ASSEMBLER

2.1. Element dari Bahasa Pemrograman Assembler

Bahasa assembly dikategorikan sebagai bahasa tingkat rendah (*low level languange*). Ini untuk menggambarkan kekhususannya sebagai bahasa yang berorientasi pada *machine dependent*. Untuk membandingkan bahasa mesin dan bahasa assembly, kita dapat melihatnya dari tiga karakteristik berikut :

1. ***Mnemonic operation code***. Sebagai pengganti *numeric operation code (opcodes)* yang digunakan pada bahasa mesin, digunakankanlah mnemonic code pada bahasa assembly. Selain kemudahan dalam penulisannya dibandingkan dari bahasa mesin juga mendukung pelacakan kesalahan seperti kesalahan penulisan operation code. Gambar 2.1. berikut menunjukkan daftar instuksi *operation codes* dari bahasa dan bahasa assembly.

Instuction Op Code	Assembly Mnemonic	Remarks
00	STOP	
01	ADD	Operand pertama yang diasumsikan sebagai akumulator
02	SUB	
03	MULT	
04	LOAD	Memanggil akumulator
05	STORE	Menyimpan akumulator ke dalam lokasi storage
06	TRANS	Mentransfer kontrol ke alamat yang disebutkan
07	TRIM	Mentransfer hanya jika akumulator < 0
08	DIV	Membagi akumulator dengan isi lokasi storage
09	READ	Membaca kartu pada lokasi storage
10	PRINT	Mencetak isi lokasi storage
11	LIR	Memanggil index register dengan 3 digit terakhir dari storage operand
12	IIR	Menaikkan index register dengan 3 digit terakhir dari storage operand
13	LOOP	Mengurangi index register, jika isi baru . 0 kemudian sama denan TRANS

2. ***Symbolic operand specification***. Penamaan simbol diasosiasikan sebagai suatu data atau instruksi. Operand lebih menunjukkan *symbolic reference* dibandingkan dengan alamat mesin suatu data atau instruksi. Hal ini akan mempermudah pada saat harus dilakukan modifikasi program.

3. ***Declaration of data/storage area***. Data dapat dinyatakan dalam notasi desimal. Ini dilakukan untuk mencegah konversi secara manual dari konstanta ke dalam representasi internal mesin. Sebagai contoh :
   -5 menjadi $(11111010)_2$ atau 10.5 menjadi $(41A80000)_{16}$

Suatu statement bahasa assembly mempunyai bentuk umum sebagai berikut :

[Label]   Menmonic OpCode   Operand [operand…]

Tanda kurung siku menunjukkan isi di dalamnya boleh digunakan atau tidak dalam statement tersebut, sebagai contoh : label bersifat optional. Jika label digunakan, hal tersebut menujukkan suatu *symbolic name* akan dibuat dalam machine word untuk keperluan *assembly statement*. Bila digunakan lebih dari satu operand, digunakan tanda "koma" untuk memisahkannya. Jika digunakan index, nomor index register ditunjukan dalam sebuah simbol, seperti contoh berikut :

```
AGAIN        LOAD   NUMBER(4)
```

Dimana '4' menunjukkan register yang memiliki index. AGAIN diasosiasikan dengan instruksi mesin yang dihasilkan untuk statement LOAD.

Gambar 2.2 berikut ini mengilustrasikan program bahasa mesin yang dipersamakan dengan bahasa assembly.

(a)		(b)	
		START	100
100) + 09 0 114		READ	A
101) + 11 4 114		LIR	4, A
102) + 04 0 115		LOAD	ONE
103) + 05 0 117		STORE	RESULT
104) + 05 0 116		STORE	TERM
105) + 04 0 117	AGAIN	LOAD	RESULT
106) + 03 0 116		MULT	TERM
107) + 05 0 117		STORE	RESULT
108) + 04 0 116		LOAD	TERM
109) + 01 0 115		ADD	ONE
110) + 05 0 116		STORE	TERM
111) + 13 4 105		LOOP	4, AGAIN
112) + 100 117		PRINT	RESULT
113) + 00 0 000		STOP	
114)	A	DS	1
115) + 00 0 001	ONE	DC	‘1’
116)	TERM	DS	1
117)	RESULT	DS	1
		END	

