Organisasi Sistem Komputer

Bab 7. Subprogram

7.1 Subprogram 101

7.2 Variabel Lokal dari Subprogram

Subprogram

- **Subprogram** (fungsi, procedure dan method) adalah kode-kode instruksi yang pengeksekusiannya dipanggil dari bagian lain dari sebuah program
- Setelah selesai mengeksekusi semua kode, subprogram harus kembali ke instruksi berikutnya dari instruksi yang memanggilnya
- Subprogram harus dapat dipanggil berkali-kali
- Umumnya dapat menerima parameter-parameter dan mengembalikan sebuah nilai

Subprogram/Fungsi dalam C

```
// Subprogram/Fungsi add
// Input: integer x dan y
// Output: integer hasil x+y
int add(int x, int y) {

    return x + y;

}


void main() {
   int sum1, sum2;
   sum1 = add(1, 4);
   sum2 = add(7, 3);
}
```

Subprogram Menggunakan JMP

- Misalkan kita membuat subprogram yang menjumlahkan dua bilangan
  - Buat label kode yang di bawahnya berisi kode-kode subprogram
  - Panggil subprogram dengan instruksi jump ke label kode subprogram
  - Buat label kode setelah instruksi yang memanggil subprogram dan label kode ini menjadi tujuan lompatan setelah subprogram selesai mengeksekusi semua kode-kodenya
  - Buat label kode setelah instruksi lompatan subprogram; label kode ini untuk melompati subprogram

```
  ; kode assembly subprogram dengan jmp
  jmp  func          ; panggil subprogram func

ret:
  …
  jmp  end           ; lompat ke akhir program

func:
  …                  ; lakukan sesuatu
  jmp ret            ; Kembali ke baris setelah
                     ; dipanggil

end:
  …
```

Kode di atas perlu ditambahkan: kode untuk menerima parameter, kode untuk mengembalikan nilai dan membuat subprogram ini dapat dipanggil berkali-kali

Subprogram Menggunakan JMP

- Bagaimana membuat subprogram dapat menerima parameter?
  - Menyimpan nilai-nilai parameter pada register-register sebelum memanggil subprogram
- Bagaimana membuat subprogram memberikan nilai return (kembali)?
  - Menyimpan nilai kembali pada suatu register (umumnya di register `EAX`)

Kode disamping belum memenuhi kriteria dari sebuah subprogram: subprogram harus bisa dipanggil lebih dari satu kali

```
  ; kode assembly subprogram dengan jmp
  ; kita menyimpan parameter-parameter
  ; dalam register-register sebelum memanggil
  ; subprogram, sehingga dapat digunakan oleh
  ; subprogram
  mov  eax, 23      ; parameter 1 = 23
  mov  ebx, 45      ; parameter 2 = 45
  jmp  func         ; panggil subprogram func

ret:
  …
  jmp  end          ; lompat ke akhir program

func:
  add eax, ebx      ; tambahkan parameter 1
                    ; dan parameter 2
                    ; nilai return disimpan di eax
  jmp ret           ; Kembali ke baris setelah
                    ; dipanggil

end:
```

Subprogram Menggunakan JMP

- Bagaimana membuat subprogram dapat dipanggil berkali-kali?
  - Kita tetap harus membuat sebuah label kode setelah instruksi pemanggilan subprogram
  - Daripada menggunakan label sebagai tujuan lompatan keluar dari subprogram, kita menyimpan alamat kembali ke salah satu register
  - Register yang digunakan pada praktik umumnya (konvensi) adalah register `ECX`
  - Tujuan lompatan untuk keluar dari subprogram menggunakan alamat di dalam `ECX`

```
; kode assembly subprogram dengan jmp
…
  mov  eax, 12      ; operand 1 = 12
  mov  ebx, 14      ; operand 2 = 14

  ; pemanggilan subprogram pertama
  mov  ecx, ret     ; simpan alamat kembali
                    ; ke ECX
  jmp  func         ; panggil ke-1 subprogram func
ret:
  …
  …

  ; pemanggilan subprogram kedua
  mov  ecx, ret2    ; simpan alamat instruksi
                    ; selanjutnya ke ECX
  jmp  func         ; panggil ke-2 subprogram func
ret2:
  …
  …
func:
  add eax, ebx      ; lakukan sesuatu
  jmp ecx           ; Kembali ke baris setelah
                    ; dipanggil
```

Subprogram

- Subprogram menggunakan JMP mempunyai kekurangan:
  - Membutuhkan label setelah instruksi yang memanggil subprogram
  - Kita harus mengingat register-register yang kita gunakan untuk menyimpan parameter-parameter dan nilai kembali
- Solusi yang memudahkan mengimplementasikan subprogram:
  - Runtime Stack
  - Dua instruksi: `CALL` dan `RET`
- Sebelum membahas Runtime Stack, `CALL`, dan `RET`, kita akan membahas mengenai konsep “Pengalamatan Tidak Langsung (Indirect Addressing)”

