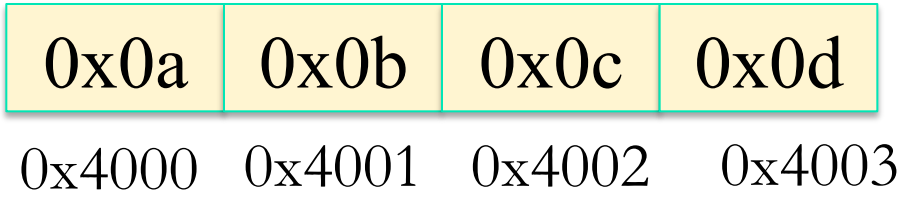
Lecture 4 (part 2): Data Transfer Instructions

CSE 30: Computer Organization and Systems Programming

Diba Mirza
Dept. of Computer Science and Engineering
University of California, San Diego

Assembly Operands: Memory

- Memory: Think of as single one-dimensional array where each cell
  - Stores a byte size value
  - Is referred to by a 32 bit address e.g. value at 0x4000 is 0x0a



- Data is stored in memory as: variables, arrays, structures
- But ARM arithmetic instructions only operate on registers, never directly on memory.
- Data transfer instructions transfer data between registers and memory:
  - Memory to register or LOAD from memory to register
  - Register to memory or STORE from register to memory

Load/Store Instructions

- The ARM is a Load/Store Architecture:
  - Does not support memory to memory data processing operations.
  - Must move data values into registers before using them.
- This might sound inefficient, but in practice isn't:
  - Load data values from memory into registers.
  - Process data in registers using a number of data processing instructions which are not slowed down by memory access.
  - Store results from registers out to memory.

Load/Store Instructions

- The ARM has three sets of instructions which interact with main memory. These are:
  - Single register data transfer (LDR/STR)
  - Block data transfer (LDM/STM)
  - Single Data Swap (SWP)
- The basic load and store instructions are:
  - Load and Store Word or Byte or Halfword
    - LDR / STR / LDRB / STRB / LDRH / STRH

Single register data transfer

LDR	**STR**	Word
LDRB	**STRB**	Byte
LDRH	**STRH**	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

- Memory system must support all access sizes
- Syntax:
  - **LDR**{<cond>}{<size>} Rd, <address>
  - **STR**{<cond>}{<size>} Rd, <address>

  e.g. **LDREQB**

Data Transfer: Memory to Register

- To transfer a word of data, we need to specify two things:
  - Register: r0-r15
  - Memory address: more difficult
    - How do we specify the memory address of data to operate on?
    - We will look at different ways of how this is done in ARM

Remember: Load value/data FROM memory

Addressing Modes

- There are many ways in ARM to specify the address; these are called addressing modes.
- Two basic classification
  1. Base register Addressing
     - Register holds the 32 bit memory address
     - Also called the base address
  2. Base Displacement Addressing mode
     - An *effective address is calculated :*

       Effective address = < Base address +offset>
     - Base address in a register as before
     - Offset can be specified in different ways

Base Register Addressing Modes

- Specify a register which contains the memory address
  - In case of the load instruction (LDR) this is the memory address of the data that we want to retrieve from memory
  - In case of the store instruction (STR), this is the memory address where we want to write the value which is currently in a register
- Example: `[r0]`
  - specifies the memory address pointed to by the value in `r0`

Data Transfer: Memory to Register

- Load Instruction Syntax:

  1 2, [3]

  - where

    1) operation name

    2) register that will receive value

    3) register containing pointer to memory

- ARM Instruction Name:

