Review Problem 2

❖ In assembly, set X0 to –X1.

SUB X0, X31, X1

does not exist

SUBI X0, X1, #19 ~~SUBI X0, #19, X1~~

Basic Operations

(Note: just subset of all instructions)

Mathematic: ADD, SUB, MUL, SDIV

Immediate (second input a constant)

```
ADD X0, X1, X2         // X0 = X1+X2
ADDI X0, X1, #100      // X0 = X1+100
```

← 64 x 2 input gates

Logical: AND, ORR, EOR

Immediate

```
AND X0, X1, X2         // X0 = X1&X2
ANDI X0, X1, #7        // X0 = X1&b0111
```

← fill the holes with zeroes

Shift: left & right logical (LSL, LSR)

```
LSL X0, X1, #4         // X0 = X1<<4
```

Example: Take bits 6-4 of X0 and make them bits 2-0 of X1, zeros otherwise:

```
LSR X1, X0, #4    // Put bits into correct spots
ANDI X1, X1, #7   // Sets all but bottom 3 to 0
```

Memory Organization

Viewed as a large, single-dimension array, with an address.
A memory address is an index into the array
"Byte addressing" means that the index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

Memory Organization (cont.)

Bytes are nice, but most data items use larger units.

Double-word = 64 bits = 8 bytes

Word = 32 bits = 4 bytes

0	64 bits of data
8	64 bits of data
16	64 bits of data
24	64 bits of data

Registers hold 64 bits of data

2^{64} bytes with byte addresses from 0 to 2^{64}-1

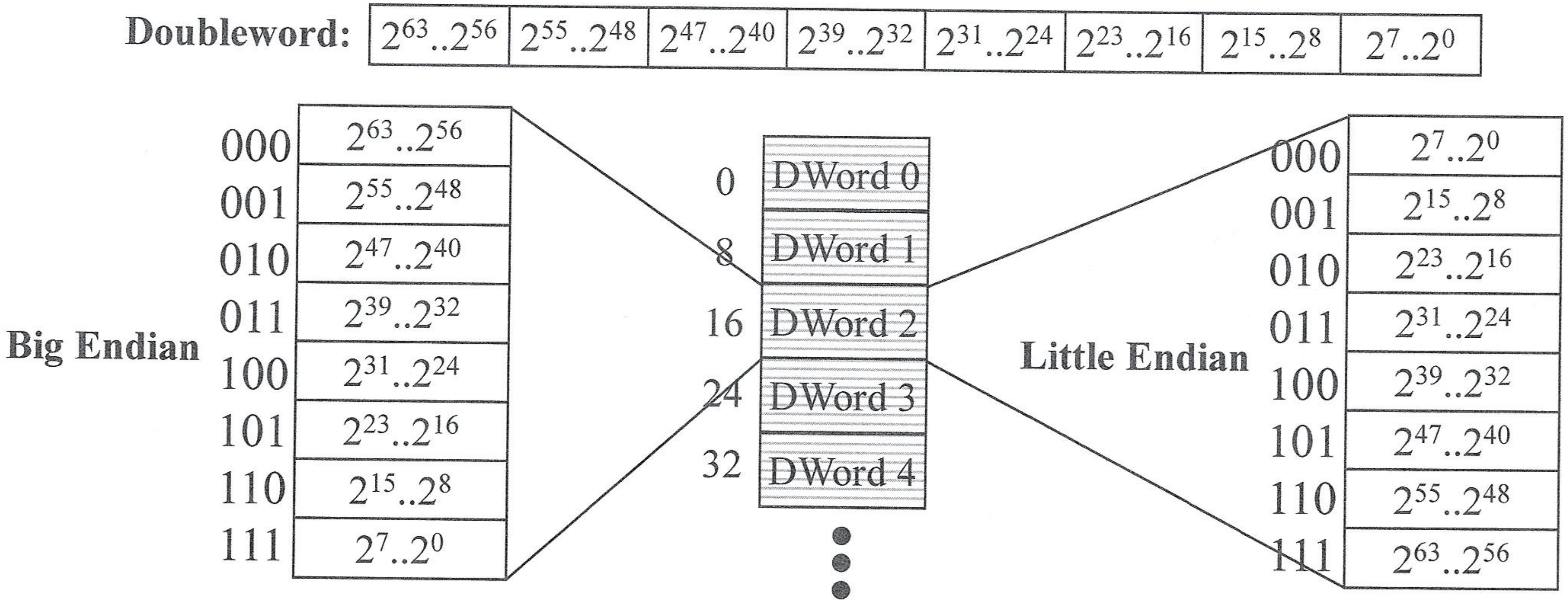
2^{61} double-words with byte addresses 0, 8, 16, ... 2^{64}-8

