# Why?

Assembly language is the “native language” of the computer processor. Therefore, for high-level (e.g., C) programs to run on the computer, we need to translate (i.e., compile) them into assembly language.

# Model 1: Addressing

Assume this ***initial*** contents of memory and registers.

|  |  |
| --- | --- |
| Address | 4-byte value |
| 0x10010000 | ? - 10991 |
| 0x10010004 | 1941 |
| 0x10010008 | ? |
| 0x1001000C | 877 |
| 0x10010010 | ? |
| 0x10010014 | ? – 0x010010004 |

|  |  |
| --- | --- |
| register | value |
| R0 | ? - 1941 |
| R1 | 0 - 877 |
| R12 | 0x10010004 |
| R3 | 10991 |

These instructions have the following effects

1. ldr R0, [R12, #0] puts value 1941 into R0
2. ldr R1, [R1, #0x1001000C] puts value 877 into R1
3. str R3, [R12, #-4] puts value 10991 into memory location 0x10010000
4. str R12, [R12, #16] puts value 0x10010004 into memory location 0x10010014
5. What memory address is accessed in instruction (a)? Is it a memory read or memory write?

0x10010004; memory read

1. What memory address is accessed in instruction (b)? Is it a memory read or memory write?

0x1001000C. Read

1. What memory address is accessed in instruction (c)? Is it a memory read or memory write?

0x10010000. Memory write.

1. What memory address is accessed in instruction (d)? Is it a memory read or memory write?

0x10010014; Memory write.

1. Does ldr read or write memory? What is the purpose of ldr’s first argument?

Ldr reads memory; the first argument is where is stores the memory it’s reading in the register.

1. Does str read or write memory? What is the purpose of str’s first argument?

Str writes memory. The first argument is where it reads from a register what value it will store.

1. For each instruction, write out the calculation of the memory address it accesses.
2. Accesses R12, whose address is 0x10010004, adds 0, so address is still 0x10010004, whose value is 1941.
3. Accesses R1, whose value is 0, adds address 0x1001000C, so accesses value there which is 877, and stores that value in R1.
4. Looks at value in R12, which is 0x10010004, subtracts 4 for 0x10010000, and stores the value in R3 (which is 10991) at memory address 0x10010000.
5. Looks at value in R12, which is 0x10010004, adds 16 for 0x10010014, stores the value in R12 (which is memory address 0x10010004) at memory address 0x10010014.
6. Generalize. For ldr and str, write a formula for calculating the memory address.

Adds the two values in brackets to get to an address.

Address = [A1 + A2]

1. From the ARM reference sheet, copy down the RTL descriptions of behavior of the ldr and str. Compare the RTL descriptions with your answers to #5, 6, and 8.

ldr: Rd := [Rb +/- offset]

str: [Rb +/- offset] := Rs

# Exercises

1. What is the contents of memory and registers when the program finishes?

mov R0, #0x3000

mov R1, #0x108

str R1, [R0, #20]

add R1, R1, #4

str R1, [R0, #24]

add R0, R0, #16

ldr R12, [R0, #4]

# Model 2: if-statements

Assume x’s value is in R0

Assume y’s value is in R1

|  |  |  |
| --- | --- | --- |
| C program | ARM program A | ARM program B |
| **if** (x > 4) {  y -= 1;  } **else** {  y += 1;  } | 1 cmp R0, #4  2 ble A  3 add R1, R1, #-1  4 b B  5 A: add R1, R1, #1  6 B: | cmp R0, #4  addgt R1, R1, #-1  addle R1, R1, #1 |
| **if** (x == 100) {  y += 4;  } **else** **if** (x <= 3) {  y -= 10;  } **else** {  y = 0;  x = 0;  } | 7 cmp R0, #100  8 bne elifCase  9 add R1, R1, #4  10 b endIf  11 elifCase: cmp R0, #3  12 bgt elseCase  13 add R1, R1, #-10  14 b endIf  15 elseCase: mov R1, #0  16 mov R0, #0  17 endIf: | cmp R0, #100  addeq R1, R1, #4  beq end  cmp R0, #3  addle R1, R1,#-10  movgt R1, #0  movgt R0, #0  end: |

