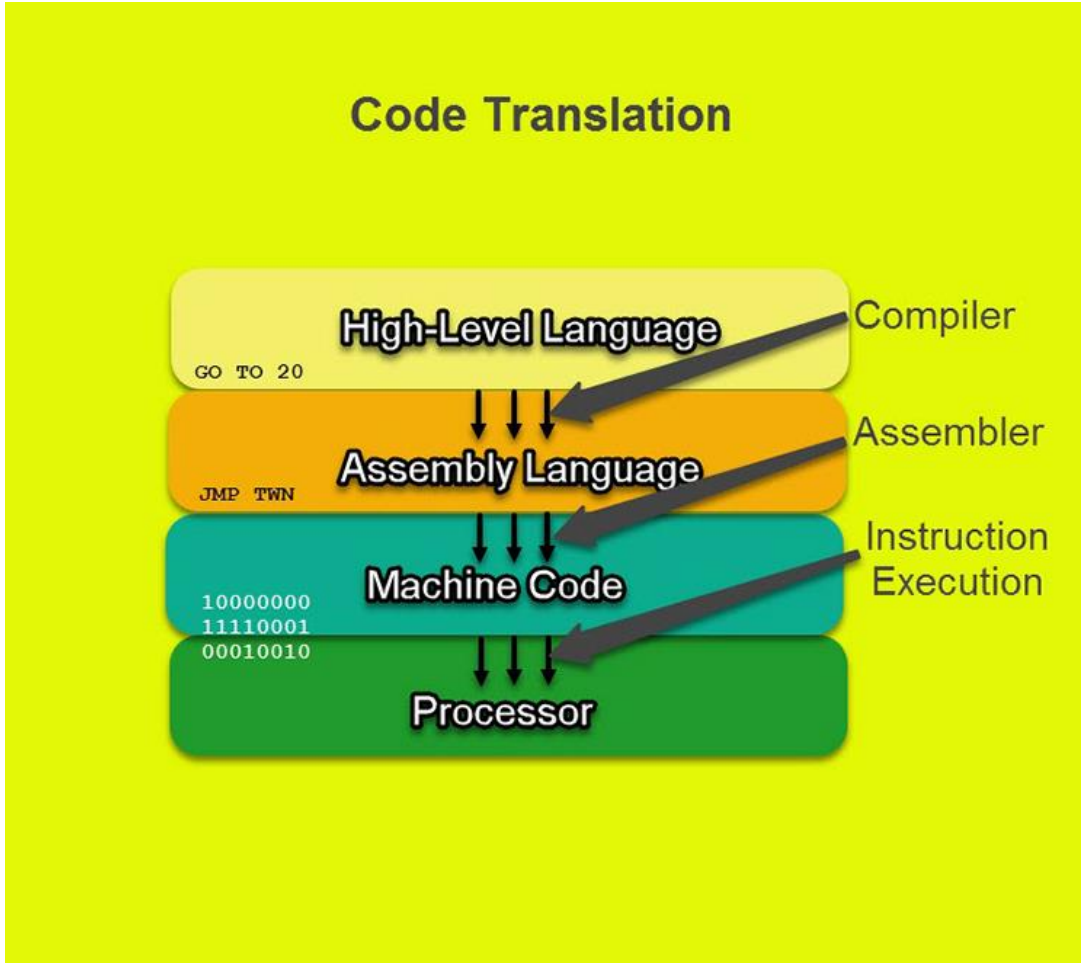
UNIT- 2

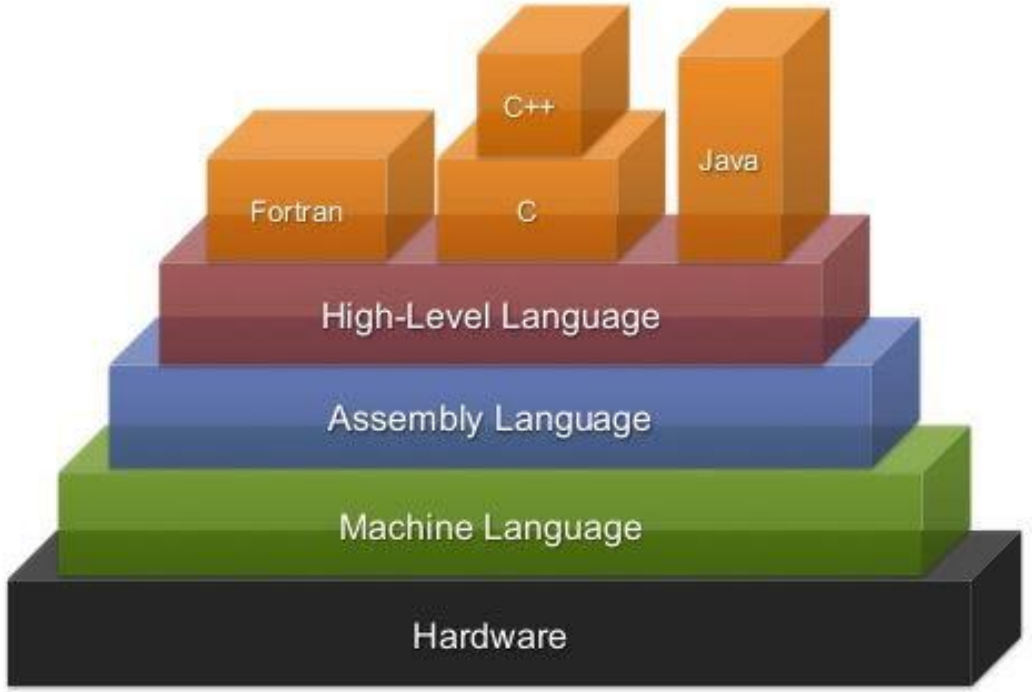
Faculty : Dr. Jisha P

Syllabus

ARM Assembly Programming: Load/Store architecture, ARM instruction set, Assembler rules and Directives, ARM-THUMB interworking, Assembly Language Programs

High level language to machine language conversion process





Addressing modes

1. **Immediate addressing**: the desired value is presented as a binary value in the instruction.

2. **Absolute addressing**: the instruction contains the full binary address of the desired value in memory.

3. **Indirect addressing**: the instruction contains the binary address of a memory location that contains the binary address of the desired value.

4. **Register addressing**: the desired value is in a register, and the instruction contains the register number.

5. **Register indirect addressing**: the instruction contains the number of a register which contains the address of the value in memory.

Addressing modes

6. **Base plus offset addressing**: the instruction specifies a register (the **base) and a** binary offset to be added to the base to form the memory address.

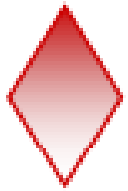
7. **Base plus index addressing**: the instruction specifies a base register and another register (the **index) which is added to the base to form the memory address.**

8. **Base plus scaled index addressing**: as above, but the index is multiplied by a constant (usually the size of the data item, and usually a power of two) before being added to the base.

9. **Stack addressing**: an implicit or specified register (the **stack pointer) points to an** area of memory (the **stack) where data items are written (pushed) or read (popped) on a last-in-first-out basis.**

Addressing modes

- Memory is addressed by generating the Effective Address (EA) of the operand by adding a signed offset to the contents of a base register *Rn*.
- Pre-indexed mode:
  - EA is the sum of the contents of the base register Rn and an offset value.
- Pre-indexed with writeback:
  - EA is generated the same way as pre-indexed mode.
  - EA is written back into Rn.
- Post-indexed mode:
  - EA is the contents of Rn.
  - Offset is then added to this address and the result is written back to Rn.



Addressing modes (contd..)

- Relative addressing mode:
  - Program Counter (PC) is used as a base register.
  - Pre-indexed addressing mode with immediate offset
- No absolute addressing mode available in the ARM processor.
- Offset is specified as:
  - Immediate value in the instruction itself.
  - Contents of a register specified in the instruction.

State and Instruction Sets

- The state of the core determines which instruction set is being executed.
- There are three instruction sets: ARM, Thumb, and Jazelle.
- The ARM instruction set is only active when the processor is in ARM state.
- Similarly the Thumb instruction set is only active when the processor is in Thumb state.
- Once in Thumb state the processor is executing purely Thumb 16-bit instructions.
- You cannot intermingle sequential ARM, Thumb, and Jazelle instructions.
- The Jazelle J and Thumb T bits in the cpsr reflect the state of the processor.
- When both J and T bits are 0, the processor is in ARM state and executes ARM instructions.
- This is the case when power is applied to the processor.
- When the T bit is 1, then the processor is in Thumb state.
- To change states the core executes a specialized branch instruction.

State and Instruction Sets

ARM instruction	`ADD`	`r0, r0, r2`	
Thumb instruction	`ADD`	`r0, r2`	

ARM and Thumb instruction set features.

	ARM (*cpsr* $T = 0$)	Thumb (*cpsr* $T = 1$)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution[a]	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +*pc*	8 general-purpose registers +7 high registers +*pc*

ARM instruction set

- ARM instructions process data held in registers and only access memory with load and store instructions.
- ARM instructions commonly take two or three operands.
- For instance the ADD instruction below adds the two values stored in registers *r1* and *r2* (the source registers).
- It writes the result to register *r3* (the destination register).