Indirect Addressing

- Misalkan deklarasi variabel berikut:

```
L       dd      01h, 02h
```

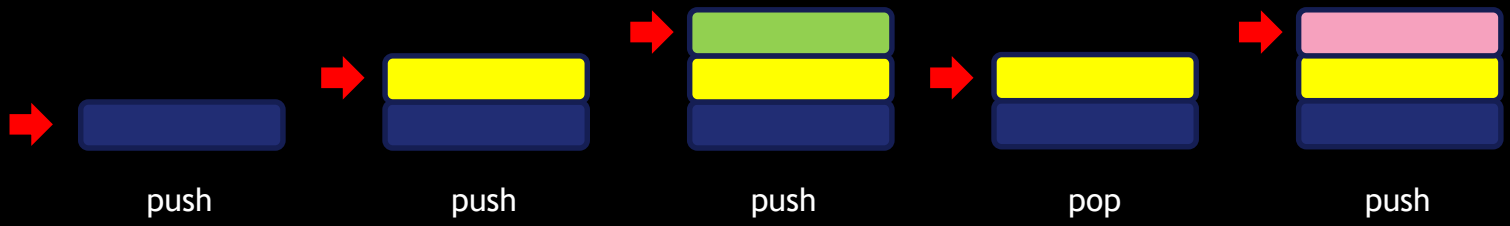
- Register dapat menyimpan “data” atau “alamat”
  - Menyimpan “data” : `mov eax, [L] ; eax = 01h`
  - Menyimpan “alamat” : `mov ebx, L ; ebx = 0000 0FF8`
- Menyimpan alamat dalam register disebut sebagai *Indirect Addressing*
- Kita dapat mengakses data dari alamat yang disimpan dalam register:

```
mov ecx, [ebx + 4]      ; ecx = 02h
```

- Menyimpan alamat dalam register, memungkinkan kita mengimplementasikan stack

Stack

- Stack (tumpukan) adalah struktur data yang menerapkan Last-In-First-Out
- Stack mempunyai dua operasi:
  - Push : menaruh elemen pada bagian paling atas stack
  - Pop : mengambil elemen paling atas dari stack
- Stack dikelola dengan menggunakan pointer yang menunjuk ke elemen top (teratas) dari stack



Runtime Stack

- Runtime Stack – stack dalam memori array yang dikelola CPU saat program berjalan dengan memanfaatkan register `ESP` (Extended Stack Pointer)
- Runtime Stack berkembang ke alamat memori yang lebih kecil
- Register `ESP` menyimpan alamat memori dari data paling atas
- Data yang di-push dan di-pop dapat sebesar 2 atau 4 byte, namun secara konvensi semua data dalam stack sebesar 4 byte
- Meng-push sebuah elemen ke stack:
  - Sintaks: `PUSH` *operand* *(operand berupa register, memori, atau immediate value)*
  - Mengurangi nilai `ESP` dengan 4 dan menulis data 4-byte ke alamat memori yang disimpan dalam `ESP`
  - Contoh:

```
push    eax             ; meng-push 4 byte nilai eax
push    dword 42        ; meng-push 4 byte nilai decimal 42
```

Runtime Stack

- Meng-pop sebuah elemen dari stack:
  - Sintaks: `POP` *operand* *(operand berupa register atau memori)*
  - Mengambil nilai dari bagian teratas stack, menyimpannya dalam operand, dan menambahkan `ESP` dengan 4
  - Contoh:

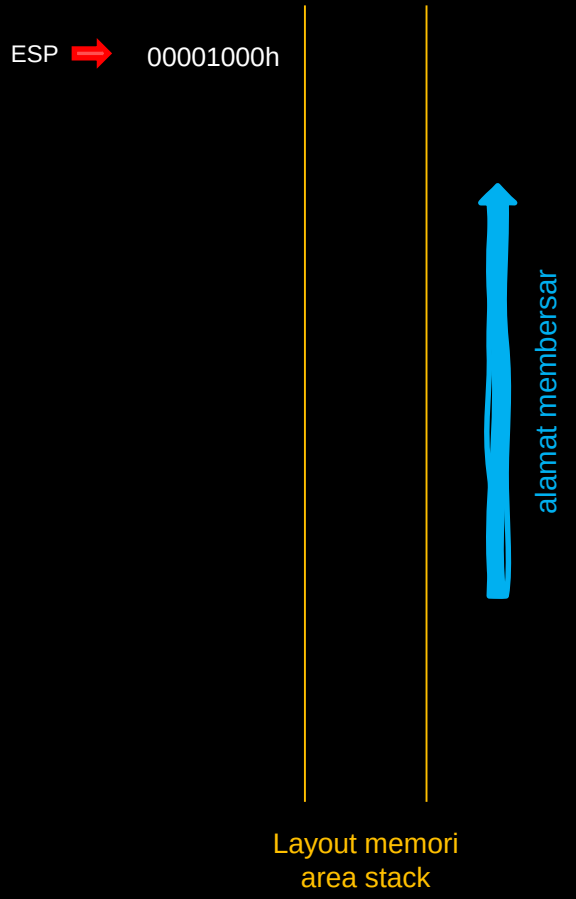
```
pop   eax          ; meng-pop 4 byte nilai dari bagian teratas
                   ; stack, simpan ke eax, tambahkan 4 ke ESP
pop   ebx          ; meng-pop 4 byte nilai dari bagian teratas
                   ; stack, simpan ke ebx, tambahkan 4 ke ESP
```

