3666 hw 3

Casey Provitera

February 2024

1

Translate function foo() in the following C code to RISC-V assembly code. Assume function bar() has already been implemented. The constraints/tips are:

1. Allocate register s1 to count, and register s2 to i.
2. 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.
3. Identify the registers that are changed in function foo() but should be preserved. Note that the callee, bar(), may change any temporary and argument registers.
4. Save registers at the beginning of the function and restore them before the exit.

Your code should follow the flow of the C code. Write concise comments.
Clearly mark instructions for saving registers, loop, function calls, restoring register, etc.
// prototype of bar
// the first argument is an address of an integer

1. int bar(int a[], int i);
2. int foo(int d[], int n) {
3. int count = 0;
4. for (int i = 0; i < n; i += 1) {
5. int t = bar(&d[i], n - i); // &d[i] means d[i]'s address
6. if (t > 0)
7. count += 1;}

```
8.      return count;}
```

```
1.  # translate foo
2.  foo:
    # going to need to store things in the stack since there is a function call
    within a function
3.      addi s2, x0, 0 # i is set to 0
4.      addi sp, sp, -8 # move stack pointer
5.      sw a1, 0(sp) # storing n on the stack
6.      sw ra, 4(sp) # storing return address onstack
7.      loop:
    # passing n-i to a1 for bar
8.          sub a1, a1, s2
    # passing address of d[i] to a0
    # need to shift because each address of the array are 4 bytes apart since
    it it an integer array
9.          slli t1, s2, 2
10.         add a0, a0, t1
    # some functionality for bar happens here
11.         jal ra, bar
    # bar returns t in a0
    # if t> 0 we increment count
12.         bgt a0, x0, upcount
13.         beq x0, x0, skip
14.         upcount:
15.             addi s1, s1, 1
16.         skip:
17.             addi s2, s2, 1 # increment i
18.         blt s2, a1, loop # see if i is less than n if so loop
19.     add a0, s1, x0 # storing count in a0 to return
20.     lw ra, 4(sp)
21.     addi sp, sp, -4
22.     jalr x0, ra, 0 # leaving function
```

2

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 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: callees may change any temporary and argument registers.

1. void merge(int c[], int d1[], int n1, int d2[], int n2);
2. void copy(int d[], int c[], int n);
3. void msort(int d[], int n) {
4.   int c[256];
5.   if (n <= 1)
6.   return;
7.   int n1 = n / 2;
8.   msort(d, n1);
9.   msort(&d[n1], n – n1); // &d[n1] means the address of d[n1]
10.   merge(c, d, n1, &d[n1], n – n1);
11.   copy(d, c, n);}

Always explain your code with concise comments. The code should clearly show how arguments are set in each function calls.

1. # translating merge sort
   # merge sort takes an array and its length as parameters
2. msort:
   # need to allocate memory on the stack for the array c[256]
   # this is done by moving the stack pointer 4*256 since it is an int array
3.   add sp, sp, -1024 # moves stack pointer so there is now room for c
   # no need to store anything yet just need to make the memory avalible for c
   # storing n and ra in the stack
4.   addi sp, sp, -8

```
5.      sw a1, 0(sp)
6.      sw ra, 4(sp)
    # set t1 to 1 to be used in checking if n<=1
7.      addi t1, x0, 1
8.      ble a1, t1, return
    # n1 which is n/2 is now in the stack
9.      srli t2, a1, 1
10.     addi sp, sp, -4
11.     sw t2, 0(sp)
    # re-call of msort with normal array and half n
    # this means the call will work the same as only passing half the array
12.     add a1, t2, x0
13.     jal ra, msort
    # after first mergsort call we need to call merge sort on the other half of
    the list
    # this means calling at the middle address with n-n1
14.     lw t1, 0(sp) # loading n1 from the stack
15.     lw a1, 4(sp) # loading n from the stack
16.     sub a1, a1, t1 # a1 is now n-n1
17.     slli t1, t1, 2
    # this line is not great
18.     add a0, a0, t1 # a0 is now the address of d[n1]
19.     addi sp, sp, -8
20.     sw a0, 0(sp) # address of d[n1] on the stack
21.     sw a1, 4(sp) # n-n1 in the stack
    # recall msort with the second half of the list
22.     jal ra, msort
23.     lw a4, 0(sp) # address of d[n1]
24.     lw a3, 4(sp) # n-n1
25.     lw a2, 8(sp) # n1
```

26. add sp, sp, 12 # move stack pointer back now it is pointing at n
# its not my favorite thing to us a0 here since i feel like it could have been changed
# but I am not sure how else to get the array d to be a function parameter for merge and copy
# this would mean we do not need to change a0 we just need to move it to a1

27. add a1, a0, x0

28. lw a0, 4(sp) # this is getting the address of c

29. jal ra, merge

30. lw a2, 0(sp) # loading n

31. lw a1, 4(sp) # loading address of c

32. jal ra, copy

33. return:
# do i need to pull from the stack here
# probably need to pull ra how

34. ra, 0(sp)

35. jalr x0, ra, 0

3

Find the machine code for the following instructions. Assume all instructions are labeled sequentially, for example, I1, I2, I3, . . . , I150.
. . .
I10 : BGE x10, x20, I100
I11 : BEQ x10, x0, I1
. . .
I140 : JAL x0, I100
Starting address for I1 is 0x00400000 and each following instruction increments by 4. This means to find the instruction location of any instruction you subtract 1 from the instruction number and then multiple by 4 then add the starting instruction location of 0x00400000.