Instruction Syntax	Destination register (*Rd*)	Source register 1 (*Rn*)	Source register 2 (*Rm*)
ADD r3, r1, r2	*r3*	*r1*	*r2*

Table 3.1 ARM instruction set.

Mnemonics	ARM ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
B	v1	branch relative +/− 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation
CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register
MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values
SMLA*xy*	v5E	signed multiply accumulate instructions ((16 × 16) + 32 = 32-bit)
SMLAL	v3M	signed multiply accumulate long ((32 × 32) + 64 = 64-bit)
SMLAL*xy*	v5E	signed multiply accumulate long ((16 × 16) + 64 = 64-bit)
SMLAW*y*	v5E	signed multiply accumulate instruction (((32 × 16) ≫ 16) + 32 = 32-bit)
SMULL	v3M	signed multiply long (32 × 32 = 64-bit)

Table 3.1 ARM instruction set. (*Continued*)

Mnemonics	ARM ISA	Description
SMUL*xy*	v5E	signed multiply instructions (16 × 16 = 32-bit)
SMULW*y*	v5E	signed multiply instruction ((32 × 16) ≫ 16 = 32-bit)
STC STC2	v2 v5	store to memory single or multiple 32-bit values from coprocessor
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interruption
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long ((32 × 32) + 64 = 64-bit)
UMULL	v3M	unsigned multiply long (32 × 32 = 64-bit)

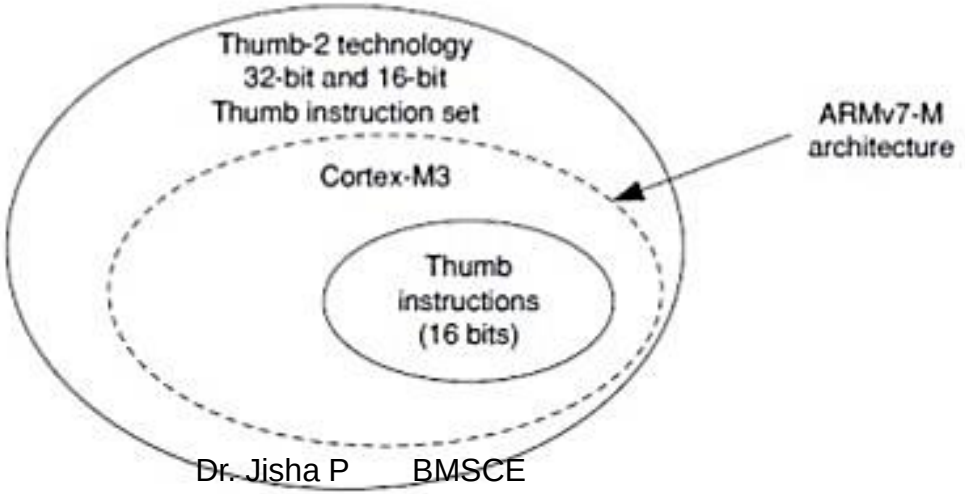
Hexadecimal numbers with the prefix **0x** and binary numbers with the prefix **0b**.

ARM instruction set- Traditional ARM instructions

- Fixed length of 32 bits
- Commonly take two or three operands
- Process data held in registers
- Shift & ALU operation in single clock cycle
- Access memory with load and store instructions only
  - Load/Store multiple register
- Can be extended to execute conditionally by adding the appropriate suffix
- Affect the CPSR status flags by adding the ‘S’ suffix to the instruction

Thumb-2 Instruction Set

- Thumb-2 instruction set is a superset of the previous 16-bit Thumb instruction set
- Provides – A large set of 16-bit instructions, enabling 2 instructions per memory fetch
- A small set of 32-bit instructions to support more complex operations



Thumb-2 Instruction Set: 16bit Thumb-2

- Some of the changes used to reduce the length of the instructions from 32 bits to 16 bits
    - reduce the number of bits used to identify the register
    - reduce the number of bits used for the immediate value
- Smaller number range
    - remove options such as ‘S’
- Make it default for some instructions
- Remove conditional fields (N, Z, V, C)
    - no conditional executions (except branch)
    - remove the optional shift (and no barrel shifter operation)
- Introduce dedicated shift instructions
    - remove some of the instructions
- More restricted coding

ARM Instruction Set -Data Sizes and Instruction Sets

- The ARM is a 32-bit architecture.

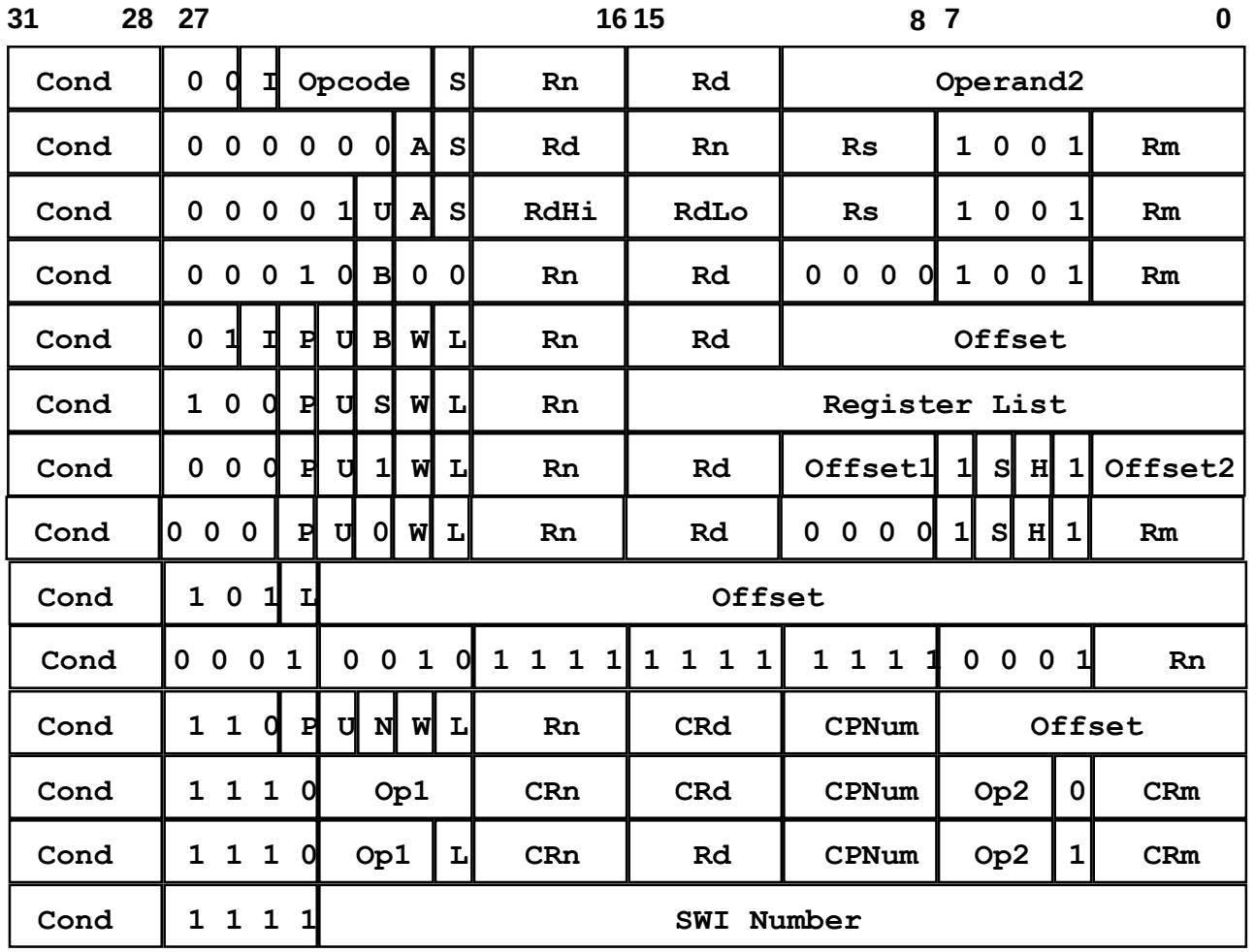
- When used in relation to the ARM:
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)

- Most ARM's implement two instruction sets
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set

ARM Instruction Set - Main features

- All instructions are 32 bits long.
- Most instructions execute in a single cycle.
- Every instruction can be conditionally executed.
- A **load/store architecture**
  - Data processing instructions act only on registers
    - Three operand format
    - Combined ALU and shifter for high speed bit manipulation
  - Specific memory access instructions with powerful auto-indexing addressing modes.
    - 32 bit and 8 bit data types and also 16 bit data types on ARM Architecture v4.
    - Flexible multiple register load and store instructions
- Instruction set extension via coprocessors

ARM Instruction Set Format



Instruction type

Data processing / PSR Transfer

Multiply

Long Multiply (v3M / v4 only)

Swap

Load/Store Byte/Word

Load/Store Multiple

Halfword transfer : Immediate offset (v4 only)

Halfword transfer: Register offset (v4 only)

