Dokumentasi Praktikum 1

Kompilasi Bahasa C dan Bahasa Assembly Intel® x86

Johan Kristianto1, Randi Hisyam Dzikroo2

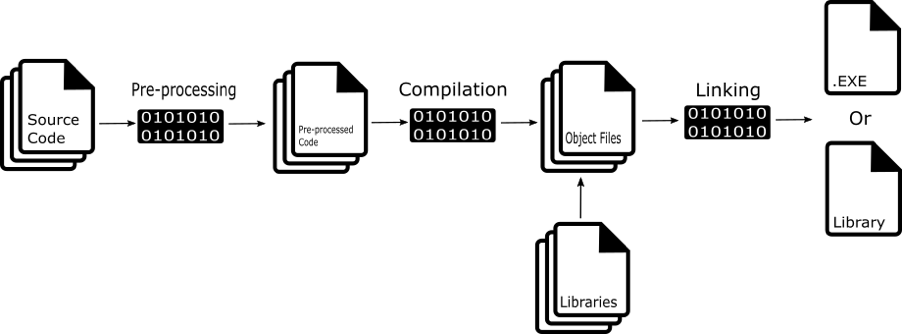
1,2Universitas Pendidikan Indonesia Kampus Daerah Cibiru, Jl. Pendidikan No. 15, Cibiru Wetan, Cileunyi, Kabupaten Bandung Jawa Barat (40625), Indonesia

|  |  |  |
| --- | --- | --- |
| **KETERANGAN** |  | **TUJUAN PRAKTIKUM** |
| Tanggal Praktikum : 24 April 2022  Perangkat Praktikum :   * MinGW-64 * Visual Studio Code * Visual Studio 2022 * Git Bash |  | 1. Praktikan memahami tahap-tahap kompilasi program dalam bahasa C sebagai bahasa tingkat tinggi hingga diperoleh bahasa tingkat rendah yang dapat dieksekusi oleh mesin. 2. Praktikan mampu melakukan kompilasi program bahasa C menggunakan compiler GCC beserta penggunaan makefile dan batch file. 3. Praktikan memahami bahasa assembly dan mampu melakukan analisis terhadap bahasa assembly Intel® x86 yang dihasilkan oleh compiler GCC. 4. Praktikan memahami penggunaan stack memory pada setiap procedure call. |
| **Kata Kunci:**  C/C++, Kompilasi, *Compiler*, Arsitektur Komputer |
|
| **Wirmanto Suteddy, S.T., M.T.**  Universitas Pendidikan Indonesia Kampus Daerah Cibiru, Jl. Pendidikan No. 15, Cibiru Wetan, Cileunyi, Kabupaten Bandung Jawa Barat (40625), Indonesia  Email: wirmanto.suteddy@upi.edu | | |

1. **Landasan Teoretis Praktikum**
   1. **Kompilasi Menggunakan GCC**

Bahasa pemrograman C adalah salah satu bahasa pemrograman tingkat tinggi, artinya bahasa C merupakan pemrograman yang memiliki karakteristik mendekati bahasa manusia sehingga mudah untuk dimengerti. Sayangnya, bahasa tingkat tinggi tidak dapat dimengerti oleh mesin karena mesin hanya dapat mengeksekusi program dengan bahasa tingkat rendah (bahasa biner). Oleh karena itu, program yang ditulis menggunakan bahasa tingkat tinggi perlu diterjemahkan terlebih dahulu ke bahasa tingkat rendah agar mesin dapat mengerti. Proses penterjemahan tersebut dikenal sebagai proses kompilasi untuk menghasilkan file berkekstensi *.exe* (*executable file*).

Kumpulan instruksi yang dimengerti oleh mesin disebut *instruction set*. Berdasarkan mikroprosesor yang digunakan, *instruction set* dibedakan menjadi dua jenis, yakni *complex instruction set computer* (CISC) dan *reduced instruction set computer* (RISC). Secara umum, proses kompilasi pada bahasa C terdiri dari empat langkah sebagai berikut.



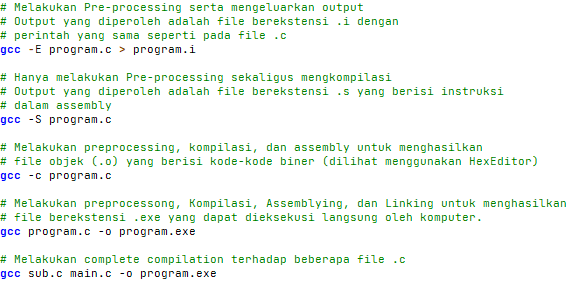
# *Pre-processor* merupakan instruksi pada program yang akan dikompilasi pertama kali oleh *compiler*. Contoh *pre-processor* adalah #include, #define, ‘\n’, dan lain-lain.

# *Compilation* merupakan penterjemahan bahasa tingkat tinggi (dalam hal ini C) ke dalam *instruction set*.

# *Assembler* merupakan proses menterjemahkan *instruction set* ke dalam file berekstensi objek (*.o*). perintah pada file berekstensi objek ditulis dalam bahasa biner.

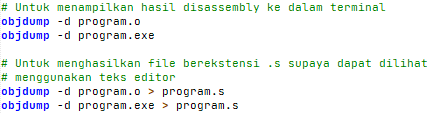
# *Linker* merupakan proses menggabungkan file objek dengan *dependency* yang berkorelasi.

Adapun beberapa perintah penting dalam melakukan kompilasi pada program C terdapat pada *ShellScript code*di bawah ini.



* 1. ***Disassembly* Menggunakan GCC**

Selain mengkonversi instruksi C ke dalam bahasa Assembly, GCC juga memberikan fitur untuk mengkonversi file berekstensi *object* atau *executable* ke dalam bahasa Assembly, yang dikenal sebagai proses *disassembly*. Berikut ini adalah cara-cara dalam melakukan disassembly dijelaskan pada *ShellScript code* di bawah ini.

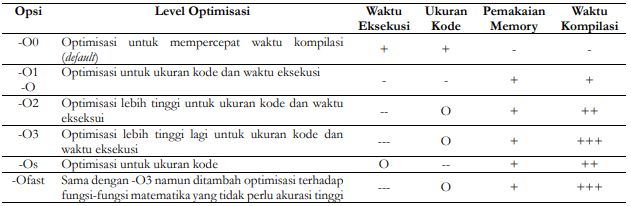


* 1. **Optimisasi Program Melalui Proses Kompilasi Menggunakan GCC**

GCC memberikan fitur untuk melakukan optimasi dari program yang telah kita kompilasi. Adapun optimasi yang dilakukan, yakni berkaitan dengan *execution speed, program size, compilation time,* dan kemudahan dalam melakukan *debugging*. Adapun contoh instruksi untuk melakukan optimasi sebagai berikut.

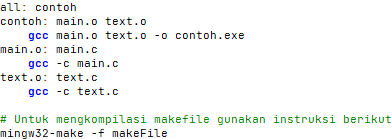


Selain instruksi di atas, ada beberapa instruksi optimasi lain dengan beberapa peningkatan yang berbeda pada setiap instruksinya. Berikut ini adalah tabel *list* dari instruksi optimasi pada GCC.

