CPSC-402 Report
Compiler Construction

Scott Fitzpatrick
Chapman University

May 18, 2022

Abstract

Contents

1 Introduction

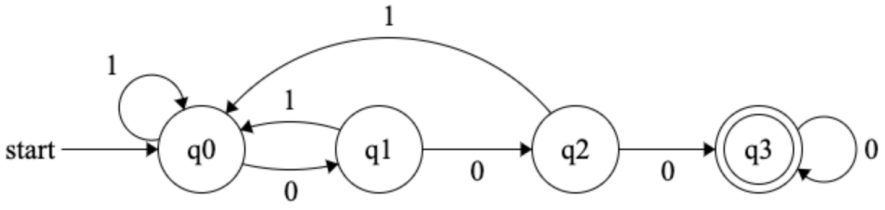
This report contains the homework from Chapman University's 2022 Compiler Construction course (CPSC 402) and a project that studies the gcc compiler by comparing it to C++ code and WebAssembly.

2 Homework

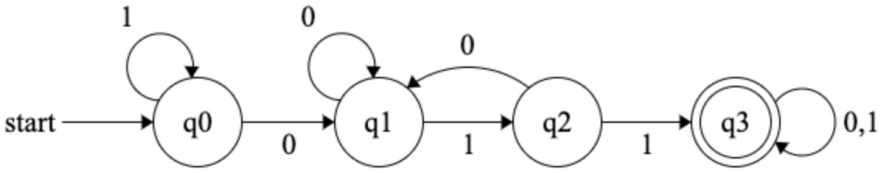
2.1 Week 1

Exercise 2.2.4: Give DFA's accepting the following languages over the alphabet $\{0, 1\}$:

b) The set of all strings with three consecutive 0's (not necessarily at the end)



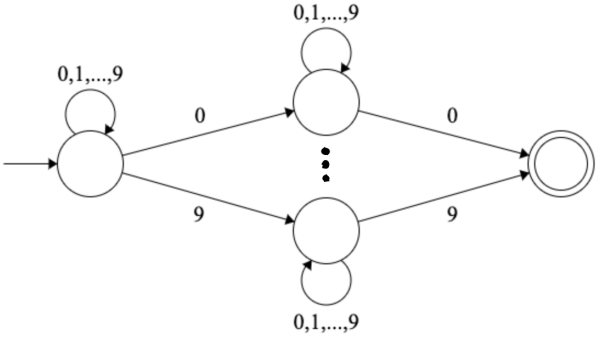
c) The set of strings with 011 as a substring



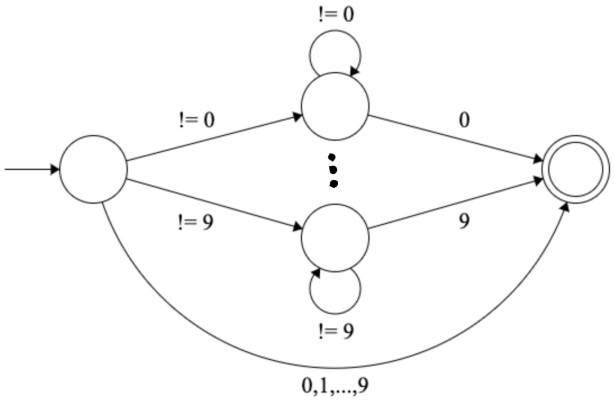
2.2 Week 2

Exercise 2.3.4:

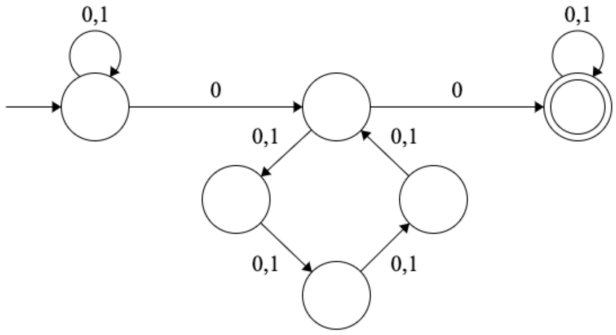
a) The set of strings over alphabet $\{0, 1, ..., 9\}$ such that the final digit has appeared before.



b) The set of strings over alphabet $\{0, 1, ..., 9\}$ such that the final digit has not appeared before.

