CSCI 6461 Computer Systems Architecture

Lecture 3

ISA: Basic Instructions

Addition and Subtraction Ops

Note the structure of these instructions:

`instr to, from, from`

Addition

High-level	ARM assembly
`a = b + c;`	`ADD a, b, c`

mnemonic (pronounced ni-mon-ik)

also called opcode

source operands

destination operand

The **mnemonic** indicates what operation to perform

Subtraction

High-level	ARM assembly
`a = b - c;`	`SUB a, b, c`

More complex example

More complex high-level code translates into multiple ARM instructions

```
a = b + c - d; //comment
```

ARM Assembly code:

```
ADD t, b, c   ; t = b + c
SUB a, t, d   ; a = t - d
```

**we are not naming our registers correctly yet - this is just an example

Operands

- An instruction operates on **operands (a and b here)** .
  - `ADD a, b, c`
- Operands can be stored in registers or memory, or they may be constants stored in the instruction itself.
- Operands stored as constants or in registers are accessed quickly, but they hold only a small amount of data.
- Additional data must be accessed from memory, which is large but slow.

Register Naming Convention

ARM register names are preceded by the letter R

```
; R0 = a, R1 = b, R2 = c
ADD R0, R1, R2 ; a = b + c
```

The following ARM assembly code uses a register, R4, to store the intermediate calculation:

```
ADD R4, R1, R2 ; t = b + c
SUB R0, R4, R3 ; a = t - d
```

Example

C to Assembly

C to Assembly Example

Translate the following high-level code into ARM assembly language.

```
a = b - c;
f = (g + h) - (i + j);
```

Solution

```
; R0 = a, R1 = b, R2 = c, R3 = j
; R4 = g, R5 = h, R6 = i, R7 =
  SUB  R0, R1, R2    ; a = b - c
  ADD  R8, R4, R5    ; R8 = g + h
  ADD  R9, R6, R7    ; R9 = i + j
  SUB  R3, R8, R9    ; f = (g + h) - (i + j)
```

Note

- We have not yet discussed how R0, R1 etc were initialized with values.
- Also, we used 9 registers in the previous example.
- This is probably wasteful
- In general, we will try to reduce the number of registers used in a particular stage

Constants (Immediates)

- In addition to register operations, ARM instructions can use constant or immediate operands.
- These constants are called immediates, because their values are immediately available from the instruction and do not require a register or memory access.

High-level	ARM assembly

`a = a + 4;`
`b = a - 12;`
	`; R7 = a, R8 = b`
	`ADD R7, R7, #4`
	`SUB R8, R7, #0xC`

Constants

- Hexadecimal constants in ARM assembly start with `0x`, for example
  - `MOV R4, #0xFA65`
- Single character constants, for example
  - `MOV R5, #'A'`

Logical Instructions

```
MVN – MoVe and Not
ORR – OR
EOR – XOR
BIC – bit clear
```

Source registers

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
`AND R3, R1, R2`	R3	0100 0110	1010 0001	0000 0000	0000 0000
`ORR R4, R1, R2`	R4	1111 1111	1111 1111	1111 0001	1011 0111
`EOR R5, R1, R2`	R5	1011 1001	0101 1110	1111 0001	1011 0111
`BIC R6, R1, R2`	R6	0000 0000	0000 0000	1111 0001	1011 0111
`MVN R7, R2`	R7	0000 0000	0000 0000	1111 1111	1111 1111

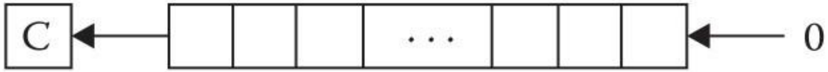
- The 1st source is always a register
- The 2nd source is either an immediate or another register
- BIC clears the bits that are asserted in R2

Shift Instructions

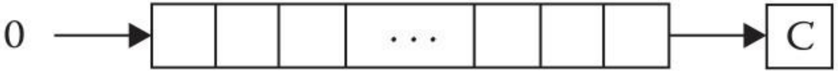
ASR – the sign bit shifts into the most significant bits.

