ARM Assembler

Logic

Logical Operations

For Boolean Operation

False: 0 (0x00000000)

True: -1 (0xFFFFFFFF)

`AND`⟨*cc*⟩⟨*S*⟩ Rd, Rn, ⟨*op1*⟩ `ORR`⟨*cc*⟩⟨*S*⟩ Rd, Rn, ⟨*op1*⟩

`MVN`⟨*cc*⟩⟨*S*⟩ Rd, ⟨*op1*⟩ `EOR`⟨*cc*⟩⟨*S*⟩ Rd, Rn, ⟨*op1*⟩

Logical Operations

For Boolean Operation

False: 0 (0x00000000)

True: -1 (0xFFFFFFFF)

AND⟨*cc*⟩⟨*S*⟩ Rd, Rn, ⟨*op1*⟩ ORR⟨*cc*⟩⟨*S*⟩ Rd, Rn, ⟨*op1*⟩

⟨*cc*⟩: ALU $\leftarrow$ R$n \wedge$ ⟨*op1*⟩

MVN⟨*cc*⟩⟨*S*⟩ Rd, ⟨*op1*⟩ EOR⟨*cc*⟩⟨*S*⟩ Rd, Rn, ⟨*op1*⟩

Logical Operations

For Boolean Operation

False: 0 (0x00000000)

True: -1 (0xFFFFFFFF)

`AND`$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

$\langle cc \rangle$: ALU $\leftarrow$ R$n \wedge \langle op1 \rangle$

$\langle cc \rangle$: Rd $\leftarrow$ ALU

`ORR`$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

`MVN`$\langle cc \rangle\langle S \rangle$ Rd, $\langle op1 \rangle$

`EOR`$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

Logical Operations

For Boolean Operation

False: 0 (0x00000000)

True: -1 (0xFFFFFFFF)

`AND`$\langle cc \rangle\langle S \rangle$ `R`d`,` `R`n`,` $\langle op1 \rangle$ `ORR`$\langle cc \rangle\langle S \rangle$ `R`d`,` `R`n`,` $\langle op1 \rangle$

$\langle cc \rangle$: ALU $\leftarrow$ R$n \wedge \langle op1 \rangle$

$\langle cc \rangle$: Rd $\leftarrow$ ALU

$\langle S \rangle\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)

`MVN`$\langle cc \rangle\langle S \rangle$ `R`d`,` $\langle op1 \rangle$ `EOR`$\langle cc \rangle\langle S \rangle$ `R`d`,` `R`n`,` $\langle op1 \rangle$

Logical Operations

For Boolean Operation

False: 0 (0x00000000)

True: -1 (0xFFFFFFFF)

AND$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

$\langle cc \rangle$: ALU $\leftarrow$ R$n \wedge \langle op1 \rangle$

$\langle cc \rangle$: Rd $\leftarrow$ ALU

$\langle S \rangle\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)

ORR$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

$\langle cc \rangle$: ALU $\leftarrow$ R$n \vee \langle op1 \rangle$

$\langle cc \rangle$: Rd $\leftarrow$ ALU

$\langle S \rangle\langle cc \rangle$: CPSR $\leftarrow$ ALU(Flags)

MVN$\langle cc \rangle\langle S \rangle$ Rd, $\langle op1 \rangle$

EOR$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

Logical Operations

For Boolean Operation

False: 0 (0x00000000)

True: -1 (0xFFFFFFFF)

AND$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

$\langle cc \rangle$: ALU $\leftarrow$ R$n \wedge \langle op1 \rangle$

$\langle cc \rangle$: Rd $\leftarrow$ ALU

$\langle S \rangle\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)

ORR$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

$\langle cc \rangle$: ALU $\leftarrow$ R$n \vee \langle op1 \rangle$

$\langle cc \rangle$: Rd $\leftarrow$ ALU

$\langle S \rangle\langle cc \rangle$: CPSR $\leftarrow$ ALU(Flags)

MVN$\langle cc \rangle\langle S \rangle$ Rd, $\overline{\langle op1 \rangle}$

$\langle cc \rangle$: ALU $\leftarrow \overline{\langle op1 \rangle}$

$\langle cc \rangle$: Rd $\leftarrow$ ALU

$\langle S \rangle\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)

EOR$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

Logical Operations

For Boolean Operation

False: 0 (0x00000000)

True: -1 (0xFFFFFFFF)

AND$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

$\langle cc \rangle$: ALU $\leftarrow$ R$n \wedge \langle op1 \rangle$

$\langle cc \rangle$: Rd $\leftarrow$ ALU

$\langle S \rangle\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)

ORR$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

$\langle cc \rangle$: ALU $\leftarrow$ R$n \vee \langle op1 \rangle$

$\langle cc \rangle$: Rd $\leftarrow$ ALU

$\langle S \rangle\langle cc \rangle$: CPSR $\leftarrow$ ALU(Flags)

MVN$\langle cc \rangle\langle S \rangle$ Rd, $\langle op1 \rangle$

$\langle cc \rangle$: ALU $\leftarrow$ $\overline{\langle op1 \rangle}$

$\langle cc \rangle$: Rd $\leftarrow$ ALU

$\langle S \rangle\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)

EOR$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

$\langle cc \rangle$: ALU $\leftarrow$ R$n \oplus \langle op1 \rangle$

$\langle cc \rangle$: Rd $\leftarrow$ ALU

$\langle S \rangle\langle cc \rangle$: CPSR $\leftarrow$ ALU(Flags)

Binary Operations

Use logical operators to manipulate bits within a value

`AND`	And	Clear bits down to zero	$\wedge$
`ORR`	Or	Set bits to one	$\vee$
`EOR`	Exclusive Or	Toggle bits	$\oplus$
`MVN`	Not	Toggle all bits	$\overline{x}$

