Lecture 7: ARM Arithmetic and Bitwise Instructions

CSE 30: Computer Organization and Systems Programming
Diba Mirza
Dept. of Computer Science and Engineering
University of California, San Diego
Basic Types of ARM Instructions

1. Arithmetic: Only processor and registers involved
   1. compute the sum (or difference) of two registers, store the result in a register
   2. move the contents of one register to another

2. Data Transfer Instructions: Interacts with memory
   1. load a word from memory into a register
   2. store the contents of a register into a memory word

3. Control Transfer Instructions: Change flow of execution
   1. jump to another instruction
   2. conditional jump (e.g., branch if registeri == 0)
   3. jump to a subroutine
ARM Addition and Subtraction

- **Syntax of Instructions:**
  1  2, 3, 4

  where:
  1) instruction by name
  2) operand getting result ( "destination" )
  3) 1st operand for operation ( "source1" )
  4) 2nd operand for operation ( "source2" )

- Syntax is rigid (for the most part):
  - 1 operator, 3 operands
  - Why? *Keep Hardware simple via regularity*
Addition and Subtraction of Integers

- Addition in Assembly
  - Example: \texttt{ADD r0, r1, r2} (in ARM)
    Equivalent to: \texttt{a = b + c} (in C)
    where ARM registers \texttt{r0, r1, r2} are associated with C variables \texttt{a, b, c}

- Subtraction in Assembly
  - Example: \texttt{SUB r3, r4, r5} (in ARM)
    Equivalent to: \texttt{d = e - f} (in C)
    where ARM registers \texttt{r3, r4, r5} are associated with C variables \texttt{d, e, f}
Setting condition bits

- Simply add an ‘S’ following the arithmetic/logic instruction
  - Example: ADDS r0, r1, r2 (in ARM)

This is equivalent to r0=r1+r2 and set the condition bits for this operation
What is the min. number of assembly instructions needed to perform the following?

\[ a = b + c + d - e; \]

A. Single instruction  
B. Two instructions  
C. Three instructions  
D. Four instructions

Assume the value of each variable is stored in a register.
What is the min. number of assembly instructions needed to perform the following?

\[ a = b + c + d - e; \]

A. Single instruction  
B. Two instructions  
C. Three instructions  
D. Four instructions

Assume the value of each variable is stored in a register.
Addition and Subtraction of Integers

- How do the following C statement?
  \[ a = b + c + d - e; \]

- Break into multiple instructions
  - ADD r0, r1, r2 ; \( a = b + c \)
  - ADD r0, r0, r3 ; \( a = a + d \)
  - SUB r0, r0, r4 ; \( a = a - e \)

- Notice: A single line of C may break up into several lines of ARM.

- Notice: Everything after the semicolon on each line is ignored (comments)
Addition and Subtraction of Integers

- How do we do this?
  - \( f = (g + h) - (i + j); \)
- Use intermediate temporary register

```
ADD r0, r1, r2 ; f = g + h
ADD r5, r3, r4 ; temp = i + j
SUB r0, r0, r5 ; f = (g+h) - (i+j)
```
Immediates

- Immediates are numerical constants.
- They appear often in code, so there are ways to indicate their existence
- Add Immediate:
  - \( f = g + 10 \) (in C)
  - \text{ADD r0, r1, #10} (in ARM)
    - where ARM registers \( r0, r1 \) are associated with C variables \( f, g \)
  - Syntax similar to \texttt{add} instruction, except that last argument is a \#number instead of a register.
Arithmetic operations: Addressing Modes

1. Register Direct Addressing: Operand values are in registers:
   - ADD r3, r0, r1; r3 = r0 + r1

2. Immediate Addressing Mode: Operand value is within the instruction
   - ADD r3, r0, #7; r3 = r0 + 7
   - The number 7 is stored as part of the instruction

3. Register direct with shift or rotate (more next lecture)
   - ADD r3, r0, r1, LSL#2; r3 = r0 + r1 << 2
What is a likely range for immediates in the immediate addressing mode

A. 0 to \((2^{32}-1)\)

B. 0 to 255
What is a likely range for immediates in the immediate addressing mode

A. 0 to \((2^{32}-1)\)

B. 0 to 255  Immediates are part of the instruction (which is a total of 32 bits). Number of bits reserved for representing immediates is 8 bits
Add/Subtract instructions

1. ADD  r1, r2, r3;  r1=r2+r3
2. ADC  r1, r2, r3;  r1=r2+r3+ C(arry Flag)
3. SUB  r1, r2, r3;  r1=r2-r3
4. SUBC r1, r2, r3;  r1=r2-r3 +C -1
5. RSB  r1, r2, r3;  r1= r3-r2;
6. RSC  r1, r2, r3;  r1=r3-r2 +C -1
Paper and pencil example (unsigned):

Multiplicand  1000
Multiplier    x1001
      1000
      0000
      0000
      +1000
-----
01001000

m bits x n bits = m + n bit product
Multiplication

- Example:
  - in C: \( a = b \times c; \)
  - in ARM:
    let \( b \) be \( r2 \); let \( c \) be \( r3 \); and let \( a \) be \( r0 \) and \( r1 \) (since it may be up to 64 bits)
    \[
    \text{MUL } r0, \ r2, \ r3 \ ; \ b \times c \text{ only 32 bits stored}
    \]

Note: Often, we only care about the lower half of the product.

