8. BAHASA ASSEMBLY

Instruksi mesin dinyatakan dengan pola 0 dan l. Pola semacam itu sangat sulit untuk dijelaskan pada saat membahas atau menyiapkan program. Oleh karena itu, kita menggunakan nama simbolik untuk menyatakan pola tersebut. Sejauh ini kita telah menggunakan kata-kata biasa seperti Move, Add, Increment, dan Branch, untuk instruksi operasi yang menyatakan pola kode biner yang sesuai. Pada saat menulis program untuk komputer tertentu, kata-kata tersebut biasanya diganti dengan akronim yang disebut mnemonic, seperti MOV, ADD, INC, dan BR Serupa dengan kita menggunakan notasi R3 untuk mengacu pada register 3, dan LOC untuk mengacu pada lokasi memori. Set lengkap nama simbolik semacam dan aturan penggunaannya membentuk bahasa pemrograman, yang biasanya disebut sebagai bahasa assembly. Set aturan untuk menggunakan mnemonic dalam spesifikasi instruksi dan program lengkap disebut syntax bahasa.

Program yang ditulis dalam bahasa assembly dapat secara otomatis ditranslasikan ke rangkaian instruksi mesin oleh suatu program yang disebut assembler. Program assembler adalah salah satu kumpulan program yang merupakan bagian dari software sistem. Assembler, seperti halnya program yang lain, disimpan sebagai rangkaian instruksi mesin dalam memori komputer. Program user biasanya dimasukkan ke dalam komputer melalui keyboard dan disimpan dalam memori atau disk magnetik. Pada titik ini, program user hanyalah kumpulan baris karakter alfanumerik. Pada saat program assembler dieksekusi, program tersebut membaca program user, menganalisanya, dan kemudian menghasilkan program bahasa mesin yang diinginkan. Bahasa mesin tersebut berisi pola 0 dan 1 yang menetapkan instruksi yang akan dieksekusi oleh komputer tersebut. Program user dalam format teks alfanumerik aslinya disebut source program, dan program bahasa mesin yang di-assemble disebut object program.

Bahasa assembly untuk suatu komputer mungkin case sensitive atau mungkin tidak, sehingga, komputer tersebut bisa membedakan antara huruf kapital dan huruf kecil atau tidak dapat. Kita akan menggunakan huruf kapital mark menunjukkan semua nama dan label dalam contoh kita untuk dapat meningkatkan kemudahan pembacaan teks. Misalnya, kita akan menuliskan instruksi Move sebagai berikut

```
MOVE R0, SUM
```

MOVE mnemonic menyatakan pola biner, atau OP code, untuk operasi yang dilakukan oleh instruksi tersebut. Assembler mentranslasi rnnemonic ini menjadi OP code biner yang dipahatni komputer.

Mnemonic OP-code diikuti oleh setidaknya satu karakter spasi kosong. Kemudian informasi yang menyatakan operand ditetapkan. Dalam contoh kita, source operand berada dalam register R0. Informasi ini diikuti oleh spesifikasi destination operand, dipisah dari source operand dengan koma, tanpa jeda kosong. Destination operand berada dalam lokasi memori yang alamat binernya dinyatakan dengan nama SUM.

Karena terdapat beberapa mode pengalamatan yang dapat digunakan untuk menetapkan lokasi operand, maka bahasa assembly barns mengindikasikan mode mana yang digunakan. Misalnya, nilai numerik atau nama yang digunakannya, seperti SUM pada instruksi sebelumnya, dapat digunakan untuk menunjukkan mode Absolute. Sehingga instruksi

```
ADD #5, R3
```

menambahkan bilangan 5 ke isi register R3 dan meletakkan hasilnya kembali ke register R3. tanda sharp bukanlah cara satu-satunya untuk menunjukkan mode pengalamatan Immediate. Dalam beberapa bahasa assembly, mode pengalamatan yang dimaksud dinyatakan dalam mnemonic OP-code. Dalam hal ini, suatu instruksi memiliki mnonemonic OP-code yang berbeda untuk mode peugalamatan yang berbeda. Misalnya, iustruksi Add sebelumnya dapat ditulis sebagai berikut

```
ADDI 5, R3
```

Akhiran I dalam mnemonic ADDI menyatakan bahwa source operand dinyatakan dalam mode pengalamatan Immediate.

Pengalamatan Indirect biasanya dinyatakan dengan meletakkan tanda kurung di sekitar nama atau simbol yang menunjukkan pointer ke operand. Misalnya, jika nomor 5 ditempatkan dalam lokasi memori yang alamatnya disimpan dalam register R2, maka aksi yang diinginkan dapat ditetapkan sebagai berikut

```
MOVE #5, (R2)
```

atau mungkin

```
MOVE 5, (R2)
```

8.1. ASSEMBLER DIRECTIVE

Selain menyediakan mekanisme untuk menyatakan instruksi dalam suatu program, bahasa assembly memungkinkan programer untuk menetapkan informasi lain yang diperlukan untuk mentranslasikan source program ke dalam object program. Kita telah menyebutkan bahwa kita perlu untuk menetapkan nilai numerik ke tiap nama yang digunakan dalam program. Misalkan nama SUM digunakan untuk menyatakan nilai 200. Fakta ini mungkin disampaikan ke program assembler melalui pernyataan seperti

```
SUM EQU 200
```

Pernyataan ini tidak menunjuk suatu instruksi yang akan dieksekusi pada saat object program dijalankan; sebenarnya, bahkan tidak akan muncul dalam object program. Hanya memberitahu assembler bahwa nama SUM seharusnya digantikan dengan nilai 200 kapanpun muncul dalam program. Pernyataan semacam itu, yang disebut assembler directive (atau perintah), digunakan oleh assembler pada saat mentranslasikan source program menjaadi object program.

	100	Move	N, R1
	104	Move	#NUM1, R2
	108	Clear	R0
LOOP	112	Add	(R2), R0
	116	Add	#4, R2
	120	Decrement	R1
	124	Branch > 0	LOOP
	128	Move	R0, SUM
	132		
		.	
		.	
		.	
SUM	200		
N	204	100	
NUM1	208		
NUM2	212		
		.	
		.	
		.	
NUMn	604		

Gambar 8.1. Pengaturan memori untuk program

Untuk menjalankan program pada komputer, maka kita perlu untuk menuliskan source code-nya dalam bahasa assembly yang diperlukan, menetapkan semua informasi yang diperlukan untuk menghasilkan object program yang sesuai. Misalkan tiap instruksi dan tiap item data memiliki satu word memori. Juga asumsikan bahwa memori tersebut adalah byte addressable dan word length-nya adalah 32 bit. Misalkan juga bahwa object program di-load dalam memori utama sebagaimana yang ditunjukkan dalam Gambar 8.1. Gambar tersebut menujukkan alamat memori dimana instruksi mesin dan item data yang diminta didapatkan setelah program di-load untuk eksekusi. Jika assembler adalah untuk menghasilkan object program sesuai dengan pengaturan ini, maka harus mengetahui

- Bagaimana menginterpretasikan nama tersebut
- Dimana barns menempatkan instruksi tersebut dalam memori
- Dimana harus menempatkan operand data dalam memori

Untuk mendapatkan informasi ini, source program dapat ditulis sebagaimana ditampilkan pada Gambar 8.2. Program dimulai dengan assembler directives. Kita telah membahas direktive Equate, EQU, yang menginformasikan pada assembler tentang nilai SUM. Assembler directive yang kedua, ORIGIN, menyatakan pada program assembler tempat untuk meletakkan blok data berikutnya di memori. Dalam hal ini, lokasi yang ditetapkan memiliki alamat 204. Karena lokasi ini di-load dengan nilai 100 (yang merupakan bilangan entri dalam list), maka directive DATAWORD digunakan untuk menginfonnasikan pada assembler persyaratan ini. Dinyatakan bahwa nilai data 100 ditempatkan dalarn word memori pada alamat 204.

Pernyataan apapun yang menghasilkan instruksi atau data ditempatkan dalam lokasi memori dapat diberi label alamat memori. Label tersebut menetapkan suatu nilai yang setara dengan alamat lokasi tersebut. Karena pernyataan DATAWORD diberi label N, maka nama N ditetapkan dengan nilai 204. Kapanpun N digunakan pada sebagian program, maka akan digantikan dengan nilai tersebut. Menggunakan N sebagai label dengan cara tersebut ekuivalen dengan menggunakan assembler directive

```
N EQU 204
```

Directive RESERVE menyatakan bahwa suatu blok memori 400 byte akan di-reserve untuk data, dan nama NUM1 dihubungkan dengan alamat 208. Directive ini

tidak menyebabkan data apapun di-load ke lokasi tersebut. Data dapat di-load dalam memori menggunakan prosedur input.

	Label alamat memori	Operasi	Pengalamatan atau informasi data
Assembler directives	sum	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that	START	MOVE	N,R1
generate		MOVE	#NUM1,R2
machine		CLR	RO
instructions	LOOP	ADD	(R2),RO
		ADD	q4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	RO,SUM
Assembler directives		RETURN	
		END	START

Gambar 8.2. Representasi bahasa assembly untuk program pada gambar 8.1.

