L11 - Load and Store

Mar 2, 2021	CSE 2312	Dr. Losh	UT Arlington	Luke Sweeney

Load and Store

Data is stored in memory. It can be loaded and stored. Typically we'll have something like 4 GiB of memory. Loaded data can be stored in registers (32 bit)

Loading

32 Bit

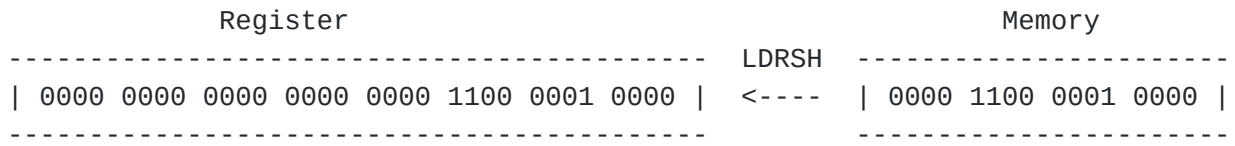
If a 32 bit number from memory is loaded into a register, the bits are just copied right over. This is done with `LDR` (load register). Because all bits are copied over as is, there is no need for padding.

16 Bit

For `uint16_t`, we load with `LDRH` (load half word). This zero-pads the number and copies it over. For `int16_t` it's a bit different. If the number is negative, then there should be `1` in the leftmost 16 bits. Copying over a 16 bit number with `LDRSH` will sign-extend (zero pad or one-pad, depending on the sign bit) the number so it's 32 bits.

```
For uint16_t (unsigned)
             Register                                    Memory
-----------------------------------------  LDRH   ----------------------
| 0000 0000 0000 0000 0000 1100 0001 0000 |  <----  | 0000 1100 0001 0000 |
-----------------------------------------         ----------------------

For int16_t (signed)
             Register                                    Memory
-----------------------------------------  LDRSH  ----------------------
| 1111 1111 1111 1111 1000 1100 0001 0000 |  <----  | 1000 1100 0001 0000 |
-----------------------------------------         ----------------------
```



8 Bit

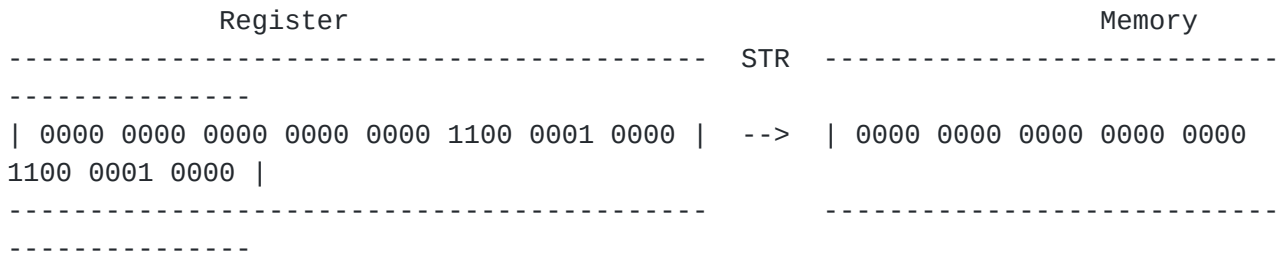
`uint8_t` and `int8_t` work the same way 16 bit numbers do, but it will pad with 24 bits instead of 16 (to get to 32). You load with `LDRB` (load register byte) and `LDRSB` (load register sign extend byte)

Storing

Storing works in much the same was as loading.

32 bit

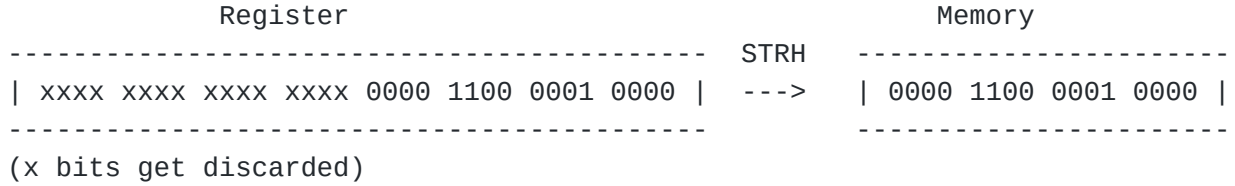
`STR` will store a 32 bit number in memory



There's no padding or loss of data.

16 bit

`STRH` will store a half word, discarding the upper bits.



8 Bit

`STRB` will store a byte

```
             Register                                    Memory
-----------------------------------------  STRB   -------------
| xxxx xxxx xxxx xxxx xxxx xxxx 0001 0000 |  --->  | 0001 0000 |
-----------------------------------------         -------------
(x bits get discarded)
```

How the Computer Stores things

Here's an example of `STR`, which equivalent `C` code

```
(assembly psuedocode)
R0 = 0x12345678
// This is the memory address of the first byte of a 32 bit number
R1 = 0x2000 0000
// We want to STR that number into R0

// This means "put the contents of R0 into the memory location of R1"
STR R0, [R1]
```

Here's that same thing in `C`

```
uint32_t x = 0x12345678;
uint32_t *p = 0x20000000;
*p = x;
```

Endianess

Big Endian

This will store the number like so

```
0x12 -> [0x2000 0000] (MSB)
0x34 -> [0x2000 0001]
0x56 -> [0x2000 0002]
0x78 -> [0x2000 0003] (LSB)
```

This is called **big endian** convention, where the big part, or most significant bit (MSB) goes at the lowest address. In this case, `0x12` (MSB) went to the lower address `0x2000 0000`.

Little Endian

This is the opposite of big endian, where the LSB is stored at the lowest address.

STRH

`STRH` will store half a word. The most significant half gets discarded ~~(that's what Losh said, but the example he wrote showed the least significant half being discarded)~~ (he corrected this, its the most significant bytes that get discarded).

```
R0 = 0x12345678
R1 = 0x2000 0000
STR R0, [R1]
// Store R0 -> [R1]

For Big Endian
    0x56 -> [0x2000 0000]
    0x78 -> [0x2000 0001]

For Little Endian
    0x78 -> [0x2000 0000]
    0x56 -> [0x2000 0001]
```

STRB

`STRB` stores a byte

```
R0 = 0x12345678
R1 = 0x2000 0000
STR R0, [R1]
```

```
0x78 -> [0x2000 0000]
```