Any function can be implemented in Boolean logic, and Boolean logic can be implemented by physical devices- gates

– Boolean operators AND (&), OR (|), NOT(~) , Precedence(high to low): ~, &, |

Canonical representation: every Boolean function(defined in a truth) can be expressed using at least one Boolean expression called the canonical representation

SUM-OF-PRODUCT canonical form:

Procedure: 1. AND together all literals (negate if 0) in each row (conjunct) 2. OR together rows that have true output 3. Repeat for each output bit of the function

PRODUCT-OF-SUM canonical form:

Procedure: 1. OR together all literals (negate if true) in each row 2. AND together rows that have false output 3. Repeat for each output bit of the function

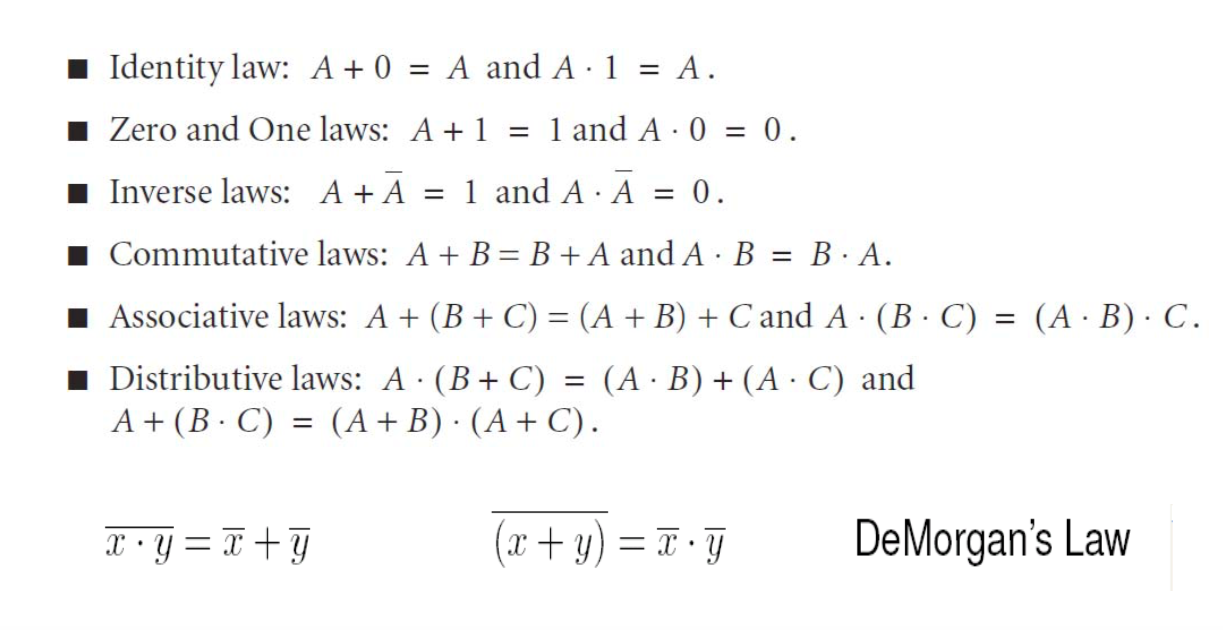
The number of Boolean functions that can be defined over n Boolean variables is 2^(2^n)

How to wire the gates according to a Boolean expression?

