Computer Systems, Spring 2019

Week 6 Lab

Instruction Execution and Arrays

Solutions

1 Review problems

1. Explain how the effective address is calculated in an RX instruction. Give an example where the effective address can be used to (a) load a constant into a register; (b) load a variable into a register; (c) load an element of an array into a register.

   Solution. An RX instruction specifies the second operand as a memory address, using two parts: a constant displacement and a register. This may be written in assembly language as x[R2], $0a3f[R8], etc. The effective address is the sum of the displacement and the contents of the register. (a) To load 39 into R1, use lea R1,39[R0]; the effective address is 39+0 = 39, and lea loads the effective address. (b) To load x into R1, use load R1,x[R0]; the effective address is the address of x, and load fetches the contents of memory at that address. (c) To load x[i] into R1, use load R2,i[R0]; load R1,x[R1]; the effective address in the second load is the address of the array x + i, which is the address of x[i].

2. Find the effective address in each of the following instructions. Assume that R2 contains 28, R3 contains 5, and the address of x is $00a8. Assume that four-digit numbers are hexadecimal, and shorter numbers (e.g. 28) are decimal.

   - `load R6,30[R2]`
   - `load R7,40[R0]`
   - `load R8,x[R3]`

   Solution.

   - `load R6,30[R2]` 30 + 28 = 58
   - `load R7,40[R0]` 40 + 0 = 40
   - `load R8,x[R3]` 00a8 + 5 = 00ad

3. Explain the difference between the `load` and `lea` instructions.

   Solution. Both instructions—lea and load—begin by calculating the effective address (call it ea). The difference lies in what the instructions do next. The lea instruction simply puts the effective address into the destination register: reg[d] := ea. The load instruction puts the contents of the word in memory at the effective address into the destination register: reg[d] := mem[ea].

4. Explain what each of the following instructions does. Use the notation mem[a] to refer to the contents of memory at address a.

```
      load   R3,7[R0]
      lea    R4,8[R0]
      load   R5,nine[R0]
      load   R6,x[R0]
      lea    R7,x[R0]
      trap   R0,R0,R0
x     data   123
nine  data   9
```

Solution.

```
      load   R3,7[R0]      R3 := mem[7]
      lea    R4,8[R0]      R4 := 8
      load   R5,nine[R0]   R5 := mem[nine] = 9
      load   R6,x[R0]      R6 := mem[x+0] = mem[x] = 123
      lea    R7,x[R0]      R7 := x+0 = x (i.e. address of x)
      trap   R0,R0,R0      terminate
x     data   123           word containing 123 at address x
nine  data   9             word containing 9 at address nine
```

5. Explain the purpose of the following registers in the Sigma16 architecture: ir, pc, adr.

   Solution. The ir (instruction register) holds the first word of the instruction currently being executed. If the current instruction is an RRR instruction, it's in the ir; but if it is an RX instruction the first word of the instruction is in the ir. The second word of an RX instruction is loaded into the adr register, and is used to calculate the effective address (which is also kept in the adr register). The pc register contains the address of the next instruction to be executed.

2 A strange program

Consider "Program Strange" below. You're asked to execute it manually several times, with different initial values of a variable `y`. Each time, assume that all the registers contain 0 after the program is booted, before it begins execution. Explain what is happening as the program runs and give the final values of the registers. If you like you can experiment with the program on the Sigma16 application, but please work out your own answers first!

1. Hand-execute the program and give the final values of the registers.
2. Same again, but change the last line to `y data 1`.
3. Same again, but change the last line to `y data 256`.
4. Same again, but change the last line to `y data 8192`.