Double-words and words are aligned

i.e., what are the least 3 significant bits of a double-word address?

000's

Addressing Objects: Endian and Alignment



Big Endian: address of most significant byte = doubleword address
Motorola 68k, MIPS, IBM 360/370, Xilinx Microblaze, Sparc

Little Endian: address of least significant byte = doubleword address
Intel x86, DEC Vax, Altera Nios II, Z80

ARM: can do either – this class assumes Little-Endian.

Data Storage

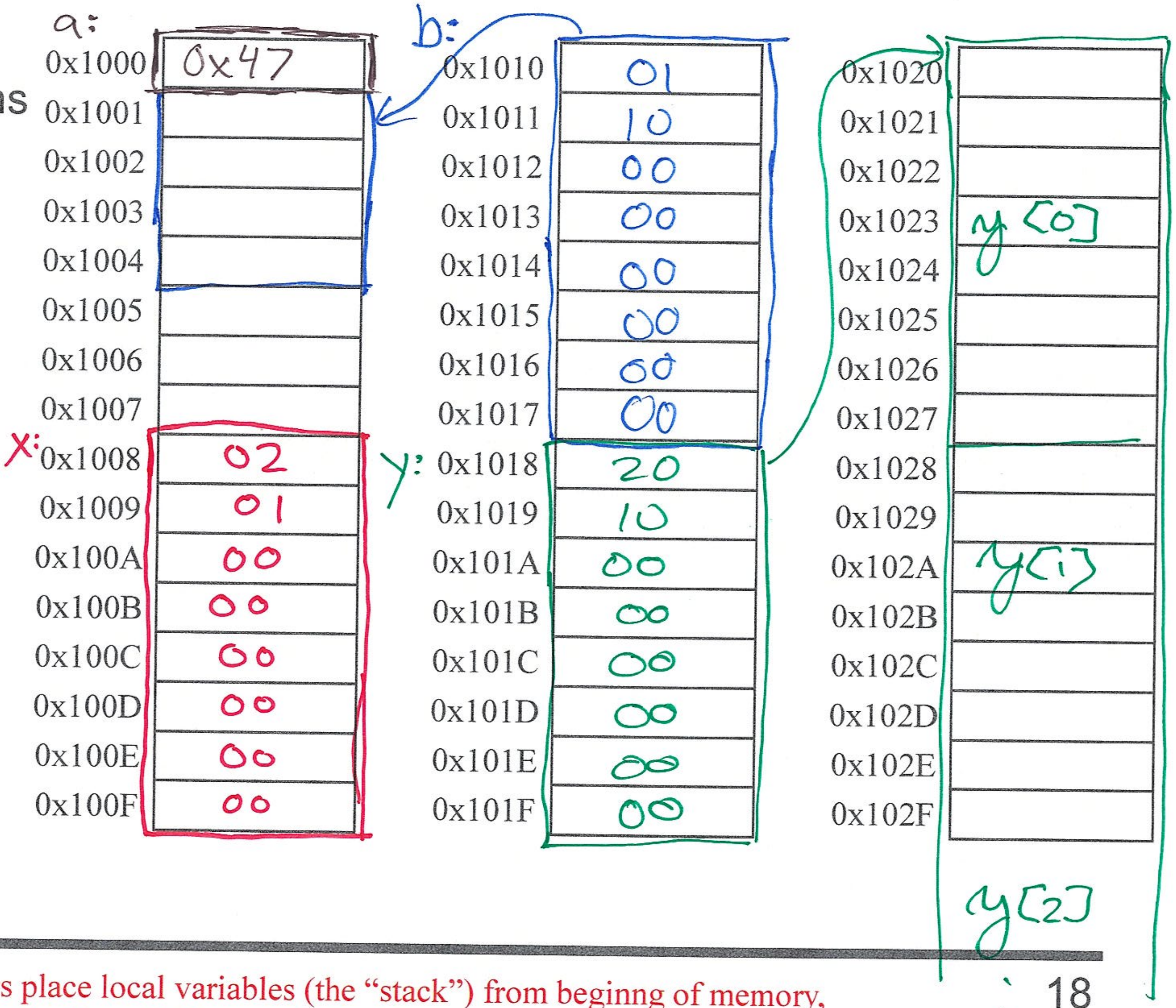
Characters: 8 bits (byte)

