महाराष्ट्र राज्य तंत्रशिक्षण मंडळ, मुंबई
(स्वायत्त) (ISO 9001:2015) (ISO/IEC 27001:2013)

अभियांत्रिकी आणि तंत्रज्ञान पदविका

शिक्षण पुस्तिका
(Learning Material)

MICROPROCESSOR PROGRAMMING
(314321)

K Scheme

मायक्रोप्रोसेसर प्रोग्रामिंग

संगणक अभियांत्रिकी गट
(के स्कीम)

मराठी–इंग्रजी (द्विभाषिक) माध्यम
(अभियांत्रिकी व तंत्रज्ञानातील चौथे सत्र पदविका)

शिक्षण पुस्तिका

(Learning Material)

मायक्रोप्रोसेसर प्रोग्रामिंग

Microprocessor Programming

(314321)

संगणक अभियांत्रिकी गट

(के-स्कीम)

K-Scheme

मराठी-इंग्रजी (द्विभाषिक) माध्यम

(अभियांत्रिकी व तंत्रज्ञानातील चौथे सत्र पदविका)

महाराष्ट्र राज्य तंत्रशिक्षण मंडळ, मुंबई

(स्वायत्त)(ISO 9001:2015) (ISO/IEC 27001:2013)

मायक्रोप्रोसेसर प्रोग्रामिंग
Microprocessor Programming
(314321)

मार्गदर्शक

प्रा. विजय नामदेवराव कुकरे

विभागप्रमुख, संगणक अभियांत्रिकी

प्रकल्प समन्वयक

प्रा. विजय नामदेवराव कुकरे

विभागप्रमुख, संगणक अभियांत्रिकी

संकलक

प्रा. विजय नामदेवराव कुकरे

विभागप्रमुख, संगणक अभियांत्रिकी

महाराष्ट्र राज्य तंत्र शिक्षण मंडळ

(स्वायत्त) (ISO: ९००१:२०१५) (ISO/IES: २७००१-२०१३)

शासकीय तंत्रनिकेतन इमारत, चौथा मजला, ४९, खेरवाडी, बांद्रा (पूर्व), मुंबई - ४०० ०५१.

दूरध्वनी क्र.: ०२२-६२५४२१७० / १६१

Email : director@msbte.com Web : www.msbte.org.in

प्रास्ताविक

महाराष्ट्र राज्यातील पदविका स्तरावरील तंत्रशिक्षणामध्ये विद्यार्थ्यांचे रोजगार कौशल्य विकसित करून विद्यार्थ्यांचा सर्वांगीण विकास घडवून आणण्याकरिता महाराष्ट्र राज्य तंत्रशिक्षण मंडळ कटिबद्ध आहे. उद्योगधंद्यातील बदलत्या तंत्रज्ञानाशी संबंधित गरजा लक्षात घेऊन महाराष्ट्र राज्य तंत्र शिक्षण मंडळाकडून पदविका अभ्यासक्रम वेळोवेळी अद्यावत करण्यात येतो . अभियांत्रिकी पदविका अभ्यासक्रम शिकत असतांना संकल्पनात्मक ज्ञान, सुसंगत संदर्भ, प्रश्न विचारणे, विश्वसनिय पुरावे, कारणमीमांसा आणि सुस्पष्ट निकष यांचा वापर करून अर्थाची उकल करण्याची, विश्लेषण व मूल्यमापन करण्याची तसेच तर्काने अनुमान काढण्याची क्षमता म्हणजेच चिकित्सक विचार विद्यार्थ्यांमध्ये अधिक दृढ होतील असा मला विश्वास आहे. जेव्हा विद्यार्थी ज्ञान मिळवण्याच्या माध्यमाशी पूर्णपणे परिचित आणि सोयीस्कर असतात, तेव्हा त्यांच्यासाठी वर्गातील चर्चेत भाग घेणे सोपे होते, संकल्पनात्मक व सैद्धांतिक बाबींचे आकलन परिपूर्ण होते, संज्ञानात्मक क्षमता सुधारते आणि त्यांचा आत्मविश्वास देखील वाढतो या सर्व गोष्टींचा विचार करून मंडळाकडून शैक्षणिक सामुग्रीची निर्मिती करण्यात आलेली आहे . भारत देश हा खेड्यापाड्यातून विकसित झालेला देश असून ग्रामीण भागातील विद्यार्थ्यांना तांत्रिक शिक्षण घेतांना भाषेचा अडसर न येता तांत्रिक बाबींचा आशय समजून घेणे शक्य होईल या दृष्टिकोनातून महाराष्ट्र राज्य तंत्र शिक्षण मंडळाने पदविका स्तरावरील तांत्रिक शिक्षणाकरीता विद्यार्थ्यांना मराठी-इंग्रजी द्विभाषिक माध्यमाचा पर्याय शैक्षणिक वर्ष २०२१-२२ पासून उपलब्ध करून दिलेला आहे .

राष्ट्रीय शैक्षणिक धोरण-२०२० प्रादेशिक भाषेतील शिक्षणास प्रोत्साहन देते, ज्यामुळे विद्यार्थ्यांना तांत्रिक अभ्यासक्रमांसाठी प्रादेशिक भाषांतुन शिक्षणाचे माध्यम निवडता येते. सदर धोरणामुळे प्रादेशिक भाषांमध्ये तांत्रिक सामग्री आणि अभ्यास सामग्रीचा विकास आणि भाषांतर निर्माण करण्याची आवश्यकता आहे . त्यास अनुसरून मंडळाने मराठी-इंग्रजी द्विभाषिक माध्यमाचा पर्याय द्वितीय व तृतीय वर्षाकरिताही उपलब्ध करून देण्यात आला आहे. तसेच त्याकरिताची शैक्षणिक सामग्रीही संबंधीत भागधारकरांना उपलब्ध करून देण्यात येत आहे.