Can implement any Boolean expression with a collection of gates:

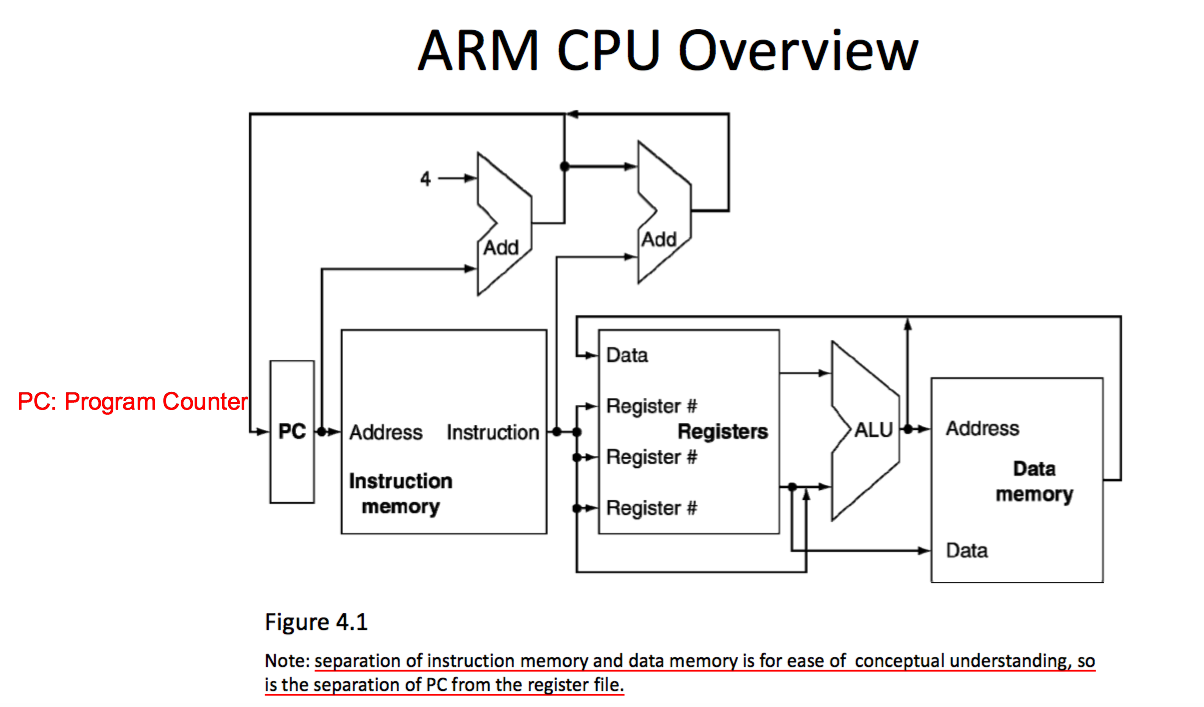
1. Find outer-most operator
2. Replace with gate
3. Work recursively on input functions

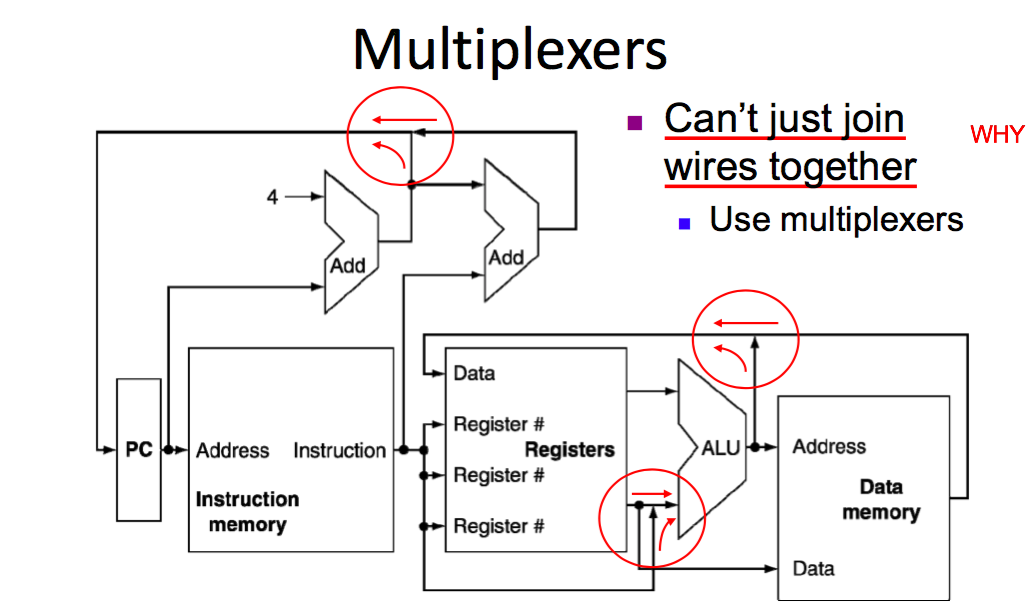
Only a small number of primitive gates(like NAND) would be enough to implement all the Boolean logics

