Lecture 8: ARM Arithmetic and Bitweise Instructions

CSE 30: Computer Organization and Systems Programming
Winter 2014
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: `ADD r0,r1,r2` (in ARM)

  Equivalent to: `a = b + c` (in C)

  where ARM registers `r0,r1,r2` are associated with C variables `a, b, c`
- Subtraction in Assembly
  - Example: `SUB r3, r4, r5` (in ARM)

  Equivalent to: `d = e - f` (in C)

  where ARM registers `r3,r4,r5` are associated with C variables `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)
  - `ADD r0,r1,#10` (in ARM)
  - where ARM registers `r0,r1` are associated with C variables `f, g`
- Syntax similar to `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

Integer Multiplication

- 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 * 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)

```
MUL r0, r2, r3    ; b*c only 32 bits stored
```

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

```
SMULL r0,r1,r2,r3 ; 64 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
- Truth Table for AND, OR and XOR

A	B	A AND B	A OR B	A XOR B	A AND (NOT B) A BIC B
0	0	**0**	**0**	**0**	**0**
0	1	**0**	**1**	**1**	**0**
1	0	**0**	**1**	**1**	**1**
1	1	**1**	**1**	**0**	**0**

Bitwise Logic Instruction Syntax

- 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: `AND r0,r1,r2` (in ARM)
    Equivalent to: `r0 = r1 & r2` (in C)
- Bitwise OR in Assembly
  - Example: `ORR r3, r4, r5` (in ARM)
    Equivalent to: `r3 = r4 | r5` (in C)
- Bitwise XOR in Assembly
  - Example: `EOR r0,r1,r2` (in ARM)
    Equivalent to: `r0 = r1 ^ r2` (in C)
- Bitwise Clear in Assembly
  - Example: `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
mask:	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 AND 00000111

B. x OR 00000111

C. x MOVN 00000111

D. x XOR 00000111

Uses for Logical Operators

- Finally, note that `BIC`ing a bit with 1 resets the bit (sets to 0) at the output while `BIC`ing a bit with 0 produces the original bit.
- This can be used to force certain bits of a string to 0s.
  - For example, 0x12345678 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 XOR 00000000

B. x XOR 11111111

C. x XOR 11111110

D. x 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: `MOV r0,r1` (in ARM)

    Equivalent to: `a = b` (in C)

    where ARM registers `r0, r1` are associated with C variables `a & b`

  - Example: `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)

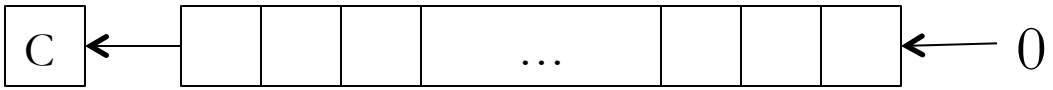
    Equivalent to: `a = -1` (in C)

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

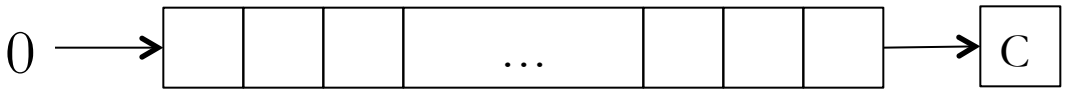
  Since `~0x00000000 == 0xFFFFFFFF`

Shifts and Rotates

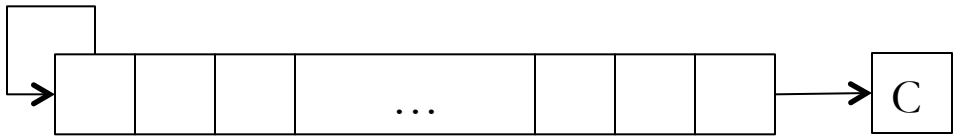
- `LSL` – logical shift by n bits – multiplication by 2^n



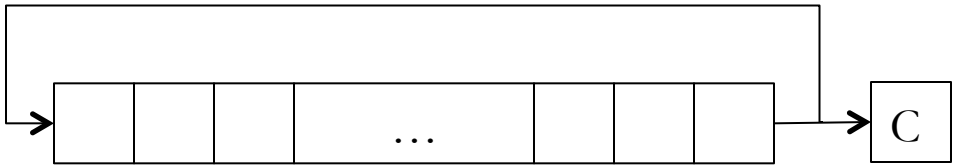
- `LSR` – logical shift by n bits – unsigned division by 2^n



- `ASR` – arithmetic shift by n bits – signed division by 2^n



- `ROR` – logical rotate by n bits – 32 bit rotate



01101001 << 2

A. 00011010

B. 00101001

C. 01101001

D. 10100100

A new instruction HEXSHIFTRIGHT shifts hex numbers over by a digit to the right.

HEXSHIFTRIGHT *i* times is equivalent to

A. Dividing by i

B. Dividing by 2^i

C. Dividing by 16^i

D. Multiplying by 16^i

A new instruction HEXSHIFTRIGHT shifts hex numbers over by a digit to the right.

HEXSHIFTRIGHT *i* times is equivalent to

A. Dividing by i

B. Dividing by 2^i

C. Dividing by 16^i

D. Multiplying by 16^i

Ways of specifying operand 2

- Opcode Destination, Operand_1, Operand_2
  - Register Direct: ADD r0, r1, r2;
    - With shift/rotate:
      1) Shift value: 5 bit immediate (unsigned integer)
         ADD r0, r1, r2, LSL #2; r0=r1+r2<<2; r0=r1+4*r2
      2) Shift value: Lower Byte of register:
         ADD r0, r1, r2, LSL r3; r0=r1+r2<<r3; r0=r1+(2^r3)*r2
  - Immediate: ADD r0, r1, #0xFF
    - With rotate-right ADD r0,r1, #0xFF, 28

    Rotate value must be even: #0xFF ROR 28 generates: 0XFF00000000

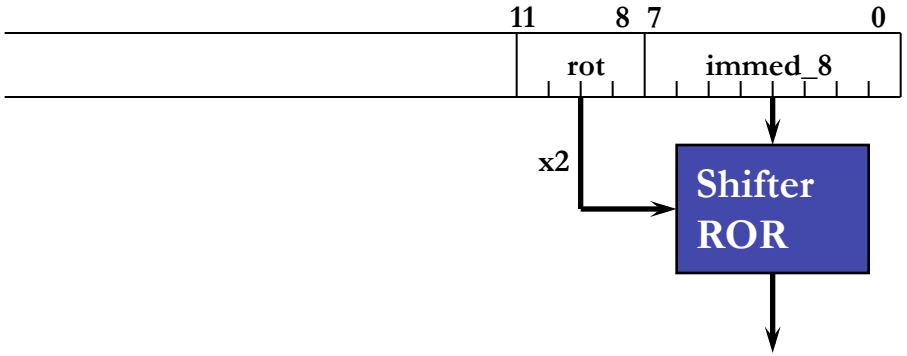
Ways of specifying operand 2

- Opcode Destination, Operand_1, Operand_2
  - Register Direct: ADD r0, r1, r2;
    - With shift/rotate:
      1) Shift value: 5 bit immediate (unsigned integer)
         ADD r0, r1, r2, LSL #2; r0=r1+r2<<2; r0=r1+4*r2
      2) Shift value: Lower Byte of register:
         ADD r0, r1, r2, LSL r3; r0=r1+r2<<r3; r0=r1+(2^r3)*r2
  - Immediate addressing: ADD r0, r1, #0xFF
    - 8 bit immediate value
    - With rotate-right ADD r0,r1, #0xFF, 8
      - Rotate value must be even

      #0xFF ROR 8 generates: 0XFF000000
      - Maximum rotate value is 30

Reasons for constraints on Immediate Addressing

- The data processing instruction format has 12 bits available for operand2