Legend

* abc: label line with the symbol abc
* currentLine means the line of the ARM program that is currently executing
* first program starts at currentLine = 1
* second program starts at currentLine = 7
* find meaning of conditions eq, ne, gt, le, ge on ARM reference sheet “condition codes” or “condition field”

|  |  |
| --- | --- |
| Syntax | RTL |
| b abc | currentLine 🡨 line labeled abc |
| cmp Rn, operand | compare Rn with operand and set CPSR, which influences subsequent instructions |
| instructions whose name ends with a condition  e.g., bCOND, addCOND, movCOND | if (Rn COND operand)  execute this instruction  else  do nothing |

1. Match these lines of C code to their line numbers in ARM program A and ARM program B.
   1. y -= 1 : 3 : 2
   2. y += 1 : 5 : 3
   3. y += 4 : 9 : 8
   4. y -= 10 : 13 : 11
   5. y = 0 : 15 : 12
   6. x = 0 : 16 : 13
2. What condition is being tested on lines 1-2 of ARM program A? Compare this condition to the corresponding condition in the C version and ARM program B.

Line 1 is comparing R0 to the number 4. Then, line two is branching if less than or equal to 4. In ARM program B, line 2 adds if it is greater than, and line 3 adds if less than.

1. When that instruction’s condition is true/false, what line will execute next?
   1. When True – 3 : 2
   2. When False – 5 : 3
2. What condition is being tested in lines 7-8 of ARM program A? Compare this condition to the corresponding condition in the C version and ARM program B.

Line 7 compares to the number 100. Then. 8 branches to elifCase if it’s not equal to 100. In ARM program B, it compares it the same way in line 7, but then adds 4 if it’s equal.

1. When that instruction’s condition is true/false, what line will execute next?
   1. When True – adds 4 (line 9)
   2. When False - Branches to elifCase.
2. What condition is being tested in lines 11-12 of ARM program A? Compare this condition to the corresponding condition in the C version and ARM program B.

Line 11 compares R0 to the number 3. Line 12 branches to elseCase if R0 is greater than 3. In ARM program B, it compares it in the same way, but adds right away if less than, or sets them equal to 0 if greater than.

1. When that that instruction’s condition is true/false, what line will execute next?
   1. When True – Adds -10 to R1.
   2. When False – Branches to elseCase.
2. Explain, in general, how to translate an if-statement to ARM using program A’s approach. You could do this in words or by showing a C template and a corresponding ARM template.

First you have to compare a register to a number (or another register). Then, you have to create ‘branches’ for each case you have. If it’s just an if/else, then there’s two “branches”: one is for the else case, and one is for the after if/else statement (where the rest of the program’s code would go). After you compare, you decide on what conditions you want to branch. Bne else means branch to else: if not equal to whatever you just compared. So, the program will jump to the line with else: in it, skipping any code before that. If the comparison is equal, however, the Bne will not be executed, so the code under that will be executed. However, if we don’t add B end: (assuming our other branch is called end) before the first line of else:, then the program will continue to run through that branch.

1. Explain, in general, how to translate an if-statement to ARM using program B’s approach. You could do this in words or by showing a C template and a corresponding ARM template.

Program B’s approach uses the same comparison and uses less branches. If there is no else if, then there are no branches. It continues through, using the same comparison, and does any calculation you want on the condition of being equal to, greater than, or less than.

# Exercises

1. Translate this program to ARM using approach A. Assume x is in R0, y is in R1, z is in R2.

if (x < 50) {

if (y < 40) {

z = x + y;

} else {

z = y;

y = 0;

}

} else {

z = x;

x = 0;

}

1. Translate this program to ARM using approach B.