5. Same again, but change the last line to `y data -5424`.

6. Discuss what is going on in this program.

Note: this program doesn't do anything very useful. It's rather strange and not a model for good programming style. But it illustrates an extremely important concept. This concept has profound significance in the history of computers, the theory of computation, computer architecture, software reliability, the design of operating systems, and computer security. Be sure to read the model solution even if you solve the problem correctly. Don't feel bad if you can't work out the answers. If you do solve this problem, please accept my congratulations!

```
; Strange: A Sigma16 program that is a bit strange
        load   R1,y[R0]
        load   R2,x[R0]
        add    R2,R2,R1
        store  R2,x[R0]
        lea    R3,3[R0]
        lea    R4,4[R0]
x       add    R5,R3,R3
        add    R0,R0,R7
        trap   R0,R0,R0
y       data   0
```

Solution. The program loads an *instruction* into a register, does arithmetic on it by adding y to it, and stores the result back into memory. This phenomenon is called *self-modifying code*, and it exploits the fact that instructions and data are held in the same memory (this is the *stored program computer* concept). The original instruction is add R5,R3,R3, and its machine language code is 0533.

1. When y=0, the final values are: R1=0, R2=$0533, R3=3, R4=4, R5=6. The only notable points are that the store instruction doesn't actually change the value of the word in memory (it was 0533 and 0533 is being stored there), and the last add instruction doesn't change the value in R0 because R0 can never change; it is always 0. (Of course if R7=0 then the result of the addition is 0 anyway.)

2. When y=1, the final values are: R1=1, R2=$0534, R3=3, R4=4, R5=7. Note that R5 is *not* 3+3=6. When y=1 is added to the instruction, the result is 0534 which means add R5,R3,R4 so instead of adding R3+R3 it adds R3+R4.

3. When y=256, the final values are: R1=256=$0100, R2=$0633, R3=3, R4=4, R5=0, R6=6. The decimal number 256 is 0100 in hexadecimal. When this is added to the instruction, the result is 0633, which means add R6,R3,R3 so R3+R3 is loaded into R6, not into R5.

4. When y=8192, the final values are: R1=4096=$2000, R2=$2533, R3=3, R4=4, R5=9. The decimal number 8192 is $2000 in hexadecimal, and when this is added to the instruction the result is 2533, which means mul R5,R3,R3. It's no longer an add instruction, it's a multiply instruction that calculates R5 := R3*R3 = 9.

5. When y=-5424 the program goes into an infinite loop. R1=$ead0 (the hexadecimal representation of -5424), R2=$f003, R3=3, and R4=4. What started out as the add instruction at x has been transformed into `jump 7[R0]`, comprising the word at x (f003) and the following word (which is 0007). This jump instruction goes back to the first lea instruction, and the program runs for ever (lea, lea, jump).

6. There is a lot to say about the phenomenon of self-modifying code. Here are a few brief points, but we don't have time for a deep analysis.

   - This program shows clearly that a computer does not execute assembly language; it executes machine language. Try running it on the Sigma16 application (single step each instruction). You'll see that the assembly language statement `add R5,R3,R3` is highlighted in red, but that is just the GUI trying to be helpful. What's important is that the machine language instruction is fetched from memory and loaded into ir (the instruction register), and that is not 0533. The machine decodes the contents of ir and does whatever that says to do; it isn't aware of the assembly language statement. Indeed, a machine doesn't even understand the concept of assembly language — everything is just bits!
   - To follow exactly what is happening in the emulator, it's important to look at the pc and ir registers. These reflect what the machine is doing—the assembly language does not.
   - Early computers (late 1940s and early 1950s) did not use an effective address (i.e. displacement + index) like Sigma16; the instructions simply specified the absolute memory address of an operand. This is ok for simple variables, but how could they process arrays? The solution was to use self modifying code. In a loop that traverses an array, there would be a load instruction using address 0. In the body of the loop, there would be instructions to calculate the address of x[i] by loading the address of x and adding i; this is then stored into the address field of the load instruction. That instruction is then executed, obtaining the value of x[i]. This technique became obsolete in the early 1950s with the invention of index registers and effective addresses.
   - The pioneers of computers considered the concept of the *stored program computer* (i.e. the program and data are in the same memory) to be fundamental and essential. One of the most important reasons was that it made arrays possible. Now we consider the stored program concept to be fundamental *for different reasons.*
   - Self modifying code is tricky, and difficult to debug. It makes programs hard to read: you can't rely on what the program says, but on what its instructions will become in the future. For these reasonas, self modifying code is now considered to be bad programming practice.
   - If a program modifies itself, you can't have one copy of the program in memory and allow it to be shared by several users. For example, it's common now to have a web browser open with several tabs. Each