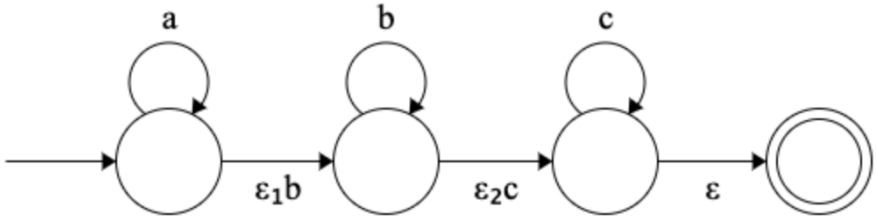


c) The set of strings of 0's and 1's such that there are two 0's separated by a number of positions that is a number of 4. Note that 0 is an allowable multiple of 4.

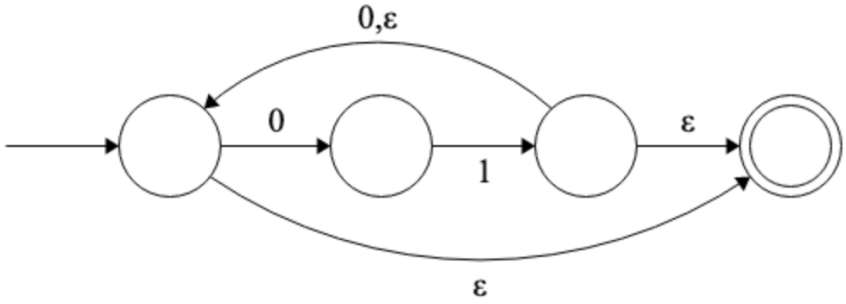


Exercise 2.5.3:

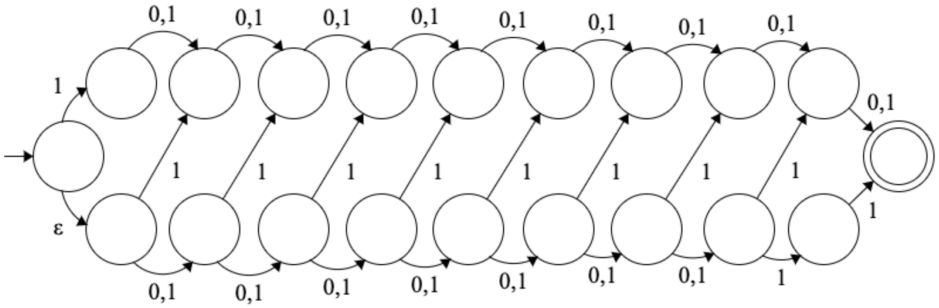
a) The set of strings consisting of zero or more a's followed by zero or more b's, followed by zero or c's.



b) The set of strings that consist of either 01 repeated one or more times or 010 repeated one or more times.



c) The set of strings of 0's and 1's such that at least one of the last ten positions is a 1.



2.3 Week 3

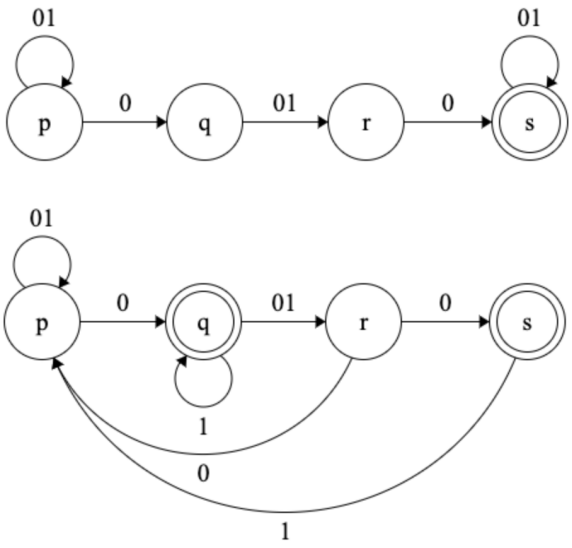
Exercise 2.3.1 : Convert to a DFA the following NFA:

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\emptyset$
$*s$	$\{s\}$	$\{s\}$

Exercise 2.3.2 : Convert to a DFA the following NFA:

	0	1
$\rightarrow p$	$\{q, s\}$	$\{q\}$
$*q$	$\{r\}$	$\{q, r\}$
r	$\{s\}$	$\{p\}$
$*s$	$\emptyset$	$\{p\}$

The following DFA's above can be drawn as:



In order to convert a dfa to an nfa, there can be only one final state. The initial state can remain the same, but the automata must result in one final state every time. Therefore, "final" should not return a set but instead a state.

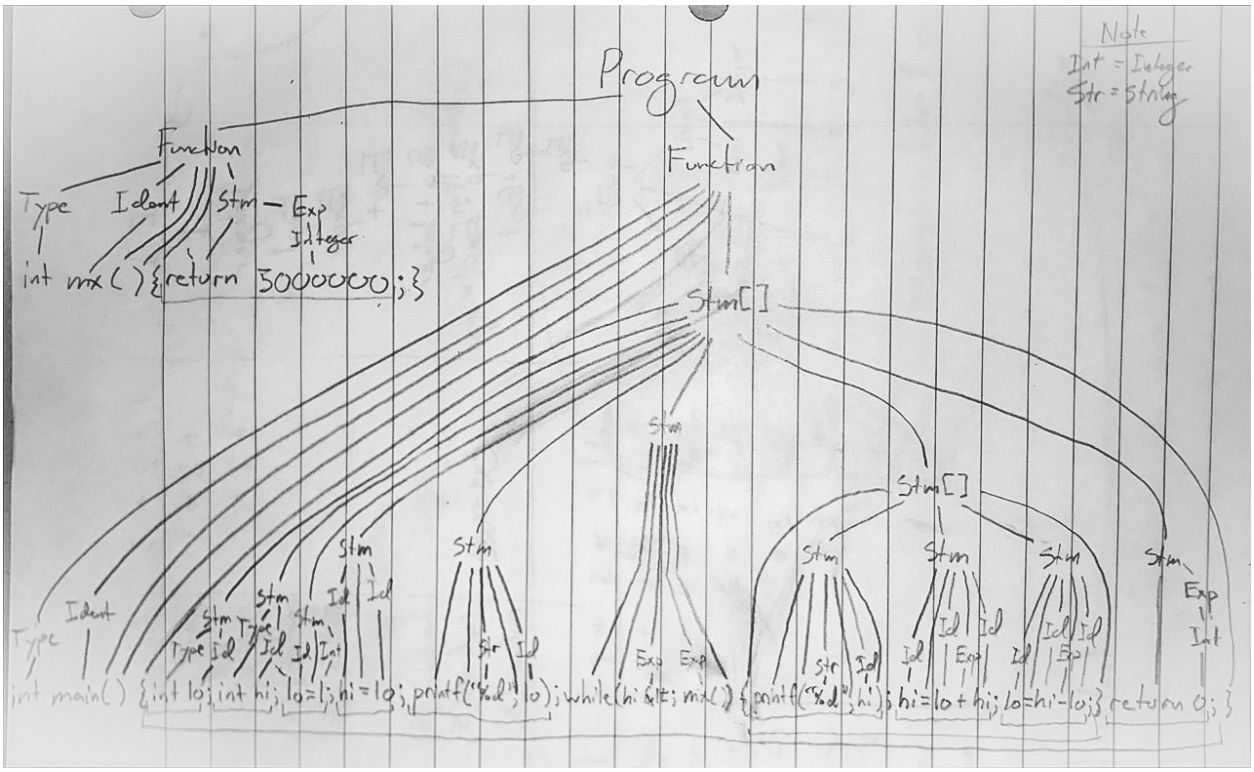
```
-- convert an NFA to a DFA
nfa2dfa :: NFA s -> DFA [s]
nfa2dfa nfa = DFA {
-- exercise: correct the next three definitions
dfa_initial = [nfa_initial nfa],
dfa_final = let final qs = disjunction (map(nfa_final nfa) qs) in final,
dfa_delta = let
    f [] c = []
    f (q:qs) c = concat [nfa_delta nfa q c, f qs c] in f }
```