Location of I10 : BGE x10, x20, I100 is: 0x00400000 + hex of (4*(10-1)) = hex of 36 = 0x24 -> 0x00400024
bge is a SB type instruction with the opcode of 1100011 and function 3 of 101
x10 is rs1 and has the binary of 01010, x20 is rs2 and has the binary of 10100
imm also known as the offset is equal to the target location - the current location
target location is I100 = 0x00400000 + hex of (4*(100-1)) = hex of 396 = 0x18C

-> 0x0040018C
target - current = 0x0040018C - 0x00400024 = 0x00000168 convert to binary 0001 0110 1000 we now take the final trailing zero away since all addresses need to be positive so the final imm is 0000 1011 0100
machine code in binary =
0001011 10100 01010 101 01000 1100011
grouped in bytes -> 0001 0111 0100 0101 0101 0100 0110 0011
machine code in hex = 0x17455463

Location of I11 : BEQ x10, x0, I1 is: 0x00400000 + hex of (4*(11-1)) = hex of 40 = 0x28 -> 0x00400028
beq is a SB type instruction with the opcode of 1100011 and function 3 of 000
x10 is rs1 and has the binary of 01010, x0 is rs2 and has the binary of 00000
imm also known as the offset is equal to the target location - the current location
target location is I1 = 0x00400000 + hex of (4*(1-1)) = hex of 0 = 0x0 -> 0x00400000
target - current = 0x00400000 - 0x00400028 = -0x28 convert to binary 1111 1101 1000 we now take the final trailing zero away since all addresses need to be positive so the final imm is 1111 1110 1100
machine code in binary =
1111110 00000 01010 000 11001 1100011
grouped in bytes -> 1111 1100 0000 0101 0000 1100 1110 0011
machine code in hex = 0xFC050CE3

Location of I140 : JAL x0, I100 is: 0x00400000 + hex of (4*(140-1)) = hex of 556 = 0x22C -> 0x0400022C
jal is a UJ type instruction with the opcode of 1101111.
x0 is rd and has the binary of 00000
imm also known as the offset is equal to the target location - the current location
target location is I100 = 0x00400000 + hex of (4*(100-1)) = hex of 396 = 0x18C -> 0x0040018C
target - current = 0x0040018C - 0x0040022C = -0xA0 in decimal -160
To get to the binary representation of -160 we will first find the binary representation of 160 = binary 0000 0000 0000 1010 0000 this is in 20 bit representation since that is the size of the imm in a jal call.
Now convert to -160 but flipping all the bits and adding 1.
1111 1111 1111 0101 1111 + 1 = 1111 1111 1111 0110 0000
now we can get rid of the final trailing 0 since all addresses are even
-> 1 11111111 1 1110110000 split into sections for machine code
1111 0110 0001 1111 1111 0000 0110 1111 binary machine code
hex machine code 0xF61FF06F

4

Decode the following instructions in machine code. Find the offset in decimal and then the target address in hexadecimal. The hexadecimal number before the colon is the instruction's address.
0x0400366C: 0xDB5A04E3
To Binary - 0b 1101 1011 0101 1010 0000 0100 1110 0011
Opcode of 1100011 denotes as a SB type call funct 3 of 000 means it is beq.
imm for SB type is in bits 31:25 as [12][10:5] and bits 11:7 as [4:1][11].
imm is 1110 1101 0100 convert to 2s compliment 0001 0010 1100 = -600 this is the imm in decimal.
Target address is current address plus the offset first convert -600 to hex, 0x258 add to current address of 0x0400366C since the offset is negative we subtract from the current address.
0x0400366C - 0x00000258 = 0x04003414
0x04208888: 0xFA9FF0EF this is the target destination.
To Binary - 0b 1111 1010 1001 1111 1111 0000 1110 1111
Opcode of 1101111 denotes as the UJ call of jal.
imm for UJ is stored in bits 31:12 as [20][10:1][11][19:12]
bits 0b 1111 1010 1001 1111 1111 need to be reordered after reordering 0b 1111 1111 1111 1101 0100
convert to 2s compliment (ignoring leading 0s) 101100, we need to multiple by 2 (same as adding a 0 to the end) since the last bit is left out by the compiler since all locations need to be even 01011000 = -88 this is the imm as decimal.
Target address is current address plus imm. Convert -88 to hex, 0x58. But since imm is negative we subtract it from the current address.
0x04008888 - 0x00000058 = 0x04008830