with them, but for all other data processing instructions a specific request must be made. At the assembly language level this request is indicated by adding an 's' to the opcode, standing for 'Set condition codes'. As an example, the following code performs a 64-bit addition of two numbers held in r0-r1 and r2-r3, using the C condition code flag to store the intermediate carry:

```
ADDS     r2, r2, r0 ; 32-bit carry out -> C..
ADC      r3, r3, r1 ; .. and added into high word
```

Multiplies

A special form of the data processing instruction supports multiplication:

```
MUL r4, r3, r2         ; r4 := (r3 x r2)
```

There are some important differences from the other arithmetic instructions:

• Immediate second operands are not supported.
• The result register must not be the same as the first source register.
• If the ' s' bit is set the V flag is preserved (as for a logical instruction) and the C flag is rendered meaningless.

Multiplying two 32-bit integers gives a 64-bit result, the least significant 32 bits of which are placed in the result register and the rest are ignored. An alternative form, subject to the same restrictions, adds the product to a running total. This is the **multiply-accumulate** instruction:

```
MLA r4, r3, r2, r1     ; r4 := (r3 x r2 + r1)
```

Multiplication by a constant can be implemented by loading the constant into a register and then using one of these instructions, but it is usually more efficient to use a short series of data processing instructions using shifts and adds or subtracts.

Data transfer instructions

Data transfer instructions move data between ARM registers and memory. There are three basic forms of data transfer instruction in the ARM instruction set:

• **Single register load and store instructions.**
These instructions provide the most flexible way to transfer single data items between an ARM register and memory. The data item may be a byte, a 32-bit word, or a 16-bit half-word.

• Multiple register load and store instructions.
These instructions are less flexible than single register transfer instructions, but enable large quantities of data to be transferred more efficiently. They are used for procedure entry and exit, to save and restore workspace registers, and to copy blocks of data around memory.

• Single register swap instructions.
These instructions allow a value in a register to be exchanged with a value in memory, effectively doing both a load and a store operation in one instruction. They are little used in user-level programs. Their principal use is to implement semaphores to ensure mutual exclusion on accesses to shared data structures in multi-processor systems.

The ARM data transfer instructions are all based around **register-indirect addressing**, with modes that include **base-plus-offset and base-plus-index** addressing.

Single register load and store instructions

These instructions compute an address for the transfer using a base register, which should contain an address near to the target address, and an offset which may be another register or an immediate value. We have just seen the simplest form of these instructions, which does not use an offset:

```
LDR    r0, [r1]        ; r0 := mem32 [r1]

STR    r0, [r1]        ;mem32[r1] := r0
```

Initializing an address pointer

To load or store from or to a particular memory location, an ARM register must be initialized to contain the address of that location, or, in the case of single register transfer instructions, an address within 4 Kbytes of that location .

As an example, consider a program which must copy data from TABLE1 to TABLE2, both of which are near to the code:

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
```

Here we have introduced **labels** (COPY, TABLE1 and TABLE2) which are simply names given to particular points in the assembly code. The first ADR pseudo instruction causes r1 to contain the address of the data that follows TABLE1; the second ADR likewise causes r2 to hold the address of the memory starting at TABLE2.

We can now copy the first word from one table to the other:

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
        LDR     r0, [r1]            ; load first value...
        STR     r0, [r2]            ; and store it in TABLE2
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
        ..
```

We could now use data processing instructions to modify both base registers ready for the next transfer:

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
LOOP    LDR     r0, [r1]            ; get TABLE1 1st word
        STR     r0, [r2]            ; copy into TABLE2
        ADD     r1, r1, #4          ; step r1 on 1 word
        ADD     r2, r2, #4          ; step r2 on 1 word
        ???                         ; if more go back to LOOP
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
```

Note that the base registers are incremented by 4 (bytes), since this is the size of a word. If the base register was word-aligned before the increment, it will be word-aligned afterwards too.

Base plus offset addressing

If the base register does not contain exactly the right address, an offset of up to 4 Kbytes may be added (or subtracted) to the base to compute the transfer address:

```
LDR   r0,   [r1,#4]       ; r0 := mem32[r1+ 4]
```

This is a **pre-indexed** addressing mode. It allows one base register to be used to access a number of memory locations which are in the same area of memory. Sometimes it is useful to modify the base register to point to the transfer address. This can be achieved by using pre-indexed addressing with **auto-indexing,** and allows the program to walk through a table of values:

```
LDR   r0,   [r1,#4]!        ; r0 := mem32[r1+ 4]
                             ; r1 := r1+ 4
```

The exclamation mark indicates that the instruction should update the base register after initiating the data transfer.
Another useful form of the instruction, called **post-indexed** addressing, allows the base to be used without an offset as the transfer address, after which it is auto-indexed:

```
LDR   r0,   [r1], #4        ; r0 := mem32 [r1]
                            ; r1 := r1 + 4
```

Using the last of these forms we can now improve on the table copying program example introduced earlier:

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
LOOP    LDR     r0, [r1], #4        ; get TABLE1 1st word
        STR     r0, [r2], #4        ; copy into TABLE2
        ???                         ; if more go back to LOOP
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
```

The load and store instructions are repeated until the required number of values has been copied into TABLE2, then the loop is exited.