2.4 Week 4

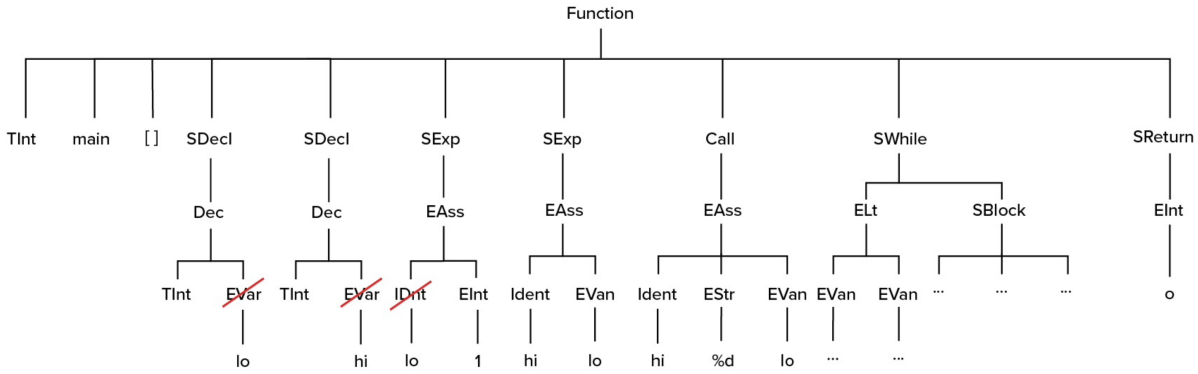
```
// a fibonacci function showing most features of the C-- language

int mx () {
    return 5000000 ;
}

int main () {
    int lo ;
    int hi ;
    lo = 1 ;
    hi = lo ;
    printf("%d",lo) ;
    while (hi &lt; mx()) {
        printf("%d",hi) ;
        hi = lo + hi ;
        lo = hi - lo ;
    }
```

```
    return 0 ;
}
```

This is the parse tree (concrete syntax tree) for the C−− code listed above:



Here is the respective abstract syntax tree:



3 Project

This portion of the report will explain compiled code from WebAssembly and the gcc (x86-64 12.1) compiler in depth. We will cover topics such as the stack, abstract syntax, and the key differences between WebAssembly and Assembly These example programs will be compiled and explained in detail using the same workflow used in Assignment 4 and this compiler explorer.

3.1 Introduction

We will take simple C++ programs and compile them to binary. Each program increases in complexity and contain simple math operations. The binary formats will be WebAssembly and x86 assembly.

3.2 easy_mult.cc

To start simple, we will start with this C++ code:

```
int main () {
    return 2*3;
}
```

This program returns $2 * 3$ within the main program. This may seem quite obvious, especially from the perspective of an experience programmer who does not usually have to think about the compilation process. However, much is actually going on underneath the hood in order to go from C++ to binary. Using node.js and the WebAssembly Binary Toolkit, we will generate a `.wat` file which contains the respective WebAssembly to our `easy_mult.cc` program. This generated `.wat` file looks like this:

```
(module
    (func
        $main             ; function name
        (result i32)      ; return type (32-bit integer)
        (i32.const 2)     ; 2
        (i32.const 3)     ; 3
        i32.mul           ; *
        return            ; return
    )
    (export "main" (func $main))
)
```

This may seem just slightly overwhelming at first, so let's quickly cover some of the basic code. The export line at the bottom will export our main function which is defined above it. The main function is denoted as: `\$main`. `result$ $i32` means that the function will return a 32-bit integer. Now we can focus on the logical code within the main function which is:

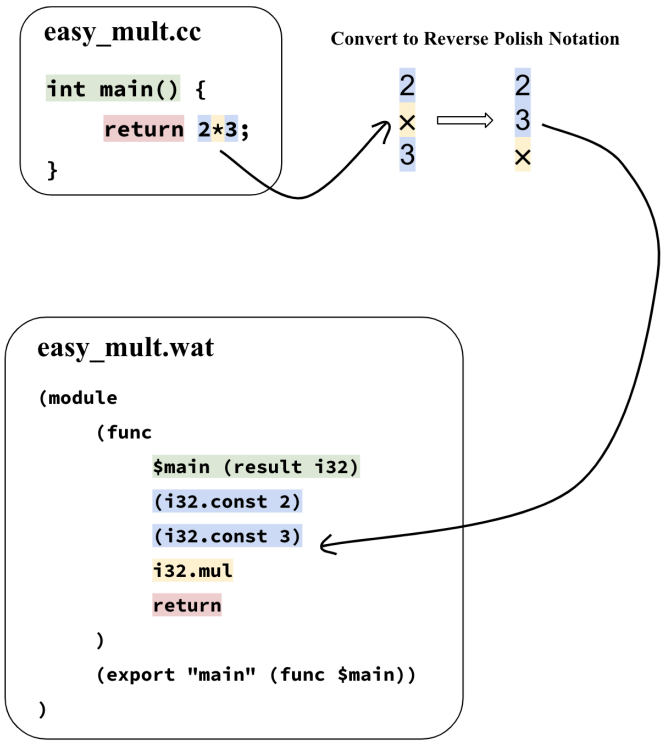
```
        (i32.const 2)
        (i32.const 3)
        i32.mul
```

First, let's quickly review some simple syntax. In WebAssembly, the `i32.const` mnemonic holds an integer value of 32 bits. Similarly, the `i32.mul` mnemonic represents multiplication for integers. For the sake of simplicity, we can view them this way for now:

```
        2
        3
        *
```

As you have noticed, this represents $2*3$. You may be wondering why these three WebAssembly lines are out of order though. If these lines were written chronologically as a math equation, it would appear to be: (2 3 $*$). This is because a stack is used to compile and execute math equations in C++. So math operations are

performed by pushing and popping values like these onto the stack. In order to properly push mathematical numbers and operators onto the stack, reverse polish notation (RPN) must be used. After the equation is fully pushed, it can be calculated as it is popped off the stack. Here is a visual of the RPN conversion and how it applies in the WebAssembly file:



In the next example, we will view a more complex equation. For now, we will compare the WebAssembly to the assembly complied by the gcc compiler. With `easy_mult.cc` inserted into this compiler explorer with x86-64 12.1 selected, we get:

```
main:
    push    rbp         ; push val of register onto stack
    mov     rbp, rsp    ; copy stack pointer val to base pointer
    mov     eax, 6      ; copy 6 to the eax register
    pop     rbp         ; cleanup
    ret                 ; cleanup
```

There is much to digest here. When reading assembly, it is crucial to think about the stack and how each line effects it. Let's start by looking at `push rbp`. This instruction pushes the value of the register, `rbp` onto the stack. `rbp` is the memory address of the base of the previous stack frame. This action will cause the value `rsp` to be the memory address of the top of the stack. This next step is to copy the value of the stack pointer, `rsp`, to the base pointer, `rbp`. This is done with `mov rbp, rsp` Now, `rbp` and `rsp` point to the top of the stack. Now we must copy the answer, 6, to the `eax` register with: `mov eax, 6`. Finally, the last two lines clean up the stack for it's next usage [AC].

The Differences Between WebAssembly and Assembly

It is also important to consider the differences between WebAssembly and assembly. Assembly language is a low-level programming language that bridges programming languages such as C++ to the computer hardware [AL]. When compiling with gcc, we obtain a binary file. The file appears as assembly language which allows us to understand what is actually going on in the stack and how the program is actually working in terms of interacting with the computer hardware. The literal binary (1's and 0's) can also be written out by referencing the opcodes. WebAssembly is like a low-level assembly language in the aspect that it is a compiled binary file. However, it is built to run in modern web-browsers. It provides a way to run code written in a variety of languages on the web at near native speed. This allows users to run client applications on the web [WA]. There are clearly some noticeable differences in syntax between assembly language and WebAssembly which we will go over in these three examples. You may notice that WebAssembly is much easier to read and this is because it is not specific to a certain type of processor, unlike the assembly language code that is generated with gcc in this report. We will continue to look at more assembly code when we compare easy_mult.cc to a modified and more advanced version.

