Computer Systems
Lecture 10

Arrays

Dr José Cano, Dr Lito Michala
School of Computing Science
University of Glasgow
Spring 2020

Outline

- Address arithmetic
- Arrays
  - Representation
  - Allocation
  - Indexed addressing
- Array traversal and for loops
- Example program ArrayMax
- Programming tips

Why [R0]?

- So far, we have always been writing [R0] after constants or names
  - lea R2,39[R0]
  - load R3,xyz[R0]
  - store R4,total[R0]

- Why?

- This is part of a general and powerful technique called address arithmetic

Address arithmetic

- Every piece of data in the computer (in registers, or memory) is a word of bits
- A word can represent several different kinds of data (and instructions)
  - So far, we've just been using integers
  - Represented with two's complement: $-2^{15}, \ldots, -1, 0, 1, 2, \ldots, 2^{15} - 1$
- Now, we'll start doing computations with addresses too
- Addresses are unsigned numbers: 0, 1, 2, … , 65535
  - Represented in binary

What can you do with address arithmetic?

- Powerful data structures
  - **Today**: Arrays
  - **Next week**: Pointers and records
  - Linked lists, queues, dequeues, stacks, trees, graphs, hash tables, … (subject of Algorithms and Data Structures)

- Powerful control structures
  - Input/Output
  - Procedures and functions
  - Recursion
  - Case dispatch
  - Coroutines, classes, methods

Data structures

- An ordinary variable holds one value (e.g. an integer)
- A data structure can hold many individual elements
- A data structure is a container
- The simplest data structure: array
- There are many more data structures!
- The **key idea**: we will do arithmetic on addresses

Outline

- Address arithmetic
- Arrays
  - Representation
  - Allocation
  - Indexed addressing
- Array traversal and for loops
- Example program ArrayMax
- Programming tips

Arrays

- In **mathematics**, an array (vector) is a sequence of indexed values x_0, x_1, ... , x_n - 1
  - x is the entire array
  - x_3 is one specific element of the array with index 3
  - It's useful to refer to an arbitrary element by using an integer variable as index (subscript notation): x_i

- In **programming languages**, we refer to x_i as x[i]

- Arrays are ubiquitous: used in all kinds of applications

Representing an array

- An array is represented in a computer by placing the elements in consecutive memory locations
- The array x starts in memory at some location: for example 01a5
- The address of the array x is the address of its first element x[0]
- The elements follow in consecutive locations

value	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]
address	… 01a5	01a6	01a7	01a8	01a9	01aa	01ab …

The address of x[i] is x+i

Allocating an array

- An array is in memory along with other data
  - After the trap instruction that terminates the program
- You can allocate the elements and give them initial value with data statements
- Use the name of the array as a label on the first element (the one with index 0)
- Don't put labels on the other elements

Example of array allocation

```
...
trap R0,R0,R0          ; terminate
; Variables and arrays
abc data 25            ; some variable
n data 6               ; size of array x
x data 13              ; x[0]
data 189               ; x[1]
data 870               ; x[2]
data 42                ; x[3]
data 0                 ; x[4]
data 1749              ; x[5]
def data 0             ; some other variable
```

What about big arrays?

- In the programs we'll work with, arrays will be small (e.g. 5-10 elements)
- In real scientific computing, it's common to have large arrays with thousands (or even millions) of elements
- It would be horrible to have to write thousands of data statements!
- In large scale software, arrays are allocated dynamically with help from the operating system
  - The user program calculates how large an array it wants, and stores that in a variable (e.g. n = 40000)
  - It uses a trap to request (from the operating system) a block of memory big enough to hold the array
  - The operating system returns the address of this block to the user program
- We won’t do this: we will just allocate small arrays using data statements

Accessing an element of an array

- Suppose we have array x with elements x[0], x[1], ..., x[n-1]
- Elements are stored in consecutive memory locations
- Use the label x to refer to the array (x is also the location of x[0])
- The address of x[i] is x+i
- To do any calculations on x[i], we must load it into a register, or store a new value into it
- But how?
  - If you try load R4,x[R0] the effect will be R4 := x[0]
  - We need a way to access x[i] where i is a variable

Effective address

- An RX instruction specifies addresses in two parts
  - Examples: result[R0] or x[R4] or $00a5[R2]
  - The displacement is a 16 bit constant: you can write the number, or use a name (the assembler will put in the address for you)
  - The index register is written in brackets
- The machine adds the displacement to the value in the index register
  - This is called the effective address
- The instruction is performed using the effective address

