Data Operation Instructions

Data Operation

ARM can only perform data operations on registers.

All data operation instructions can be extended by adding a suffix to the instructions.

- execute conditionally by adding the conditional suffix.
- alter the status flags by appending the ‘S’ suffix.

When both are used

- the conditional suffix has the first priority.

Types of Data Operation

Data processing operations include the following instruction types:

- Bit shifting
- Register movement
- Arithmetic
- Multiply
- Logical boolean
- Status flag operation

Instruction Format

Most data operation instructions use one destination register plus two operands, where the first operand is always a register `Rn` (when it is available).

Basic syntax: `<OPERATION>     Rd,   Rn,  Op2`

The second operand `Op2` can be:

- an immediate value
- a register
- a register shifted by an immediate value
- a register shifted by a register

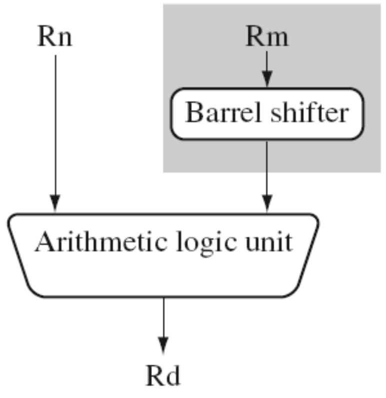
Bit-Shifting Operation

Bit shifting is achieved by the Barrel Shifter

- a hardware circuit that performs the bit shifting within the same instruction cycle time

Implements four types of shift operation

- `LSL`: logical left shift
- `LSR`: logical right shift
- `ASR`: arithmetic right shift
- `ROR`: rotate right



Bit shifting increases the power and flexibility of many data processing instructions, as shall be seen.

Register Move Instruction

MOV instructions do not use first operands. These instructions simply move the second operands to the destination register Rd.

Examples:
Copy from one register to another register

```
MOV    r1, r2          ; r1 = r2
```

Copy an immediate value to a register

```
MOV    r1, #0x32       ; r1 = 0x32

MVN    r0,  r1         ; r0 = ~r1
                       ; inversion of the r1 content
```

But the following will not work

```
MOV    r1, #0x504f0000    ; r1 = 0x504f0000 is desired
```

Constant is larger than 255!!!!

Restricted Use of the Immediate Value

The ARM instruction is of a 32-bit fixed length.

Typical instruction encoding for the immediate operand:

- 4 bits for the condition code
- 3 bits for the class code (i.e., a data processing class)
- 4 bits for the opcode
- 1 bit for the set status flag
- 4 bits for the destination register (i.e., Rd)
- 4 bits for the first operand register (i.e., Rn)

12 bits left for storing the immediate value in most arithmetic/logic instructions, which implies that only up to 4096 distinct bit patterns, 8 bits for immediate and 4 bits for shift.

ARM Approach

Instead of using the whole 12 bits for a single integer, the 12 bits are split onto two parts, which extend the range but reduce the precision.

- an 8-bit immediate number (*n*), giving the number a range between 0 to 255
- a 4-bit rotation field (*r*), operated using the Barrel Shifter

The full immediate value is calculated by rotating the number *n* to the right by 2r:

$$value = n \text{ ror } 2r$$

Examples: 200 is coded as *n* = 200, *value* = 0, *r*=0

0x06000000 is coded as *n* = 6, *value* = 4, *r*=2

Restricted Numbers

But only numbers whose bit pattern can fit into one byte can be generated.

Examples:

a) 0x08200000 is valid as it can be generated using n = 0x82 (and then rotate it by 12 positions)

b) 0x08210000 is invalid as its ‘821’ hex bit pattern stretches over 12 bits and hence cannot be specified by the 8-bit *n* field

Invalid numbers can be generated in multiple instructions, but more conveniently, loaded from memory (accessed using PC-relative addressing)

Pseudo-Instruction LDR

Instead of letting the programmer check whether a number is ‘constructible’

- a special form of the LDR instruction is available

```
LDR Rd,=num
```

- a Load immediate instruction, replacing ‘#’ with ‘=‘.