In the following R1 has the value 00111100 (&3C)

Instruction	*Operation*
`MOV r1, #&3C`	
`AND r0, r1, #&0F`	$00111100 \wedge 00001111 =$
`ORR r0, r1, #&0F`	$00111100 \vee 00001111 =$
`EOR r0, r1, #&0F`	$00111100 \oplus 00001111 =$
`MVN r0, r1`	$\overline{00111100} =$

Binary Operations

Use logical operators to manipulate bits within a value

`AND`	And	Clear bits down to zero	$\wedge$
`ORR`	Or	Set bits to one	$\vee$
`EOR`	Exclusive Or	Toggle bits	$\oplus$
`MVN`	Not	Toggle all bits	$\overline{x}$

In the following R1 has the value 00111100 (&3C)

Instruction	*Operation*
`MOV r1, #&3C`	
`AND r0, r1, #&0F`	00111100 $\wedge$ 00001111 = 00001100 (&0C)
`ORR r0, r1, #&0F`	00111100 $\vee$ 00001111 =
`EOR r0, r1, #&0F`	00111100 $\oplus$ 00001111 =
`MVN r0, r1`	$\overline{00111100}$ =

Binary Operations

Use logical operators to manipulate bits within a value

`AND`	And	Clear bits down to zero	$\wedge$
`ORR`	Or	Set bits to one	$\vee$
`EOR`	Exclusive Or	Toggle bits	$\oplus$
`MVN`	Not	Toggle all bits	$\overline{x}$

In the following R1 has the value 00111100 (&3C)

Instruction	*Operation*
`MOV r1, #&3C`	
`AND r0, r1, #&0F`	$00111100 \wedge 00001111 = 00001100$ (&0C)
`ORR r0, r1, #&0F`	$00111100 \vee 00001111 = 00111111$ (&3F)
`EOR r0, r1, #&0F`	$00111100 \oplus 00001111 =$
`MVN r0, r1`	$\overline{00111100} =$

Binary Operations

Use logical operators to manipulate bits within a value

`AND`	And	Clear bits down to zero	$\wedge$
`ORR`	Or	Set bits to one	$\vee$
`EOR`	Exclusive Or	Toggle bits	$\oplus$
`MVN`	Not	Toggle all bits	$\overline{x}$

In the following R1 has the value 00111100 (&3C)

Instruction	*Operation*
`MOV r1, #&3C`	
`AND r0, r1, #&0F`	00111100 $\wedge$ 00001111 = 00001100 (&0C)
`ORR r0, r1, #&0F`	00111100 $\vee$ 00001111 = 00111111 (&3F)
`EOR r0, r1, #&0F`	00111100 $\oplus$ 00001111 = 00110011 (&33)
`MVN r0, r1`	$\overline{00111100}$ =

Binary Operations

Use logical operators to manipulate bits within a value

`AND`	And	Clear bits down to zero	$\wedge$
`ORR`	Or	Set bits to one	$\vee$
`EOR`	Exclusive Or	Toggle bits	$\oplus$
`MVN`	Not	Toggle all bits	$\overline{x}$

In the following R1 has the value 00111100 (&3C)

Instruction	*Operation*
`MOV r1, #&3C`	
`AND r0, r1, #&0F`	00111100 $\wedge$ 00001111 = 00001100 (&0C)
`ORR r0, r1, #&0F`	00111100 $\vee$ 00001111 = 00111111 (&3F)
`EOR r0, r1, #&0F`	00111100 $\oplus$ 00001111 = 00110011 (&33)
`MVN r0, r1`	$\overline{00111100}$ = 11000011 (&C3)

Program: nibble.s

```
1.   ; Disassemble a byte into its high and low order nibbles
7. Main
8.          LDR     R1, Value           ; Load value to be disassembled
9.          LDR     R2, Mask            ; Load the bitmask
10.         MOV     R3, R1, LSR #0x4    ; Copy high order nibble into R3
11.         MOV     R3, R3, LSL #0x8    ; Now left shift it one byte
12.         AND     R1, R1, R2          ; AND number with bitmask
13.         ADD     R1, R1, R3          ; Add the result of that to
14.                                     ; What we moved into R3
15.         STR     R1, Result          ; Store the result
16.         SWI     &11
17.
18. Value   DCB     &5F                 ; Value to be shifted
19.         ALIGN
20. Mask    DCW     &000F               ; Bitmask = %. . . 0001111
21.         ALIGN
22. Result  DCD     0                   ; Space to store result
```

Program: nibble.s

```
1.   ; Disassemble a byte into its high and low order nibbles
7. Main
8.          LDR     R1, Value           ; Load value to be disassembled
9.          LDR     R2, Mask            ; Load the bitmask
10.         MOV     R3, R1, LSR #0x4    ; Copy high order nibble into R3
11.         MOV     R3, R3, LSL #0x8    ; Now left shift it one byte
12.         AND     R1, R1, R2          ; AND number with bitmask
13.         ADD     R1, R1, R3          ; Add the result of that to
14.                                     ; What we moved into R3
15.         STR     R1, Result          ; Store the result
16.         SWI     &11
17.
18. Value   DCB     &5F                 ; Value to be shifted
19.         ALIGN
20. Mask    DCW     &000F               ; Bitmask = %...0001111
21.         ALIGN
22. Result  DCD     0                   ; Space to store result
```