Branch

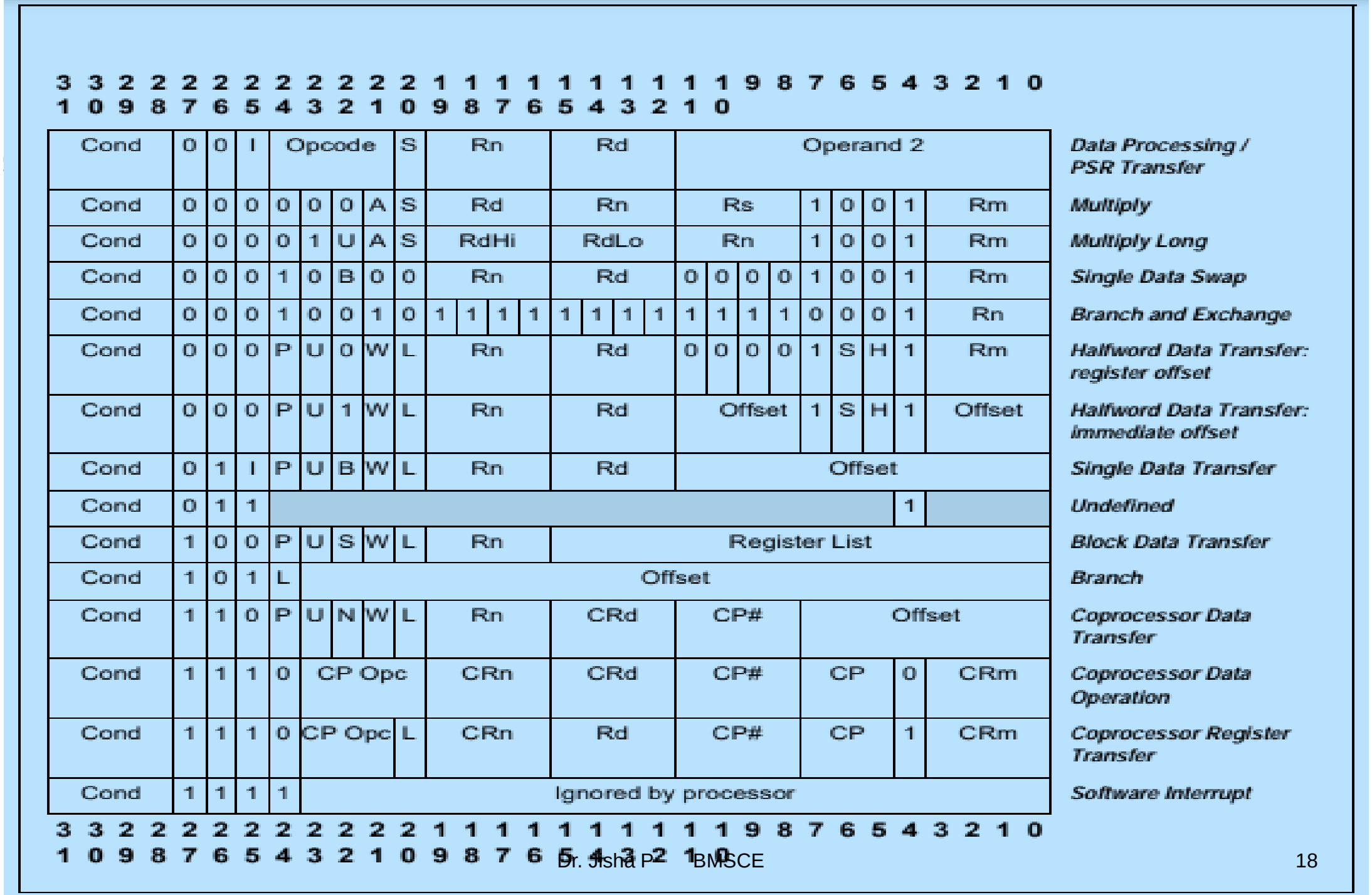
Branch Exchange (v4T only)

Coprocessor data transfer

Coprocessor data operation

Coprocessor register transfer

Software interrupt


3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
Cond 0 0 I Opcode S Rn Rd Operand 2
Data Processing / PSR Transfer
Cond 0 0 0 0 0 0 A S Rd Rn Rs 1 0 0 1 Rm
Multiply
Cond 0 0 0 0 1 U A S RdHi RdLo Rn 1 0 0 1 Rm
Multiply Long
Cond 0 0 0 1 0 B 0 0 Rn Rd 0 0 0 0 1 0 0 1 Rm
Single Data Swap
Cond 0 0 0 1 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 Rn
Branch and Exchange
Cond 0 0 0 P U 0 W L Rn Rd 0 0 0 0 1 S H 1 Rm
Halfword Data Transfer: register offset
Cond 0 0 0 P U 1 W L Rn Rd Offset 1 S H 1 Offset
Halfword Data Transfer: immediate offset
Cond 0 1 I P U B W L Rn Rd Offset
Single Data Transfer
Cond 0 1 1 1
Undefined
Cond 1 0 0 P U S W L Rn Register List
Block Data Transfer
Cond 1 0 1 L Offset
Branch
Cond 1 1 0 P U N W L Rn CRd CP# Offset
Coprocessor Data Transfer
Cond 1 1 1 0 CP Opc CRn CRd CP# CP 0 CRm
Coprocessor Data Operation
Cond 1 1 1 0 CP Opc L CRn Rd CP# CP 1 CRm
Coprocessor Register Transfer
Cond 1 1 1 1 Ignored by processor
Software Interrupt
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

ARM Instruction Set: summary

Mnemonic	Instruction	Action
ADC	Add with carry	Rd := Rn + Op2 + Carry
ADD	Add	Rd := Rn + Op2
AND	AND	Rd := Rn AND Op2
B	Branch	R15 := address
BIC	Bit Clear	Rd := Rn AND NOT Op2
BL	Branch with Link	R14 := R15, R15 := address
BX	Branch and Exchange	R15 := Rn, T bit := Rn[0]

ARM Instruction Set: summary

CDP	Coprocesor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	CPSR flags := Rn + Op2
CMP	Compare	CPSR flags := Rn - Op2
EOR	Exclusive OR	Rd := (Rn AND NOT Op2) OR (op2 AND NOT Rn)
LDC	Load coprocessor from memory	Coprocessor load
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	Rd := (address)

ARM Instruction Set: summary

MCR	Move CPU register to coprocessor register	cRn := rRn {<op>cRm}
MLA	Multiply Accumulate	Rd := (Rm * Rs) + Rn
MOV	Move register or constant	Rd : = Op2
MRC	Move from coprocessor register to CPU register	Rn := cRn {<op>cRm}
MRS	Move PSR status/flags to register	Rn := PSR
MSR	Move register to PSR status/flags	PSR := Rm
MUL	Multiply	Rd := Rm * Rs
MVN	Move negative register	Rd := 0xFFFFFFFF EOR Op2
ORR	OR	Rd := Rn OR Op2
RSB	Reverse Subtract	Rd := Op2 - Rn
RSC	Reverse Subtract with Carry	Rd := Op2 - Rn - 1 + Carry

ARM Instruction Set: summary

Mnemonic	Instruction	Action
SBC	Subtract with Carry	Rd := Rn - Op2 - 1 + Carry
STC	Store coprocessor register to memory	address := CRn
STM	Store Multiple	Stack manipulation (Push)
STR	Store register to memory	<address> := Rd
SUB	Subtract	Rd := Rn - Op2
SWI	Software Interrupt	OS call
SWP	Swap register with memory	Rd := [Rn], [Rn] := Rm
TEQ	Test bitwise equality	CPSR flags := Rn EOR Op2
TST	Test bits	CPSR flags := Rn AND Op2

ARM Instruction Set :Condition code summary

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

ARM Instruction Set

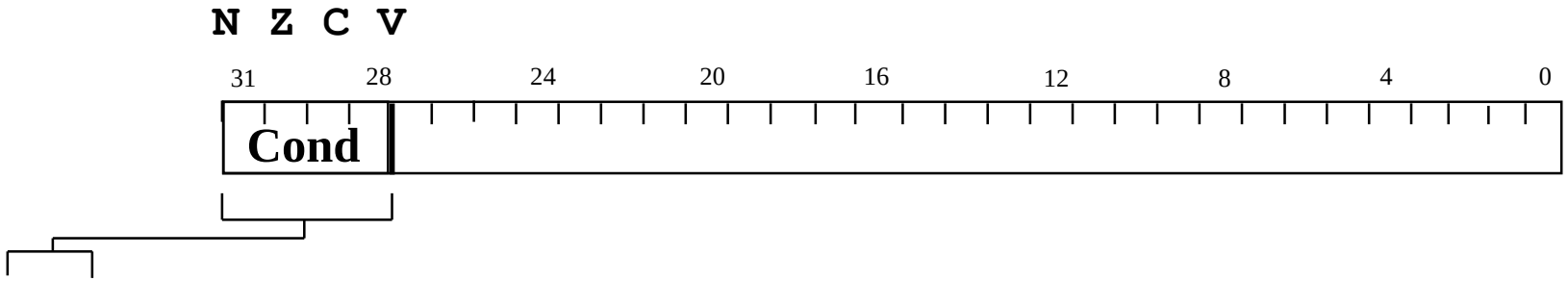
ARM Instruction Classification

- Data Processing Instructions
- Control flow instructions - Conditional and Branching Instructions
- Data Transfer Instructions - Single-Register Load-Store Instructions
- Stack Instructions
- Software Interrupt Instructions

Control Flow Instructions: Conditional Execution

- Most instruction sets only allow branches to be executed conditionally.
- However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.
  - All instructions contain a condition field which determines whether the CPU will execute them.
  - Non-executed instructions soak up 1 cycle.
  - Still have to complete cycle so as to allow fetching and decoding of following instructions.
- This removes the need for many branches, which stall the pipeline (3 cycles to refill).
  - Allows very dense in-line code, without branches.
  - The Time penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed.