The assembler will check the number

- use `MOV` if the number can be generated
- store the invalid number into memory (somewhere nearby the instruction), and load it using PC-relative `LDR`

The space that the number needs will be created automatically by the assembler – the literal pool.

Examples

Data value 511 = 0x0000 01FF

0x1FF = 0b0001 1111 1111, where the non '0' bit pattern spreads over nine bits

So the following is an invalid instruction

```
        MOV   r1,   #511       ; Illegal instruction
```

Possible alternatives:

```
(i)  MOV r1,#1, 24  ; 256 = (1 ror 24), i.e., constructible
     ADD r1,#255    ; 256+255 = 511

(ii) LDR r1, _lp0   ; load from memory
               :
 _lp0: .word  511   ; declared with 0x1FF

(iii) LDR r1, =511  ; let the assembler do the work
```

MOV Variations

(i) Usage of the bit-shifting operation with the MOV instruction

(a) Performs very fast multiplication and division
Left shift = multiply by 2
Right shift = divide by 2

Example:

```
MOV   r1, r2, lsl #2              ; r1 = r2 << 2
                                  ;    = 4 x r2
```

(b) Convenient way to perform bit masking

Example:

```
MOV   r1, #0x1, lsl 15     ; r1 = 0x00000001 << 15
                           ;    = 0x00008000
                           ; i.e. set bit 15
```

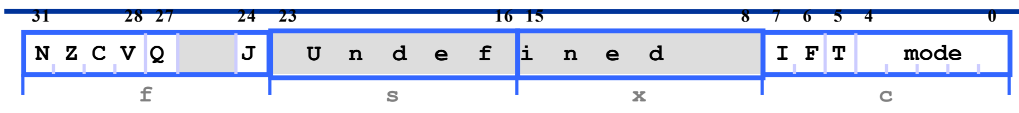
(ii) Adding an 'S' suffix on a data processing instruction will also update the corresponding flags in the CPSR.

Example:

```
MOVS  r1, r2, LSL #1  ; r1= (r2<<1)
```

Arithmetic Instructions

Program Status Registers



- Condition code flags
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed
- Sticky Overflow flag - Q flag
  - Architecture 5TE/J only
  - Indicates if saturation has occurred
- J bit
  - Architecture 5TEJ only
  - J = 1: Processor in Jazelle state
- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- T Bit
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- Mode bits
  - Specify the processor mode

Status Flag Operation

- Can only be Set/Reset by:
  - Appending an ‘S’ to end of Instruction Mnemonic (eg. `EORS`)
  - Executing Special Instructions Made to Set/Reset Flags
- N Flag – Checks for a Negative results by Checking MSb of Value
- V Flag – Signed Overflow (XOR of carry-in and carry-out of MSb)
- Z Flag – Zero flag, set when result is all ‘0’s
- C Flag – Set when:
  - Addition result greater than or equal to 2^{32}
  - Result of subtraction is positive
  - result of inline barrel shifter operation in `MOV` or logic instruction

Logical Instructions

Logical instructions perform the boolean operation on the pair of operands, and are useful for bit masking purposes.
(E.g., clear status bit or change interrupt masks in CPSR)

`and`: Logical bitwise AND
`orr`: Logical bitwise inclusive OR
`eor`: Logical bitwise exclusive OR
`bic`: Logical bit clear [bitwise AND(op1, NOT(op2))]

Examples: LSB mask, single bit clear, multiple bit clear

```
and   r0, r1, #0xff     ; r0 <- lowest byte of r1
bic   r0, r1, #0x10     ; clear bit 5, result in r0
bic   r0, r1, #0x5      ; clear bits 0 and 2
```

Comparison Instructions

These four instructions set the status bits/flags (N, Z, C, V) in the PSR according to the results of their operations.

`CMP`: compare, using subtraction

`CMN`: compare negated, using addition

`TEQ`: test for equality, using XOR – does not affect V flag

`TST`: test bit(s), using AND – does not affect V flag

Example:

```
CMP r0, #2    ; execute r0 - 2  and set N,Z,C,V
              ; flag bits accordingly
              ; do not store r0-2 anywhere
```

Equivalent of Compare

Equivalent functionalities can also be done using regular instructions with the ‘S’ suffix.