- Mengakses sebuah elemen pada stack:
  - Dilakukan dengan membaca 4 byte pada alamat dalam `ESP`
  - Contoh:

```
mov   eax, [esp]    ; membaca elemen teratas pada stack
```

Ilustrasi Runtime Stack

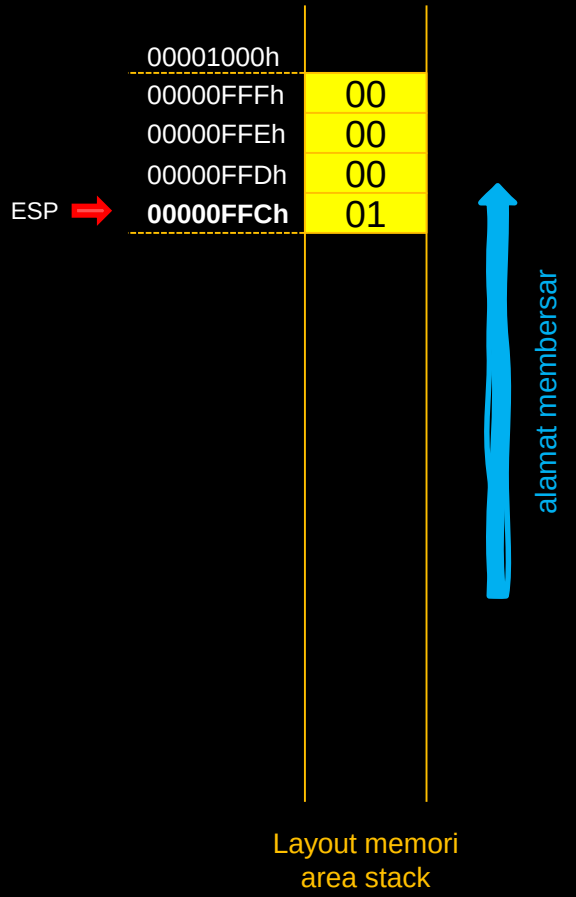
- Asumsikan `ESP` = 0000 1000h, sebelum stack digunakan



Ilustrasi Runtime Stack

- Asumsikan `ESP` = 0000 1000h, sebelum stack digunakan

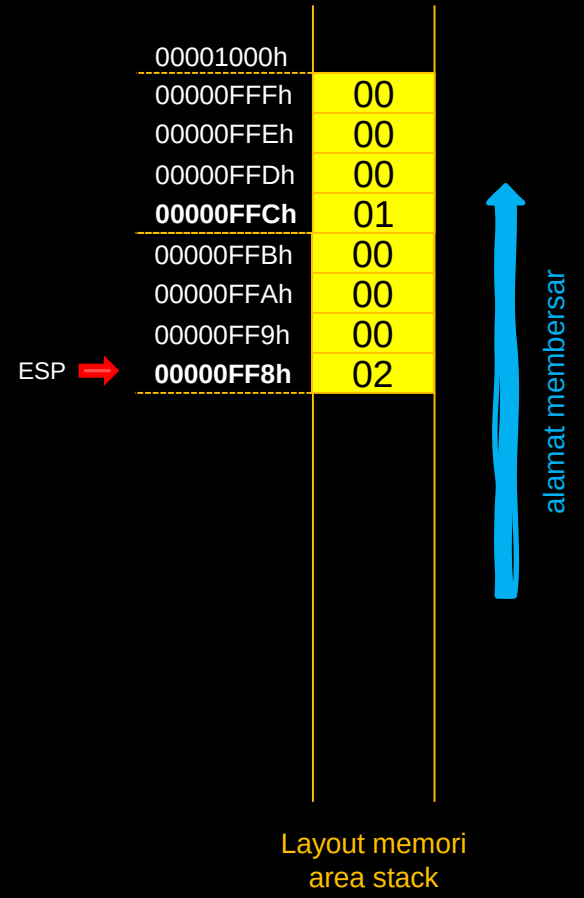
```
push    dword    1          ; ESP = 0000 0FFCh
```