Integers: 64 bits (D-word)

Array: Sequence of locations

Pointer: Address (64 bits)

```
 // G = ASCII 71: 0x47
char a = 'G';
int x = 258; // 0x102
char *b;
int *y;
b = new char[4];
y = new int[10];
```



(Note: real compilers place local variables (the "stack") from beginng of memory, new'ed structures (the "heap") from end. We ignore that here for simplicity)

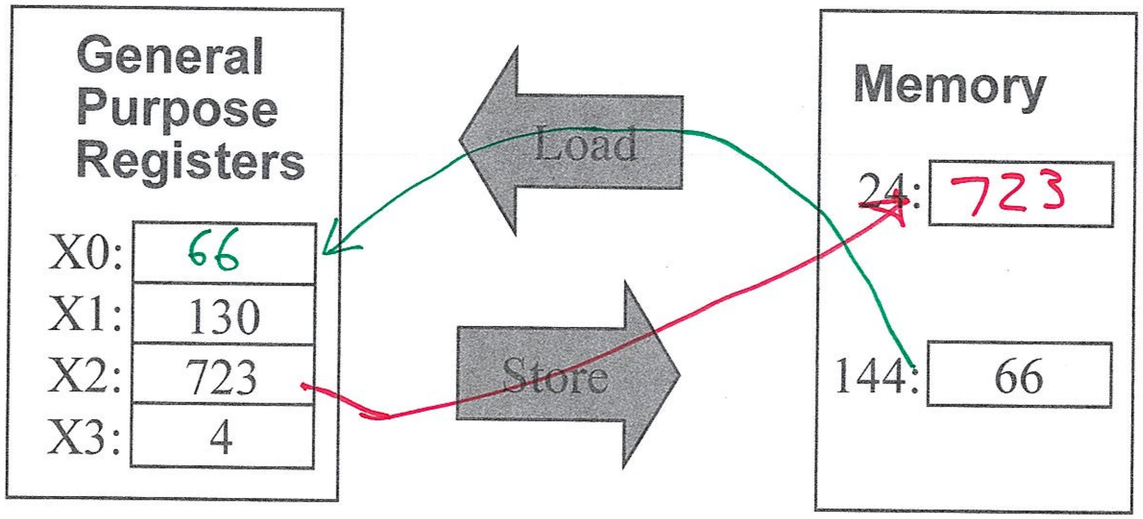
Loads & Stores

Loads & Stores move data between memory and registers

All operations on registers, but too small to hold all data

```
LDUR X0, [X1, #14]          // X0 = Memory[X1+14]
         130+14 = 144

STUR X2, [X3, #20]          // Memory[X3+20] = X2
         4+20 =24
```



Note: LDURB & STURB load & store bytes