पदविका स्तरावरील तंत्रशिक्षण अधिक दर्जेदार करण्यासाठी महाराष्ट्रातील अनुभवी व तज्ञ अध्यापकांनी व्यवहारिक मराठी भाषा व इंग्रजी भाषेतील तांत्रिक शब्दावली यांचा वापर करून मराठी - इंग्रजी भाषेचा सुवर्णमध्य साधण्याचा प्रयत्न केलेला आहे. मंडळाच्या स्तरावर गठीत सुकाणू समितीमार्फत सदर शैक्षणिक सामुग्रीचा दर्जा, तसेच इतर बाबींची तपासणी करण्यात आलेली आहे. त्यामुळे सदर शैक्षणिक सामुग्री अधिक सम्पन्न झालेली असून विद्यार्थी त्यांच्या व्यक्तिमत्त्वाचा सुसंवादी आणि सर्वांगीण विकास साधतील. परिणामतः विश्वस्तरीय मनुष्यबळाच्या गरजा पूर्ण करण्यात महाराष्ट्र राज्य अग्रेसर राहील व पर्यायाने राष्ट्रनिर्मिती करीता निश्चितच हातभार लागेल , असा मला विश्वास आहे.

अभियांत्रिकी पदविका अभ्यासक्रमातील प्रमुख विषयांची मराठी-इंग्रजी द्विभाषिक शैक्षणिक सामुग्री बनविण्यासाठी अध्यापक व सुकाणू समितीचे सदस्य यांनी दर्शविलेले समर्पण व वचनबद्धता कौतुकास पात्र आहे , या सर्वांचे मी मनः पूवक अभिनंदन करतो !

(प्रमोद नाईक)

संचालक

म. रा. तंत्र शिक्षण मंडळ, मुंबई.

अनुक्रमणिका

अ. नु.	युनिटचे नाव	पृष्ठ क्रमांक
1	8086 16 बिट मायक्रोप्रोसेसर (8086-16 Bit Microprocessor)	1
2	असेम्ब्ली लँग्वेज प्रोग्रामिंगची कला (The Art of Assembly Language Programming)	13
3	8086 मायक्रोप्रोसेसरचे इंस्ट्रक्शन सेट (Instruction Set of 8086 Microprocessor)	18
4	असेम्ब्ली लँग्वेज प्रोग्रामिंग (Assembly Language Programming)	43
5	प्रोसिजर आणि मॅक्रो (Procedure and Macro)	67

युनिट-1
8086 16-बिट मायक्रोप्रोसेसर
(8086 16-Bit Microprocessor)

विषय निष्पत्ती (Course Outcome):

CO1: 8086 मायक्रोप्रोसेसरच्या फंक्शनल ब्लॉक आकृतीचे विश्लेषण करा.

घटक निष्पत्ती (Theory Learning Outcome):

1. 8086 च्या दिलेल्या पिनच्या कार्याचे वर्णन करा.
2. 8086 मायक्रोप्रोसेसर मधील बस इंटरफेस युनिट आणि एक्झिक्युशन युनिटचे कार्य स्पष्ट करा.
3. 8086 मायक्रोप्रोसेसरमधील दिलेल्या रजिस्टरचे कार्ये सांगा.
4. 8086 मायक्रोप्रोसेसरच्या दिलेल्या सेगमेंटसाठी फिजिकल ॲड्रेस काढा.

1.1 8086 मायक्रोप्रोसेसरचा परिचय (Introduction to 8086 Microprocessor)

8086 मायक्रोप्रोसेसर ही 8085 मायक्रोप्रोसेसरची सुधारित आवृत्ती आहे, हे 1976 मध्ये 8085 मायक्रोप्रोसेसरच्या कमतरतेवर मात करण्यासाठी विकसित केले गेले होते. 8086 हा 16-बिट HMOS मायक्रोप्रोसेसर आहे जो 40 पिन IC मध्ये उपलब्ध आहे आणि 5 व्होल्ट डीसी पॉवर सप्लाय वर चालते. हा एक 16-बिट-आधारित मायक्रोप्रोसेसर आहे ज्यात 20 bit (Address Lines) ॲड्रेस लाईन आणि 16 bit डेटा लाइन्स (Data Lines)आहेत. 8086 मायक्रोप्रोसेसर दोन ऑपरेशन मोड मध्ये काम करतो. मॅक्झिमम (Maximum) मोड ऑपरेशन, एका पेक्षा अधिक प्रोसेसर असलेल्या सिस्टमसाठी उपयुक्त आहे आणि मिनिमम (Minimum) मोड ऑपरेशन एकच प्रोसेसर असलेल्या सिस्टमसाठी उपयुक्त आहे.

1.1.1 8086 मायक्रोप्रोसेसरची वैशिष्ट्ये (Features of 8086 Microprocessor)

8086 मायक्रोप्रोसेसरची सर्वात महत्त्वाची वैशिष्ट्ये पुढीलप्रमाणे -

1. 8086 मायक्रोप्रोसेसरला 20 ॲड्रेस लाईन्स (Address lines) आहेत, त्यामुळे 1024 Kbyte (1Mbytes) मेमरी ॲड्रेस करता येते.
2. IC वर पिनची संख्या कमी करण्यासाठी मल्टिप्लेक्सड 16-बिट ॲड्रेस आणि डेटा बस AD_0-AD_7 (Address/Data bus) आहे.
3. ऑपरेटिंग क्लॉक फ्रिक्वेन्सी (Clock frequency) 5 MHz, 8 MHz, 10 MHz आहेत.
4. अरीथमॅटिक आणि लॉजिकल ऑपरेशन गुणाकार आणि भागाकारासह 8 बिट किंवा 16 बिट साईंड (Signed) आणि अनसाईंड (Unsigned) डेटावर केले जाऊ शकते.
5. सिंगल प्रोसेसर आणि मल्टीप्रोसेसर (Multiprocessor) कॉन्फिगरेशन म्हणजेच ऑपरेटिंग मोडमध्ये कार्य करते.
6. इंस्ट्रक्शन सेट (Instruction set) शक्तिशाली, लवचिक आहे आणि ‘C’ लँग्वेजसारख्या हाय लेव्हल लँग्वेजमध्ये प्रोग्राम केला जाऊ शकतो.
7. 256 प्रकारचे व्हेक्टर्ड सॉफ्टवेअर इंटरप्ट (Vectored Software Interrupt) आहेत.
8. इंस्ट्रक्शन एक्सिक्युशनचा पाइपलाइनसाठी 6-बाइट इंस्ट्रक्शन क्यू (Instruction Queue) प्रदान करतो.
9. 8 बिट किंवा 16 बिट I/O ॲड्रेस जनरेट करतो जे वापरून जास्तीत जास्त 64 K I/O डिवाइसना ॲक्सेस करू शकतो.
10. उच्च परफॉर्मन्स पातळी प्राप्त करण्यासाठी मॅक्सिमम (Maximum) आणि मिनिमम (Minimum) मोडमध्ये कार्य करतो .
11. 24 वेगवेगळ्या ॲड्रेसिंग मोडला (Addressing Mode) समर्थन करते.
12. मल्टीप्रोग्रामिंगला (Multiprogramming) समर्थन करते.
13. स्ट्रिंग मॅनिप्युलेशनसाठी (String Manipulation) वेगळ्या इंस्ट्रक्शन्स दिलेल्या आहेत.

