Computer Systems, Spring 2019

Week 7 Lab

Procedures

Solutions

1 Review problems

1. *(Warning! Bad code ahead! Don't use the following code as an example; let's fix it instead.)*
   The following assembly language code fragment is supposed to be equivalent to `x := 2+y` but it is poorly written. Find as many things wrong with it as you can, and indicate whether each fault is poor style or would cause incorrect results. Rewrite the code fragment as it should be.

```
      load  R1,x[R0]
  load  R2,2[R0]
      load     R3,y
add   R4,R2,R3 ; R4 := R2+R3
   store x[R0],R4
```

   Solution.

   - The following errors would result in error messages from the assembler, and the program wouldn't run:
     - In line 3, y appears without [R0]
     - In line 4, add isn't preceded by a space character (the assembler would think add is a label)
     - In the last line the register has to be specified before the variable.
   - This error would result in the program getting the wrong answer:
     - The second instruction should use lea rather than load, because its purpose is to put a constant (2) into the register. However, the instruction is syntactically correct and the computer would execute it. What the instruction *should* say is "put the variable x into R1" but what it *actually* says is "put whatever variable is at address 2 into R1". Probably some other random piece of information is at address 2, so the instruction will execute and lead to a wrong answer.
   - The following are examples of bad style. They do not prevent the program from running or getting correct results, but they do look unprofessional and make it harder to read. (Also, they would cause you to lose marks in an assessed exercise!)
     - There isn't a full-line comment explaining what the entire block of code is doing (x := 2+y).

  - The first instruction is completely pointless. The statement x := 2+y is going to *modify* the value of x, not *use* it. There's no reason to put x into a register. The only instruction that refers to x should be at the end, where we store the result into x. This instruction doesn't actually cause any harm, but it does waste execution time and it gives the impression that the programmer doesn't understand what load does.
  - Some of the individual instructions don't have comments saying what the purpose of the instruction is.
  - The comment on the add instruction isn't very helpful; it just says what the add instruction means but not *why* we're doing this instruction. In writing comments, you should assume that the reader knows the language; this isn't the place to explain what an add instruction is. It's more helpful for the comment to explain what value is being calculated; here it should be 2+y. Even an expert programmer won't have memorised what's in each register, and it's annoying to have to read back through the program to find out what's in R2 and R3. A good comment like R4 := 2+y is extremely helpful, and indicates good professional programming skill.
  - The indentation is poor: the operations, operands, and comments should be lined up neatly.

```
; x := 2+y
   lea    R2,2[R0]    ; R2 := 2
   load   R3,y[R0]    ; R3 := y
   add    R4,R2,R3    ; R4 := 2+y
   store  R4,x[R0]    ; x := 2+y
```

2. Translate each of the following instructions into machine language. Assume that the memory address of x is 00c3 and the memory address of y is 00f8.
   Quick review:

   - The format of a RRR instruction is op d a b, and opcode of add is 0, opcode of sub is 1.
   - The format of an RX instruction is two words:
     (a) op d a b where op = Hex f, d is the register operand, a is 0 (for the [R0]), and b is the code indicating which instruction: 0 for lea, 1 for load, 2 for store.
     (b) A constant, which might be specified in the instruction (e.g. 42) or might be the memory address of a variable (e.g. x).

```
add    R3,R9,R4
sub    R2,R12,R1
load   R8,x[R0]
lea    R9,42[R0]
store  R10,y[R0]
```

3. Give a reason why it's useful for the machine to guarantee that R0 always contains 0.

   Solution. This enables a program to specify the effective address of a variable x in memory as x[R0]. If the machine didn't guarantee that R0 is always 0, then you would need an extra instruction to force one of the registers to contain 0 before you could access it.
   (There is another reason which is more subtle and deeper, and it involves interrupts, which we haven't covered yet. After an interrupt, the operating system needs to be able to access its own variables, yet when it receives control after an interrupt the registers contain data belonging to the process that was interrupted. If the operating system put 0 into some register (say R5) then it would be able to access its own variables — but then it would have destroyed some of the user's data and the process would be unable to resume. On the other hand, if the operating system didn't put 0 into some register, it would be stymied — it wouldn't be able to do anything at all. The guarantee that R0 contains 0 solves these problems.)