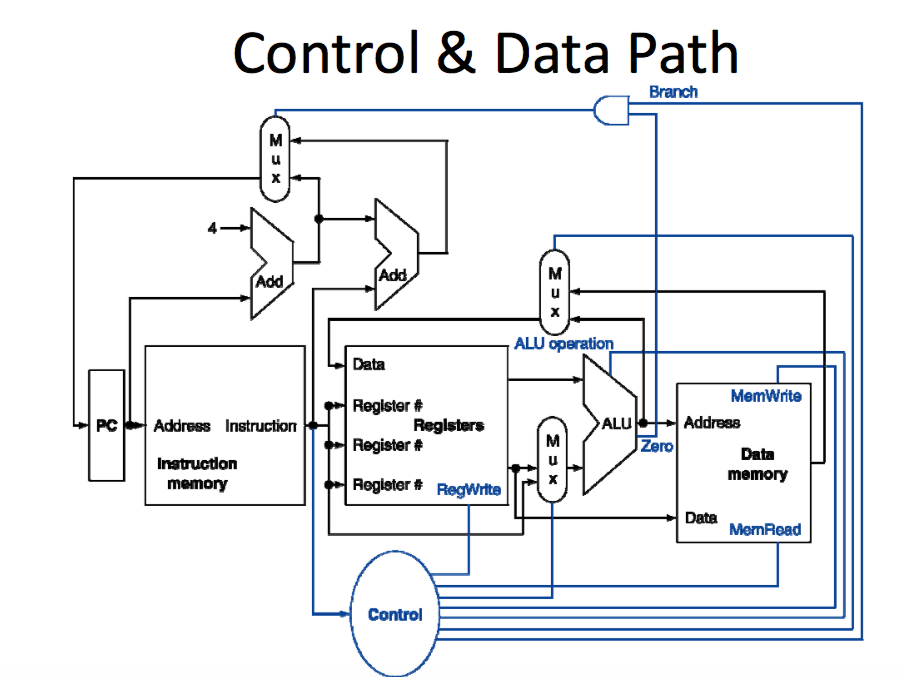


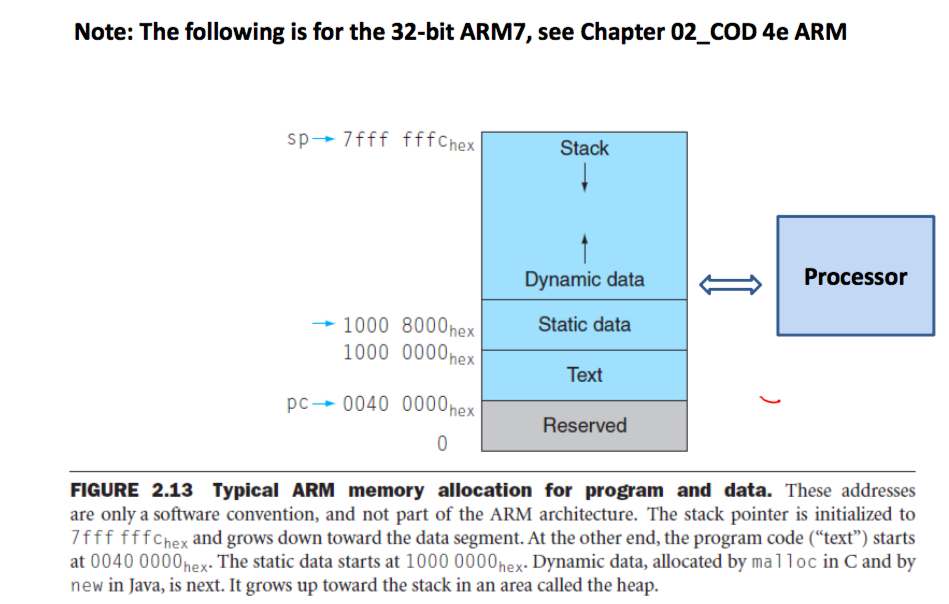
Sequential logic V.S. combinational logic