Directive ORIGIN kedua menetapkan bahwa instruksi object program akan di-load dalam memori mulai dari alamat 100. Diikuti dengan instruksi yang ditulis dengan mnemonic dan syntax yang sesuai. Pernyataan terakhir dalam source program adalah assembler directive END, yang menyatakan pada assembler bahwa ini merupakan akhir teks source program. Directive END menyertakan label START, yang merupakan alamat lokasi dimana eksekusi program dimulai.

Kita telah menjelaskan semua pernyataan dalam Gambar 8.2 kecuali RETURN. Ini merupakan assembler directive yang menyatakan titik dimana eksekusi program harus dihentikan. Pernyataan ini menyebabkan assembler menyisipkan suatu instruksi mesin yang sesuai yang mengembalikan kontrol ke sistem operasi komputer tersebut.

Kebanyakan bahasa assembly meminta pernyataan dalam source program dituliskan dalam bentuk

```
Label          Operation      Operand        Comment
```

Empat field tersebut dipisahkan oleh delimiter yang sesuai, biasanya satu atau lebih karakter kosong. Label adalah nama opsional yang dihubungkan dengan alamat memori dimana instruksi bahasa mesin yang dihasilkan dari pernyataan tersebut akan di-load. Label juga dapat dihubungkan denagn alamat item data. Pada Gambar 8.2 terdapat lima label: SUM, N, NUM1, START, dan LOOP.

Field Operation berisi mnemonic OP-code dari instruksi yang dimaksud atau directive assembler. Field Operand berisi informasi pengalamatan untuk mengakses satu atau dua operand, tergantung pada tipe informasinya. Field Comment diabaikan oleh program assembly. Field tersebut digunakan untuk tujuan dokumentasi sehingga program lebih mudah dipahami.

Kita hanya telah memperkenalkan karakteristik bahasa assembly yang sangat dasar. Bahasa ini berbeda detil dan kompleksitasnya dari satu komputer ke komputer lain.

8.2. ASSEMBLY DAN EKSEKUSI PROGRAM

Source program yang ditulis dalam bahasa assembly harus di-assemble menjadi object program bahasa mesin sebelum dapat dieksekusi. Hal ini dilakukan oleh program assembler, yang mengganti semua simbol untuk mode operasi dan penggalamatan dengan kode biner yang digunakaa dalam instruksi mesin, dan mengganti semua nama dan label dengan nilai sebenarnya.

Assembler menetapkan alamat untuk instruksi dan blok data, mulai dari alamat yang ada dalam asembler directive ORIGIN. Juga menyisipkan konstanta yang dapat dinyatakan dalam perintah DATAWORD dan ruang memori cadangan sebagaimana yang diminta oleh perintah RESERVE.

Bagian utama proses assembly menetapkan nilai-nilai untuk menggantikan nama-nama tersebut. Pada beberapa kasus, dimana nilai suatu nama ditetapkan oleh directive EQU, maka ini merupakan tugas langsung. Pada kasus lain, dimana suatu nama didefinisikan dalam field Label suatu instruksi, maka nilai yang diwakili nama ini ditentukan dengan lokasi instruksi ini dalam object program ter-assemble. Karenanya, assembler barns mencatat alamat-alamat yang menghasilkan kode mesin untuk instruksi yang berurutan. Misalnya, nama START dan LOOP masing-masing akan menetapkan nilai 100 dan 112.

Pada beberapa kasus, assembler tidak secara langsung mengganti nama yang mewakili alamat dengan nilai yang sebenarnya dari alamat ini. Misalnya, pada instruksi branch, nama yang menetapkan lokasi dimana branch dilakukan (branch target) tidak diganti dengan alamal yang sebenarnya. Instruksi branch biasanya diimplementasikan dalam kode mesin dengan menetapkan branch target menggunakan mode penggalamatan Relative. Assembler menghitung branch offset, yaitu jarak ke target, clan meletakkannya dalam instruksi mesin.

Pada saat assembler memindai source program, assembler akan mencatat semua nama dan nilai numerik yang berhubungan dengannya dalam tabel simbol. Sehingga, pada saat nama tersebut muncul untuk kedua kalinya, akan digantikan dengan nilainya dari tabel tersebut. Persoalan muncul pada saat strain nama muncul sebagai operand sebelum mendapatkan nilainya. Misalnya, hal ini terjadi jika diperlukan suatu forward branch. Assembler tidak akan mampu menetapkan branch target, karena nama yang dimaksud belum direkam dalam tabel symbol. Pada akhir lewatan ini, semua nama harus telah ditetapkan dengan nilai numerik. Assembler kemudian memasuki source program untuk kedua kalinya dan mensubstitusi nilai untuk semua nama dari tabel simbol. Assembler tersebut disebut two-pass assembler.

Assembler tersebut menyimpan object program pada disk magnetik. Object program harus di-load ke dalam memori komputer sebelum dieksekusi. Pada saat hal ini berjalan, program utiliti lain yang disebut loader harus telah tersedia dalam memori. Mengeksekusi loader adalah menjalankan serangkaian operasi input yang diperlukan untuk mentransfer program bahasa mesin dari disk ke tempat tertentu dalam memori. Loader harus mengetahui panjang program dan alamat dalam memori yang akan digunakan untuk menyimpannya. Assembler biasanya menyimpan informasi ini pada header sebelum kode objek. Setelah me-load kode objek, loader mulai mengeksekusi object program dengan melakukan branching ke instruksi pertama yang akan dieksekusi. Ingatlah bahwa alamat instruksi tersebut telah disertakan dalam program bahasa assembly sebagai operand assembler directive END. Assembler menyertakan alamat ini di dalarn header yang mendahului kode objek pada disk.

Pada saat object program mulai mengeksekusi, akan diselesaikan hingga akhir kecualijika terdapat kesalahan logika dalam program tersebut. User harus mampu

menemukan kesalahan tersebut dengan mudah. Assembler dapat mendeteksi clan melaporkan kesalahan syntax. Unmk membantu user menemukan kesalahan pemrograman yang lain, software sistem biasanya menyertakan program debugger. Program ini memungkinkan user menghentikan eksekusi object program pada suatu titik yang diinginkan dan memeriksa berbagai register prosesor dan lokasi memori.

8.3. NOTAS1 BILANGAN

Pada saat berhadapan dengan nilai numerik, seringkali lebih mudah untuk menggunakan notasi desimal yang telah dikenal. Tentu saja, nilai tersebut disimpan dalam komputer sebagai bilangan biner. Pada beberapa situasi, lebih mudah untuk menetapkan pola biner secara langsung.

Kebanyakan assembler memungkinkan bilangan numerik dinyatakan dengan berbagai cara yang berbeda, menggunakan konvensi yang ditetapkan oleh syntax bahasa assembly. Misalkan, bilangan 93, yang dinyatakan dengan bilangan biner 8-bit 01011101. Jika nilai ini digunakan sebagai operand Immediate, maka dapat dinyatakan sebagai bilangan desimal, sebagaimana dalam instruksi

```
ADD #93, RI
```

atau sebagai bilangan bitter yang diidentifikasi dengan simbol awalan seperti tanda persen, sebagaimana dalam

```
ADD #%01011101, R1
```

Bilangan biner dapat dituliskan lebih padat sebagai bilangan heksadesimal, atau hex, dengan empat bit dinyatakan dengan digit hex tunggal. Notasi hex adalah ekstensi langsung dari kode BCD yang terdapat dalam Apendiks E. Sepuluh pola pertama 0000, 0001, ..., 1001, dinyatakan dengan digit 0, 1, ..., 9 sebagaimana dalam BCD. Sisa enam pola 4-bit, 1010, 1011, ..., I111, dinyatakan dengan huruf A, B, ..., F. Dalam representasi heksadesimal, nilai desimal 93 menjadi SD. Dalam bahasa assembly, representasi hex seringkali diidentifikasi dengan awalan tanda dolar. Sehingga kita menuliskannya

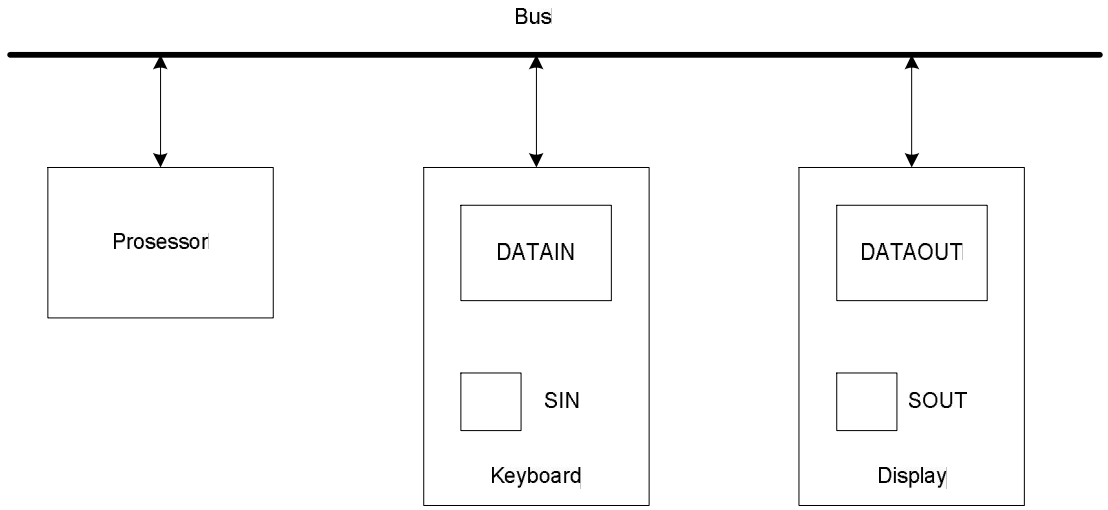
```
ADD       #$SD, R1
```

8.4. OPERASI INPUT/OUTPUT DASAR

Bagian sebelumnya dalam bab ini mendeskripsikan instruksi mesin dan mode pengalamatan. Kita telah mengasumsikan bahwa data yang dikenai operasi instruksi

ini telah disimpan dalam memori. Kita sekarang membahas sarana yang digunakan untuk mentransfer data antara memori komputer dan dunia luar. Operasi Input/Output (I/O) sangat penting, dan cara operasi tersebut dijalankan dapat memiliki efek yang signifikan pada performa komputer.

