Lecture Two: Embedded ARM Microcontrollers

4.6) Arithmetic Operations

It is important to remember that arithmetic operations (addition, subtraction, multiplication, and division) have constraints when performed with finite precision on a processor. An overflow error occurs when the result of an arithmetic operation cannot fit into the finite precision of the register into which the result is to be stored. For example, consider an 8-bit unsigned number system, where the numbers can range from 0 to 255. If we add two numbers together the result can range from 0 to 510, which is a 9-bit unsigned number. If we subtract two 8-bit unsigned numbers the result can range from -255 to +255, which is a 9-bit signed number.

The overflow bit, V, is set after a signed addition or subtraction when the result is incorrect. In the arithmetic operations below, the 32-bit value can be specified by the **#im12** constant or generated by the flexible second operand, **<op2>** . When **Rd** is absent, the result is placed back in **Rn** .

ADD{S}{cond} {Rd,} Rn, <op2> ;Rd = Rn + op2
ADD{S}{cond} {Rd,} Rn, #im12 ;Rd = Rn + im12
SUB{S}{cond} {Rd,} Rn, <op2> ;Rd = Rn - op2
SUB{S}{cond} {Rd,} Rn, #im12 ;Rd = Rn - im12
RSB{S}{cond} {Rd,} Rn, <op2> ;Rd = op2 - Rn
RSB{S}{cond} {Rd,} Rn, #im12 ;Rd = im12 - Rn
CMP{cond} Rn, <op2> ;Rn - op2
CMN{cond} Rn, <op2> ;Rn - (-op2)

The compare instructions **CMP** and **CMN** do not save the result of the subtraction or addition but always set the condition code. The compare instructions are used to create conditional execution, such as if-then, for loops, and while loops. The compiler may use **RSB** or **CMN** to optimize execution speed. If the optional **S** suffix is present, addition and subtraction set the condition code bits as shown in Table 2.7. The addition and subtraction instructions work for both signed and unsigned values. As designers, we must know in advance whether we have signed or unsigned numbers. The computer cannot tell from the binary which type it is, so it sets both C and V. Our job as programmers is to look at the C bit if the values are unsigned and look at the V bit if the values are signed.

Bit	Name	Meaning after addition or subtraction
N	Negative	Result is negative
Z	Zero	Result is zero
V	Overflow	Signed overflow
C	Carry	Unsigned overflow

Table 2.7. Condition code bits contain the status of the previous arithmetic operation.

If the two inputs to an addition operation are considered as unsigned, then the C bit (carry) will be set if the result does not fit. In other words, after an unsigned addition, the C bit is set if the answer is wrong. If the two inputs to a subtraction operation are considered as unsigned, then the C bit (carry) will be clear if the result does not fit. If the two inputs to an addition or subtraction operation are considered as signed, then the V bit (overflow) will be set if the result does not fit. In other words, after a signed addition, the V bit is set if the answer is wrong. If the result is unsigned, the N=1 means the result is greater than or equal to 2^{31}. Conversely, if the result is signed, the N=1 means the result is negative.

Example 2.4: Write code that reads from variable **N** adds 10 and stores the result in variable **M**. Both variables are 32-bit.

Solution: First, we perform a 32-bit read, bringing **N** into Register R1. Second we add 10, and lastly we store the result into **M**. Since the value gets larger, no overflow could occur. This solution ignores the overflow error.

```
LDR R3, =N     ; R3 = &N (R3 points to     // C implementation
N)                                          M = N+10;
 LDR R1, [R3]   ; R1 = N
 ADD R0, R1, #10 ; R0 = N+10
 LDR R2, =M     ; R2 = &M (R2 points
to M)
 STR R0, [R2]   ; M = N+10
```

Program 2.4. Example code showing a 32-bit add.