ALIGN Make sure next value is 32-bit aligned

Program: nibble.s

```
1.   ; Disassemble a byte into its high and low order nibbles
7. Main
8.          LDR     R1, Value           ; Load value to be disassembled
9.          LDR     R2, Mask            ; Load the bitmask
10.         MOV     R3, R1, LSR #0x4    ; Copy high order nibble into R3
11.         MOV     R3, R3, LSL #0x8    ; Now left shift it one byte
12.         AND     R1, R1, R2          ; AND number with bitmask
13.         ADD     R1, R1, R3          ; Add the result of that to
14.                                     ; What we moved into R3
15.         STR     R1, Result          ; Store the result
16.         SWI     &11
17.
18. Value   DCB     &5F                 ; Value to be shifted
19.         ALIGN
20. Mask    DCW     &000F               ; Bitmask = %...0001111
21.         ALIGN
22. Result  DCD     0                   ; Space to store result
```

LDR Read Byte to Split R1 ← 0x5F

Should be LDRB

Program: nibble.s

```
1.   ; Disassemble a byte into its high and low order nibbles
7. Main
8.          LDR     R1, Value           ; Load value to be disassembled
9.          LDR     R2, Mask            ; Load the bitmask
10.         MOV     R3, R1, LSR #0x4    ; Copy high order nibble into R3
11.         MOV     R3, R3, LSL #0x8    ; Now left shift it one byte
12.         AND     R1, R1, R2          ; AND number with bitmask
13.         ADD     R1, R1, R3          ; Add the result of that to
14.                                     ; What we moved into R3
15.         STR     R1, Result          ; Store the result
16.         SWI     &11
17.
18. Value   DCB     &5F                 ; Value to be shifted
19.         ALIGN
20. Mask    DCW     &000F               ; Bitmask = %...0001111
21.         ALIGN
22. Result  DCD     0                   ; Space to store result
```

LDR Read Bit Mask R2 ← 0x0F

Program: nibble.s

```
 1.   ; Disassemble a byte into its high and low order nibbles
 7. Main
 8.         LDR     R1, Value           ; Load value to be disassembled
 9.         LDR     R2, Mask            ; Load the bitmask
10.         MOV     R3, R1, LSR #0x4    ; Copy high order nibble into R3
11.         MOV     R3, R3, LSL #0x8    ; Now left shift it one byte
12.         AND     R1, R1, R2          ; AND number with bitmask
13.         ADD     R1, R1, R3          ; Add the result of that to
14.                                     ; What we moved into R3
15.         STR     R1, Result          ; Store the result
16.         SWI     &11
17.
18. Value   DCB     &5F                 ; Value to be shifted
19.         ALIGN
20. Mask    DCW     &000F               ; Bitmask = %...0001111
21.         ALIGN
22. Result  DCD     0                   ; Space to store result
```

LSR Shift R1 right by 4 bits $R3 \leftarrow 0x5B \gg 4$ (0x05)

Program: nibble.s

```
 1.   ; Disassemble a byte into its high and low order nibbles
 7. Main
 8.         LDR     R1, Value           ; Load value to be disassembled
 9.         LDR     R2, Mask            ; Load the bitmask
10.         MOV     R3, R1, LSR #0x4    ; Copy high order nibble into R3
11.         MOV     R3, R3, LSL #0x8    ; Now left shift it one byte
12.         AND     R1, R1, R2          ; AND number with bitmask
13.         ADD     R1, R1, R3          ; Add the result of that to
14.                                     ; What we moved into R3
15.         STR     R1, Result          ; Store the result
16.         SWI     &11
17.
18. Value   DCB     &5F                 ; Value to be shifted
19.         ALIGN
20. Mask    DCW     &000F               ; Bitmask = %...0001111
21.         ALIGN
22. Result  DCD     0                   ; Space to store result
```

LSL Shift R3 left by 8 bits $R3 \leftarrow 0x05 \ll 8$ (0x500)

Program: nibble.s

```
 1.   ; Disassemble a byte into its high and low order nibbles
 7. Main
 8.         LDR     R1, Value           ; Load value to be disassembled
 9.         LDR     R2, Mask            ; Load the bitmask
10.         MOV     R3, R1, LSR #0x4    ; Copy high order nibble into R3
11.         MOV     R3, R3, LSL #0x8    ; Now left shift it one byte
12.         AND     R1, R1, R2          ; AND number with bitmask
13.         ADD     R1, R1, R3          ; Add the result of that to
14.                                     ; What we moved into R3
15.         STR     R1, Result          ; Store the result
16.         SWI     &11
17.
18. Value   DCB     &5F                 ; Value to be shifted
19.         ALIGN
20. Mask    DCW     &000F               ; Bitmask = %...0001111
21.         ALIGN
22. Result  DCD     0                   ; Space to store result
```

AND Mask out upper 4-bits $R1 \leftarrow 0x5B \wedge 0x0F$ (0x0B)

Program: nibble.s

```
 1.   ; Disassemble a byte into its high and low order nibbles
 7. Main
 8.         LDR     R1, Value           ; Load value to be disassembled
 9.         LDR     R2, Mask            ; Load the bitmask
10.         MOV     R3, R1, LSR #0x4    ; Copy high order nibble into R3
11.         MOV     R3, R3, LSL #0x8    ; Now left shift it one byte
12.         AND     R1, R1, R2          ; AND number with bitmask
13.         ADD     R1, R1, R3          ; Add the result of that to
14.                                     ; What we moved into R3
15.         STR     R1, Result          ; Store the result
16.         SWI     &11
17.
18. Value   DCB     &5F                 ; Value to be shifted
19.         ALIGN
20. Mask    DCW     &000F               ; Bitmask = %...0001111
21.         ALIGN
22. Result  DCD     0                   ; Space to store result
```

ADD Add lower nibble back in R3 ← 0x500 + 0x0B (0x50B)