Example:

`SUBS` can be used in place of `CMP`

The difference is no register is used to hold the result when using the `CMP` status operation.

These instructions are typically used to implement flow control with the Branch instructions.

CMN Instruction

Inverse of CMP instruction – adds values instead of subtracting

Negates second operand, then subtracts (so effectively adds)

Example:

decimal

```
cmn    r0, #-20    ;-20=0xffffffec
```

same as

hexadecimal

```
cmp    r0, #0x14   ;+20=0x00000014
```

TEQ and TST Instructions

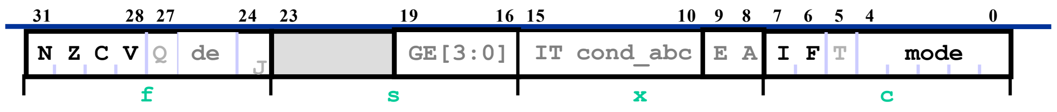
Does not affect V Flag

`TST` is Useful to determine if one or more bits are set (or clear)

`TST` is often used with a constant called a “MASK”

`TEQ` is useful for determining if the content of two registers contains identical values

PSR access



- `MRS` and `MSR` allow contents of CPSR / SPSR to be transferred to / from a general purpose register or take an immediate value
  - `MSR` allows the whole status register, or just parts of it to be updated
- Interrupts can be enable/disabled and modes changed, by writing to the CPSR
  - Typically a read/modify/write strategy should be used:

```
MRS r0,CPSR        ; copy CPSR into r0
BIC r0,r0,#0x80    ; clear bit 7 to enable IRQ
MSR CPSR_c,r0      ; write modified value to ‘c’ byte only
```

- In User Mode, all bits can be read but ***only the condition flags*** (f) can be modified
- (Note: These instructions clear the IRQ bit in CPSR, which enables the IRQ interrupt.)

Logic Instructions

Perform Boolean algebra operations on operands

• `movn`:	copy value and negate
• `and`:	logical AND operation (bit-by-bit)
• `orr`:	logical (inclusive) OR operation (bit-by-bit)
• `eor`:	logical (exclusive) OR operation (bit-by-bit)
• `bic`:	bit clear operation

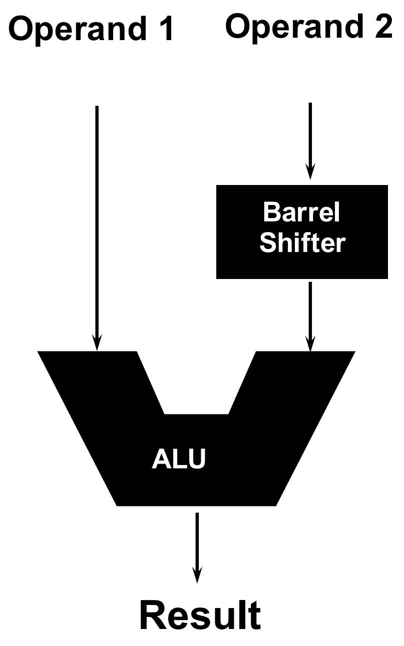
Example:

```
and  r1, r2, r3;r1 <- r2 AND r3
orr  r1, r2, r3;r1 <- r2 OR r3
eor  r1, r2, r3;r1 <- r2 EOR r3
bic  r1, r2, r3;r1 <- r2 AND (NOT r3)
```

To clear upper byte of r3:

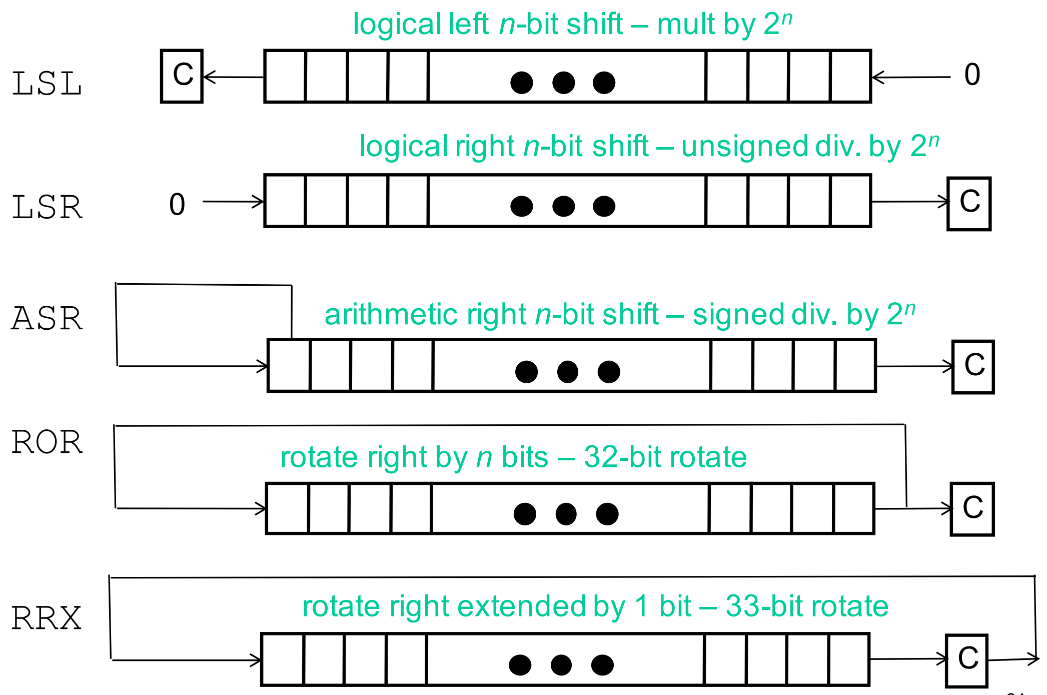
```
bic  r2, r3, #0xff000000
```

Shifts and Rotates



- Unique Feature of ARM Internal Datapath
- Before we saw this for Constants and Literals
- Two Types of Shifts:
  - logical (unsigned data)
  - arithmetic (signed data)
- No Rotate Left since Same as Rotate (32-n) Right
- No ASL Since Regular LSL does Same Thing

Shifts and Rotates


logical left n-bit shift – mult by 2^n
LSL
C
0
logical right n-bit shift – unsigned div. by 2^n
LSR
0
C
ASR
arithmetic right n-bit shift – signed div. by 2^n
C
ROR
rotate right by n bits – 32-bit rotate
C
RRX
rotate right extended by 1 bit – 33-bit rotate
C

Shift and Rotate Examples

```
mov   r4, r6, LSL #4       ;r4 <- r6 << 4 bits
mov   r4, r6, LSL r3       ;r4 <- r6 << # in r3
mov   r4, r6, ROR #12;r4 <- r6 rotated
                           ;left by 20 bits
```

- All shifts take one clock cycle EXCEPT register specified
- Register specified take 2 since only two read ports on register file
- Shift count is either unsigned 5-bit value OR LSB in register

Moving Byte in Reg to Another Reg

```
mov   r0, r2, LSR #24     ;LSB of r0<-MSB of r2
orr   r3, r0, r3, LSL #8 ;shift up r3 and
                          ;insert r0
```

- Above Sequence Moves MSB byte in r2 to LSB of r3
- Sequence Requires 2 clock cycles - only two read ports on register file

•EXAMPLE

before instr sequence

•`r0: 0xffffffff, r2: 0xaaeeeeee, r3: 0x55555555`

after instr sequence

•`r0<-0x000000aa, r2: 0xaaeeeeee, r3<-0x555555aa`

Adding and Subtracting

Perform addition and subtraction of 32-bit signed and unsigned values.

