Assembly: Tracing; conditionals

COSC 208, Introduction to Computer Systems, 2022-03-24

Outline

- Warm-up
- Tracing assembly code
- Conditionals

Warm-up

- Q1: *Write the C code equivalent for each line of assembly, treating registers as if they were variable names. For example, the C code equivalent for* `sub sp, sp, #0x20` *is* `sp = sp - 0x20`

```
0000000000400544 <sum>:

    400544: d10083ff    sub sp, sp, #0x20

    400548: b9001fe0    str w0, [sp, #28]

    40054c: f9000be1    str x1, [sp, #16]

    400550: f9400be8    ldr x8, [sp, #16]

    400554: b9400109    ldr w9, [x8]

    400558: b9000fe9    str w9, [sp, #12]

    40055c: b9401fe9    ldr w9, [sp, #28]

    400560: b9400fea    ldr w10, [sp, #12]

    400564: 0b0a0129    add w9, w9, w10

    400568: b9000be9    str w9, [sp, #8]

    40056c: b9400be0    ldr w0, [sp, #8]

    400570: 910083ff    add sp, sp, #0x20
```

Tracing assembly code

- Q2: *The assembly corresponds to the following C code. Label each line of assembly code with the line number of the line of C code from which the assembly instruction was derived.*

```c
1  int sum(int a, int *b) {
2     int c = *b;
3     int d = a + c;
4     return d;
5  }
```

```
0000000000400544 <sum>:
    400544: d10083ff    sub sp, sp, #0x20
    400548: b9001fe0    str w0, [sp, #28]
    40054c: f9000be1    str x1, [sp, #16]
    400550: f9400be8    ldr x8, [sp, #16]
    400554: b9400109    ldr w9, [x8]
    400558: b9000fe9    str w9, [sp, #12]
    40055c: b9401fe9    ldr w9, [sp, #28]
    400560: b9400fea    ldr w10, [sp, #12]
    400564: 0b0a0129    add w9, w9, w10
    400568: b9000be9    str w9, [sp, #8]
    40056c: b9400be0    ldr w0, [sp, #8]
    400570: 910083ff    add sp, sp, #0x20
```

- Q3: *Place in the stack below the parameters `a`, `b` and local variables `c` and `d` (before executing last assembly instruction; and assuming `sp = 0xF0` initially)*

```
       0  1  2  3  4  5  6  7  8
     +--+--+--+--+--+--+--+--+--+ <- sp
0xD0 |                          |
     +--+--+--+--+--+--+--+--+--+
0xD8 |           |              |
     +--+--+--+--+--+--+--+--+--+`
0xE0 |                          |
     +--+--+--+--+--+--+--+--+--+
0xE8 |           |              |
     +--+--+--+--+--+--+--+--+--+
```

Conditionals

- Q4: *The following C code was compiled into assembly. Label each line of assembly code with the line number of the line of C code from which the assembly instruction was derived.*

```c
int divide(int numerator, int denominator) {
    int result = -1;
    result = numerator / denominator;
    return result;
}
```

```
0000000000400544 <divide>:
    400544: d10043ff    sub sp, sp, #0x10
    400548: 12800008    mov w8, #0xffffffff
    40054c: b9000fe0    str w0, [sp, #12]
    400550: b9000be1    str w1, [sp, #8]
    400554: b90007e8    str w8, [sp, #4]
    400558: b9400fe8    ldr w8, [sp, #12]
    40055c: b9400be9    ldr w9, [sp, #8]
    400560: 1ac90d08    sdiv    w8, w8, w9
    400564: b90007e8    str w8, [sp, #4]
    400568: b94007e0    ldr w0, [sp, #4]
    40056c: 910043ff    add sp, sp, #0x10
    400570: d65f03c0    ret
```

- *Why is `#0xffffffff` being stored in `w8`?*

- *When might this function cause an error?*

- The following code prevents this error.

```c
int divide_safe(int numerator, int denominator) {
    int result = -1;
    if (denominator != 0) {
        result = numerator / denominator;
    }
    return result;
}
```

Conditional assembly code

- Q5: *Its compiled assembly include a branch. Label each line of assembly code with the line number of the line of C code from which the assembly instruction was derived*

```
000000000000076c <divide_safe>:                     // Line
   76c:    d10083ff     sub    sp, sp, #0x20       //
   770:    b9000fe0     str    w0, [sp, #12]       //
   774:    b9000be1     str    w1, [sp, #8]        //
   778:    12800000     mov    w0, #0xffffffff     //
   77c:    b9001fe0     str    w0, [sp, #28]       //
   780:    b9400be0     ldr    w0, [sp, #8]        //
   784:    7100001f     cmp    w0, #0x0            //
   788:    540000a0     b.eq   79c <divide_safe+0x30> //
   78c:    b9400fe1     ldr    w1, [sp, #12]       //
   790:    b9400be0     ldr    w0, [sp, #8]        //
   794:    1ac00c20     sdiv   w0, w1, w0          //
   798:    b9001fe0     str    w0, [sp, #28]       //
   79c:    b9401fe0     ldr    w0, [sp, #28]       //
   7a0:    910083ff     add    sp, sp, #0x20       //
   7a4:    d65f03c0     ret                        //
```

- *What does the `cmp` instruction do?*

- *What does the `b.eq` instruction do?*

- *Why does the assembly check if `w0 == 0` when the C code contains `!= 0`?*

- Q6: *How would we express this in C code?* — using an if statement and a `goto` statement

Practice with conditionals

Q7: *The C code below was compiled into assembly (using `gcc`). Label each line of assembly code with the line number of the line of C code from which the assembly instruction was derived.*

```c
int flip(int bit) {
   int result = -1;
   if (bit == 0) {
        result = 1;
    }
    else {
        result = 0;
    }
    return result;
}
```

```
000000000000071c <flip>:                          // Line
    71c:    d10083ff     sub    sp, sp, #0x20       //
    720:    b9000fe0     str    w0, [sp, #12]       //
    724:    12800000     mov    w0, #0xffffffff     //
    728:    b9001fe0     str    w0, [sp, #28]       //
    72c:    b9400fe0     ldr    w0, [sp, #12]       //
    730:    7100001f     cmp    w0, #0x0            //
    734:    54000081     b.ne   744 <flip+0x28>     //
    738:    52800020     mov    w0, #0x1            //
    73c:    b9001fe0     str    w0, [sp, #28]       //
    740:    14000002     b      748 <flip+0x2c>     //
    744:    b9001fff     str    wzr, [sp, #28]      //
    748:    b9401fe0     ldr    w0, [sp, #28]       //
    74c:    910083ff     add    sp, sp, #0x20       //
    750:    d65f03c0     ret                        //
```

Q8: *Write a function called `flip_goto` that behaves the same as `flip` but matches the structure of the assembly code that will be generated for `flip`. (Hint: you'll need two `goto` statements.)*

Extra practice

Q4: *Write a function called* `adjust_goto` *that behaves the same as* `adjust` *but matches the structure of the asssembly code that will be generated for* `adjust`*. (Hint: you'll need two* `goto` *statements.)*

```
int adjust(int value) {
    if (value < 10) {
        value = value * 10;
    }
    else {
        value = value / 10;
    }
    return value;
}
```