Misalkan suatu tugas untuk membaca input karakter dari keyboard dan menghasilkan output karakter pada layar display. Cara sederhana untuk menjalankan tugas I/O tersebut adalah dengan menggunakan metode yang dikenal sebagai program-controlled I/O. Kecepatan transfer data dari keyboard ke komputer dibatasi oleh kecepatan mengetik user, yang tampaknya tidak melebihi beberapa karakter per detik. Kecepatan transfer output dari komputer untuk ditampilkan jauh lebih tinggi. Hal ini ditentukan oleh kecepatan karakter ditransmisikan melalui link antara komputer dan perangkat display, biasanya, beberapa ribu karakter per dedtik. Akan tetapi ini masih jauh lebih lambat daripada kecepatan prosesor yang dapat mengeksekusi jutaan instruksi per detik. Perbedaan kecepatan antara prosesor dan perangkat I/O menimbulkan kebutuhan akan adanya mekanisme untuk mensinkronisasikan transfer data diantara keduanya.



Gambar 8.3 Koneksi bus untuk prosesar, keyboard, dan display.

Solusi untuk persoalan tersebut adalah sebagai berikut: pada output, prosesor mengirim karakter pertama dan kemudian menunggu sinyal dari display bahwa karakter telah diterima.

Kemudian mengirim karakter kedua, dan seterusnya. Input dikirim dari keyboard dengan cara yang sama; prosesor menunggu sinyal yang mengindikasikan bahwa suatu tombol karakter telah ditekan dan kodenya tersedia dalam beberapa register

buffer yang diasosiasikan dengan keyboard. Kemudian prosesor membaca kode tersebut.

Keyboard dan display adalah perangkat yang terpisah. Tindakan menekan suatu tombol pada keyboard tidak secara otomatis menyebabkan karakter yang sesuai ditampilkan pada layar. Satu blok instruksi dalam program I/O mentransfer karakter tersebut ke prosesor, dan blok lain yang berhubungan dengan instruksi tersebut menyebabkan ditampilkannya karakter tersebut.

Misalkan suatu persoalan pemindahan kode karakter dari keyboard ke prosesor. Menekan suatu tombol akan menyimpan kode karakter yang sesuai dalam register buffer 8-bit yang sesuai dengan keyboard. Mari kita sebut register ini DATAIN. Untuk memberitahu prosesor bahwa karakter yang valid berada dalam DATAIN, suatu status control flag, SIN, di-set ke 1. Suatu program memonitor SIN, dan saat SIN di-set ke 1, prosesor membaca isi DATAIN. Pada saat karakter ditransfer ke prosesor, maka SIN secar otomatis dikosongkan ke 0. Jika karakter kedua dimasukkan melalui keyboard, maka SIN diset lagi ke 1 dan proses tersebut diulang.

Proses yang analog terjadi pada saat karakter ditransfer dari prosesor ke display. Register buffer, DATAOUT, dan status control flag, SOUT, digunakan untuk transfer ini. Pada saat SOUT setara dengan I, maka display siap untuk menerima suatu karakter. Di bawah kontrol program, prosesor memonitor SOUT, dan pada saat SOUT diset ke 1, prosesor mentransfer kode karaktei ke DATAOUT. Transfer karakter ke DATAOUT mengosongkan SOUT ke 0; pada saat perangkat display siap untuk menerima karakter kedua, maka SOUT sekali lagi di set ke 1. Register buffer DATAIN dan DATAOUT dan flag status SIN dan SOUT adalah bagian dari sirkuit yang biasanya dikenal sebagai device interface. Sirkuit untuk tiap perangkat dihubungkan ke prosessor melalui bus, sebagaimana yang diperlihatkan pada Gambar 2.19.

Untuk menjalankan transfer I/O, kita memerlukan instruksi mesin yang dapat memeriksa keadaan flag status dan transfer data antara prosesor dan perangkat I/O. Instruksi tersebut memiliki kemiripan format dengan yang digunakan untuk memindahkan data antara prosesor dan memori. Misalnya, prosesor dapat memonitor flag status keyboard SIN dan transfer karakter dari DATAIN ke register Rl dengan rangkaian operasi sebagai berikut:

```
READWAIT        Branch to READMIT if SIN = 0
                Input from DATAIN to R1
```

Operasi Branch biasanya diimplementasikan oleh dua instruksi mesin. Instruksi pertama menguji flag status dan yang kedua menjalankan branch. Sekalipun detilnya bervariasi dan satu komputer ke komputer lain, ide utamanya adalah prosesor memonitor flag status dengan mengeksekusi wait loop pendek dan melanjutkan untuk mentransfer data input pada saat SIN diset ke 1 sebagai hasil dari adanya suatu tombol yang ditekan. Operasi Input me-reset SIN ke 0.

Rangkaian yang analog dari operasi tersebut digunakan untuk mentransfer output ke dis play. Contohnya adalah

```
WRITEWAIT Branch to WRITEWAIT if SOUT = 0
          Output from R1  to DATAOUT
```

Lagi, operasi Branch biasanya diimplementasikan oleh dua instruksi mesin. Wait loop dieksekusi berulangkali hingga flag status SOUT di-set ke 1 oleh display pada saat display bebas untuk menerima suatu karakter. Operasi Output mentransfer suatu karakter dari RI ke DATAOUi untuk ditampilkan, dan operasi tersebut mengosongkan SOUT ke 0.

Kita mengasumsikan bahwa keadaan awal SIN adalah 0 dan keadaan awal SOUT adalah 1. Awalan ini biasanya dilakukan oleh sirkuit kontrol perangkat pada saat perangkat ditempatkan di bawah kontrol komputer sebelum eksekusi program dimulai.

Hingga sekarang, kita telah mengasumsikan bahwa alamat yang dinyatakan oleh prosesor untuk untuk mengakses iustruksi dan operand selalu mengacu ke lokasi memori. Banyak komputer menggunakan pengaturan yang disebut memory-mapped I/O dimana beberapa nilai alamat memori digunakan untuk mengacu ke register buffer perangkat periferal, seperti DATAIN dan DATAOUT. Sehingga, tidak ada instruksi khusus untuk mengakses isi register tersebut; data dapat ditransfer antara register dan prosesor menggunakan instruksi yang telah kita bahas, seperti Move, Load, atau Store. Misalnya, isi buffer karakter keyboard DATAIN dapat ditransfer ke register R1 dalam prosesor dengan instruksi

```
MoveByte DATAIN, RI
```

Serupa dengan isi register RI dapat ditransfer ke DATAOUT dengan instruksi

```
MoveByte R1, DATAOUT
```

Flag status SIN dan SOUT secara otomatis dikosongkan pada saat masing-masing mengacu pada register buffer DATAIN dan DATAOUT. Kode operasi MoveByte menunjukkan bahwa ukuran operand adalah satu byte, untuk membedakannya dari kode operasi Move yang telah digunakan untuk operand word. Kita telah menetapkan bahwa dua buffer data pada Gambar 2.19 dapat diberi alamat seakan keduanya adalah dua lokasi memori. Sangat dimungkinkan untuk menangani flag status SIN dan SOUT dengan cara yang sama, dengan menetapkan alamat yang berbeda padanya. Akan tetapi, lebih umum untuk menyertakan SIN dan SOUT pada register device status, satu untuk tiap dua perangkat. Mari kita asumsikan bahwa bit b_3 dalam register INSTATUS dan OUTSTATUS masing-masing berhubungan dengan SIN dan SOUT. Operasi baca yang barn saja dideskripsikan dapat diimplementasikan dengan rangkaian instruksi mesin

```
READWAIT Testbit    #3,INSTATUS
Branch=0            READWAIT
                    MoveByte DATAIN, R1
```

Operasi tulis dapat diimplementasikan sebagai berikut

```
WRITEWAIT Testbit   #3, OUTSTATUS
Branch=0            WRITEWAIT
                    MoveByte RI, DATAOUT
```

Instruksi Testbit menguji keadaan satu bit pada lokasi destinasi, dimana posisi bit yang diuji diindikasikan oleh operand pertama. Jika bit yang diuji setara dengan 0, maka kondisi instruksi branch adalah benar, dan suatu branch dibuat pada awal wait loop. Pada saat perangkat tersebut siap, yaitu pada saat bit yang diuji menjadi setara dengan 1, data dibaca dari buffer input atau ditulis ke buffer output.