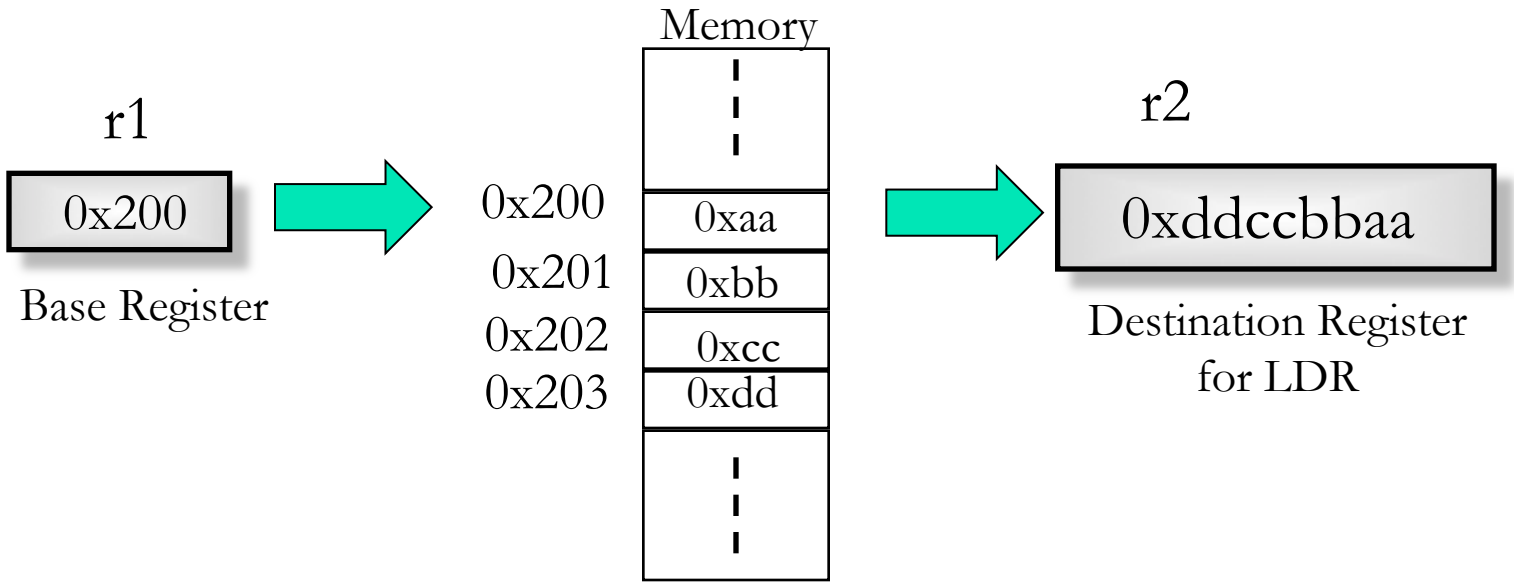
  - `LDR` (meaning Load Register, so 32 bits or one word are loaded at a time)

Data Transfer: Memory to Register

- `LDR r2,[r1]`

  This instruction will take the address in `r1`, and then load a 4 byte value from the memory pointed to by it into register `r2`