- \[
\text{SMULL } r0,r1,r2,r3 \ ; \ 64 \text{ bits in } r0:r1
\]
Multiply and Divide

- There are 2 classes of multiply - producing 32-bit and 64-bit results
- 32-bit versions on an ARM7TDMI will execute in 2 - 5 cycles
  
  ▪ 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
Logical Operations operate on

A. Bits

B. Instructions

C. Numbers

D. Strings
Logical Operations operate on

A. Bits
B. Instructions
C. Numbers
D. Strings
Logical Operators

- Basic logical operators:
  - AND
  - OR
  - XOR
  - BIC (Bit Clear)

- In general, can define them to accept >2 inputs, but in the case of ARM assembly, both of these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware
Logical Operators

- Truth Table: standard table listing all possible combinations of inputs and resultant output for each logical operator.
- Truth Table for AND, OR and XOR

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
<th>A OR B</th>
<th>A XOR B</th>
<th>A BIC B</th>
<th>A AND (NOT B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Syntax of Instructions:

1 2, 3, 4

where:

1) instruction by name
2) operand getting result ( “destination” )
3) 1st operand for operation ( “source1” )
4) 2nd operand for operation ( “source2” )

Syntax is rigid (for the most part):

1 operator, 3 operands
Why? Keep Hardware simple via regularity
Bitwise Logic Operations

- **Bitwise AND in Assembly**
  - Example: \texttt{AND r0, r1, r2} (in ARM)
  - Equivalent to: \( r0 = r1 \& r2 \) (in C)

- **Bitwise OR in Assembly**
  - Example: \texttt{ORR r3, r4, r5} (in ARM)
  - Equivalent to: \( r3 = r4 | r5 \) (in C)

- **Bitwise XOR in Assembly**
  - Example: \texttt{EOR r0, r1, r2} (in ARM)
  - Equivalent to: \( r0 = r1 \oplus r2 \) (in C)

- **Bitwise Clear in Assembly**
  - Example: \texttt{BIC r3, r4, r5} (in ARM)
  - Equivalent to: \( r3 = r4 \& (!r5) \) (in C)
Bit wise operations

r0:  01101001
r1:  11000111

__________

ORR r3, r0, r1; r3:  11101111
AND r3, r0, r1;  r3:  01000001
EOR r3, r0, r1;  r3:  10101110
BIC  r3, r0, r1; r3:  00101000
Uses for Logical Operators

- 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 |
    | 0000 0000 0000 0000 0000 1111 1111 1111 |

  - The result of ANDing these:
    
    | 0000 0000 0000 0000 0000 1101 1001 1010 |

  - mask last 12 bits
Uses for Logical Operators

- Similarly, note that ORing a bit with 1 produces a 1 at the output while ORing a bit with 0 produces the original bit.
- This can be used to force certain bits of a string to 1s.
  - For example, 0x12345678 OR 0x0000FFF results in 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).
Invert bits 0–2 of x

A. x AND 00000111

B. x OR 00000111

C. x MOVN 00000111

D. x XOR 00000111
Invert bits 0–2 of x

A. \(x \text{ AND } 00000111\)

B. \(x \text{ OR } 00000111\)

C. \(x \text{ MOVN } 00000111\)

D. \(x \text{ XOR } 00000111\)
Uses for Logical Operators

- Finally, note that \texttt{BICing} a bit with 1 resets the bit (sets to 0) at the output while \texttt{BICing} a bit with 0 produces the original bit.

- This can be used to force certain bits of a string to 0s.
  - For example, 0x12345678 \texttt{OR} 0x0000FFFF results in 0x12340000 (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 0s).
Find the 1's complement of x

A. $x \text{ XOR } 00000000$

B. $x \text{ XOR } 11111111$

C. $x \text{ XOR } 11111110$

D. $x \text{ BIC } 11111111$
Find the 1's complement of $x$

A. $x$ XOR 00000000

B. $x$ XOR 11111111

C. $x$ XOR 11111110

D. $x$ BIC 11111111
Assignment Instructions

- **Assignment in Assembly**
  - Example: \texttt{MOV r0, r1} (in ARM)
    Equivalent to: \( a = b \) (in C)
    where ARM registers \texttt{r0, r1} are associated with C variables \texttt{a, b}
  
  - Example: \texttt{MOV r0, #10} (in ARM)
    Equivalent to: \( a = 10 \) (in C)
Assignment Instructions

- MVN – Move Negative – moves one’s complement of the operand into the register.

Assignment in Assembly

- Example: `MVN r0, #0` (in ARM)
  
  Equivalent to: `a = -1` (in C)

  where ARM registers `r0` are associated with C variables `a`

  Since \(-0x00000000 = 0xFFFFFFFF\)
Conclusion

- In ARM Assembly Language:
  - Registers replace C variables
  - One Instruction (simple operation) per line
  - Simpler is Better
  - Smaller is Faster

- Instructions so far:
  - ADD, SUB, MUL, MULA, [U|S]MULL, [U|S]MLAL

- Registers:
  - Places for general variables: r0–r12