3.3 easy_mult_var.cc

Now lets look at this code. It is the same as easy_mult.cc, but it has a different equation and uses an integer variable, `foobar`, to store it.

```
int main () {
    int foobar = (2+3)*2;
    return foobar;
}
```

The respective WebAssembly code is:

```
(module
    (func
        $main
        (result i32)
        (local $ifoobar$0 i32)       ; define local variable integer
        (i32.const 2)                ; 2
        (i32.const 3)                ; 3
        i32.add                      ; +
        (i32.const 2)                ; 2
        i32.mul                      ; *
        (local.set $ifoobar$0)       ; set local variable integer
        (local.get $ifoobar$0)       ; get local variable integer
        return
    )
    (export "main" (func $main))
)
```

To keep things simple, we will again start by focusing on the equation.

```
        ...
        (i32.const 2)                ; 2
        (i32.const 3)                ; 3
        i32.add                      ; +
        (i32.const 2)                ; 2
        i32.mul                      ; *
        ...
```

As you can see, the RPN is the very similar to the previous C++ example. However, now it is important to consider the parentheses. Below, we can view the conversion from the original equation to the RPN equation. We can also see how the RPN equation eventually leads to the answer as values are popped from the stack from top down.

```
(2                2
 +                3      5
 3)   -- to -->   +  =   2  =  10
 *                2      *
 2                *
```

We will see that 10 is the correct value when we analyze the assembly code. Before we do this, we will go over the new WebAssembly code within this main function which is:

```
        ...
        (local $ifoobar$0 i32)       ; define local variable
        ...
        (local.set $ifoobar$0)       ; set local variable
        (local.get $ifoobar$0)       ; get local variable
        ...
```

`(local $ifoobar$0 i32)` creates a local variable for the register named foobar with a 32-bit integer value. This line pertains to our integer declaration of `foobar`. The next line, `(local.set $ifoobar$0)`, sets the value of `foobar`. This is a result of initializing `foobar` with the value of $(2 + 3) * 2$ which equals 10. Also, the `$i` denotes that `foobar` is an integer. Lastly, `(local.get $ifoobar$0 i32)` gets the value from the register, allowing the local variable which is the integer value, 10, to be returned.

Let's again view the assembly code complied by the gcc compiler. Now we can also compare easy_mult.cc and easy_mult_var.cc in this regard.

```
main:
    push   rbp
    mov    rbp, rsp
    mov    DWORD PTR [rbp-4], 10
    mov    eax, DWORD PTR [rbp-4]
    pop    rbp
    ret
```

If you compare this assembly code to `easy_mult.cc`'s assembly code, you will notice that the only change is that `mov` is on an additional line. This is because to declare/initialize `foobar` as a local variable it must first be moved to a specific memory address, and then store the variable in a register. The two lines can be seen below:

```
    ...
    mov    DWORD PTR [rbp-4], 10    ; set memory at address [rbp-4] to 10
    mov    eax, DWORD PTR [rbp-4]   ; store address [rbp-4] in the register
    ...
```

"`mov DWORD PTR [rbp-4], 10`" sets the memory at address `rbp-4` to 10. `rbp-4` is where our iconic local variable, `foobar` will be stored. we can also see that the stored value, 10, is the correct output from

our equation. The reason why "foobar" isn't actually found in the assembly code is because the computer only needs the memory address on the stack instead of the actual variable name. The next line, `mov eax, DWORD PTR [rbp-4]`, simply stores `rbp-4` in the `eax` register.

Considering the Abstract Syntax Tree

Abstract syntax trees (AST's) can be used to form main intermediate representations of the compiler's front-end (for both WebAssembly and assembly). For an example of an abstract syntax tree, see Week 4 in the homework section. As you can see, AST's represent the grammatical structure of the parsed input. If you have noticed, there are no keywords or punctuation such as, `(),;,else`, in AST's because at this point, the compiler only cares about the abstract structure [AS]. The compiler uses recursion on the abstract syntax to generate the code found in our WebAssembly and assembly files. The branches and levels of abstract syntax within each tree correlate to the structure of assembly code. It is useful to think about this correlation when reviewing the assembly code.