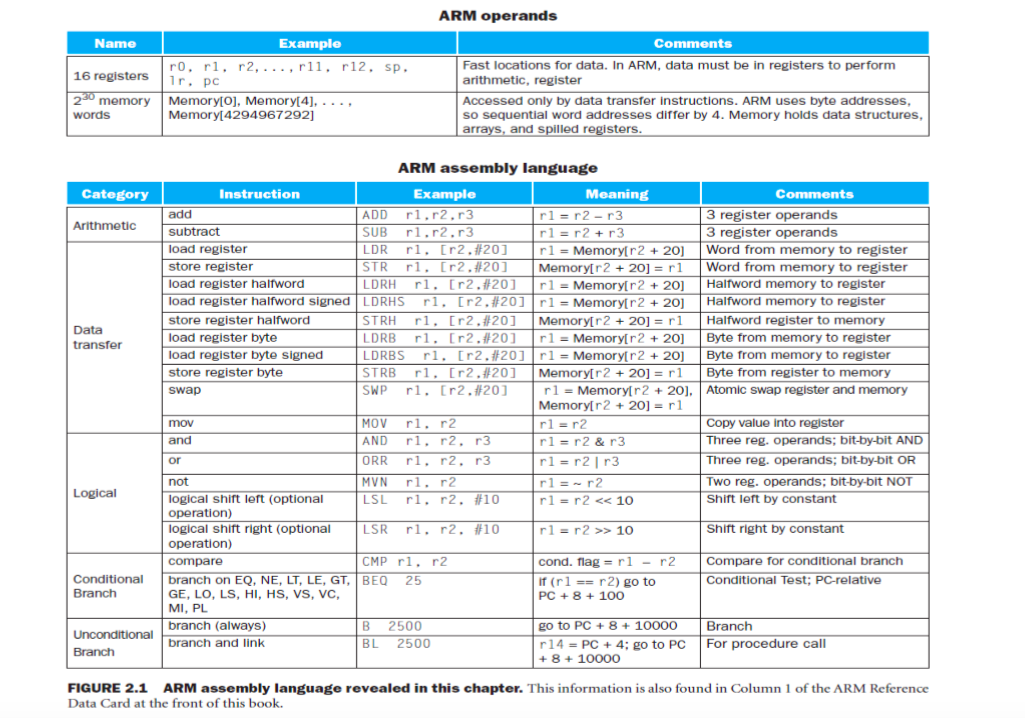
Logic devises can be of 2 types: one is combinational and the other is sequential. While for combinational, the output is determined by the input solely, for sequential, the output is determined by the order/history and the input





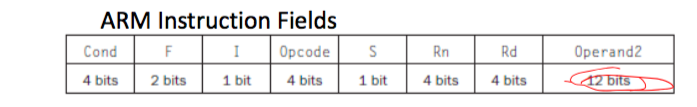






Features of ARM instruction set:

* Load-store architecture
* 3-address instructions
* Conditional execution of every instruction
* Possible to load/store multiple register at once
* Possible to combine shift and ALU operations in a single instruction



