Aryan Rao

264954748

aryanrao@iastate.edu

COMS 321 WA-2

2.2

SUB X9,X3,X4 // Compute (i-j) and store the result in X9
LSL X9,X9,#3 // Calculate the byte offset for accessing array A by multiplying (i-j) by 8)
ADD X9,X9,X6 // Find the effective address of A[i-j]
LDUR X10,[X9,#0] // Load the value of A[i-j] into register X10
STUR X10,[X7,#64] // Store the value from X10 into the 9th position of array B

2.4

The assembly instructions basically correspond to the C statement:

B[g] = A[f] + A[f+1]

It is done by performing the intermediary instructions:

f = A[f];

f = A[f+1] + A[f];

B[g] = f

2.6

Our hexadecimal is : 0xabcdef12, it can be split in to bytes as,
ab, cd, ef, 12 : ab is most significant and 12 is least significant.
Big-Endian: highest priority bit given lowest address
Little-Endian: lowest priority bit given lowest address (incrementing address by 1 for every byte)

```
Big-Endian: | ab | cd | ef | 12 |
Address:      0    1    2    3

Little-Endian: |12 | ef | cd | ab |
Address:        0    1    2    3
```

2.8

LSL X9, X3, #3 //X9=[i*8] is the offset value of [i]
ADD X9, X9, X6 // X9=X9+X6 Address of i^{th} element in A
LDUR X10, [X9, #0] // temporary register X10=A[i]
LSL X9, X4, #3 // X9 = [j *8] is the offset value of [j]
ADD X9, X9, X6 // X9=X9+X6 Address of the j^{th} element in A
LDUR X11, [X9, #0] // temporary register X10 = A[j]
ADD X11, X10, X11 // X11 gets the value A[i]+ A[j]
STUR X11, [X7, #64] // Store the register X11 into B[8]

2.11.1

ADD X9, X0, X1
X0 (last 4 bits) = 1000 which is 8
X1 (last 4 bits) = 1101 which is 13 in decimal
Rest all of them are 0.
Sum of these would be:

```
        1 0 0 0
+       1 1 0 1
-------------------------
        1 0 1 0 1
```

Value: 0x5000000000000000

2.11.2

Since there was a carry in the addition overflow does exist.

2.11.3

SUB X9, X0, X1
2s complement of X1 is:
0011000000000000
Subtraction of last 4 bits, as only they matter.

```
        1 0 0 0
    -   0 0 1 1
-----------------------------
        1 0 1 1
```

Result is: 0xB000000000000000

2.11.4

No carry happens therefore no overflow happens.

2.11.5

ADD X9, X0, X1 gives 0x5000000000000000
ADD X9, X9, X0

```
        0 1 0 1
+       1 0 0 0
---------------------------
        1 1 0 1
```

Value: 0xD000000000000000

Value of X9 will be 8+D
= 0xF000000000000000

2.11.6

Yes, there has been an overflow in the first operation but not in second.

2.17

Assume that we would like to expand the LEGv8 register file to 128 registers and expand the instruction set to contain four times as many instructions.

2.17.1

We would first need 7 bits to represent registers, as (2^7 = 128), so we need 7 bits each to represent Rm, Rn, Rd.
Now the instruction set has 4 times as many instructions, so we would have to increase the bits for opcode by: a factor of 4, so it goes up by a 2^2, so it needs: 11 + 2 = 13 bits.
So, our new Instruction would need 8 more bits.

```
13 + 7 + 6 + 7 + 7  = 40 bits (opcode Rm shamt Rn Rd)
```

2.17.2

We would first need 7 bits to represent registers, as (2^7 = 128), so we need 7 bits each to represent Rn, Rd.
Now the instruction set has 4 times as many instructions, so we would have to increase the bits for opcode by: a factor of 4, so it goes up by a 2^2, so it needs: 10 + 2 = 12 bits.
So, our new Instruction would need 6 more bits.

```
12 + 12  + 7 + 7  = 38 bits (opcode immediate Rn Rd)
```

2.17.3

Ways it reduces the size of a program, having more registers could reduce the need for memory access instructions, which can be relatively long and slow. Additionally, having more instructions could allow for more efficient and concise ways of expressing certain operations that require the calling of other operations (eases the code when calling compound operations).

Ways it increases the size of a program, with more registers and instructions, the program may become more complex and harder to read and understand (the number of instructions we need to know to understand the code would go up). Expanding the instruction set could require more memory to store the larger number of instructions, potentially increasing the program's overall size. We are also increasing the size of the instructions itself so it may end up taking up more space even though we have more instructions.

2.20

NOR X10, X11, X11

2.23.1

The LEGv8 `B` instruction changes where the program is running by adjusting the Program Counter (PC). If we start at `0x20000000`, the `B` instruction lets us jump to a new spot within ±128MB of that starting point. So, we can jump to any address between `0x18000000` and `0x28000000`.

2.23.2

The LEGv8 CBZ instruction uses a 19-bit signed immediate value, which is shifted left by 2 bits to form the branch offset. The offset formula is: Offset = Immediate value * 4

The 19-bit signed immediate value ranges from: -2^{18} to $2^{18} - 4$

Multiplying by 4 to get byte offsets:

$-2^{18} * 4$ to $(2^{18} - 4) * 4$

= -1 MB to 1 MB - 16 bytes

Given a starting PC of 0x2000 0000, the possible range after applying the CBZ branch offset is:

0x2000 0000 - 1 MB to 0x2000 0000 + 1 MB - 16 bytes 0x1FFF0000 to 0x201FFFF0.

2.37

MOVZ X0, 4386, LSL #48 // 4386 decimal = 00010001 00100010 in binary
MOVK X0, 13124, LSL #32 // 13124 decimal = 00110011 01000100 in binary
MOVK X0, 21862, LSL #16 // 21862 decimal = 01010101 01100110 in binary
MOVK X0, 30600, LSL #0 // 30600 decimal = 01110111 10001000 in binary

2.41.1

Initial CPI = 10* 300/900 +1* 500/900 + 3* 100/900 = 4.22
New number of arithmetic instructions = 0.75* 500 = 375
Initial ex time = 38/9 *f*900 = 3800f

New CPI = 10*(300/775) + 1* (375/775) + 3*(100/775) = 4.74
New ex time = (3675/775) *f*1.1*775 = 4042.5f

Ex time increases so not a good choice.

2.41.2

New CPI for arithmetic instructions = 0.5

New CPI = 10 *(3/9) +0.5* (5/9) + 3* (1/9)
= 3.944
Overall speed up = 4.22/3.944
= 1.069
On improved by 10 times
New CPI = (30+0.5+3)/9
= 33.5/9
=1.134