Multiple register data transfers

Where considerable quantities of data are to be transferred, it is preferable to move several registers at a time. These instructions allow any subset (or all) of the 16 registers to be transferred with a single instruction. A simple example of this instruction class is:

```
LDMIA   r1, {r0,r2,r5}      ; r0 := mem32[r1]
                            ; r2 := mem32[r1 + 4]
                            ; r5 := mem32[r1 + 8]
```

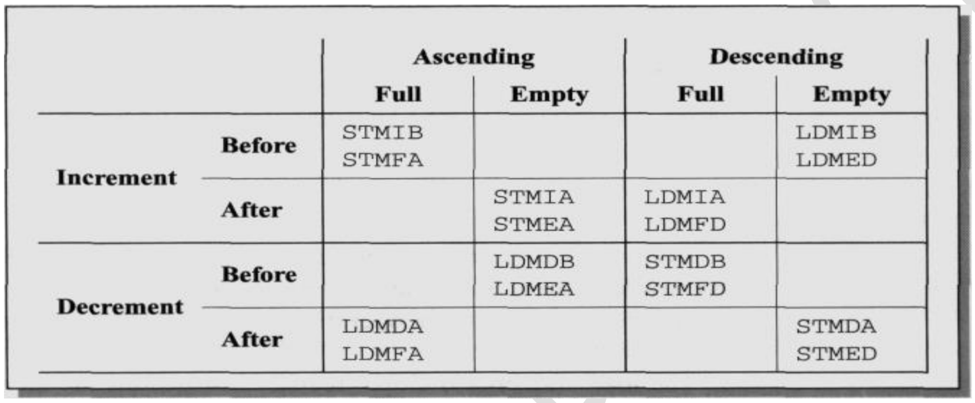
Stack addressing
A stack is a form of last-in-first-out store which supports simple dynamic memory allocation, that is, memory allocation where the address to be used to store a data value is not known at the time the program is compiled or assembled. A stack is usually implemented as a linear data structure which grows up (an **ascending** stack) or down (a **descending** stack) memory as data is added to it and shrinks back as data is removed. A **stack pointer** holds the address of the current top of the stack, either by pointing to the last valid data item pushed onto the stack (a **full** stack), or by pointing to the vacant slot where the next data item will be placed (an **empty** stack). The ARM multiple register transfer instructions support all four forms of stack:

• Full ascending: the stack grows up through increasing memory addresses and the base register points to the highest address containing a valid item.

• Empty ascending: the stack grows up through increasing memory addresses and the base register points to the first empty location above the stack.
• Full descending: the stack grows down through decreasing memory addresses and the base register points to the lowest address containing a valid item.
• Empty descending: the stack grows down through decreasing memory addresses and the base register points to the first empty location below the stack.

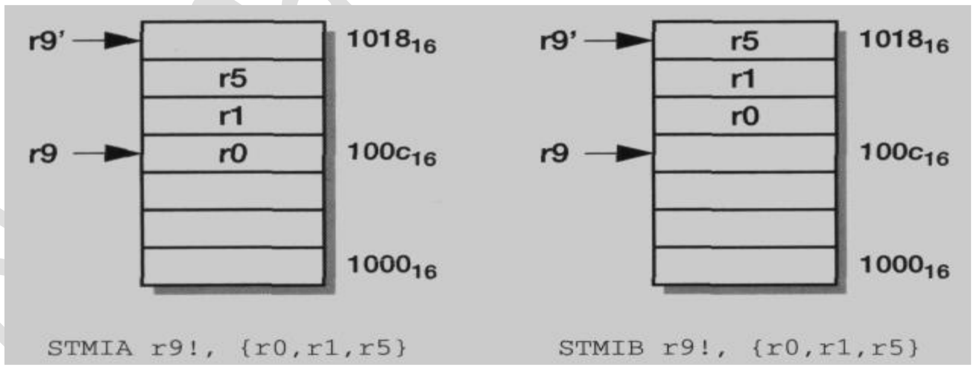
Block copy addressing

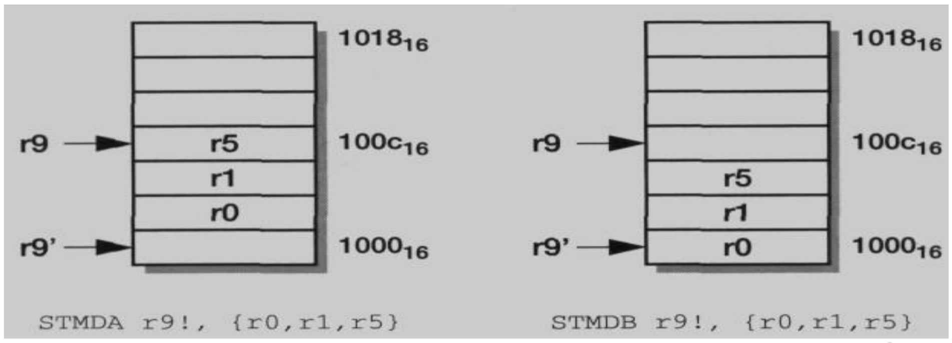
The block copy view is based on whether the data is to be stored above or below the address held in the base register and whether the address incrementing or decrementing begins before or after storing the first value. The mapping between the two views depends on whether the operation is a load or a store, and is detailed in Table below.



		Ascending		Descending	
		Full	**Empty**	**Full**	**Empty**
Increment	**Before**	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	**Before**		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED

Multiple register transfer addressing modes are shown below which shows how each variant stores 3 registers into memory and how the base register is modified if auto-indexing is enabled. The base register value before the instruction is r9 and after auto-indexing it is r9’.



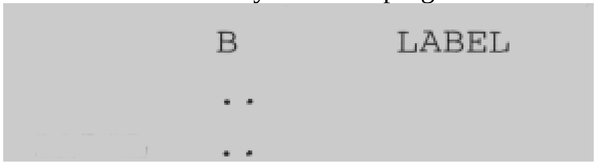


Control flow instructions

This third category of instructions neither processes data nor moves it around; it simply determines which instructions get executed next.

Branch instructions

The most common way to switch program execution from one place to another is use the branch instruction:



```
        B       LABEL
        ..
        ..
LABEL
```

The processor normally executes instructions sequentially, but when it reaches the branch instruction it proceeds directly to the instruction at LABEL instead of executing the instruction immediately after the branch. In this example LABEL comes after the branch instruction in the program, so the instructions in between are skipped.

Conditional branches

The mechanism used to control loop exit is conditional branching. Here the branch has a condition associated with it and it is only executed if the condition codes have the correct value. A typical loop control sequence might be:

```
        MOV  r0, #0          ; initialize counter
LOOP

        ADD   r0,  r0,  #1   ; increment loop counter


        CMP   BNE  r0,  #10  ; compare with limit


        LOOP                 ; repeat if not equal
                             ; else fall through
```

Branch conditions.