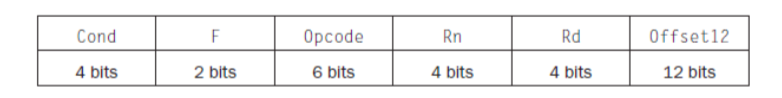
* Opcode: basic operation of the instruction, traditionally called the opcode
* Rd: The register destination operand. It gets the result of the operation
* Rn: The first register source operand
* Operand2: the second source operand
* I: Immediate. If I is 0, the second source operand is a register. If I is 1, the second source operand is a 12-bit immediate
* S: Set Condition Code. This field is related to conditional branch instructions
* Cond: Condition. This field is related to conditional branch instructions
* F: Instruction Format. This field allows ARM to different instruction formats when needed

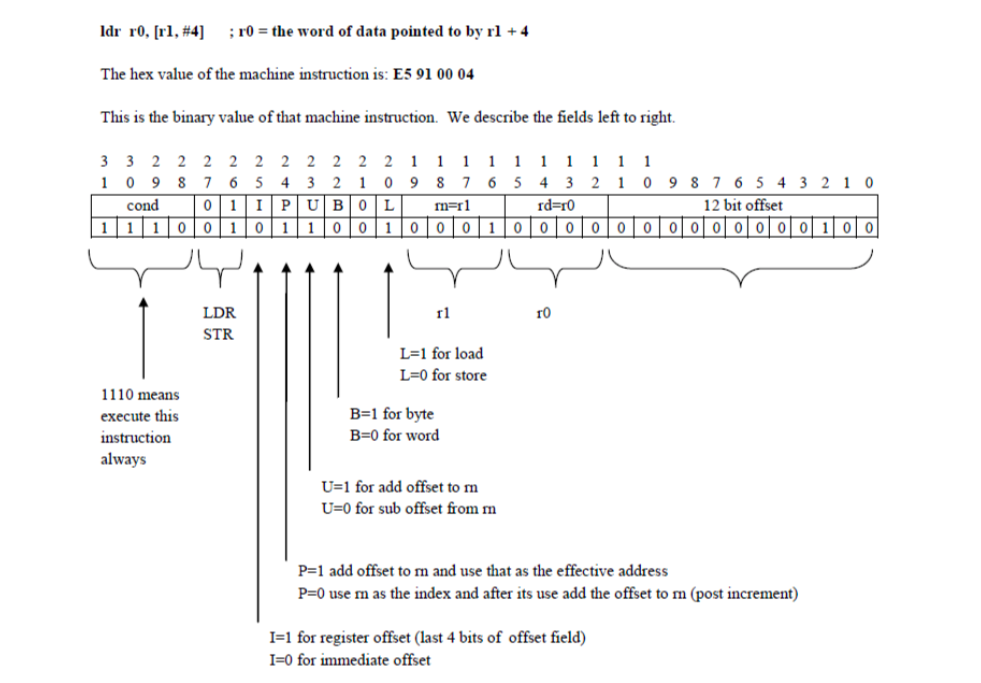
e.g.: ADD r3, r3, #4 ; r3=r3+4

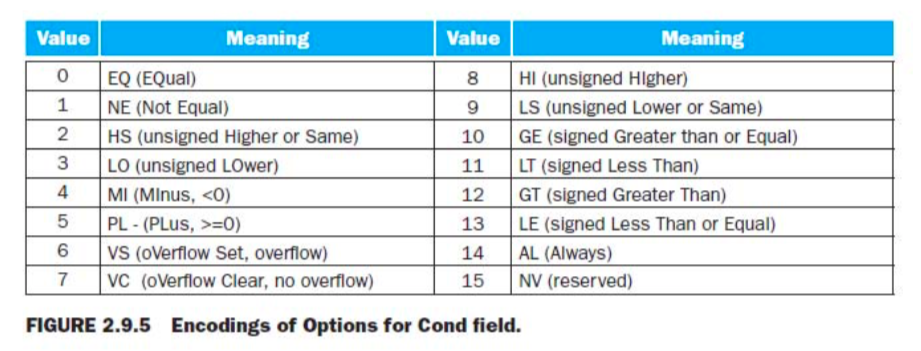
Explanation: The constant 4 is placed in the Operand2 field and the I field is set to 1.