Multiplication and division occurring in computers utilize a variety of complex algorithms to reduce power and minimize execution time. However, to illustrate binary multiplication we will present a very simple 8-bit unsigned algorithm, which uses a combination of shift and addition operations. Let A and B be two unsigned 8-bit numbers. The goal is to make R=A•B. Simple calculations of 0•0=0 and 255•255=65025 illustrate the fact that the multiplication of two 8-bit numbers will fit into a 16-bit product. In general, an n-bit number multiplied by an m-bit number yields an (n+m)-bit product. First, we define one of the multiplicands in terms of its basis representation.

B = 128•b7 + 64•b6 + 32•b5 + 16•b4 + 8•b3 + 4•b2 + 2•b1 + b0

Next, we distribute multiplication over addition

R = A•128•b7 + A•64•b6 + A•32•b5 + A•16•b4 + A•8•b3 + A•4•b2 + A•2•b1 + A•b0

We can simplify the equation leaving only one-bit shifts

R = 2•(2•(2•(2•(2•(2•(2• A•b7+ A•b6)+ A•b5)+ A•b4)+ A•b3)+ A•b2)+ A•b1)+ A•b0

The multiplication by a power of 2 is a logical shift left, and the multiplication by a binary bit (0 or 1) is an add or no-add conditional. For an 8-bit multiply, we will use 16-bit shifts and additions, yielding a 16-bit product. Since the product, R, is a 16-bit unsigned number, there can be no overflow error in this 8 by 8 into 16-bit multiply.

Multiply (MUL), multiply with accumulate(MLA), and multiply with subtract(MLS) use 32-bit operands and produce a 32-bit result. These three multiply instructions only save the bottom 32 bits of the result. They can be used for either signed or unsigned numbers, but no overflow flags are generated. If the Rd register is omitted, the Rn register is the destination. If the S suffix is added to MUL , then the Z and N bits are set according to the result. The division instructions do not set condition code flags and will round towards zero if the division does not evenly divide into an integer quotient.

MUL{S}{cond} {Rd,} Rn, Rm ;Rd = Rn * Rm

MLA{cond} Rd, Rn, Rm, Ra ;Rd = Ra + Rn*Rm

MLS{cond} Rd, Rn, Rm, Ra ;Rd = Ra - Rn*Rm

UDIV{cond} {Rd,} Rn, Rm ;Rd = Rn/Rm unsigned

SDIV{cond} {Rd,} Rn, Rm ;Rd = Rn/Rm signed

Example 3.6: Write code that reads from variable **N** multiplies by 5, adds 25, and stores the result in variable **M**. Both variables are 32-bit.

Solution: First, we perform a 32-bit read, bringing **N** into Register R1. Second we multiply by 5 and add 10, and lastly we store the result into **M**. Since the value gets larger, overflow could occur. This solution ignores the overflow error.

```
 LDR R3, =N     ; R3 = &N (R3 points to    // C implementation
N)                                          M = 5*N+25;
 LDR R1, [R3]   ; R1 = N
 MOV R0, #5     ; R0 = 5
 MUL R1, R0, R1 ; R1 = 5*N
 MOV R0, #25    ; R0 = 25
 ADD R0, R0, R1 ; R0 = 25+5*N
 LDR R2, =M     ; R2 = &M (R2 points
to M)
 STR R0, [R2]   ; M = 25+5*N
```

Program 2.8. Example code showing a 32-bit multiply and addition.

Example 2.7: Write code to convert a variable **N** ranging from 0 to 1023 into a variable **M**, which ranges from 0 to 3000. Essentially compute **M** = 2.93255* **N**. Both variables are 32-bit.