- `ADD`: Addition
- `ADC`: Addition with Carry – Useful for Multiword Arith (need S flag)
- `SUB`: Subtract
- `SBC`: Subtract with Carry – Useful for Multiword Arith (need S flag)
- `RSB`: Reverse Subtract – Useful for Arithmetic
- `RSC`: Reverse Subtract with Carry

Example:

```
SUB     r0, r1, r2      ;r0 <- r1 - r2
SUB     r0, #0, r1      ;r0 <- #0 - r1 = -r1  (negate a number)
RSB     r0, r1, #0      ;r0 <- #0 - r1 = -r1  (negate a number)
```

Used for loop counting

```
SUBS    r0, r0, #1       ;r0 <- r0 - 1, S  suffix to set Zero flag
                        ; when r0 <- zero
```

Adding and Subtracting

Perform addition and subtraction of 32-bit signed and unsigned values.

```
add    r1, r2, r3    ;r1 <- r2+r3
adc    r1, r2, r3    ;r1 <- r2+r3+C
sub    r1, r2, r3    ;r1 <- r2-r3
sbc    r1, r2, r3    ;r1 <- ((r2-r3)+C)-1
rsb    r1, r2, r3    ;r1 <- r3-r2
rsc    r1, r2, r3    ;r1 <- ((r3-r2)+C)-1
```

C=1 when carry occurs, C=0 otherwise

C=0 when borrow occurs, C=1 otherwise

Example: Add 64-bit value in r3:r2 with 64-bit value in r1:r0 result in r5:r4

```
adds    r4, r0, r2      ;r4 <- r0 + r2, set flags
add     r5, r1, r3      ;r0 <- #0 - r1 = -r1  (negate a number)
```

Subtract r3:r2 from r1:r0 with difference in r5:r4

```
subs    r4, r0, r2      ;r4 <- r0 - r2, set flags
sbc     r5, r1, r3      ;r5 <- (r1 - r3)+(C-1)
```

`adc    r1, r2, r3     ;r1<-(r2-r3)+C`

- *Example*: add one 8-bit value from another

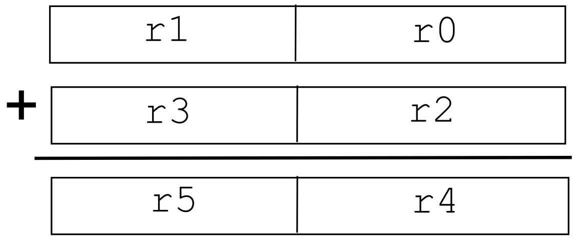
```
  1111 1101
 +1101 1110
  ---------
  1101 1011
```

  - Now Assume Two 4-bit Registers

```
   1101                1111
 + 1110               +1101
   ----                ----
   1011                1100
                      +   1
                       ----
                       1101
```

Carry-out so C=1

Add two 32-bit Integers



```
adds r4, r0, r2;r4<-(r0+r2) & set flags
adc  r5, r1, r3;r5<-(r1+r3)+ C
                         ;  & don't set flags
```

`sbc    r1, r2, r3     ;r1<-((r2-r3)+C)-1`

- *Example*: Subtract one 8-bit value from another

```
  1111 0001
 -1101 0010
  ---------
  0001 1111
```

- Now Assume Two 4-bit Registers

```
   0001                 1111
 - 0010                -1101
   ----                 ----
   1111                 0010
                       -   1
                        ----
                        0001
```

Need Borrow-in so C=0

Subtract two 96-bit Integers

	r8	r7	r6
-	r11	r10	r9
	r5	r4	r3

```
subs r3, r6, r9;r3<-(r6-r9) & set flags
sbcs r4, r7, r10     ;r4<-(r7-r10)+(C-1)
                     ;  & set flags
sbc  r5, r8, r11     ;r5<-(r8-r11)+(C-1)
                     ;  & don't set flags
```

Bit-Shifting Arithmetic

Bit shifting by the barrel shifter, when coupled with the Arithmetic instruction, provide powerful flexibility to the ARM instruction set.

Examples:

```
add r0, r1, r1, lsl #3 ; r0 <- r1+(r1<<3)
                       ;    = r1*9
rsb r0, r1, r1, lsl #4 ; r0 <- (r1<<4)-r1
                       ;    = r1*15
```

Bit-Shifting Arithmetic

Bit shifting by the barrel shifter, when coupled with the Arithmetic instruction, provide powerful flexibility to the ARM instruction set.