The carry flag is updated to the last bit shifted out

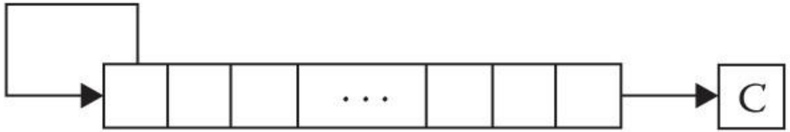
LSL Logical shift left by n bits Multiplication by 2^n



LSR Logical shift right by n bits Unsigned division by 2^n

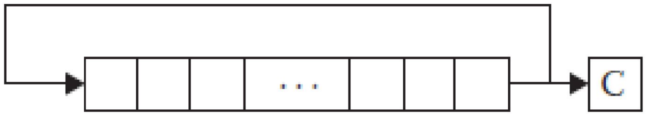


ASR Arithmetic shift right by n bits Signed division by 2^n



Rotate Right

ROR Rotate right by n bits 32-bit rotate



- The amount by which to shift can be an immediate or a register.
- We can use ROR to invert the bit pattern of a register
  - `R0 = 111000`
  - `ROR, R0, #3`
  - `R0 = 000111`
- Again, the carry flag is updated to the **last** bit rotated out

Examples, 1/2

Source register

R5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly Code		Result			
LSL R0, R5, #7	R0	1000 1110	0000 1000	0111 0011	1000 0000
LSR R1, R5, #17	R1	0000 0000	0000 0000	0111 1111	1000 1110
ASR R2, R5, #3	R2	1111 1111	1110 0011	1000 0010	0001 1100
ROR R3, R5, #21	R3	1110 0000	1000 0111	0011 1111	1111 1000

Examples, 2/2

Source registers

R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code

```
LSL R4, R8, R6
ROR R5, R8, R6
```

Result

R4	0110 1110	0111 0000	0000 0000	0000 0000
R5	1100 0001	0110 1110	0111 0000	1000 0001

No ASL

- There is no ASL, or an arithmetic shift left
- You would never need such an instruction,
    - since arithmetic shifts need to preserve the sign bit,
    - and shifting signed data to the left will do so as long as the number doesn't overflow.
    - for example:

`-1 is 0xFFFFFFFF`

shifting it left results in `0xFFFFFFFE`, which is –2 and correct

LSL as an operand

- When the final argument of basic arithmetic instruction is a register
  - we can optionally add a shift distance operator
    - LSL/LSR
  - when used in this mode, the shift operator does not change the register value it operates on
  - it merely uses the value as a parameter to the operand

LSL as a parameter

```
ADD R0, R0, R1, LSL #1
```

- This adds a left-shifted version of R1 before adding it to R0.
- R1 itself remains unchanged

The shift distance can be an immediate between 1 and 32, or it can be based on a register value:

```
MOV R0, R1, ASR R2
```

Example

```
MOV R0, #10
MOV R1, #20
ADD R0, R0, R1, LSL #1
```

What is the state of R0 and R1 at the end of this code?

Initializing values using immediates

`MOV` is a useful way to initialize register values:

High-level

```
i = 0;
x = 4080;
```

ARM assembly

```
; R4 = i, R5 = x
MOV R4, #0
MOV R5, #0xFF0
```

The instruction `MOV R5, #0xFF1` generates an error

You can overcome this using

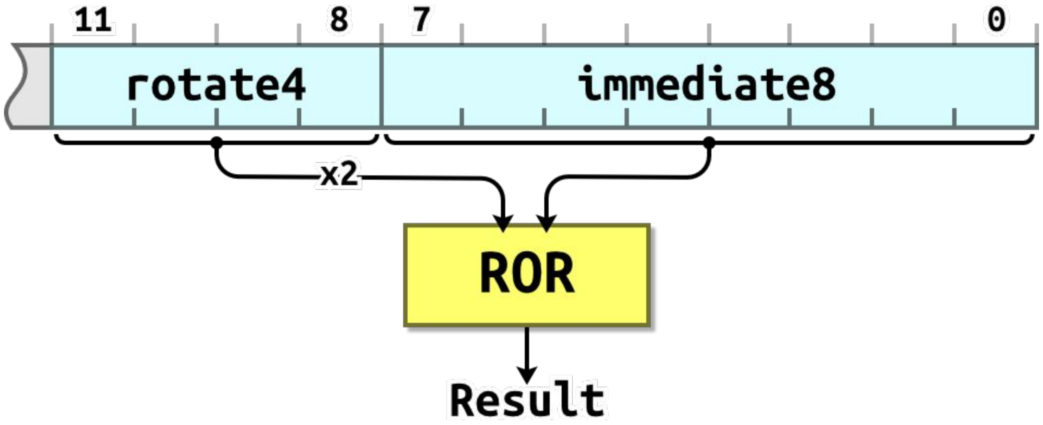
```
MOV   R5,   #0xFF0
ADD   R5,   #0x1
```