Ilustrasi Runtime Stack

- Asumsikan `ESP` = 0000 1000h, sebelum stack digunakan

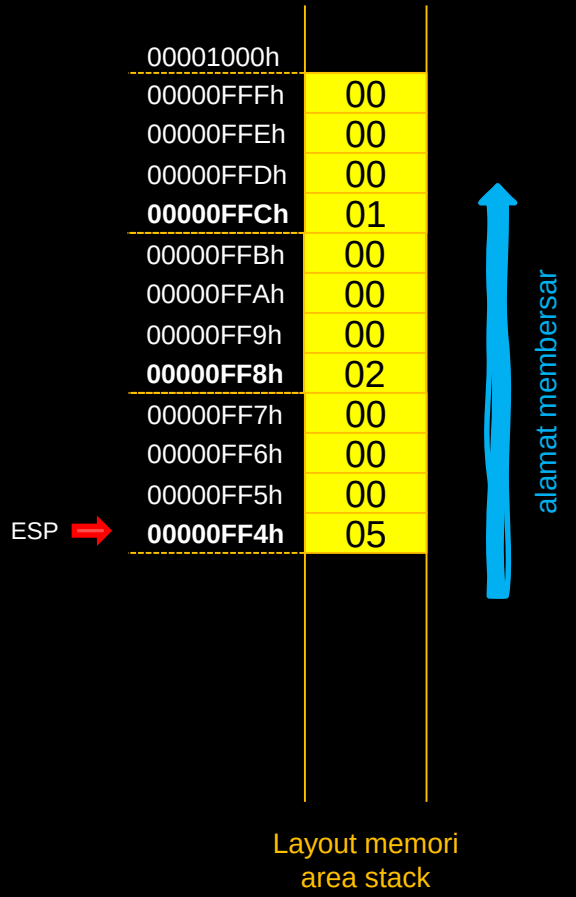
```
push    dword    1          ; ESP = 0000 0FFCh
push    dword    2          ; ESP = 0000 0FF8h
```



Ilustrasi Runtime Stack

- Asumsikan `ESP` = 0000 1000h, sebelum stack digunakan

```
push    dword    1          ; ESP = 0000 0FFCh
push    dword    2          ; ESP = 0000 0FF8h
push    dword    5          ; ESP = 0000 0FF4h
```

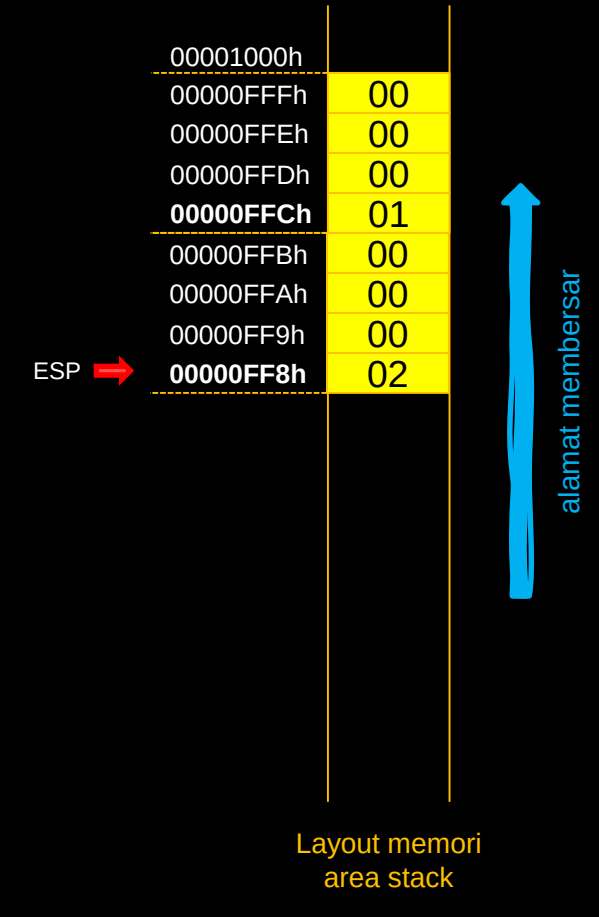


Ilustrasi Runtime Stack

- Asumsikan `ESP` = 0000 1000h, sebelum stack digunakan

```
push    dword   1       ; ESP = 0000 0FFCh
push    dword   2       ; ESP = 0000 0FF8h
push    dword   5       ; ESP = 0000 0FF4h

pop     eax             ; ESP = 0000 0FF8h
                        ; EAX = 5
```