```
0xFF000000
MOV r0, #0xFF,8
```

Immed_8=0xFF, rot =4

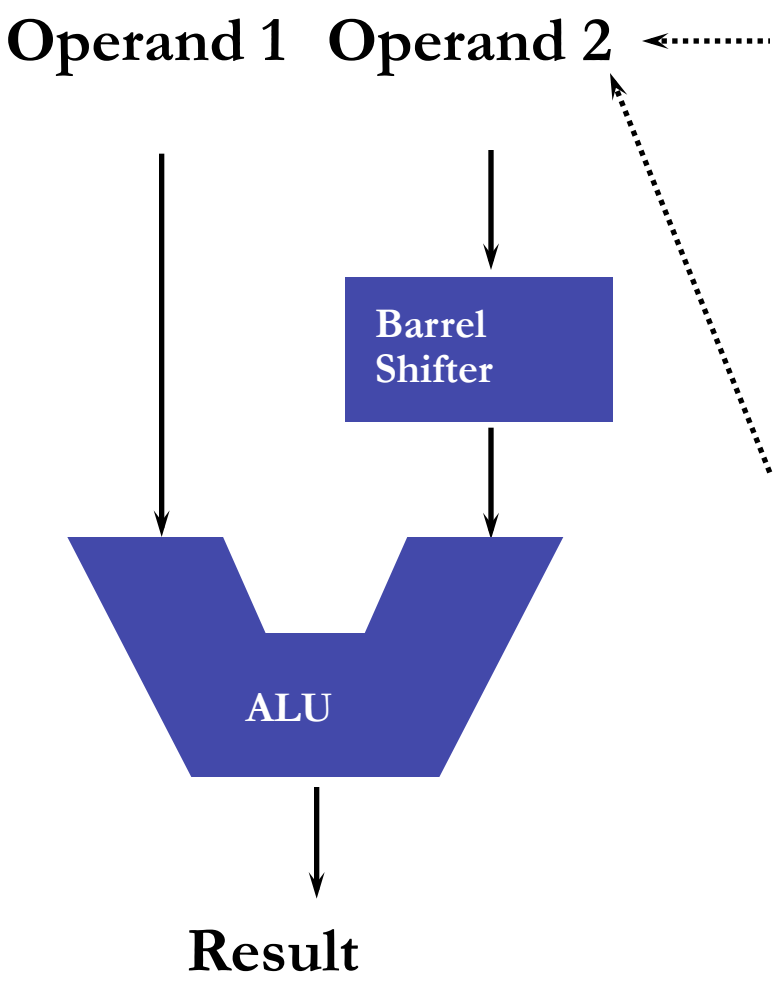
- 4 bit rotate value (0-15) is multiplied by two to give range 0-30 in steps of 2
- Rule to remember is "8-bits rotated right by an even number of bit positions"

Generating Constants using immediates

Rotate Value	Binary	Decimal	Hexadecimal
0	00000000000000000000000xxxxxxxx	0-255	0-0xFF
Right, 30 bits	00000000000000000000xxxxxxxx00	4-1020	0x4-0x3FC
Right, 28 bits	0000000000000000000xxxxxxxx0000	16-4080	0x10-0xFF0
Right, 26 bits	000000000000000000xxxxxxxx000000	128-16320	0x40-0x3FC0
…	…	…	…
Right, 8 bits	xxxxxxxx000000000000000000000000	16777216-$255x2^{24}$	0x1000000-0xFF000000
Right, 6 bits	xxxxxx00000000000000000000xx	-	-
Right, 4 bits	xxxx0000000000000000000000xxxx	-	-
Right, 2 bits	xx00000000000000000000xxxxxx	-	-

- This scheme can generate a lot, but not all, constants.
- Others must be done using literal pools (more on that later)

Implementation in h/w using a Barrel Shifter



1. Register, optionally with shift operation
   - Shift value can either be:
     - 5 bit unsigned integer
     - Specified in bottom byte of another register.
   - Used for multiplication by constant
2. Immediate value
   - 8 bit number, with a range of 0-255.
     - Rotated right through even number of positions
   - Allows increased range of 32-bit constants to be loaded directly into registers

Shifts and Rotates

- Shifting in Assembly

  Examples:

