Computer Systems 1
Lecture 15

Arrays and Pointers

Dr. John T. O'Donnell
School of Computing Science
University of Glasgow

Topics

1. Programming techniques
   - Compound Boolean expressions
   - Condition code
   - Repeat-until loop
   - Input/Output
2. Arrays and pointers
3. Stack overflow

A collection of programming techniques

- Compound Boolean expressions: “short circuit” evaluation
- The condition code and “cmp jumpgt” style comparisons
- loops: for loop, while loop, repeat until loop
- Input/Output: write characters, not numbers

Compound Boolean expressions

Notation: various programming languages use several slightly different notations are used for Boolean operators

```
i<n and x[i]>0        i<n && x[i]>0      i<n & x[i]>0
i<n or  j<n           i<n || j<n         i<n | x[i]>0
```

"Short circuit" expressions

- Suppose `x` is an array with `n` elements
- Consider `i<n && x[i]>0`
- If the first expression `i<n` is False, then the whole expression is False
- In that case, there is no need to evaluate the second expression `x[i]>0`
- We can "short circuit" the evaluation
- Big advantage: if `i<n` is False, then `x[i]` does not exist and evaluating it could cause an error
- So it is *essential* not to evaluate the second expression if the first one is false

Implementing a compound boolean expression

```
while i<n && x[i]>0 do S

; if not (i<n && x[i]>0) then goto loopDone
    load   R1,i[R0]          ; R1 := i
    load   R2,n[R0]          ; R2 := n
    cmplt  R3,R1,R2          ; R3 := i<n
    jumpf  R3,loopDone[R0]   ; if not (i<n) then goto loopDone
    load   R3,x[R1]          ; R3 := x[i]   safe because i<n
    cmpgt  R4,R3,R0          ; R4 := x[i]>0
    jumpf  R4,loopDone[R0]   ; if not (x[i]>0) then goto loopD
```

This is better than evaluating both parts of the expression and calculating logical and

Condition code

- We have seen one style for comparison and conditional jump

```
cmplt  R3,R8,R4
jumpt  R3,someplace[R0]
```

- There is also another way you can do it

```
cmp    R8,R4              ; no destination register
jumplt someplace[R0]      ; jump if less than
```

- The cmp instruction sets a result (less than, equal, etc) in R15 which is called the condition code
- There are conditional jumps for all the results: jumpeq, jumplt, jumple (jump if less than or equal), etc
- An advantage is that you don't need to use a register for the boolean result

Repeat-until loop

This is similar to a while loop, except you decide whether to continue at the end of the loop

```
repeat
  {S1; S2; S3}
until i>n;
```

This is equivalent to

```
S1; S2; S3;
while not (i>n) do
  {S1; S2; S3}
```

The while loop is used far more often, but if you need to go through the loop at least one time, the repeat-until is useful

Input/Output

- A character is represented by a code using ASCII or Unicode
  - http://www.asciitable.com/
  - https://unicode-table.com/en/
- digit characters 0..9 have codes (in decimal) 48..57
  - Example: '3' is represented by the number 51, not by the number 3
- lower case a..z have codes (in decimal) 97..122
- upper case A..Z have codes (in decimal) 65..90
- To print a number, we need to convert it to a string of characters

Converting a number to a string

- We actually need to do arithmetic to convert a binary number to decimal, and to a string of decimal digits
- The lab exercise gives the algorithm to do this
- It needs to divide the number by 10 to get the quotient and the remainder
- div R1,R2,R3
  - Divides R2/R3
  - The quotient goes into R1 (the destination register)
  - The remainder goes into R15 (always R15, you cannot change this)
- The algorithm repeatedly divides the number by 10; the remainder is used to get a digit character

Arrays and pointers

We have seen how to access an array element using an index register

```
; R5 := x[i]
    load   R2,i[R0]     ; R2 := i
    load   R5,x[R2]     ; R5 := x[i]
```

Sum of an array x using index: high level

Suppose x is an array of numbers, and sizeX is the number of elements. We want to add up all the elements of x.