1.1.2 8086 मायक्रोप्रोसेसरचे पिन डायग्राम (Pin diagram of 8086 Microprocessor)

8086 मायक्रोप्रोसेसर हा 40 पिन आयसी आहे ज्यामध्ये आयसीच्या प्रत्येक बाजूला 20 पिन आहेत. त्याची डायग्रॅम खालीलप्रमाणे आहे.

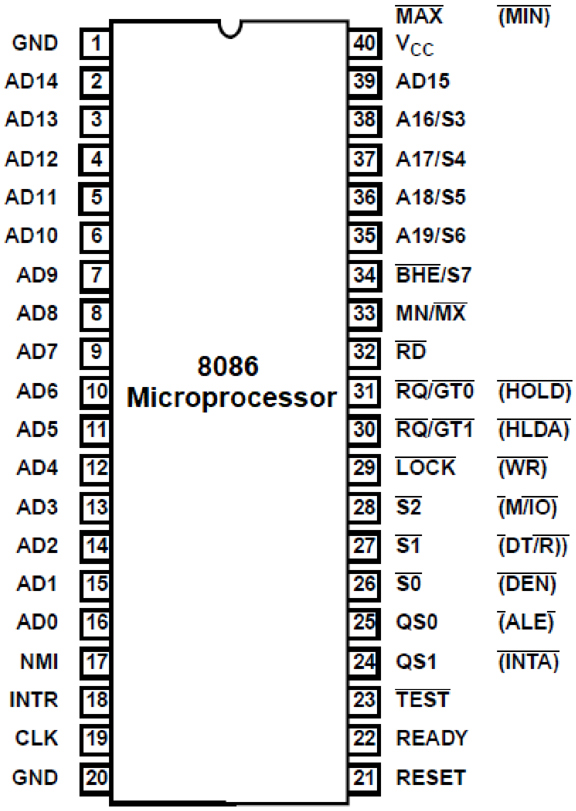


Fig. 1.1:8086 मायक्रोप्रोसेसर पिन डायग्राम (Courtesy: Intersil Corporation)

8086 मायक्रोप्रोसेसर पिनचे फंक्शन (Pin Functions):

1. **AD_{15} – AD_0: (Address/Data bus)**

   हि 16 बिट ऍड्रेस/डेटा बस आहे. AD_0-AD_7 लोअर ऑर्डर (Lower order) डेटा बाइट वाहून नेतो आणि AD_8-AD_{15} हायर ऑर्डर बाइट डेटा वाहून नेतो. पहिल्या क्लॉक सायकलमध्ये, हि बस bit-16 (Address) हायर ऍड्रेस वाहून नेते आणि त्यानंतर तीच बस 16-bit डेटा (Data) वाहून नेते.

2. **A_{16}/S_3-A_{19}/S_6: (Address/Status bus)**

   हि बस मल्टीप्लेक्स्ड ऍड्रेस/स्टेट (Address/Status)ज्या मध्ये 4-बिट ऍड्रेस घेऊन जाते आणि तसेच ते स्टेटस सिग्नल वाहते. T2, T3 आणि T4 दरम्यान, S3 आणि S4 स्टेटस सिग्नल टेबल 1.1 मध्ये दर्शविल्याप्रमाणे मेमरी सेगमेंट ओळखण्यासाठी या स्टेटस लाइनचा वापर केला जातो.

Table 1.1

S_4	S_3	Segment register
0	0	ES
0	1	SS
1	0	CS or none
1	1	DS

3. **CLK(CLOCK)**:

   ह्या पिन द्वारे क्लॉक (CLK) सिग्नल दिला जातो. हे प्रोसेसरला ऑपरेशनसाठी वेळ प्रदान करते. 5 ,8 ,10 (MHz) मेगाहर्टझ अशा वेगवेगळ्या व्हर्जनसाठी याची फ्रिक्वेन्सी वेगवेगळी असते.

4. $\overline{BHE}/S7$: (BUS HIGH ENABLE)

$\overline{BHE}$ म्हणजे बस हाय इनेबल (Bus High Enable) आणि डेटा बस D_8-D_{15} वापरुन डेटाचे हस्तांतरण (Transfer) दर्शविण्यासाठी वापरले जाते. पहिल्या क्लॉक सायकल मध्ये हा सिग्नल लो (LOW) असतो नंतर हाच सिग्नल स्टेटस दर्शवतो. बस हाय इनेबल सिग्नलचा वापर टेबल 1.2 मध्ये दर्शविलेल्या उच्च ऑर्डर (High Order) D_{15}-D_8 डेटा बसवर डेटाचे ट्रान्स्फर करण्यासाठी केला जातो.