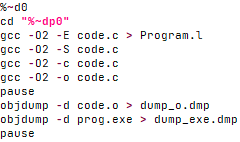


* 1. **Menggunakan *Make File* dan *Batch File***

*Make File* dan *Batch File* merupakan file yang dikhususkan untuk mengekskusi suatu perintah teriminal di dalam satu file tunggal. Make File dan Batch File akan memudahkan programmer apabila instruksi kompilasi dan eksekusi yang dilakukan sangat banyak. Dengan menggunakanMake File atau Batch File, Proses kompilasi tidak lagi diketik manual satu per satu, melainkan dengan mengklik file .bat yang telah dibuat. Pada sistem operasi Windows, instruksi di dalam Make File atau Batch File dapat ditulis dengan file berekstensi “.bat”. Berikut ini adalah contoh instruksi di dalam suatu Make File.



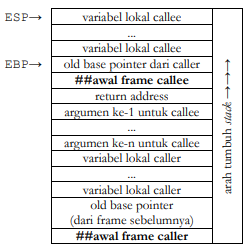
Konten di dalam file Make File akan berbeda dengan konten yang ada di dalam Batch File. Berikut ini adalah contoh instruksi yang ditulis di dalam file Batch File.



* 1. **Instruksi dan Bahasa Assembly Intel® x86**

Assembly merupakan bahasa pemrograman tingkat rendah dengan instruksi-instruksi di dalamnya disimbolkan dengan *mnemonic* yang disesuaikan dengan arsitektur yang digunakan (MIPS, Intel® x86, dan lain-lain). Arsitektur mikroprosesor Intel® x86 merupakan salah satu arsitektur yang banyak digunakan, oleh karena itu memahami instruksi pada bahasa Assembly Intel® x86 akan memudahkan programmer dalam melakukan debugging dan mengoptimasi program yang ditulis. Di dalam Intel® x86, terdapat banyak register yang digunakan, berikut ini adalah beberapa contoh register yang digunakan khususnya di dalam praktikum

* EAX, EBX, ECX, dan EDX adalah register 32-bit yang bersifat general storage.
* ESI dan EDI adalah register 32-bit yang digunakan sebagai indexing register. Register ini juga dapat digunakan sebagai general storage.
* ESP adalah register 32-bit yang digunakan sebagai stack pointer. Dengan demikian, ESP akan berisi nilai alamat (address) elemen puncak (top element) dari stack. Perlu diingat bahwa stack membesar dari alamat tinggi (high address) ke arah alamat rendah (low address). Dengan demikian, memasukkan elemen baru ke dalam stack akan mengurangi nilai alamat yang tersimpan pada ESP (menurun) sedangkan mengeluarkan elemen dari dalam stack akan menambah ESP (meningkat).
* EBP adalah register 32-bit yang digunakan sebagai base pointer. Dengan demikian, EBP akan berisi alamat dari current activation frame pada stack.
* EIP adalah register 32-bit yang digunakan sebagai instruction pointer. Dengan demikian, EIP akan berisi alamat dari instruksi selanjutnya yang akan dieksekusi.
  1. **Stack dan Procedure Call**

*Stack* adalah media penyimpanan yang menyimpan berbagai data dengan sistem FILO (*First In Last Out*). Stack umumnya disusun atas be berapa *activation frame*, setiap *frame* memiliki *base pointer* yang menunjukkan alamat tertinggi (*highest address*)pada *frame* tersebut. Ketika suatu program *caller* memanggil suatu prosedur bernama *calle*, *caller* akan memasukkan argumen-argumen untuk memanggil *calle* dari argumen terakhir hingga argumen paling awal secara berurutan ke dalam *stack*. Selanjutnya, *caller* akan memasukkan *return address* ke dalam *stack*. Kemudian, *calle* akan memasukkan alamat *old base pointer* milik *caller* ke dalam *stack* dan memperbarui nilai *base pointer* yang sesuai dengan *frame calle* (nilai base pointer yang baru sama dengan nilai stack pointer setelah old base pointer disimpan ke dalam stack). Kemudian callee melakukan alokasi terhadap variabel lokal dan melakukan komputasi sesuai dengan fungsi callee tersebut. Ketika callee selesai dieksekusi, callee akan menyimpan return value pada register EAX. Kemudian, callee akan membersihkan framenya sendiri dengan mengganti alamat base pointer dengan old base pointer yang telah disimpan pada stack. Kemudian, return address digunakan untuk melanjutkan eksekusi instruksi pada caller.

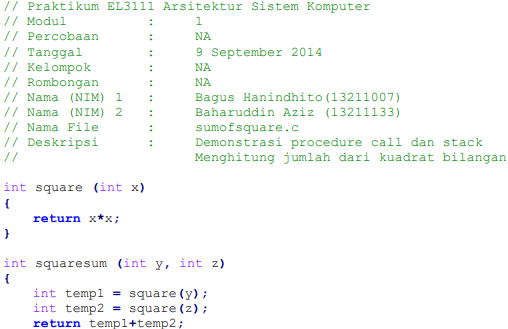
1. **Tugas Pendahuluan**
   * + 1. Jelaskan perbedaan antara masing-masing pilihan optimisasi dalam GCC (–O0, –O1, –O2, –O3, –Os, dan –Ofast)!

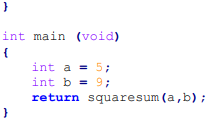
* **-O0** adalah bentuk optimasi untuk mempercepat waktu kompilasi dan pemakaian memory, tetapi juga menambah ukuran kode dan waktu pengeksekusian program tersebut.
* **-O1** adalah bentuk optimasi untuk mengurangi waktu mengeksekusi dan ukuran kode, tetapi menambah pemakaian memory dan waktu kompilasinya.
* **-O2** adalah optimasi waktu eksekusi dan ukuran kode yang lebih tinggi dari -O1. Adapun waktu kompilasi akan semakin bertambah, tetapi ukuran kode tidak berubah.
* **-O3** adalah optimasi yang lebih tinggi dari -O2 dengan waktu kompilasi semakin besar.
* **-Os** adalah optimasi yang khusus untuk mengurangi ukuran kode tetapi menambah waktu kompilasi.
* **-Ofast** adalah optimasi yang sama seperti -O3 tetapi beserta optimasi terhadap fungsi-fungsi matematika yang tidak memerlukan akurasi tinggi.

# Bahasa C merupakan bahasa yang banyak digunakan dalam membuat program pada beberapa platform. Sebagai contoh, bahasa C dapat digunakan untuk membuat program pada mikroprosesor berbasis Intel® x86. Bahasa C juga dapat digunakan untuk membuat program pada mikrokontroler AVR®. Di sisi lain, mikroprosesor Intel® x86 memiliki set instruksi yang jauh berbeda dibanding mikrokontroler AVR® ATmega. Menurut pengetahuan Anda tentang proses kompilasi bahasa C, apa yang menyebabkan bahasa C tetap dapat digunakan meskipun platform-nya berbeda?