The Condition Field



000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear, or N clear and V set (<, or =)

1110 = AL - always

1111 = NV - reserved.

Condition Codes

The possible condition codes are listed below

- Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	**Z=1**
NE	Not equal	**Z=0**
CS/HS	Unsigned higher or same	**C=1**
CC/LO	Unsigned lower	**C=0**
MI	Minus	**N=1**
PL	Positive or Zero	**N=0**
VS	Overflow	**V=1**
VC	No overflow	**V=0**
HI	Unsigned higher	**C=1 & Z=0**
LS	Unsigned lower or same	**C=0 or Z=1**
GE	Greater or equal	**N=V**
LT	Less than	**N!=V**
GT	Greater than	**Z=0 & N=V**
LE	Less than or equal	**Z=1 or N=!V**
AL	Always	

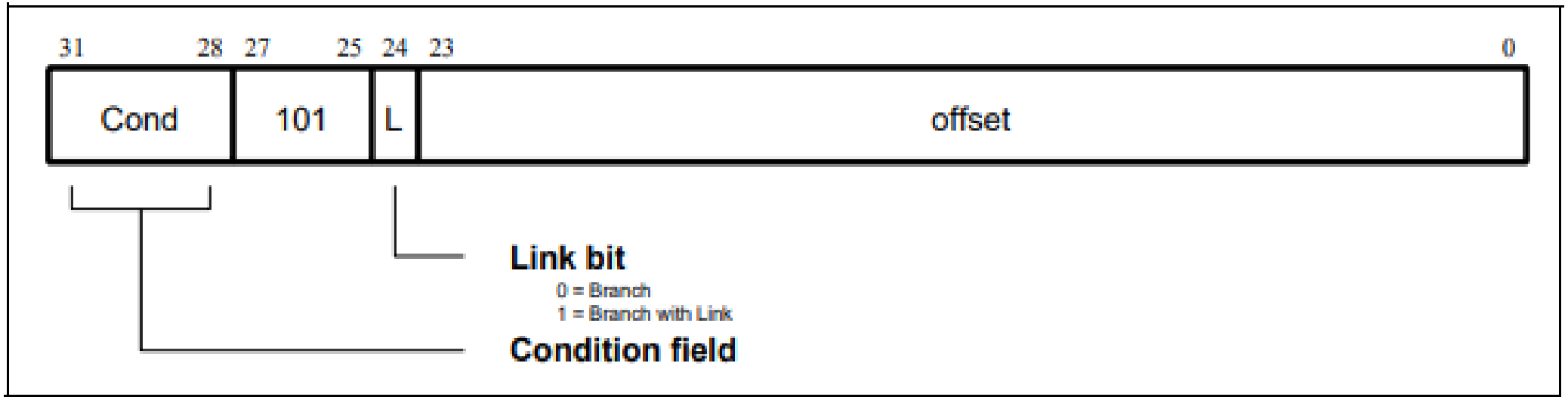
Using and updating the Condition Field

- To execute an instruction conditionally, simply postfix it with the appropriate condition:
  - For example an add instruction takes the form:
    - `ADD r0,r1,r2 ; r0 = r1 + r2 (ADDAL)`
  - To execute this only if the zero flag is set:
    - `ADDEQ r0,r1,r2 ; If zero flag set then…`
      `; ... r0 = r1 + r2`
- By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect).
- To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.
  - For example to add two numbers and set the condition flags:
    - `ADDS r0,r1,r2; r0 = r1 + r2 ; ... and set flags`

Branch instructions

Branch and Branch with Link (B, BL)

The instruction is only executed if the condition is true. The various conditions are defined ***Table 4-2: Condition code summary*** on page 4-5. The instruction encoding is shown in ***Figure 4-3: Branch instructions***, below.

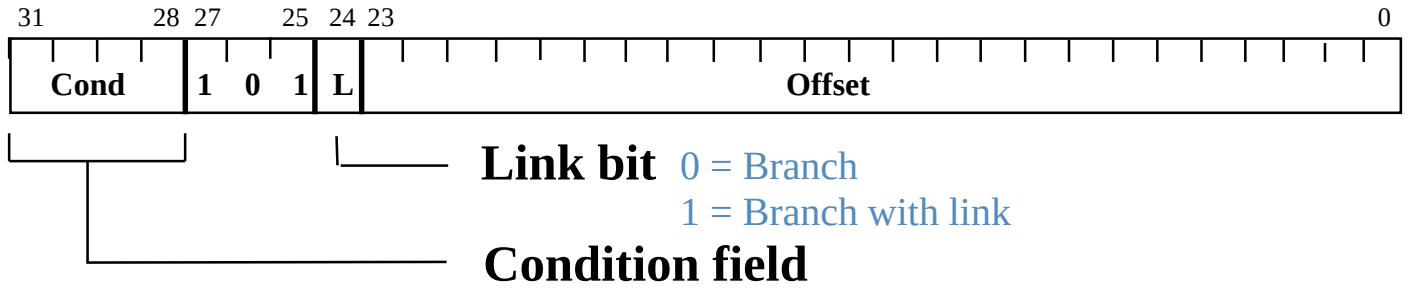


Branch instructions

- Branch instructions contain a signed 2's complement 24 bit offset. This is shifted left two bits, sign extended to 32 bits, and added to the PC. The instruction can therefore specify a branch of +/- 32Mbytes.
- The branch offset must take account of the prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction.
- Branches beyond +/- 32Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case the PC should be manually saved in R14 if a Branch with Link type operation is required.
- **The link bit:**
  - Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction. Note that the CPSR is not saved with the PC and R14[1:0] are always cleared. To return from a routine called by Branch with Link use MOV PC,R14 if the link register is still valid or LDM Rn!,{..PC} if the link register has been saved onto a stack pointed to by Rn.

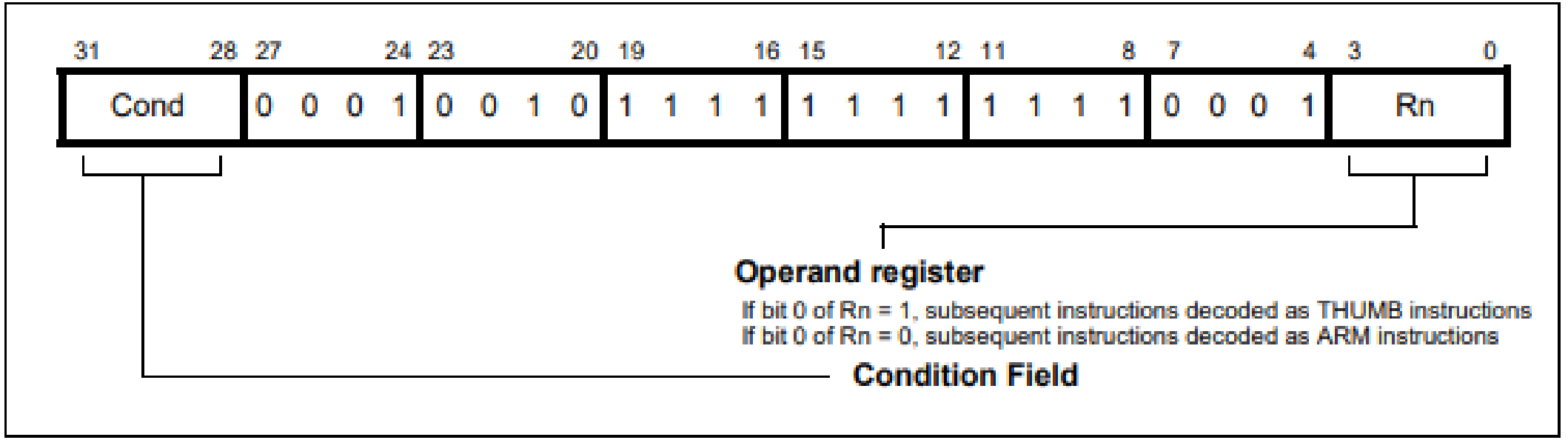
Branch instructions

- **Branch :** `B{<cond>} label`
- **Branch with Link :** `BL{<cond>} sub_routine_label`



- The offset for branch instructions is calculated by the assembler:
  - By taking the difference between the branch instruction and the target address minus 8 (to allow for the pipeline).
  - This gives a 26 bit offset which is right shifted 2 bits (as the bottom two bits are always zero as instructions are word – aligned) and stored into the instruction encoding.
  - This gives a range of ± 32 Mbytes.