Table 1.2

$\overline{BHE}$	A_0	Word / Byte Access
0	0	Whole word from even address
0	1	Upper byte from/to odd address
1	0	Lower byte from/to even address
1	1	None

5. $\overline{RD}$:

हा सिग्नल ऍक्टिव्ह लो आउटपुट सिग्नल आहे. जेव्हा हा सिग्नल लो (LOW) जातो तेव्हा सूचित करतो की प्रोसेसर मेमरी किंवा I/O रीड (READ) ऑपरेशन करत आहे.

6. READY:

हा एक ऍक्टिव्ह हाय (Active High) सिग्नल आहे जो (I/O) डिव्हाइसेसकडून डेटा मिळवण्यासाठी वापरला जातो. जेव्हा I/O डिव्हाइस डेटा ट्रान्सफरसाठी तयार असते, तेव्हा I/O डिव्हाइसद्वारे रेडी सिग्नल मायक्रोप्रोसेसरला पाठवला जातो. जेव्हा हा सिग्नल लो (LOW) असतो, तेव्हा ते डेटा ट्रान्सफरची प्रतीक्षा करण्यास मायक्रोप्रोसेसरला सूचित करते.

7. RESET

RESET हा ऍक्टिव्ह हाय (HIGH) इनपुट सिग्नल आहे. RESET सिग्नल प्रोसेसरचे करंट ऑपेरेशन समाप्त करून प्रोसेसर रीसेट करतो. त्यासाठी हा सिग्नल कमीतकमी चार क्लॉक सायकल हाय (High) असायला पाहिजे.

8. INTR

INTR हा ऍक्टिव्ह हाय लेवल ट्रीगर्ड (HIGH Level Triggered) मास्केबल इंटरप्ट (Maskable Interrupt) इनपुट सिग्नल आहे जो प्रोसेसरला इंटरप्ट (Interrupt) करण्यासाठी वापरला जातो.हा सिग्नल प्रत्येक इन्स्ट्रकशनचा शेवटचा क्लॉक सायकल (last clock cycle) दरम्यान चेक केल्या जातो.

9. NMI (Non Maskable Interrupt**)**

NMI (नॉन-मास्केबल इंटरप्ट) हा ऍक्टिव्ह हाय एज ट्रीगर्ड (HIGH Edge Triggered) इनपुट सिग्नल आहे जो प्रोसेसरला इंटरप्ट (Interrupt) करण्यासाठी वापरला जातो.हा सिग्नल प्रत्येक इन्स्ट्रकशनचा शेवटचा क्लॉक सायकल (last clock cycle) दरम्यान चेक केल्या जातो**.**

10. $\overline{TEST}$:

हा ऍक्टिव्ह लो (LOW) इनपुट सिग्नल आहे जो "WAIT " इंस्ट्रकशननि चेक केल्या जातो .ह्या पिन वर जर इनपुट लो (LOW) असल्यास प्रोसेसरचे एक्सिक्युशन सुरू राहते, अन्यथा प्रोसेसर “आयडल ”(Idle) स्थितीत थांबतो.

11. V_{CC}: हि पॉवर सप्लायची पिन आहे ज्याला 5v वोल्टचा पॉवर सप्लाय लागतो.

12. GND : हि ग्राउंड पिन आहे जी सिस्टिम ग्राउंडला जोडली जाते.

13. $MN/\overline{MX}$:

प्रोसेसर कोणत्या मोडमध्ये ऑपरेट करायचा आहे ते हि पिन ठरवते. जर हि पिन +5V पॉवर सप्लायला कनेक्ट केला तर प्रोसेसर मिनिमम मोड (Minimum Mode) मध्ये कार्य करतो. जर हि पिन GND ग्राउंडला कनेक्ट केला तर प्रोसेसर मॅक्सिमम मोड मध्ये कार्य करतो. पिन नंबर 24 ते 31 चे कार्य मिनिमम आणि मॅक्सिमम मोड मध्ये वेगवेगळे आहेत.

- **मिनिमम मोडे मध्ये पिन नंबर 24 ते 31 चे कार्य (Function of pins number 24 to 31 in minimum mode)**

14. $\overline{\text{INTA}}$:(Pin No 24)

हा सिग्नल ॲक्टिव्ह लो (LOW) आउटपुट सिग्नल आहे. जेव्हा हा सिग्नल लो (LOW) होतो तेव्हा इंटरपटींग डिवाइसला INTR पिनवर इंटरप्ट मिळाल्याची पोचपावती (Acknowledgement) देतो.

15. ALE (Address Latch Enable):(Pin No.25)

हा सिग्नल ॲक्टिव्ह हाय (HIGH) आउटपुट सिग्नल आहे. जेव्हा हा सिग्नल हाय होतो तेव्हा हा सिग्नलॲड्रेस/डेटा बसवर (AD_0 -AD_{15}) वैध (VALID) ॲड्रेसची उपलब्धता दर्शवितो.

16. $\overline{\text{DEN}}$ (Data Enable):(Pin No.26)

ही पिन ॲक्टिव्ह लो (LOW) आउटपुट सिग्नल आहे आणि बस ट्रान्सीव्हरसाठी आउटपुट एनेबल सिग्नल प्रदान करते जी ट्रान्सीव्हर (IC 8286) एनेबल करण्यासाठी वापरल्या जाते. ट्रान्सरिसीव्हर हे एक उपकरण आहे जे ॲड्रेस/डेटा बसमधून डेटा वेगळे करण्यासाठी वापरले जाते.

17. $DT/\overline{R}$: (Data Transmit/Receive) (Pin No.27)

ट्रान्सीव्हरद्वारे डेटा प्रवाहाची दिशा नियंत्रित करण्यासाठी ह्या सिग्नलचा वापर केला जातो. जर हा सिग्नल हाय (HIGH) असेल तर प्रोसेसर डेटा बसवर डेटा ट्रान्स्मिट करतो.जर हा सिग्नल लो (LOW) असेल तर प्रोसेसर डेटा बसवरचा डेटा रिसिव्ह (Receive) करतो.

