**Arm Data Transfer Instructions**

A diagram of a computer

Description automatically generated

- load register w/ offset steps: X11 contains the memory ADDRESS of where we originally want to get data from and X10 contains the offset of how much we want to add to that memory address to get a new memory address of where we want to get data from. We would take the data stored in that memory address and load it in X17

-  STR = store register, X9 = source register, X10 = address, #64 = offset by 64 bytes (optional to have offset)

- store register w/ offset steps: X10 contains the memory ADDRESS of where in memory we want to store our data from X9. The offset of 64 bytes means that if, for example, we had a memory address of 0x800000A1, we would add 64 (40 in hexadecimal) to that memory address to get 0x800000E1 and that is the memory address where we store the data from X9

- A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

- Load Byte (LDRB): loads 1 byte into bits 0:7 of the destination register (use W instead of X for register)

- Load Halfword (LDRH): Loads 2 bytes (halfword) into bits 0:15 of the destination register (Use W)

- Load Word (LDW): Loads 4 bytes (word) into bits 0:31 of the destination register

- Load Signed Word (LDRSW): Loads 4 bytes (word) into bits 0:31 of the destination register; Bit 31 is sign extended (repeated in bits 32:63) (Use X)

- Store Word (STRW): stores bits 0:31 of the source register at the memory location (Use X)

- memory is a linear array of 247 bytes and each byte has a 64-bit address and can store an 8-bit pattern

- there are two hexadecimal numbers in a byte which is why when moving memory addresses, two hex numbers are moved at a time

- D: data transfer to and from memory; Load Instruction (LDR): transfers data from memory to register; Store instruction (STR): transfers data from register to memory

-  LDR = load register, D = destination register, [address] = memory address that needs to be transferred, X10 = offset by memory address in X10 (optional to have offset), loads 8 bytes

- the address is kept in a register and it’s a pointer pointing to the first element of the data

- when sign extending, check to see the binary form of the first digit in the register and if the bit is 1, sign extend F or if it’s 0, sign extend 0’s

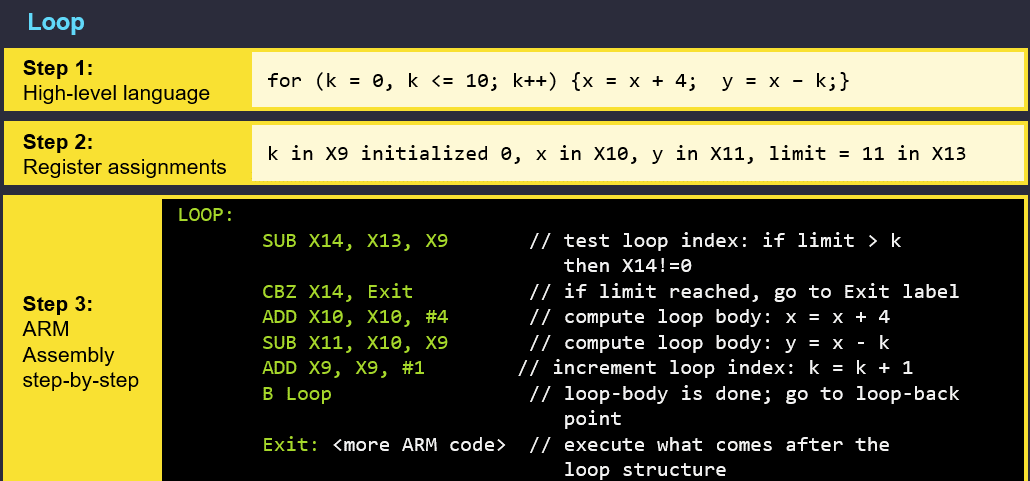
**Decisions and Loops**

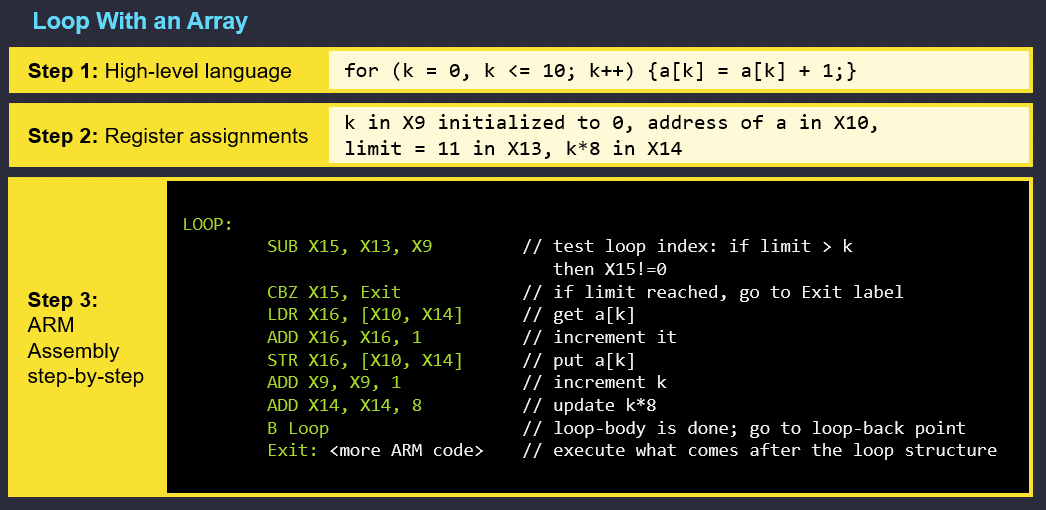
- Conditional Branch (CB): conditional flow control statement