Using the effective address

- The addressing mechanism is flexible!
- You can access an ordinary variable: load R2,sum[R0]
  - R0 always contains 0, so the effective address is just the address of sum
- You can access an array element
  - If R8 contains an index i, then load R2,x[R8] will load x[i] into R2
- Other cases: e.g. effective address is the content of a register load R2,0[R8]
- **There's more**: effective addresses are used to implement…
  - pointers, functions, procedures, methods, classes, instances, jump tables, case dispatch, coroutines, records, interrupt vectors, lists, heaps, trees, forests, graphs, hash tables, activation records, stacks, queues, dequeues, etc

Addressing modes

- An addressing mode is a scheme for specifying the address of data
- **Sigma16** has one addressing mode: displacement[index], e.g. x[R4]
- Many older computers provided many addressing modes
  - One for variables, another for arrays, yet another for linked lists, still another for stacks, and so on
- It's more efficient to provide just one or two flexible addressing modes, rather than a big collection of them

Using effective address for an array

- Suppose we want to execute x[i] := x[i] + 50

```
lea R1,50[R0]         ; R1 := 50
load R5,i[R0]         ; R5 := i
load R6,x[R5]         ; R6 := x[i]
add R6,R6,R1          ; R6 := x[i] + 50
store R6,x[R5]        ; x[i] := x[i] + 50
```

Outline

- Address arithmetic
- Arrays
  - Representation
  - Allocation
  - Indexed addressing
- Array traversal and for loops
- Example program ArrayMax
- Programming tips

Array traversal

- A typical operation on an array is to traverse it

- That means to perform a calculation on each element

- Here's a loop that doubles each element of x

```
i := 0;
while i < n do
  {  x[i] := x[i] * 2;
     i := i + 1;
  }
```

For loops

- A for loop is designed specifically for array traversal
- It handles the loop index automatically
- It sets the index to each array element index and executes the body
- The intuition is “do the body for every element of the array”

```
for i := exp1 to exp2 do
      { statements }
```

Array traversal with while and for

- Here is the program that doubles each element of x, written with both constructs

```
i := 0;
while i < n do
  {  x[i] := x[i] * 2;
     i := i + 1;
  }
```

```
for i := 0 to n-1 do
  { x[i] := x[i] * 2; }
```

Translating the for loop to low level

- High level for loop (with any number of statements in the body)

```
for i := exp1 to exp2 do
   {  statement1;
      statement2;
   }
```

- Translate to low level with this pattern

```
      i := exp1;
loop: if i > exp2 then goto loopdone;
      statement1;
      statement2;
      i := i + 1;
      goto loop;
loopdone:
```

Alternative syntax for for loops

- In languages derived from C (C++, Java, C#, and many more) you will see for loops written like this

```
for (i=x; i<y; i++)
  { S1; }
S2;
```

Outline

- Address arithmetic
- Arrays
  - Representation
  - Allocation
  - Indexed addressing
- Array traversal and for loops
- Example program ArrayMax
- Programming tips

Example program ArrayMax

- A complete programming example

- **The problem**: find the maximum element of an array

- To do this we need to
  - Allocate an array
  - Loop over the elements
  - Access each element
  - Perform a conditional

- This example puts all our techniques together into one program

State what the program does

```
; Program ArrayMax
; John O'Donnell

;---------------------------------------------------------------------
; The program finds the maximum element of an array.  It is given
;    *  a natural number n, assume n>0
;    *  an n-element array x[0], x[1], ..., x[n-1]
; and it calculates
;    * max = the maximum element of x

; Since n>0, the array x contains at least one element, and a maximum
; element is guaranteed to exist.
```

High level algorithm

```
;---------------------------------------------------------
; High level algorithm

;    max := x[0];
;    for i := 1 to n-1 do
;        { if x[i] > max
;             then max := x[i];
;        }
```

Translate high level code to low level “goto form”

```
;------------------------------------------
; Low level algorithm

;       max := x[0]
;       i := 1
; forloop:
;       if i >= n then goto done
;       if x[i] <= max then goto skip
;       max := x[i]
; skip:
;       i := i + 1
;       goto forloop
; done:
;       terminate
```

Specify how the registers are used

- Note that the program is written in the “register variable style”

```
;------------------------------------------
; Assembly language

; Register usage
;   R1 = constant 1
;   R2 = n
;   R3 = i
;   R4 = max
```

Block of statements to initialise registers

```
; Initialise
        lea     R1,1[R0]          ; R1 = constant 1
        load    R2,n[R0]          ; R2 = n
; max := x[0]
        load    R4,x[R0]          ; R4 = max = x[0]
; i := 1
        lea     R3,1[R0]          ; R3 = i = 1
```

Beginning of loop

```
; Top of loop, determine whether to remain in loop
forloop
; if i >= n then goto done
       cmp    R3,R2              ; compare i, n
       jumpge done[R0]           ; if i>=n then goto done
```