Examples: Subtraction is NOT commutative

```
sub r0, r2, r3, lsl #2 ; r0 <- r2-(4*r3)
 ;diff(r0)<-subtrahend(r2)-minuend(4*r3)
rsb r0, r3, r2, lsl #2 ; r0 <- (4*r2)-r3
 ;diff(r0)<-subtrahend(4*r2)-minuend(r3)
```

Barrell Shifter on last operand side of ALU so rsb allows its use in subtrahend operand

Absolute Value (2's Comp)

Set flags then use Condition form of `rbs` (`lt` suffix-less than)

Example: `lt` suffix conditionally executes `rbs`

```
cmp   r1, #0    ;set/reset N flag
                ;N=1 if r1 is negative
rsblt r0, r1, #0  ; three cases:
                  ;nop if r1=0x0 (N=0)
                  ;r0 <- 0x0 - r1 if N=1
                  ;nop if r0>0x0 (N=0)
```

Absolute Value (Signed Mag)

Clear MSb of `r0`

Example: Clear MSb of `r0` and place result in `r0`

```
bic  r0, r0, 0x2, lsl #30 ;clear MSb of r0
```

Multiply Instructions

These instructions multiply the contents of a pair of registers, with support for 32-bit operand and LSW 32-bit product

For 32-bit multiplication:

- `mul`: multiply - unsigned
- `mla`: multiply and accumulate – unsigned

Example: Unsigned multiply and multiply/accumulate

```
mul  r1, r2, r3          ;r1 <- r2 x r3
mla  r0, r1, r2, r3      ;r0 <- (r1 x r2) + r3
mla  r3, r1, r2, r3      ;r3 <- (r1 x r2) + r3
```

Only the lower 32 bits (LSW) of the 64-bit results are stored

Multiply Instructions (cont)

These instructions multiply the contents of a pair of registers, with support for 16-bit operands and 32-bit product. Specify which halfword of operand registers to use with `<x>` and `<y>` set to "`b`" (LSHW) or "`t`" (MSHW)

For 16-bit multiplication

- `smul<x><y>`: multiply - signed
- `smla<x><y>`: multiply and accumulate – signed

Example: Unsigned multiply and multiply/accumulate

```
smulbt  r1, r2, r3      ; r1<- LSHW(r2)x MSHW(r3)
smlatt  r0, r1, r2, r3  ; r0<- MSHW(r1)x MSHW(r2)+r3
smlatt  r3, r1, r2, r3  ; r3<- MSHW(r1)x MSHW(r2)+r3
```

Multiply Instructions (cont)

These instructions multiply the contents of a pair of registers, with support for 32-bit operand with 16-bit operand and 32-bit product. Specify which halfword of operand register to use with `<y>` set to "`b`" (LSHW) or "`t`" (MSHW)

For 16-bit multiplication

- `smulw<y>`: multiply - signed
- `smlaw<y>`: multiply and accumulate – signed

Example: Unsigned multiply and multiply/accumulate

```
smulwt  r1, r2, r3      ; r1 = r2 x MSHW(r3)
smlawb  r0, r1, r2, r3  ; r0 = r1 x LSHW(r2)+r3
smlawb  r3, r1, r2, r3  ; r3 = r1 x LSHW(r2)+r3
```

Multiply Instructions (cont)

These instructions multiply the contents of a pair of registers, with support for 32-bit operands with 64-bit product.

For 64-bit multiplication

- `smull`: signed multiply long
- `umull`: unsigned multiply long

MSW Product is r1 !!

Example:

```
smull r0,r1,r2,r3   ; (r1:r0)<-sign(r2) x sign(r3)
```

Note that Multiply instructions do not support the immediate values!

A constant value has to be loaded into the register first.

Multiply Instructions (cont)

These instructions multiply/accum the contents of a pair of registers, with support for 32-bit operands with 64-bit product.