Program yang ditampilkan pada Gambar 8.4 menggunakan dua operasi untuk membaca baris karakter yang diketikkan pada keyboard dan dikirim ke parangkat display. Pada saat karakter tersebut dibaca, satu demi satu, karakter tersebut disimpan pada area data dalam memori dan kemudian ditampilkan pada display. Program berakhir pada saat karakter carriage return, CR, dibaca, disimpan, dan dikirim ke display. Alamat lokasi byte pertama dari memori data area tempat baris

tersebut disimpan adalah LOC dengan instruksi pertama dari program tersebut. RO dinaikkan untuk tiap karakter yang dibaca dan ditampilkan oleh mode pengalamatan Autoincrement yang digunakan dalam instruksi Compare.

	Move	#LOC,RO	Initialize pointer register RO to point to the address of the first location in memory where the characters are to be stored.
READ	TestBit	#3,INSTATUS	Wait for a character to be entered in the keyboard buffer DATAIN.
	Branch=0	READ	
	MoveByte	DATAIN,(RO)	Transfer the character from DATAIN. into the memory (this clears SIN to 0).
ECHO	TestBit	#3,OUTSTATUS	Wait for the display to become ready.
	Branch=0	ECHO	
	MoveByte	(RO),DATAOUT	Move the character just read to the display buffer register (this clears SOUT to 0).
	Compare	#CR,(RO)+	Check if the character just read is CR (carriage return). If it is not CR, then
	Branch≠0	READ	branch back and read another character. Also, increment the pointer to store the next character.

Gambar 8.4. Program yang membaca sebaris karakter dan menampilkannya

Program-controlled IO memerlukan keterlibatan kontinyu prosesor dalam aktifitas l/0. Hampir semua waktu eksekusi untuk program dalam Gambar 8.4 dihitung dalam dua wait loop, pada saat prosesor menunggu karakter yang akan ditekan atau menunggu display tersedia. Sangat diinginkan untuk menghindari terbuangnya waktu eksekusi prosesor pada situasi ini. Teknik I/O yang lain, berdasar pada penggunaan interrupt, dapat digunakan untuk meningkatkan utilisasi prosesor.

8.5. STACK DAN QUEU

Program komputer seringkali memerlukan subtask tertentu yang menggunakan struktur subroutine yang umum. Untuk mengatur hubungan kontrol dan infonnasi antara program utama dan subrotine, maka digunakan suatu struktur data yang disebut stack. Bagian ini akan mendeskripsikan stack, dan struktur data yang berhubungan erat dengannya yang disebut queu.

Data yang dikenai operasi oleh suatu program dapat diatur dengan berbagai cara. Kita telah menangani struktur data sebagai list. Sekarang, kita membahas struktur data penting yang dikenal sebagai stack. Stack adalah list element data, biasanya word atau byte, dengan batasan akses yaitu elemen hanya dapat ditambahkan

atau dihapus pada satu ujung list. Ujung ini disebut puncak stack, dan ujung yang lain disebut dasar. Struktur tersebut terkadang disebut sebagai pushdown stack. Bayangkan tumpukan baki di kafetaria; konsumen mengambil baki baru dari puncak tumpukan, dan baki bersih ditambahkan ke tumpukan tersebut dengan meletakkannya di puncak tumpukan. Frase lain yang deskriptif, stack last-in-first-out (LIFO), juga digunakan untuk mendeskripsikan tipe mekanisme penyimpanan ini; item data terakhir yang diletakkan pada stack tersebut adalah yang pertama kali diambil pada saat pengambilan dimulai. Istilah push dan pop masing-masing digunakan untuk mendeskripsikan peletakkan item baru pada stack dan pemindahkan item paling atas dari stack.

Data yang disimpan dalam memori komputer dapat diatur sebagai stack, dengan elemen yang berurutan memiliki lokasi memori yang berurutan. Asumsikanlah elemen pertama diletakkan pada lokasi BOTTOM, dan pada saat elemen baru di-push ke stack, maka akan ditempatkan pada urutan lokasi alamat yang lebih rendah. Kita menggunakan stack yang berkembang pada arah alamat memori berkurang dalam diskusi kita, karena ini adalah praktek yang umum.

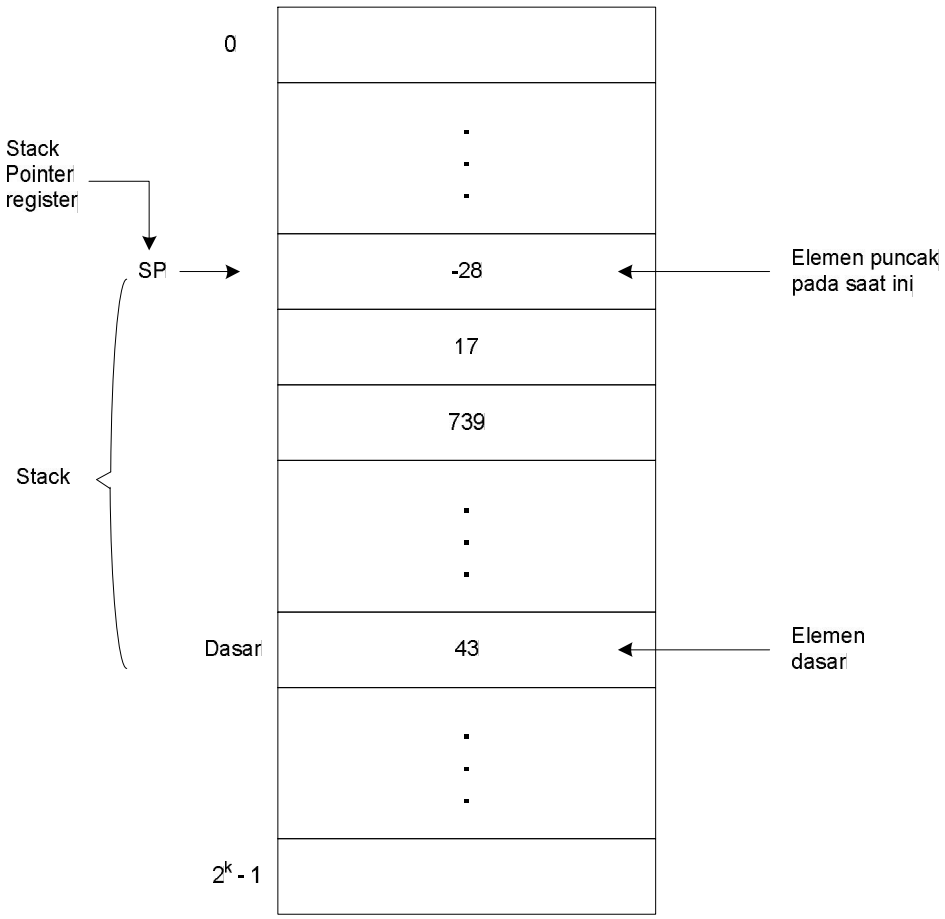
Gambar 8.5 menunjukkan stack item data word dalam memori komputer. Stack tersebut berisi nilai numerik, dengan 43 pada dasar dan -28 pada puncak. Register prosesor digunakan untuk mencatat alamat elemen stack yang berada di puncak pada tiap waktu. Register ini disebut stack pointer (SP). Register tersebut dapat berupa salah satu dari general-purpose register atau register yang didedikasikan untuk fungsi ini. Jika kita mengasumsikan suatu memori byteaddressable dengan word length 32-bit, maka operasi push dapat diimplementasikan sebagai berikut

```
Subtract #4, SP
Move      NEWITEM, (SP)
```

dimana instruksi Subtract mengurangi source operand 4 dari destination operand yang terdapat dalam SP dan meletakkan hasilnya dalam SP. Dua instruksi ini memindahkan word dari lokasi NEWITEM ke puttcak stack, menurunkan stack pointer sebesar 4 sebelum pemindahan. Operasi pop dapat diimplementasikan sebagai berikut

```
Move (SP), ITEM
Add #4, SP
```

Dua instruksi tersebut memindahkan nilai puncak dari stack ke lokasi ITEM dan kemudian menaikkan stack pointer sebesar 4 sehingga pointer tersebut menunjuk ke elemen puncak yang baru. Gambar 8.6 menunjukkan efek tiap operasi tersebut pada stack dalam Gambar 8.5.