18. $M/\overline{IO}$: (Pin No.28)

मेमरी आणि I/O ऑपरेशन्स मध्ये फरक करण्यासाठी या सिग्नलचा वापर केला जातो. जेव्हा हा सिग्नल हाय (HIGH) जातो तेव्हा प्रोसेसर मेमरी रीड/राईट (Read/Write) ऑपेरेशन करत असते.जेव्हा हा सिग्नल लो (LOW) जातो तेव्हा प्रोसेसर I/O रीड/राईट (Read/Write) ऑपेरेशन करत असते.

19. $\overline{\text{WR}}$:(Pin No.29)

हा सिग्नल ॲक्टिव्ह लो (LOW) आउटपुट सिग्नल आहे. जेव्हा हा सिग्नल लो (LOW) जातो तेव्हा सूचित करतो की प्रोसेसर मेमरी किंवा I/O राईट (Write) ऑपरेशन करत आहे.

20. HLDA : (Pin No. 30)

हा ॲक्टिव्ह हाय (HIGH) आउटपुट सिग्नल आहे जो प्रोसेसर दुसऱ्या मास्टरला सिग्नल पाठवून अकनॉलेज (Acknowledge) करतो आणि ॲड्रेस बस, डेटा बस व कंट्रोल बस दुसऱ्या मास्टरला डेटा ट्रान्स्फरसाठी सुपूर्द करतो.

21. HOLD: (Pin No. 31)

हा ॲक्टिव्ह हाय (HIGH) इनपुट सिग्नल आहे जो सिस्टिम मध्ये दुसरा मास्टर जसे कि DMA कंट्रोलर (डायरेक्ट मेमरी ॲक्सेस) लोकल बस "होल्ड" करण्याची विनंती करत असल्याचे सूचित करतो आणि प्रोसेसरला सिग्नल पाठवतो.

- **मॅक्सिमम मोडे मध्ये पिन नंबर 24 ते 31 चे कार्य (Functions of the Pin Nos. 24 to 31 in Maximum Mode)**

1. QS_0 and QS_1 (Pin No. 24 and 25)

QS_1 आणि QS_0 सिग्नल्स, 8086 च्या इंस्ट्रकशन क्यूला ट्रॅकिंगला अनुमती देण्यासाठी स्थिती प्रदान करतात जेणे करून दुसऱ्या मास्टरला क्यूचे स्टेटस ट्रॅक करता येईल. हे सिग्नल खाली दिल्याप्रमाणे इंस्ट्रकशन क्यूची स्थिती प्रदान करतात.

QSI	QSO	
0	0	No Operation
0	1	First byte of op code from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

2. **$\overline{S0}, \overline{S1}, \overline{S2}$ (Status Signals)-(Pin Nos. 26,27,28):**

हे स्टेटस सिग्नल आहेत जे ऑपरेशनची स्थिती प्रदान करतात, जे बस कंट्रोलर 8288 द्वारे मेमरी किंवा I/O कंट्रोल सिग्नल्स तयार करण्यासाठी वापरले जातात. प्रोसेसर खालील प्रमाणे ओपरेशन स्टेटस दर्शवतो.

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Status
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O Port
0	1	0	Write I/O Port
0	1	1	Halt
1	0	0	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

3. **$\overline{\text{LOCK}}$ (Pin No. 29):**

हा सिग्नल प्रोसेसरद्वारे जनरेट केलेला ऍक्टिव्ह लो (LOW) आउटपुट सिग्नल आहे जे इतर सिस्टम बस मास्टर्सना सिस्टम बसवर जसे कि ऍड्रेस बस, डेटा बस आणि कंट्रोल बस, नियंत्रण मिळवायचे नाही हे सूचित करते. हा सिग्नल जनरेट करण्यासाठी LOCK इंस्ट्रकशन वापरली जाते .ह्याच पिनवर प्रोसेसर सिस्टिम बसेस सोडल्याची पोचपावती (Acknowledgment) सिग्नल इतर सिस्टिम मास्टरला पाठवतो.

4. **$\overline{\text{RQ}}/\overline{\text{GT0}}$ $\overline{\text{RQ}}/\overline{\text{GT1}}$: (Pin Nos. 30 and 31)**

करंट प्रोसेसरच्या बस सायकलच्या शेवटी सिस्टम बस जसे कि ऍड्रेस बस, डेटा बस आणि कंट्रोल बस सोडण्याची विनंती करण्यासाठी हे सिग्नल इतर सिस्टम मास्टर्स जसे कि कोप्रोसेसर (Coprocessor) आणि DMA कंट्रोलरद्वारे वापरले जातात. $\overline{\text{RQ}}/\overline{\text{GT1}}$ पेक्षा $\overline{\text{RQ}}/\overline{\text{GT0}}$ ला जास्त प्राधान्य असते.

1.2 8086 चे आर्किटेक्चर (Architecture of 8086)

8086 मायक्रोप्रोसेसरमध्ये दोन मुख्य ब्लॉक आहेत जे Fig. 1.2 मध्ये दर्शविलेले आहे.

a. बस इंटरफेस युनिट-बीआययू (Bus Interface Unit-BIU)

b. एक्झिक्युशन युनिट- ईयू (Execuiton Unit)

a. एक्झिक्युशन युनिट (Execution Unit)

एक्झिक्युशन युनिटची कार्ये खाली दिलेले आहेत:

1. इंस्ट्रक्शन किंवा डेटा कुठून आणायचा हे BIU ला सांगणे.
2. इंस्ट्रक्शन डीकोड करणे
3. इंस्ट्रक्शन एक्सिक्युट करणे

EU मध्ये 16-बिट ALU आहे, जे 8-बिट तसेच 16-बिट डेटावर अरीथमेटिक आणि लॉजिकल ऑपरेशन्स करू शकते.