We could also have written: `ORR R3, R1, R3`

Program: nibble.s

```
 1.   ; Disassemble a byte into its high and low order nibbles
 7. Main
 8.         LDR     R1, Value           ; Load value to be disassembled
 9.         LDR     R2, Mask            ; Load the bitmask
10.         MOV     R3, R1, LSR #0x4    ; Copy high order nibble into R3
11.         MOV     R3, R3, LSL #0x8    ; Now left shift it one byte
12.         AND     R1, R1, R2          ; AND number with bitmask
13.         ADD     R1, R1, R3          ; Add the result of that to
14.                                     ; What we moved into R3
15.         STR     R1, Result          ; Store the result
16.         SWI     &11
17.
18. Value   DCB     &5F                 ; Value to be shifted
19.         ALIGN
20. Mask    DCW     &000F               ; Bitmask = %...0001111
21.         ALIGN
22. Result  DCD     0                   ; Space to store result
```

ADD/LSL Merge LSL with ADD $R1 \leftarrow R1 + (R3 \ll 8)$

```
ADD R1, R1, R3 LSL #8
```

Comparison

- Compare two values, setting the flags accordingly
- Only the flags are changed, value of Rs does not change
- Compare two values: difference (effects N, Z, V, and C flags)
  `CMP`⟨*cc*⟩ Rn`,` ⟨*op1*⟩
- Test Equal: are two values equal (only effects N and Z flags)
  `TEQ`⟨*cc*⟩ Rn`,` ⟨*op1*⟩
- Test Bits: are specific bits set (only effects N and Z flags)
  `TST`⟨*cc*⟩ Rn`,` ⟨*op1*⟩

Comparison

- Compare two values, setting the flags accordingly
- Only the flags are changed, value of Rs does not change
- Compare two values: difference (effects N, Z, V, and C flags)
  `CMP`$\langle cc \rangle$ Rn`,` $\langle op1 \rangle$ $\langle cc \rangle$: ALU $\leftarrow$ Rn $-$ $\langle op1 \rangle$
  $\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)
- Test Equal: are two values equal (only effects N and Z flags)
  `TEQ`$\langle cc \rangle$ Rn`,` $\langle op1 \rangle$
- Test Bits: are specific bits set (only effects N and Z flags)
  `TST`$\langle cc \rangle$ Rn`,` $\langle op1 \rangle$

Comparison

- Compare two values, setting the flags accordingly
- Only the flags are changed, value of Rs does not change
- Compare two values: difference (effects N, Z, V, and C flags)

  `CMP`$\langle$*cc*$\rangle$ Rn, $\langle$*op1*$\rangle$ $\langle$*cc*$\rangle$: ALU $\leftarrow$ R$n - \langle$*op1*$\rangle$

  $\langle$*cc*$\rangle$: CSPR $\leftarrow$ ALU(Flags)
- Test Equal: are two values equal (only effects N and Z flags)

  `TEQ`$\langle$*cc*$\rangle$ Rn, $\langle$*op1*$\rangle$ $\langle$*cc*$\rangle$: ALU $\leftarrow$ R$n \oplus \langle$*op1*$\rangle$

  $\langle$*cc*$\rangle$: CSPR $\leftarrow$ ALU(Flags)
- Test Bits: are specific bits set (only effects N and Z flags)

  `TST`$\langle$*cc*$\rangle$ Rn, $\langle$*op1*$\rangle$

Comparison

- Compare two values, setting the flags accordingly
- Only the flags are changed, value of Rs does not change
- Compare two values: difference (effects N, Z, V, and C flags)

  `CMP`$\langle cc \rangle$ Rn`,` $\langle op1 \rangle$ $\langle cc \rangle$: ALU $\leftarrow$ R$n - \langle op1 \rangle$

  $\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)
- Test Equal: are two values equal (only effects N and Z flags)

  `TEQ`$\langle cc \rangle$ Rn`,` $\langle op1 \rangle$ $\langle cc \rangle$: ALU $\leftarrow$ R$n \oplus \langle op1 \rangle$

  $\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)
- Test Bits: are specific bits set (only effects N and Z flags)

  `TST`$\langle cc \rangle$ Rn`,` $\langle op1 \rangle$ $\langle cc \rangle$: ALU $\leftarrow$ R$n \wedge \langle op1 \rangle$

  $\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)

Flow Control

- Change location of next instruction
  relative to the current instruction
- Assembler calculates the offset from target for us
- Target may be $\pm$32MB from current instruction
- Unconditional Branch
  BAL $\langle label \rangle$ PC $\leftarrow$ PC + 8 + IR(offset)
- Conditional Branch
  B$\langle cc \rangle$ $\langle label \rangle$ $\langle cc \rangle$: PC $\leftarrow$ PC + 8 + IR(offset)

Condition Codes: ⟨cc⟩

General

AL	*Al*ways		*Al*ways
CS	*C*arry *S*et	CC	*C*arry *C*lear
EQ	*Eq*ual (Zero Set)	NE	*N*ot *E*qual (Zero Clear)
VS	O*v*erflow *S*et	VC	O*v*erflow *C*lear

Signed Numbers

GT	*G*reater *T*han	LT	*L*ess *T*han
GE	*G*reater Than or *E*qual	LE	*L*ess Than or *E*qual
PL	*Pl*us (Positive)	MI	*Mi*nus (Negative)

Unsigned Numbers

HI	*Hi*gher Than	LO	*Lo*wer Than (aka CC)
HS	*H*igher or *S*ame (aka CS)	LS	*L*ower or *S*ame