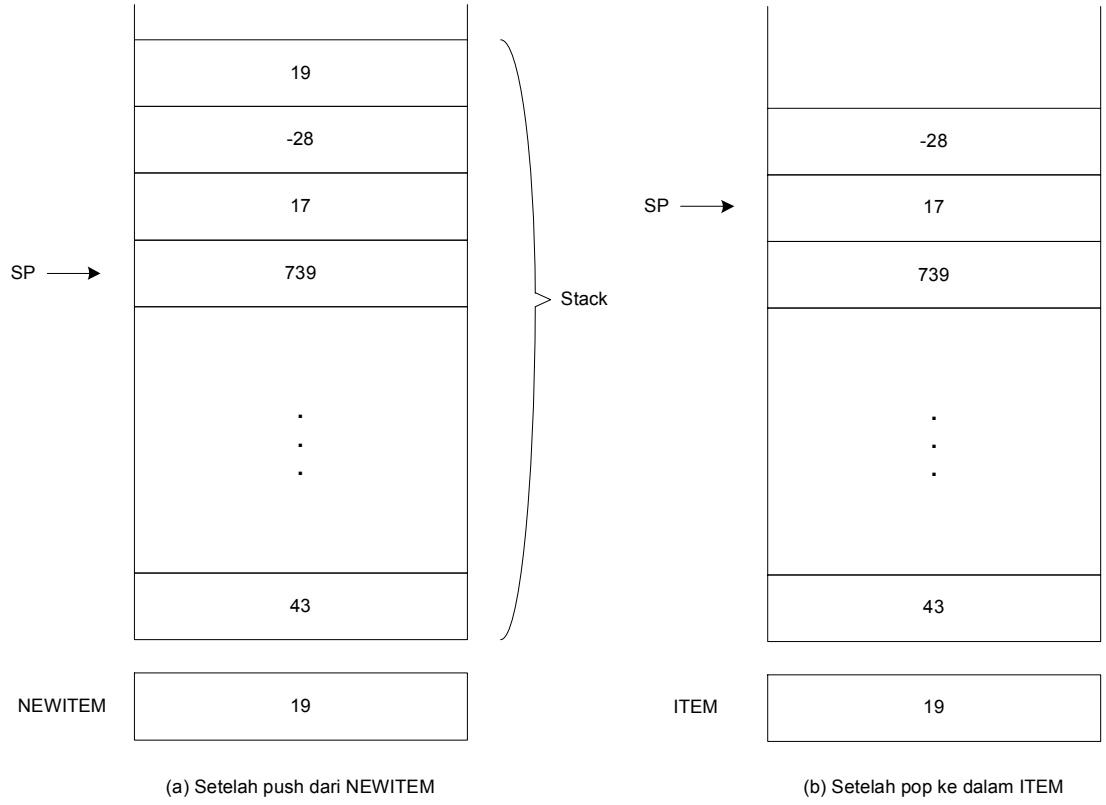
Gambar 8.5 Stack word dalam memori.

Jika prosesor memiliki mode pengalamatan Autodecrement dan Autoincrement, maka operasi push dapat dilakukan dengan instruksi tunggal

```
Move        NEWITEM, -(SP)
```

dan operasi pop dapat dilakukan sebagai berikut

```
Move (SP)+, ITEM
```



Gambar 8.6. Efek operasi stack yang terdapat pada gambar 8.5.

Pada saat suatu stack digunakan dalam program, maka biasanya dialokasikan sejumlah ruang tertentu di dalam memori. Dalam hal ini, kita sebaiknya tidak mem-pop off item pada stack kosong, yang dapat mengakibatkan kesalahan pemrograman. Misalkan suatu stack dijalankan dari lokasi 2000 (BOTTOM) turun hingga tidak lebih dari lokasi 1500. Stack pointer di-load pada awalnnya dengan nilai alamat 2004. Ingatlah bahwa SP diturunkan sebesar 4 sebelum data baru disimpan pada stack. Karenanya, nilai awal 2004 berarti bahwa item pertama yang di-push ke dalam stack akan berada pada lokasi 2000. Untuk mencegah pushing item pada full stack atau popping off item pada empty stack, maka operasi push dan pop tunggal dapat digantikan dengan rangkaian instruksi yang ditampilkan pada Gambar 2.23.

Instruksi Compare

```
Compare          src, dst
```

menjalankan operasi

```
[dst] – [src]
```

dan men-set condition code flag msesuai dengan hasil tersebut. Tidak mengubah nilai salah satu operand tersebut.

SAFEPOP	Compare	#2000,SP	Check to see if the stack pointer
	Branch>0	EMPTYERR	contains an address value
		OR	greater than 2000. If it does, the
			stack is empty. Branch to the
			routine EMPTYERROR for
	Move		appropriate action.
		(SP)+,ITEM	Otherwise, pop the top of the
			stack into memory location
			ITEM.

(a) Routine untuk operasi safe pop

SAFEPUSH	Compare	#1500, SP	Check to see if the stack pointer
	Branch ≤ 0	FULLERROR	contains an address value equal to
			or less than 1500. If it does, the
			stack is full. Branc to the routin
			FULLERROR for appropriate
	Move	NEWITEM, -	action
		(SP)	Otherwise, push the element in
			memory location NEWITEM on
			to the stack

(b) Routine untuk operasi safe push

Gambar 8.7. Mengecek kesalahan empty dan full pada operasi pop dan push

Struktur data lain yang mirip dengan stack disebut queu. Data disimpan clan diarnbil dari queu pada basis first-in-first-out (FIFO). Sehingga, jika kita mengasumsikan bahwa queu berkembang dengan arah peningkatan alamat dalam memori, yang merupakan praktek umum, data baru ditambahkan pada bagian belakang (ujung beralamat-tinggi) dan diambil dari bagian depan (ujung beralamat-rendah) dari queu.

Terdapat perbedaan yang umum antara bagaimana stack dan queu diimplementasikan. Satu ujung dari stack tersebut tetap (bagian dasar), sedangkan ujung yang lain naik dan turun pada saat data di-push dan pop. Sebaliknya, kedua ujung queu bergerak ke alamat yang lebih tinggi pada saat data ditambahkan pada bagian belakang dan diambil dari bagian depan. Sehingga diperlukan dua pointer untuk mcncatat dua aping queu.

Perbedaan lain antara stack dan queu adalah,tanpa control lebih lanjut suatu queu akan terus bergerak melalui memori kontputer ke arah alamat yang lebih tinggi. Satu cara untuk membatasi queu pada wilayah tertentu dalam memori adalah dengan menggunakan circular buffer. Marilah kita asumsikan hahwa alamat memori dari BEGINNING ke END ditetapkan pada queu. Entri pertama pada queu dimasukkan ke dalam lokasi BEGINNING, dan entri yang berurutan di ditambahkan ke queu dengan memasukkannya pada alamat-alamat yang lebih tinggi secara berurutan. Pada saat bagian belakang queu mencapai END, suatu ruang akan tercipta pada bagain awal apabila beberapa item dihilangkan dari queu. Karenanya, back pointer di-reset ke nilai BEGINNING dan proses berlanjut. Sedangkan pada stack, diperlukan perhatian untuk mendeteksi kapan daerah yang ditetapkan untuk struktur data tersebut sepenuhnya lengkap atau sepenuhnya kosong.

8.6. SUBROUTINE

Pada suatu program, seringkali perlu untuk melakukan subtask tertentu berulangkali pada nilai data yang berbeda. Subtask semacam itu biasanya disebut subroutine. Misalnya, suatu subroutine dapat mengevaluasi fungsi sinus atau mensortir suatu list nilai menjadi urutan meningkat atau menurun.

Sangat dimungkinkan untuk menyertakan blok instruksi yang terdiri dari subroutine pada setiap tempat yang memerlukan program tersebut. Akan tetapi untuk menghemat ruang, hanya satu copy dari instruksi yang merupakan subroutine ditempatkan dalam memori, tiap program yang perlu menggunakan subroutine tersebut hanya perlu branch ke lokasi mulainya. Pada saat sualu program branch ke suatu subroutine kita katakan program tersebut memanggil subroutine. Instruksi yang melakukan operasi branch ini disebut instruksi Call.

Setelah subroutine dieksekusi, calling program hams mcresume eksekusi, melanjutkan dengan segera setelah instruksi yang memanggil subroutine tersebut. Subroutine kembali ke program yang memanggilnya dengan mengeksekusi instruksi Return. Karena subroutine dapat dipanggil dari tempat yang berbeda dalam calling program, maka harus dibuat ketetapan untuk kembali ke lokasi yang sesuai. Lokasi dimana calling program meresume eksekusi adalah lokasi yang ditunjuk oleh PC ter-update pada saat instruksi Call sedang dieksekusi. Karenanya, isi PC harus disimpan oleh instruksi Call tmtuk memungkinkan return yang tepat ke calling program.

Cara computer memungkinkan call dan return dari subroutine disebut metode linkage subroutine. Metode linkage subroutine yang paling sederhana adalah untuk menyimpan return address dalam lokasi tertentu, yang dapat berupa suatu register yang didedikasikan untuk fungsi ini. Register semacam itu disebut link register. Pada saat subroutine menyelesaikan tugasnya, instruksi Return kembali ke calling program dengan branching secara tidak langsung melalui link register.