Branch instructions



- This instruction performs a branch by copying the contents of a general register, Rn, into the program counter, PC. The branch causes a pipeline flush and refill from the address specified by Rn.
- This instruction also permits the instruction set to be exchanged. When the instruction is executed, the value of Rn[0] determines whether the instruction stream will be decoded as ARM or THUMB instructions

Branch instructions

Assembler syntax

BX - branch and exchange.

```
BX{cond} Rn
```

{cond}	Two character condition mnemonic. See ***Table 4-2: Condition code summary*** on page 4-5.
Rn	is an expression evaluating to a valid register number.

Assembler syntax

Items in {} are optional. Items in <> must be present.

```
B{L}{cond} <expression>
```

{L}	is used to request the Branch with Link form of the instruction. If absent, R14 will not be affected by the instruction.
{cond}	is a two-character mnemonic as shown in ***Table 4-2: Condition code summary*** on page 4-5. If absent then AL (ALways) will be used.
<expression>	is the destination. The assembler calculates the offset.

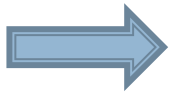
Branch instructions

- When executing the instruction, the processor:
  - shifts the offset left two bits, sign extends it to 32 bits, and adds it to PC.
- Execution then continues from the new PC, once the pipeline has been refilled.
- The "Branch with link" instruction implements a subroutine call by writing PC-4 into the LR of the current bank.
  - i.e. the address of the next instruction following the branch with link (allowing for the pipeline).
- To return from subroutine, simply need to restore the PC from the LR:
  - `MOV pc, lr`
  - Again, pipeline has to refill before execution continues.
- The "Branch" instruction does not affect LR.

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

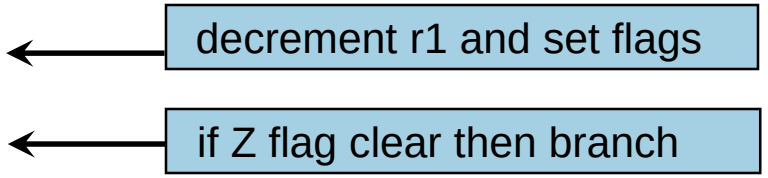
```
    CMP   r3,#0
    BEQ   skip
    ADD   r0,r1,r2
skip
```



```
CMP   r3,#0
ADDNE r0,r1,r2
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using “S”. CMP does not need “S”.

```
loop
  …
  SUBS r1,r1,#1
  BNE loop
```



Branch instructions

- Using conditional instructions rather than conditional branches can save both code size and cycles.
- This example shows the difference between using branches and using conditional instructions. It uses the Euclid algorithm for the Greatest Common Divisor (gcd) to show how conditional instructions improve code size and speed.
- In C the gcd algorithm can be expressed as:

```
int gcd(int a, int b)
{
    while (a != b)
      {
        if (a > b)
            a = a - b;
        else
            b = b - a;
      }
    return a;
}
```

Examples show implementations of the gcd algorithm with and without conditional instructions

- Conditional execution by using conditional branches
- The code is seven instructions long because of the number of branches.
- Every time a branch is taken, the processor must refill the pipeline and continue from the new location.
- The other instructions and non-executed branches use a single cycle each.

```
gcd     CMP     r0, r1
        BEQ     end
        BLT     less
        SUBS    r0, r0, r1  ; could be SUB r0, r0, r1 for A32
        B       gcd
less
        SUBS    r1, r1, r0  ; could be SUB r1, r1, r0 for A32
        B       gcd
end
```

The following table shows the number of cycles this implementation uses on an Arm7™ processor when R0 equals 1 and R1 equals 2.

Table 1. Conditional branches only

R0: a	R1: b	Instruction	Cycles (Arm7)
1	2	`CMP r0, r1`	1
1	2	`BEQ end`	1 (not executed)
1	2	`BLT less`	3
1	2	`SUB r1, r1, r0`	1
1	2	`B gcd`	3
1	1	`CMP r0, r1`	1
1	1	`BEQ end`	3
-	-	-	Total = 13

Conditional execution using conditional instructions

- Implementation of the gcd algorithm using individual conditional instructions in A32 code. The gcd algorithm only takes four instructions:

```
gcd
        CMP     r0, r1
        SUBGT   r0, r0, r1
        SUBLE   r1, r1, r0
        BNE     gcd
```

- In addition to improving code size, in most cases this code executes faster than the version that uses only branches.

The following table shows the number of cycles this implementation uses on an Arm7 processor when R0 equals 1 and R1 equals 2.

R0: a	R1: b	Instruction	Cycles (Arm7)
1	2	`CMP r0, r1`	1
1	2	`SUBGT r0,r0,r1`	1 (not executed)
1	1	`SUBLT r1,r1,r0`	1
1	1	`BNE gcd`	3
1	1	`CMP r0,r1`	1
1	1	`SUBGT r0,r0,r1`	1 (not executed)
1	1	`SUBLT r1,r1,r0`	1 (not executed)
1	1	`BNE gcd`	1 (not executed)
-	-	-	Total = 10

Comparing this with the example that uses only branches:

- Replacing branches with conditional execution of all instructions saves three cycles.
- Where R0 equals R1, both implementations execute in the same number of cycles. For all other cases, the implementation that uses conditional instructions executes in fewer cycles than the implementation that uses branches only.

Conditional execution examples

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
  CMP r0, #0
  ADDEQ r1, r1, #1
  ADDNE r2, r2, #1
  ...
```

- 3 instructions
- 3 words
- 3 cycles

Example:1

```
        AREA prg1,CODE,READONLY
        EXPORT __main
        ENTRY
__main
        MOV r0,#2
        MOV r1,#8
gcd     CMP r0,r1
        BEQ over
        BLT less
        subs r0,r0,r1
        b gcd
less    subs r1,r1,r0
        b gcd
over
        end
```