- Branch (B): unconditional flow of control to a specific address

- CBNZ = not zero, CBZ = zero, CMP = compare, ADD = add, SUB = sub, MUL = multiply, MOV = copy to another register

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generatedA screenshot of a computer code

Description automatically generated

A black background with white text

Description automatically generated

A close up of black text

Description automatically generated

First, X13=11, X9=0, if

X13–X9<=10, loop body executes. Basically, when X9=11, the second line is activated and will go to the Exit line

-B.\_\_\_ myLabel (use for conditions and whatnot)

-EQ = equal to, NE = not equal to, LT = less than, GT = greater than, append an S to other arithmetic instructions to set flags

-to determine command to activate else statement, use complement of if condition

- **memory** stores instructions and data, **processing unit** performs calculations on data, **arithmetic logic unit (ALU)** performs calculations, **registers** are where the data that the ALU operates on resides, **control unit** fetches and interprets the current instructions to be executed and gives signals to processing unit so that it can execute the instruction

Consider two different implementations of the same instruction set architecture, with instructions divided into classes A, B, C, and D. Processor P1 has a clock rate of **3 GHz** and **CPIs of 2A, 2B, 3C, and 4D**. Processor P2 has a clock rate of **2 GHz** and **CPIs of 1A, 2B, 3C, and 3D**. Given a program with **109 instructions**, and classes divided as follows: **30% A, 30% B, 20% C, and 20% D**, which processor is faster?

**P1 – CPI = 2\*0.3+2\*0.3+3\*0.2+4\*0.2 = 2.6 cycles/instruction**

**tP1 = (2.6 cycles/instruction \* 109 instructions/program) / 3 x 109 cycles/sec**

**= 0.867 sec/program**

**\* P2 has the same working as above**

P1 is improved to have the following **CPIs: 1A, 1B , 2C , and 4D**, and has a clock rate of **3.5 GHz**. Everything else is **kept the same from the old P1**. What is the speedup? Note: You are comparing the old P1 to the new P1. Speedup has no units.

**CPInew = 0.3\*1+0.3\*1+0.2\*2+0.2\*4=1.8 cycles/instruction**

**tP1 NEW = (1.8 cycles/instruction \* 109 instructions/program)/3.5 x 109 cycles/sec**

**Speedup = tOLD/tNEW =**

**=((2.6 cycles/instruction \* 109 instructions/program) / 3x109 cycles/sec) / ((1.8 cycles/instruction \* 109 instructions) / 3.5 x 109 cycles/sec) )**

**= (2.6/3) / (1.8/3.5) =1.685**

**8 BITS = 1 BYTE**

**Signed Integers**

- the leftmost bit denotes the sign: 1 = negative, 0 = positive

- ex.) -13 --> 11110011

A black line with green lines

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

- Make an 8-bit number into a 16-bit number by sign extending it (adding extra zeros in the front) and perform the above steps

- unsigned 8 bit binary represents numbers 0 to 255 and signed 8 bit binary is -128 to 127, over this amount is overflow

- A screenshot of a computer

Description automatically generated

**Instruction Set Architecture**

- A diagram of instructions

Description automatically generated

- ex.) which type of instruction set architecture requires less power? **RISC**

-ex.) which type of instruction set architecture is generally better for battery powered device **RISC**

- ex.) which type of instruction set architecture creates larger executables? **RISC**

OR

AND

**Arm Basics**

- A black and white text

Description automatically generated -> A yellow and purple rectangles

Description automatically generated : 1st box = mnemonic operation, 2nd box = destination register, 3rd box = first operand, 4th box = second operand

- second operand can be a value and # indicates that it’s a literal value

- 0x indicates hexadecimal, 0b indicates binary

- A table with numbers and symbols

Description automatically generated

- LSL: shift left, LSR: shift right, AND: bitwise AND, OR: bitwise or, NOT: bitwise not

- A number and a number on a blue background

Description automatically generated with medium confidence

- A diagram of a computer code

Description automatically generated A diagram of a computer code

Description automatically generated

- MOV: mnemonic operation that makes a copy of the operand and moves it to the destination register

**Benchmarking**

- benchmark: a set of programs that run on computer systems under test

- A screenshot of a computer error

Description automatically generated

**Computer Performance**

- performance: how fast a computer is

- response time/ execution time/ latency: how long it takes to run a program (seconds)

- throughput/bandwidth: task per unit time (instructions / second)

- decrease the execution time: make it go faster

- increase the throughput: do more things per unit time

- relative performance: , speed up > 1

- when it asks for speedup, make sure larger number goes on top

- how fast a program runs is based on how long a clock cycle is

- clock: varies between 0 and 1; logical values change with the clock

- cycle time: period of the clock; measured in seconds

- clock rate: measured in cycles per second (Hz); inverse of cycle time

- A screenshot of a table

Description automatically generated

- CPU Equation: texe =IC x CPI x Cycle Time

- IC: instruction count, CPI: cycles per instruction,

Cycle Time: seconds per cycle; units are seconds / program; CPIaverage = ∑