Computer Systems, Spring 2019

Week 8 Lab

Pointers: Records, Arrays, Procedures, I/O

Solutions

1 Review pointers

It's important to understand what the operators `&` and `*` mean, and how to implement them in assembly language. Each of them requires just one instruction! These operators are no more complicated than addition or subtraction.

2 Accessing record fields via pointer

Now we will apply the technique illustrated in the `Pointer` program to allow flexible access to records.

Download the program `RecordsEXERCISE.asm.txt`. Study the program, and step through it with the Sigma16 system. Complete the program by filling in the necessary instructions at the points labelled `; INSERT SOLUTION HERE`. The comments in the program tell you what to do.

```
; Records --- Solution
; Sigma16 program showing how to access record fields
; John O'Donnell, 2019

;---------------------------------------------------------------------
; High level algorithm in Sigma

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

;---------------------------------------------------------------------
; Simplistic approach, with every field of every record named
; explicitly.

; In record x,  fieldA := fieldB + fieldC
; x.fieldA := x.fieldB + x.fieldC
    load   R1,x_fieldB[R0]
```

```
    load   R2,x_fieldC[R0]
    add    R1,R1,R2
    store  R1,x_fieldA[R0]

; In record y,  fieldA := fieldB + fieldC
; y.fieldA := y.fieldB + y.fieldC
    load   R1,y_fieldB[R0]
    load   R2,y_fieldC[R0]
    add    R1,R1,R2
    store  R1,y_fieldA[R0]

;---------------------------------------------------------------------
; A much better approach: Access the record fields through a pointer
; to the record.  This way, we can make the same code work for any
; record with the same fields

; Set x as the current record by making R3 point to it
; R3 := &x;

; SOLUTION
    lea    R3,x[R0]     ; R3 := &x

; Perform the calculation on the record that R3 points to
; *R3.fieldA := *R3.fieldB + *R3.fieldC
; This will be equivalent to x.fieldA := x.fieldB + x.fieldC

; SOLUTION
    load   R1,1[R3]     ; R1 := (*R3).fieldB
    load   R2,2[R3]     ; R2 := (*R3).fieldC
    add    R1,R1,R2     ; R1 := (*R3).fieldB + (*R3).fieldC
    store  R1,0[R3]     ; *R3.fieldA := (*R3).fieldB + (*R3).fieldC

; Set y as the current record by making R3 point to it
; R3 := &y;

; SOLUTION
    lea    R3,y[R0]     ; R3 := &y

; Perform the calculation on the record that R3 points to
; *R3.fieldA := (*R3).fieldB + (*R3).fieldC
; This will be equivalent to y.fieldA := y.fieldB + y.fieldC

; SOLUTION
    load   R1,1[R3]     ; R1 := (*R3).fieldB
    load   R2,2[R3]     ; R2 := (*R3).fieldC
    add    R1,R1,R2     ; R1 := (*R3).fieldB + (*R3).fieldC
    store  R1,0[R3]     ; *R3.fieldA := (*R3).fieldB + (*R3).fieldC

; The conclusion is that we could have a program do this computation
; (fieldA := fieldB + fieldC) on *any* record.  We don't even need to
```

```
; have the records defined with data statements giving them individual
; names.

;-------------------------------------------------------------------
; So let's do that, with a loop that iterates over an array of
; records, performs the fieldA := fieldB + fieldC computation on each
; of them, and also computes the sum of all the fieldA results.  An
; array of nrecords is defined below, with initial values of the
; records.

; We could use array indexing, like this (note that we would need to
; multiply the index i by the array element size to get the address of
; an element).

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

; But let's use pointers to access the array elements instead. A
; variable p points to the current element of the array, and on each
; iteration we need to add the size of the record (which is 3) to p.

; Here's the high level algorithm:
;    sum := 0;
;    p := &RecordArray;
;    q := &RecordArrayEnd;
;    while p < q do
;      { *p.fieldA := *p.fieldB + *p.fieldC;
;        sum := sum + *p.fieldA;
;        p := p + RecordSize; }

; Notice that we have two different approaches.  It's interesting to
; compare them:
;    (1)  access element of array by index
;         Need to do arithmetic on index (multiply it by 3)
;         Need to have a variable giving number of elements in array
;         Don't need to know the address of the end of the array
;         Use a for loop for the iteration
;    (2)  access element of array by pointer
;         Need to do arithmetic on pointer (add 3 to it)
;         Don't need a variable giving number of elements in array
;         Do need to know the address of the end of the array
;         Use a while loop for the iteration

; Which of these approaches is better?  That depends entirely on the
; situation; sometimes the index version is better, sometimes the
; pointer version is better.  And what does "better" mean?  There are
; many things to consider, including simplicity, readability of the
```

```
; code, runtime efficiency, flexibility in providing the input, and
; more.

; Translate the high level algorithm to low level (pointer/while version)
; SOLUTION

;    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

; Translate it to assembly language
; SOLUTION

; Register usage
;   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 RecordLoopDone
    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

; Terminate
    trap   R0,R0,R0    ; halt

;------------------------------------------------------------------------
; Data definitions

nrecords   data   5    ; an array with nrecords elements is definedbelow
RecordSize data   3    ; there are 3 words in the record
```

```
RecordArray            ; this is the address of the array

; The record x, record[0]
x
x_fieldA   data   3    ; offset 0 from x  &x_fieldA = &x
x_fieldB   data   4    ; offset 1 from x  &x_fieldB = &x + 1
x_fieldC   data   5    ; offset 2 from x  &x_fieldC = &x + 2

; The record y, record[1]
y
y_fieldA   data  20    ; offset 0 from y  &y_fieldA = &y
y_fieldB   data  21    ; offset 1 from y  &y_fieldB = &y + 1
y_fieldC   data  22    ; offset 2 from y  &y_fieldC = &y + 2

; More records, we haven't even given them individual names
; record[2]
           data  30    ; fieldA
           data  31    ; fieldB
           data  32    ; fieldC
; record[3]
           data  30    ; fieldA
           data  31    ; fieldB
           data  32    ; fieldC
; record[4]
           data  40    ; fieldA
           data  41    ; fieldB
           data  42    ; fieldC
RecordArrayEnd         ; this is the address of the end of the array
```

