**Floating Point Numbers:**

- An unsigned 64-bit integer can represent numbers form 0 to 264 – 1 and a signed 64-bit integer can represent numbers from -263 to 263 – 1

- overflow: result is too large as it’s greater than 2.0 x 1038 but exponent larger that can be represented in 8-bit exponent field

- underflow: result is too small as it’s greater than 0, but less than 2.0 x 10-38 but negative exponent larger than the magnitude that can be represented by the 8-bit exponent field

- what is a reason to use a double precision (64 bit) floating point number rather than a single precision (32 bit) floating point number?

Need to express a smaller number than possible with single precision, Need to express a larger number than possible with single precision, need more precision than is possible with a single precision number

- the exponent uses a bias of 127 so it has a range of 2-126 to 2127

- A screenshot of a graph

Description automatically generated- A screenshot of a math test

Description automatically generated

- if it’s a denormalized number, then subtract 126 instead of 127

- A screenshot of a math test

Description automatically generated- A close-up of a screen

Description automatically generated

**Integer Multiplication:**

- multiplication is expensive in terms of hardwards, space, and time (which is why if possible, we just use LSL)

- A white paper with red text

Description automatically generated- A table with numbers and symbols

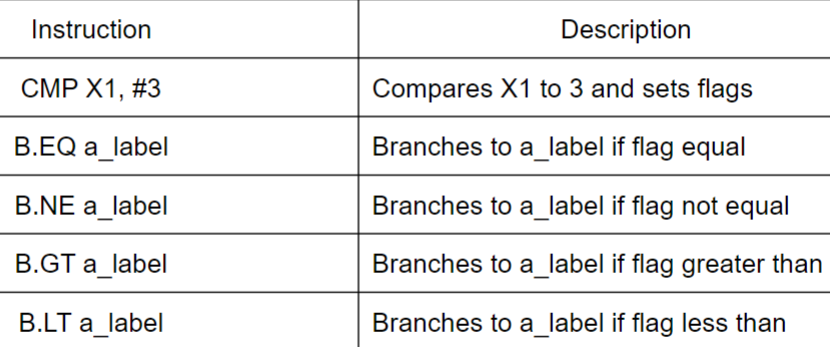
Description automatically generated

- between the multplicand and the multiplier, always look at the rightmost bit of the multiplier to see if we need to add the multiplicand or not. Basically, if the rightmost bit of the multiplier is 1, add the multiplicand to the left four bits of the multiplier but if the rightmost bit of the multiplier is 0, do not do anything. Then, shift the multiplier to the right by one bit and then repeat the process for the number of bits in the multiplicand (if the multiplicand is 4 btis, there should be four iterations with two steps each)