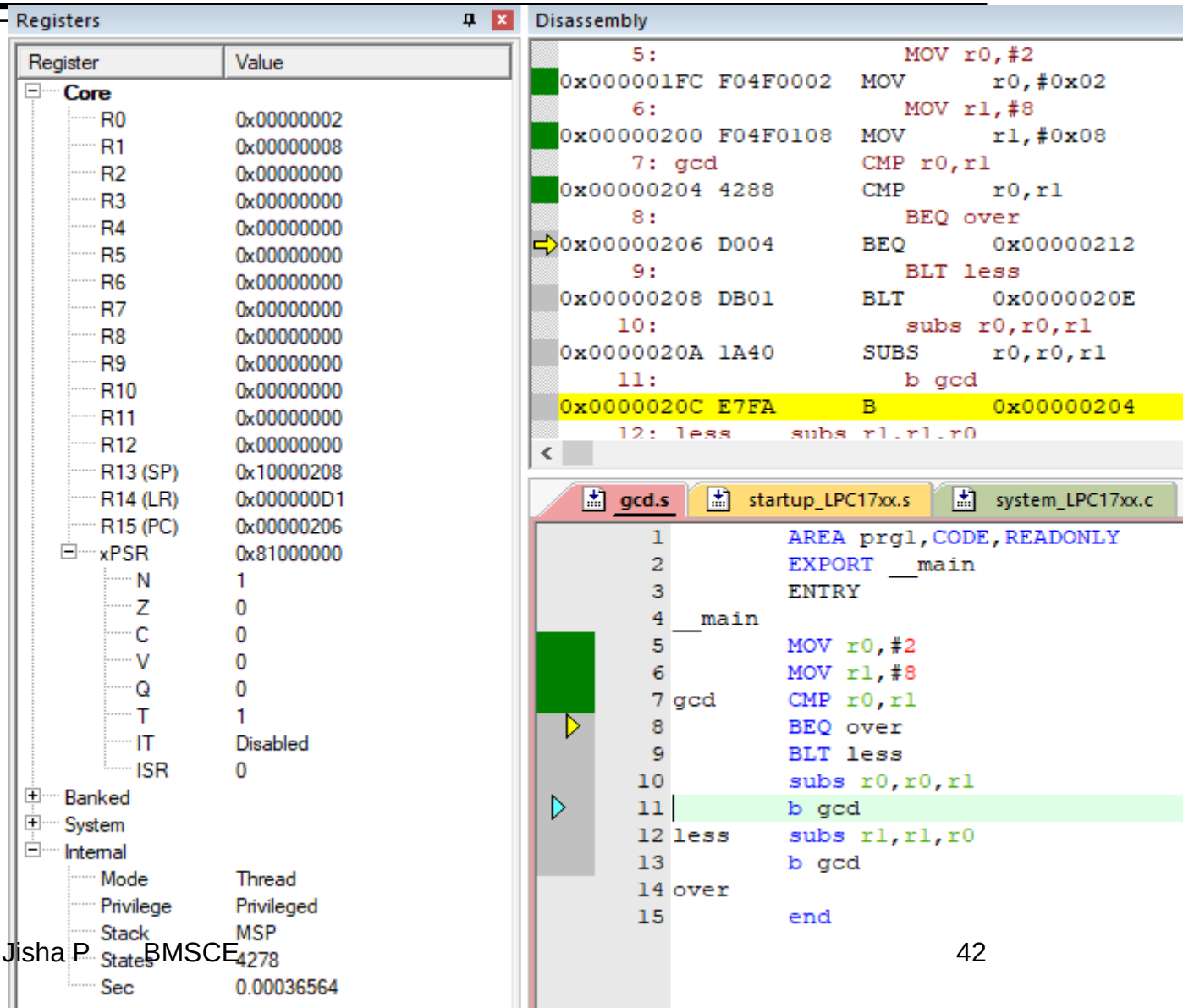


Table 4.3 16-Bit Branch Instructions

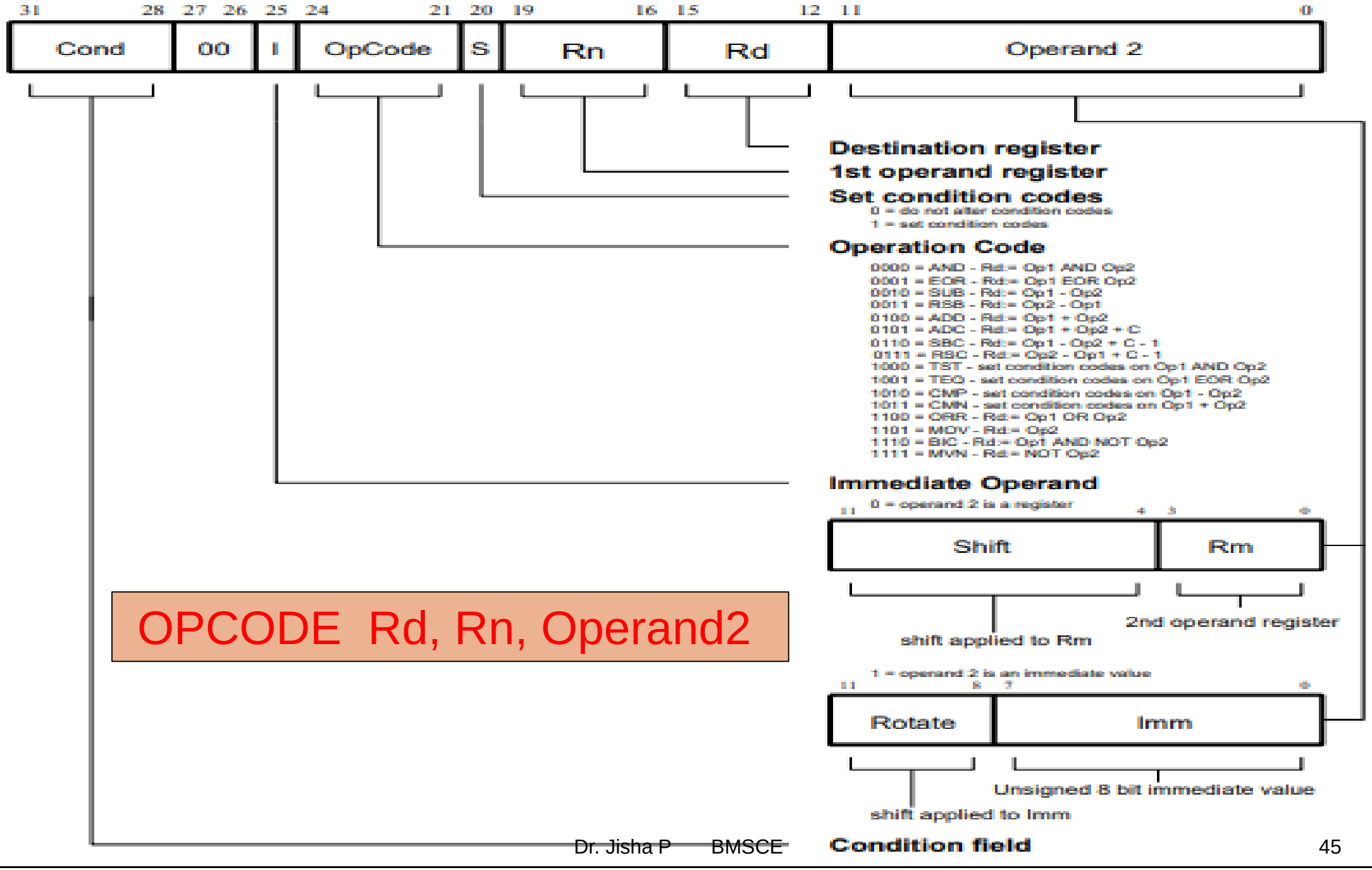
Instruction	Function
B	Branch
B<cond>	Conditional branch
BL	Branch with link; call a subroutine and store the return address in LR (this is actually a 32-bit instruction, but it is also available in Thumb in traditional ARM processors)
BLX	Branch with link and change state (BLX <reg> only)[1]
BX <reg>	Branch with exchange state
CBZ	Compare and branch if zero (architecture v7)
CBNZ	Compare and branch if nonzero (architecture v7)
IT	IF-THEN (architecture v7)

Table 4.8 32-Bit Branch Instructions

Instruction	Function
B	Branch
B<cond>	Conditional branch
BL	Branch and link
TBB	Table branch byte; forward branch using a table of single byte offset
TBH	Table branch half word; forward branch using a table of half word offset

Data processing Instructions

- Largest family of ARM instructions, all sharing the same instruction format.
- Contains:
  - Arithmetic operations
  - Comparisons (no results - just set condition codes)
  - Logical operations
  - Data movement between registers
- Remember, this is a load / store architecture
  - These instruction only work on registers, ***NOT*** memory.
- They each perform a specific operation on one or two operands.
  - First operand always a register - Rn
  - Second operand sent to the ALU via barrel shifter.



OPCODE Rd, Rn, Operand2

OPCODE Rd, Rn, Operand2

ADD R1, R2, R3 ; R1 = R2 + R3
SUB R4, R5, #10 ; R4 = R5 - 10

Data processing Instructions

- The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn).
- The second operand may be a shifted register (Rm) or a rotated 8 bit immediate value (Imm) according to the value of the I bit in the instruction.
- The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction.
- Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd.
- They are used only to perform tests and to set the condition codes on the result and always have the S bit set

Data processing Instructions

- The data processing instructions manipulate data within registers.
- They are move instructions, arithmetic instructions, logical instructions, comparison instructions, and multiply instructions.
- Most data processing instructions can process one of their operands using the barrel shifter.
- If you use the S suffix on a data processing instruction, then it updates the flags in the cpsr.
- Move and logical operations update the carry flag *C*, negative flag *N*, and zero flag *Z*.
- The carry flag is set from the result of the barrel shift as the last bit shifted out.
- The *N* flag is set to bit 31 of the result. The *Z* flag is set if the result is zero.

Data processing Instructions

Data Processing Instruction

a) MOVE INSTRUCTIONS.
b) BARREL SHIFTER.
c) ARITHMETIC INSTRUCTIONS.
d) USING THE BARREL SHIFTER WITH ARITHMETIC INSTRUCTIONS.
e) LOGICAL INSTRUCTIONS.
f) COMPARISION INSTRUCTIONS.
g) MULTIPLY INSTRUCTIONS.

Data Processing Instructions

- Consist of :
  - Arithmetic: **ADD ADC SUB SBC RSB RSC**
  - Logical: **AND ORR EOR BIC**
  - Comparisons: **CMP CMN TST TEQ**
  - Data movement: **MOV MVN**

- These instructions only work on registers, NOT memory.

  Syntax:

  `<Operation>{<cond>}{S} Rd, Rn, Operand2`

  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn

- Second operand is sent to the ALU via barrel shifter.

Data processing Instructions

Examples:

```
MOV R0, #5        ; Load immediate value 5 into R0
 MOV R1, #10      ; Load immediate value 10 into R1
 ADD R2, R0, R1   ; R2 = R0 + R1 = (5 + 10)
 SUB R3, R1, R0   ; R3 = R1 - R0 = (10 - 5)
 AND R4, R0, R1   ; Bitwise AND operation on R0 and R1
```

Data Processing Instructions - Move Instructions

- Move is the simplest ARM instruction. It copies *N* into a destination register *Rd*, where *N* is a register or immediate value.
- This instruction is useful for setting initial values and transferring data between registers.

```
Syntax: <instruction>{<cond>}{S} Rd, N
```

MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

- A MOV instruction where **N is a simple register**. But N can be more than just a register or immediate value; it can also be a register Rm that has been pre-processed by the barrel shifter prior to being used by a data processing instruction.

Barrel Shifter

- Data processing instructions are processed within the arithmetic logic unit (ALU).
- A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.
- This shift increases the power and flexibility of many data processing operations.
- There are data processing instructions that do not use the barrel shift, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions.
- Pre-processing or shift occurs within the cycle time of the instruction.
- This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.