- Note: `r1` is called the base register

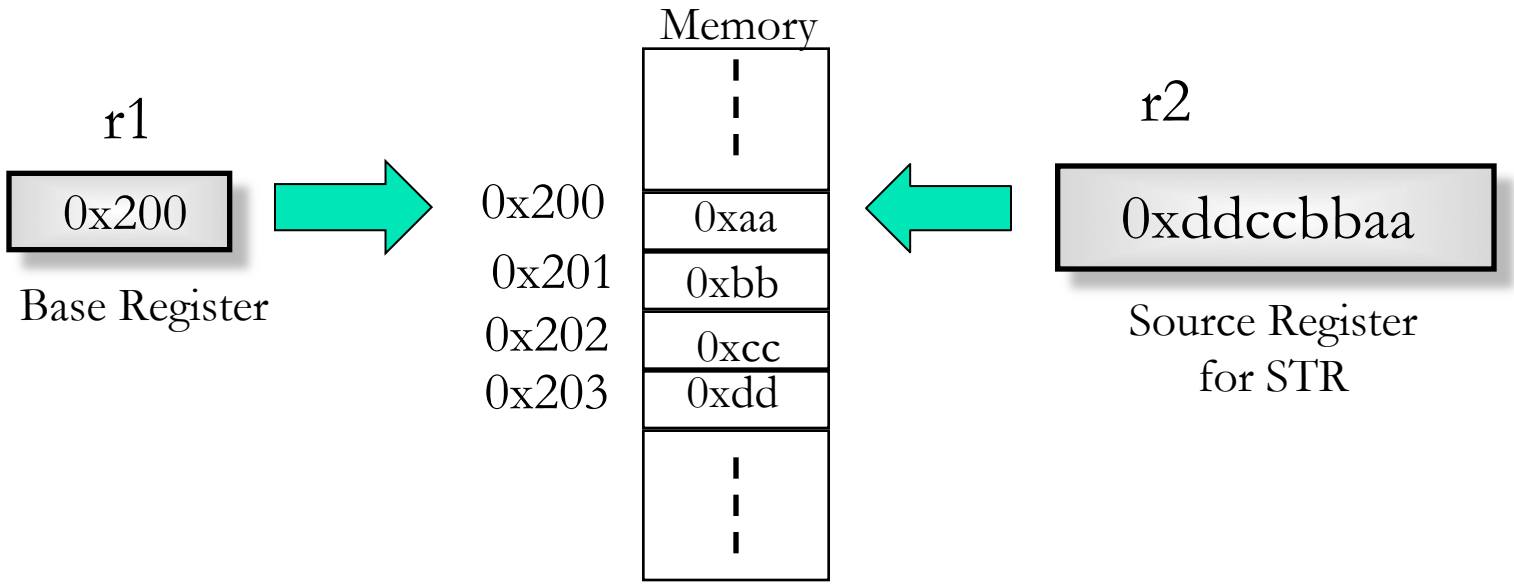


Data Transfer: Register to Memory

- `STR r2,[r1]`

  This instruction will take the address in `r1`, and then store a 4 byte value from the register r2 to the memory pointed to by r1.

- Note: `r1` is called the base register



Base Displacement Addressing Mode

- To specify a memory address to copy from, specify two things:
  - A register which contains a pointer to memory
  - A numerical offset (in bytes)
- The effective memory address is the sum of these two values.
- Example: `[r0,#8]`
  - specifies the memory address pointed to by the value in `r0`, plus 8 bytes

Base Displacement Addressing Mode

1. Pre-indexed addressing syntax:

I. Base register is not updated

```
LDR/STR  <dest_reg>[<base_reg>,offset]
```

Examples:

```
LDR/STR r1 [r2, #4];  offset: immediate 4
   ;The effective memory address is calculated as r2+4
LDR/STR r1 [r2, r3];  offset: value in register r3
   ;The effective memory address is calculated as r2+r3
LDR/STR r1 [r2, r3, LSL #3];  offset:  register value *2^3
   ;The effective memory address is calculated as r2+r3*2^3
```

Base Displacement Addressing Mode

1. Pre-indexed addressing:

I. Base register is not updated:

```
LDR/STR  <dest_reg>[<base_reg>,offset]
```

II. Base register is first updated, the updated address is used

```
LDR/STR  <dest_reg>[<base_reg>,offset]!
```

Examples:

```
LDR/STR r1 [r2, #4]!;  offset: immediate 4
   ;r2=r2+4
LDR/STR r1 [r2, r3]!;  offset: value in register r3
   ;r2=r2+r3
LDR r1 [r2, r3, LSL #3]!;  offset:  register value *2^3
   ;r2=r2+r3*2^3
```

Base Displacement, Pre-Indexed

- Example: `LDR r0,[r1,#12]`

  This instruction will take the pointer in `r1`, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register `r0`

- Example: `STR r0,[r1,#-8]`

  This instruction will take the pointer in `r0`, subtract 8 bytes from it, and then store the value from register `r0` into the memory address pointed to by the calculated sum

