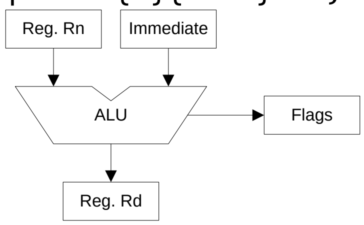
Standard data processing instructions with two sources, one destination

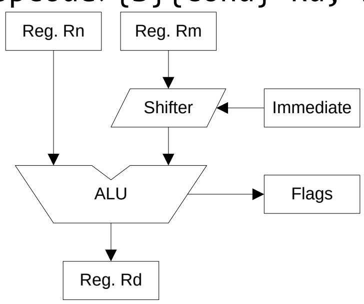
```
{label} <ALU_opcode>{S}{cond} Rd, Rn, <operand2> ; comment
```

variants of <operand2>:

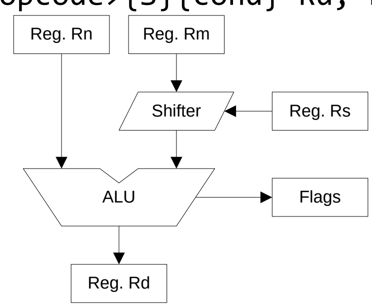
```
<ALU_opcode>{S}{cond} Rd, Rn, #<immediate>
```




```
<ALU_opcode>{S}{cond} Rd, Rn, Rm{, <shifter_opcode> #<immediate>}
```




```
<ALU_opcode>{S}{cond} Rd, Rn, Rm, <shifter_opcode> Rs
```



Category	Operation	ALU opcode	Action
Arithmetic	**Add**	**ADD**	**Rd = Rn + <operand2>;**
	Add with carry	ADC	Rd = Rn + <operand2> + Carry;
	Subtract	**SUB**	**Rd = Rn - <operand2>;**
	Subtract with carry	SBC	Rd = Rn - <operand2> - !Carry;
	Reverse subtract	RSB	Rd = <operand2> - Rn;
Bitwise logical	**AND**	**AND**	**Rd = Rn & <operand2>;**
	Bit clear	BIC	Rd = Rn & ~<operand2>;
	OR	**ORR**	**Rd = Rn | <operand2>;**
	OR NOT	ORN	Rd = Rn | ~<operand2>;
	Exclusive OR	**EOR**	**Rd = Rn ^ <operand2>;**

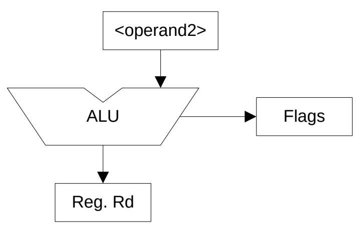
Shifter opcodes

<table>
<tr><th>Operation</th><th>Opcode</th><th>Action</th></tr>
<tr><td>Logical Shift Left</td><td>LSL</td><td><operand2> = Rm << <#shift>;</td></tr>
<tr><td>Logical Shift Right</td><td>LSR</td><td><operand2> = (unsigned)Rm >> <#shift>;</td></tr>
<tr><td>Arithmetic Shift Right</td><td>ASR</td><td><operand2> = (signed)Rm >> <#shift>;</td></tr>
<tr><td>Rotate Right</td><td>ROR</td><td><operand2> = ((unsigned)Rm >> <#shift>) |
(Rm << (32 - <#shift>));</td></tr>
<tr><td>Rotate Right with Extend</td><td>RRX</td><td><operand2> = ((unsigned)Rm >> 1) |
(Carry << 31);
Carry = Rm[0]; // if S suffix present</td></tr>
</table>

Optional features:
{label} = label for branch instructions
{S} = modify condition code bits (flags)
{cond} = condition code

Standard data processing instructions with one source, one destination

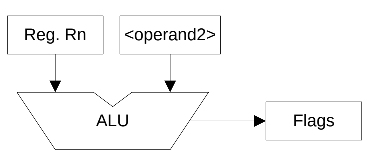
```
{label} <ALU_opcode>{S}{cond} Rd, <operand2> ; comment
```



Category	Operation	Opcode	Action
Data movement	**Copy ("Move")**	**MOV**	**Rd = <operand2>;**
	Move into top	**MOVT**	**Rd[31:16] = <16-bit immediate>;**
Bitwise logical	**NOT**	**MVN**	**Rd = ~<operand2>;**

Standard data processing instructions with two sources, no destination

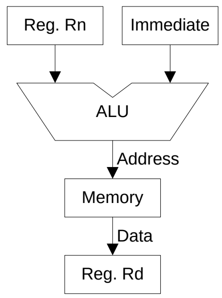
```
{label} <ALU_opcode>{cond} Rn, <operand2> ; comment
```