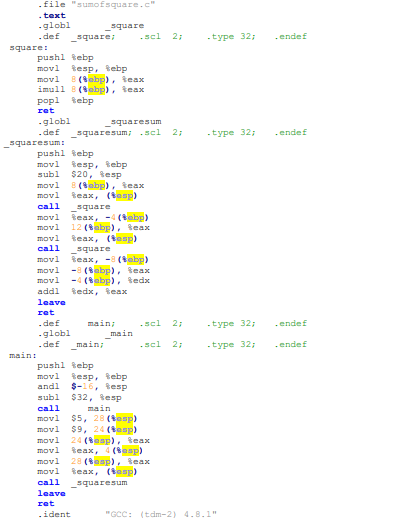
* C merupakan bahasa pemrograman yang terkompilasi, artinya program yang ditulis dengan bahasa C akan ditranslasi oleh *compiler* ke dalam bahasa yang lebih rendah (Assembly). Setiap arsitektur memiliki set instruksi yang berbeda satu dengan yang lain tetapi *compiler* C dapat mentranslasikan kode programnya ke dalam set instruksi yang sesuai dengan arsitektur yang dimiliki komputer tersebut. Oleh karena itu, C dapat digunakan untuk berbagai arsitektur dengan set instruksi yang berbeda karena penerjemahan kode-kode program C ke dalam Assembly disesuaikan dengan set instruksi pada arsitektur komputer yang dimilikinya.

# Diberikan contoh program sederhana dalam bahasa C sebagai berikut.





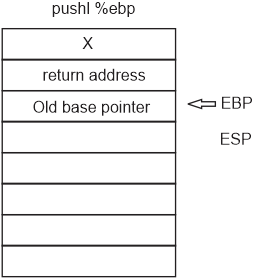
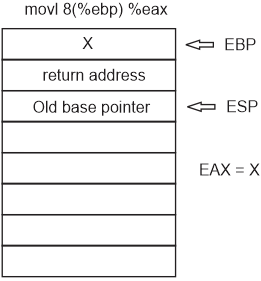
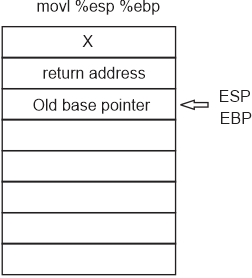
Kode program diatas di kompilasi untuk menghasilkan *ouput* berupa kode program dalam bahasa *assembly* sebagai berikut.



# Pada file *Assembly* tersebut, terdapat barisan kode *assembly* yang selalu dieksekusi di awal sebuah prosedur. Apa fungsi kode-kode *assembly* tersebut?



* EBP adalah *base pointer* yang berisi alamat dari *current activation frame* (*frame* suatu fungsi). Oleh karena itu, **pushl %ebp** artinya memasukkan *current activation frame* suatu fungsi ke posisi paling atas *stack*.
* **movl %esp %ebp** artinya adalah membuat register ESP menunjuk *old base pointer* yang ditunjuk oleh EBP, yakni yang berisi alamat dari *current activation frame*. Tujuannya agar frame memiliki *fixed offset* yang menjadi patokan ketika ESP terus bertambah.
  1. Gambarkan isi *stack* sebelum instruksi **imull 8(%ebp), %eax** pada saat prosedur *square* dipanggil pertama kali oleh prosedur *squaresum*!.



* 1. Prosedur rekursif merupakan prosedur yang memanggil dirinya sendiri secara berulangulang hingga kondisi berhenti dipenuhi. Berdasarkan pengetahuan Anda tentang procedure call dan stack ini, bagaimanakah penggunaan memory pada prosedur rekursif?
* Fungsi rekursif adalah fungsi yang memanggil dirinya sendiri untuk melaksanakan suatu instruksi dan akan berhenti ketika diberikan suatu kondisi tertentu. Oleh karena itu, fungsi rekursif memiliki kelebihan yakni menghemat penggunaan memori, karena pada saat pemanggilan prosedur, program menggunakan *activation frame* yang sama tanpa perlu membuat *activation frame* kembali. Selain itu, fungsi rekursif mengurangi penggunaan *slot* pada *stack* untuk argumen dan parameter. Kedua hal tersebut tentunya akan mengoptimalkan program dengan meminimalisir penggunaan *slot* pada *stack* secara berlebihan.

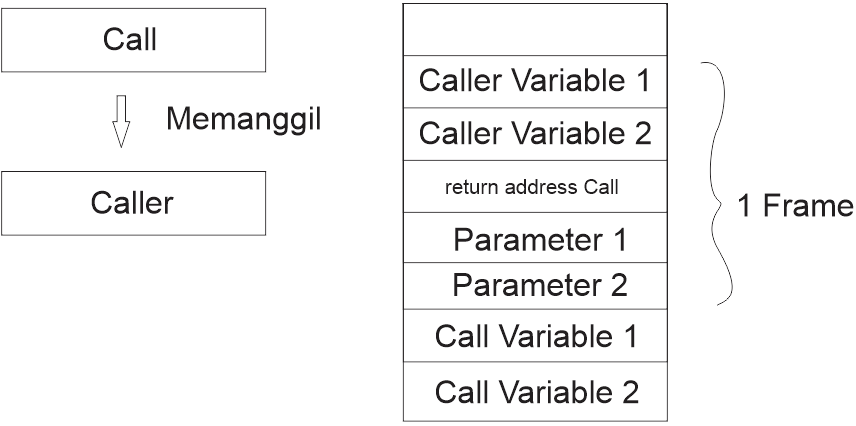
# Apa itu *Stack* dalam arsitektur komputer? Jelaskan fungsinya!

* Stack adalah media penyimpanan yang menyimpan berbagai data dengan sistem FILO (First In Last Out). Adapun fungsi utama dari *stack* adalah mrepresentasikan sistem penyimpanan memori yang di dalamnya memuat variable, argumen, dan objek-objek lainnya dalam suatu program. Stack umumnya disusun atas beberapa activation frame dan setiap frame memiliki base pointer yang menunjukkan alamat tertinggi (highest address) pada frame tersebut.

# Jelaskan apa saja yang terjadi pada stack ketika fungsi memanggil fungsi fungsi yang lain!

* Misalkan sutu fungsi bernama *call* akan memanggil fungsi yang bernama *caller*. Maka yang terjadi di dalam *stack* adalah parameter yang diminta *caller* akan di*-push* ke dalam stack secara terbalik, misalnya dari parameter kedua kemudian parameter yang pertama. Setelah itu, *return address* dari *call* akan di-*push* ke dalam *stack* diikuti dengan argumen-argumen yang ada di dalam *caller*.

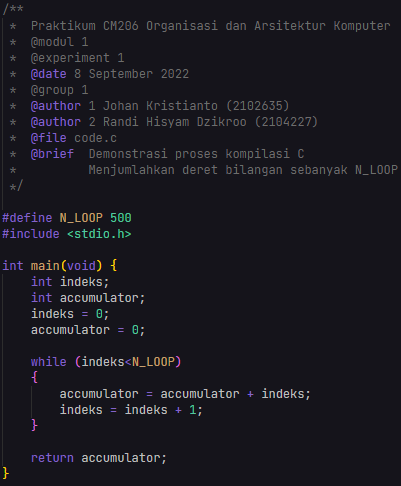
# Gambarkan *stack* pada keadaan nomor 5!



1. **Dokumentasi Praktikum**

# Tugas 1 : Proses Kompilasi Bahasa C Menggunakan GCC

* + - 1. Buatlah program dengan kode program sebagai berikut menggunakan teks editor Visual Studio Code dan simpan dalam file bernama **code.c**.



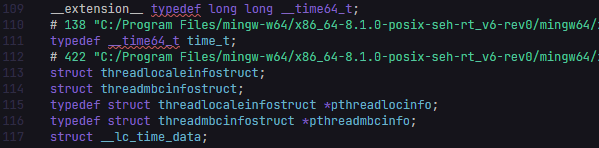
* + - 1. Lakukan *pre-processing* untuk menghasilkan file **code.i** dan amati hasilnya dengan teks editor Visual Studio Code.

