Programming Language Design & Implementation : Assignment 01

Santosh Adhikari

September 2023

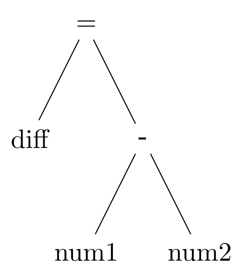
1 Ouestion 2

Part a)
The compiler as a whole receives the input in High Level Language(HLL) and converts into Assembly Language through series of phases.

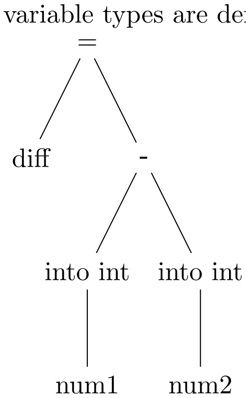
<u>Phase 1: FrontEnd</u>
a) Lexical Analyzer
It receives High level language as the stream of characters and converts them into tokens.
For example: the Initilization step for n1 i.e const int n1 = 25 gets broken down into const, int, = , 25
In this part of code, const is a keyword, '=' is a assignment operator, '25' is iconst,25.
In this part of code, *if (num1 - num2* ¡ 0), if is a keyword, '-' and '¡' are operators, num1 and num2 contains integer constants.

b) Syntax Analysis
The output tokens from the Lexical Analysis phase becomes the input to this phase. The syntax analyzer checks if tokens are syntaxically correct, organised or not. This phase creates parse tree.

In the part of the code above, diff = num1 - num2, where ¡id,1¿ = num1, ¡id,2¿ = num2, ¡id,3¿ = diff let us create a tree



c) Semantic Analysis
Input for this phase is parse tree. This phase checks for the variable declaration, scope of the variables, if variable types are defined or not. This phase is also for the logical error in the code.



d) Intermediate Code Generator

This phase produces Three Address Code (TAC). Every node of the tree defines a computation which involves maximum of three addresses. For example,

num1 = n1
num2 = n2
temp = num1-num2
if temp ¡ 0 jump to X: (*If temp is less than 0 go to label X*)
diff = temp
printf(diff)
return 0

Label X:
temp1 = -temp
diff = temp1
printf(diff)
return 0

Phase 2: BackEnd

a) Code Optimization

Code Optimizer is responsible for optimization of the original code. For example, we can use absolute function to remove the manual computation of absolute difference in the above code. Also we can directly use n1 and n2 for difference calculation instead of assigning them again with num1 and num2

diff = num1 - num2 can be done by simply doing diff = n1 - n2
diff = abs(diff) can be done instead of doing diff = -diff

b) Target Code Generation

Input is the optimized intermediate representation of the source code from the previous step.

LOAD n1 (*Load n1 onto the stack*)
STORE num1 (*Store the top of the stack into num1*) LOAD n2 (*Load n2 onto the stack*)
STORE num2 (*Store the top of the stack into num2*)
LOAD num1 (*Load num1 onto the stack*)
LOAD num2 (*Load num2 onto the stack*)
SUBTRACT diff, num1, num2 (*Subtract num2 from num1*)
STORE diff (*Store the result in diff*)

It produces a machine language code as output which can be placed in a location in memory and can be executed.