A for loop is convenient (the whole purpose of the for loop is for writing this kind of loop):

```
sum := 0;
for i := 0 to sizeX do
  { sum := sum + x[i]; }
```

You can also use a while loop:

```
sum := 0;
i := 0;
while i < sizeX do
  { sum := sum + x[i];
    i := i + 1; }
```

Arrays and pointers

- There is also another way to access an array element, using pointers instead of indexes
- To do this, we will perform arithmetic on pointers

Accessing an array element using a pointer

- Create a pointer p to the beginning of the array x, so p is pointing to x[0]
  lea R1,x[R0]
- To access the current element, follow p:
  load R2,0[R1]
- To move on to the next element of the array, increment p
  lea R1,1[R1]
- Notice that we are doing arithmetic on pointers

```
x    data   34    ; first element of x
     data   82
     data   91
     data 29      ; last element of x
xEnd              ; address of first word after array x
```

Sum of an array x using pointers: high level

```
sum := 0;
p := &x;
q := &xEnd;
while p<q do
  { sum := sum + *p;
    p := p + 1; }
```

- In assembly language, we can use lea to increment the pointer.
- Suppose p is in R1, then
  `lea R1,1[R1] ; p := p + 1`
- We are incrementing p by the size of an array element

Sum of an array x using pointers: assembly language

```
;   R1 = p = pointer to current element of array x
;   R2 = q = pointer to end of array x
;   R3 = sum of elements of array x

    lea    R1,x[R0]       ; p := &x
    lea    R2,xEnd[R0]    ; q := %xEnd
    add    R3,R0,R0       ; sum := 0
sumLoop
    cmplt R4,R1,R2        ; R4 := p<q
    jumpf sumLoopDone     ; if not p<q then goto sumLoopDone
    load   R4,0[R1]       ; R4 := *p  (this is current element of x)
    add    R3,R3,R4       ; sum := sum + *p
    lea    R1,1[R1]       ; p := p+1  (point to next element of x)
    jump   sumLoop[R0]    ; goto sumLoop
sumLoopEnd

x   data  23   ; first element of x
    data  42   ; next element of x
    data 19    ; last element of x
xEnd
```

Comparing the two approaches

- Accessing elements of an array using index
  - Get x[i] with load R5,x[R1] where R1=i
  - Move to next element of array by i := i+1
  - Determine end of loop with `i < xSize`
  - Know in advance how many iterations: xSize
  - A for loop is convenient
- Accessing elements of an array using pointer
  - Initialize p with lea R1,x[R0]
  - Get x[i] with load R5,0[R1] where R1=p
  - Move to next element of array by `p := p+1`
  - Determine end of loop with `p < q` (q points to end of array)
  - Don't need to know in advance how many iterations
  - Need to use a while loop
- Both techniques are important
- If you have an array of records, it's easier to use a pointer

Records

```
program Records
 { x, y :
     record
       { fieldA : int;
         fieldB : int;
         fieldC : int; }
   x.fieldA := x.fieldB + x.fieldC;
   y.fieldA := y.fieldB + y.fieldC;
 }
```

Suppose we have an array of these records, and want to

- set fieldA := fieldB + fieldC in every record in the array
- Calculate the sum of the fieldA in every record

Traverse array of records with indexing

```
sum := 0;
for i := 0 to nrecords do
  { RecordArray[i].fieldA :=
        RecordArray[i].fieldB + RecordArray[i].fieldC;
    sum := RecordArray[i].fieldA; }
```

- This is ok
- But it is a little awkward

Traverse array of records with pointers: high level

```
sum := 0;
p := &RecordArray;
q := &RecordArrayEnd;
while p < q do
  { *p.fieldA := *p.fieldB + *p.fieldC;
    sum := sum + *p.fieldA;
    p := p + RecordSize; }
```

In professional programming, this is often preferred because accessing the elements of the records is easier (it's easier to access an "element of an element" via pointer)

Traverse array of records with pointers: low level

```
;     sum := 0;
;     p := &RecordArray;
;     q := &RecordArrayEnd;
; RecordLoop
;     if (p<q) = False then goto recordLoopDone;
;     *p.fieldA := *p.fieldB + *p.fieldC;
;     sum := sum + *p.fieldA;
;     p := p + RecordSize;
;     goto recordLoop;
; RecordLoopDone
```

Traverse array of records with pointers: assembly language

```
;    R1 = sum
;    R2 = p (pointer to current element)
;    R3 = q (pointer to end of array)
;    R4 = RecordSize

    lea     R1,0[R0]                 ; sum := 0
    lea     R2,RecordArray[R0]       ; p := &RecordArray;
    lea     R3,RecordArrayEnd[R0]    ; q := &RecordArray;
    load    R4,RecordSize[R0]        ; R4 := RecordSize
RecordLoop
    cmplt   R5,R2,R3                 ; R5 := p<q
    jumpf   R5,RecordLoopDone[R0]    ; if (p<q) = False then goto RecordLoop
    load    R5,1[R2]                 ; R5 := *p.fieldB
    load    R6,2[R2]                 ; R6 := *p.fieldC
    add     R7,R5,R6                 ; R7 := *p.fieldB + *p.fieldC
    store   R7,0[R2]                 ; *p.fieldA := *p.fieldB + *p.fieldC
    add     R1,R1,R7                 ; sum := sum + *p.fieldA
    add     R2,R2,R4                 ; p := p + RecordSize
    jump    RecordLoop[R0]           ; goto RecordLoop
RecordLoopDone
```

