Chapter 5 – Memory Load and Store Instructions

1 Introduction

The programs we have written so far use CPU registers to hold all program variables. This approach is simple but has an obvious limitation: large data structures like arrays or lists are too big to fit entirely in the processor. Because of that, we'll have to leave our data in **main memory** most of the time, and only access it in small pieces when required. The SCAT processor provides two instructions for this purpose: one for **storing** values contained in registers **into** memory, and another for **loading** values **from** memory to registers.

Our goal for today is to understand these memory instructions and their **addressing modes**, i.e. how to specify the memory cell we want to write to or read from. We will write two programs: one to find the maximum value in an array of integers, and another to sort the entire array with "bubble sort".

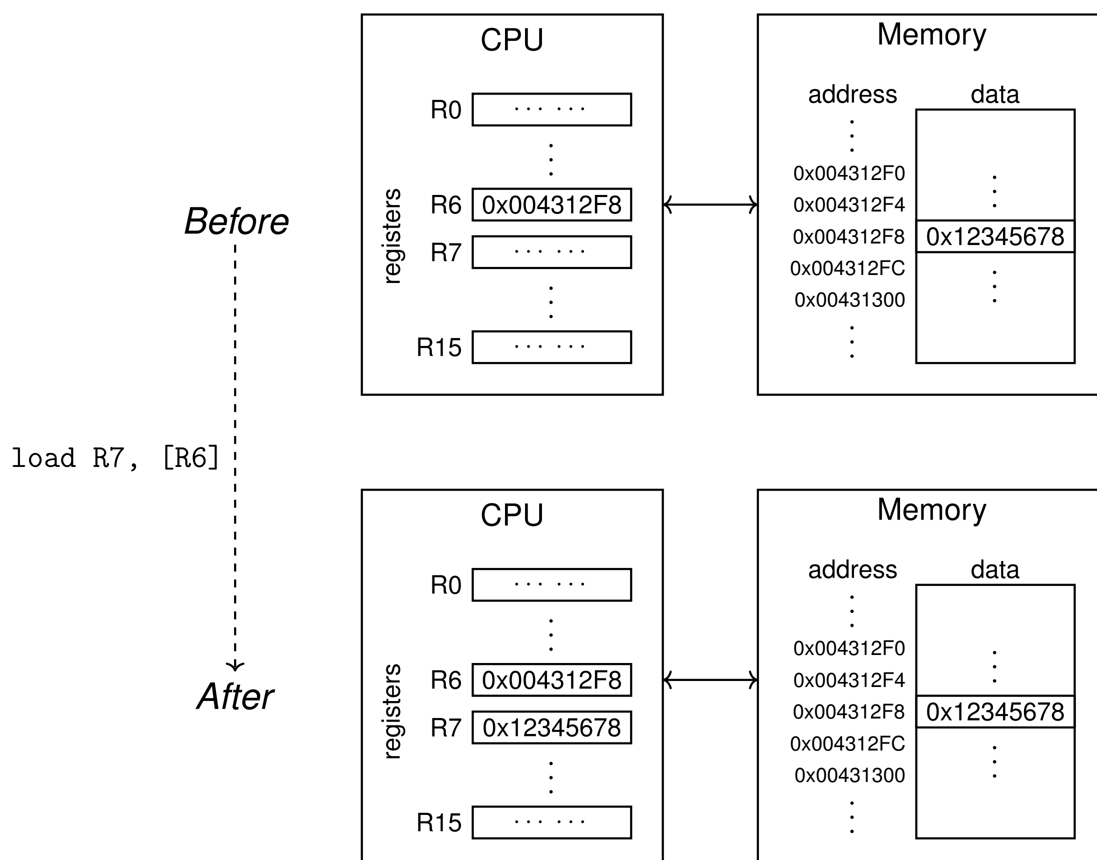
2 Memory transfer instructions

Idea The SCAT processors offers two instructions for accessing memory, `load` and `store`.