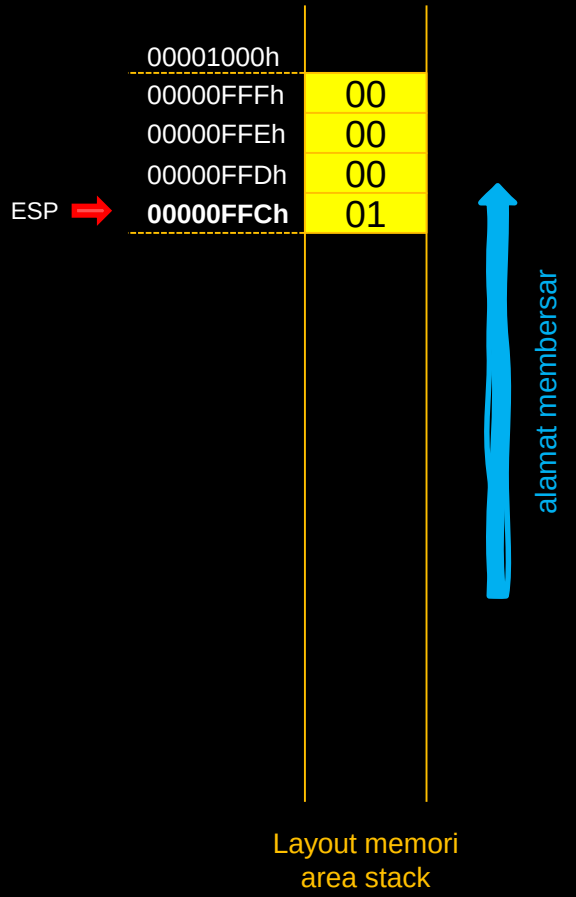


Ilustrasi Runtime Stack

- Asumsikan `ESP` = 0000 1000h, sebelum stack digunakan

```
push    dword   1       ; ESP = 0000 0FFCh
push    dword   2       ; ESP = 0000 0FF8h
push    dword   5       ; ESP = 0000 0FF4h

pop     eax             ; ESP = 0000 0FF8h
                        ; EAX = 5
pop     ebx             ; ESP = 0000 0FFCh
                        ; EBX = 2
```

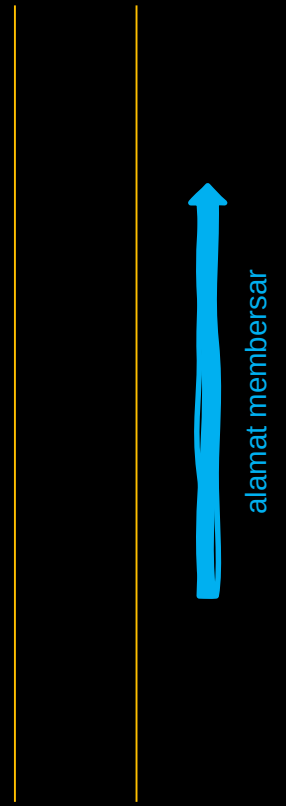


Ilustrasi Runtime Stack

- Asumsikan `ESP` = 0000 1000h, sebelum stack digunakan

```
push    dword   1       ; ESP = 0000 0FFCh
push    dword   2       ; ESP = 0000 0FF8h
push    dword   5       ; ESP = 0000 0FF4h

pop     eax             ; ESP = 0000 0FF8h
                        ; EAX = 5
pop     ebx             ; ESP = 0000 0FFCh
                        ; EBX = 2
pop     ecx             ; ESP = 0000 1000h
                        ; ECX = 1
```



Register ESP

- Register `ESP` selalu berisi alamat dari elemen teratas pada stack
  - Catatan: Pada ilustrasi sebelumnya `ESP` menunjuk ke alamat terendah karena stack membesar ke alamat memori lebih rendah
- Meskipun `ESP` termasuk GPR (General Purpose Register) yang bisa digunakan sebagai tempat penyimpan apapun, JANGAN gunakan `ESP` untuk data
- Isi dari `ESP` di-update otomatis oleh instruksi `PUSH` dan `POP`

Subprogram Menggunakan Stack

- Kegunaan utama dari stack untuk subprogram adalah untuk menyimpan dan mengembalikan nilai-nilai pada register
- Misalkan program Anda menggunakan `EAX` untuk menyimpan suatu data dan memanggil suatu subprogram yang ditulis oleh orang lain
- Anda tidak mengetahui apakah subprogram tersebut menggunakan `EAX`
  - Jika ya, maka nilai `EAX` Anda akan hilang
- Solusi dengan stack:
  - Push `EAX` ke stack
  - Panggil subprogram dan biarkan subprogram tersebut berjalan sampai selesai dan kembali
  - Pop `EAX` untuk mengambil nilai `EAX` semula
- Instruksi yang memudahkan penggunaan stack untuk subprogram:
  - `PUSHA` : untuk push semua GPR ke stack
  - `POPA` : mengambil nilai yang di-push oleh `PUSHA`