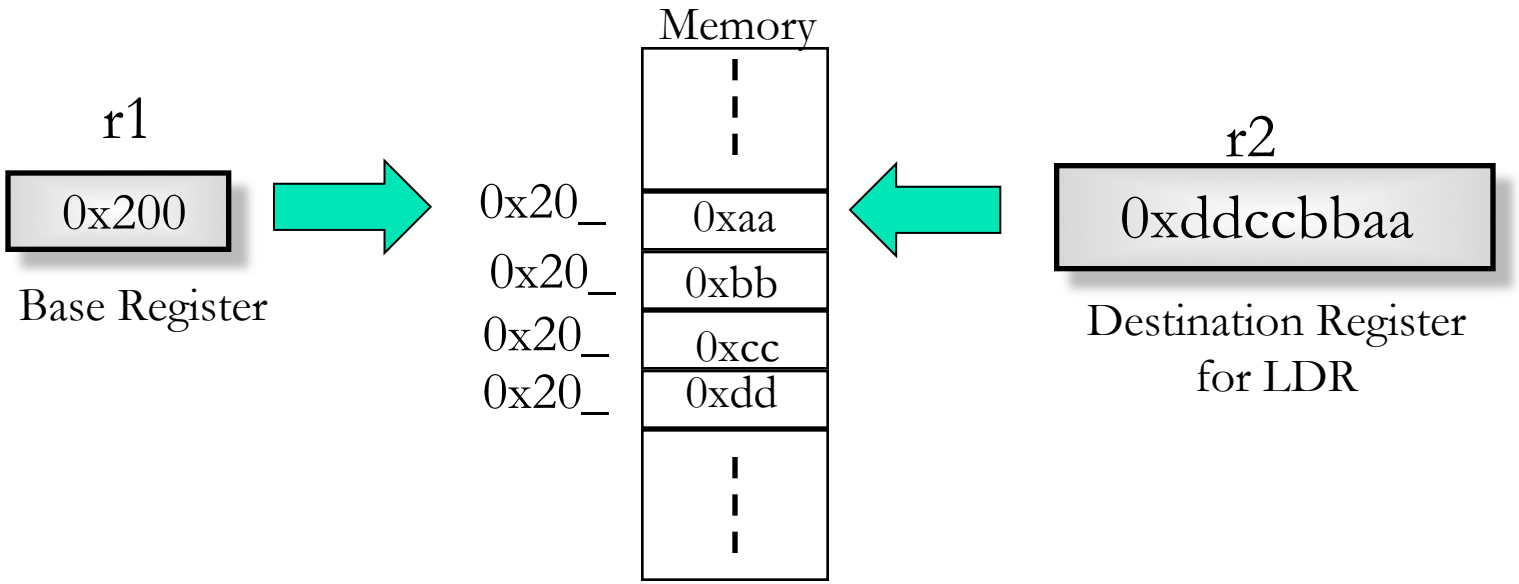
- Notes:
  - `r1` is called the base register
  - `#constant` is called the offset
  - offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure

Pre indexed addressing

What is the value in r1 after the following instruction is executed?

```
STR r2,[r1, #-4]!
```

A. 0x200
B. 0x1fc
C. 0x196
D. None of the above



Base Displacement Addressing Mode

1. Post-indexed addressing: Base register is updated after load/store

```
LDR/STR  <dest_reg>[<base_reg>] ,offset
```

Examples:

```
LDR/STR r1 [r2], #4;  offset: immediate 4
```

; Load/Store to/from memory address in r2, update r2=r2+4

```
LDR/STR r1 [r2], r3;  offset: value in register r3
```