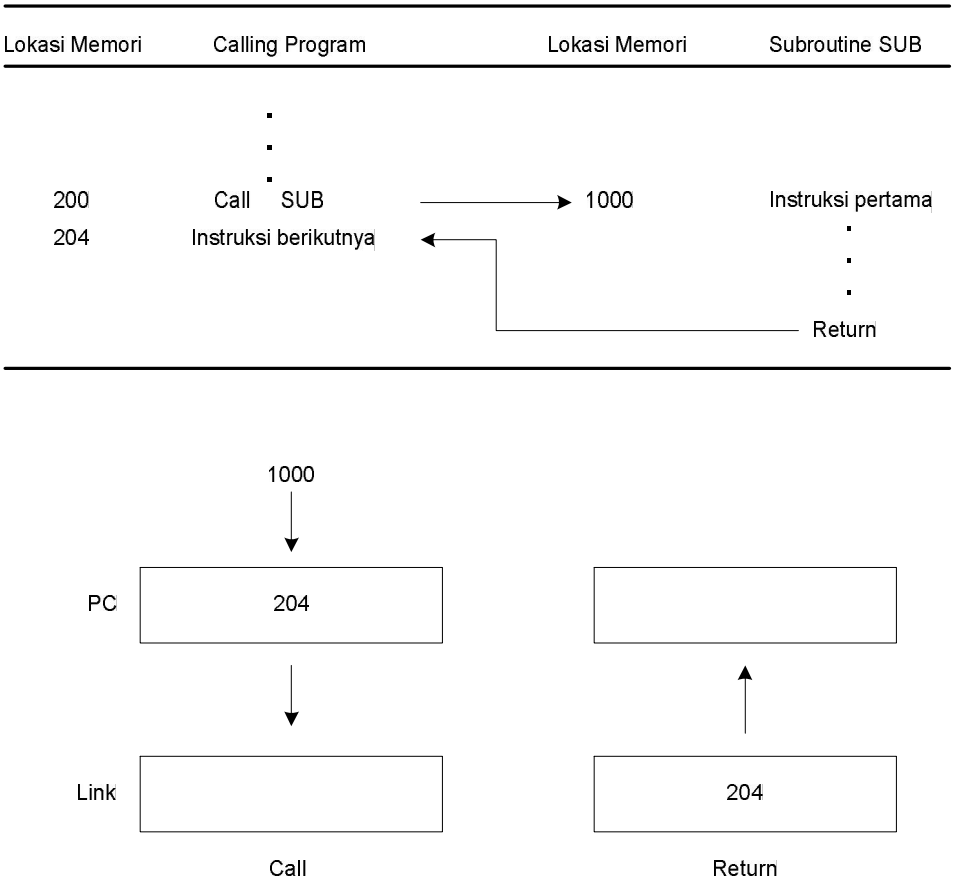
Instruksi Call hanyalah instruksi branch khusus yang melakukan operasi berikut:

- Menyimpan isi PC dalam link register
- Branch ke alamat target yang ditetapkan oleh instruksi

Instruksi Return adalah instruksi branch khusus yang melakukan operasi berikut:

- Branch ke alamat yang terdapat dalam link register

Gambar 8.8. mengilustrasikan prosedur ini.



Gambar 8.8. Linkage subroutine menggunakan register link.

8.7. SUBROUTINE NESTING DAN STACK PROSESOR

Praktek pemrograman umum, yang disebut subroutine nesting, adalah menggunakan satu sub routine untuk memanggil subroutine lain. Dalam hat ini, return address call yang kedua juga disimpan dalam link register, menghancurkan isi sebelumnya. Karenanya, sangat penting untuk menyimpan isi link register dalam lokasi lain sebelum memanggil subroutine lain. lika tidak return address subroutine pertama akan hilang.

Subroutine nesting dapat dilakukan hingga kedalaman berapapun. Pada akhirnya, subroutine terakhir dinyatakan menyelesaikan komputasinya dan kembali ke subroutine yang memanggilnya. Return address yang diperlukan untuk return pertama ini adalah yang terakhir yang dihasilkan dari rangkaian nested call. Sehingga, return address dihasilkan dan digunakan dalam urutan last-in-first-out. Hal ini menyatakan bahwa return address yang diasosiasikan dengan panggilan subroutine sebaiknya dapat di-push ke stack. Banyak prosesor menjalankan hal ini secara otomatis sebagai salah satu operasi yang dilakukan oleh instruksi Call. Register tertentu ditetapkan sebagai stack pointer, SP, untuk digunakan dalam operasi ini. Stack pointer menunjuk ke suatu stack yang disebut stack prosesor. Instruksi Call push isi PC ke stack prosessor dan me-load alamat subroutine ke PC. Instruksi return pop return address dari stack prosessor ke dalam PC.

8.8. PARAMETER PASSING

Pada saat calling suatu subroutine, suatu program harus menyediakan parameter ke subroutine, yaitu operand atau alamatnya, yang akan digunakan dalam komputasi. Selanjutnya, subroutine mengembalikan parameter lain, dalam hal ini, hasil komputasi tersebut. Pertukaran informasi antara calling program dan subroutine disebut sebagai parameter passing. Parameter passing dapat dilakukan dengan beberapa cara. Parameter tersebut dapat ditempatkan dalam register atau dalam lokasi memori, sehingga dapat diakses oleh subroutine. Atau alternatif lainnya, parameter tersebut dapat ditempatkan pada stack prosesor yang digunakan untuk menyimpan return address.

Parameter passing melalui register prosesor adalah langsung dan efisien. Gambar 8.9 menunjukkan bagaimana program yang digunakan untuk penambahan sejumlah bilangan dapat diimplementasikan sebagai subroutine, dengan parameter dilewatkan melalui register. Ukuran list tersebut, n, dimuat dalam lokasi memori

N, dan alamat NUM I dari bilangan pertama dilewatkan melalui register R 1 dan R2. Jumlah yang dihitung oleh subroutine dilewatkan kembali ke calling program melalui register R0. Empat instruksi pertama dalam Gambar 8.9 terdiri dari bagian yang relevan dari calling program. Dua instruksi pertama meload n dan NUM 1 ke dalam RI dan R2. Instruksi Call branch ke subroutine mulai dari lokasi LISTADD. Instruksi ini juga push return address ke stack prosesor. Subroutine menghitung jumlah dan meletakkannya dalam R0. Setelah operasi return dilakukan oleh subroutine, maka jumlah disimpan dalam lokasi memori SUM oleh calling program.

Calling program

```
                Move        N,Ri        Rl serves as a counter.
                Move        #NUM1,R2    R2 points to the list.
                Call        LIST ADD    Call subroutine.
                Move        RO,SUM      Save result.

                .
                .
                .
```

Subroutine

```
LIST ADD        Clear       RO          Initialize sum to 0.
LOOP            Add         (R2)+,RO    Add entry from list.
                Decrement   Rl
                Branch>0    LOOP
                Return                  Return to calling program.
```

Program Gambar 8.9. ditulis sebagai subroutine; parameter dilewatkan melalui register

Jika banyak parameter terlibat, maka mungkin jumlah general-purpose register untuk melewatkannya ke subroutine tidak cukup. Sebaliknya dengan menggunakan stack akan sangat fleksibel; stack dapat menangani sejumlah besar parameter. Contoh berikut mengilustrasikan pendekatan ini. Gambar 8.10a menampilkan program ditulis sebagai subroutine, LISTADD, yang dapat dipanggil oleh program lain untuk menambahkan suatu list bilangan. Parameter yang dilewatkan ke subroutine ini adalah alamat bilangan pertama dalam list dan bilangan entri. Subroutine melakukan penambahan dan mengembalikan hasil yang telah terhitung. Parameter di-push ke

stack prosesor yang ditunjuk oleh register SP Asumsikan bahwa sebelum subroutine dipanggil, puncak stack berada pada level 1 pada Gambar 8.10b. Calling program push alamat NUM 1 dan nilai n ke stack. Puncak stack sekarang berada pada level 2. Subroutine menggunakan tiga register. Karena register tersebut dapat berisi data valid yang merupakan bagian calling program, maka isinya harus disimpan dengan push ke stack. Kita telah menggunakan instruksi tunggal, MoveMultiple, untuk menyimpan isi register RO da R2 pada stack. Banyak prosesor memiliki instruksi semacam itu. Puncak stack sekarang berada pada level 3. Subroutine mengakses parameter n dan NUM 1 dari stack menggunakan pengalamatan ter-index. Perhatikanlah bahwa hal ini tidak mengubah stack pointer karena item data valid masih berada di puncak stack. Nilai n di-load ke dalam RI menggunakan nilai awal perhitungan, dan alamat NUMl di-load ke R2, yang digunakan sebagai pointer untuk memindai entri list. Pada akhir komputasi, register RO berisi jumlah tersebut. Sebelum subroutine tersebut kembali ke calling program, isi RO diletakkan pada stack, menggantikan parameter NUMI, yang tidak lagi diperlukan. Kemudian isi tiga register tersebut yang digunakan oleh subroutine dipulihkan dari stack. Sekarang item puncak pada stack adalah return address pada level 2. Setelah subroutine kembali, maka calling program menyimpan hasilnya dalam lokasi SUM dan menurunkan puncak stack ke level awalnya dengan menaikkan SP sebesar 8.