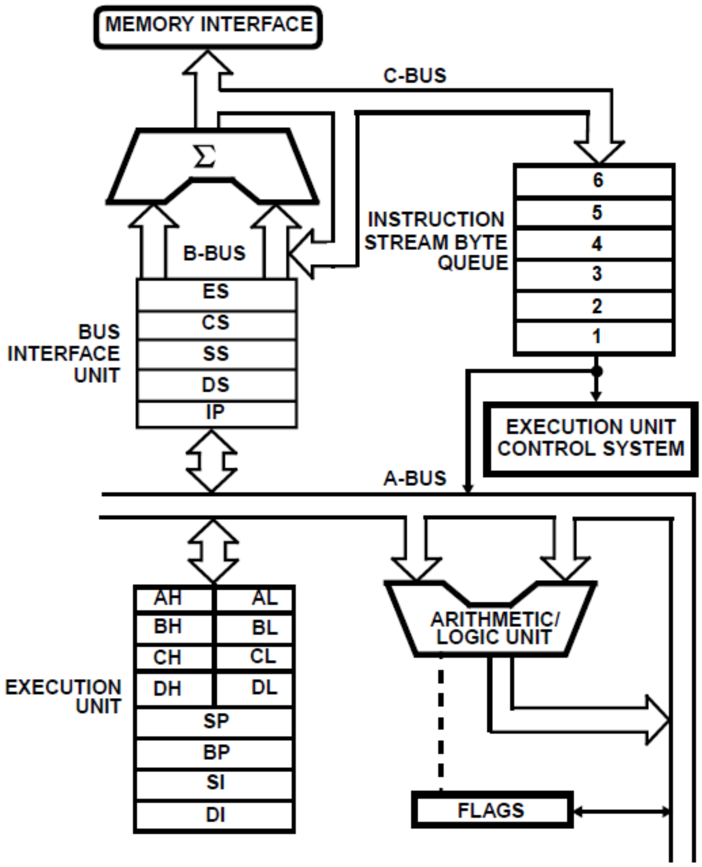


Fig. 1.2: **8086 चे आर्किटेक्चर (Courtesy: Intersil Corporation)**

1. **अरीथमेटिक आणि लॉजिकल युनिट (Arithmetic and Logical Unit-ALU) :** EU मध्ये 16-बिट ALU आहे, जे 8-बिट तसेच 16-बिट डेटावर अरीथमेटिक आणि लॉजिकल ऑपरेशन्स करू शकते.
2. **8086 चे जनरल पर्पज रेजिस्टर्स (General purpose registers of 8086):**

या रजिस्टर्सपैकी AX, BX, CX आणि DX रेजिस्टर्सला एकतर आठ 8-बिट रजिस्टर्स म्हणून वापरले जाऊ शकतात म्हणजे AL, AH, BL, BH, CL, CH, DL, DH किंवा चार 16-बिट म्हणजे AX , BX, CX आणि DX.म्हणून वापरले जाऊ शकतात. AL रजिस्टरला 8-बिट अक्युम्युलेटर (Accumulator) आणि AX ला 16-बिट अक्युम्युलेटर म्हणतात. 8086 चे जनरल पर्पज रेजिस्टर्स Fig. 1.3 मध्ये दिलेले आहेत.

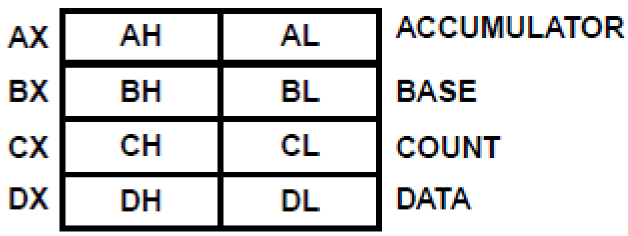


Fig. 1.3: 8086 चे जनरल पर्पज रेजिस्टर्स (General purpose registers)

प्रत्येक जनरल पर्पज रजिस्टर्सची कार्ये खाली नमूद केली आहेत:

- **AX (Accumulator Register) :** जेव्हा ALU अरीथमेटिक आणि लॉजिकल ऑपरेशन्स करते तेव्हा accumulator रजिस्टर अशा ऑपरेशन्सचे ऑपरेंड संग्रहित करते.
- **BX (Base register) :** बेस रजिस्टरचा उपयोग मेमरीमधून डेटा वाचण्यासाठी (Read) आणि मेमोरीमध्ये डेटा लिहिण्यासाठी (Write), त्या मेमरी लोकेशनचा बेस ॲड्रेस ठेवण्यासाठी केला जातो.
- **CX (Counter Register):** काउंटर रजिस्टरचा वापर रोटेट (Rotate) आणि शिफ्ट (Shift) इंस्ट्रकशनचा दरम्यान 8-बिट डेटा ज्याला काउंटर व्हॅलू ठेवण्यासाठी केला जातो. हे लूप इंस्ट्रकशन साठी 16 बिट लूप काउंटर व्हॅलू संचयित करण्यासाठी देखील वापरले जाते.

- **DX (Data Register):** मल्टिप्लिकेशन आणि डिव्हिजन ऑपरेशन्स दरम्यान, जर रिझल्ट 32-बिट्सचा असेल, तर MSB मधील 16 बिट DX रजिस्टरमध्ये आणि LSB मधील 16 बिट्स AX रजिस्टरमध्ये साठवला जातो. हे रजिस्टर I/O पोर्ट साठीचा ॲड्रेस ठेवण्यासाठी देखील वापरले जाते.

3. फ्लॅग रेजिस्टर (Flag Register) : EU मधील फ्लॅग रेजिस्टर (Flag Register) 16-बिट आहे ज्याचा फॉरमॅट (Format) Fig. 1.4 मध्ये दर्शविला आहे.

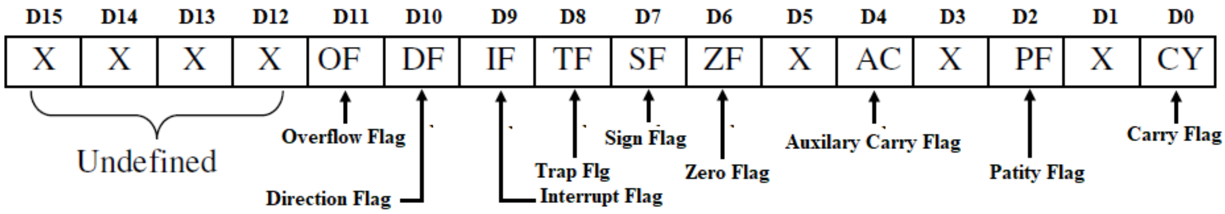


Fig 1.4: 8086 फ्लॅग रेजिस्टर

फ्लॅग रजिस्टरमध्ये दोन प्रकारचे फ्लॅग्स आहेत.