Barrel Shifter

PRE : R5 = 5

R7 = 8

MOVS R7, R5, LSL #2 ; let, R7 = R5 x 4 = (R5 << 2)

POST : R5 = 5

R7 = **20** **(20 = 0b10100 = 0x14)**

- The example multiplies register *R5* by four and then places the result into register *R7*.
- For example, the contents of bit 0 are shifted to bit 1. Bit 0 is cleared.
- The *C* flag is updated with the last bit shifted out of the register.
- This is bit (32−*y*) of the original value, where *y* is the shift amount. When *y* is greater than one, then a shift by *y* positions is the same as a shift by one position executed *y* times.

Barrel Shifter

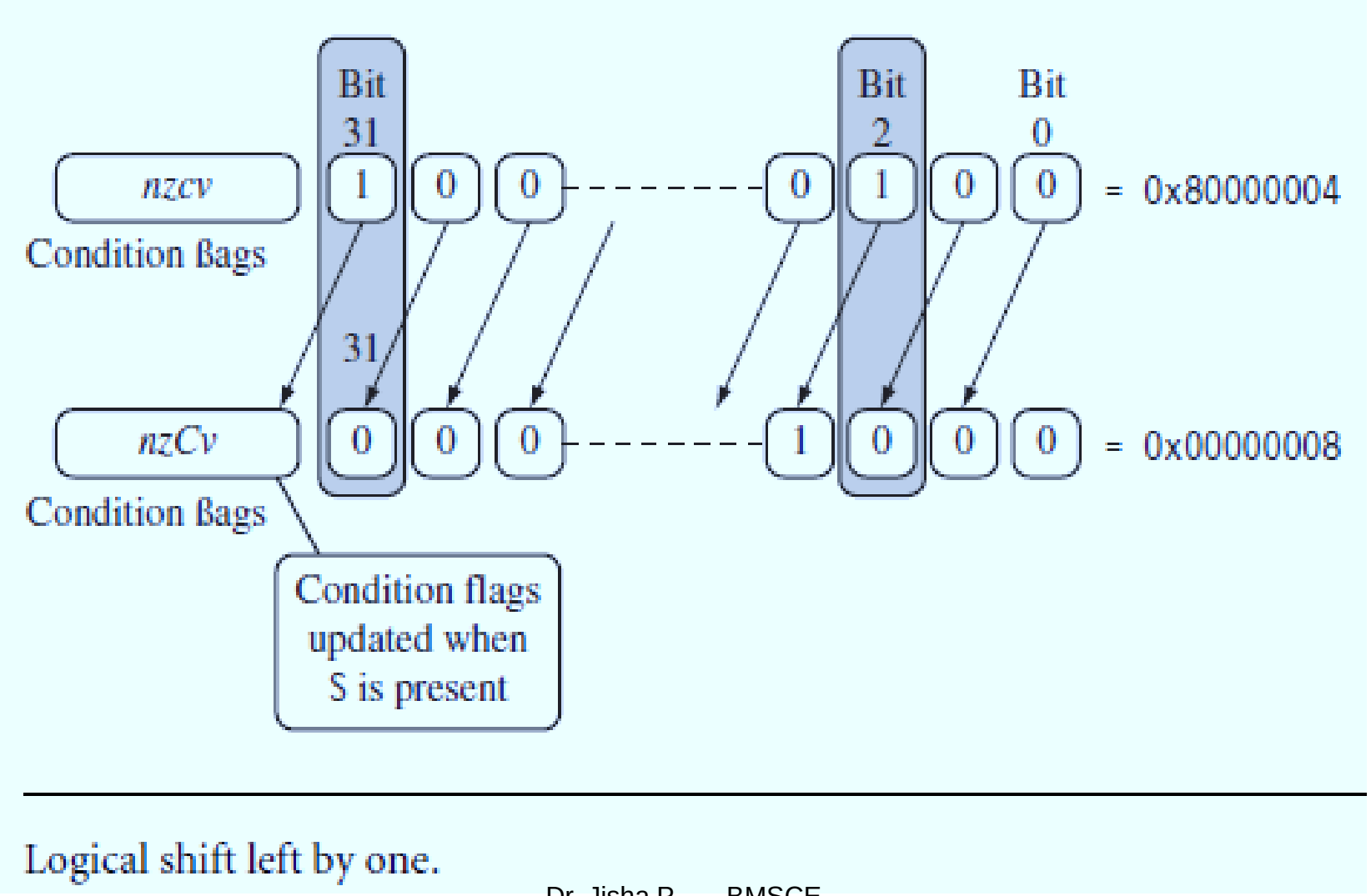
EXAMPLE 3.3 This example of a MOVS instruction shifts register *r1* left by one bit. This multiplies register *r1* by a value 2^1. As you can see, the *C* flag is updated in the *cpsr* because the S suffix is present in the instruction mnemonic.

```
PRE     cpsr = nzcvqiFt_USER
        r0 = 0x00000000
        r1 = 0x80000004

        MOVS    r0, r1, LSL #1

POST    cpsr = nzCvqiFt_USER
        r0 = 0x00000008
        r1 = 0x80000004
```

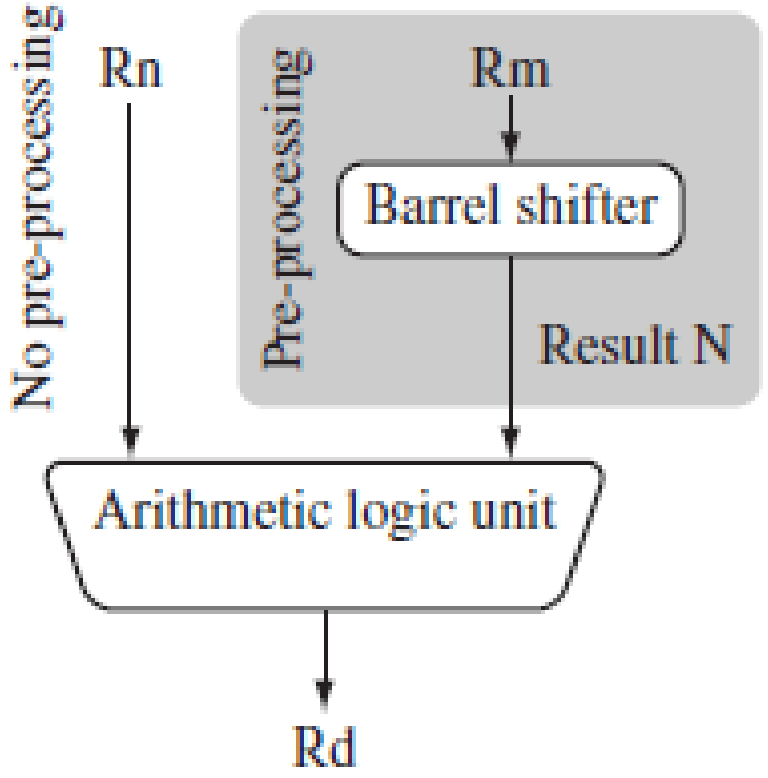
Barrel Shifter



Logical shift left by one.

Barrel Shifter

- Register *Rn* enters the ALU without any pre-processing of registers.
- Figure shows the data flow between the ALU and the barrel shifter.
- We apply a logical shift left (LSL) to register Rm before moving it to the destination register.
- This is the same as applying the standard C language shift operator to the register.
- The MOV instruction copies the shift operator result N into register Rd. N represents the result of the LSL operation



Barrel shifter and ALU.

Barrel Shifter

Table 3.2 Barrel shifter operations.

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	xLSLy	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	xLSRy	(unsigned)$x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	xASRy	(signed)$x \gg y$	#1–32 or *Rs*
ROR	rotate right	xRORy	((unsigned)$x \gg y$) | ($x \ll (32 - y)$)	#1–31 or *Rs*
RRX	rotate right extended	xRRX	(c flag $\ll 31$) | ((unsigned)$x \gg 1$)	none

Note: x represents the register being shifted and y represents the shift amount.

Barrel Shifter

Barrel shift operation syntax for data processing instructions.

N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm, ASR #shift_imm`
Arithmetic shift right by register	`Rm, ASR Rs`
Rotate right by immediate	`Rm, ROR #shift_imm`
Rotate right by register	`Rm, ROR Rs`
Rotate right with extend	`Rm, RRX`

Arithmetic Operations

- Operations are:
  - ADD operand1 + operand2
  - ADC operand1 + operand2 + carry
  - SUB operand1 - operand2
  - SBC operand1 - operand2 + carry -1
  - RSB operand2 - operand1
  - RSC operand2 - operand1 + carry - 1
- Syntax:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- Examples:
  - ADD R0, R1, R2
  - SUBGT R3, R3, #1
  - RSBLES R4, R5, #5

Arithmetic Operations

Syntax: <instruction>{<cond>}{S} Rd, Rn, N

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

N is the result of the shifter operation. The syntax of shifter operation is shown in Table 3.3.

