C to ASM pt. 3 (Control Structures)

Intro to Computer Systems, Fall 2022

Instructor: Travis McGaha

TAs:

Ali Krema	Andrew Rigas	Anisha Bhatia
Audrey Yang	Craig Lee	Daniel Duan
David LuoZhang	Eddy Yang	Ernest Ng
Heyi Liu	Janavi Chadha	Jason Hom
Katherine Wang	Kyrie Dowling	Mohamed Abaker
Noam Elul	Patricia Agnes	Patrick Kehinde Jr.
Ria Sharma	Sarah Luthra	Sofia Mouchtaris

Upcoming Due Dates

- HW10/11 (J compiler) to be released soon
  - (Hopefully tonight)
  - Should have everything you need
  - HW10 & 11 make up a 2-part assignment that take a while to complete.
  - Recitation for this assignment has been VERY helpful

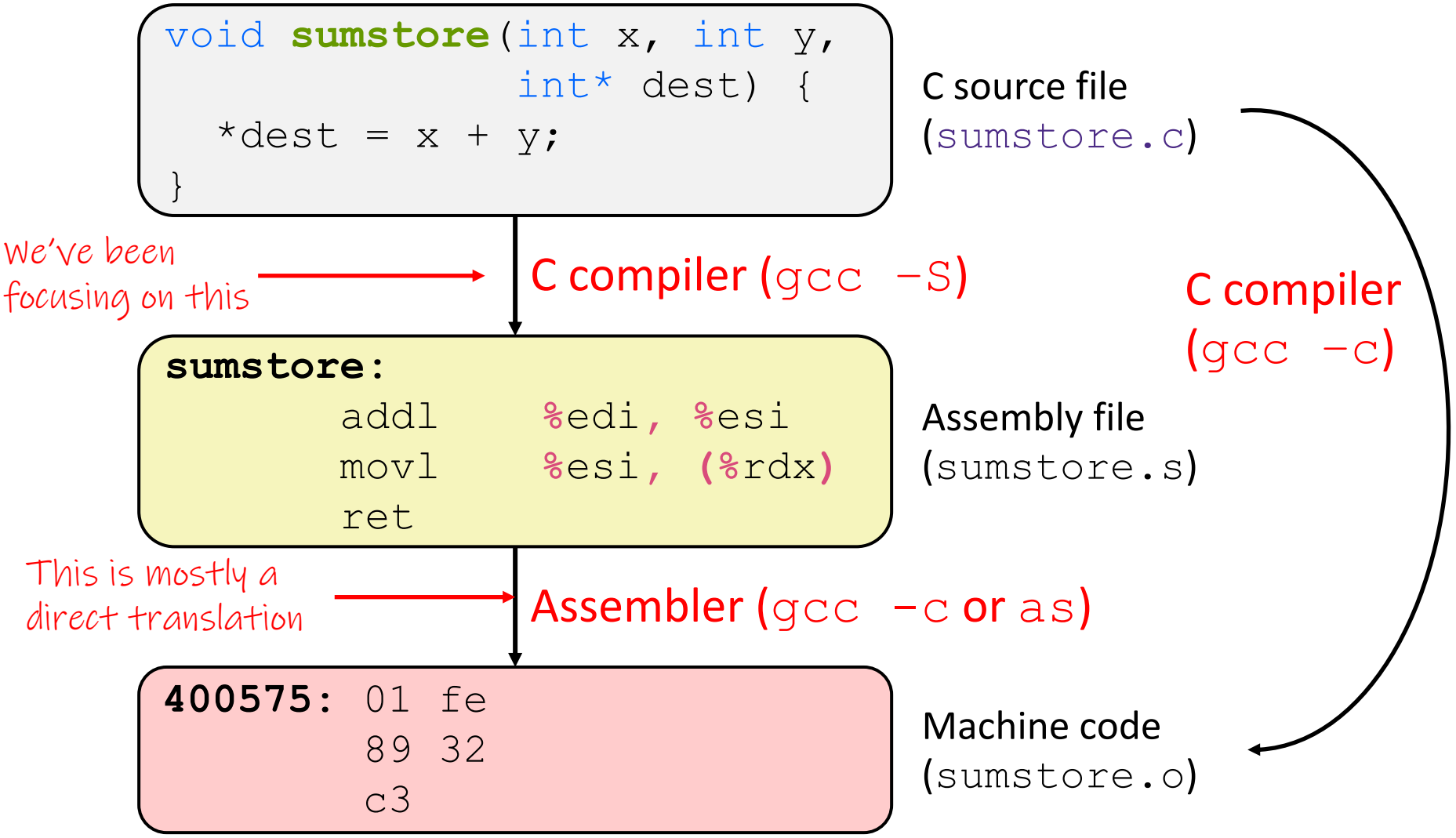
- Midterm regrade requests
  - Close tomorrow at 11:59 pm the next Tuesday (11/22)
  - Please look at the sample solution before submitting a regrade request
- Check-in 9 “the make-up” check-in
  due Monday 11/28 @ 4:59 pm before lecture.