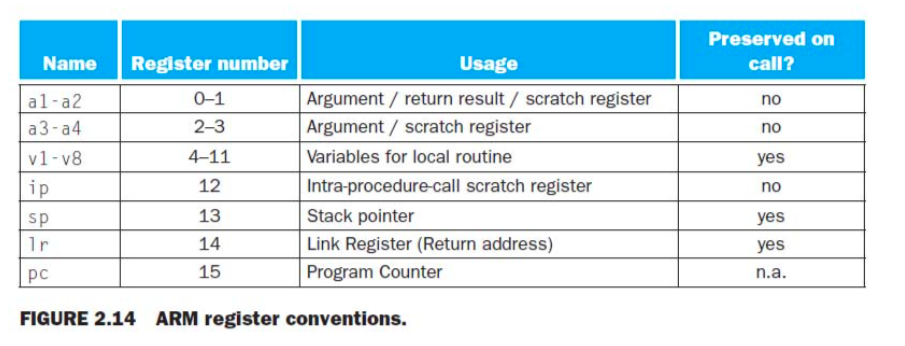
e.g.: LDR r5, [r3, #32]. ; Temporary reg r5 gets A[8]

**Loads and stores use a different instruction format from above, with just 6 fields, and to tell ARM that the format is different, the F field now as 1, meaning that this is a data transfer instruction format. The operand field has 24, showing that this instruction does load word. The rest of the fields are straightforward: Rn field has 3 for the base register, the Offset12 field has 32 as the offset to add to the base register, and the Rd field has 5 for the destination register, which receives the result of the load**









When you target the GNU assembler for ARM, each assembling line has the format:

{<label>:} {<instruction or directive>} @comment

Labels are recognized by the following colon rather than their position at the start of the line.

e.g.:

.section .text, “x”

.global. add @give the symbol add external linkage

Add:

ADD r0, r0, r1. @add input arguments

MOV pc, lr @return from subroutine

Assembler directives”:

.align n

.ascii “<string>”

.asciiz “<string>”

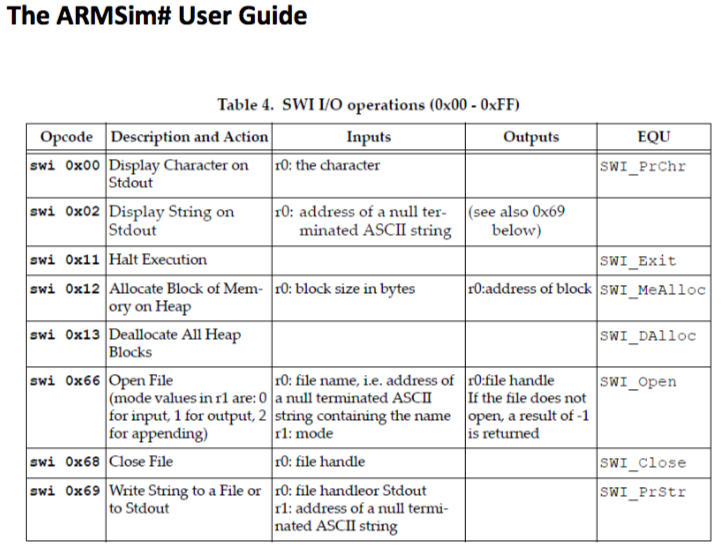
.byte <byte1> {, <byte2>, …}

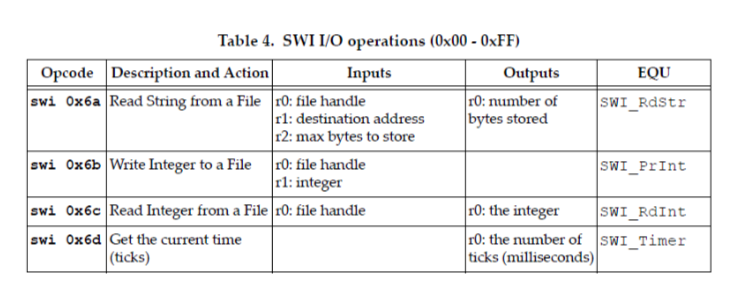
.data {<addr>}

.global <symbol>

.text {<addr>}

.word <word1> {, <word2>, …}





e.g.:

.data

.str: .asciz “Hello World!\n”

.text

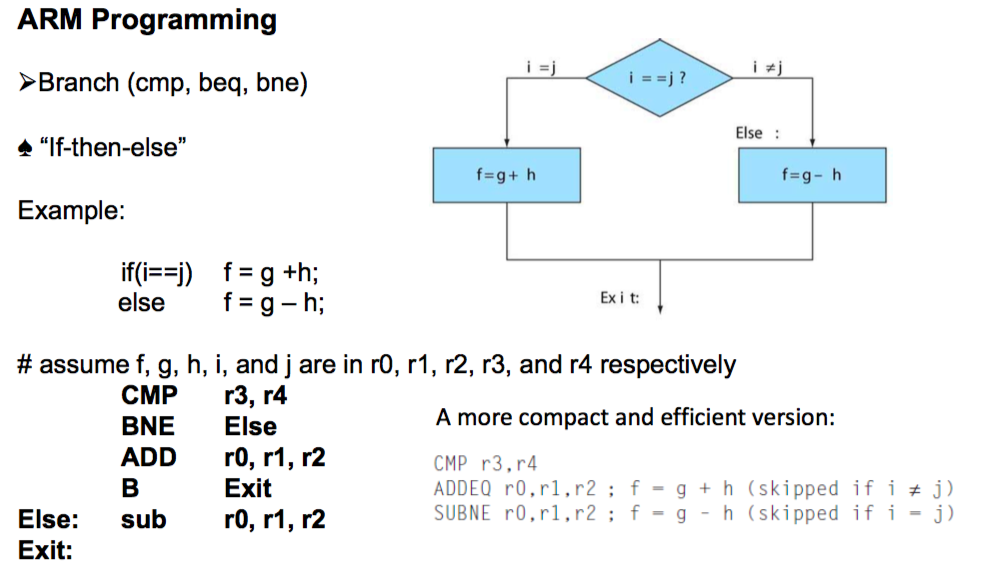
.align2

.global main

Main: MOV r0, #1

LDR r1, =str

SWI 0x69



This example is in page 108 ‐ 109, of Chapter 02\_COD 4e ARM.pdf).

r0 = 1111 1111 1111 1111 1111 1111 1111 1111

r1 = 0000 0000 0000 0000 0000 0000 0000 0001

So, the result of r0 ‐ r1 is computed (**see page B1‐15 in Appendix\_B1.pdf for details**) as r0 + (‐r1) = r0 + (~r1 +1)

1111 1111 1111 1111 1111 1111 1111 1111 (r0)

+) 1111 1111 1111 1111 1111 1111 1111 1111 (~r1 + 1)

‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐

11111 1111 1111 1111 1111 1111 1111 1110

Based on this result, N = 1 (the result is negative, treated as two's complement) Z = 0 (the result is not zero) C = 1 (there is carry out of the left‐most bit) V = 0 (there is no overflow)

Therefore, the instruction "BLO" is not taken because of suffix "LO" indicates unsigned lower which is taken when carry bit is clear (**See Table B1.2, in Appendix\_B1.pdf for details**). Instead instruction "BLT" is taken when N != V, which is the case as shown above.