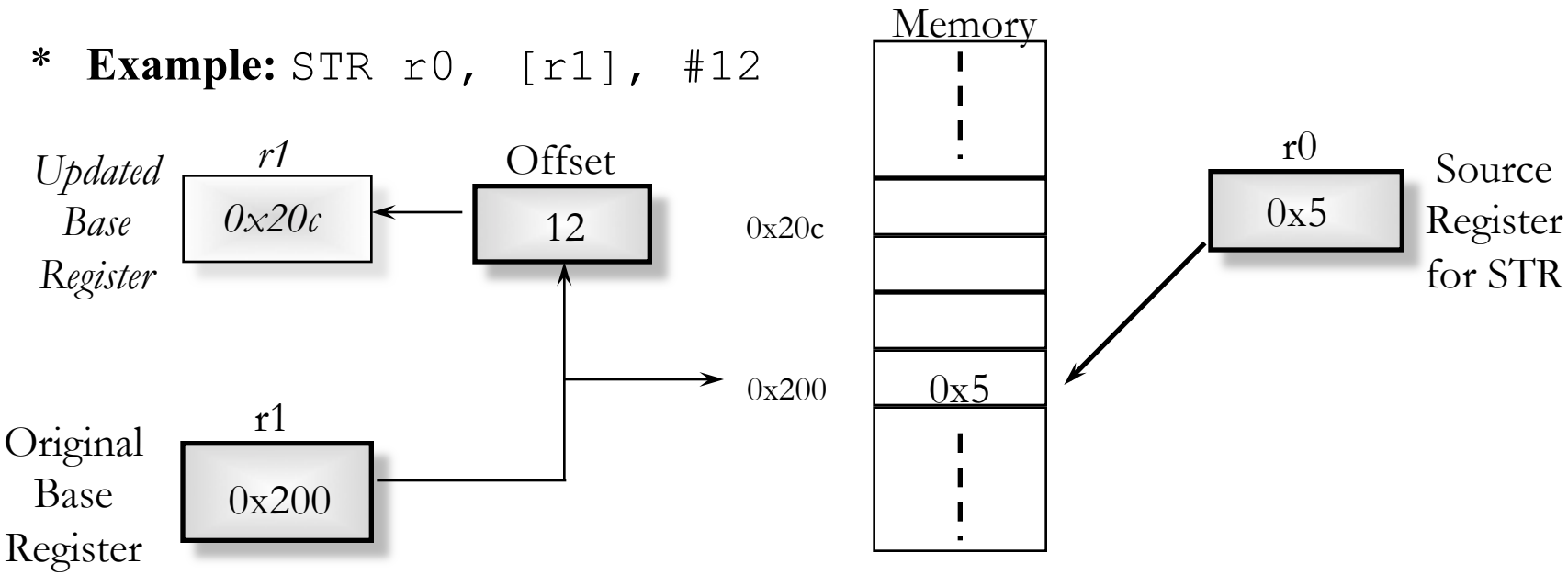
; Load/Store to/from memory address in r2, update r2=r2+r3

```
LDR r1 [r2] r3, LSL #3;  offset:  register value left shifted
```

; Load/Store to/from memory address in r2, update $r2=r2+r3*2^3$

Post-indexed Addressing Mode

* **Example:** `STR r0, [r1], #12`

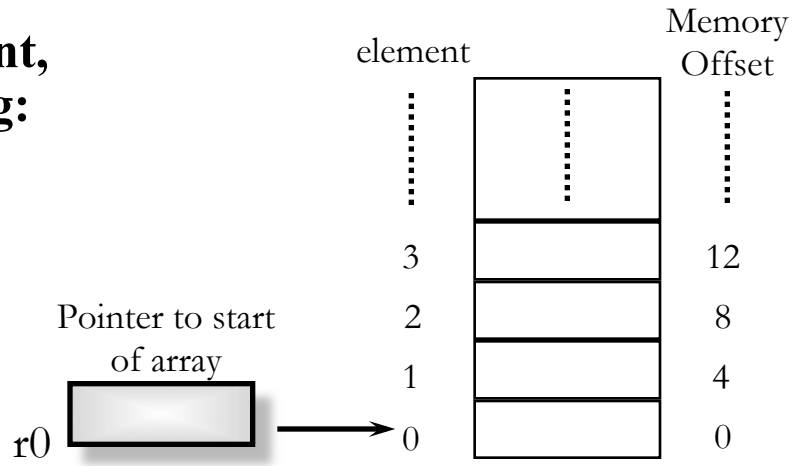


* **If r2 contains 3, auto-increment base register to 0x20c by multiplying this by 4:**
  - `STR r0, [r1], r2, LSL #2`

* **To auto-increment the base register to location 0x1f4 instead use:**
  - `STR r0, [r1], #-12`

Using Addressing Modes Efficiently

* **Imagine an array, the first element of which is pointed to by the contents of `r0`.**
* **If we want to access a particular element, then we can use pre-indexed addressing:**
  - `r1` is element we want.
  - `LDR r2, [r0, r1, LSL #2]`



* **If we want to step through every element of the array, for instance to produce sum of elements in the array, then we can use post-indexed addressing within a loop:**
  - `r1` is address of current element (initially equal to r0).
  - `LDR r2, [r1], #4`

Use a further register to store the address of final element, so that the loop can be correctly terminated.

Pointers vs. Values

- Key Concept: A register can hold any 32-bit value. That value can be a (signed) `int`, an `unsigned int`, a pointer (memory address), and so on
- If you write `ADD r2,r1,r0`
  then `r0` and `r1` better contain values
- If you write `LDR r2,[r0]`
  then `[r0]` better contain a pointer
- Don't mix these up!