Berikut ini adalah instruksi Git Bash untuk melakukan *pre-processing*.

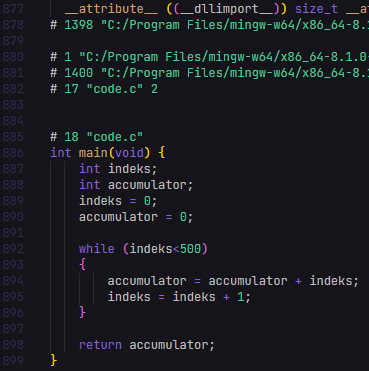


Hasil dari proses *pre-processing* pada file code.c akan menghasilkan file code.i. Kode program di dalam code.c terkandung di dalam code.i tetapi pada code.i kode-kode *macros* ditampilkan dan kode program di dalam code.c ditempatkan di baris terakhir pada file code.i.

Berikut ini adalah hasil *screen shot* dari sebagian kode-kode *macros* di dalam file code.i.



Berikut ini adalah kode program pada code.c yang terkandung di baris terakhir pada file code.i.



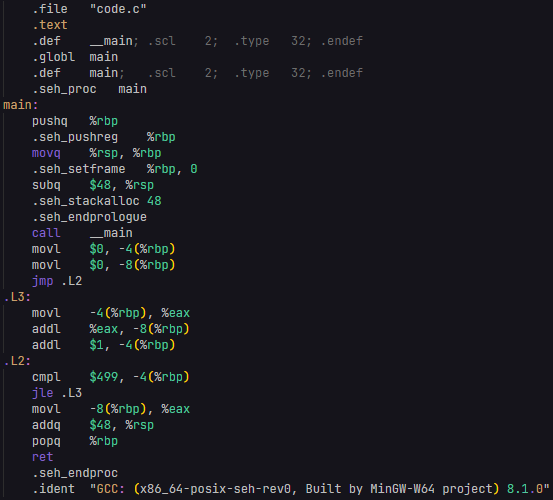
* + - 1. Lakukan kompilasi terhadap file code.c untuk memperoleh file **code.s**. Selanjutnya, amati kode-kode program pada code.s dengan teks editor Visual Studio Code.

Berikut ini adalah instruksi Git Bash untuk melakukan kompilasi.



Proses kompilasi pada file code.c akan menghasilkan file code.s. Instruksi di dalam code.s ditulis dengan bahasa *assembly* yang merupakan terjemahan dari instruksi dalam bahasa C pada file code.c.

Berikut ini adalah instruksi pada code.c dalam bahasa *assembly* yang disimpan di dalam file code.s.



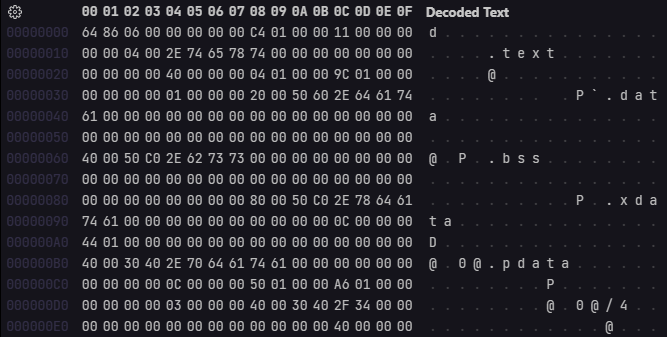
* + - 1. Lakukan proses *assemblying* pada file code.c untuk menghasilkan file **code.o**. Selanjutnya amati hasilnya menggunakan HeksEditor *extension* pada Visual Studio Code.

Berikut ini adalah instruksi Git Bash untuk melakukan *assemblying*.



Proses *assemblying* pada file code.c akan menghasilkan file code.o. Instruksi pada file code.o telah diterjemahkan ke dalam bahasa mesin (*binary instruction*) tetapi instruksi di dalam code.o tidak bisa langsung dieksekusi oleh komputer karena belum melalui proses *linking* dengan *file*s atau *libraries* yang bersangkutan. Instruksi pada code.o dapat diamati menggunakan HexEditor *extension* pada Visual Studio Code untuk merepresentasikan kode-kode biner ke dalam heksadesimal.

Berikut ini adalah hasil *screen shot* dari sebagian instruksi pada code.o yang diamati dengan menggunakan HexEditor.

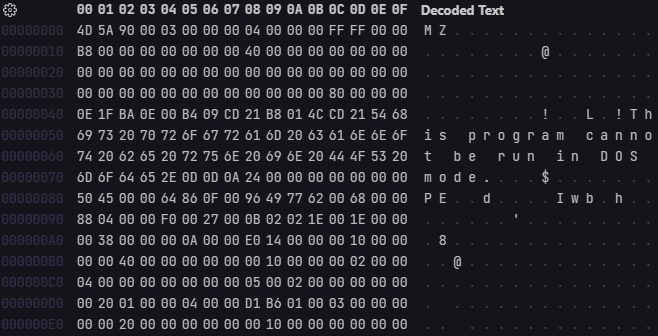


* + - 1. Lakukan seluruh proses kompilasi pada file code.c untuk memperoleh file code.exe. Selanjutnya, amati hasilnya menggunakan HexEditor *extension* pada Visual Studio Code.

Berikut ini adalah instruksi Git Bash untuk melakukan keseluruhan proses kompilasi.

Setelah code.c melalui proses *assemblying* untuk menghasilkan *object file* code.o, selanjutnya *object file* tersebut melalui proses *linking* untuk menghasilkan *executable file* code.exe yang dapat dieksekusi oleh komputer. Apabila diamati dengan HexEditor, *binary instruction* di dalam code.exe memiliki perbedaan yang signifikan dibandingkan dengan *binary instruction* code.o yang belum melalui proses *linking*. Adapun perbedaan yang tampak jelas antara instruksi pada code.exe dan code.o adalah kode-kode heksadesimal yang berbeda serta jumlah baris instruksi pada code.exe yang lebih banyak dari code.o.

Berikut ini adalah hasil *screen shot* dari sebagian instruksi pada code.exe yang diamati menggunakan HexEditor.

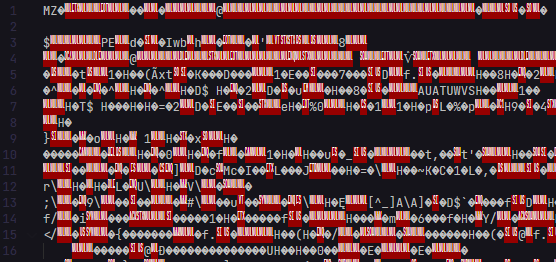


*Binary instruction* pada code.o dan code.exe hanya dapat diamati menggunakan HexEditor. Apabila *binary instruction* pada code.o dan code.exe dibuka menggunakan teks editor biasa seperti Notepad++, Atom, dan lain-lain maka instruksi tidak dapat terbaca dengan benar dan hanya tampak simbol-simbol enkripsi yang tidak beraturan. Hal ini disebabkan karena teks editor biasa tidak mendukung fitur untuk menampilkan kode-kode instruksi di dalam *executable file*.

Berikut ini adalah *screen shot* dari sebagian kode-kode instruksi pada code.o apabila diamati menggunakan teks editor biasa.