For 64-bit multiply and accumulate

`smlal`: signed multiply and accumulate long

`umlal`: unsigned multiply accumulate long

Example:

```
smlal r0,r1,r2,r3   ;(r1:r0)<-sign(r2)x sign(r3)+(r1:r0)
```

Multiply Instructions (cont)

These are DUAL multiply instructions that use 16-bit operands with 32-bit result.

For dual 16-bit multiply and add/subtract products

`smuad`: signed dual multiply and add products together. Multiplies LSHW 16-bits of two operand regs and also mulitplies MSHW 16-bits of two operands then adds the two products together and stores 32-bit result

`smusd`: signed dual multiply and subtract products. Multiplies LSHW 16-bits of two operand regs and also mulitplies MSHW 16-bits of two operands then subtracts the two products and stores 32-bit result

Example:

```
smusd r1,r2,r3   ;r1<-[signed(LSHW(r2)xLSHW(r3))]
                 ;    - [signed(MSHW(r2)xMSHW(r3))]
```

Note: MSW products Subtracted from LSW products!!!

Multiply Instructions (cont)

These are DUAL multiply and accumulate instructions that use 16-bit operands with 32-bit result.

For dual 16-bit multiply and add/subtract products

`smlad`: signed dual multiply and add products together. Multiplies LSHW 16-bits of two operand regs and also mulitplies MSHW 16-bits of two operands then adds the two products together and accumulates with 32-bit result

`smlsd`: signed dual multiply and subtract products. Multiplies LSHW 16-bits of two operand regs and also mulitplies MSHW 16-bits of two operands then subtracts the two products and accumulates with 32-bit result

Example:

```
smlsd r1,r2,r3   ;r1<-[signed(LSHW(r2)xLSHW(r3))]
                 ;     - [signed(MSHW(r2)xMSHW(r3))] + r1
```

Note: MSW products Subtracted from LSW products!!!

Multiply Instructions (cont)

32 Multiply with MS 32-bits of Product

For 32-bit multiply and MS 32-bits Product:

`smmul{r}`: Keeps MSW of Product only, Optional "`r`" causes rounding, otherwise truncated result.

Example:

```
smmul  r1,r2,r3   ;r1<-truncated[MSW(r2xr3)]
smmulr r1,r2,r3   ;r1<-rounded[MSW(r2xr3)]
```

Multiply Instructions (cont)

32 Multiply with MS 32-bits of Product with 32-bit Accumulation

For dual 32-bit multiply and MS 32-bits Product:

`smmla{r}`: Keeps MSW of Product only, Optional "`r`" causes rounding, otherwise truncated result and accumulates.

`smmls{r}`: Keeps MSW of Product only, Optional "`r`" causes rounding, otherwise truncated result and subtracts from destination.

Example:

```
smmla   r1,r2,r3   ;r1<- r1 + truncated[MSW(r2xr3)]
smmlsr  r1,r2,r3   ;r1<- r1 - rounded[MSW(r2xr3)]
```

Multiply Instructions (cont)

Dual 16-bit signed multiply with addition or subtraction of products and 64-bit accumulation

`smlald{x}`: Dual multiply of halfwords of operands and adds them together. Optional "`x`" exchanges LS and MS words of second operand before multiplication. Accumulates the sum of products.

`smlsld{r}`: Dual multiply of halfwords of operands and adds them together. Optional "`x`" exchanges LS and MS words of second operand before multiplication. Accumulates the difference of products.

Example:

```
smlald   r1,r2,r3,r4    ;(r2:r1)<-signed[LS(r3)xLS(r4)]
                        ; + signed[MS(r3)xMS(r4)]+(r2:r1)
smlaldx   r1,r2,r3,r4    ;(r2:r1)<-signed[LS(r3)xMS(r4)]
                        ; + signed[MS(r3)xLS(r4)]+(r2:r1)
```

Multiply Instructions (cont)

Unsigned mulitply/accumulate for long operands

`umaal`: Multiplies 32-bits in `r3` and `r4`, adds the two values in `r1` and `r2`, and stores the result in `r1` and `r2`.