8.9. PARAMETER PASSING BY VALUE DAN BY REFERENCE

Perhatikanlah sifat dua parameter NUMl dan n, yang dilewatkan ke subroutine pada Gambar 8.9 dan 8.10. Tujuan subroutine tersebut adalah menambahkan suatu list bilangan. Bukannya melewatkan entri list sebenarnya, calling program melewatkan alamat bilangan pertama dalam list. Teknik ini disebut passing by reference. Parameter yang kedua passed by value, yaitu bilangan entri aktual, n, dilewatkan ke subroutine.

8.10. STACK FRAME

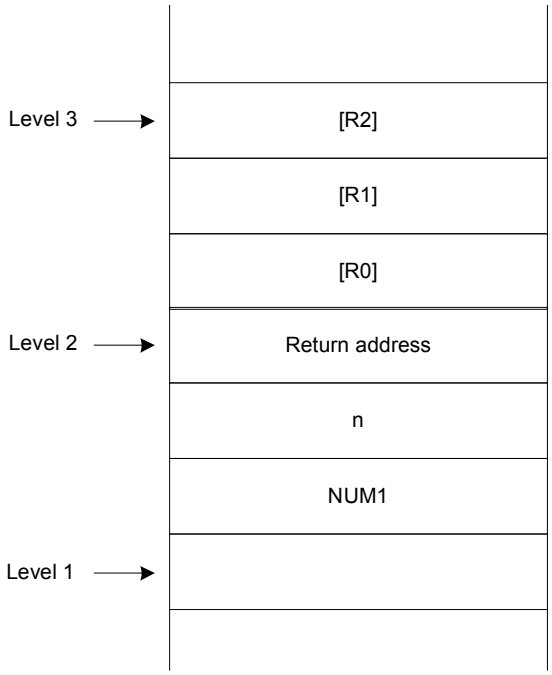
Sekarang amatilah bagaimana stack digunakan dalam contoh pada Gambar 8.10. Selama eksekusi subroutine, enam lokasi pada puncak stack berisi entri yang diperlukan oleh subroutine tersebut. Lokasi ini merupakan ruang kerja privat untuk subroutine, dibuat pada saat subroutine dimasuki dan dikosongkan pada saat

subroutine mengembalikan kontrol ke calling program. Ruang semacam itu disebut stack frame. Jika subroutine memerlukan lebih banyak ruang untuk variabel mernori lokal, maka dapat juga dialokasikan pada stack.

Assume top of stack is at level 1 below.

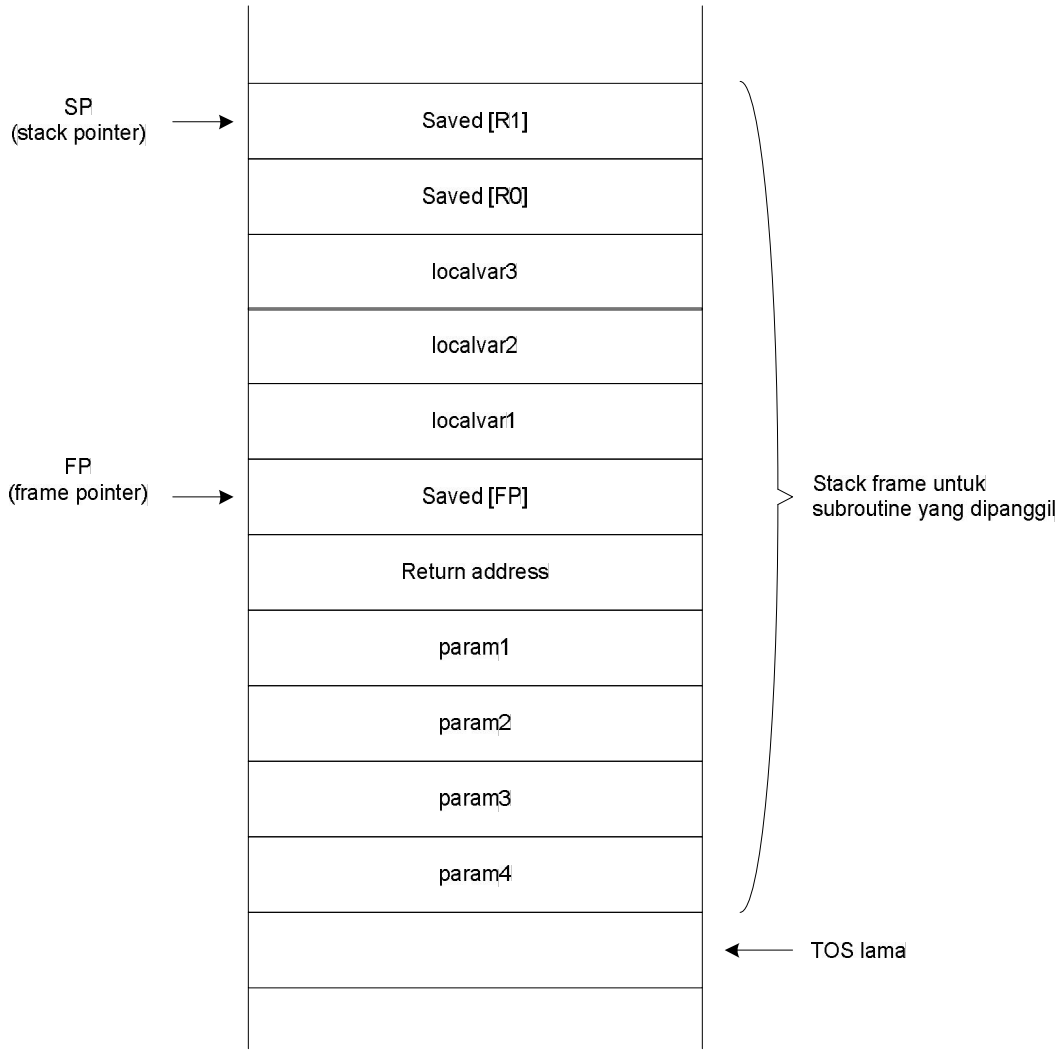
```
            Move        #NUMl,-(SP)     Push parameters onto stack.
            Move        N,-(SP)
            CaLL        LIST ADD        Call subroutine
                                        (top of stack at Level 2).
            Move        4(SP),SUM       Save result.
            Add         #8,SP           Restore top of stack
                                        (top of stack at Level 1).
        .
        .
        .
LIST ADD Mo             RO-R2,-(SP)     Save registers
                                        (top of stack at Level 3).
            Move        16(SP),R1       Initialize counter to n.
            Move        20(SP),R2       Initialize pointer to the list.
            Clear       RO              Initialize sum to 0.
LOOP Add                (R2)+,RO        Add entry from List.
    Decrement           R1
    Branch>0            LOOP
     Move               R0,20(SP)       Put result on the stack.
     MoveMultiple       (SP)+,RO-R2     Restore registers.
          Return                        Return to calling program.
```

(a) Calling program dan subroutine



(b) Puncak stack pada berbagai waktu

Gambar 8.10. Parameter dilewatkan pada stack



Gambar 8.11. Contoh stack frame subroutine

Gambar 8.11 menampilkan suatu contoh layout yang biasa digunakan sebagai informasi dalam suatu stack frame. Selain stack pointer, SP, perlu juga untuk memiliki register pointer lainnya, yang disebut frame pointer (FP), untuk akses mudah ke parameter yang dilewatkan ke subroutine dan ke variabel memori lokal yang digunakan oleh subroutine. Variabel lokal tersebut hanya digunakan dalam subroutine, sehingga tepat untuk mengalokasikan ruang dalam stack frame yang diasosiasikan dengan subroutine tersebut. Pada Gambar tersebut, kita mengasumsikan bahwa empat parameter dilewatkan ke subroutine, tiga variabel lokal digunakan dalam subroutine, dan register RO dan RI perlu disimpan karena juga akan digunakan dalam subroutine.

Dengan register FP menunjuk ke lokasi tepat di atas return address yang disimpan, sebagaimana ditunjukkan pada Gambar 8.11, kita dapat dengan mudah mengakses parameter dan variabel lokal dengan menggunakan mode pengalamatan

Index. Parameter dapat diakses dengan menggunakan alamat 8(FP), 12(FP), ... Variabel lokal dapat diakses menggunakan alamat -4(FP), -8(FP), ...Isi FP tetap selama eksekusi subroutine, berbeda dengan stack pointer, SP, yang harus selalu menunjuk ke elemen teratas pada stack saat ini.

Sekarang marilah kita membahas bagaimana pointer SP dan FP dimanipulasi pada saat stack frame dibuat, digunakan, dan dibongkar untuk pemanggilan subroutine tersebut. Kita mulai dengan mengasumsikan bahwa SP menunjuk ke elemen top-of-the-stack (TOS) lama pada Gambar 8.11. Sebelum subroutine dipanggil, calling program mendorong empat parameter tersebut ke stack. Instruksi stack kemudian dieksekusi, hasilnya adalah return address didorong ke stack. Sekarang SP menunjuk ke return address ini, dan instruksi pertama dari subroutine tersebut dieksekusi. Ini adalah titik dimana frame pointer FP di-set untuk mengisikan alamat memori yang sesuai. Karena SP biasanya merupakan general-purpose register, maka register tersebut dapat berisi informasi penggunaan calling program. Oleh karena itu, isinya disimpan dengan push ke dalam stack. Karena SP sekarang menunjuk ke posisi ini, maka isinya di-copy ke dalam FP.