Part b)

```c
#include <stdio.h>
const int n1 = 25;
const int n2 = 39;

int main() {
    int num1, num2, diff;

    num1 = n1;
    num2 = n2;
    diff = num1 - num2;

    if (num1 - num2 < 0)
        diff = -diff;

    printf("\nThe absoute difference is: %d", diff);

    return 0;
}
```

```asm
; Listing generated by Microsoft (R) Optimizing Compiler Version 18.00.21005.1
    .686P
    .XMM
    include listing.inc
    .model flat

INCLUDELIB MSVCRTD
INCLUDELIB OLDNAMES

PUBLIC _n1
PUBLIC _n2
CONST SEGMENT ;Starting of a segment for defining constants
_n1 DD 019H    ;_n1 is a doubleword 32 bit constant with 019H(hex value)
_n2 DD 027H ; -n2 is a doubleword 32 bit constant with 027H(hex value)
CONST ENDS ; This directive is meant to end constant segment
PUBLIC _main
PUBLIC ??_C@_0BP@CMAHBJAF@?6The?5absoute?5difference?5is?3?5?$CFd?$AA@ ; ‘string’
EXTRN __imp__printf:PROC
EXTRN __RTC_CheckEsp:PROC
EXTRN __RTC_InitBase:PROC
EXTRN __RTC_Shutdown:PROC
; COMDAT rtc£TMZ
rtc$TMZ SEGMENT
__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
rtc$TMZ ENDS
; COMDAT rtc£IMZ
rtc$IMZ SEGMENT
__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
rtc$IMZ ENDS
; COMDAT ??_C@_0BP@CMAHBJAF@?6The?5absoute?5difference?5is?3?5?£CFd?£AA@

CONST SEGMENT
??_C@_0BP@CMAHBJAF@?6The?5absoute?5difference?5is?3?5?$CFd?$AA@ DB 0aH, 'T'
DB 'he absoute difference is: %d', 00H ; ‘string’
CONST ENDS
```

```
; Function compile flags: /Odtp /RTCsu /ZI
; COMDAT _main
_TEXT SEGMENT ; Start the segment where actual code is written
;Description of the stack frame of _main function

_diff$ = -32 ; size = 4 ; Local variable _diff£ = -32 is offset for diff. Address of diff is ebp-32. Here size is a 4 byte integer.
_num2$ = -20 ; size = 4 ; Local variable declaration 'num2' _num2£ = -20 is offset for num2. Address of num2 is ebp-20. Size = 4 menas 4 byte integer
_num1$ = -8 ; size = 4 ; _num1£ = -8 is offset for num1. Address of num1 is ebp-8

_main PROC ; COMDAT ; PROLOGUE OF main funtion STARTS


; 5 : int main() {
push ebp ;Save ebp (base pointer) to remember the frame information for caller. Register and Stack addressing in use

mov ebp, esp ; ebp is a stack pointer. 'mov' results in moving of stack pointer(esp) to the base pointer(ebp). { Frame of this function will be allocated from here.

sub esp, 228 ; 000000e4H ; Subtract 288 from the stack pointer in order to allocate 288 bytes of storage on stack for the local variable we declared above i.e diff, num1, num2.

push ebx ;ebx (extended base register) is a register usually used for holding temporary data. Here it is saved into the stack in order to save its value.
push esi ; save esi (index register) { it will be used in this function as a temporary register. Register and Stack addressing in use

push edi ; Destination index register helps saving the destination address into the stack

lea edi, DWORD PTR [ebp-228] ; Loading the address of [ebp-228] into edi to initialize a memory block to local variables defined above to a value 0xcccccccc.

mov ecx, 57 ; 00000039H ; Simply moves the value 57 to ecx register and to set up the rep stosd instruction to let know how many times to repeat.
mov eax, -858993460 ; ccccccccH ; Moves the value=-858993460 to eax register and let know  rep stosd instruction what value to store.
; 6 : int num1, num2, diff;
; 7 :
; 8 : num1 = n1;
mov eax, DWORD PTR _n1 ; Whatever is in _n1, move that value to eax register
mov DWORD PTR _num1$[ebp], eax ; This step replicates num1 = n1 in our code. It takes the value stored in eax register above and move it to _num1£.

; 9 : num2 = n2;
mov eax, DWORD PTR _n2 ; Move the value of constant _n2 into the eax register
mov DWORD PTR _num2$[ebp], eax ; In our code we have num2 = n2. So value stored in eax registor i.e n2 is moved to _num2£

; PROLOGUE OF main ENDS

; 10 : diff = num1 - num2; Now we are calculating the difference of num1 and num2 and storing it in diff which becomes -14 (25-39)
```

```

mov eax, DWORD PTR _num1$[ebp] ; Load num1 to eax (accumulator). Register and Memory addressing in use

sub eax, DWORD PTR _num2$[ebp]  ; Subtract num2 to eax. eax becomes num1 - num2 = -14. Register and Memory addressing in use

mov DWORD PTR _diff$[ebp], eax ; Here we store eax to difference. Difference becomes -14. Memory and Register addressing in use

; 11 :
; 12 : if (num1 - num2 < 0)
mov eax, DWORD PTR _num1$[ebp] ; Loads the value of _num1£ into eax registor
sub eax, DWORD PTR _num2$[ebp] ;Here we sub peforms subtraction of _num2£ varible from eax which contains the _num1£ variable
jns SHORT $LN1@main ; It is useful for checking if the subtraction result is signed or unsigned. If unsigned, it means result is less than 0 and condition jumps to the £LN1@main label

; 13 : diff = -diff;
mov eax, DWORD PTR _diff$[ebp] ; The difference of the values after subtraction is present in _diff£ variable and is loaded into eax registor
neg eax ;neg takes the value present in eax registor and makes it negative i.e -14 = -(-14) = 14
mov DWORD PTR _diff$[ebp], eax ;The negated value present in eax registor is moved back to _diff£ variable
$LN1@main:

; 14 :
```

```
; 15 : printf("\nThe absoute difference is: %d", diff);
mov esi, esp ; This is done to save the value present in esp into esi registor. Register addressing in use

mov eax, DWORD PTR _diff$[ebp] ; Move the value present in _diff£ which is the difference of two n1 and n2 into eax
push eax ; eax registor contains the result of difference of two number which now gets pushed into the stack for the function call for print. Register and Stack addressing in use

push OFFSET ??_C@_0BP@CMAHBJAF@?6The?5absoute?5difference?5is?3?5?$CFd?$AA@ ; Here the address of the format string is pushed into the stack
call DWORD PTR __imp__printf ; call printf by address (PTR). printf is external and is imported. It gets two parameters on top of stack

add esp, 8 ; esp += 8 to realize pop of two parameters passed before call. Register and Immediate addressing in use

cmp esi, esp ; compare esp with esi { the value of esp before call. This sets a compare bit. Register addressing in use

call __RTC_CheckEsp ; heck the compare bit to confirm that esp matches its value before call. This is a system check for correctness


; 16 :
; 17 : return 0;
xor eax, eax ;  eax = eax ^ eax = 0. A one cycle instruction to clear eax. Register addressing in use.
; 18 : }

;Now since our program is completed, we need to clean up the stack and restore the calling function's state before returning.

; EPILOGUE OF main STARTS

pop edi ;Removes the value form the top of the stack to restore the value of the edi register to its previous state.
pop esi ; pops a value from the stack, store in esi register and restores its previous state.
pop ebx ;removes the value from stack and restores ebx's previous value
add esp, 228 ; 000000e4H ;Adjusts the stack pointer upwards by 228 bytes to deallocate space allocated for local variables num1, num2, diff
cmp ebp, esp ;Compares the value in base and stack registor to check if they are equal or not.
call __RTC_CheckEsp ;Similarly like above, this is meant for checking runtime issues
mov esp, ebp  ; Here we restore esp. Register addressing in use


pop ebp ; restore ebp { the frame of the parent (caller) function. Register and Stack addressing in use

ret 0 ; Return 0. Control returns through indirect jump
; EPILOGUE OF main ENDS
_main ENDP ; End of the _main function
_TEXT ENDS ; End of the _TEXT section
```

```

END
```