Example:

```
umaal   r1,r2,r3,r4    ;(r2,r1)<-(r2,r1)+unsigned(r3xr4)
```

Multiply Instructions (cont)

Some ARM processors have special purpose Internal Accumulators named `acc<x>`. `<x>` is an integer from `1` to `n`, and `n` differs for various processors. This internal Accumulator is 40-bits in length.

`mia`: Multiplies 32-bits in `r1` and `r2`, accumulates product in internal `acc<x>`

`miaph`: Multiply packed halfwords (16-bits) and accumulate. Multiplies signed halfwords from LS of `r1` and `r2`, and also multiplies MS signed halfwords of `r1` and `r2`. Then accumulates both 32-bit products in internal 40-bit `acc<x>`

`mia<x><y>`: Multiplies signed 16-bit value from selected half of `r1` with that of selected half of `r2`. Then accumulates the 32-bit result in `acc<x>`. `<x>` and `<y>` can be either "`b`" or "`t`" for bottom or top.

Example:

```
mia   acc0,r1,r2    ;acc0<-acc0 + signed(r1xr2)
```

Multiply and Divide Summary

- There are several classes of multiply - producing 32-bit and 64-bit results
- 32-bit versions on an ARM7TDMI will execute in 2 - 5 cycles (*RISC???*)

```
    – mul r0, r1, r2        ; r0 <- r1 * r2
    – mla r0, r1, r2, r3    ; r0 <- (r1 * r2) + r3
```

- 64-bit multiply instructions offer both signed and unsigned versions
  - For these instruction there are 2 destination registers

```
    – [u|s]mull r4, r5, r2, r3    ; r5:r4 <- r2 * r3
    – [u|s]mlal r4, r5, r2, r3    ; r5:r4 <- (r2 * r3) + r5:r4
```

- Most ARM cores do not offer integer divide instructions
  - Division operations will be performed by C library routines or inline shifts
  - Cortex-M3 does have division circuitry

Multiply Examples

```
mul   r0, r1, r2          ; r0 <- r1 * r2
mla   r0, r1, r2, r3      ; r0 <- (r1 * r2) + r3
muls r7, r8, r9           ; r7 <- r8*r9, set flags
smull r4, r8, r2, r3      ; r4 <- LSW of sign(r2*r3)
                          ; r8 <- MSW of sign(r2*r3)
                          ; signed arithmetic
umull r6, r8, r0, r1      ; r8:r6 <- r0*r1 (uns. arith)
smlal r4, r8, r2, r3      ; r8:r4 <- (LSW of r2*r3 +
                         ; MSW of r28r3) + r8:r4
                         ; signed arithmetic
umlal r5 r8, r0, r1       ; r8:r5 <- (r0*r1)+(r8:r5)
                          ; unsigned arithmetic
```

More Examples

Let `r0` contain A, `r1` contain B, and `r2` contain C

```
mov   r1, r0, LSL #2          ; B = ?

add   r0, r1, r1, LSL #2      ; A = ?

rsb   r0, r2, r2, LSL #3      ; A = ?

sub   r0, r0, r1, LSL #4      ; A = ?

add   r0, r0, r1, LSL #7      ; A = ?
```

More Examples

Let `r0` contain A, `r1` contain B, and `r2` contain C

```
mov   r1, r0, LSL #2          ; B = 4A

add   r0, r1, r1, LSL #2      ; A = 5A

rsb   r0, r2, r2, LSL #3      ; A = 7C

sub   r0, r2, r1, LSL #4      ; A = C - 16B

add   r0, r2, r1, LSL #7      ; A = 128B+C
```

Another Example

`r1` contains A, What does `r0` Contain after instructions:

```
add  r0, r1, r1, LSL #1   ; r0 <- ?
sub  r0, r0, r1, LSL #4   ; r0 <- ?
add  r0, r0, r1, LSL #7   ; r0 <- ?
```

Another Example

`r1` contains A, What does `r0` Contain after instructions:

```
add  r0, r1, r1, LSL #1   ; r0 <- 3A
sub  r0, r0, r1, LSL #4   ; r0 <- 3A – 16A = -13A
add  r0, r0, r1, LSL #7   ; r0 <- -13A + 128A = 115A
```