1. स्टेटस फ्लॅग (Status Flag)
2. कंट्रोल फ्लॅग (Control Flag)

(i) स्टेटस फ्लॅग्स (Status Flags)

8 BIT किंवा 16-BIT नंबरवर कोणतेही अरीथमेटिक आणि लॉजिकल (Arithmetic and Logical) ऑपरेशन केल्यानंतर स्टेटस फ्लॅग्स सेट (Set) किंवा रीसेट (Reset) केले जातात. स्टेटस फ्लॅग्स ग्रुप मध्ये एकूण सहा स्टेटस फ्लॅग्स आहेत जी खाली वर्णन केलेली आहे.

1. **कॅरी फ्लॅग (CF - Carry Flag):** 8086 मायक्रोप्रोसेसरमध्ये कोणतेही 8 बिट किंवा 16 बिट अरीथमेटिक आणि लॉजिकल ऑपरेशन केल्यानंतर निकालाच्य (MSB) एमएसबीमधून कॅरी तयार झाल्यास कॅरी फ्लॅग सेट केला जातो.
2. **पॅरिटि फ्लॅग (PF – Parity Flag) :** जेव्हा निकालामधील बायनरी 1 ची संख्या सम (EVEN) असेल तर पॅरिटि फ्लॅग सेट केला जातो आणि विषम (ODD) असल्यास रीसेट केला जातो.
3. **ऑक्सिलरी कॅरी फ्लॅग (AF–Auxiliary Carry Flag):** जेव्हा अरीथमेटिक किंवा लॉजिकल ओपरेशन मध्ये लोवर निब्बल (Lower Nibble) म्हणजेच D_3 बिट कॅरी तयार करतो त्याला ऑक्सिलरी कॅरी फ्लॅग म्हणतात आणि तो सेट होतो. BCD अरीथमेटिक मध्ये हा फ्लॅग वापरला जातो.
4. **झिरो फ्लॅग (ZF – Zero Flag) :** अरीथमेटिक किंवा लॉजिकल ऑपरेशन्स केल्या नंतर जर निकाल (Result) झिरो (0) असेल तर हा फ्लॅग सेट होतो अन्यथा रिसेट होतो.
5. **साइन फ्लॅग (SF – Sign Flag):** अरीथमेटिक किंवा लॉजिकल ऑपरेशन्स केल्या नंतर जर निकाल (Result) निगेटिव्ह (Negative) असेल तर हा फ्लॅग सेट होतो अन्यथा रिसेट होतो.
6. **ओव्हरफ्लो फ्लॅग (OF – Overflow Flag):** जर साईन्ड नंबर्स वर अरीथमेटिक किंवा लॉजिकल ओपरेशन केल्या नंतर जर निकाल (Result) मोठा असेल म्हणजे उपलब्द असलेल्या बिट्स मध्ये बसत नसेल तर ओव्हरफ्लो फ्लॅग सेट होतो अन्यथा तो रीसेट होतो.

(ii) कंट्रोल फ्लॅग्स (Control Flags)

8086 मायक्रोप्रोसेसर मध्ये तीन कंट्रोल फ्लॅग्स आहेत जे मायक्रोप्रोसेसरचे काही ऑपरेशन्स एनेबल (Enable) किंवा डिसेबल (Disable) करण्यासाठी वापरले जातात.

1. **ट्रॅप फ्लॅग (TF – Trap Flag):** ट्रॅप (Trap) फ्लॅग सिंगल स्टेप मोडमध्ये प्रोग्राम डीबग करण्यासाठी वापरला जातो. ट्रॅप फ्लॅग सेट केल्यास (1), CPU आपोआप प्रत्येक इंस्ट्रकशननंतर इंटर्नल इंटरप्ट निर्माण करतो, ज्यामुळे एखाद्या प्रोग्रामची तपासणी करता येते कारण ते इंस्ट्रकशन बाय इंस्ट्रकशन एक्सिक्युट करते. ट्रॅप फ्लॅग रीसेट केल्यास (0), CPU नॉर्मल एक्सिक्युशन करते.

2. **इंटरप्ट फ्लॅग (IF - Interrupt Flag):** हा फ्लॅग इंटरप्ट हॅण्डलिंग साठी म्हणजेच इंटरप्ट एनेबल (Enable) किंवा डिसेबल (Disable) करण्यासाठी वापरला जातो .जर इंटरप्ट फ्लॅग सेट (1) केला तर प्रोसेसर पेरिफेरल्स कडून आलेले इंटरप्टस हाताळलेला जातो .जर इंटरप्ट फ्लॅग रीसेट (0) केला तर प्रोसेसर पेरिफेरल्स कडून आलेले इंटरप्टस हाताळले जात नाही.
3. **डिरेकशन फ्लॅग (DF - Direction Flag):** जर डिरेकशन फ्लॅग सेट केला असेल (1), तर प्रोसेसर हायर मेमरी लोकेशनवरून (Higher memory location) लोवर मेमरी लोकेशनकडे (Lowest memory location) स्ट्रिंग डेटा ऍक्सेस करतो. जर डिरेकशन फ्लॅग रीसेट केला असेल (0), तर प्रोसेसर लोवर मेमरी लोकेशनवरून (Lower memory location) हायर मेमरी लोकेशनकडे (Higher memory location) स्ट्रिंग डेटा ऍक्सेस करतो.