Branch	Interpretation	Normal uses
B BAL	Unconditional	Always
BEQ	Equal	Comparison equal or zero result

BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set Higher	Arithmetic operation gave carry-out
BHS	or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Conditional execution

An unusual feature of the ARM instruction set is that conditional execution applies not only to branches but to all ARM instructions. A branch which is used to skip a small number of following instructions may be omitted altogether by giving those instructions the opposite condition. For example, consider the following sequence:

```
        CMP     r0, #5
        BEQ     BYPASS              ; if (r0 != 5) {
        ADD     r1, r1, r0          ;    r1 := r1 + r0 - r2
        SUB     r1, r1, r2          ; }
BYPASS  ..
```

This may be replaced by:

```
CMP      r0, #5               ; if (r0 != 5) {
ADDNE    r1, r1, r0           ;    r1 := r1 + r0 - r2
SUBNE    r1, r1, r2           ; }
```

Conditional execution is invoked by adding the 2-letter condition after the 3-letter opcode

It is sometimes possible to write very compact code by the use of conditionals, for example:

```
; if ((a==b) && (c==d)) e++;

CMP     r0, r1

CMPEQ       r2, r3

ADDEQ       r4, r4, #1
```

Note how if the first comparison finds unequal operands the second is skipped, causing the increment to be skipped also. The logical 'and' in the if clause is implemented by making the second comparison conditional.

Branch and link instructions

A common requirement in a program is to be able to branch to a subroutine in a way which makes it possible to resume the original code sequence when the subroutine has completed. It performs a branch in exactly the same way as the branch instruction, also saves the address of the instruction following the branch in the link register,r14.

```
        BL      SUBR                ; branch to SUBR
        ..                          ; return to here
SUBR    ..                          ; subroutine entry point
        MOV     pc, r14             ; return
```

Subroutine return instructions
To get back to the calling routine, the value saved by the branch and link instruction in r14 must be copied back into the program counter.

```
SUB2
     MOV  pc,  r14    ; copy r14 into r15 to return
```

Supervisor calls
Whenever a program requires input or output, for instance to send some text to the display, it is normal to call a supervisor routine. The instruction set includes a special instruction, SWI, to call these functions, (SWI stands for 'Software Interrupt', but is usually pronounced 'Supervisor Call'.)

```
SWI    SWI_WriteC    ; output r0[7:0]
SWI    SWI_Exit      ; return to monitor
```

Jump tables
Jump tables are normally used by experienced programmers. The idea of a jump table is that a programmer sometimes wants to call one of a set of subroutines, the choice depending on a value computed by the program.

```
               BL      JUMPTAB


JUMPTAB        CMP    r0, #0
```

```
                BEQ    SUB0
                CMP     r0, #1
                BEQ    SUB1
SUB0
SUB1
```

Writing simple assembly language programs

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

• The declaration of the code 'AREA', with appropriate attributes.
•EQU - initialising constants – used here to define SWI number
•ENTRY - code entry point
• END - the end of the program source
• labels are aligned left – opcodes are indented
• The definitions of the system calls which will be used in the routine
• The use of the ADR pseudo instruction to get an address into a base register.
• The use of auto-indexed addressing to move through a list of bytes.
• Conditional execution of the SWI instruction to avoid an extra branch.
Note also the use of a zero byte to mark the end of the string. In order to run this program you will need the following tools, all of which are available within the ARM software development toolkit:
• A text editor to type the program into.
• An assembler to turn the program into ARM binary code.
• An ARM system or emulator to execute the binary on. The ARM system must have some text output capability.

ARM Organization and Implementation

Pipeline

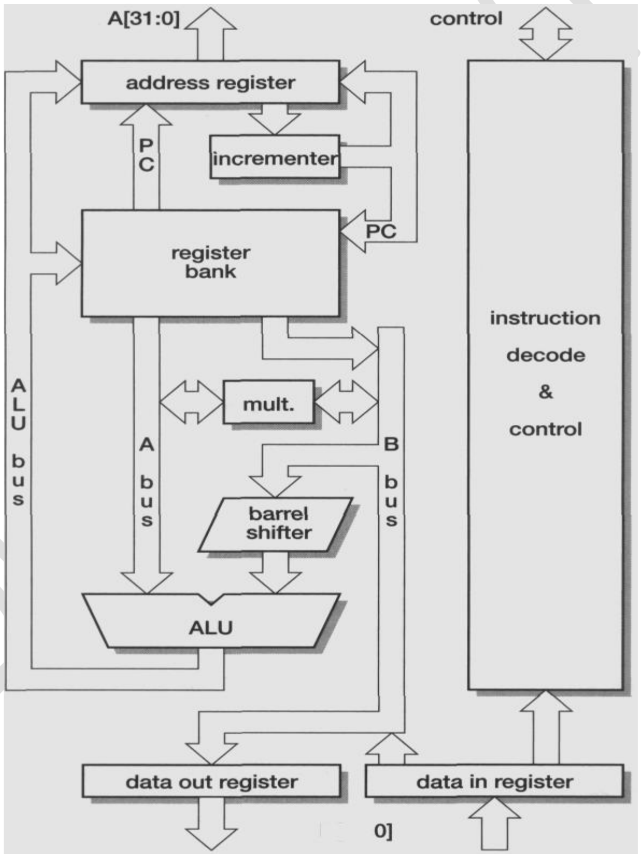
Pipelining attempts to keep every part of the processor busy with some instruction by dividing incoming instructions into a series of sequential steps performed by different processor units with different parts of instructions processed in parallel. It allows faster CPU throughput than would otherwise be possible at a given clock rate, but may increase latency due to the added overhead of the pipelining process itself.

Instruction pipelining is a technique for implementing instruction-level parallelism within a single processor.