Solution: First, we perform a 32-bit read, bringing **N** into Register R1. Second we multiply by 3000 and divide by 1023, and lastly we store the result into **M**.

```
 LDR R3, =N     ; R3 = &N (R3 points to    // C implementation
N)                                          M = (3000*N)/1023;
 LDR R1, [R3]    ; R1 = N
 MOV R0, #3000  ; R0 = 3000
 MUL R1, R0, R1  ; 3000*N (0 to
3069000)
 MOV R0, #1023  ; R0 = 1023
 UDIV R0, R1, R0 ; R0 = R1/R0 =
3000*N/1023
 LDR R2, =M     ; R2 = &M (R2 points
to M)
 STR R0, [R2]    ; M = (3000*N)/1023
```

3.3.7. Stack

The **stack** is a last-in-first-out temporary storage. To create a stack, a block of RAM is allocated or this temporary storage. On the ARM ® Cortex™-M processor, the stack always operates on 32-bit data. The stack pointer (SP) points to the 32-bit data on the top of the stack. The stack grows downwards in memory as we push data on to it so, although we refer to the most recent item as the "top of the stack" it is actually the item stored at the lowest address! To push data on the stack, the stack pointer is first decremented by 4, and then the 32-bit information is stored at the address specified by SP. To pop data from the stack, the 32-bit information pointed to by SP is first retrieved, and then the stack pointer is incremented by 4. SP points to the last item pushed, which will also be the next item to be popped. The boxes in Figure 2.27 represent 32-bit storage elements in RAM. The grey boxes in the figure refer to actual data stored on the stack, and the white boxes refer to locations in memory that do not contain stack data. This figure illustrates how the stack is used to push the contents of Registers R0, R1, and R2 in that order. Assume Register R0 initially contains the value 1, R1 contains 2, and R2 contains 3. The drawing on the left shows the initial stack. The software executes these six instructions in this order:

PUSH {R0}
PUSH {R1}
PUSH {R2}
POP {R3}
POP {R4}
POP {R5}

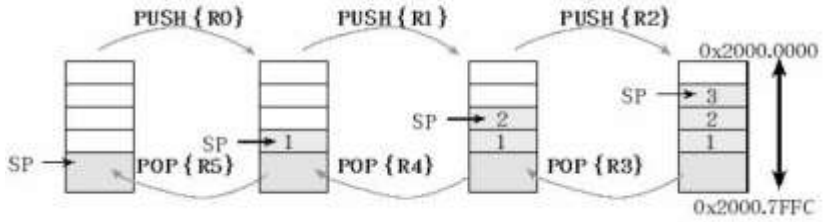

Figure 2.27. Stack picture showing three numbers first being pushed, then three numbers being popped. You are allowed to draw stack pictures so that the lowest address is on the top (like this one) or so that lowest address is on the bottom.

The important matter is to be clear, accurate, and consistent. The instruction **PUSH {R0}** saves the value of R0 on the stack. It first decrements SP by 4, and then it stores the contents of R0 into the memory location pointed to by SP. The right-most drawing shows the stack after the push occurs three times. The stack contains the numbers 1, 2,and 3, with 3 on top. The instruction **POP{R3}** retrieves data from the stack. It first moves the value from memory pointed to by SP into R3, and then it increments SP by 4. After the pop occurs three times the stack reverts to its original state and registers R3, R4, and R5 contain 3 2 1 respectively. We define the 32-bit word pointed to by SP as the **top** entry of the stack. If it exists, we define the 32-bit data immediately below the top, at SP+4, as **next** to top.

3.3.8. Functions and Control Flow

Normally the computer executes one instruction after another in a linear fashion. In particular, the next instruction to execute is found immediately following the current instruction. More specifically, we use branch instructions to deviate from this straight line path. Table 2.2 lists the conditional execution available on the ARM ® Cortex™-M processor. In this section, we will use the conditional branch instruction to implement if-then, while loop, and for-loop control structures.

B{cond} label ;branch to label

BX{cond} Rm ;branch indirect to location specified by Rm

BL{cond} label ;branch to subroutine at label

BLX{cond} Rm ;branch to subroutine indirect specified by Rm

Subroutines, **procedures**, and **functions** are code sequences that can be called to perform specific tasks. They are important conceptual tools because they allow us to develop modular software. In assembly language, we use the term subroutine for all subprograms whether or not they return a value. In this section we present a short introduction on the syntax for defining subroutines. We define a subroutine by giving it a name in the label field, followed by instructions, which when executed, perform the desired effect. The last instruction in a subroutine will be **BX LR**, which we use to return from the subroutine.