3.4 find_avg.cc

Let's take a look at a less basic example. The following program also consists of a main function. However, it calls a function, `findAverage()` to calculate the average of the given double values, 12.3 and 27.5. Lastly, main simply closes by returning the typical value of 0. Here is the C++ code:

```
double findAverage(double a, double b) {
    double avg = (a+b)/2;
    return avg;
}

int main() {
    double myAvg;
    myAvg = findAverage(12.3, 27.5);

    return 0;
}
```

It is helpful to try to think about what the WebAssembly and assembly code may look like before looking at it. You would be correct to assume there would be a whole new function above the main function in both WebAssembly and assembly. Here is the respective WebAssembly code:

```
(module
    (func
        $findAverage              ; function name
        (param $da$0 f64)         ; input type double
        (param $db$0 f64)         ; input type double
        (result f64)              ; return 64-bit double
        (local $davg$0 f64)       ; define local variable double
        (local.get $da$0)         ; get input a
        (local.get $db$0)         ; get input b
        f64.add                   ; add 64-bit doubles
        (i32.const 2)             ; 2 (integer)
        f64.div                   ; divide 64-bit
        (local.set $davg$0)       ; set avg
        (local.get $davg$0)       ; get avg
        return                    ; return
    )
```

```
    (func
        $main                   ; function name
        (result i32)            ; return 32-bit integer
        (local $dmyAvg$0 f64)   ; define local variable double
        (f64.const 12.3)        ; 12.3 (double)
        (f64.const 27.5)        ; 27.5 (double)
        (call $findAverage)     ; call findAverage function
        (local.set $dmyAvg$0)   ; set myAvg
        (i32.const 0)           ; 0 (integer)
        return                  ; return
    )
    (export "main" (func $main))
)
```

Some of the primary differences in AssemblyLanguage code from this example from the last, are the usage of `param`, `f64`, and `call`. `param`, which can be found in the 2nd and 3rd lines of `$findAverage`, are simply the input parameters for the function. In this case, they input 12.3 and 27.5. In those same lines, we can see that there is now `f64`, which is different from `easy_mult_var` which used `i32`. This is because `f64` is capable of storing 64-bit double values unlike `i32` which does not have enough space. `f64` can be found on a variety of other lines where 64-bit doubles are required. Lastly, We should take a look at `(call $findAverage)` in `$main`. This simply is WebAssembly's way of calling a function and is using the provided constant double value on the line above. Furthermore, `func $findAverage` contains the math two add two inputted double values which is later returned as a type double. We can again see the Reverse Polish Notation (RPN):

```
        ...
        (local.get $da$0)       ; a
        (local.get $db$0)       ; b
        f64.add                 ; +
        (i32.const 2)           ; 2
        f64.div                 ; /
        ...
```

This equation is very similar to `easy_mult_var.cc`. It just uses division instead of multiplication with the purpose of finding an average.

```
(a                  a
 +                  b       a+b
 b)   -- to -->     +   =    2    =   (a+b)/2
 /                  2        /
 2                  /
```