- The `store` instruction copies the contents of one register (the **source register**) into a memory location. The address of that location is given by the **destination register**.
- The `load` instruction works the other way around: it reads a value from memory and copies it to the destination register. The address of the memory location is given by the source register.

The diagram below illustrates the effect of loading `[R6]` into register R7.
Notes:

- You should read `[R6]` as "the contents of the memory location at address R6".
- Each transfer is register-sized: the CPU always reads or writes 32 bits (4 bytes) at once.



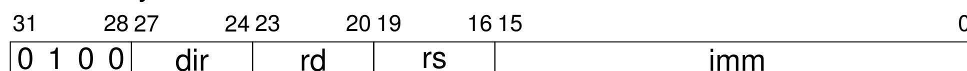
ASM syntax and binary encoding (This section is for reference only. Skip it on your first read). Both memory transfer instructions use **register-indirect** addressing: the address of the desired memory location is given by a CPU register, e.g. `[R6]` in the example on the preceding page. In our ASM syntax, a memory operand is always surrounded with square brackets.
In addition, SCAT implements a "base+displacement" addressing scheme: load/store instructions may include an integer constant which will be added to the base address. This **offset** can be positive or negative, e.g. `[R1+4]` or `[R3-8]`.
The table below sums up the various possibilities. Note that the destination always comes first, and the source always comes second.

asm	Name	dir code	Description
`load rd, [rs + offset]`	Load Word	0000	`rd = MEM[rs + sxt(imm)]`
`store [rd + offset], rs`	Store Word	0001	`MEM[rd + sxt(imm)] = rs`

In machine code, memory transfers are encoded as follows:

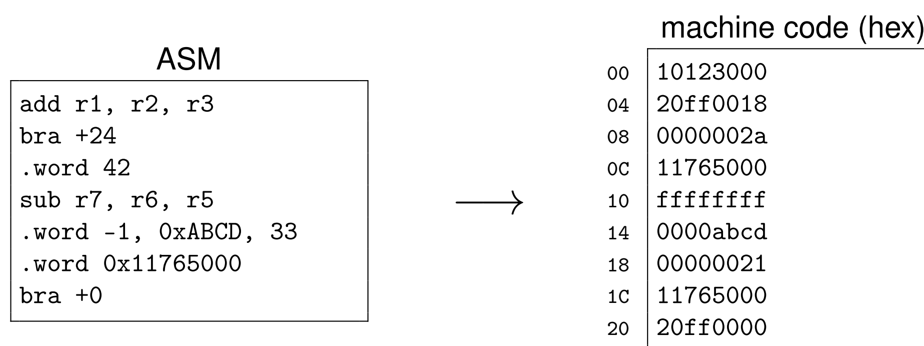
- bits 31–28 are always 0b0100 to indicate a type 4 instruction.
- bits 27–24 indicate the direction of the transfer: memory→CPU (load) or CPU→memory (store)
- bits 23–20 and 19–16 select the destination and source registers, respectively.
- the remaining 16 bits contain the offset encoded in two's complement.

In other words, the binary format of the load/store instructions is:



3 Allocating data with the `.word` ASM directive

Idea In the von Neuman model, instructions and data live in the same memory space. Just as our assembler translates instructions from readable text to binary, it can also encode arbitrary data values to include in the program.
As shown below, the asm syntax is "`.word VAL`" (or "`.word VAL1, VAL2`" for multiple values).



Symbolic labels for data In the previous chapter, we saw that named labels can be associated with program locations. Labels can actually be used to reference any location in memory, whether this location contains a program instruction or a piece of data. This is very useful to declare named constants or global variables, for instance "`var: .word 0xff241300`".

Such named locations can then be referenced in the rest of the program, for example:

- "`leti rd, var`" sets register `rd` to the address of `var`, so that it can be accessed through `[rd]`.
- "`load rd, [var]`" reads (the value of) the variable into register `rd`.
- "`store [var], rs`" writes the contents of `rs` into memory, at the correct address.

To translate those forms to machine code, the assembler uses PC-relative addressing. (If that sentence makes no sense to you, go read the "binary encoding" section above, then ask us for help).