In Program 2.9, we define the subroutine named **Change**, which adds 25 to the variable **Num** . In assembly language, we will use the **BL** instruction to call this subroutine. At run time, the **BL** instruction will save the return address in the LR register. The return address is the location of the instruction immediately after the **BL** instruction. At the end of the subroutine, the **BX LR** instruction will retrieve the return address from the LR register, returning the program to the place from which the subroutine was called. More precisely, it returns to the instruction immediately after the instruction that performed the subroutine call. The comments specify the order of execution. The while-loop causes instructions 4–10 to be repeated over and over.

In C, input parameters, if any, are passed in R0–R3. If there are more than 4 input parameters, they are pushed on the stack. The output parameter, if needed, is returned in R0.

```
Change LDR  R1,=Num  ; 5) R1 =                uint32_t Num;
&Num                                           void Change(void){
    LDR  R0,[R1]  ; 6) R0 = Num                 Num = Num+25;
    ADD  R0,R0,#25 ; 7) R0 = Num+25            }
    STR  R0,[R1]  ; 8) Num = Num+25            void main(void){
    BX  LR      ; 9) return                     Num = 0;
main  LDR  R1,=Num  ; 1) R1 = &Num             while(1){
    MOV  R0,#0    ; 2) R0 = 0                   Change();
    STR  R0,[R1]  ; 3) Num = 0                  }
loop  BL   Change   ; 4) function call         }
    B    loop    ; 10) repeat
```

Program 2.9. Assembly and C versions that define a simple function.

Decision making is an important aspect of software programming. Two values are compared and certain blocks of program are executed or skipped depending on the results of the comparison. In assembly language it is important to know the precision (e.g., 16-bit, 32-bit) and the format of the two values (e.g., unsigned, signed). It takes three steps to perform a comparison. We begin by reading the first value into a register. The second step is to compare the first value with the second

value. We can use either a subtract instruction (**SUBS**) or a compare instruction (**CMP**). These instructions set the condition code bits. The last step is a conditional branch. The available conditions are listed in Table 2.2. The branch will occur if the condition is true.

Program 2.10 illustrates an if-then structure involving testing for unsigned greater than or equal to. It will increment **Num** if it is less than 25600. Since the variable is unsigned, we use an unsigned conditional. Furthermore, we want to execute the increment if **Num** is less than 25600, so we perform the opposite conditional branch (greater than or equal to) to skip over.

```
Change LDR  R1,=Num  ; R1 = &Num      uint32_t Num;
     LDR  R0,[R1]  ; R0 = Num         void Change(void){
     CMP  R0,#25600                    if(Num < 25600){
     BHS  skip                           Num = Num+1;
     ADD  R0,R0,#1  ; R0 = Num+1       }
     STR  R0,[R1]  ; Num = Num+1      }
skip BX   LR       ; return
```

Program 2.10. Software showing an if-then control structure (BHS used because it is unsigned).

Program 2.11illustrates an if-then-else structure involving signed numbers. It will increment **Num** if it is less than 100, otherwise it will set it to -100. Since the variable is signed, we use assigned conditional. Again, we want to execute the increment if **Num** is less than 100, so we perform the opposite conditional branch (greater than or equal to) to skip over.

```
Change LDR  R1,=Num  ; R1 = &Num      int32_t Num;
     LDR  R0,[R1]  ; R0 = Num         void Change(void){
     CMP  R0,#100                      if(Num < 100){
     BGE  else                           Num = Num+1;
     ADD  R0,R0,#1  ; R0 = Num+1       }
     B    skip                         else{
else MOV  R0,#-100 ; -100                Num = -100;
skip STR  R0,[R1]  ; update Num        }
     BX   LR       ; return          }
```

Program 2.11. Software showing an if-then-else control structure (BGE used because it is signed).