Program: bigger.s

```
1.  ; Find the larger of two numbers
2.
7.  Main
8.          LDR     R1, Value1  ; Load the first value to be compared
9.          LDR     R2, Value2  ; Load the second value to be compared
10.         CMP     R1, R2      ; Compare them
11.         BHI     Done        ; If R1 contains the highest
12.         MOV     R1, R2      ; otherwise overwrite R1
13. Done
14.         STR     R1, Result  ; Store the result
15.         SWI     &11
16.
17. Value1 DCD     &12345678 ; Value to be compared
18. Value2 DCD     &87654321 ; Value to be compared
19. Result  DCD     0         ; Space to store result
```

Program: bigger.s

```
1.  ; Find the larger of two numbers
2.
7.  Main
8.          LDR     R1, Value1  ; Load the first value to be compared
9.          LDR     R2, Value2  ; Load the second value to be compared
10.         CMP     R1, R2      ; Compare them
11.         BHI     Done        ; If R1 contains the highest
12.         MOV     R1, R2      ; otherwise overwrite R1
13. Done
14.         STR     R1, Result  ; Store the result
15.         SWI     &11
16.
17. Value1  DCD     &12345678   ; Value to be compared
18. Value2  DCD     &87654321   ; Value to be compared
19. Result  DCD     0           ; Space to store result
```

ALIGN — No `ALIGN` necessary as `DCD` is used

Program: bigger.s

```
 1.  ; Find the larger of two numbers
 2.
 7.  Main
 8.          LDR     R1, Value1  ; Load the first value to be compared
 9.          LDR     R2, Value2  ; Load the second value to be compared
10.          CMP     R1, R2      ; Compare them
11.          BHI     Done        ; If R1 contains the highest
12.          MOV     R1, R2      ; otherwise overwrite R1
13.  Done
14.          STR     R1, Result  ; Store the result
15.          SWI     &11
16.
17.  Value1 DCD     &12345678 ; Value to be compared
18.  Value2 DCD     &87654321 ; Value to be compared
19.  Result  DCD     0         ; Space to store result
```

CMP Compare R1 with R2 ALU ← R1 - R2

Program: bigger.s

```
 1.  ; Find the larger of two numbers
 2.
 7.  Main
 8.          LDR     R1, Value1  ; Load the first value to be compared
 9.          LDR     R2, Value2  ; Load the second value to be compared
10.          CMP     R1, R2      ; Compare them
11.          BHI     Done        ; If R1 contains the highest
12.          MOV     R1, R2      ; otherwise overwrite R1
13.  Done
14.          STR     R1, Result  ; Store the result
15.          SWI     &11
16.
17.  Value1 DCD     &12345678 ; Value to be compared
18.  Value2 DCD     &87654321 ; Value to be compared
19.  Result  DCD     0         ; Space to store result
```

BHI Branch if R1 higher than R2 HI: PC $\leftarrow$ Done

Program: bigger.s

```
1.  ; Find the larger of two numbers
2.
7.  Main
8.          LDR     R1, Value1  ; Load the first value to be compared
9.          LDR     R2, Value2  ; Load the second value to be compared
10.         CMP     R1, R2      ; Compare them
11.         BHI     Done        ; If R1 contains the highest
12.         MOV     R1, R2      ; otherwise overwrite R1
13. Done
14.         STR     R1, Result  ; Store the result
15.         SWI     &11
16.
17. Value1  DCD     &12345678   ; Value to be compared
18. Value2  DCD     &87654321   ; Value to be compared
19. Result  DCD     0           ; Space to store result
```

Done Define label (*target*) for branch

Program: bigger.s

```
1.  ; Find the larger of two numbers
2.
7.  Main
8.          LDR     R1, Value1  ; Load the first value to be compared
9.          LDR     R2, Value2  ; Load the second value to be compared
10.         CMP     R1, R2      ; Compare them
11.         BHI     Done        ; If R1 contains the highest
12.         MOV     R1, R2      ; otherwise overwrite R1
13. Done
14.         STR     R1, Result  ; Store the result
15.         SWI     &11
16.
17. Value1  DCD     &12345678 ; Value to be compared
18. Value2  DCD     &87654321 ; Value to be compared
19. Result  DCD     0         ; Space to store result
```

MOV

Using Conditional Execution we could replace this with

`MOVLO R1, R2` LO: R2 ← R1

Does not flush the pipeline so faster than `BHI`

Advanced Arithmetic

- Used to calculate values larger than 32-Bits
- Split value into 32-bit sections
  Start with the least signification section and work up to the most signification section, using the Carry to bridge sections
- Add with Carry
  ADC⟨*cc*⟩⟨*S*⟩ Rd, Rn, ⟨*op1*⟩

- Subtract with Carry
  SBC⟨*cc*⟩⟨*S*⟩ Rd, Rn, ⟨*op1*⟩

Advanced Arithmetic

- Used to calculate values larger than 32-Bits
- Split value into 32-bit sections
  Start with the least signification section and work up to the most signification section, using the Carry to bridge sections
- Add with Carry

  ADC$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

  $\langle cc \rangle$: ALU $\leftarrow$ Rn + $\langle op1 \rangle$ + CSPR(C)
  $\langle cc \rangle$: Rd $\leftarrow$ ALU
  $\langle S \rangle\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)
- Subtract with Carry

  SBC$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

Advanced Arithmetic

- Used to calculate values larger than 32-Bits
- Split value into 32-bit sections
  Start with the least signification section and work up to the most signification section, using the Carry to bridge sections
- Add with Carry

  ADC$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

  $\langle cc \rangle$: ALU $\leftarrow$ Rn + $\langle op1 \rangle$ + CSPR(C)
  $\langle cc \rangle$: Rd $\leftarrow$ ALU
  $\langle S \rangle\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)
- Subtract with Carry