Lecture Outline

- **Compilation Process**
- Control Structures

Important for Hw10 & Hw11

C to Machine Code



Compile Time vs Runtime

- Compile time: translates input code to Assembly
  - LC4 instructions & pseudo instructions
  - LC4 directives for assembler to setup memory
  - Labels, Function epilogue/prolouges
  - The output is an ASM file (THAT IS IT)

- Run time: runs the assembly code output
  - Executes instructions
  - Performs computation specified by the program

- Common conceptual error for HW10/11 is mixing up the two

Lecture Outline

- Compilation Process
- **Control Structures**

Important for Hw10 & Hw11

LC4 Review: If & Loops in LC4

- Not all programming constructs have direct LC4 instructions

- How would we implement

```
if (R0 >= 3)
   R1 = R0;
```

```
START
        CMPI R0, #3
        BRn AFTER_IF
        ADD R1, R0, #0
AFTER_IF
        ; ...
```

LC4 Review: If & Loops in LC4

- Not all programming constructs have direct LC4 instructions

- How would we implement

```
if (R0 != R2) {
   R1 = R2;
} else {
   R1 = 0;
}
```

```
START
        CMP R0, R2
        BRz ELSE
        ADD R1, R2, #0
        JMP AFTER
ELSE    CONST R1, #0
AFTER
        ; ...
```

LC4 Review: If & Loops in LC4

- Not all programming constructs have direct LC4 instructions

- How would we implement

```
while (R0 != 2) {
   // ...
}
```

```
START_LOOP
        CMPI R0, #2
        BRz AFTER_LOOP
        ; ...
        JMP START_LOOP
AFTER_LOOP
        ; ...
```

LC4 Review: If & Loops in LC4

- Not all programming constructs have direct LC4 instructions

- How would we implement

```
for (R0 = 0; R0 < R6; R0++) {
   // ...
}
```

```
        CONST R0, #0
START_LOOP
        CMP R0, R6
        BRzp AFTER_LOOP
        ; ...
        ADD R0, R0, #1
        JMP START_LOOP
AFTER_LOOP
        ; ...
```

Note On Labels

- When you are writing LC4 assembly, the labels you use must be unique.
  - If you use the same label more than once, the assembler will not know which location you are referring to with JMP <LABEL>

- To avoid name conflicts, it is common to number the labels or give more specific names.
- Instead of just using **`LOOP`**
  - **`LOOP_1`**
  - **`LOOP_2`**
  - **`SUM_NUM_LOOP`**
  - etc.

Compiler Pseudo Code

- The compiler you create for HW10/11 will read tokens sequentially from a file and generate the output asm based on those tokens

NOTE: this is PSEUDO CODE, lots of details missing and you don't have to follow this in HW10/11

```
void gen_asm(token t) {
  if (token.type == PLUS) {
     // generate assembly for addition
  } else if (token.type == AND) {
     // generate assembly for AND operation
  } else if (token.type == DEFUN) {
     // generate assembly for the start of
     // DEfining a FUNction
  } else if (token.type == IF) {
     // generate assembly for the start of an IF
  } else if ...
}
```

Compiling an If Statement

- How do we generate the ASM for an IF statement?

Input J pseudo code

```
if
   // ...
else
   // ...
endif
```

Compiler pseudo code

```
void gen_asm(token t) {
  // ...
  } else if (token.type == IF) {
     // generate assembly for the start of an IF
  } else if (token.type == ELSE) {
     // generate assembly for the ELSE
  } else if (token.type == ENDIF) {
     // generate assembly for the end of the IF
  }
}
```

Compiling an If Statement

- How do we generate the ASM for an IF statement?
- How do we generate this while going Sequentially one token at a time?