Stack overflow

- The mechanism for calling a procedure and returning is fairly complicated
- Rather than introducing all the details at once, we have looked at several versions, introducing the concepts one at a time
- Now we introduce the next level:
  - Simplify calling a procedure
  - The procedure checks for stack overflow
- We need two more registers dedicated to procedures
  - R12 holds stack top (the highest address in current stack frame)
  - R11 holds stack limit (the stack is not allowed to grow beyond this address)

Register usage

See the PrintIntegers program for examples

```
; Global register usage
;    R0 = constant 0
;    R1, R2, R3 are used for parameters and return values
;    R4 - R10 are available for local use in a procedure
;    R11 = stack limit
;    R12 = stack top
;    R13 = return address
;    R14 = stack pointer
;    R15 is transient condition code
```

Initialize the stack

```
; Structure of stack frame for main program, frame size=1
;    0[R14]  dynamic link is nil

; Initialize the stack
     lea     R14,CallStack[R0]   ; initialise stack pointer
     store   R0,0[R14]           ; main program dynamic link = ni
     lea     R12,1[R14]          ; initialise stack top
     load    R1,StackSize[R0]    ; R1 := stack size
     add     R11,R14,R1          ; StackLimit := &CallStack + Sta
```

Calling a procedure

- To call a procedure PROC:
  - Place any parameters you're passing to PROC in R1, R2, R3
  - jal R13,PROC[R0]

Structure of Procedure stack frame

(This is procedure PrintInt, see lab exercise)

```
; Arguments
;   R1 = x          = two's complement number to print
;   R2 = FieldSize = number of characters for print field
;        require FieldSize < FieldSizeLimit

; Structure of stack frame, frame size = 6
;     5[R14]  save R4
;     4[R14]  save R3
;     3[R14]  save R2 = argument fieldsize
;     2[R14]  save R1 = argument x
;     1[R14]  return address
;     0[R14]  dynamic link points to previous stack frame
```

Called procedure creates its stack frame

```
PrintInt
; Create stack frame
    store  R14,0[R12]          ; save dynamic link
    add    R14,R12,R0          ; stack pointer := stack top
    lea    R12,6[R14]          ; stack top := stack ptr + fran
    cmp    R12,R11             ; stack top ~ stack limit
    jumpgt StackOverflow[R0]   ; if top>limit then goto stack
    store  R13,1[R14]          ; save return address
    store  R1,2[R14]           ; save R1
    store  R2,3[R14]           ; save R2
    store  R3,4[R14]           ; save R3
    store  R4,5[R14]           ; save R4
```

Procedure finishes and returns

```
; return
    load   R1,2[R14]        ; restore R1
    load   R2,3[R14]        ; restore R2
    load   R3,4[R14]        ; restore R3
    load   R13,1[R14]       ; restore return address
    load   R14,0[R14]       ; pop stack frame
    jump   0[R13]           ; return
```

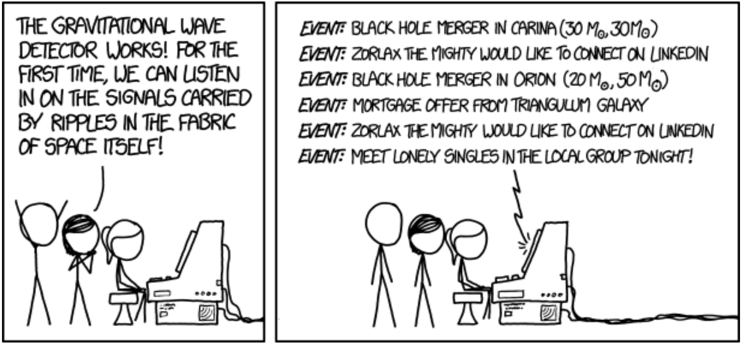
Stack overflow

If the stack is full and a procedure is called, this is a fatal error

```
StackOverflow
    lea     R1,2[R0]
    lea     R2,StackOverflowMessage[R0]
    lea     R3,15[R0]    ; string length
    trap    R1,R2,R3     ; print "Stack overflow\n"
    trap    R0,R0,R0     ; halt

StackOverflowMessage
    data     83    ; 'S'
    data    116    ; 't'
    data     97    ; 'a'
    data     99    ; 'c'
    data    107    ; 'k'
...
```

Gravitational waves



https://xkcd.com/1642/