PUSHA dan POPA pada Skeleton

```
segment .data
    ; directive Dx

segment .bss
    ; directive RESx

segment .text
    global _main
    _main:
        ; Routine "setup"
        enter   0, 0
        pusha

        ; Program Anda di bawah

        ; Routine "cleanup"
        popa
        mov     eax, 0
        leave
        ret
```

Simpan semua nilai register GPR pada stack sebelum menjalankan kode program utama, karena nilai-nilai pada register ini mungkin digunakan oleh program driver

Kembalikan nilai dari stack ke register-register seperti pada awal sebelum program utama berjalan, sehingga program driver dapat berjalan semestinya

Instruksi CALL dan RET

- Salah satu hal yang menyulitkan dalam pembuatan subprogram menggunakan `JMP` adalah kita harus menyimpan alamat kembali ke suatu tempat (`ECX`) sebelum memanggil subprogram
- Dua instruksi x86 yang memudahkan:
  - `CALL`:
    - Mem-push alamat dari instruksi selanjutknya ke stack
    - Lompat ke suatu label (memanggil subprogram)
  - `RET`:
    - Meng-pop stack dan ambil alamat kembali
    - Lompat ke alamat kembali tersebut (keluar dari subprogram)

Subprogram dengan CALL dan RET

Subprogram dengan JMP

```
  …
  mov  ecx, ret     ; simpan alamat kembali
                    ; ke ECX
  jmp  func         ; panggil ke-1 subprogram func
ret:
  …
  …
  mov  ecx, ret2    ; simpan alamat instruksi
                    ; selanjutnya ke ECX
  jmp  func         ; panggil ke-2 subprogram func
ret2:
   …
   …
func:
  …                 ; lakukan sesuatu
  jmp ecx           ; Kembali ke baris setelah
                    ; dipanggil
```

Subprogram dengan CALL dan RET

```
  …
  call  func  ; panggil ke-1 subprogram func
  …
  call  func  ; panggil ke-2 subprogram func
  …
func:
  …           ; lakukan sesuatu
  ret         ; kembali
```

CALL pada Skeleton

```
segment .data
   ; directive Dx

segment .bss
   ; directive RESx

segment .text
   global _main
   _main:
      ; Routine "setup"
      enter  0, 0
      pusha

      ; Program Anda di bawah

      ; Routine "cleanup"
      popa
      mov    eax, 0
      leave
      ret
```

Instruksi untuk kembali ke program driver dari program _main

Activation Records

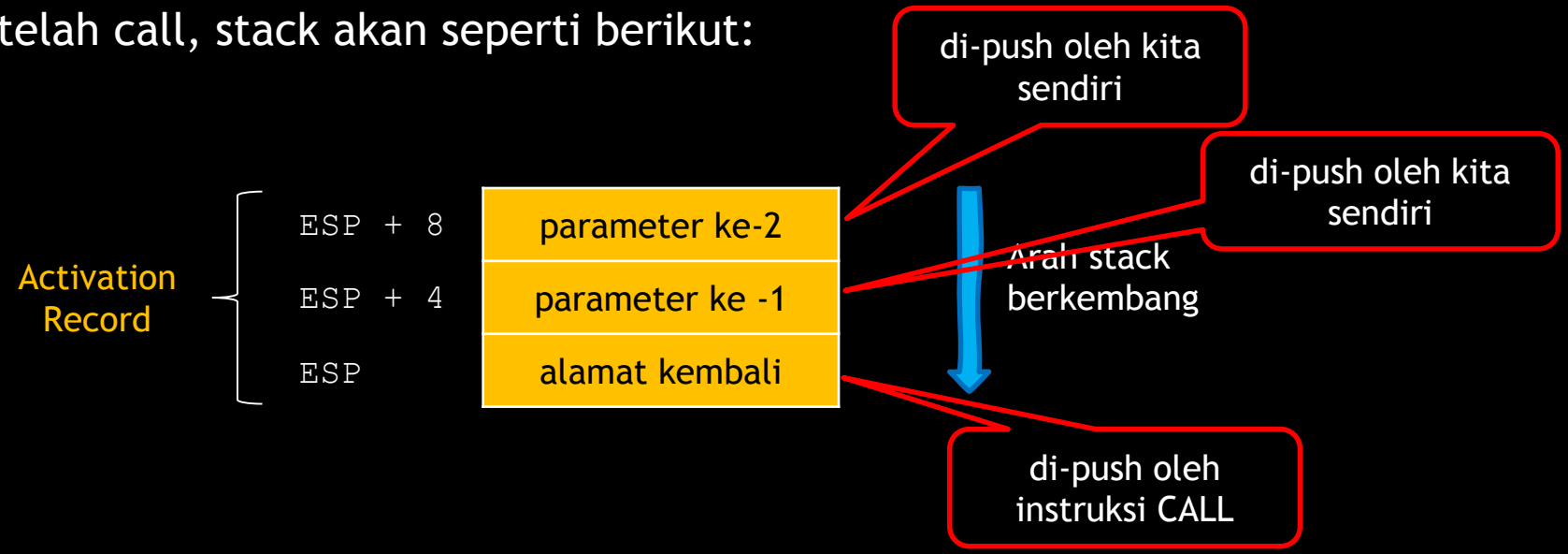
- Stack berguna untuk menyimpan dan mengambil alamat kembali (return address), yang dilakukan dengan instruksi `CALL` dan `RET`
- Namun, stack mempunyai kegunaan lebih dari ini
- Secara umum, ketika memanggil sebuah subprogram, kita menaruh semua informasi berguna ke stack
- Ketika subprogram kembali, informasi ini di-pop keluar dari stack dan pemanggil subprogram dapat secara aman melanjutkan eksekusi
- Kumpulan “informasi berguna” ini disebut sebagai *activation record* (atau “stack frame”)
- Salah satu komponen terpenting dari activation record adalah parameter-parameter yang diberikan ke fungsi
- Hal lainnya adalah alamat kembali, seperti yang telah kita lihat

