Overview

° Shift Operations
  • Field Insertion

° Multiplication Operations
  • Multiplication
  • Long Multiplication
  • Multiplication and accumulation
  • Signed and unsigned multiplications

Review: ARM Instructions So far

add
sub
mov
and
bic
orr
eor

Data Processing Instructions with shift and rotate

lsl, lsr, asr, ror

Review: Masking via Logical AND

° AND: Note that ANDing a bit with 0 produces a 0 at the output while ANDing a bit with 1 produces the original bit.

° This can be used to create a mask.
  • Example:
    
    1011 0110 1010 0100 0011 1101 1001 1010
    
    Mask: 0000 0000 0000 0000 0000 0000 1111 1111
  • The result of ANDing these two is:
    
    0000 0000 0000 0000 0000 0000 1001 1010
Review: Masking via Logical BIC

- **BIC (AND NOT):** Note that \textit{bic}ing a bit with 1 produces a 0 at the output while \textit{bic}ing a bit with 0 produces the original bit.
- This can be used to create a \textit{mask}.
  - Example:
    
    $\begin{array}{cccccccc}
    1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
    0 & 1 & 1 & 0 & 1 & 0 & 0 & 0
    \end{array}$

    \textbf{Mask:} $\begin{array}{cccccccc}
    0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
    1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
    \end{array}$
  - The result of \textit{bic}ing these two is:
    
    $\begin{array}{cccccccc}
    1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
    0 & 1 & 1 & 0 & 1 & 0 & 0 & 0
    \end{array}$

Extracting a field of bits (#1/2)

- Extract bit field from bit 9 (left bit no.) to bit 2 (size=8 bits) of register v1, place in rightmost part of register a1

- Shift field as far left as possible (9 $\rightarrow$ 31) and then as far right as possible (31 $\rightarrow$ 7)

Inserting a field of bits

- Insert bit field into bit 9 (left bit no.) to bit 2 (size=8 bits) of register a1 from rightmost part of register v1 (rest is 0)

- Shift left field 2 bits, Mask out field, OR in field

- \texttt{mov a2, v1, lsl #2} ; field left 2
  \texttt{bic a1, a1, #0x3FC} ; mask out 9-2
  \texttt{orr a1, a1, a2} ; OR in field

  \texttt{bic} stands for \textquoteleft bit clear\textquoteright, where \textquoteleft 1\textquoteright~in the second operand clears the corresponding bit in the first
Bit manipulation in C (#1/2)

° Convert C code to ARM ASM

° Bit Fields in C (Word as 32 bits vs int/unsigned!)

    struct {
        unsigned int ready: 1; /* bit 0 */
        unsigned int enable: 1; /* bit 1 */
    } rec;
    rec.enable = 1;
    rec.ready = 0;
    printf("%d %d", rec.enable, rec.ready);
    ...

Brian Kernighan & Dennis Ritchie: The C Programming Language, 2nd Ed., PP 150

Bit manipulation in C (#2/2)

    struct {
        unsigned int ready: 1; /* bit 0 */
        unsigned int enable: 1; /* bit 1 */
    } rec; /* v1 */
    rec.enable = 1;
    rec.ready = 0;
    printf("%d %d", rec.enable, rec.ready);

orr v1, v1, #0x2           ; 1 in bit 1
bic v1, v1, #1             ; 0 in bit 0,
ldr a1, =LCO              ; printf format
mov a2, v1, lsr #1       ; just bit 1
and a2, a2, 0x0001       ; mask down to 1
and a3, v1, 0x0001       ; just bit 0
bl printf             ; call printf

bic stands for ‘bit clear, where ‘1’ in the second operand clears the corresponding bit in the first

Multiply by Power of 2 via Shift Left (#1/3)

° In decimal:

• Multiplying by 10 is same as shifting left by 1:
  - 714_{10} \times 10_{10} = 7140_{10}
  - 56_{10} \times 10_{10} = 560_{10}

• Multiplying by 100 is same as shifting left by 2:
  - 714_{10} \times 100_{10} = 71400_{10}
  - 56_{10} \times 100_{10} = 5600_{10}

• Multiplying by 10^n is same as shifting left by n

Multiply by Power of 2 via Shift Left (#2/3)

° In binary:

• Multiplying by 2 is same as shifting left by 1:
  - 1_2 \times 10_2 = 110_2
  - 1010_2 \times 10_2 = 10100_2

• Multiplying by 4 is same as shifting left by 2:
  - 1_2 \times 100_2 = 1100_2
  - 1010_2 \times 100_2 = 101000_2

• Multiplying by 2^n is same as shifting left by n
Multiply by Power of 2 via Shift Left (#3/3)

° Since shifting is so much faster than multiplication (you can imagine how complicated multiplication is), a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

\[ a *= 8; \] (in C)

would compile to:

\[ \text{mov } a0, a0, lsl #3 \] (in ARM)

Shift, Add and Subtract for Multiplication

Add and Subtract Examples:

\[ f = 5* g \quad / \quad ^* f = (4+1) \times g \quad */ \quad \text{ (in C)} \]
\[ \text{add v1, v2, v2 lsl #2; v1 = v2 + v2 \times 4} \quad \text{ (in ARM)} \]

\[ f = 105 \times g \quad / \quad ^* f = (15 \times 7) \times g \quad */ \quad \text{ (in C)} \]
\[ ^* f = (16-1) \times (8-1) \times g \]  
\[ \text{rsb v1, v2, v2 lsl #4; v1 = -v2 + v2 \times 16} \quad \text{ (in ARM)} \]
\[ ; f = (16-1) \times g \]
\[ \text{rsb v1, v1, v1 lsl #3; v1 = -v1 + v1 \times 8} \quad \text{ (in ARM)} \]
\[ ; f = (8-1) \times f \]

Shift, Add and Subtract for Division

• ARM does not have division.
• Division A/B produces a quotient and a remainder.
• It should be done via sequence of subtraction and shifting (See Experiment 3)
• For B in A/B a constant value (eg 10) simpler technique via Shift, Add and Subtract is available (Will be discussed later)

Shift Right Arithmetic; Divide by 2???

° Shifting left by n is same as Multiplying by \(2^n\)

° Shifting right by n bits would seem to be the same as dividing by \(2^n\)

° Problem is signed integers
  • Zero fill is wrong for negative numbers

° Shift Right Arithmetic (\texttt{asr}); sign extends (replicates sign bit);

\[ 1111 \ 1111 \ 1111 \ 1000 = -8 \]
\[ 1111 \ 1111 \ 1111 \ 1100 = -4 \]
\[ 1111 \ 1111 \ 1111 \ 1110 = -2 \]
\[ 1111 \ 1111 \ 1111 \ 1111 = -1 \]
Is asr really divide by 2?

° Divide +5 by 4 via asr 2; result should be 1
  0000 0000 0000 0000 0000 0000 0000 0101
  0000 0000 0000 0000 0000 0000 0000 0001
  ° = +1, so does work
° Divide -5 by 4 via asr 2; result should be -1
  1111 1111 1111 1111 1111 1111 1111 1011
  1111 1111 1111 1111 1111 1111 1111 1110
  ° = -2, not -1; Off by 1, so doesn’t always work
° Rounds to –∞

MULTIPLY (unsigned): Terms, Example

° Paper and pencil example (unsigned):

| Multiplicand | 1000 |
| Multiplier  | 1001 |
| Product     | 01001000 |

• m bits x n bits = m+n bit product
• 32-bit value x 32-bit value = 64-bit value

Multiplication Instructions

° The Basic ARM provides two multiplication instructions.
° Multiply
  • mul Rd, Rm, Rs ; Rd = Rm * Rs
° Multiply Accumulate - does addition for free
  • mla Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn
° (Lower precision multiply instructions simply throws top 32 bits away)
° Restrictions on use:
  • Rd and Rm cannot be the same register
    - Can be avoided by swapping Rm and Rs around. This works because multiplication is commutative.
  • Cannot use PC.

These will be picked up by the assembler if overlooked.
° Operands can be considered signed or unsigned
  • Up to user to interpret correctly.