Arithmetic Operations

- With the traditional Thumb instruction syntax, when 16-bit Thumb code is used, an ADD instruction can change the flags in the PSR.
- 32-bit Thumb-2 code can either change a flag or keep it unchanged.
- To separate the two different operations, the *S* suffix should be used if the following operation depends on the flags:
  - **ADD R0, R1, R2 ; Flag unchanged**
  - **ADDS R0, R1, R2 ; Flag change**
- Aside from ADD instructions, the arithmetic functions that the Cortex-M3 supports include subtract (SUB), multiply (MUL), and unsigned and signed divide (UDIV/SDIV)
- The Cortex-M3 also supports 32-bit multiply instructions and multiply accumulate instructions that give 64-bit results.
- These instructions support signed or unsigned values

Arithmetic Operations

EXAMPLE 3.4 This simple subtract instruction subtracts a value stored in register *r2* from a value stored in register *r1*. The result is stored in register *r0*.

```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000001

        SUB r0, r1, r2

POST    r0 = 0x00000001
```

EXAMPLE 3.5 This reverse subtract instruction (RSB) subtracts *r1* from the constant value #0, writing the result to *r0*. You can use this instruction to negate numbers.

```
PRE     r0 = 0x00000000
        r1 = 0x00000077

        RSB r0, r1, #0      ; Rd = 0x0 - r1

POST    r0 = -r1 = 0xffffff89
```

Arithmetic Operations

EXAMPLE 3.6 The SUBS instruction is useful for decrementing loop counters. In this example we subtract the immediate value one from the value one stored in register *r1*. The result value zero is written to register *r1*. The *cpsr* is updated with the *ZC* flags being set.

```
PRE     cpsr = nzcvqiFt_USER
        r1 = 0x00000001

        SUBS r1, r1, #1

POST    cpsr = nZCvqiFt_USER
        r1 = 0x00000000
```

Using the Barrel Shifter with Arithmetic Instructions

EXAMPLE 3.7 Register *r1* is first shifted one location to the left to give the value of twice *r1*. The ADD instruction then adds the result of the barrel shift operation to register *r1*. The final result transferred into register *r0* is equal to three times the value stored in register *r1*.

```
PRE     r0 = 0x00000000
        r1 = 0x00000005

        ADD     r0, r1, r1, LSL #1

POST    r0 = 0x0000000f
        r1 = 0x00000005
```

Example:2 – Arithmetic Instructions

```
    AREA EXP2, CODE, READONLY
    ENTRY
    ; Define memory locations
    LDR R4, =RESULT  ; Memory address to store result
; Load operands
    MOV R0, #10      ; First number (change as needed)
    MOV R1, #5       ; Second number (change as needed)
    MOV R2, #2
; Operator selection: 1=Add, 2=Sub, 3=Mul, 4=Div
    CMP R2, #1       ; Check if addition
    BEQ ADDITION
    CMP R2, #2       ; Check if subtraction
    BEQ SUBTRACTION
    CMP R2, #3       ; Check if multiplication
    BEQ MULTIPLICATION
    CMP R2, #4       ; Check if division
    BEQ DIVISION
    B END
; Exit program if no valid operation
```

Example:2

ADDITION

```
ADD R3, R0, R1   ; R3 = R0 + R1
STR R3, [R4]     ; Store result in memory
LOOP  B LOOP
```

MULTIPLICATION

```
MUL R3, R0, R1   ; R3 = R0 * R1
STR R3, [R4]     ; Store result in memory
LOOP  B LOOP
```

SUBTRACTION

```
SUB R3, R0, R1   ; R3 = R0 - R1
STR R3, [R4]     ; Store result in memory
LOOP  B LOOP
```

DIVISION

```
        CMP R1, #0       ; Check for divide by zero
        BEQ STOP
        UDIV R3, R0, R1  ; R3 = R0 / R1
        STR R3, [R4]     ; Store result in memory
LOOP    B LOOP
        END
```

Example:1 : Check whether given number is ODD / EVEN

```
AREA  exp1, CODE, READONLY
   ENTRY
        MOV R0, #7        ; Load number 7 into R0
        AND R1, R0, #1
        ; Perform bitwise AND with 1 (if result is 1, it's odd)
        CMP R1, #0        ; Compare result with 0
        BEQ EVEN          ; If result is 0, branch to EVEN label

   ; Odd case
        MOV R3,#1
        LDR R2, =ODD_MSG  ; Load odd message address
        STR r3,[r2]
loop1   B loop1
```

```
 ; even case
EVEN
        MOV R3,#0
        LDR R2, =EVEN_MSG
 ; Load even message address
        STR R3,[r2]
loop    B loop

        AREA Data1, DATA, READWRITE

ODD_MSG   DCB 0
EVEN_MSG   DCB 0

        END
```

Table 4.18 Examples of Arithmetic Instructions

Instruction	Operation

`ADD Rd, Rn, Rm ; Rd = Rn + Rm`
`ADD Rd, Rd, Rm ; Rd = Rd + Rm`
`ADD Rd, #immed ; Rd = Rd + #immed`
`ADD Rd, Rn, # immed ; Rd = Rn + #immed`
	ADD operation

`ADC Rd, Rn, Rm ; Rd = Rn + Rm + carry`
`ADC Rd, Rd, Rm ; Rd = Rd + Rm + carry`
`ADC Rd, #immed ; Rd = Rd + #immed + carry`
	ADD with carry

`ADDW Rd, Rn,#immed ; Rd = Rn + #immed`	ADD register with 12-bit immediate value

`SUB Rd, Rn, Rm ; Rd = Rn – Rm`
`SUB Rd, #immed ; Rd = Rd – #immed`
`SUB Rd, Rn,#immed ; Rd = Rn – #immed`
	SUBTRACT

`SBC Rd, Rm ; Rd = Rd – Rm – borrow`
`SBC.W Rd, Rn, #immed ; Rd = Rn – #immed – borrow`
`SBC.W Rd, Rn, Rm ; Rd = Rn – Rm – borrow`
	SUBTRACT with borrow (not carry)

`RSB.W Rd, Rn, #immed ; Rd = #immed -Rn` `RSB.W Rd, Rn, Rm ; Rd = Rm – Rn`	Reverse subtract
`MUL Rd, Rm ; Rd = Rd * Rm` `MUL.W Rd, Rn, Rm ; Rd = Rn * Rm`	Multiply
`UDIV Rd, Rn, Rm ; Rd = Rn/Rm` `SDIV Rd, Rn, Rm ; Rd = Rn/Rm`	Unsigned and signed divide

Table 4.19 32-Bit Multiply Instructions

Instruction	Operation
`SMULL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} = Rn * Rm` `SMLAL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} += Rn * Rm`	32-bit multiply instructions for signed values
`UMULL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} = Rn * Rm` `UMLAL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} += Rn * Rm`	32-bit multiply instructions for unsigned values

- These instructions can be used with or without the “S” suffix to determine if the APSR should be updated.
- In most cases, if UAL syntax is selected and if “S” suffix is not used, the 32-bit version of the instructions would be selected as most of the 16-bit Thumb instructions update APSR.

Arithmetic Operations

- Eg:1
- ADD R2, R1, R3
- SUBS R8, R6, #240

  ; Sets the flags on the result
- RSB R4, R4, #1280

  ; Subtracts contents of R4 from 1280
- ADCHI R11, R0, R3

  ; Only executed if C flag set and Z flag clear.

=

- Eg:2 - Instructions that add a 64-bit integer contained in R2 and R3 to another 64-bit integer contained in R0 and R1, and place the result in R4 and R5.
- ADDS R4, R0, R2

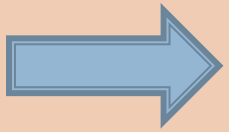
  ; add the least significant words
- ADC R5, R1, R3

  ; add the most significant words with carry

- Eg:3 - Multiword values do not have to use consecutive registers.
- SUBS R6, R6, R9 ; subtract the least significant words
- SBCS R9, R2, R1 ; subtract the middle words with carry
- SBC R2, R8, R11 ; subtract the most significant words with carry

Eg. 1 :- Write an ALP for ARM7 demonstrating the data transfer.

```
        AREA DATATRANSFER, CODE, READONLY
        EXPORT __main
        ENTRY
__main
        LDR R0, =0x40000000
        LDR R1, =0x40000080
        LDR R2, =0x0000000A
UP      LDR R3, [R0]
        STR R3, [R1]
        ADD R0, R0, #4
        ADD R1, R1, #4
        ADD R2, R2, #-1
        CMP R2, #0
        BEQ NEXT
        B UP
NEXT    END
```



Eg2 :- Write an ALP to add two 64 bit numbers

```
        AREA ADDITION, CODE, READONLY
        EXPORT __main
        ENTRY
_main
        LDR R0,=0X1234E640 ;LOAD THE FIRST VALUE IN R0 (MSB),R1(LSB)
        LDR R1,=0X43210010
        LDR R2,=0X12348900 ;LOAD THE SECOND VALUE IN R2 (MSB),R3 (LSB)
        LDR R3,=0X43212102
        ADDS R4,R1,R3 ;RESULT IS STORED IN R4,R5
        ADC R5,R0,R2
        END ;Mark end of file
;VALUE1 0X1234E640 0X43210010 (R0,R1)
;VALUE2 0X12348900 0X43212102 (R2,R3)
;RESULT 0X24696F40 0X86422112 (R5,R4)
```