Activation Records

- Untuk memanggil sebuah subprogram, kita harus ikuti langkah berikut:
  - Push parameter-parameter ke stack
  - Jalankan instruksi `CALL`, yang meng-push alamat kembali ke stack
- Jika subprogram membutuhkan lebih dari satu parameter, bagaimana urutan push parameter-parameter tersebut ke stack ?
- Dalam konvensi (compiler bahasa C) parameter-parameter yang di-push ke stack dalam urutan terbalik:
  - Misal, subprogram f(a, b, c)
  - Parameter c di-push ke stack pertama kali
  - Parameter b di-push ke stack kedua kali
  - Parameter a di-push ke stack ketiga kali

Membentuk Activation Record

- Misal, kita ingin memanggil (call) sebuah subprogram dengan dua parameter 32-bit
- Setelah call, stack akan seperti berikut:



Menggunakan Parameter

- Dalam kode subprogram, parameter-parameter dapat diakses melalui *indirection* dari pointer stack
- Pada contoh:

```
mov  eax, [ESP + 4] ; taruh parameter ke-1 ke EAX
mov  eax, [ESP + 8] ; taruh parameter ke-2 ke EBX
```

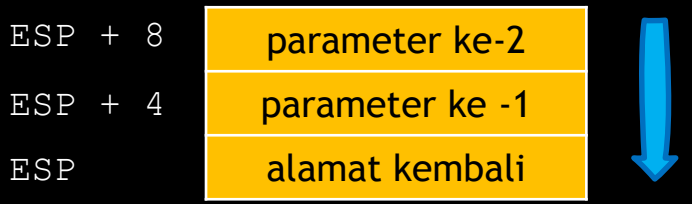
- Umumnya, subprogram tidak meng-pop parameter dari stack, karena subprogram masih membutuhkan alamat kembali yang berada di bagian paling atas dari stack

ESP dan EBP

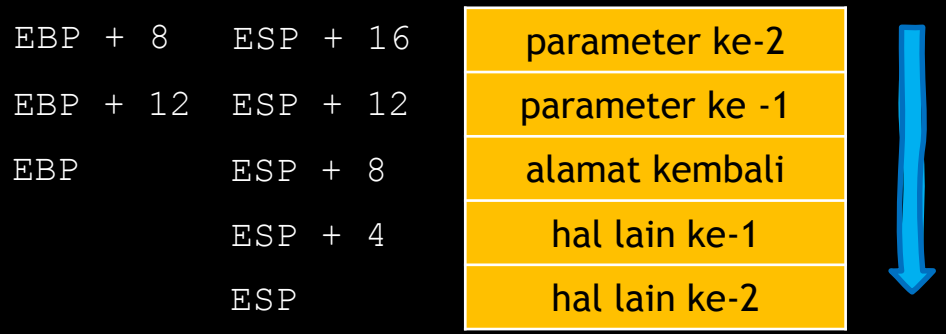
- Terdapat satu masalah ketika mereferensi parameter-parameter menggunakan `ESP`
- Jika subprogram juga menggunakan stack untuk hal lain, `ESP` akan termodifikasi
  - Jadi pada satu titik, parameter ke-2 dapat diakses dengan `[ESP + 8]`
  - Dan pada satu titik lain, parameter ke-2 mungkin diakses dengan `[ESP + 12]` atau `[ESP + 16]` tergantung seberapa besar stack berkembang
- Untuk menghindari permasalahan ini, berdasarkan konvensi, kita harus menggunakan register `EBP` sebagai sebuah basis untuk menyimpan nilai `ESP` sesaat setelah subprogram mulai
- Setelahnya, parameter kedua selalu diakses dengan `[EBP + 8]` dan parameter kesatu selalu diakses dengan `[EBP + 4]`