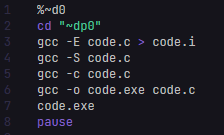


Berikut ini adalah *screen shot* dari sebagian kode-kode instruksi pada code.exe apabila diamati menggunakan teks editor biasa.



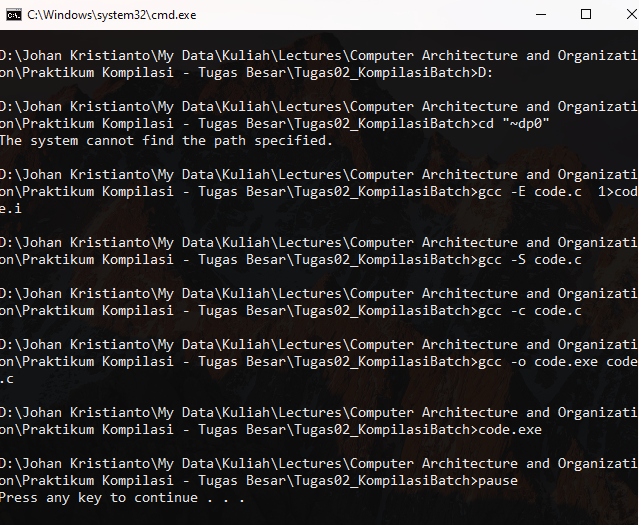
# Tugas 2 : Proses Kompilasi Bahasa C Menggunakan GCC dengan Bantuan Batch File

Salin file code.c dari tugas 1 ke dalam folder tugas 2. Kemudian, buat *Batch File* bernama **batch.bat** menggunakan teks editor Visual Studio Code dengan kode sebagai berikut dan simpan di folder yang sama dengan code.c.

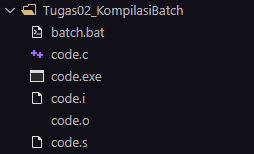


Klik dua kali batch.bat hingga muncul jendela *command prompt* dan amati semua file yang dihasilkan di dalam folder tugas 2.

Berikut ini adalah *screen shot* dari jendela *command prompt* ketika file batch.bat dibuka.



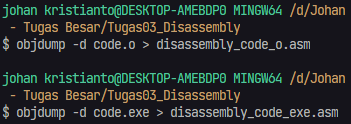
Setelah batch.bat dijalankan, di dalam folder tugas 2 terdapat beberapa *file* yang dihasilkan. Apabila diamati, *files* yang terbentuk sesuai dengan instruksi yang ditulis di dalam batch.bat yang sebelumnya ditulis secara manual melalui Git Bash pada tugas 1 sehinggas seluruh *file* yang dihasilkan sama seperti *file-file* yang dibuat pada tugas 1.



# Tugas 3 : Melakukan *Disassembly* Terhadap *Object File* dan *Executable File*

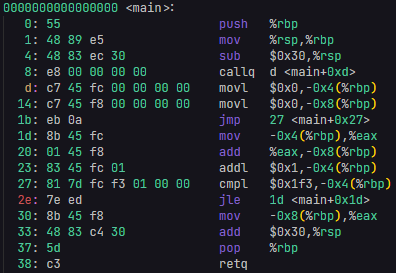
1. Salin code.o dan code.exe dari tugas 1 ke dalam folder tugas 3. Kemudian, lakukan *disassembly* dengan **objdump** untuk menghasilkan *assembly file* berekstensi “.asm” dari kedua *file* tersebut. Beri nama **disassembly\_code\_o** untuk *assembly file* dari code.o dan **disassembly\_code\_exe** untuk*assembly file* dari code.exe.

Berikut ini adalah instruksi Git Bash untuk melakukan *disassembly* terhadap code.o dan code.exe.

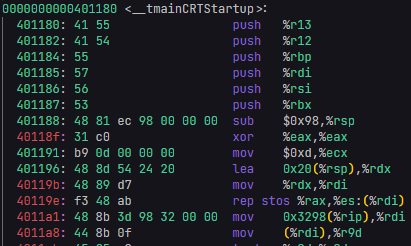
****

Apabila diamati, instruksi pada disassembly\_code\_o.asm dan disassembly\_code\_exe.asm memiliki beberapa perbedaan. Pada disassembly\_code\_o.asm, kode program *assembly* yang dihasilkan adalah kode program yang semula ditulis dengan bahasa C pada file code.c. Hal ini berbeda dengan kode *assembly* pada disassembly\_code\_exe.asm yang di dalamnya juga memuat kode-kode *assembly* untuk *header file* yang diakses oleh program. Hal ini disebabkan karena *executable file* telah melalui proses *linking* dengan *files* dan *libraries* yang bersangkutan, berbeda dengan *object file* yang belum melalui proses *linking*.

Berikut ini adalah *screen shot* sebagian kode *assembly* pada *file* disassembly\_code\_o.asm.

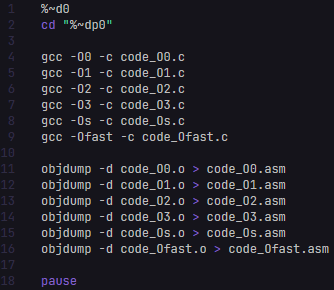


Berikut ini adalah *screen shot* salah satu kode *assembly* untuk *header file* pada *file* disassembly\_code\_exe.asm.

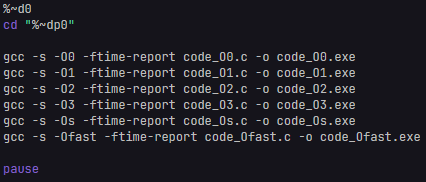


# Tugas 4 : Optimasi Kompilasi Program pada GCC

1. Salin file code.c dari folder tugas 1 menjadi enam file baru masing-masing bernama **code\_O0.c**, **code\_O1.c**, **code\_O2.c**, **code\_O3.c**, **code\_Os.c**, dan **code\_Ofast.c**. Kemudian, tempatkan kelima file tersebut ke dalam folder tugas 4.
2. Buat *batch file* bernama **batch.bat** kemudian masukkan instruksi sebagai berikut agar kelima file tersebut dapat dikompilasi dan dilakukan *disassembly* secara otomatis.



1. Buka seluruh *assembly file* yang telah dibuat kemudian amati menggunakan teks editor Visual Studio Code. Kemudian, gunakan fitur **-ftime-report** pada GCC untuk menganalisis waktu kompilasi dan penggunaan memori pada program serta catat data-data yang diperoleh. Instruksi dapat ditulis di dalam *batch file* seperti contoh berikut.

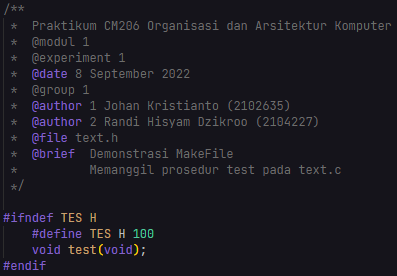


Fitur-fitur optimasi yang disediakan oleh GCC masing-masing memiliki kelebihan dan kekurangan. Peningkatan pada suatu aspek seringkali mempengaruhi performa pada aspek lainnya. Pada percobaan ini, diperolehlah data-data sebagai berikut.