tab is served by an independent process (a separate running instance of a program that updates the window showing the web page). If you have 5 tabs open, there are 5 processes, each running the same machine language code, and there's only one copy of that in memory. This wouldn't work if the program modified itself!

- Self modifying code leads to security holes: if a hacker has the ability to change your machine language code in memory, they could make your own program act against you.
- Modern computers use a technique called *segmentation* that prevents a program from modifying itself. This leads to increased reliability and security.
- Some computers have a facility that allows you to gain the power of self modifying code without actually modifying the code in memory. The idea is to have an instruction execute R1,x[R0] which calculates the logical *or* of the two operands and then executes the result; x is the address of an instruction and R1 contains the modification to it. The modified instruction is executed, but there is no change to the machine code in memory. This idea was used in the IBM 360 and its successors. However, as the design of effective addresses has become more sophisticated, the execute instruction is rarely needed, and most modern computers don't provide it.

3 Study an example program: ArrayMax

This program illustrates how to traverse an array. As you study the program, notice how the high and low level algorithms are given in comments, the general style of comments, the way the control structures are translated, and how the array element is accessed.

4 Writing a complete program: DotProduct

```
; Sigma16 program DotProduct
; John O'Donnell

; We are given an integer variable n and two arrays X = x[0], x[1],
; ..., x[n-1] and Y = y[0], y[1], ..., y[n-1].  This program computes
; the dot product of X and Y, which is x[0] * y[0] + x[1] * y[1] +
; ... + x[n-1] * y[n-1].  The result is stored in p.

; Initial data values are: n = 3, x = 2, 5, 3, and y = 6, 2, 4.
; The expected result is p = 2*6 + 5*2 + 3*4 = 34.

; High level algorithm
;   p := 0
;   for i := 0 to n-1
;     p := p + x[i] * y[i]

; Low level (goto-style) algorithm
```

```
;       p := 0
;       i := 0
; loop: if not (i<n) then goto done
;       p := p + x[i] * y[i]
;       i := i + 1
;       goto loop
; done: terminate

; The low level algorithm is translated to assembly language,
; keeping the variables in registers during the loop.

; R1 = constant 1
; R2 = i
; R3 = n
; R4 = p
; R5, R6 used for temporary calculations

; Initialize the variables
     lea    R1,1[R0]      ; R1 := 1
     lea    R2,0[R0]      ; i := 0
     load   R3,n[R0]      ; R3 := n
     lea    R4,0[R0]      ; p := 0

loop
; if not (i<n) then goto done
     cmp    R2,R3         ; compare i, n
     jumpge done[R0]      ; if not (i<n) then goto done

; p := p + x[i] * y[i]
     load   R5,x[R2]      ; R5 := x[i]
     load   R6,y[R2]      ; R6 := y[i]
     mul    R5,R5,R6      ; R5 := x[i] * y[i]
     add    R4,R4,R5      ; p := p + x[i] * y[i]

; i := i + 1
     add    R2,R2,R1      ; i := i + 1

; goto loop
     jump   loop[R0]      ; goto loop

done
     store  R4,p[R0]      ; store result into p
     trap   R0,R0,R0      ; terminate

n    data   3
p    data   0
x    data   2             ; x[0]
     data   5             ; x[1]
     data   3             ; x[2]
y    data   6             ; y[0]
```

```
data   2              ; y[2]
data   4              ; y[3]
```