3 PrintIntegers

Here is a translation of the high level algorithm for ShowInt into low level:

```
;------------------------------------------------------------------
; ShowInt: low level algorithm

; SOLUTION

; procedure ShowInt (x:Int, *bufstart:Char, bufsize:Int) : Int
;   negative := False
;   bufend := bufstart + bufsize - 1  ; ptr to last char in buf
;   if x >= 0 then goto NotNeg
;   x := -x
;   negative := True
; NotNeg
;   p := bufend

; DigitLoop
;     r := x mod 10
;     x := x div 10
```

```
;      *p := digits[r]
;      p := p - 1
;      if x = 0 then goto DigitLoopDone
;      if p < bufstart then goto DigitLoopDone
;      goto DigitLoop
; DigitLoopDone

;    if x > 0 then goto ShowIntTooBig
;    if negative /= 0 then goto ShowIntFinish
;    if not p >= bufstart then goto ShowIntFinish
;    goto ShowIntTooBig

; ShowIntTooBig
;    p := bufstart
; ShowIntHashLoop
;    if p < bufend then goto ShowIntHashLoopDone
;    *p := HashChar
;    p := p + 1
;    goto ShowIntHashLoop
; ShowIntHashLoopDone
;    k := 0
;    goto ShowIntDone

; ShowIntFinish
;    if not negative then goto ShowIntNotNeg
;    *p := MinusSign
;    p := p - 1
; ShowIntNotNeg
;    k := p + 1 - bufstart
; ShowIntSpaceLoop
;    if p < bufstart then goto ShowIntSpaceLoopDone
;    *p := Space
;    p := p - 1
;    goto ShowIntSpaceLoop
; ShowIntSpaceLoopDone
; ShowIntDone
;    return k
```