4. Describe what the following program fragment does, and translate it to assembly language.

```
p := &x
q := p
*q := *q + 1
```

   Solution. The result is that p points to x, q also points to x, and the final statement adds 1 to x. This is written in the "statement by statement style"; it's possible to make it shorter by reusing the values of registers that contain pointers.

```
lea    R1,x[R0]    ; R1 := &x
store  R1,p[R0]    ; p := &x
add    R2,R1,R0    ; q := &x
store  R1,q[R0]    ; q := p
load   R3,0[R2]    ; R3 := *q (and *q = x)
lea    R4,1[R0]    ; R4 := 1
add    R5,R3,R4    ; R5 := *q + 1 (which is x + 1)
store  R5,0[R2]    ; *q := *q + q (adds 1 to x)
```

5. Translate the following high level code into low level form.

```
while x<y do
   { S1
     if x=p
        then { S2 }
        else for i := x to y
               { S3 }
               { S4 }
    { S5 }
{ S6 }
```

Solution.

```
Loop
   if x >= y then goto AfterLoop
   S1
; This is the if-then-else
   if x /= p then goto Else
   S2
   goto AfterIf
Else
   i := x
ForLoop
   if i > y then goto AfterFor
   S3
   i := i + 1
   goto ForLoop
AfterFor
   S4
AfterIf
; This is the end of the if-then-else
   S5
   goto Loop
AfterLoop
   S6
```

6. Suppose you forget to terminate the execution of your program with `trap R0,R0,R0`. Explain what would happen. Don't say *exactly* what will happen — that requires knowing the exact contents of memory — but explain in general terms what will happen.

   Solution. After executing the last instruction in the program, the computer will continue on to execute the next word in memory, which will be the first of the variables defined with a `data` statement. Whatever value that variable currently has, it will specify some meaningless instruction which will nevertheless be executed. It would be like giving random instructions to a killer robot—not a good idea!

   Every possible value of a 16-bit word represents an instruction, just as it also represents a binary number, and a 2's complement number, and a Unicode character, and more. Just as the computer hardware doesn't know whether `fc01` is a binary number or a two's complement number, it also doesn't know whether it happens to be the first word of a load instruction. It will simply execute the data as if it's a random instruction.

7. Suppose that you put the data statements defining your variables (and giving their initial values) at the beginning of a program, rather than at the end where they belong. Explain what would happen (just in general terms).

   Solution. The machine begins by executing the instruction in memory at address 0. If this is a variable, then its bits correspond to some instruction, which will be executed.

8. Translate the following low level code to assembly language (the operator with two vertical bars is logical or).

```
if x<y || p<q then goto L
```

Solution.

```
load   R1,x[R0]     ; R1 := x
load   R2,y[R0]     ; R2 := y
cmplt  R3,R1,R2     ; R3 := (x<y)
jumpt  R3,L[R0]     ; if x<y then goto L
load   R3,p[R0]     ; R3 := p
load   R4,q[R0]     ; R4 := q
cmplt  R5,R3,R4     ; R5 := p<q
jumpt  R5,L[R0]     ; if p<q then goto L
```

2 Study an example program: zapR13crash

You should confirm the comment made in the lab sheet: *You should find that the program works fine as long as a called function never calls another function. The program successfully stores result1. However, when there is a nested call (main calls mult6, which calls double) the first return address gets destroyed by the second call, and the program is unable to return properly. Consequently, the program is unable to store result2.* Also, you should understand why the program is behaving this way.