Sehingga, dua instruksi pertama yang dieksekusi dalam subroutine adalah

```
Move        FP, -(SP)
            Move        SP, FP
```

Setelah instruksi tersebut dieksekusi, SP dan FP menunjuk ke isi FP yang disimpan. Ruang untuk tiga variabel lokal tersebut sekarang dialokasikan pada stack dengan mengeksekusi instruksi

```
Subtract  #12, SP
```

Akhirnya, isi register prosesor RO dan Rl disimpan dengan push ke stack. Pada titik ini, stack frame telah di-set up sebagaimana yang ditampilkan pada Gambar tersebut.

Subroutine sekarang mengeksekusi tugasnya. Pada saat tugas tersebut selesai, subroutine pop nilai RI dan RO kembali ke register tersebut, memindahkan variabel lokal dari stack frame dengan mengeksekusi instruksi sebagai berikut

```
Add  #12, SP
```

dan pop nilai FP lama yang tersimpan kembali ke FE Pada titik ini, SP menunjuk ke return address, sehingga instruksi return dapat dieksekusi, mentransfer kontrol kembali ke calling program.

Calling program bertanggungjawab untuk memindahkan parameter dari stack frame, yang beberapa diantaranya mungkin merupakan hasil'yang dikembalikan oleh subroutine. Stack pointer sekarang menunjuk ke TOS lama, dan kita kembali ke tempat kita mulai.

8.11. STACK FRAME UNTUK NESTED SUBROUTINE

Stack adalah struktur data yang sesuai untuk menyimpan return address pada saat subroutine nested. Jelas bahwa stack frame lengkap untuk nested subroutine dibuat pada stack prosesor pada saat dipanggil. Dalam hal ini, perhatikanlah bahwa isi FP yang tersimpan dalam frame terbaru pada puncak stack adalah isi frame pointer untuk stack frame subroutine yang memanggil subroutine terbaru.

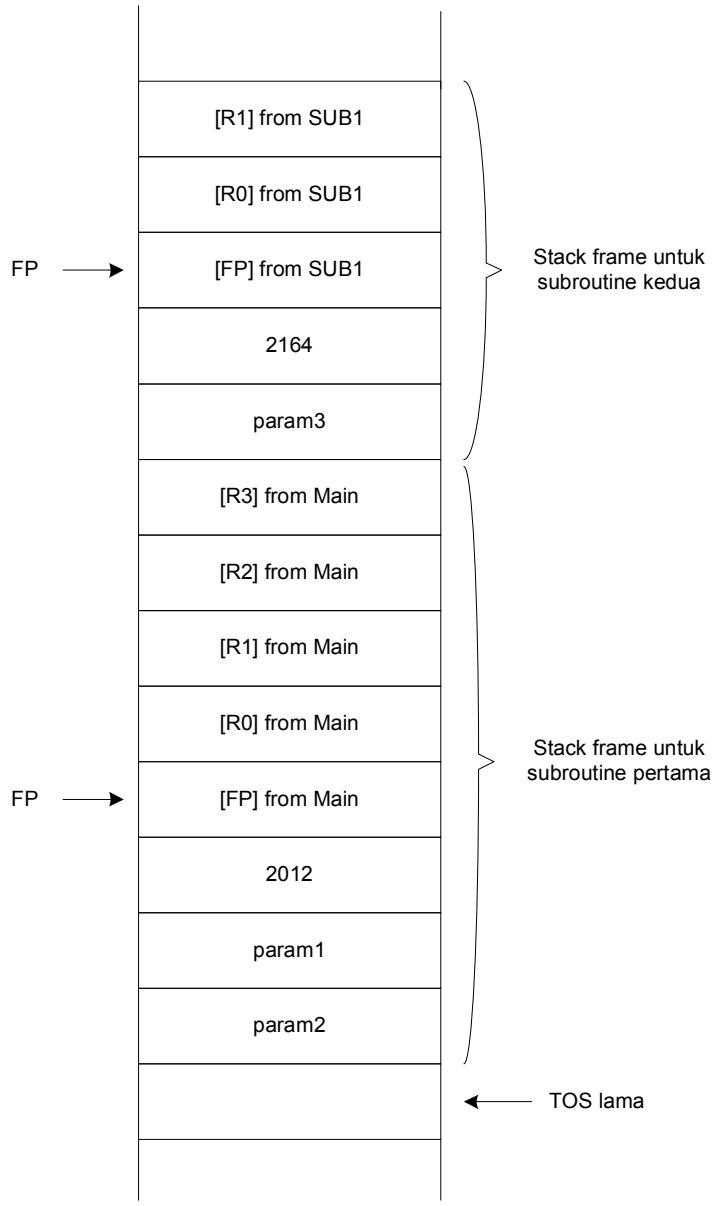
Suatu contoh program utama yang memanggil subroutine pertama SUB I, kemudian memanggil subroutine kedua SUB2, ditampilkan pada Gambar 8.12. Stack frame yang berhubungan dengan dua nested subroutine ini ditampilkan pada Gambar 8.13. Semua parameter yang terlibat dalam contoh ini dilewatkan ke stack. Gambar tersebut hanya menampilkan aliran kontrol dan data diantara tiga program tersebut. Komputasi yang sebenarnya tidak ditampilkan.

Aliran eksekusinya adalah sebagai berikut. Program utama push dua parameter param2 dan paraml ke stack dalam susunan tersebut dan kemudian memanggil SUB 1. Subroutine pertama ini bertanggung jawab untuk menghitung jawaban tunggal dan melewatkannya kembali ke program utama pada stack. Selama rangkaian komputasi, SUB 1 memanggil subroutine kedua SUB2, untuk menjalankan beberapa subtugas. SUB I melewatkan parameter tunggal param3 ke SUB2 dan mendapatkan suatu hasil yang dilewatkan kembali padanya. Setelah SUB2 mengeksekusi instruksi Return-nya, maka hasil ini disimpan dalam register R2 oleh SUBI. SUB1 kemudian melanjutkan komputasinya dan pada akhirnya melewatkan jawaban yang diminta kembali ke program utama pada stack. Pada saat SUB 1 mengeksekusi return-nya ke program utama, maka program utama menyimpan jawaban ini dalam lokasi memori RESULT dan melanjutkan komputasinya pada “instruksi berikutnya.”

Memory Location	Instructions		Comments
Main program			
	.		
	.		
	.		
2000	Move	PARAM2,-(SP)	Place parameters on stack.
2004	Move	PARAMI,-(SP)	
2008	Call	SUB1	
2012	Move	(SP),RESULT	Store result.
2016	Add	#8,SP	Restore stack level.
2020	next instruction		
	.		
	.		
	.		
First subroutine			
2100	SUB1 Move	FP,-(SP)	Save frame pointer register.
2104	Move	SP,-(SP)	Load the frame pointer.
2108	Move Multiple	RO-R3,-(SP)	Save registers.
2112	Move	8(FP),RO	Get first parameter.
	Move	12(FP),Rl	Get second parameter.
	.		
	.		
	.		
	Move	PARAM3,-(SP)	Place a parameter on stack.
2160	Call	SU82	
2164	Move	(SP)+,R2	Pop SUB2 result into R2.
	.		
	.		
	.		
	Move	R3,8(FP)	Place answer on stack.
	Move Multiple	(SP)+,RO-R3	Restore registers.
	Move	(SP)+,FP	Restore frame pointer register.
	Return		Return to Main program.
Second subroutine			
3000	SUB2 Move	FP,-(SP)	Save frame pointer register.
	Move	SP,FP	Load the frame pointer.
	Move Multiple	RO-Rl,-(SP)	Save registers RO and R1.
	Move	8(FP),RO	Get the parameter.

```
.
.
.
Move            R1,8(FP)        Place SUB2 result on stack.
Move Multiple   (SP)+,RO-R3     Restore registers RO and Rl.
Move            (SP)+,FP        Restore frame pointer register.
Return                          Return to Subroutine 1.
```

Gambar 8.12. Nested Subroutine



Gambar 8.13. Stack frame untuk gambar 8.12.

Komentar pada Gambar 8.12 menyediakan detil bagaimana pengaturan aliran eksekusi ini. Aksi pertama yang dilakukan oleh tiap subroutine adalah men-set frame pointer, setelah menyimpan isi sebelumnya ke stack, dan menyimpan register lain yang diperlukan. SUBI menggunakan empat register, RO hingga R3, dan SUB2

menggunakan dua register, RO dan RI. Register dan frame pointer dipulihkan tepat sebelum return dieksekusi.

Mode pengalamatan Index yang melibatkan register frame pointer FP digunakan untuk me-load parameter dari stack dan menempatkanjawabannya kembali ke stack. Byte offset yang digunakan dalam operasi ini selalu 8, 12, ..., sebagaimana telah dibahas untuk general stack frame pada Gambar 8.11. Akhirnya, perhatikanlah bahwa routine calling bertanggung jawab untuk memindahkan parameter dari stack. Hal ini dilakukan oleh instruksi Add dalam program utama, dan oleh instruksi Move pada lokasi 2164 dalam SUB 1.