Input J pseudo code

```
if
   // ...
else
   // ...
endif
```

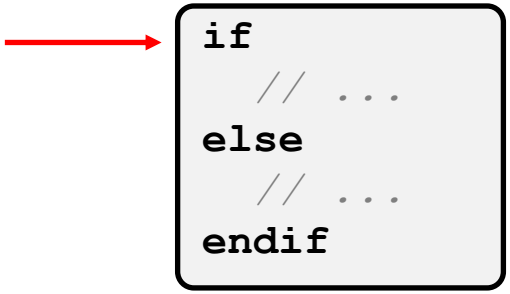
Sample Output:

```
        CMP ...
        BRz ELSE_N
        ; ...
        JMP AFTER_N
ELSE_N
        ; ...
AFTER_N
```

Compiling an If Statement

- How do we generate the ASM for an IF statement?

Input J pseudo code



```
if
   // ...
else
   // ...
endif
```

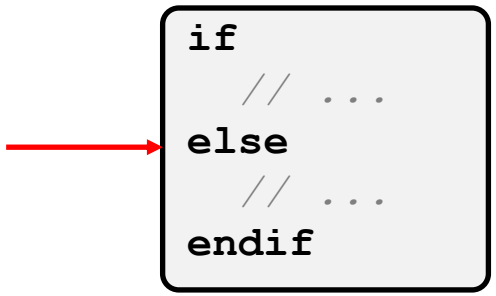
Sample Output:

```
CMP ...
BRz ELSE_N
```

Compiling an If Statement

- How do we generate the ASM for an IF statement?

Input J pseudo code



```
if
   // ...
else
   // ...
endif
```

- We could continue, but this assumes that we will eventually read an ELSE token

Sample Output:

```
        CMP ...
        BRz ELSE_N
        ; ...
        JMP AFTER_N
ELSE_N
```

- What if there isn't an else statement?

```
if
   // ...
endif
```

This lecture is not trying to give you the answer to how to generate code, but is trying to point out things you should think about before/while writing your compiler

Nested Control Structures

- Control structures can have other nested control structures. These can be if/else/endif and loops

```
if
  if
    // ...
  endif
  // ...
else
  // ...
endif
```

- How do we handle these?
- Two common options:
  - Use recursion! E.g. when an IF token is identified, generate the ASM you can generate, then recursively call to generate the body of the IF Branch or other control structure bodies.
  - Maintain a Deque or Stack data structure to keep track of which "level of nestedness" you are currently in. When you enter a control structure push some data into the structure, when you leave, pop some data

Many Translations

Your compiler can translate things to ASM however it likes, as long as it works

- There are many ways to translate C code to ASM. Consider the following C code and two different LC4 translations.

```
for (B = 0; B > 0; B--) {
  C += A;
}
```

```
        CONST R1, #0
START_LOOP
        CMPI R1, #0
        BRnz AFTER_LOOP
        ADD R2, R2, R0
        ADD R1, R1, #-1
        JMP START_LOOP
AFTER_LOOP
        ; ...
```

```
        CONST R1, #0
        JMP TEST
BODY
        ADD R2, R2, R0
        ADD R1, R1, #-1
TEST
        CMPI R1, #0
        BRp BODY
AFTER_LOOP
        ; ...
```

Poll Everywhere

pollev.com/tqm

- Which of the two implementations of a for loop will generally take less instructions to execute

A.

```
        CONST R1, #0
START_LOOP
        CMPI R1, #0
        BRnz AFTER_LOOP
        ADD R2, R2, R0
        ADD R1, R1, #-1
        JMP START_LOOP
AFTER_LOOP
        ; ...
```

B.

```
        CONST R1, #0
        JMP TEST
BODY
        ADD R2, R2, R0
        ADD R1, R1, #-1
TEST
        CMPI R1, #0
        BRp BODY
AFTER_LOOP
        ; ...
```

C. We're lost..

Performance

There will likely be a question related to this on the final. There have been these on old finals

- LCC uses the more efficient loop ASM translation.
  - More Instructions -> More Clock Cycles -> More Time

- Modern Compilers perform many operations to squeeze out as much performance as possible. To make your code run as fast as possible
  - Keeping arguments/variables in registers avoids having to load and store memory (which takes more instructions)

  - **`inline`** functions in C: instead of dealing with the overhead of calling a function, put the body of this function's code directly in the body of the caller.