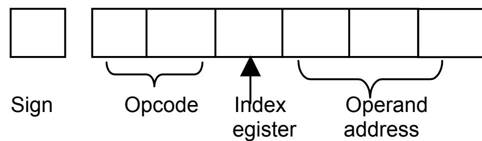
Gambar 2.2. (a) program bahasa mesin (b) Eguivalent program bahasa assembly

Program assembly mengenal tiga jenis statement : (i) *imperative statement* (ii) *declarative statement* (iii) *assembler directive statement*.

Imperative Statement

Statement imperative dalam bahasa assembly ditunjukan dengan suatu tindakan yang dikerjakan selama eksekusi program assembly. Karena itu setiap statement imperative ditranslasikan ke dalam instruksi mesin.

Format instruksi :



Declarative Statement

Statement declarative dalam bahasa assembly menunjukkan konstanta atau storage area pada suatu program. Sebagai contoh :

A DS 1

secara sederhana storage area sebesar 1 word ditunjukkan dengan sebuah label A. DS di sini menunjukkan *Declare Storage* (DS).

Suatu konstanta dideklarasikan melalui *Declare Constant* (DC) statement, contohnya :

ONE DC ‘1’

maksud dari statement di atas adalah label ONE berisi konstanta 1. Programmer dapat mendeklarasikan kontanta dalam desimal, binary, hexadesimal, dsb. Assembler akan mengkonversi bentuk tersebut ke dalam bentuk internal yang tepat.

Beberapa assembler sering pula menggunakan’literal’ khususnya pada konstanta yang dipakai sebagai operand, seperti contoh berikut :

```
            ADD    ONE                    ADD   = '1'
             -                             -
             -            ==>              -
      ONE   DC     '1'
```

Penggunaan tanda "=" pada posisi awal suatu operand menunjukkan sebuah literal. Nilai konstanta yang ditulis dengan cara demikian sama dengan nilai yang dihasilkan bila menggunakan statement DC.

Assembler Directive
Statement jenis ini tidak merepresentasikan instruksi mesin ke dalm suatu objek program atau mengalokasikan storage untuk konstanta atau variable program. Sebaliknya, statement ini secara langsung mengarahkan assembler untukmengambil alih aksi selama proses assembling program. Statement ini digunakan untuk menunjukkan bagaimana input program assembly dibentuk, sebagi contoh : START 100
statement tersebut merupakan kata pertama dari objek program yang dibuat oleh assembler untuk menempatkan lokasi mesin pada alamat '100'. Begitupula dengan statement : END, yang mengindikasikan tidak ada lagi bahasa statement bahasa assembly yang akan diproses.

2.2. Proses Assembly

Untik membangun skema proses translasi dari satu bahasa ke bentuk lainnya, hal pertama yang harus dilakukan adalah mengindentifikasikan tugas-tugas dasar yang harus dikerjakannya dalam proses translasi tersebut.

2.2.1. Proses Translasi

Secara umum model proses translasi dapat direpresentasikan sebagai berikut :

{ Analysis of Source Text } + { Synthesis of Target Text } = { Translation from Source Text to Target Text }

Model di atas diterapkan untuk mentranslasikan dari sautu bahasa pemrograman ke bentuk lain, translasi dari satu bahasa natural (Inggris, Perancis) ke bentuk coding / decoding pesan rahasia. Untuk mengaplikasikan model di atas, kita perlu menentukan komponen-komponen yang dibutuhkan selama proses analisis dan sintesis.

Dalam fase analisis, focus perhatian kita adalah kepada penentuan arti dari *source text*. Untuk memahami arti dari *source text* tersebut, kita mengetahui aturan yang membentuk *source text* tersebut. Dalam aturan struktur tatabahasa (*grammar*), dikenal istilah *syntax* dan *semantic*. Perhatikan statement berikut :

AGAIN LOAD RESULT + 4

Dalam statement di atas, AGAIN menunjukkan *label field*, LOAD menunjukkan *opcode mnemonic field* dan RESULT + 4 menunjukkan *operand field*. Bila kita melihat lebih dalam lagi ke dalam operand field, kita dapat menukan bahwa RESULT + 4 adalah expression operand yang valid dan sesuai dengan aturan bahasa. Dalam bahasa assembly, aturan penulisan suatu statement sangat sederhana. Pembahasan mengenai tata bahasa akan dilanjutkan pada materi-materi berikutnya.