Two types of pipelining occurs:

3-stage pipelining
5-stage pipelining

3-stage pipeline ARM organization



The principal components are:
• The **register bank**, which stores the processor state. It has two read ports and one write port which can each be used to access any register, plus an additional read port and an additional write port that give special access to r15, the program counter.

• The **barrel shifter**, which can shift or rotate one operand by any number of bits.

• The **ALU**, which performs the arithmetic and logic functions required by the instruction set.

• The **address register and incrementer**, which select and hold all memory addresses and generate sequential addresses when required.

• The **data registers**, which hold data passing to and from memory.

• The instruction decode and control is associated with instruction decoder and control logic.

In a single-cycle data processing instruction, two register operands are accessed, the value on the B bus is shifted and combined with the value on the A bus in the ALU, then the result is written back into the register bank. The program counter value is in the address register, from where it is fed into the incrementer, then the incremented value is copied back into rl5 in the register bank and also into the address register to be used as the address for the next instruction fetch.

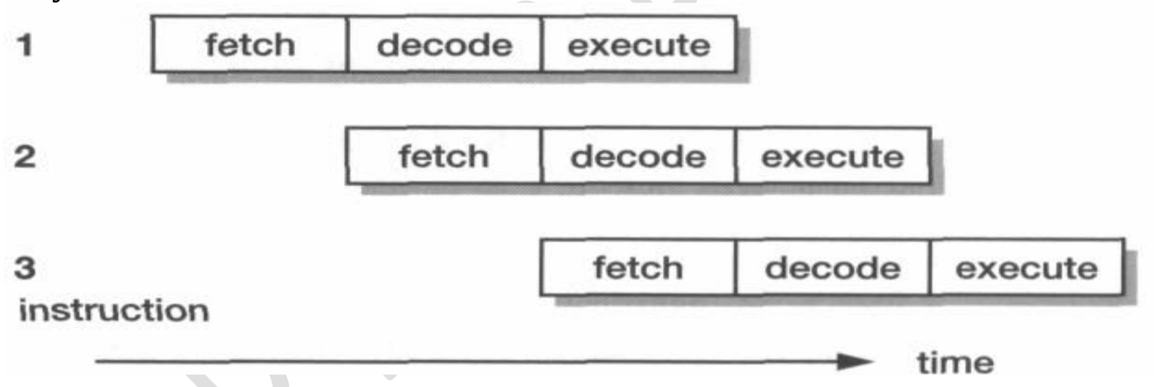
3-stage pipeline with the following pipeline stages:

• **Fetch**;the instruction is fetched from memory and placed in the instruction pipeline.
• **Decode:**the instruction is decoded and the datapath control signals prepared for the next cycle. In this stage the instruction 'owns' the decode logic but not the datapath.
• **Execute**:the instruction 'owns' the datapath; the register bank is read, an operand shifted, the ALU result generated and written back into a destination register.

3-stage pipeline with the following instructions:

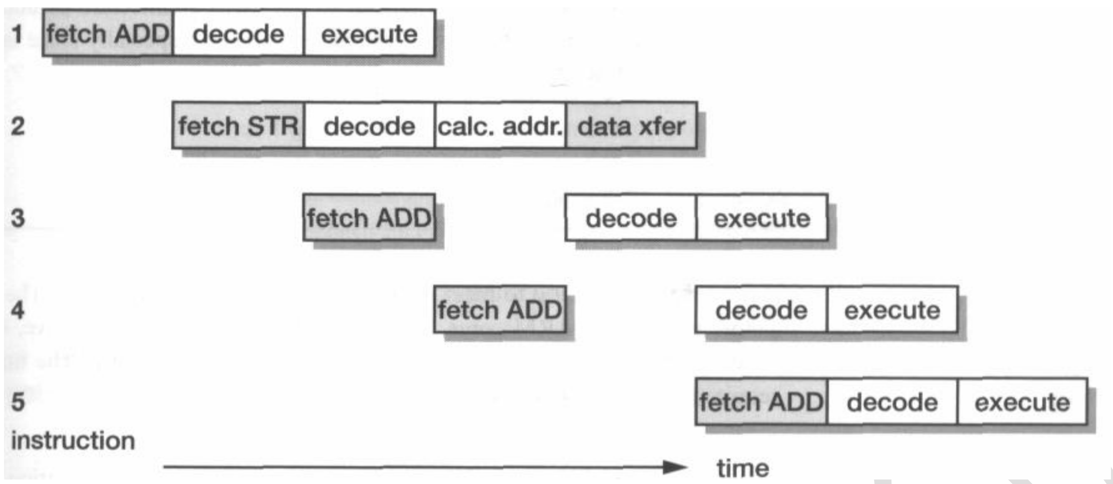
Single cycle instruction
Multi cycle instruction

Single cycle instruction



When the processor is executing single data processing instructions:

1. The pipeline enables one instruction to be completed every clock cycle.
2. An individual instruction takes three clock cycles to complete, so it has a three-cycle latency, but the throughput is one instruction per cycle.
3. These are simple and regular.



When the processor is executing multiple data processing instructions:

1. When a multi-cycle instruction is executed the flow is less regular, as shown above.
2. This shows a sequence of single-cycle ADD instructions with a data store instruction STR, occurring after the first ADD.
3. It can be seen that memory is used in every cycle.
4. The datapath is likewise used in every cycle, being involved in all the execute cycles, the address calculation and the data transfer.
5. The decode logic is always generating the control signals for the datapath to use in the next cycle, in addition it is also generating the control for the data transfer during the address calculation cycle of the STR.
6. Thus, in this instruction sequence, all parts of the processor are active in every cycle.

The simplest way to view how pipeline works?

• All instructions occupy the datapath for one or more adjacent cycles.

• For each cycle that an instruction occupies the datapath, it occupies the decode logic in the immediately preceding cycle.

• During the first datapath cycle each instruction issues a fetch for the next instruc
tion but one.

• Branch instructions flush and refill the instruction pipeline.

Here PC must point eight bytes (two instructions) ahead of the current instruction.

5-stage pipeline ARM organization

Fetch

The instruction is fetched from memory and placed in the instruction pipeline.