Body of loop: if-then

```
; if x[i] <= max then goto else
        load    R5,x[R3]            ; R5 = x[i]
        cmp     R5,R4               ; compare x[i], max
        jumple skip[R0]             ; if x[i] <= max then goto skip

; max := x[i]
        add    R4,R5,R0             ; max := x[i]
```

End of loop

```
skip
; i := i + 1
       add    R3,R3,R1          ; i = i + 1
; goto forloop
       jump   forloop[R0]       ; go to top of forloop
```

Finish

```
; Exit from forloop
done    store R4,max[R0]          ; max = R4
; terminate
        trap  R0,R0,R0            ; terminate
```

Data definitions

```
; Data area

n           data    6
max         data    0
x           data   18
            data    3
            data   21
            data   -2
            data   40
            data   25
```

Outline

- Address arithmetic
- Arrays
  - Representation
  - Allocation
  - Indexed addressing
- Array traversal and for loops
- Example program ArrayMax
- Programming tips

A useful convention

- The instruction set is designed to be regular, and follow consistent conventions
  - This makes programming easier
  - It also helps with the hardware design!
- For most instructions, the operands follow the pattern of an assignment statement: information goes right to left
  - Assignment statement: reg1 := reg2 + reg3
  - Add instruction: add R1,R2,R3
  - The two operands on the right (R2, R3) are added, and placed in the destination on the left (R1)
  - Load instruction: load R1,x[R0] means R1 := x
- An **exception**: store
  - store R1,x[R0] means x := R1: the information goes from left to right
  - Why? Doing it this way makes the digital circuit (the processor) a bit faster

Programming tip: Copying one register to another

- Here's a useful tip (a standard programming technique)

- Sometimes you want to copy a value from one register to another
  - R3 := R12

- There's a standard way to do it
  - add R3,R12,R0   ; R3 := R12

- The **idea** is that R12 + 0 = R12!

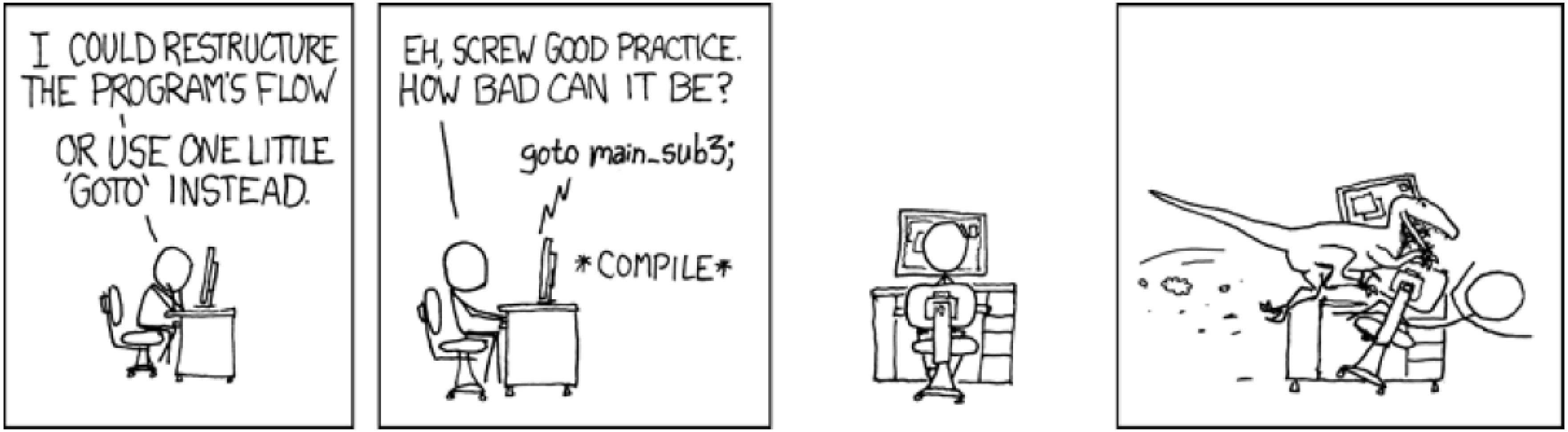
- Why do it this way?
  - It's actually more efficient than providing a separate instruction just to copy the register!

Using load and store

- A common error is to confuse load and store

- The main points to remember
  - We need to keep variables in memory (most of the time) because memory is big (there aren't enough registers to hold all your variables)
  - The computer hardware can do arithmetic on data in registers, but it cannot do arithmetic on data in memory
  - Therefore, to do arithmetic on variables, you must
    - Copy the variables from memory to registers (load)
    - Do the arithmetic in the registers (add, sub, …)
    - Copy the result from registers back to memory (store)

goto



https://xkcd.com/292/