Dalam fase sintesis, dilakukan pemilihan *machine operation code* yang sesuai dengan *mnemonic* LOAD dan menempatkannya pada *machine instruction opcode field*. Evaluasi korespondensi pengalamatan dilakukan untuk operand expression 'RESULT + 4' dan menempatkannya pada alamat dari *machine instruction*.

2.2.2. Skema Sederhana Assembly

Fase Analysis

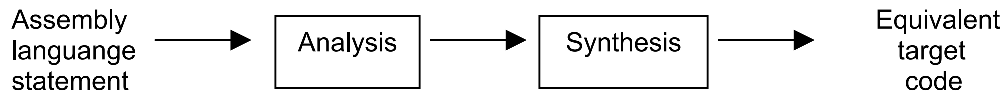
- Mengisolasikan / memisahkan *label*, *mnemonic operation code* dan *operand field* yang ada pada statement
- Memasukkan simbol yang ditemukan pada label field dan alamat yang akan dituju *machine word* ke dalam *Symbol table*.
- Melakukan validasi menmonic operation code dengan melihat pada *Mnemonic table*
- Menentukan alamat yang dibutuhkan statement berdasar pada *mnemonic operation code* dan *operand field* pada statement. Proses penghitungan alamat awal *machine word* mengikuti target code yang dibangkitkan untuk statement tersebut (*Location Counter (LC) processing*)

Fase Syntesis

- Menghasilkan machine operation code yang berkorespondensi dengan *mnemonic operation code* yang telah dicari pada *mnemonic table*
- Menghasilkan alamat operand dari *Symbol table*
- Melakukan sintesa instruksi machine

2.2.3. Pass Structure pada Assembler

Pada pembahasan di atas, kita telah mengidentifikasikan fungsi fase analisi dan sintesis dari assembler. Sekarang kita akan melihat fase program assembly ini berdasarkan statment demi statement hingga menghasilkan target program, seperti terlihat dari gambar 2.3 berikut ini :



Gambar 2.3. Translasi statement demi statement program bahasa assembly

Perhatikan contoh program assembly berikut ini :

```
        MOVE. L   FOUR, DO
        ADD.L     FIVE, DO
        MOVE.L    DO, SUM
FOUR    DC.L      4
FIVE    DC.L      5
SUM     DS.L      1
```

Pada contoh di atas terlihat bahwa FIVE merupakan label yang menunjuk pada alamat dimana isi dari FIVE tersebut disimpan. Dalam kasus ini FIVE tidak terdefinisi sebelumnya . Oleh karena ini, penterjemahan bentuk ini disebut *forward reference*. Pada kasus *forward reference* alamat dari label harus dikenali dalam jenisnya untuk diterjemahkan ke dalam suatu program yang semestinya. Solusi yang ditawarkan adalah kita perlu melakukan proses terhadap *source statement* lebih dari satu kali atau dilakukan secara beberapa tahap. Hal ini dikenal dengan konsep *translator pass*.

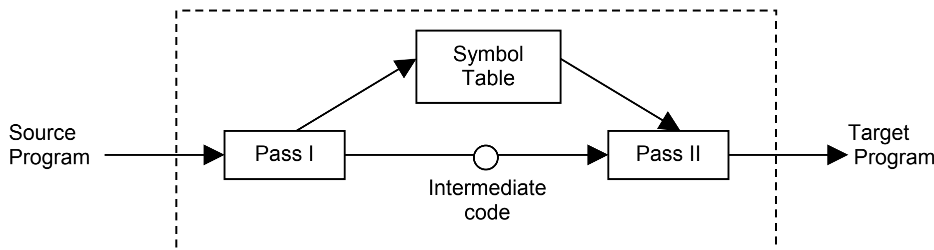
Translator pass adalah penelusuran secara menyeluruh *source program input* oleh translator hingga mencapai *equivalent representation*.