2 Question 1

	i) Computation Paradigm	ii) Time & Space Effeciency	iii) Portablity
C++	Imperative, Object Oriented	Language's compilers comes with powerful optimization features and results in better performance. C++ provides low level control over memory management which allows to write optimized code	Highly portable across different porcessors, devices including embedded systems. Code can be written in platform independent using Standard library.
Dlang	Functional, Imperative, Object Oriented	Faster code execution as D is compiled language. Allows for manual memory management i.e programmer can allocate or free the memory according to the need.	Allows cross platform compiler support, Graphical User Interface development
Haskell	Fully functional programming language	Heskell do not perform computations until results are needed i.e Lazy evaluation which can lead for time and space usage more complicated. Garbage collector also adds more overhead.	Language supports cross platform so can be used on Mac, Windows. However using multi-parameter type classes (MTPC) with Functional dependency, it becomes platform specific.
Java	Object Oriented, Generic, functional, Imperative, reflective and concurrent	Code is compiled to bytecode and executed by JVM (Java Virtual Machine). JVM's garbage collector highly helps with space efficiency by reclaming memory that is no longer used.	Platfrom independent language. JVM provides abstraction layer which allows for Java code to run on any CPU architecture.
Prolog	Logic, Declarative	Excels in tasks involving symbiolic manipualation which sometimes result in extensive computation.This may lower the efficiency.	Portable across ISO compliant implementations. However, not all features are covered by this standard which results portablity issue.
Perl	Functional, Imperative, Object Oriented, Reflective	Pearl offers rich set of operators, data types for speed optimization.	Mainly used for text processing in Unix Systems, however ported to virtually support on Windows and Mac.
Python	Object oriented, Imperative, Functional	Being interpreted language, it runs slower than compiled languages, Allows only one thread to execute Python bytecode at a time. which can limit parallelism.	Supports cross platform compatiblity, Offers vast ecosystem of libraries, frameworks, Supports containerization - applications can be packaged into containers.
SQL	Declarative	Uses query optimization techniques to determine the most efficient way query execution, Properly maintained indexes can improve query performance but increases space complexity.	Adheres to the ANSI SQL standard : defines a common syntax and features for relational databases and enhances portablity.

	iv) Productivity	v) Application Area
C++	Versatile language, offers, wide range of applications, Offers vast libraries and framework accelerating productivity. However due to language complexity, initially it may take more time to understand memory management, pointers.	For developing embedded Systems, Database software development, Game development for resource intensive games
Dlang	Supports manual memory management, garbage collection, high level of abstraction like classes allows reuse of code, Built-in support for unit testing, making it easier to write and maintain test cases,	System Programming, Game engine development, Desktop applications
Haskell	Generally codes are short and easier to maintain and facilitates faster development being purely functional. Language has minimal semantic gap which reduces the complexity during coding.	Widely used in aerospace and defence field. In software development eg. Zoom, webservers etc. Popular proof assistant Agda is also uses Haskell.
Java	Easier to learn. Java is backward compatible i.e older versions can run on new platform version without modifications.	Development of desktop GUI applications, Mobile applications and game development.
Prolog	Due to its declarative nature, language leads to more concise code. Moreover, it leads to faster development time as code can be reperesent in formal logic.	Natural Language Processing, Database retrival and problem solving.
Perl	Allows for concise code, well-suited for parsing log files, extracting data from text due to its built in features.	Automation of repetitive system administrative tasks, Database and Network programming tasks. In security for penetration testing.
Python	Offers clean syntax, promotes formatting which enhances readblity, provides libraries, built in modules which saves time during development.	Data Analysis, Web Development, Machine Learning, Artificial Intelligence, Game Development, Database Development
SQL	Known for its simplicity, SQL has a standardized syntax ensuring consistancy, Offers rich ecosystem for developers use.	Database management systems, Data Warehousing, Financial Systems, Online gaming.