How do we make use of a certain piece of code:

Solution1: cut-and-paste: embed the code of MUL to where it is needed

Solution2: “call-by-name”: Jump to the MUL code and jump back when it is done

However, when you have multiple returning addresses with the same name, a hard-coded “call-by-name” is not working well- it can only return to one place

Solutions:

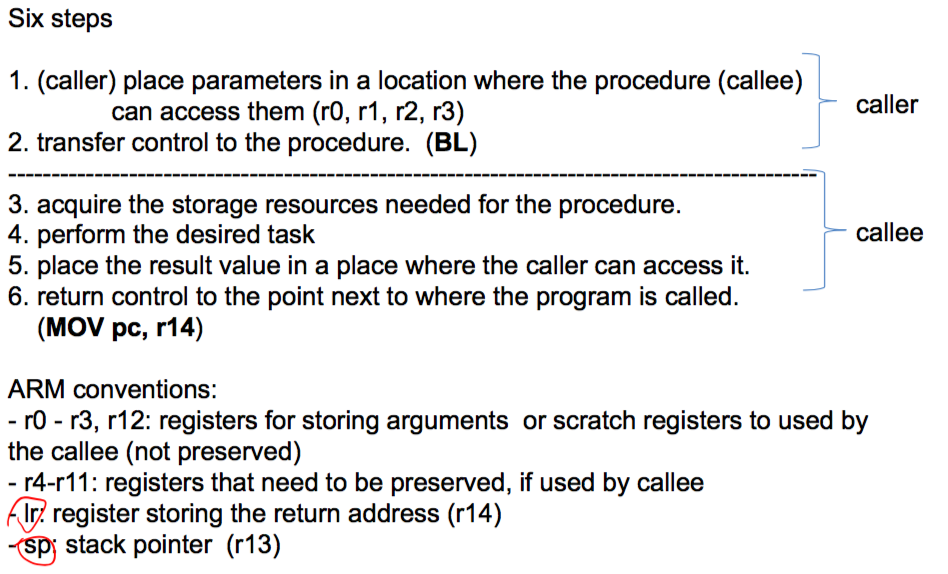
1. Save the return address to a register before calling the subroutine
2. Jump to the saved return address when the subroutine is done

Hardware support:

Bl ProcedureAddress: @save the return address (pc+4) into a designated register r14 and then

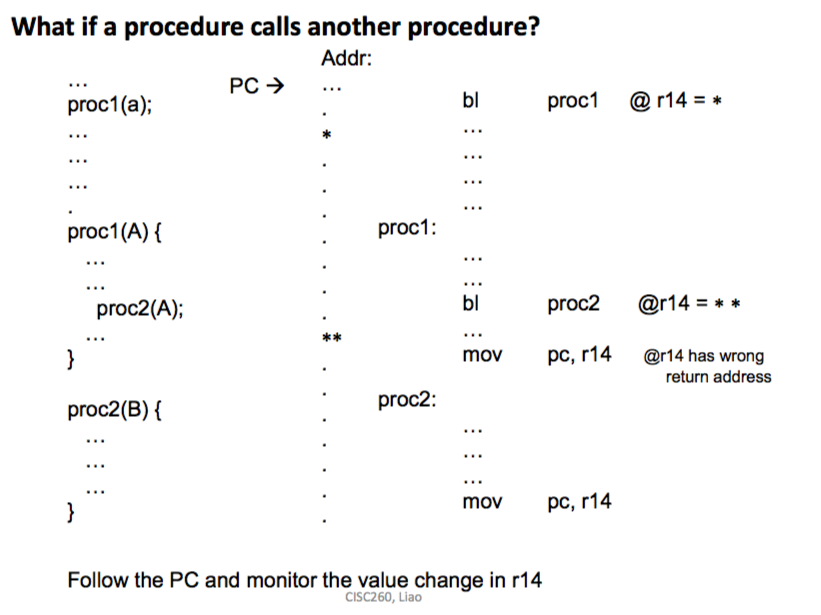
@jump to the address of the callee.

Mov pc, r14 @ reset PC to the return address stored in r14



What if a procedure calls another procedure?

The address stored in R14 is changed, so when mov pc, r14 happens, r14 now stores the wrong return address



Solution: Stack (LIFO)

Acquire storage space, called activation frame