Translasi yang dilakukan statement demi statement disebut *single pass translation*, sedangkan translasi yang dilakukan sekelompok statement yang membutuhkan banyak pass disebut *multipass translation*.

A. Multi-Pass Translation

Multi pass translation dalam program bahasa assembly dapat menangani masalah *forward reference*. Unit pada *source program* digunakan untuk tujuan mentranslasi semua bagian program. Ketika fase analisis statement program pertama kali dilakukan, proses LC akan dikerjakan dan simbol yang didefinisikan dalam program dimasukkan ke dalam simbol table. Selama *second pass*, statement diproses dengan tujuan mensintesa *target form*. Semua simbol dan alamat yang dapat ditemukan dalam simbol table tidak akan menimbulkan *forward reference* pada assembly.

Kalimat "*equivalent representation*" digunakan pada translasi yang membutuhkan elaborasi. Sering kali ketika proses pemisahan field *label*, *mnemonic opcode* dan *operand field* terjadi duplikasi. Untuk mengurangi duplikasi tersebut, hasil analisa source statement dari *first pass* direpresentasikan dalam *internal form* pada *source statement*. Bentuk ini disebut *intermediate code*. Ilustrasi dari skema *two pass assembler* dengan menggunakan *intermediate code form* dapat dilihat pada gambar 2.4 berikut ini :



Gambar 2.4. Skema multi pass assembler

Selain membangun *intermediate code*, suatu *assembler pass* juga membangun dan/atau mengganti data base yang digunakan *subsequent pass*. Karena kebutuhan untuk membangun dan memproses *intermediate code*, suatu *multi pass translator* menjalankan fungsinya lebih lambat dibandingkan dengan *single pass translation*.

B. Single Pass Translation

Dalam *single pass translation*, pemecahan *forward reference* dapat ditangani sebagai berikut : instruksi yang memuat *forward reference* dapat ditinggalkan dalam keadaan tidak selesai hingga alamat *reference symbol* diketahui. Untuk meletakkan alamat operand pada bagain akhir dapat disimpan pada *Table of Incomplete Instruction* (TII). Di akhir program *assembly*, semua masukan pada table dapat diproses secara lengkap sesuai instruksinya.

Keuntungan menggunakan *single pass translation* adalah setiap *source statement* hanya diproses satu kali. Proses ini lebih cepat bila dibandingkan dengan *multi pass translation*. Namun demikian, ada kekurangan yang terdapat pada *single pass translation*, yaitu besarnya *area storage* yang dibutuhkan oleh assembler, sebagai akibat dijalankannya fase *analysis* dan *synthesis* pada pass yang sama.

Secara umum dikenal ada dua macam tipe *single pass translation / one pass*, yaitu :

- *single pass translation* yang menghasilkan kode objek langsung ke memori, yang mengakibatkan eksekusi menjadi lebih cepat.
- *single pass translation* yang menghasilkan berbagai tipe pemrograman untuk keperluan eksekusi selanjutnya