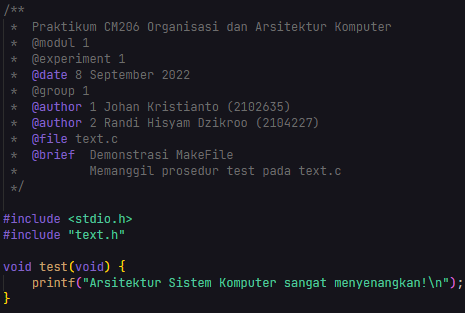
* Optimasi -O0 menghasilkan program dengan waktu kompilasi tercepat, yakni 0.03 detik dan penggunaan memori yang paling sedikit sebesar 2409 kB tetapi memiliki ukuran kode program *assembly* yang lebih besar.
* Optimasi -O1 menghasilkan program dengan waktu kompilasi 0.04 detik dan penggunaan memori sebesar 2436 kB. Selain itu, kode program *assembly*-nya memiliki ukuran yang lebih kecil dari optimasi -O0.
* Optimasi -O2 menghasilkan program dengan waktu kompilasi 0.04 detik dan penggunaan memori sebesar 2439 kB serta ukuran kode *assembly* yang sama dengan optimasi -O3, -Os, dan -Ofast serta lebih kecil dari optimasi -O1.
* Optimasi -O3 menghasilkan program dengan waktu kompilasi 0.05 detik dan penggunaan memori sebesar 2439 kB.
* Optimasi -Os menghasilkan program dengan waktu kompilasi 0.05 detik dan penggunaan memori sebesar 2438 kB.
* Optimasi -Ofast menghasilkan program dengan waktu kompilasi 0.05 detik dan memiliki penggunaan memori terbesar dari keseluruhan optimasi, yakni sebesar 2450 kB

# Tugas 5 : Kompilasi Beberapa File Kode dengan GCC

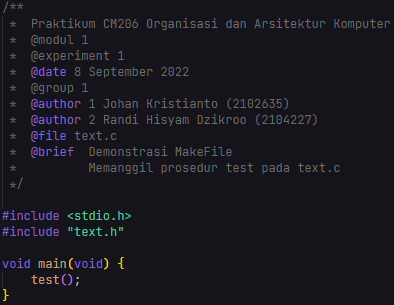
1. Buat *file* bernama **text.h** menggunakan teks editor Visual Studio Code dengan kode program sebagai berikut.



1. Buat *file* bernama **text.c** menggunakan teks editor Visual Studio Code dengan kode program sebagai berikut.

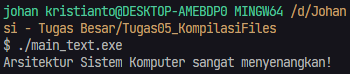


1. Buat file bernama **main\_text.c** menggunakan teks editor Visual Studio Code dengan kode program sebagai berikut.



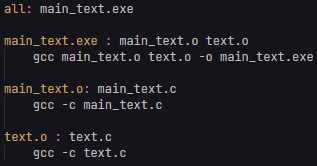
Selanjutnya, lakukan kompilasi terhadap file-file tersebut untuk menghasilkan *executable file* **main\_test.exe**. Adapun kompilasi dilakukan dengan menuliskan instruksi Git Bash sebagai berikut.

Setelah main\_text.exe berhasil dibuat, selanjutnya adalah menjalankan file tersebut untuk diamati hasilnya. Adapun hasil pengeksekusian dari main\_text.exe seperti pada gambar di bawah ini.





# Tugas 6 : Penggunaan Makefile pada GCC

1. Salin file main\_text.c, text.c, dan text.h dari folder tugas 5 ke dalam folder tugas 6.
2. Buat **Makefile** menggunakan teks editor dengan kode program sebagai berikut. Kemudian simpan dengan nama **makefile** tanpa menggunakan ekstensi.
3. Lakukan kompilasi pada **makefile** dengan menuliskan instruksi Git Bash sebagai berikut. Kemudian, jalankan main\_text.exe dan bandingkan dengan tugas 5.

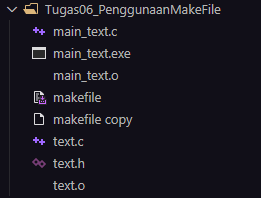


Berikut ini adalah *screen shot* dari hasil main\_text.exe setelah dieksekusi.



Ketika dijalankan, tidak ada perbedaan dari tugas 5, tetapi yang membedakan adalah *files* yang terbentuk tidak hanya main\_text.exe. Berdasarkan instruksi pada makefile, *files* yang terbentuk tidak hanya berekstensi *exeutable* melainkan terbentuk pula *object files*, seperti **main\_text.o** dan **text.o**. Oleh karena itu, setelah makefile dikompilasi kedua *object file* tersebut juga terbentuk.

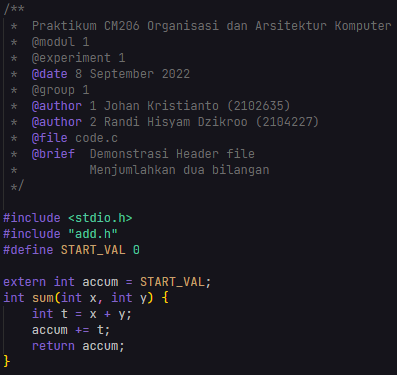
Berikut ini adalah *screen shot* dari keseluruhan *files* yang terbentuk di dalam *folder* tugas 6.



# Tugas 7 : Penggunaan Header Files

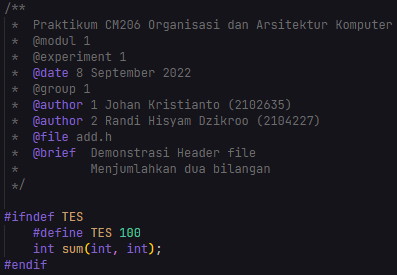
Percobaan ini bertujuan untuk melihat bagaimana penggunaan dari *header files* dan memahami bagaimana *keyword* *extern* bekerja terhadap suatu variable pada *header files*. Adapun langkah-langkah yang dilakukan dirumuskan sebagai berikut.

1. Buat program dengan kode sebagai berikut dan tuliskan pada teks editor Visual Studio Code kemudian simpan dengan nama **code.c**.

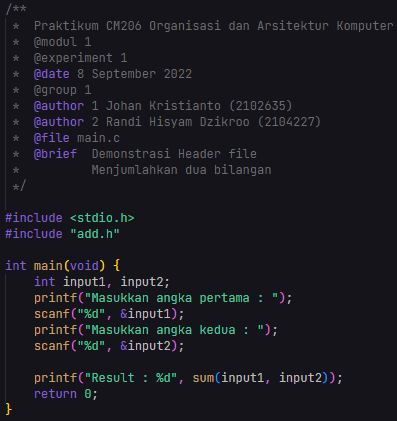


1. Buat sebuah *header file* bernama add.h yang di-*include* pada main.c. Gunakan *keyword extern* pada variable accum agar dapat dikenali dan dapat diakses pada main.c.

Berikut ini adalah *screen shot* dari *header file* add.h



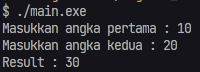
1. Buat program main bernama **main.c** yang menginput dua nilai dari user kemudian menjumlahkan kedua nilai tersebut dengan fungsi sum pada code.c dengan pemanggilan header files. Berikut ini adalah *screen shot* dari program main.c.



1. Lakukan kompilasi pada file code.c dan main.c dengan instruksi Git Bash sebagai berikut sehingga terbentuk file main.exe, Kemudian amati hasilnya.



