Computer Architecture

Exercise 5
(variables and memory)

Peter Stallinga
Universidade do Algarve 2024-2025

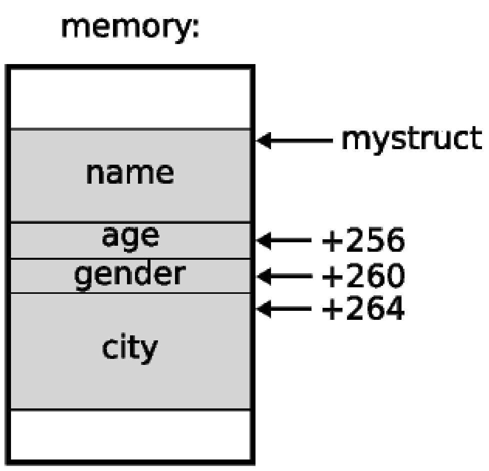
Access to main memory we have with MIPS instructions load word (`lw`) and store word (`sw`), where we can copy from registers to memory and back. There also exist versions copying less information at a time (`lb`, `sb`, `lh`, `sh` for bytes and halfwords). With this we implement the concept of variables of imperative programming:

```
.data

# ‘declare’ variables a b and c (each occupying 4 bytes)
#   int a, b, c;
  var_a: .space 4
  var_b: .space 4
  var_c: .space 4

.text

#   C language: c = a + b;
  lw $t0, var_a
  lw $t1, var_b
  add $t2, $t1, $t0
  sw $t2, var_c
```

The concept of arrays and structures does not exist in MIPS. We have to calculate the address of each element of the array or field of the structure we want to use. As an example, imagine in C the following structure

```
     struct {
          char name[256];
          int age;
          char gender;
          char city[256];
     } mystruct;
```

As shown in the figure:



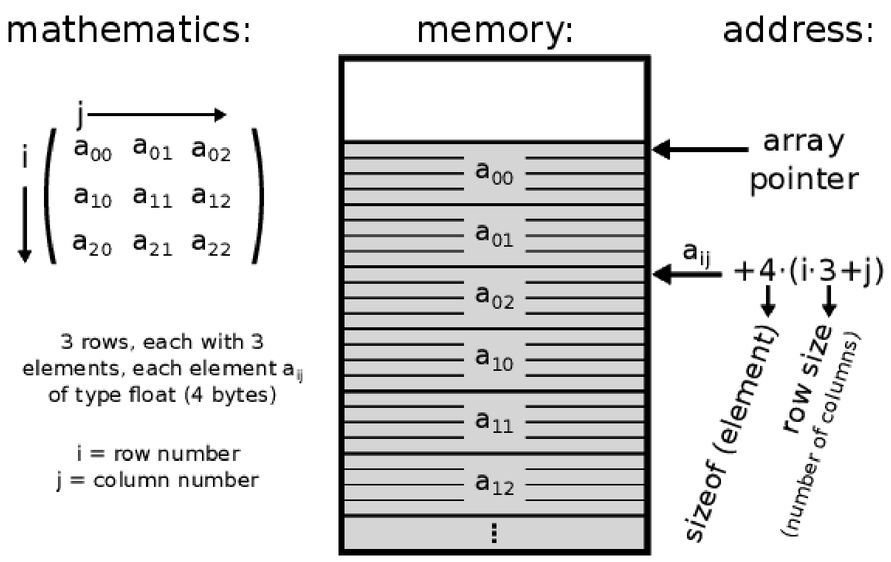
One such structure occupies 130 words, or 520 bytes. Each word (32 bits, 4 consecutive addresses) can contain 4 chars (8 bits). A single char would be 1/4 of a word and a single address. However, we do not know if our architecture can address individual bytes. In many architectures the addressability distance is the width of the data communication bus (in MIPS 32 bits). To avoid problems we reserve a full word (4 bytes) for a single char `gender`. This is called **data alignment**. If we do data alignment in the code, we can use the non-existing instruction `nop`, meaning "no operation".
In any case, after we have designed our memory structure, it is for us, the Assembly programmer, to calculate the addresses of the fields. As an example, to print the city field of the above structure, we have to do

```
.data
   mystruct: .word 0:129
.text
   la $a0, mystruct
   addi $a0, $a0, 264
   li $v0, 4   # print string
   syscall
```

In some cases we can directly use the offset to a pointer, as in
`264($a0)`
(Which is an example of direct indexed addressing).

Struct: Write a MIPS program that uses a structure for a football player (name, club, age, number of times played, number of goals). The program asks for values of the fields and then prints the structure.

Likewise, arrays do not exist in Assembly. It is us who have to do the calculation. An example is the matrix below:


mathematics:
memory:
address:
j
i
a_{00} a_{01} a_{02}
a_{10} a_{11} a_{12}
a_{20} a_{21} a_{22}
3 rows, each with 3 elements, each element a_{ij} of type float (4 bytes)
i = row number
j = column number
array pointer
a_{ij}
$+4\cdot(i\cdot 3+j)$
sizeof (element)
row size
(number of columns)

Have you ever wondered why in C you can pass a pointer to an array to a function, but you have to specify the second dimension? (All dimensions, except the first). As in

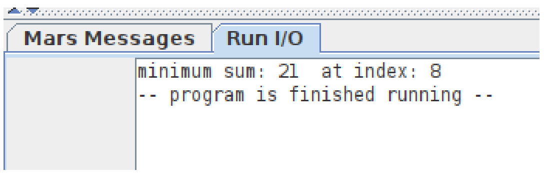
```
int myfunction(int a[][3])
```

It is because without the information of the dimension the location in memory of elements cannot be determined! Look in the image above, the row size has to be known. The column size is not needed, though.

Array: Write a MIPS program that, given an array of 10 points in 3-dimensional space (x, y, z), defined in the `.data` segment (no need for syscall-reading), determines which has the lowest sum x+y+z.

The data points are:

```
1, 10, 18,
2, 2, 20,
13, 13, -1,
20, 20, 100,
8, 9, 10,
11, 12, 1,
20, 1, 2,
18, 8, 8,
9, 9, 3,
10, 9, 5
```



```
minimum sum: 21  at index: 8
-- program is finished running --
```

The relevant (new) instructions for today are:

`syscall`	Operating system call (I/O)
`lw`, `sw`	Load word, store word
`nop`	No operation