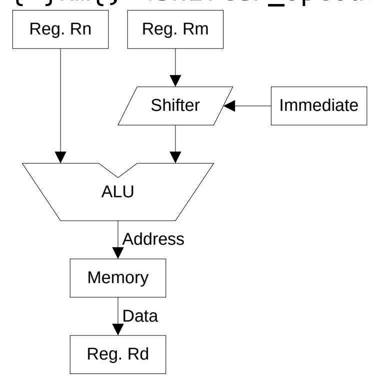


Category	Operation	Opcode	Action
Arithmetic	**Compare (Subtract)**	**CMP**	**Rn - <operand2>; // update flags**
	Compare Negative	CMN	Rn + <operand2>; // update flags
Bitwise logical	**Test (AND)**	**TST**	**Rn & <operand2>; // update flags**
	Test Equivalence (XOR)	TEQ	Rn ^ <operand2>; // update flags

Load data from memory into a register

```
LDR{B|SB|H|SH}{cond} Rd, [Rn{, #<immediate>}]
LDR{B|SB|H|SH}{cond} Rd, [Rn, {-}Rm{, <shifter_opcode> #<immediate>}]
```





- ALU operation is Add; Subtract if minus sign present
- No `{B|SB|H|SH}` suffix = 32-bit word: `Rd = Data;`
- `B` = unsigned byte: `Rd[7:0] = Data; Rd[31:8] = 0;`
- `SB` = signed byte: `Rd[7:0] = Data; Rd[31:8] = Data[7];`
- `H` = unsigned halfword: `Rd[15:0] = Data; Rd[31:16] = 0;`
- `SH` = signed halfword: `Rd[15:0] = Data; Rd[31:16] = Data[15];`

```
ldr  r3, [r0]
     r3 = mem32[r0]; // r3 = memory contents at address r0
```

PC-relative addressing

```
LDR{cond} Rd, <label> ; encoded as LDR{cond} Rd, [PC, #<immed>]
```

- `label` must be close to the current instruction (typ. holds a constant)

Store data from register into memory

```
STR{B|H}{cond} Rd, [Rn{, #<immediate>}]
STR{B|H}{cond} Rd, [Rn, {-}Rm{, <shifter_opcode> #<immediate>}]
```

- Format almost identical to **`LDR`**, but **data transfer direction is reversed**
- Data types
  - No suffix = 32-bit word: `Data = Rd;`
  - `B` = byte: `Data = Rd[7:0];`
  - `H` = halfword: `Data = Rd[15:0];`

Auto-indexing with `LDR`/`STR`

```
LDR Rd, [Rn, <operand2>]!    ; pre-indexed
   { Rn += <operand2>;
   { Rd = mem32[Rn];

LDR Rd, [Rn], <operand2>     ; post-indexed
   { Rd = mem32[Rn];
   { Rn += <operand2>;
```

Multiple register data transfer instructions

```
LDM{IA|FD}{cond} Rn, <registers>
```

- Load multiple registers listed in {} from memory starting at the address Rn
- Transfer order: **increasing register number** = **increasing memory address**

example:

```
ldm r2, {r5-r7, r0}
    r0 = mem32[r2];
    r5 = mem32[r2 + 4];
    r6 = mem32[r2 + 8];
    r7 = mem32[r2 + 12];
```

```
STM{IA|EA}{cond} Rn, <registers>
```

- Nearly identical to `LDM`, but the data transfer direction is reversed: registers to memory

```
STMDB{cond} Rn, <registers>
STMFD{cond} Rn, <registers>
```

- Decrement Before addressing mode
- Used to implement stack PUSH

example:

```
stmfd r2, {r5-r7, r0}
    mem32[r2 - 16] = r0;
    mem32[r2 - 12] = r5;
    mem32[r2 - 8] = r6;
    mem32[r2 - 4] = r7;
```

Auto-indexing with `LDM`/`STM`

- Add “!” after the address register `Rn` to have it updated

```
ldmia r6!, {r1, r2}
    r1 = mem32[r6];
    r2 = mem32[r6 + 4];
    r6 += 8;
```

```
stmdb r6!, {r1, r2}
    r6 -= 8;
    mem32[r6] = r1;
    mem32[r6 + 4] = r2;
```