ESP dan EBP

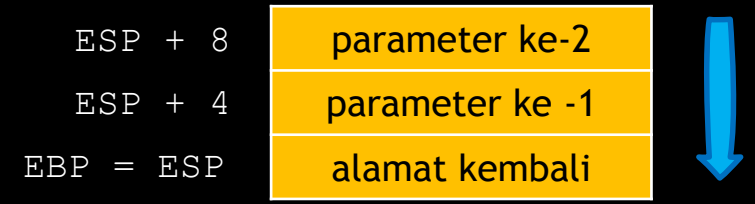
- Stack ketika subprogram mulai



- Subprogram dapat menggunakan stack untuk hal lain



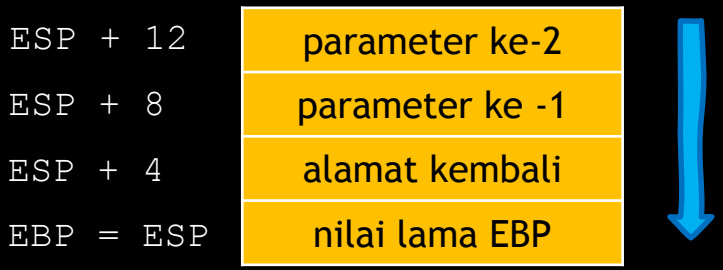
- Jadikan `EBP = ESP`



Parameter-parameter masih dapat direferensikan sebagai EBP+4 dan EBP+8

ESP dan EBP

- Terdapat masalah lain: pemanggil subprogram mungkin menggunakan `EBP`
  - Misalkan pemanggil adalah subprogram lain yang juga menggunakan `EBP` sebagai basis untuk mengakses parameter-parameternya sendiri
- Solusi: Berdasarkan konvensi, kita harus menyimpan nilai `EBP` ke stack lalu men-set `EBP=ESP`, sesaat setelah subprogram mulai
- Jadi, stack harus berisi seperti berikut sebelum menjalankan kode-kode subprogram



Skeleton Subprogram

```
func:

    push    ebp             ; simpan EBP dari pemanggil
    mov     ebp, esp        ; set EBP = ESP

     …                      ; kode-kode subprogram


    pop     ebp             ; kembalikan EBP pemanggil
    ret                     ; kembali ke pemanggil
```

Kembali dari Subprogram

- Setelah subprogram selesai, kita harus membersihkan isi dari stack
- Stack mempunyai:
  - Nilai lama `EBP`, alamat kembali, dan parameter-parameter
- Nilai lama `EBP` di-pop dari subprogram dengan instruksi: `pop  ebp`
- Alamat kembali di-pop oleh instruksi `ret`, sehingga tersisa parameter-parameter
- Konvensi dari compiler C adalah kode pemanggil subprogram yang bertanggung jawab untuk membersihkan parameter-parameter dari stack
  - Beberapa Bahasa lainnya mengharuskan subprogram yang membersihkan parameter-parameter

Memanggil Subprogram

- Untuk memanggil subprogram:

```
push    dword  2        ; parameter ke-dua
push    dword  1        ; parameter ke-satu
call    func            ; panggil subprogram
add     esp, 8          ; pop kedua parameter
```

- Perhatikan, untuk melakukan pop kedua parameter tersebut, kita menambahkan nilai 8 ke pointer stack `ESP`
  - Karena kita tidak memerlukan lagi nilai-nilai dari parameter ini, ini lebih cepat daripada memanggil pop dua kali
  - Ini satu kasus dimana kita memodifikasi `ESP` secara langsung
- Nilai dua parameter tetap berada dalam memori, namun nantinya akan di-overwrite ketika kita memanggil subpgrogram lain atau ketika stack digunakan untuk hal lain

Nilai Kembali?

- Berdasarkan konvensi compiler C, nilai kembali selalu disimpan dalam `EAX` ketika subprogram kembali
  - Menjadi tanggung jawab pemanggil untuk menyimpan nilai `EAX` sebelum memanggil subprogram dan mengembalikan nilai `EAX` setelah subprogram kembali