Compiler Construction: COM 5202
Tutorial 1

Pratyay Sarkar

January 2025

0.1 Question 1:

How do you make a function name non-global?

We can remove the following line from the assembly code, or optionally, we can mark the ‘main‘ as local.

```
# .globl main
.local main
```

0.2 Question 2:

What is the purpose of the instruction cltq (convert long to quad)?

‘cltq’ in assembly stands for Convert Long to Quad. It is used for extending ‘eax’ which is 32-bit to ‘rax’ which is 64-bit. Suppose we have eax(32-bit signed value) = $0xFFFFFFFF$ in the case if we want to extend it to ‘rax’, ‘cltq’ extends it and ‘rax’ now stores $0xFFFFFFFFFFFFFFFF$.

0.3 Question 3:

What is the displacement of a[5] with respect to ‘rbp’?

The displacement of a[5] with respect to rbp is -28 bytes.

0.4 Question 4:

How do you modify the assembly language code to print a[5] instead of a[9]?

We need to change the following lines:

```
movl $5, %eax           # Load the index 5
cltq                    # Sign-extend to 64 bits
movl -28(%rbp), %eax # eax <-- Mem[rbp-12](a[9])
```

0.5 Question 5:

How do you modify the assembly language code corresponding to a[i] = 3*a[i-1]+b; by adding one more instruction to compute a[i] = 5*a[i-1]+b;?

We can just add another ‘addl %eax, %eax # eax ¡– eax+eax (4a[i-1])’ and we can get 5a[i-1]. Here is the code below:

```
# edx <-- a[i-1]
movl %edx, %eax # eax <-- edx (a[i-1])
addl %eax, %eax # eax <-- eax+eax (2a[i-1])
addl %eax, %eax # eax <-- eax+eax (4a[i-1])
addl %eax, %edx # edx <-- edx (4a[i-1]) + eax (a[i-1])
# rdx has 5*a[i-1]
```

0.6 Question 6:

Explain the following pair of instructions:

```
cmpl $9, -52(%rbp)
jle .L3
```

0.7 Question 7:

The test ($i \leq 9$) is at the end of the body of the for-loop in the assembly language code. Modify the code to bring it to the beginning of the body of for-loop.

- We have to take the following snippet of code and place it before '.L3'

```
.L2:
    # if i <= 9 loop
    cmpl $9, -52(%rbp) #
    jle .L3
```

- In the place of '.L2' we have to write '.L4'.

```
.L4:
    cmpl $9, -52(%rbp) #
    jg .L4
    movl -12(%rbp), %eax # eax <-- Mem[rbp-12](a[9])
    movl %eax, %esi # esi <-- eax
    ...
```

- In '.L3' we have to do the following changes.

```
...
    cltq # rax <-- i
    movl %edx, -48(%rbp,%rax,4) # Mem[rbp-48+4*rax] <--
        edx
    # a[i] <-- 3*a[i-1] + b
    # i++
    addl $1, -52(%rbp) # Mem[rbp-52] <-- Mem[rbp-52]+1
    # i <-- i+1
    jmp .L2
```

0.8 Question 8:

Explain the assembly code for scanf("%d", &b); Assembly snippet for scanf:

```
# scanf("%d", &b);
leaq -56(%rbp), %rax #
movq %rax, %rsi #
#
leaq .LC1(%rip), %rdi #
```

```
#
#
movl $0, %eax # eas <-- 0
call __isoc99_scanf@PLT # call to scanf
```

- The first line 'leaq -56(%rbp), %rax' computes the address of b athe location -56(%rbp) and stores it in the register %rax,
- Second line 'movq %rax, %rsi' moves the address of b(which is stored in %rax) to %rsi.
- Third line 'leaq .LC1(%rip), %rdi' loads the address of the format string "%d" (stored at .LC1) into %rdi.
- 'movl $0, %eax' sets the value 0 to the register eax.
- 'call __isoc99_scanf@PLT' finally makes the system call for the function scanf using its entry point in the Procedure Linkage Table (PLT).

0.9 Question 9:

On my computer the objdump of the a.out file shows the following code at the virtual memory location 00 00 06 d9 to 00 00 06 dd corresponding to the call of scanf().

```
6d9: e8 a2 fe ff ff callq 580 <__isoc99_scanf@plt>
6de: next instruction
```

The next instruction starts from the address '00 00 06 de'. 'e8' is the op-code for 'callq'. 'a2 fe ff ff' specifies the PC-relative address of the call-location. Corresponding assembly code shows 580. How do you relate '580' with 'a2 fe ff ff'?

In this section the 'e8' is the op-code for the function 'callq' and then the next 4 bytes offset specifies pc relative address. The 4 byte offset 'a2 fe ff ff' is in the little-endian format (least significant byte first). Now when we convert the offset to big-endian format we get 'ff ff fe a2' and the corresponding hex is, '0xfffffea2'. Now this hex value is a 32-bit signed value, which corresponds to -350 in decimal 2's component. Now the effective address is calculated by adding the offset with the address of the next instruction.

$$TargetAddress = AddressofNextInstruction + Offset \tag{0.1}$$

As the next address is $0x6de$ we subtract $15e$(which is 350 in decimal) to it and get 244 in hex which is 580 in decimal.