CSE3666 — Homework 3

Mike Medved

February 21st, 2022

1 Question 5

Translate function *f* in the following C code to RISC-V assembly code. Assume function *g* has already been implemented. The constraints are:

1. Allocate register s1 to sum, and register s2 to i.
2. Save registers at the beginning of the function and restore them before the exit.
3. There are no load or store instructions in the loop. If we want to preserve values across function calls, place the value in a saved register before the loop. For example, we keep variable i in register s2.

```
// prototype of g
// the first argument of g is an address of an integer
int g(int * a, int i);

int f(int d[1024]) {
    int sum = 0;
    for (int i = 0; i < 1024; i += 1) {
        sum += g(&d[i], i); // pass d[i']s address to g
    }

    return sum;
}
```

Your code should follow the flow of the C code. Write concise comments. Clearly mark instructions for saving registers, loop control, function, restoring register, and so on.

Reminder: the callee can change any temporary and argument registers.

2 Deliverable

```
# CSE 3666 Homework 3 - Question 5
f:
    # allocate 20 bytes on the stack
    # for storing ra, s1, s2, s3, and s4.
    addi sp, sp, -20

    # store ra, s1, s2, s3, and s4 respectively
    sw   ra, 16(sp)
    sw   s4, 12(sp)
    sw   s3, 8(sp)
    sw   s2, 4(sp)
    sw   s1, 0(sp)

    addi s1, x0, 0    # initialize sum as 0
    addi s2, x0, 0    # initialize loop counter (i) as 0
    addi s3, x0, 1024 # save the loop bound (1024)
    addi s4, a0, 0    # save the base address of d

loop:
    slli a0, s2, 2    # compute offset for current index, i *= 4
    add  a0, a0, s4   # save the address of d[i] in a0 for passing to g
    addi a1, s2, 0    # save the loop counter in a1 for passing to g
    jal  ra, g        # invoke g(&d[i], i)

    # with a0 as the result of g,
    # add to the current sum and continue
    add  s1, s1, a0

    addi s2, s2, 1    # increment the loop counter, i++
    blt  s2, s3, loop # if i < 1024, rerun the loop
    addi a0, s1, 0    # save the sum in a0

    # restore ra, s1, s2, s3, and s4
    lw   ra, 16(sp)
    lw   s4, 12(sp)
    lw   s3, 8(sp)
    lw   s2, 4(sp)
    lw   s1, 0(sp)

    # move stack pointer back to start
    addi sp, sp, 20

    # return from current context
    jr ra
```

3 Question 6

Translate function msort() in the following C code to RISC-V assembly code. Assume merge() and copy() are already implemented. The array passed to msort() has at most 256 elements.

Your code should follow the flow of the C code. Write concise comments. Clearly mark instructions for saving registers, function calls, restoring register, and so on.

To make the code easier to read, we can change sp twice at the beginning of the function: once for saving registers and once for allocating memory for array c.

The function should have only one exit. There is only one return instruction.

Another reminder: the callees can change any temporary and argument registers.

```
void merge(int c[], int d1[], int n1, int d2[], int n2);
void copy(int d[], int c[], int n);

void msort(int d[], int n) {
    int c[256];
    if (n <= 1)
        return;

    int n1 = n / 2;
    msort(d, n1);
    msort(&d[n1], n - n1); // &d[n1] means the address of d[n1]
    merge(c, d, n1, &d[n1], n - n1);
    copy(d, c, n);
}
```

4 Deliverable

```
# CSE 3666 Homework 3 - Question 6
msort:
    # save ra, s1, s2
    # allocate 1036 bytes on the stack
    # to be able to store up to 256 elements
    addi sp, sp, -1036

    # store ra, s1, and s2 respectively
    # with s1 being the given array d
    # growing downwards on the stack
    # as elements are populated
    sw   ra, 1032(sp)
    sw   s2, 1028(sp)
    sw   s1, 1024(sp)

    addi s1, a0, 0    # save the given array in s1
    addi s2, a1, 0    # save the array partition in s2
    addi t0, t0, 2    # n <= 1?

    blt  a1, t0, exit # if n <= 1, exit
    addi t0, t0, 257  # store 257 in t0 to use for the n > 256 check
    bge  a1, t0, exit # if n > 256, exit

    srai a1, s2, 1    # compute n1 = n / 2 and store in a1
    jal  ra, msort    # invoke msort(d, n1)

    srai a1, s2, 1    # compute n = n / 2
    slli a0, a1, 2    # compute n1 = n - n1 and store in a0
    add  a0, a0, s1   # compute the address of d[n1]
    jal  ra, msort    # invoke msort(&d[n1], n - n1)

    addi a0, sp, 0    # save the address of c in a0
    addi a1, s1, 0    # save the address of d in a1
    srai a2, s2, 1    # compute n1 = n / 2 and store in a2
    slli a3, a2, 2    # compute n2 = n - n1 and store in a3
    add  a3, a3, s1   # compute the address of d[n1]
    sub  a4, s2, a2   # compute n2 = n - n1 again
    jal  ra, merge    # invoke merge(c, d, n1, &d[n1], n - n1)

    addi a0, s1, 0    # save the address of d in a0
    addi a1, sp, 0    # save the address of c in a1
    addi a2, s2, 0    # save the array partition in a2
    jal ra, copy      # invoke copy(d, c, n)

exit:
    # restore ra, s1, and s2
    lw ra, 1032(sp)
    lw s2, 1028(sp)
    lw s1, 1024(sp)

    # move stack pointer back to start
    addi sp, sp, 1036

    # return from current context
    jr ra
```