Decode

The instruction is decoded and register operands read from the register file. There are three operand read ports in the register file, so most ARM instructions can source all their operands in one cycle.

Execute

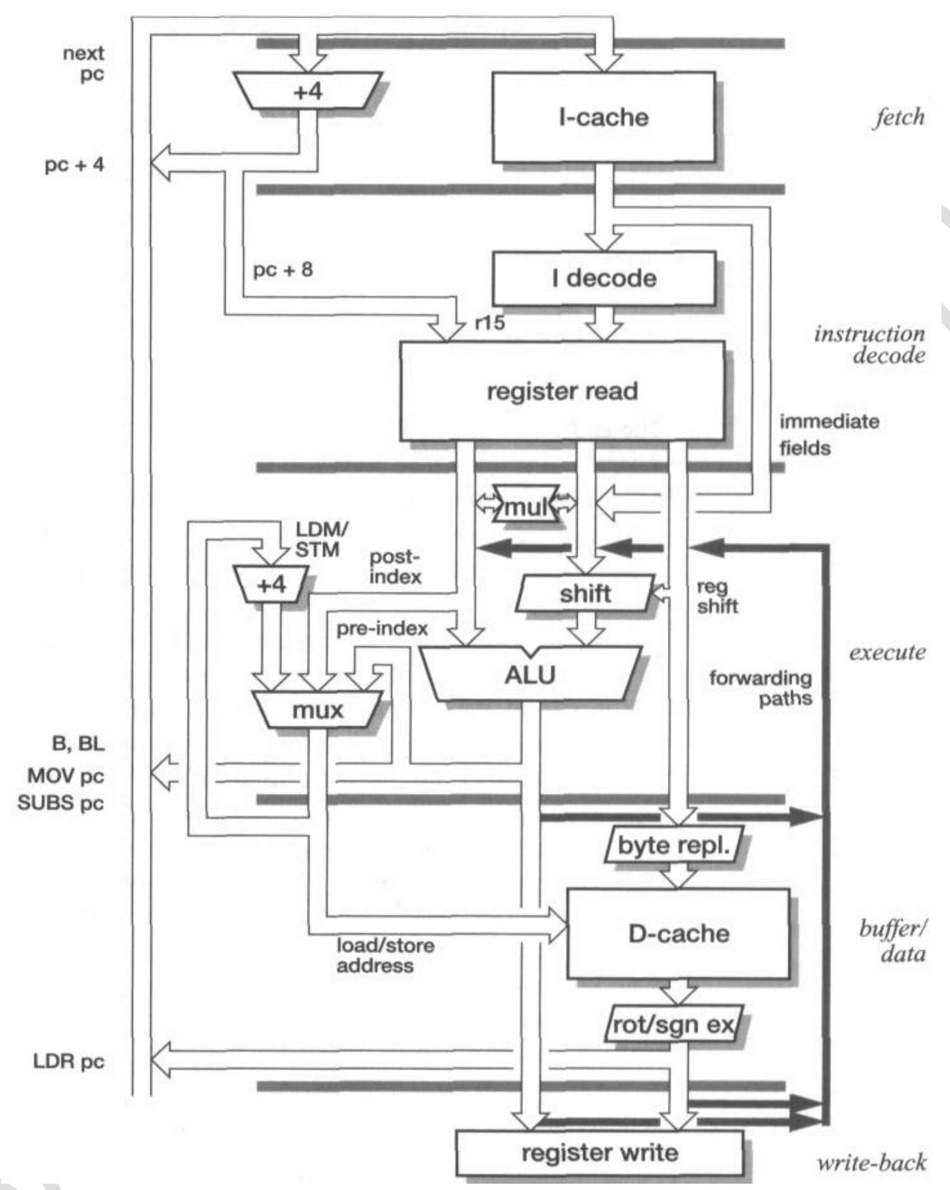
An operand is shifted and the ALU result generated. If the instruction is a load or store the memory address is computed in the ALU.

Buffer/data

Data memory is accessed if required. Otherwise the ALU result is simply buffered for one clock cycle to give the same pipeline flow for all instructions.

Write Back

The results generated by the instruction are written back to the register file, including any data loaded from memory.



Data Forwarding

Data dependencies are resolved without stalling the pipeline by introducing ***forwarding* paths.** Data dependencies arise when an instruction needs to use the result of one of its predecessors before that result has returned to the register file. **Forwarding paths** allow results to be passed between stages as soon as they are available.

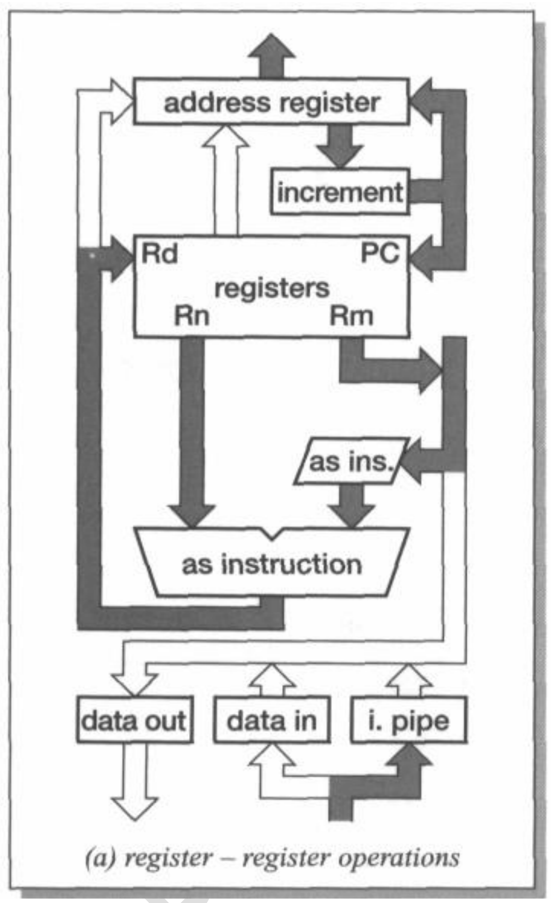
The only way to avoid memory stall is to encourage the compiler (or assembly language programmer) not to put a dependent instruction immediately after a load instruction.