Multiplication Example

° Example:
  • in C: a = b * c;
  • in ARM:
    - let b be v1; let c be v2; and let a be v3 (It may be up to 64 bits)
      mul v3, v2, v1; a = b*c
      ; lower half of product into v3. Upper half is thrown up
° Note: Often, we only care about the lower half of the product.
Multiplication and Accumulate Example

° One example of use of `mla` is for string to number conversion: eg

Convert string="123" to value=123

value = 0
loop = 0
len = length of string
Rd = value
while loop <> len
  c = extract(string, len - loop, 1)
  Rm = 10 ^ loop
  Rs = ASC(c) - ASC ('0')
  mla Rd, Rm, Rs, Rd
  loop = loop + 1
endwhile

Multiply-Long and Multiply-Accumulate Long

° Instructions are
  • `MULL` which gives RdHi,RdLo:=Rm*Rs
  • `MLAL` which gives RdHi,RdLo:=(Rm*Rs)+RdHi,RdLo

° However the full 64 bit of the result now matter (lower precision multiply instructions simply throws top 32 bits away)
  • Need to specify whether operands are signed or unsigned

° Therefore syntax of new instructions are:
  • `umull` RdLo,RdHi,Rm,Rs    ;RdHi,RdLo:=Rm*Rs
  • `umlal` RdLo,RdHi,Rm,Rs    ;RdHi,RdLo:=(Rm*Rs)+RdHi,RdLo
  • `smull` RdLo, RdHi, Rm, Rs ;RdHi,RdLo:=Rm*Rs (Signed)
  • `smlal` RdLo, RdHi, Rm, Rs ;RdHi,RdLo:=(Rm*Rs)+RdHi,RdLo (Signed)

° Not generated by the compiler. (Needs Hand coding)

Division

° No Division Instruction in ARM

° Division has two be done in software through a sequence of shift/ subtract / add instruction.
  • General A/B implementation (See Experiment 3)
  • For B in A/B a constant value (eg 10) simpler technique via Shift, Add and Subtract is available (Will be discussed later)

Quiz

1. Specify instructions which will implement the following:
   a) \( a_1 = 16 \)
   b) \( a_2 = a_1 \times 4 \)
   c) \( a_1 = a_2 / 16 \) (r1 signed 2's comp.)
   d) \( a_2 = a_3 \times 7 \)

2. What will the following instructions do?
   a) `add a1, a2, a2, lsl #2`
   b) `rsb a3, a2, #0`

3. What does the following instruction sequence do?
   `add a1, a2, a2, lsl #1`
   `sub a1, a1, a2, lsl #4`
   `add a1, a1, a2, lsl #7`
Quiz Solution (#1/2)

1. Specify instructions which will implement the following:
   a) \(a1 = 16\)
      \[
      \text{mov a1, #16}
      \]
   b) \(a2 = a1 \times 4\)
      \[
      \text{mov a2, a1, lsl #2}
      \]
   c) \(a1 = a2 \div 16\) (r1 signed 2's comp.)
      \[
      \text{mov a1, a2, asr #4}
      \]
   d) \(a2 = a3 \times 7\)
      \[
      \text{rsb a2, a3, a3, lsl #3}
      \]
      \[
      \text{a2 = a3 \times (8-1)}
      \]

   whereas \(\text{sub a2, a3, a3, lsl #3}\) would give \(a3 \times -7\)

2. What will the following instructions do?
   a) \(\text{add a1, a2, a2, lsl #2}\)
      \[
      a1 = a2 + (a2 \times 4)
      \]
      \[
      \text{ie } a1 := a2 \times 5
      \]
   b) \(\text{rsb a3, a2, #0}\)
      \[
      r2 := 0-r1 \text{ ie } r2 := -r1
      \]

Quiz Solution (#2/2)

3. What does the following instruction sequence do?
   \[
   \text{add a1, a2, a2, lsl #1}
   \]
   \[
   \text{sub a1, a1, a2, lsl #4}
   \]
   \[
   \text{add a1, a1, a2, lsl #7}
   \]
   \[
   a1 = a2 + (a2 \times 2) = a2 \times 3
   a1 = a1 - (a2 \times 16) = (a2 \times 3) - (a2 \times 16) = a2 \times -13
   a1 = a1 + (a2 \times 128) = (a2 \times -13) + (a2 \times 128) = r1 \times 115
   \]
   \[
   \text{ie. } a1 = a2 \times 115
   \]

“And in Conclusion…”

° New Instructions:
   \[
   \text{mul}
   \]
   \[
   \text{mla}
   \]
   \[
   \text{umull}
   \]
   \[
   \text{umlal}
   \]
   \[
   \text{smull}
   \]
   \[
   \text{smlal}
   \]