And here is a translation into assembly language:

```
; ShowInt: assembly language

; SOLUTION

; Arguments (x:Int, *bufstart:Char, bufsize:Int)
;    R1 = x = integer to convert
;    R2 = bufstart = address of string
;    R3 = bufsize = number of characters in string
;    R12 = return address
; Result
;    R1 = k = number of leading spaces; -1 if overflow
```

```
; Local register usage
;   R4  = constant 1
;   R5  = negative
;   R6  = bufend
;   R7  = p
;   R8  = temp
;   R9  = r
;   R10 = constant 10

; Structure of stack frame, frame size = 12
;  11[R14]  save R10
;  10[R14]  save R9
;   9[R14]  save R8
;   8[R14]  save R7
;   7[R14]  save R6
;   6[R14]  save R5
;   5[R14]  save R4
;   4[R14]  save R3
;   3[R14]  save R2
;   2[R14]  save R1
;   1[R14]  return address
;   0[R14]  dynamic link points to previous stack frame

ShowInt
; Create stack frame
    store  R14,0[R12]          ; save dynamic link
    add    R14,R12,R0          ; stack pointer := stack top
    lea    R12,12[R14]          ; stack top := stack ptr + frame size
    cmp    R12,R11             ; stack top ~ stack limit
    jumpgt StackOverflow[R0]   ; if top>limit then goto stack overflow
    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
    store  R5,6[R14]           ; save R5
    store  R6,7[R14]           ; save R6
    store  R7,8[R14]           ; save R7
    store  R8,9[R14]           ; save R8
    store  R9,10[R14]          ; save R9
    store  R10,11[R14]         ; save R10

    lea    R4,1[R0]         ; R4 := constant 1
    lea    R10,10[R0]       ; R10 := constant 10
    add    R5,R0,R0         ; negative := False
    add    R6,R2,R3         ; bufend := bufstart + bufsize
    sub    R6,R6,R4         ; bufend := bufstart + bufsize - 1

    cmp    R1,R0            ; compare x, 0
```

```
    jumpge SInotNeg[R0]     ; if nonnegative then goto SInotNeg
    sub    R1,R0,R1         ; x := -x
    add    R5,R1,R0         ; negative := True
SInotNeg
    add    R7,R6,R0         ; p := bufend

SIdigLp
    div    R1,R1,R10        ; x := x div 10
    add    R9,R15,R0        ; r := x mod 10
    load   R8,Digits[R9]    ; temp := Digits[r]
    store  R8,0[R7]         ; *p := digits[r]
    sub    R7,R7,R4         ; p := p - 1
    cmp    R1,R0
    jumpeq SIdigLpEnd[R0]   ; if x = 0 then goto SIdigLpEnd
    cmp    R7,R2            ; compare p, bufstart
    jumplt SIdigLpEnd[R0]   ; if p < bufstart then goto SIdigLpEnd
    jump   SIdigLp[R0]      ; goto SIdigLp
SIdigLpEnd

    cmp    R1,R0            ; compare x, 0
    jumpgt SItooBig[R0]     ; if x > 0 then goto SItooBig
    cmp    R5,R0            ; is x negative?
    jumpeq SIfinish[R0]     ; if nonnegative then goto SIfinish
    cmp    R7,R2            ; compare p, bufstart
    jumpge SIfinish[R0]     ; if p >= bufstart then goto SIfinish
    jump   SItooBig[R0]     ; goto SItooBig

SItooBig
    add    R7,R2,R0         ; p := bufstart
SIhashLp
    cmp    R7,R6            ; compare p, bufend
    jumpgt SIhashLpEnd[R0]  ; if p > bufend then goto SIhashLpEnd
    load   R8,Hash[R0]      ; R8 := '#'
    store  R8,0[R7]         ; *p := '#'
    add    R7,R7,R4         ; p := p + 1
    jump   SIhashLp[R0]     ; goto SIhashLp
SIhashLpEnd
    add    R1,R0,R0         ; k := 0
    jump   SIend[R0]        ; goto SIend

SIfinish
    cmp    R5,R0            ; compare R5, False
    jumpeq SInoMinus[R0]    ; if not negative then goto SInoMinus
    load   R8,Minus[R0]     ; R8 := '-'
    store  R8,0[R7]         ; *p := '-'
    sub    R7,R7,R4         ; p := p - 1
SInoMinus
    add    R1,R7,R4         ; k := p + 1
    sub    R1,R1,R2         ; k := p + 1 - bufstart
SIspaceLp
```

```
    cmp    R7,R2            ; compare p, bufstart
    jumplt SIspaceLpEnd[R0] ; if p < bufstart then goto SIspaceLpEnd
    load   R8,Space[R0]     ; temp := ' '
    store  R8,0[R7]         ; *p := ' '
    sub    R7,R7,R4         ; p := p - 1
    jump SIspaceLp[R0]      ; goto SIspaceLp
SIspaceLpEnd

SIend
; return
    load   R1,2[R14]         ; save R1
    load   R2,3[R14]         ; save R2
    load   R3,4[R14]         ; save R3
    load   R4,5[R14]         ; save R4
    load   R5,6[R14]         ; save R5
    load   R6,7[R14]         ; save R6
    load   R7,8[R14]         ; save R7
    load   R8,9[R14]         ; save R8
    load   R9,10[R14]        ; save R9
    load   R10,11[R14]       ; save R10
    load   R13,1[R14]        ; save return address
    load   R14,0[R14]        ; pop stack frame
    jump   0[R13]            ; return
```

Here is what the output looks like:

```
37
Cat
(    23)
(0)
(32767)
(####)
(-1)
(#)
(-32768)
(#####)
(   32)
(   17)
(  456)
(  1066)
(-30978)
(2001)
(3)
(   47)
(   13)
(19)
(  103)
(  103)
(##)
(   47)
(    48)
```

```
(   49)
(29371)
(6285)
(   264)
(##)
(  -92)
(-1)
(###)
(42)
```