ARM instruction execution

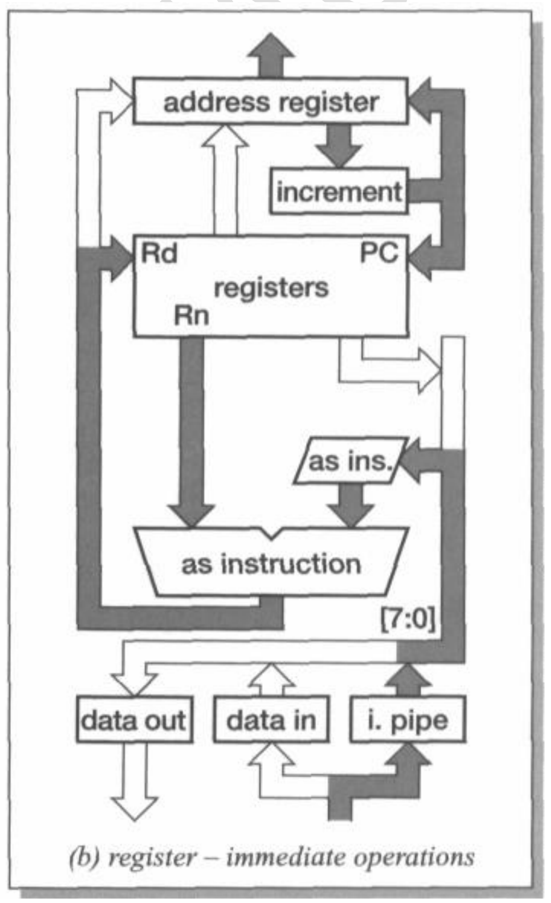
Data processing instruction

A data processing instruction requires two operands, one of which is always a register and the other is either a second register or an immediate value. The second operand is passed through the barrel shifter where it is subject to a general shift operation, then it is combined with the first operand in the ALU using a general ALU operation. Finally, the result from the ALU is written back into the destination register.

All these operations take place in a single clock cycle as shown in Figure below. Note also how the PC value in the address register is incremented and copied back into both the address register and r15 in the register bank, and the next instruction but one is loaded into the bottom of the instruction pipeline *(i. pipe).* The immediate value, when required, is extracted from the current instruction at the top of the instruction pipeline. For data processing instructions only the bottom eight bits (bits [7:0]) of the instruction are used in the immediate value.