3 Write a program: saveR13stack

```
; Sigma16 program saveR13stack
; John O'Donnell, 2019

; See the similar program zapR13crash.  This version of the program
; pushes the return address onto a stack when a procedure is called,
; so it allows nested calls.

; Note: for simplicity, this program doesn't check for stack overflow,
; and it doesn't save or restore registers, and it doesn't provide
; local variables, and it doesn't use dynamic links.  The purpose of
; this program is to demonstrate the basics of using a stack to save
; return addresses, but a full system for calling procedures needs
; those additional features.

; There are several functions.  They all take one argument x which is
; passed in R1, and return one result f(x) which is also passed back
; in R1.  Each function multiplies x by a constant.  Some of the
```

```
; functions (double, triple, quadruple) do the work by themselves, but
; one of them (mult6) calculates 6*x by evaluating triple (double
; (x)).  It gets the right answer, and it returns successfully!

; The main program initializes the stack as follows:
;    lea    R14,CallStack[R0]  ; R14 := &stack, R14 is the stack pointer
;    store  R0,0[R14]          ; (not actually necessary)

; A function is called as follows:
;    (put the argument into R1)
;    lea    R14,1[R14]         ; advance the stack pointer to new frame
;    jal    R13,function[R0]   ; goto function, R13 := return address

; The function begins as follows:
;    store  R13,0[R14]         ; save return address on top of stack

; The function finishes and returns as follows:
;    (put the result into R1)
;    load   R13,0[R14]         ; restore return address from top of stack
;    lea    R2,1[R0]           ; R2 := size of stack frame
;    sub    R14,R14,R2         ; remove top frame from stack
;    jump   0[R13]             ; return to caller

MainProgram

; Initialize the stack pointer
    lea    R14,CallStack[R0]  ; R14 = stack pointer := &CallStack
    store  R0,0[R14]          ; (not actually necessary)

; R1 := double(2)
    lea    R1,2[R0]           ; R1 = argument := 2
    lea    R14,1[R14]         ; advance the stack pointer
    jal    R13,double[R0]     ; R1 := double(R1) = 2*2 = 4

; R1 := triple(R1)
    lea    R14,1[R14]         ; advance the stack pointer
    jal    R13,triple[R0]     ; R1 := triple(R1) = 3*4 = 12

; R1 := quadruple(R1)
    lea    R14,1[R14]         ; advance the stack pointer
    jal    R13,quadruple[R0]  ; R1 := quadruple(R1) = 4*12 = 48

    store  R1,result1[R0]     ; result1 := 4*(3*((2*2)) = 48

; This next call works because the return addresses are being saved on
; the stack.  In the previous program (zapR13crash) it doesn't work
; because a called function calls another function.

; R1 := mult6(2)
    lea    R1,2[R0]           ; R1 = x = 2
```

```
    lea    R14,1[R14]         ; advance the stack pointer
    jal    R13,mult6[R0]      ; R1 = triple(double(x)) = 3*(2*x)

    store  R1,result2[R0]     ; result2 := 6*2 = 12

    trap   R0,R0,R0           ; terminate main program

double
; receive argument x in R1
; return result in R1 = 2*x
    store  R13,0[R14]         ; save return address on top of stack
    lea    R2,2[R0]           ; R2 := 2
    mul    R1,R2,R1           ; result := 2*x
    load   R13,0[R14]         ; restore return address from top of stack
    lea    R2,1[R0]           ; R2 := size of stack frame
    sub    R14,R14,R2         ; remove top frame from stack
    jump   0[R13]             ; return R1 = 2*x

triple
; receive argument x in R1
; return result in R1 = 3*x
    store  R13,0[R14]         ; save return address on top of stack
    lea    R2,3[R0]
    mul    R1,R2,R1           ; R1 := 3*x
    load   R13,0[R14]         ; restore return address from top of stack
    lea    R2,1[R0]           ; R2 := size of stack frame
    sub    R14,R14,R2         ; remove top frame from stack
    jump   0[R13]             ; return R1 = 3*x

quadruple
; receive argument x in R1
; return result in R1 = 4*x
    store  R13,0[R14]         ; save return address on top of stack
    lea    R2,4[R0]
    mul    R1,R2,R1           ; R1 := 4*x
    load   R13,0[R14]         ; restore return address from top of stack
    lea    R2,1[R0]           ; R2 := size of stack frame
    sub    R14,R14,R2         ; remove top frame from stack
    jump   0[R13]             ; return R1 = 4*x

mult6
; receive argument x in R1
; return result in R1
; This is a common kind of function: it doesn't do much work, it
; just calls other functions to do all the real work
    store  R13,0[R14]         ; save return address on top of stack
    lea    R14,1[R14]         ; advance the stack pointer
    jal    R13,double[R0]     ; R1 := double(x) = 2*x
    lea    R14,1[R14]         ; advance the stack pointer
    jal    R13,triple[R0]     ; R1 : triple(2*x) = 3*(2*x)
```

```
    load   R13,0[R14]         ; restore return address from top of stack
    lea    R2,1[R0]           ; R2 := size of stack frame
    sub    R14,R14,R2         ; remove top frame from stack
    jump   0[R13]             ; return R1 = 3*(2*x)

result1    data  0            ; result of first sequence of calls
result2    data  0            ; result of the mult6 call
CallStack  data  0            ; The stack grows beyond this point
```

4 Extended example: recursive factorial

Step through the program and observe carefully how the call stack is maintained. Pay particular attention to the dynamic links, the return addresses, and the saved registers.