  SBC$\langle cc \rangle\langle S \rangle$ Rd, Rn, $\langle op1 \rangle$

  $\langle cc \rangle$: ALU $\leftarrow$ (Rn − $\langle op1 \rangle$) − CSPR(C)
  $\langle cc \rangle$: Rd $\leftarrow$ ALU
  $\langle S \rangle\langle cc \rangle$: CSPR $\leftarrow$ ALU(Flags)

Program: add64.s

```
 7.  Main
 8.          LDR     R0, =Value1 ; Pointer to first value
 9.          LDR     R1, [R0]    ; Load first part of value1
10.          LDR     R2, [R0, #4] ; Load lower part of value1
11.          LDR     R0, =Value2 ; Pointer to second value
12.          LDR     R3, [R0]    ; Load upper part of value2
13.          LDR     R4, [R0, #4] ; Load lower part of value2
14.          ADDS    R6, R2, R4  ; Add lower 4 bytes and set carry flag
15.          ADC     R5, R1, R3  ; Add upper 4 bytes including carry
16.          LDR     R0, =Result ; Pointer to Result
17.          STR     R5, [R0]    ; Store upper part of result
18.          STR     R6, [R0, #4] ; Store lower part of result
19.          SWI     &11
20.
21.  Value1 DCD     &12A2E640, &F2100123 ; Value to be added
22.  Value2 DCD     &001019BF, &40023F51 ; Value to be added
23.  Result  DCD     0           ; Space to store result
```

Program: add64.s

```
 7.  Main
 8.          LDR     R0, =Value1 ; Pointer to first value
 9.          LDR     R1, [R0]    ; Load first part of value1
10.          LDR     R2, [R0, #4] ; Load lower part of value1
11.          LDR     R0, =Value2 ; Pointer to second value
12.          LDR     R3, [R0]    ; Load upper part of value2
13.          LDR     R4, [R0, #4] ; Load lower part of value2
14.          ADDS    R6, R2, R4  ; Add lower 4 bytes and set carry flag
15.          ADC     R5, R1, R3  ; Add upper 4 bytes including carry
16.          LDR     R0, =Result ; Pointer to Result
17.          STR     R5, [R0]    ; Store upper part of result
18.          STR     R6, [R0, #4] ; Store lower part of result
19.          SWI     &11
20.
21.  Value1 DCD     &12A2E640, &F2100123 ; Value to be added
22.  Value2 DCD     &001019BF, &40023F51 ; Value to be added
23.  Result  DCD     0           ; Space to store result
```

=label Load Address of *label* not value at *label*

Program: add64.s

```
 7.  Main
 8.          LDR     R0, =Value1 ; Pointer to first value
 9.          LDR     R1, [R0]    ; Load first part of value1
10.          LDR     R2, [R0, #4] ; Load lower part of value1
11.          LDR     R0, =Value2 ; Pointer to second value
12.          LDR     R3, [R0]    ; Load upper part of value2
13.          LDR     R4, [R0, #4] ; Load lower part of value2
14.          ADDS    R6, R2, R4  ; Add lower 4 bytes and set carry flag
15.          ADC     R5, R1, R3  ; Add upper 4 bytes including carry
16.          LDR     R0, =Result ; Pointer to Result
17.          STR     R5, [R0]    ; Store upper part of result
18.          STR     R6, [R0, #4] ; Store lower part of result
19.          SWI     &11
20.
21.  Value1 DCD     &12A2E640, &F2100123 ; Value to be added
22.  Value2 DCD     &001019BF, &40023F51 ; Value to be added
23.  Result  DCD     0          ; Space to store result
```

R1,R2 R1 = 0x12A2E640, R2 = 0xF2100123

Program: add64.s

```
 7.  Main
 8.          LDR     R0, =Value1 ; Pointer to first value
 9.          LDR     R1, [R0]    ; Load first part of value1
10.          LDR     R2, [R0, #4] ; Load lower part of value1
11.          LDR     R0, =Value2 ; Pointer to second value
12.          LDR     R3, [R0]    ; Load upper part of value2
13.          LDR     R4, [R0, #4] ; Load lower part of value2
14.          ADDS    R6, R2, R4  ; Add lower 4 bytes and set carry flag
15.          ADC     R5, R1, R3  ; Add upper 4 bytes including carry
16.          LDR     R0, =Result ; Pointer to Result
17.          STR     R5, [R0]    ; Store upper part of result
18.          STR     R6, [R0, #4] ; Store lower part of result
19.          SWI     &11
20.
21.  Value1 DCD     &12A2E640, &F2100123 ; Value to be added
22.  Value2 DCD     &001019BF,  &40023F51 ; Value to be added
23.  Result  DCD     0           ; Space to store result
```

R3,R4 R3 = 0x001019BF, R4 = 0x40023F51

Program: add64.s

```
 7.  Main
 8.          LDR     R0, =Value1 ; Pointer to first value
 9.          LDR     R1, [R0]    ; Load first part of value1
10.          LDR     R2, [R0, #4] ; Load lower part of value1
11.          LDR     R0, =Value2 ; Pointer to second value
12.          LDR     R3, [R0]    ; Load upper part of value2
13.          LDR     R4, [R0, #4] ; Load lower part of value2
14.          ADDS    R6, R2, R4  ; Add lower 4 bytes and set carry flag
15.          ADC     R5, R1, R3  ; Add upper 4 bytes including carry
16.          LDR     R0, =Result ; Pointer to Result
17.          STR     R5, [R0]    ; Store upper part of result
18.          STR     R6, [R0, #4] ; Store lower part of result
19.          SWI     &11
20.
21.  Value1 DCD     &12A2E640, &F2100123 ; Value to be added
22.  Value2 DCD     &001019BF, &40023F51 ; Value to be added
23.  Result  DCD     0          ; Space to store result
```