Stack operations

```
PUSH <reglist>
```

same as

```
STMFD SP!, <reglist>
```

```
POP <reglist>
```

same as

```
LDMFD SP!, <reglist>
```

Branch

```
B{cond} <label>
BX{cond} Rm
```

```
if (r3 > 10) {  // r3 is signed
    r3 = r0;
}
else {
    r3++;
}
```

```
        cmp   r3, #10
        ble   else1
        mov   r3, r0
        b     endif1
else1   add   r3, #1
endif1
```

Condition Field {cond}	
Mnemonic	**Description**
EQ	Equal
NE	Not equal
CS / HS	Carry Set / Unsigned higher or same
CC / LO	Carry Clear / Unsigned lower
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always (normally omitted)

Subroutine call

```
BL{cond} <label>
BLX{cond} Rm
```

- Save the **address of the next instruction** into **LR**
- Instruction to **return from the subroutine:**

```
BX LR
```

Subroutine example

```
      bl    subr       ; call subroutine
      add   r1, r0
      ...

subr  mov   r0, #20    ; place return value in r0
      bx    lr         ; return from subroutine
```

Register conventions in C functions

Register Type	Description
Argument register	Passes arguments during a function call
Return register	Holds the return value from a function call
Expression register	Holds a value
Stack pointer	Holds the address of the top of the software stack
Link register	Contains the return address of a function call
Program counter	Contains the current address of code being executed

Register	Alias	Usage	Preserved by Function[a]
`R0`	`A1`	Argument register, return register, expression register	Parent
`R1`	`A2`	Argument register, return register, expression register	Parent
`R2`	`A3`	Argument register, expression register	Parent
`R3`	`A4`	Argument register, expression register	Parent
`R4`	`V1`	Expression register	Child
`R5`	`V2`	Expression register	Child
`R6`	`V3`	Expression register	Child
`R7`	`V4`	Expression register	Child
`R8`	`V5`	Expression register	Child
`R9`	`V6`	Expression register	Child
`R10`	`V7`	Expression register	Child
`R11`	`V8`	Expression register	Child
`R12`	`V9, IP`	Expression register, Intra-Procedure-call scratch register	Parent
`R13`	`SP`	Stack pointer	Child[b]
`R14`	`LR`	Link register, expression register	See note
`R15`	`PC`	Program counter	N/A

[a] The parent function refers to the function making the function call. The child function refers to the function being called.

[b] The `SP` is preserved by the convention that everything pushed on the stack is popped off before returning.

- Function arguments are passed in `R0-R3`, then on the stack
- Function return value is placed in `R0`
- `R4-R11` must be pushed on the stack before using them in the function
    - Pop them back when returning
- `LR` must be pushed on the stack before making other function calls
    - The value of `LR` may be popped directly into `PC` to return from the function
    - After its value is saved on the stack, `LR` may be used as an expression register
- `R0-R3`, `R12` and `LR` are assumed overwritten after a function call
- Never store anything at addresses below `SP` (beyond the top of stack)
    - ISRs can use this area

ARM Cortex-M4 CPU Performance

Instruction type	Clock cycles
Data operations	1 (+P[a] if PC is destination)
MUL	1
MLA, MLS	2
Divide	2 to 12
LDR/STR	1 or 2[b] (+P[a] if PC is destination)
LDM/STM	1+N[b] (+P[a] if PC is destination)
Branch	1+P[a]

Cycle count information:

- P = pipeline reload cycles
- N = number of registers to be loaded or stored

[a] Branches take one cycle for instruction and then pipeline reload for target instruction.

- Non-taken branches are 1 cycle total.
- Taken branches with an immediate (label) operand are normally 1 cycle of pipeline reload (2 cycles total).
- Taken branches with register operand are normally 2 cycles of pipeline reload (3 cycles total).
- Pipeline reload is longer when branching to unaligned 32-bit instructions in addition to accesses to **slower memory**.

[b] Load and store instructions are subject to these rules:

- STR with immediate or no offset is always 1 cycle.
- LDR/STR instruction sequences are **pipelined** as long as the data of the previous LDR is not the address of the next LDR/STR
  - o 2 cycles for the first LDR
  - o 1 cycle each subsequent LDR/STR
- STR with register offset is 2 cycles (pipelining after another LDR/STR reduces this to 1 cycle) and the **next** LDR/STR is not pipelined.
- LDM and STM cannot be pipelined with other load/store instructions.
  - o The multiple loads/stores in the same instruction are pipelined
  - o LDM and STM are interruptible instructions
- **Unaligned** Word or Halfword loads or stores add penalty cycles.
  - o Halfword-aligned word or byte-aligned halfword = 1 additional cycle
  - o Byte-aligned word = 2 additional cycles
- Memory accesses may stall with slower memory: additional penalty cycles added