2.3. Perancangan Two Pass Assembler

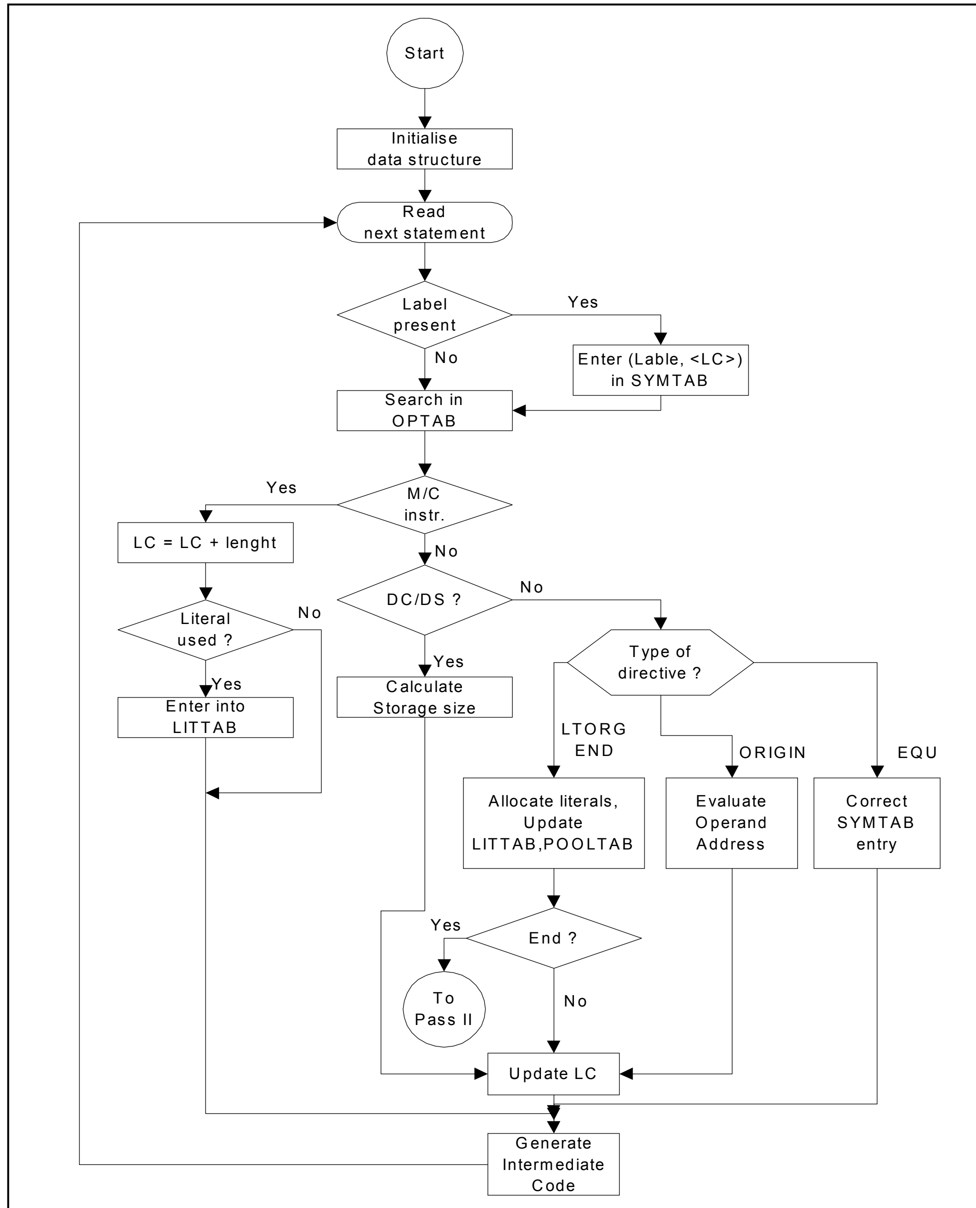
Pass I

- memisahkan *symbol, mnemonic opcode* dan *operand field*
- menentukan *storage* yang dibutuhkanuntuk setiap statement bahasa assembly dan meng-update *location counter*
- membangun *symbol table*
- merancang *intermediate code* untuk setiap statement bahasa assembly

Pass II

mensintesa target code dengan memproses intermediate code yang dibangkitkan selama pass I

Flowchart yang menggambarkan Pass I assembler dapat dilihat di bawah ini :



Gambar 2.5 Pass I assembler

Pada Pass I digunakan beberapa table, yaitu :
(i) OPTAB : table mnemonic opcode dan informasi lain yang terkait
(ii) SYMTAB : symbol table
(iii) LITTAB : table literal yang digunakan dalam program

Mnemonic OpCode	Class	Machine opcode / Routine id	Length
LOAD	1 (Imperative)	04	1
DS	2 (Declarative)	R#7	-
START	3 (Directive)	R#11	-
STORE	1 (Imperative)	05	1

OPTAB

Symbol	Address	Length	Other Information

SYMTAB