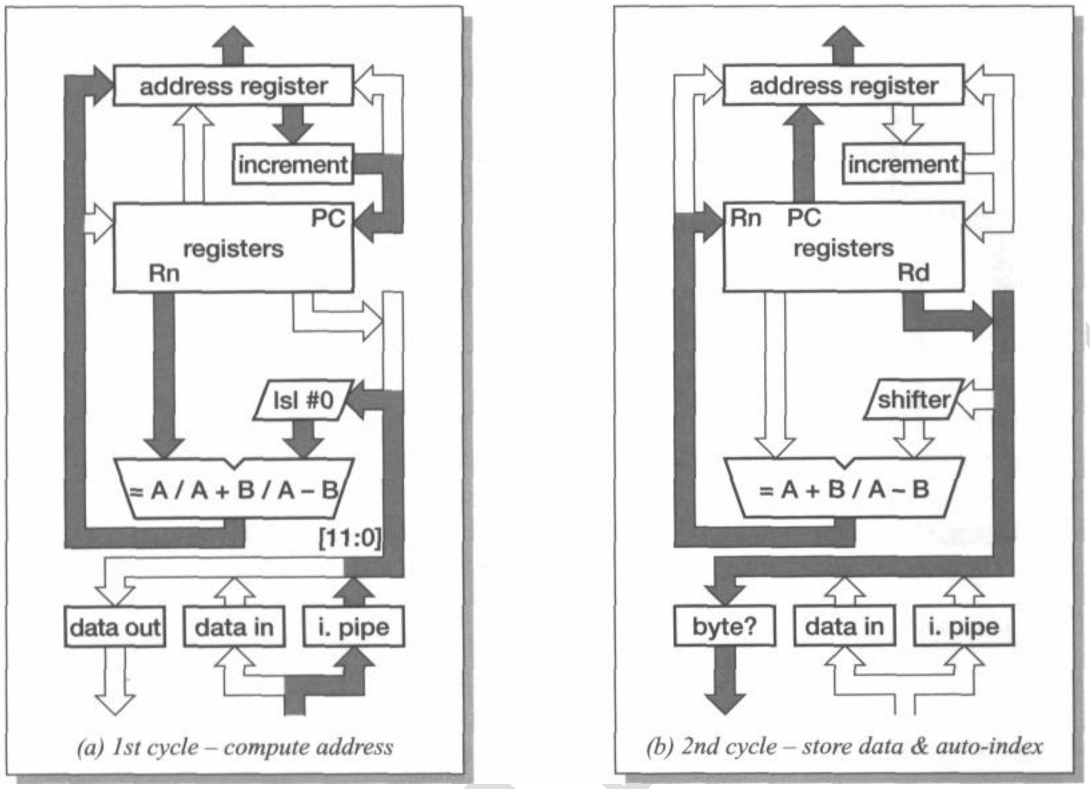
(a) register – register operations



(b) register – immediate operations

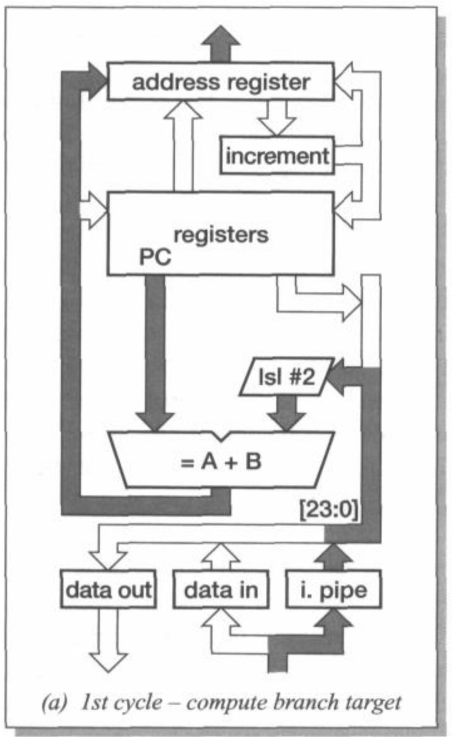
Data transfer instructions

A data transfer (load or store) instruction computes a memory address in a manner very similar to the way a data processing instruction computes its result. A register is used as the base address, to which is added (or from which is subtracted) an offset which again may be another register or an immediate value. This time, however, a 12-bit immediate value is used without a shift operation rather than a shifted 8-bit value. The address is sent to the address register, and in a second cycle the data transfer takes place. Rather than leave the datapath largely idle during the data transfer cycle, the ALU holds the address components from the first cycle and is available to compute an auto-indexing modification to the base register if this is required.

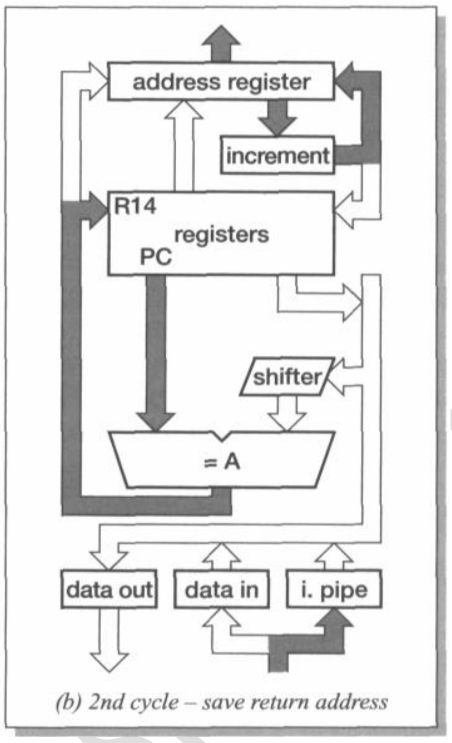


Branch instructions

Branch instructions compute the target address in the first cycle. A 24-bit immediate field is extracted from the instruction and then shifted left two bit positions to give a word-aligned offset which is added to the PC. The result is issued as an instruction fetch address, and while the instruction pipeline refills the return address is copied into the link register (r14) if this is required (that is, if the instruction is a 'branch with link'). The third cycle, which is required to complete the pipeline refilling, is also used to make a small correction to the value stored in the link register in order that it points directly at the instruction which follows the branch. This is necessary because r15 contains pc + 8 whereas the address of the next instruction is pc + 4.



(a) 1st cycle – compute branch target



(b) 2nd cycle – save return address

ARM Implementation

- **Datapath Section**
- **Controlpath Section**

Datapath Section

Clocking Scheme

The design of ARM is based on 2-phase non-overlapping clocks, The non-overlapping property of the phase 1 and phase 2 clocks ensures that there are no race conditions in the circuit.

Datapath timing

Most ARMs do not operate with edge-sensitive registers; instead the design is based around 2-phase non-overlapping clocks. The minimum datapath cycle time is the sum of:

- the register read time;
- the shifter delay;
- the ALU delay;
- the register write set-up time;
- the phase 2 to phase 1 non-overlap time.

The ARM datapath is laid out to a constant pitch per bit. The pitch will be a compromise between the optimum for the complex functions (such as the ALU) which are best suited to a wide pitch and the simple functions (such as the barrel shifter) which are most efficient when laid out on a narrow pitch.

Adder Design

ARM uses AND-OR-INVERT gates and alternating AND/OR logic.

ALU functions

It must perform the full set of data operations defined by the instruction set, including address computations for memory transfers, branch calculations, bit-wise logical functions, and so on.

The ARM6 carry-select adder
This form of adder computes the sums of various fields of the word for a carry-in of both zero and one, and then the final result is selected by using the correct carry-in value to control a multiplexer.

ARM6 ALU structure
A separate logic unit runs in parallel with the adder, and a multiplexer selects the output from the adder or from the logic unit as required.

Carry arbitration adder
This adder computes all intermediate carry values using a 'parallel-prefix' tree, which is a very fast parallel logic structure.

The barrel shifter
A cross-bar switch matrix is used to steer each input to the appropriate output.

Multiplier design
Recent ARM cores have high-performance multiplication hardware and support the 64-bit result multiply and multiply-accumulate instructions.

High-speed multiplier
The high-speed multiplier speeds up multiplication by a factor of around 3 and it supports the added functionality of the 64-bit result forms of the multiply instruction.

The register bank

This is where all the user-visible state is stored in 31 general-purpose 32-bit registers, amounting to around 1 Kbits of data altogether.

Datapath layout
The ARM datapath is laid out to a constant pitch per bit. The pitch will be a compromise between the optimum for the complex functions (such as the ALU) which are best suited to a wide pitch and the simple functions (such as the barrel shifter) which are most efficient when laid out on a narrow pitch.

Control Path Section
The control logic on the simpler ARM cores has three structural components which relate to each other as shown in Figure below:
1. An instruction decoder PLA (programmable logic array). This unit uses some of the instruction bits and an internal cycle counter to define the class of operation to be performed on the datapath in the next cycle.
2. Distributed secondary control associated with each of the major datapath function blocks. This logic uses the class information from the main decoder PLA to select other instruction bits and/or processor state information to control the datapath.
3. Decentralized control units for specific instructions that take a variable number of cycles to complete (load and store multiple, multiply and coprocessor operations). Here the main decoder PLA locks into a fixed state until the remote control unit indicates completion.