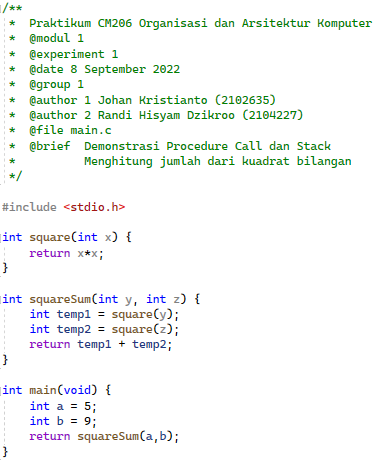
Berikut ini adalah main.exe setelah dieksekusi.

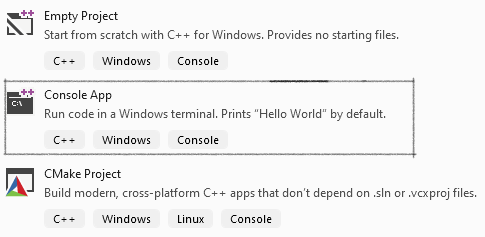


*Header File* merupakan salah satu fitur pada pemrograman C yang memungkinkan adanya pendeklarasian suatu fungsi di dalam suatu *file* tunggal dan dapat digunakan pada file-file lainnya. *Header File* sangat berguna apabila kita memiliki beberapa fungsi yang akan diimplementasikan pada file-file yang berbeda dengan hanya cukup mendeklarasikannya sekali di dalam *header file*. Penggunaan *header file* sangat mengoptimalkan efiisensi pemrograman karena suatu fungsi tidak perlu dideklarasikan ulang pada setiap file yang mengimplementasikannya melainkan cukup mengakses *header file* yang bersangkutan. Supaya suatu variable pada *header file* dapat diakses dan dikenali oleh file lain, variable tersebut diberikan *keyword extern* sebelum mendeklarasikan tipe datanya.

# Tugas 8 : Pemanggilan Prosedur dan Stack Memory

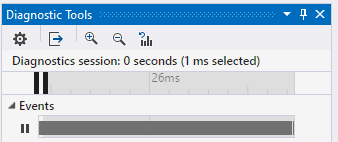
1. Buka program Visual Studio, kemudian buat project baru dengan kategori program **Console App** dan buat project tersebut dengan nama **Stack\_Procedure**.
2. Selanjutnya, hapus seluruh kode program *default* pada halaman **Stack\_procedure.cpp** dan tuliskan kode program pada tugas pendahuluan.



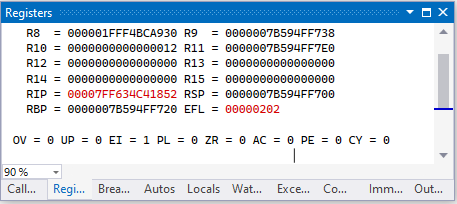


1. Tempatkan *breakpoint* pada masing-masing baris instruksi dan mulai lakukan *debugging* dengan mengklik *shortcut* F5.
2. Klik *shortcut* F11 untuk melakukan *step into*. Kemudian, amati perubahan pada seluruh *window* dari Visual Studio.

* Sebelum melakukan *step into*, tidak ada perubahan pada seluruh *windows* dari Visual Studio. Perubahan terjadi ketika *step into* dilakukan, *events* pada Diagnostic Tools menunjukkan pada *breakpoint* mana processor sedang mengeksekusi.



* Perubahan terjadi pada alamat yang ditunjukkan oleh beberapa *register* serta ditandai oleh tanda merah pada alamatnya.



1. Saat anak panah berada pada **return squareSum(a,b)** lihat isi *register RSP* dan *RBP* pada *registers*. Isikan nilai *registers RSP* dan *RBP* pada *address* di jendela *memory* kemudian catat nilai tersebut dan rekonstruksikan bentuk *stack* pada kondisi tersebut.

Isi register RSP : 000000E1147FFCB0

Isi register RBP : 000000E1147FFCD0

Nilai pada *memory* RSP :



Nilai pada *memory* RBP :

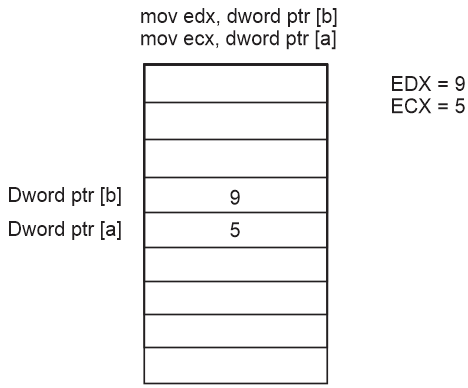


Rekonstruksi *stack* pada kondisi tersebut berdasarkan kode *assembly* dibawah ini digambarkan sebagai berikut.

Kode *Assembly* dari kondisi tersebut sebagai berikut.

Rekonstruksi *stack* digambarkan sebagai berikut.

1. Saat anak panah berada pada **int temp1 = square(y)** lihat isi *register RSP* dan *RBP* pada *registers*. Isikan nilai *registers RSP* dan *RBP* pada *address* di jendela *memory* kemudian catat nilai tersebut dan rekonstruksikan bentuk *stack* pada kondisi tersebut.



Nilai pada *memory* RSP :



Nilai pada *memory* RBP :

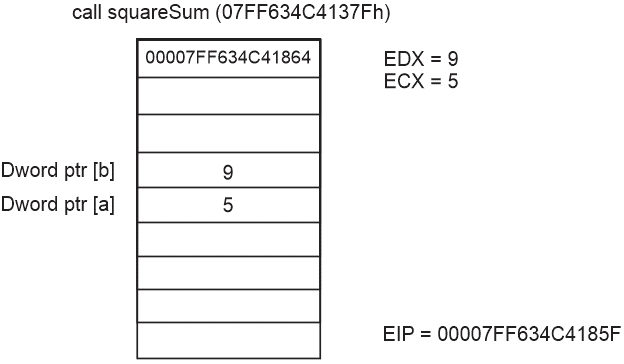


Rekonstruksi *stack* pada kondisi tersebut berdasarkan kode *assembly* dibawah ini digambarkan sebagai berikut.

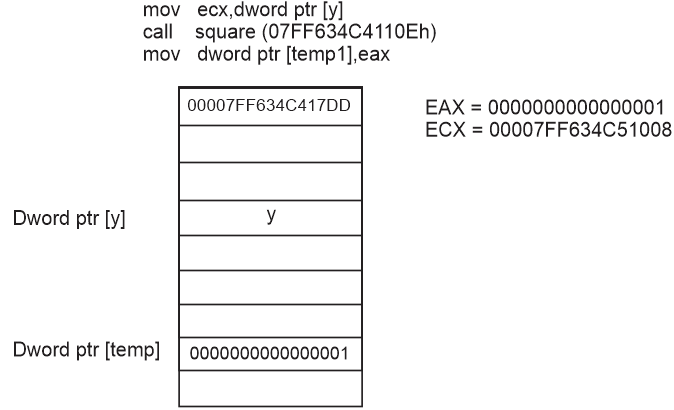
Kode *Assembly* dari kondisi tersebut sebagai berikut.



Rekonstruksi *stack* digambarkan sebagai berikut.







1. Saat anak panah berada pada **return x\*x** lihat isi *register RSP* dan *RBP* pada *registers*. Isikan nilai *registers RSP* dan *RBP* pada *address* di jendela *memory* kemudian catat nilai tersebut dan rekonstruksikan bentuk *stack* pada kondisi tersebut.