Compilation with Memory

- What offset in `LDR` to select `A[8]` in C?
- 4x8=32 to select `A[8]`: byte vs word
- Compile by hand using registers:

```
g = h + A[8];
```

  - `g`: `r1`, `h`: `r2`, `r3`:base address of **`A`**
- 1st transfer from memory to register:

```
LDR r0,[r3, #32]   ; r0 gets A[8]
```

  - Add `32` to `r3` to select `A[8]`, put into `r0`
- Next add it to h and place in g

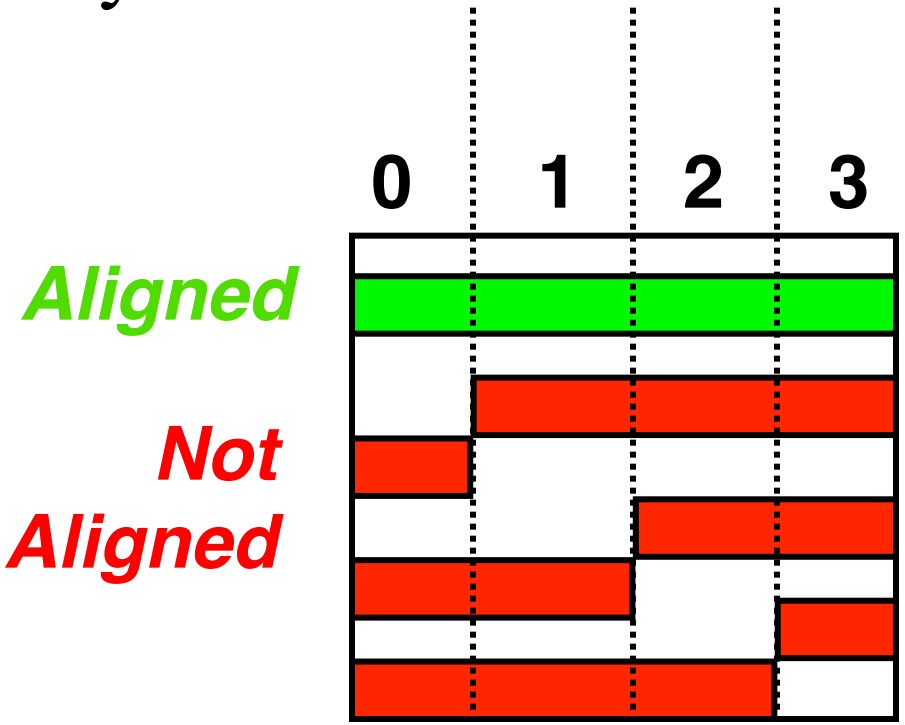
```
ADD r1,r2,r0       ; r1 = h+A[8]
```

Notes about Memory

- Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
  - Many assembly language programmers have toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.
  - So remember that for both `LDR` and `STR`, the sum of the base address and the offset must be a multiple of 4 (to be word aligned)

More Notes about Memory: Alignment

- ARM typically requires that all words start at byte addresses that are multiples of 4 bytes



Last hex digit of address is:

0, 4, 8, or C_{hex}

1, 5, 9, or D_{hex}

2, 6, A, or E_{hex}

3, 7, B, or F_{hex}

- Called Alignment: objects must fall on address that is multiple of their size.

Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variables in registers
- Why not keep all variables in memory?
  - Smaller is faster:
    registers are faster than memory
  - Registers more versatile:
    - ARM arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - ARM data transfer only read or write 1 operand per instruction, and no operation

Conclusion

- Memory is byte-addressable, but `LDR` and `STR` access one word at a time.
- A pointer (used by `LDR` and `STR`) is just a memory address, so we can add to it or subtract from it (using offset).

Conclusion

- Instructions so far:
  - Previously:

    `ADD, SUB, MUL, MULA, [U|S]MULL, [U|S]MLAL, RSB`

    `AND, ORR, EOR, BIC`

    `MOV, MVN`

    `LSL, LSR, ASR, ROR`

  New:

  `LDR, LDR, STR, LDRB, STRB, LDRH, STRH`