ADDS Add lower words and record any carry over (S)
0xF2100123 + 0x40023F51 = 0x32124074 with Carry

Program: add64.s

```
 7.  Main
 8.          LDR     R0, =Value1 ; Pointer to first value
 9.          LDR     R1, [R0]    ; Load first part of value1
10.          LDR     R2, [R0, #4] ; Load lower part of value1
11.          LDR     R0, =Value2 ; Pointer to second value
12.          LDR     R3, [R0]    ; Load upper part of value2
13.          LDR     R4, [R0, #4] ; Load lower part of value2
14.          ADDS    R6, R2, R4  ; Add lower 4 bytes and set carry flag
15.          ADC     R5, R1, R3  ; Add upper 4 bytes including carry
16.          LDR     R0, =Result ; Pointer to Result
17.          STR     R5, [R0]    ; Store upper part of result
18.          STR     R6, [R0, #4] ; Store lower part of result
19.          SWI     &11
20.
21.  Value1 DCD     &12A2E640, &F2100123 ; Value to be added
22.  Value2 DCD     &001019BF, &40023F51 ; Value to be added
23.  Result  DCD     0          ; Space to store result
```

ADC Add upper words, including carry over from lower words

0x12A2E640 + 0x001019BF + Carry = 0x12B30000

Program: add64.s

```
 7.  Main
 8.          LDR     R0, =Value1 ; Pointer to first value
 9.          LDR     R1, [R0]    ; Load first part of value1
10.          LDR     R2, [R0, #4] ; Load lower part of value1
11.          LDR     R0, =Value2 ; Pointer to second value
12.          LDR     R3, [R0]    ; Load upper part of value2
13.          LDR     R4, [R0, #4] ; Load lower part of value2
14.          ADDS    R6, R2, R4  ; Add lower 4 bytes and set carry flag
15.          ADC     R5, R1, R3  ; Add upper 4 bytes including carry
16.          LDR     R0, =Result ; Pointer to Result
17.          STR     R5, [R0]    ; Store upper part of result
18.          STR     R6, [R0, #4] ; Store lower part of result
19.          SWI     &11
20.
21.  Value1 DCD     &12A2E640, &F2100123 ; Value to be added
22.  Value2 DCD     &001019BF,  &40023F51 ; Value to be added
23.  Result  DCD     0           ; Space to store result
```

R5,R6 R5 = 0x12B30000, R6 = 0x32124074

Program: factorial.s

```
 4.          AREA    Program, CODE, READONLY
 7.  Main
 8.          LDR     R0, =DataTable    ; Load address of lookup table
 9.          LDR     R1, Value         ; Offset of value to be looked up
10.          MOV     R1, R1, LSL #0x2  ; Read value from table
13.          ADD     R0, R0, R1        ; index into table at R0
14.          LDR     R2, [R0]          ; Read value from table entry R1
15.          LDR     R3, =Result       ; Load address of result
16.          STR     R2, [R3]          ; Store the answer
18.          SWI     &11
19.
20.          AREA    DataTable, DATA
22.          DCD     1 ;0! = 1         ; Table containing factorials
23.          DCD     1 ;1! = 1
24.          DCD     2 ;2! = 2
30.  Value   DCB     5
32.  Result  DCW     0
```

Program: factorial.s

```
 4.          AREA    Program, CODE, READONLY
 7.  Main
 8.          LDR     R0, =DataTable     ; Load address of lookup table
 9.          LDR     R1, Value          ; Offset of value to be looked up
10.          MOV     R1, R1, LSL #0x2 ; Read value from table
13.          ADD     R0, R0, R1         ; index into table at R0
14.          LDR     R2, [R0]           ; Read value from table entry R1
15.          LDR     R3, =Result        ; Load address of result
16.          STR     R2, [R3]           ; Store the answer
18.          SWI     &11
19.
20.          AREA    DataTable, DATA
22.          DCD     1 ;0! = 1          ; Table containing factorials
23.          DCD     1 ;1! = 1
24.          DCD     2 ;2! = 2
30.  Value   DCB     5
32.  Result  DCW     0
```

AREA Place Program code into CODE area

Program: factorial.s

```
 4.          AREA    Program, CODE, READONLY
 7.  Main
 8.          LDR     R0, =DataTable    ; Load address of lookup table
 9.          LDR     R1, Value         ; Offset of value to be looked up
10.          MOV     R1, R1, LSL #0x2  ; Read value from table
13.          ADD     R0, R0, R1        ; index into table at R0
14.          LDR     R2, [R0]          ; Read value from table entry R1
15.          LDR     R3, =Result       ; Load address of result
16.          STR     R2, [R3]          ; Store the answer
18.          SWI     &11
19.
20.          AREA    DataTable, DATA
22.          DCD     1 ;0! = 1         ; Table containing factorials
23.          DCD     1 ;1! = 1
24.          DCD     2 ;2! = 2
30.  Value   DCB     5
32.  Result  DCW     0
```

AREA Place DataTable into DATA area

Program: factorial.s

```
 4.          AREA    Program, CODE, READONLY
 7.  Main
 8.          LDR     R0, =DataTable    ; Load address of lookup table
 9.          LDR     R1, Value         ; Offset of value to be looked up
10.          MOV     R1, R1, LSL #0x2  ; Read value from table
13.          ADD     R0, R0, R1        ; index into table at R0
14.          LDR     R2, [R0]          ; Read value from table entry R1
15.          LDR     R3, =Result       ; Load address of result
16.          STR     R2, [R3]          ; Store the answer
18.          SWI     &11
19.
20.          AREA    DataTable, DATA
22.          DCD     1 ;0! = 1         ; Table containing factorials
23.          DCD     1 ;1! = 1
24.          DCD     2 ;2! = 2
30.  Value   DCB     5
32.  Result  DCW     0
```