Nilai pada *memory* RSP :



Nilai pada *memory* RBP :

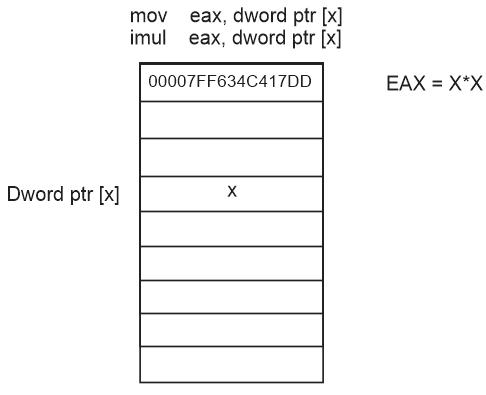


Rekonstruksi *stack* pada kondisi tersebut berdasarkan kode *assembly* dibawah ini digambarkan sebagai berikut.

Kode *Assembly* dari kondisi tersebut sebagai berikut.



Rekonstruksi *stack* digambarkan sebagai berikut.



1. Saat anak panah berada pada **return temp1 + temp2** lihat isi *register RSP* dan *RBP* pada *registers*. Isikan nilai *registers RSP* dan *RBP* pada *address* di jendela *memory* kemudian catat nilai tersebut dan rekonstruksikan bentuk *stack* pada kondisi tersebut.

Nilai pada *memory* RSP :



Nilai pada *memory* RBP :

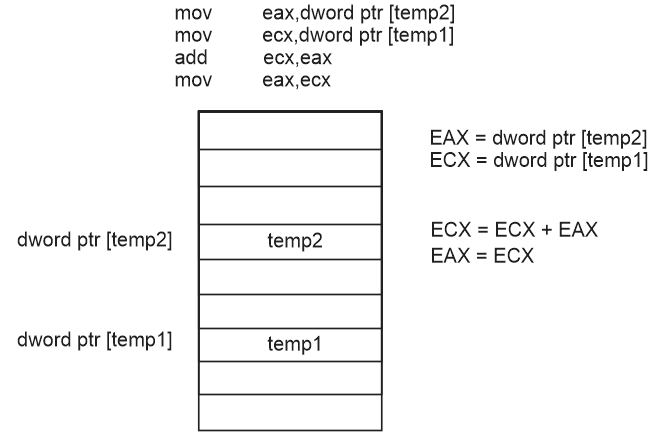


Rekonstruksi *stack* pada kondisi tersebut berdasarkan kode *assembly* dibawah ini digambarkan sebagai berikut.

Kode *Assembly* dari kondisi tersebut sebagai berikut.



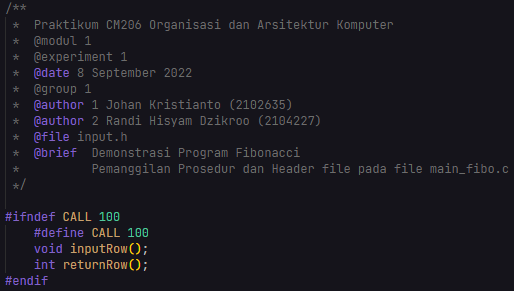
Rekonstruksi *stack* digambarkan sebagai berikut.



# Tugas 9 : Program Fibonacci

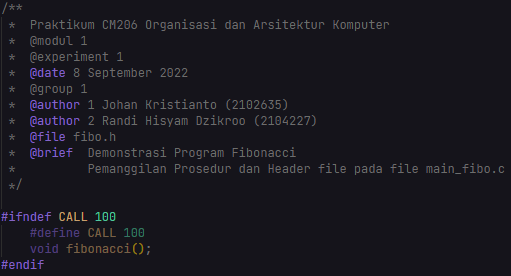
Tugas 9 adalah membuat program untuk menampilkan deret angka Fibonacci sebanyak n yang sesuai dengan input dari pengguna. Program ini terdiri atas tiga file utama, yakni **input.c** untuk menangani *user input*,**fibo.c**untuk menangani kalkulasi dalam menampilkan deret, dan **fibo\_main.c** sebagai program utama. Adapun *header file* untuk inputn.c dan fibo.c diperlukan dalam pemanggilan fungsi-fungsi tersebut. Kompilasi dilakukan dengan menggunakan makefile serta fungsionalitas program juga turut diperiksa.

Berikut ini adalah kode program pada header file **input.h** yang berisi *prototype* fungsi pada file **input.c.**

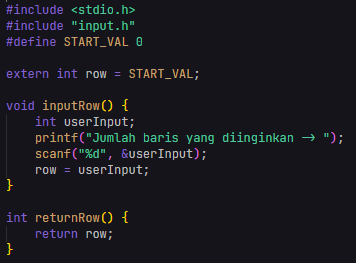


Di dalam header file **input.h** terdapat dua *prototype* untuk dua fungsi, yakni fungsi **inputRow()** untuk menangani penginputan banyaknya deret oleh user dan **returnRow()** untuk mengembalikan nilai dari banyaknya baris yang telah diinput oleh user.

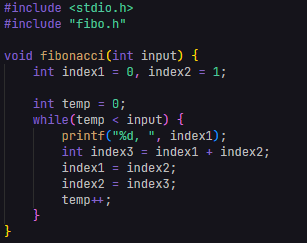
Berikut ini adalah kode program pada header file **fibo.h** yang berisi *prototype* fungsi pada file **fibo.c**



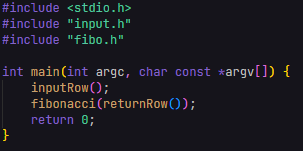
Berikut ini adalah instruksi di dalam file **input.c** yang mendefinisikan fungsi untuk menangani penginputan banyaknya deret yang diinginkan oleh *user* dan mengembalikan nilai tersebut oleh suatu fungsi.



Berikut ini adalah instruksi di dalam file **fibo.c** yang mendefinisikan *prototype* fungsi **fibonacci** pada **fibo.h**. Fungsi fibonacci akan memproses banyaknya deret yang dikeluarkan oleh program sesuai dengan banyaknya deret yang diinput oleh *user*.



Seluruh fungsi pada *header file* diatas akan diakses oleh file utama bernama **main\_fibo.c** dengan meng-*include* seluruh *header file* yang telah dibuat. Adapun instruksi pada file **main.fibo.c** sebagai berikut.



Hasil kompilasi dari seluruh file-file tersebut menghasilkan *executable file* bernama **main\_fibo.exe**. Berikut ini adalah *screen shot* hasil dari **main\_fibo.exe** ketika dijalankan.

Penggunaan *header file* berpengaruh signifikan terhadap proses pembuatan program. Penggunaan *header file* dapat mengoptimalkan efisiensi kode program sehingga fungsi-fungsi yang digunakan berulang tidak perlu didefinisikan secara berulang juga, melainkan dapat didefinisikan cukup sekali kemudian lakukan *include* terhadap *header file* tersebut. Penggunaan *header file* akan sangat membantu apabila program yang ditulis berskala besar, tetapi akan menjadi tidak efektif apabila program yang ditulis merupakan program sederhana. Pada contoh program fibonacci diatas, penggunaan *header file* tidak begitu membantu bahkan menjadikan proses pembuatan program tidak efektif karena program tersebut merupakan program sederhana dengan fungsi-fungsi sekali pakai pada *main function*.