A table with numbers and symbols

Description automatically generated

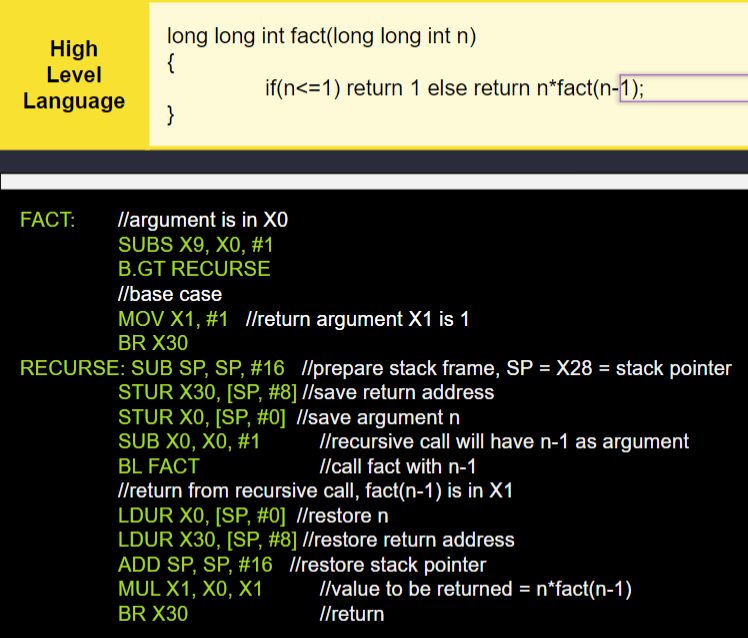
- for division, use SDIV





FP is optional

**Recursion:**

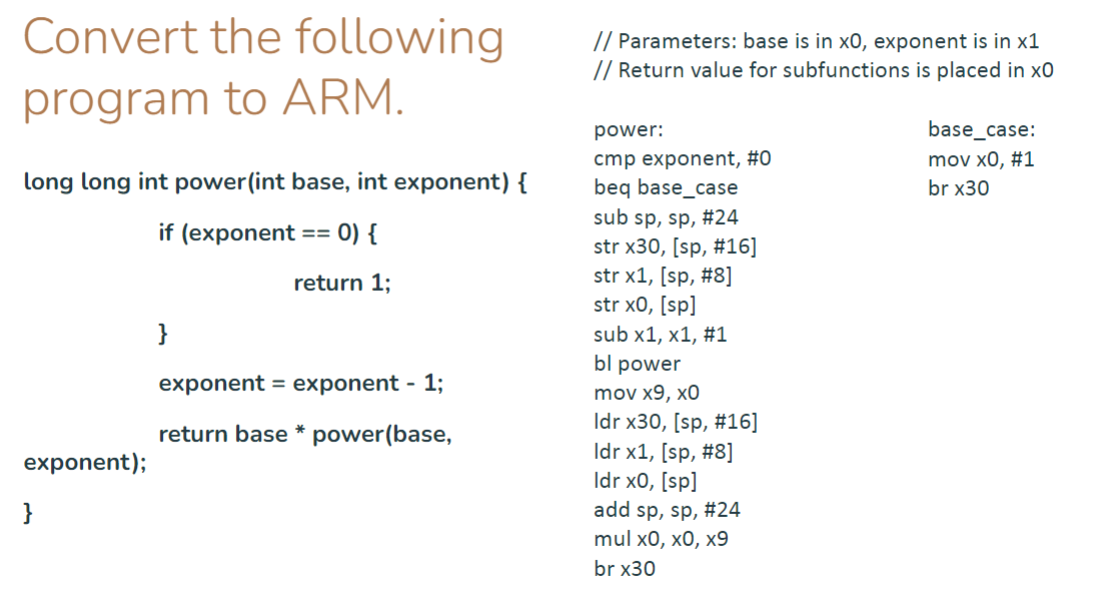


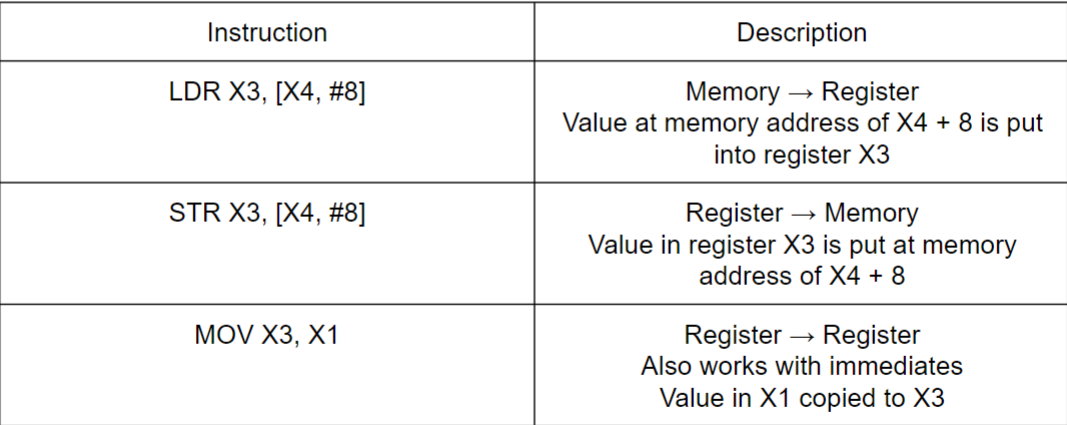
**Branching**:

- B <label> – branches to the instruction after a label

- BR <register> - branch to the instruction at the address found in the register

- BL <label> - put the address of the current instruction +4 into X30 and then branch to the instruction after the label





A screenshot of a computer

Description automatically generated

A yellow box with black text

Description automatically generated

A close-up of a document

Description automatically generated

**Memory Model:**

- memory is a linear array of 247 bytes

- each byte has a 64-bit address and can store an 8-bit pattern

- ARM addresses are 64 bits long and range from 0 to 247

A screenshot of a computer

Description automatically generated

- reserved = used by the operating system

- text = stores code for user programs

- static data = stores data early-bound by compiler (static variables)

- heap = stores dynamic user data structures (memory allocated for reference-based data structures)

- stack = stores dynamic program data structures (local variables, return addresses)

- stack grows down, heap grows up which is why when we update stack pointer, we subtract to allocate it and add to deallocate it

- we store stuff on the stack because when we call another function, the registers that were used for the previous function will get overwritten with the data of the newer function. So to keep track of the data from the previous function, we store it on the stack

**Stored Program Concept:**

- each machine code instruction is 32 bites (4 bytes)

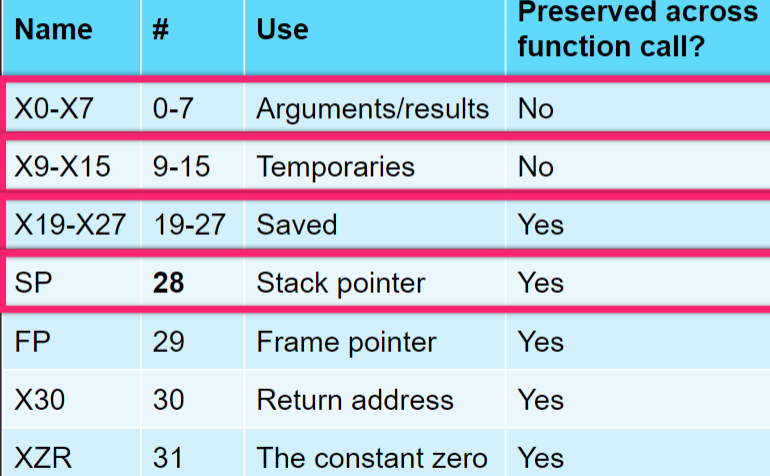
- the instruction’s address will be address will be the address of the first byte

- subsequent instructions will have an address that is different by 4

- program counter (PC) = address of the instruction being executed

- stack frame = a block of data that contains register contents and variable’s values that you want to save when a procedure is called (so the data is not lost)

- When a procedure is called local variables and data necessary for the calling procedures are placed in a stack frame



A screen shot of a computer screen

Description automatically generated

-

PC = Program Counter