=label Load Address of *label* not value at *label*

Program: factorial.s

```
 4.          AREA    Program, CODE, READONLY
 7.  Main
 8.          LDR     R0, =DataTable    ; Load address of lookup table
 9.          LDR     R1, Value         ; Offset of value to be looked up
10.          MOV     R1, R1, LSL #0x2  ; Read value from table
13.          ADD     R0, R0, R1        ; index into table at R0
14.          LDR     R2, [R0]          ; Read value from table entry R1
15.          LDR     R3, =Result       ; Load address of result
16.          STR     R2, [R3]          ; Store the answer
18.          SWI     &11
19.
20.          AREA    DataTable, DATA
22.          DCD     1 ;0! = 1         ; Table containing factorials
23.          DCD     1 ;1! = 1
24.          DCD     2 ;2! = 2
30.  Value   DCB     5
32.  Result  DCW     0
```

label Load value at *label*

Program: factorial.s

```
 4.          AREA    Program, CODE, READONLY
 7.  Main
 8.          LDR     R0, =DataTable    ; Load address of lookup table
 9.          LDR     R1, Value         ; Offset of value to be looked up
10.          MOV     R1, R1, LSL #0x2  ; Read value from table
13.          ADD     R0, R0, R1        ; index into table at R0
14.          LDR     R2, [R0]          ; Read value from table entry R1
15.          LDR     R3, =Result       ; Load address of result
16.          STR     R2, [R3]          ; Store the answer
18.          SWI     &11
19.
20.          AREA    DataTable, DATA
22.          DCD     1 ;0! = 1         ; Table containing factorials
23.          DCD     1 ;1! = 1
24.          DCD     2 ;2! = 2
30.  Value   DCB     5
32.  Result  DCW     0
```

LSL Multiply R1 by 4 – Size of entry in DataTable

Program: factorial.s

```
 4.          AREA    Program, CODE, READONLY
 7.  Main
 8.          LDR     R0, =DataTable    ; Load address of lookup table
 9.          LDR     R1, Value         ; Offset of value to be looked up
10.          MOV     R1, R1, LSL #0x2  ; Read value from table
13.          ADD     R0, R0, R1        ; index into table at R0
14.          LDR     R2, [R0]          ; Read value from table entry R1
15.          LDR     R3, =Result       ; Load address of result
16.          STR     R2, [R3]          ; Store the answer
18.          SWI     &11
19.
20.          AREA    DataTable, DATA
22.          DCD     1 ;0! = 1         ; Table containing factorials
23.          DCD     1 ;1! = 1
24.          DCD     2 ;2! = 2
30.  Value   DCB     5
32.  Result  DCW     0
```

ADD Add offset (R1) to start of table

Program: factorial.s

```
 4.          AREA    Program, CODE, READONLY
 7.  Main
 8.          LDR     R0, =DataTable    ; Load address of lookup table
 9.          LDR     R1, Value         ; Offset of value to be looked up
10.          MOV     R1, R1, LSL #0x2  ; Read value from table
13.          ADD     R0, R0, R1        ; index into table at R0
14.          LDR     R2, [R0]          ; Read value from table entry R1
15.          LDR     R3, =Result       ; Load address of result
16.          STR     R2, [R3]          ; Store the answer
18.          SWI     &11
19.
20.          AREA    DataTable, DATA
22.          DCD     1 ;0! = 1         ; Table containing factorials
23.          DCD     1 ;1! = 1
24.          DCD     2 ;2! = 2
30.  Value   DCB     5
32.  Result  DCW     0
```

[R0] Read value from DataTable + Offset

Program: factorial.s

```
 4.          AREA    Program, CODE, READONLY
 7.  Main
 8.          LDR     R0, =DataTable    ; Load address of lookup table
 9.          LDR     R1, Value         ; Offset of value to be looked up
10.          MOV     R1, R1, LSL #0x2  ; Read value from table
13.          ADD     R0, R0, R1        ; index into table at R0
14.          LDR     R2, [R0]          ; Read value from table entry R1
15.          LDR     R3, =Result       ; Load address of result
16.          STR     R2, [R3]          ; Store the answer
18.          SWI     &11
19.
20.          AREA    DataTable, DATA
22.          DCD     1 ;0! = 1         ; Table containing factorials
23.          DCD     1 ;1! = 1
24.          DCD     2 ;2! = 2
30.  Value   DCB     5
32.  Result  DCW     0
```

[R3] Store value at address in R3

Program: factorial.s

```
 4.          AREA    Program, CODE, READONLY
 7.  Main
 8.          LDR     R0, =DataTable     ; Load address of lookup table
 9.          LDR     R1, Value          ; Offset of value to be looked up
10.          MOV     R1, R1, LSL #0x2   ; Read value from table
13.          ADD     R0, R0, R1         ; index into table at R0
14.          LDR     R2, [R0]           ; Read value from table entry R1
15.          LDR     R3, =Result        ; Load address of result
16.          STR     R2, [R3]           ; Store the answer
18.          SWI     &11
19.
20.          AREA    DataTable, DATA
22.          DCD     1 ;0! = 1          ; Table containing factorials
23.          DCD     1 ;1! = 1
24.          DCD     2 ;2! = 2
30.  Value   DCB     5
32.  Result  DCW     0
```

LDR Using Scaled Register offset we can write:

```
LDR R2, [R0, R1, LSL #2]
```