Another method:

```
MOV   R5,   #0xFF0
ORR   R5,   R5, #0x001
```

Literal Operands -

- You can’t fit an arbitrary 32-bit value into a 32-bit instruction word.
- ARM data processing instructions have 12 bits of space for values in their instruction word. This is arranged as a four-bit rotate value and an eight-bit immediate value:



Literal Operands -

- The 4-bit rotate value stored in bits 11-8 is multiplied by two giving a range of 0-30 in steps of two.
- Using this scheme we can express immediate constants such as:
  - 0x000000FF
  - 0x00000FF0
  - 0xFF000000
  - 0xF000000F
- But immediate constants such as:
  - 0x000001FE
  - 0xF000F000
  - 0x55550000
- ...are not possible.

REF Introduction to ARM: Immediate Values | DaveSpace

Multiply Instructions

```
MUL R1, R2, R3
```

- multiplies the values in R2 and R3
- and places the least significant bits of the product in R1
- the most significant 32 bits of the product are discarded

UMULL – unsigned multiply long

SMULL – signed multiply long

UMULL/SMULL

Multiply two 32-bit numbers and produce a 64-bit product

Example:

```
UMULL R1, R2, R3, R4
```

- Performs an unsigned multiply of R3 and R4
- The least significant 32 bits of the product is placed in R1 The most significant 32 bits are placed in R2

Multiply-accumulate instructions

```
MLA r7, r8, r9, r3          ;  r7 = r8 * r9 + r3
```

the least significant 32 bits of the result

```
SMLAL   r4,   r8,   r2,   r3      ;  {r8,r4}    =  r2*r3    +  {r8,r4}
UMLAL   r5,   r8,   r0,   r1      ;  {r8,r5}    =  r0*r1    +  {r8,r5}
```

MS32bits LS32bits

These instructions can boost the math performance in applications such as matrix multiplication and signal processing consisting of repeated multiplies and adds.

RSB

The **RSB** (Reverse SuBtract) instruction subtracts the value in **Rn** from the value of **Operand2** . This is useful because of the wide range of options for Operand2 .

```
RSB Rd, Rn, Src2        ; Rd ← Src2 - Rn
```

Example 1

Consider the following operation:

```
SUB r0, r2, r3, LSL #2 ; r0 = r2 - r3*4
```

Suppose we want modify (shift) register r2 before the subtraction instead of register r3.

This is done using the reverse subtract operation

```
RSB r0, r3, r2, LSL #2 ; r0 = r2*4 - r3
```

Example 2

An assembly program to perform the function of absolute value with only two instructions. Solution:

```
MOVS r1,r0
RSBLT r1, r1, #0
```

- `MOVS` performs the same function as `MOV`, but also updates the N and Z flags.
- Perform a reverse subtract if `r1` is negative. In other words: `r1 = 0 - r1` if `r1` negative

Consider the following instruction:

```
// r0 = r1 +  r1*4 =  r1*5
ADD r0, r1, r1, LSL #2;
```

- Why do it this way instead of using MUL?
- Answer: the size and power usage of a multiplier array are large.
- In very low power applications, it's often necessary to play every trick in the book to save power.

MRS/MSR "Special Registers"

MRS (Move PSR to general-purpose register) instruction to read the flags

MSR (Move general-purpose register to PSR) instruction to write the flags

```
MRS r0, CPSR ; load the contents of the CPSR into r0
MRS r1, SPSR ; load the contents of the SPSR into r1
```

From there, you can examine any flags that you like. You cannot use register r15 as the destination register.