```
MOV     r4, r6, LSL #4 ; r4 = r6 << 4
MOV     r4, r6, LSR #8 ; r4 = r6 >> 8
```

- Rotating in Assembly

  Examples:

```
MOV     r4, r6, ROR #12
; r4 = r6 rotated right 12 bits
; r4 = r6 rotated left by 20 bits (32 -12)
```

  Therefore no need for rotate left.

Variable Shifts and Rotates

- Also possible to shift by the value of a register
- Examples:

```
MOV     r4, r6, LSL r3
; r4 = r6 << value specified in r3
MOV     r4, r6, LSR #8 ; r4 = r6 >> 8
```

- Rotating in Assembly
  - Examples:

```
MOV     r4, r6, ROR r3
; r4 = r6 rotated right by value specified
in r3
```

Constant Multiplication

- Constant multiplication is often faster using shifts and additions

```
MUL r0, r2, #8 ; r0 = r2 * 8
```

Is the same as:

```
MOV r0, r2, LSL #3 ; r0 = r2 * 8
```

- Constant division

```
MOV r1, r3, ASR #7 ; r1 = r3/128
```

Treats the register value like signed values (shifts in MSB).

Vs.

```
MOV r1, r3, LSR #7 ; r1 = r3/128
```

Treats register value like unsigned values (shifts in 0)

Constant Multiplication

- Constant multiplication with subtractions

```
MUL r0, r2, #7 ; r0 = r2 * 7
```

Is the same as:

```
RSB r0, r2, r2, LSL #3 ; r0 = r2 * 7
; r0 = -r2 + 8*r2 = 7*r2
```

`RSB r0, r1, r2` is the same as

```
SUB r0, r2, r1 ; r0 = r1 - r2
```

Multiply by 35:

```
ADD     r9,r8,r8,LSL #2     ; r9=r8*5
RSB     r10,r9,r9,LSL #3    ; r10=r9*7
```

Why have RSB? B/C only the second source operand can be shifted.

Conclusion

- Instructions so far:
  - Previously:

    `ADD, SUB, MUL, MLA, [U|S]MULL, [U|S]MLAL`
  - New instructions:

    `RSB`

    `AND, ORR, EOR, BIC`

    `MOV, MVN`

    `LSL, LSR, ASR, ROR`
- Shifting can only be done on the second source operand
- Constant multiplications possible using shifts and addition/subtractions

Comments in Assembly

- Another way to make your code more readable: comments!
- Semicolon (;) is used for ARM comments
  - anything from semicolon to end of line is a comment and will be ignored
- Note: Different from C
  - C comments have format /* comment */, so they can span many lines

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`