Exercise 1 Complete the program below so that it reads two numbers A and B from memory, then finds the maximum value of the two, then writes this result to a third memory location C.

```
    bra main ; jump over A,B,C variables

A: .word ...
B: .word ...
C: .word 0

main:
    ...
```

Warning: The contents of memory changes during execution, so our listing file will become out of date. To observe memory from within the simulator, you can use the `memdump` command (type "`help memdump`" to get started).

4 Maximum of an array

Exercise 2 Write a program that loops over an array of numbers and finds the maximum value. The length of the array is a known constant. Here is an example with length 10:

```
T: .word 13, 18, 5, 3, 10, 8, 20, 1, 14, 6
```

Remark: remember that each integer is 32 bits long, so address(T[i]) = address(T) + 4$\times$i.

5 Bubble Sort

In this section you will implement a simple sorting algorithm. As stated by Wikipedia[1], **bubble sort** *"repeatedly steps through the input list element by element, comparing the current element with the one after it, swapping their values if needed. These passes through the list are repeated until no swaps had to be performed during a pass, meaning that the list has become fully sorted. The algorithm, which is a comparison sort, is named for the way the larger elements "bubble" up to the top of the list. "*

Example We start with the array T initialized as follows: (6 5 3 1 8 7 2 4). During each pass, the algorithm iterates on all pairs T[i]/T[i+1] and swaps values if needed:
i=0: 6 $>$ 5 so we swap them: (**6 5** 3 1 8 7 2 4) $\rightarrow$ (**5 6** 3 1 8 7 2 4)
i=1: 6 $>$ 3 so we swap them: (5 **6 3** 1 8 7 2 4) $\rightarrow$ (5 **3 6** 1 8 7 2 4)
i=2: 6 $>$ 1 so we swap them: (5 3 **6 1** 8 7 2 4) $\rightarrow$ (5 3 **1 6** 8 7 2 4)
i=3: 6 $\leqslant$ 8 so we leave the 6 in place and we move on
i=4: 8 $>$ 7 so we swap them: (5 3 1 6 **8 7** 2 4) $\rightarrow$ (5 3 1 6 **7 8** 2 4)
i=5: 8 $>$ 2 so we swap them: (5 3 1 6 7 **8 2** 4) $\rightarrow$ (5 3 1 6 7 **2 8** 4)
i=6: 8 $>$ 4 so we swap them: (5 3 1 6 7 2 **8 4**) $\rightarrow$ (5 3 1 6 7 2 **4 8**)

We're now finished with the first pass, let's start again:
i=0: 5 $>$ 3 so we swap them: (**5 3** 1 6 7 2 8 4) $\rightarrow$ (**3 5** 1 6 7 2 4 8)
i=1: 5 $>$ 1 so we swap them: (3 **5 1** 6 7 2 8 4) $\rightarrow$ (3 **1 5** 6 7 2 4 8)
i=2: 5 $\leqslant$ 6 so we leave the 5 in place and we move on
i=3: 6 $\leqslant$ 7 so we leave the 6 in place and we move on
i=4: 7 $>$ 2 so we swap them: (3 5 1 6 **7 2** 4 8) $\rightarrow$ (3 5 1 6 **2 7** 4 8)
i=5: 7 $>$ 4 so we swap them: (3 5 1 6 2 **7 4** 8) $\rightarrow$ (3 5 1 6 2 **4 7** 8)
i=6: 7 $\leqslant$ 8 so we leave the 7 in place, and we have reached the end of the array.

We're now finished with the second pass.

Exercise 3 **(pen & paper)** Continue unrolling the algorithm until a whole pass does no swap.

[1] `https://en.wikipedia.org/wiki/Bubble_sort`

Exercise 4 (assembly) Write a program that implements bubble sort in an array of integers. Like in exercise 2, the length of the array is an explicit parameter.

To make your work easier, you may want to go after simpler subgoals first, for example:

- Given an index i, swap array elements T[i] and T[i+1]
- Perform a single pass on the entire array, swapping elements as needed
- Repeat such passes until the array is fully sorted