Now we can move onto the assembly code version which should seem a bit more complicated. There is also a key below that contains the meanings of the register types, instruction types and more. Please view them on the next two pages.

```
findAverage(double, double):
    push   rbp
    mov    rbp, rsp
    movsd  QWORD PTR [rbp-24], xmm0     ; set mem address ptr [rbp-24] to 64-bit double reg xmm0
    movsd  QWORD PTR [rbp-32], xmm1     ; set mem address ptr [rbp-32] to 64-bit double reg xmm1
    movsd  xmm0, QWORD PTR [rbp-24]     ; move ptr of first user input to reg xmm0
    addsd  xmm0, QWORD PTR [rbp-32]     ; w/ ptr, add 2nd user input to val at reg xmm0
    movsd  xmm1, QWORD PTR .LC0[rip]    ; move RIP-rel address ptr (str double-word)to reg xmm1
    divsd  xmm0, xmm1                   ; divide xmm0 val (39.8) by val xmm1 (2)
    movsd  QWORD PTR [rbp-8], xmm0      ; move mem address ptr [rbp-8] to reg xmm0
    movsd  xmm0, QWORD PTR [rbp-8]      ; move reg xmm0 to mem address ptr [rbp-8]
    movq   rax, xmm0                    ; move xmm0 val to rax register
    movq   xmm0, rax                    ; move rax val to xmm0 register
    pop    rbp
    ret
main:
    push   rbp
    mov    rbp, rsp
    sub    rsp, 16                      ; create space for local variables
    movsd  xmm0, QWORD PTR .LC1[rip]    ; move RIP-rel address ptr (str double-word)to reg xmm0
    mov    rax, QWORD PTR .LC2[rip]     ; move RIP-relative address ptr to 64-bit rax reg
    movapd xmm1, xmm0                   ; move double in xmm0 to xmm1
    movq   xmm0, rax                    ; move rax ptr to xmm0
    call   findAverage(double, double)  ; call findAverage() w/ xmm1 (12.3) and xmm0 (27.5)
    movq   rax, xmm0                    ; move xmm0 ptr to rax reg
    mov    QWORD PTR [rbp-8], rax       ; move ptr in rax reg to [rbp-8] mem address
    mov    eax, 0                       ; move 0 to eax for return value
    leave
    ret
.LC0:                                   ; RIP-relative address for avg within findAverage()
    .long  0
    .long  1073741824
.LC1:                                   ; RIP-relative address for myAvg within main()
    .long  0
    .long  1077641216
.LC2:                                   ; RIP-relative address for resulting myAvg within main()
    .long  -1717986918
    .long  1076402585
```

find_avg.s Key

Instructions

`push`	Push data onto sstack
`mov`	Move to/from special registers
`movsd`	Move string double-wprd
`movapd`	Move aligned packed double-precision floating-point values
`movq`	Move quadword
`addsd`	Add low double-precision floating-point value
`divsd`	Divide scalar double-precision floating-point values
`sub`	Subtraction
`call`	Call procedure (function)
`leave`	Leave stack frame
`pop`	Pop data from stack
`ret`	Return from procedure

Registers

`rbp`	Special purpose register that points to the base of the current stack frame
`rsp`	Special purpose register stack pointer which points to the top of the current stack frame
`xmm0`	128-bit register which can store up to 2 64-bit floating points (double precision)
`xmm1`	128-bit register which can store up to 2 64-bit floating points (double precision)
`rax`	64-bit register used for return values
`eax`	64-bit general purpose register for temporary data storage/memory access

Other

`QWORD`	Quad-word 64-bit (16*4) value
`.LC_[rip]`	RIP-relative address that is relative to the instruction pointer

[AG]

As we can see, the assembly code is much more complicated than the WebAssembly. This is because there is much more involvement with the hardware since gcc is compiled to run on a local machine, and not the web. In the above key, we can view all of the different kinds of registers that are required for `find_avg.cc`. `rbp`, for example, points to the base of the current stack frame, and `xmm0` is designed to store 64-bit doubles. Keep in mind that these respective registers and instructions are specific to the architecture of x86 processors. There are also a variety of instructions that aren't found in the WebAssembly code. There are no `mov` instructions in WebAssembly, and this is because it does not have to worry about local hardware. The stack also seems much simpler when viewing WebAssembly code because it does not have to push and pop at the hardware level with registers.

4 Conclusion

As shown, studying the assembly language and especially WebAssembly behind individual is good for getting an understand of what the computer, or web service, needs in order to run programs. WebAssembly is great for viewing programs on the stack, and comparing the stack structure to the abstract syntax tree. Traditional assembly language, on the other hand, is better for getting an understanding of how the code eventually is processed by the computer hardware, specific to individual processor types. Even though it is not as practical as WebAssembly for understanding abstract syntax, it can be very interesting to see where software and hardware meet. Overall, there is and always will be an abundance of material that can be learned from low-level languages.

References

[PL] Compiler Construction 2022, Alexander Kurz, 2022.

[AC] Assembly Code & the Stack, Julien Barbier, 2017.

[AG] x86 Assembly Guide, University of Virginia Computer Science, 2006.

[WA] WebAssembly, MDN Web Docs, 2022.

[AL] Assembly Language, Jason Fernando, 2020.

[AS] Abstract Syntax, Christophe Dubach, 2017.