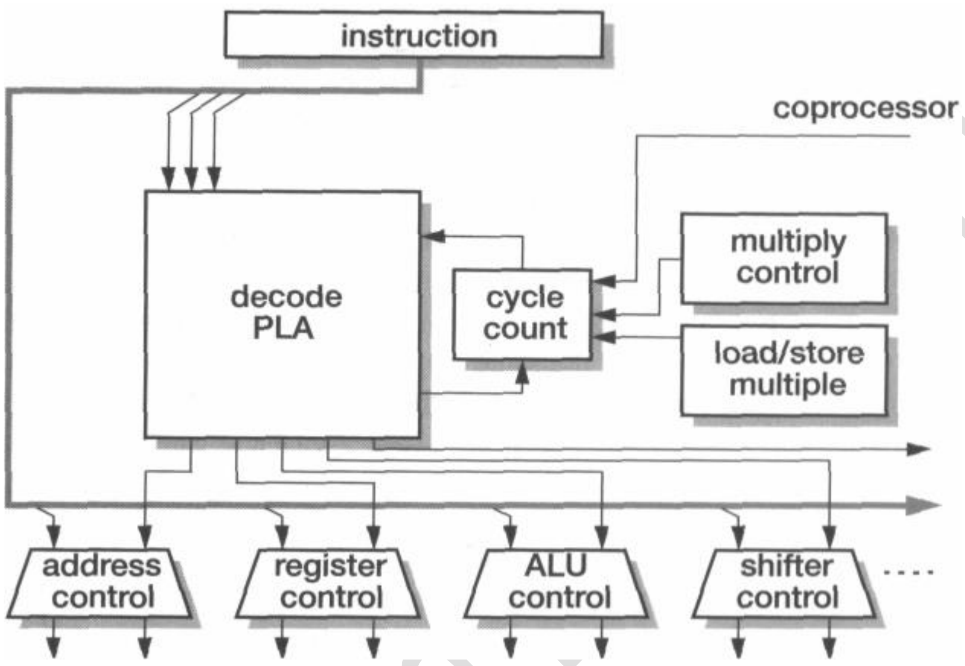
Physical design

There are two principal mechanisms used to implement an ARM processor core (or any other core, for than matter) on a particular process:

• a hard macrocell is delivered as physical layout ready to be incorporated into the final design;

• a soft macrocell is delivered as a synthesizable design expressed in a hardware description language such as VHDL.

Recent ARM cores have been available in both hard and soft forms.



The ARM coprocessor interface

A coprocessor is a computer processor used to supplement the functions of the primary processor (the CPU). Operations performed by the coprocessor may be floating point arithmetic, graphics, signal processing, string processing, cryptography or I/O interfacing with peripheral devices. By offloading processor-intensive tasks from the main processor, coprocessors can accelerate system performance.

Coprocessor Architecture

Its most important features are:

• Support for up to 16 logical coprocessors.

• Each coprocessor can have up to 16 private registers of any reasonable size; they are not limited to 32 bits.

• Coprocessors use a load-store architecture, with instructions to perform internal operations on registers, instructions to load and save registers from and to memory, and instructions to move data to or from an ARM register.

ARM7TDMI coprocessor interface

The ARM7TDMI coprocessor interface is based on 'bus watching' (other ARM cores use different techniques). The coprocessor is attached to a bus where the ARM instruction stream flows into the ARM, and the coprocessor copies the instructions into an internal pipeline that mimics the behaviour of the ARM instruction pipeline.

As each coprocessor instruction begins execution there is a 'hand-shake' between the ARM and the coprocessor to confirm that they are both ready to execute it. The handshake uses three signals:

1. *cpi* (from ARM to all coprocessors).

This signal, which stands for 'Coprocessor Instruction', indicates that the ARM has identified a coprocessor instruction and wishes to execute it.
2. *cpa* (from the coprocessors to ARM).
This is the 'Coprocessor Absent' signal which tells the ARM that there is no coprocessor present that is able to execute the current instruction.
3. *cpb* (from the coprocessors to ARM).
This is the 'CoProcessor Busy' signal which tells the ARM that the coprocessor cannot begin executing the instruction yet.

Handshake Outcomes

Once a coprocessor instruction has entered the ARM7TDMI and coprocessor pipelines, there are four possible ways it may be handled depending on the handshake signals:
1. The ARM may decide not to execute it, either because it falls in a branch shadow or because it fails its condition code test. (All ARM instructions are con ditionally executed, including coprocessor instructions.) ARM will not assert *cpi*, and the instruction will be discarded by all parties.
2. The ARM may decide to execute it (and signal this by asserting *cpi),* but no present coprocessor can take it so *cpa* stays active. ARM will take the undefined instruction trap and use software to recover, possibly by emulating the trapped instruction.
3. ARM decides to execute the instruction and a coprocessor accepts it, but cannot execute it yet. The coprocessor takes *cpa* low but leaves *cpb* high. The ARM will 'busy-wait' until the coprocessor takes *cpb* low, stalling the instruction stream at this point. If an enabled interrupt request arrives while the coprocessor is busy, ARM will break off to handle the interrupt, probably returning to retry the coprocessor instruction later.
4. ARM decides to execute the instruction and a coprocessor accepts it for immediate execution, *cpi*, *cpa* and *cpb* are all taken low and both sides commit to com plete the instruction.

Data transfers

If the instruction is a coprocessor data transfer instruction the ARM is responsible for generating an initial memory address (the coprocessor does not require any connection to the address bus) but the coprocessor determines the length of the transfer; ARM will continue incrementing the address until the coprocessor signals completion.
The *cpa* and *cpb* handshake signals are also used for this purpose. Since the data transfer is not interruptible once it has started, coprocessors should limit the maximum transfer length to 16 words (the same as a maximum length load or store multiple instruction) so as not to compromise the ARM's interrupt response.

Pre-emptive execution

A coprocessor may begin executing an instruction as soon as it enters its pipeline so long as it can recover its state if the handshake does not ultimately complete. All activity must be id em potent (repeatable with identical results) up to the point of commitment.