4. स्पेशल पर्पज रेजिस्टर्स (Special purpose registers)

इंडेक्स (Index) आणि बेस पॉईंटर (Base Pointer) रजिस्टरला एकत्रितपणे स्पेशल पर्पज रेजिस्टर्स म्हणून संबोधले जाते. हे मेमरी पॉईंटर्स म्हणून वापरले जाणारे 16-बिट रजिस्टर आहेत आणि या रजिस्टरचा वापर 20-बिट फिजिकल ऍड्रेस तयार करण्यासाठी केला जातो. स्टॅक पॉईंटर (SP), बेस पॉईंटर (BP), सोर्स इंडेक्स (SI), डेस्टिनेशन इंडेक्स (DI) असे चार 16-बिट स्पेशल पर्पज रेजिस्टर्स आहेत जे Fig. 1.5 मध्ये दर्शविले आहे.

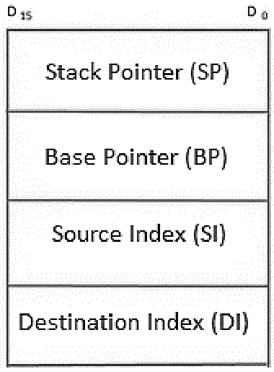


Fig. 1.5: स्पेशल पर्पज रेजिस्टर्स (Special purpose registers)

प्रत्येक स्पेशल पर्पज रेजिस्टर्सचे कार्य खाली नमूद केले आहे.

1. **स्टॅक पॉईंटर रेजिस्टर (SP -Stack Pointer Register):** स्टॅक पॉईंटर रजिस्टर स्टॅक मेमरीचा सर्वात वरचा पत्ता ठेवण्यासाठी वापरला जातो, म्हणजेच हे स्टॅक मेमरी लोकेशनचा ऍड्रेस स्टोअर करते ज्यामध्ये डेटा अलीकडेच स्टोअर केला गेला होता.
2. **बेस पॉईंटर रेजिस्टर (BP -Base Pointer Register) :** याचा वापर मेमरीचा बेस ऍड्रेस स्टोअर करण्यासाठी केला जातो. हे स्टॅक सेगमेंट (SS) साठी ऑफसेट म्हणून कार्य करते आणि हा 16-बिट्स रेजिस्टर आहे.
3. **सोर्स इंडेक्स रेजिस्टर (SI -Source Index Register):** हे एक मेमरी पॉईंटर आहे जे स्ट्रिंग इंस्ट्रक्शन्सचा एक्सिक्युशन दरम्यान सोअर्स स्ट्रिंगचा ऑफसेट ऍड्रेस स्टोअर करण्यासाठी वापरले जाते आणि ज्यांचे बेस सेगमेंट रजिस्टर नेहमीच डेटा सेगमेंट रजिस्टर (DS) असते.
4. **डेस्टिनेशन इंडेक्स रेजिस्टर (DI -Destination Index Register):** हे एक मेमरी पॉईंटर आहे जे स्ट्रिंग इंस्ट्रक्शन्सचा एक्सिक्युशन दरम्यान डेस्टिनेशन स्ट्रिंगचा ऑफसेट ऍड्रेस स्टोअर करण्यासाठी वापरले जाते आणि ज्यांचे बेस सेगमेंट रजिस्टर नेहमीच एक्स्ट्रा सेगमेंट रजिस्टर (ES) असते.

b. बस इंटरफेस युनिट-बीआययू (Bus Interface Unit-BIU)

BIU चे कार्य खालील ऍड्रेस पाठवणे आहे जेणेकरून :

1. मेमरीमधून इंस्ट्रक्शन किंवा डेटा फेंच (Fetch) करणे
2. डेटा मेमरीमध्ये लिहिणे .
3. पोर्टवर डेटा लिहिणे .
4. पोर्टवरील डेटा वाचणे .

BIU मध्ये प्रत्येकी 16-बिटचे 4 सेगमेंट रजिस्टर्स (Segment Registers) आहेत जसे की CS, DS, SS आणि ES जे Fig. 1.6 मध्ये दर्शविले आहे.

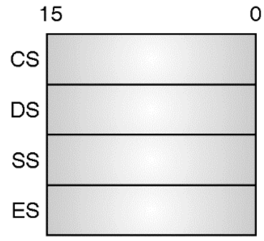


Fig. 1.6: बेस सेगमेंट रेजिस्टर्स (Base Segment registers)

(a) **कोड सेगमेंट (Code Segment):** CS रजिस्टरचा वापर मेमरीच्या कोड सेगमेंटमधील मेमरी लोकेशनला ॲड्रेस करण्यासाठी केला जातो, जिथे प्रोग्रामचा ऑप-कोड (Opcode) स्टोअर केला जातो.

(b) **डेटा सेगमेंट (Data Segment):** DS रजिस्टर मेमरीच्या डेटा सेगमेंटमधील मेमरी लोकेशनला ॲड्रेस करतो, जिथे डेटा स्टोअर केला जातो.

(c) **एक्स्ट्रा सेगमेंट (Extra Segment):** ES रजिस्टरचा वापर सेगमेंटला संबोधित करण्यासाठी केला जातो जो डेटा स्टोर करण्यासाठी वापरला जाणारा अतिरिक्त डेटा सेगमेंट आहे.

(d) **स्टॅक सेगमेंट (Stack Segment):** SS रजिस्टरचा वापर मेमरीच्या स्टॅक सेगमेंटमधील स्टॅक लोकेशन पॉइंट करण्यासाठी केला जातो आणि स्टॅकवर तात्पुरता डेटा साठवण्यासाठी वापरला जातो.

- **इंस्ट्रक्शन क्यू (Instruction queue):**

एक्सिक्युशनचा वेग वाढवण्यासाठी, BIU मेमरीमधून वेळोवेळी जास्तीत जास्त सहा इंस्ट्रक्शन बाइट्स मिळवते. सर्व सहा इंस्ट्रक्शन बाइट्स नंतर फर्स्ट-इन-फर्स्ट-आउट (FIFO) 6-बाइट इंस्ट्रक्शन रजिस्टरमध्ये ठेवल्या जातात ज्याला इंस्ट्रक्शन क्यू IQ म्हणतात, नंतर सर्व इंस्ट्रक्शन बाइट्स EU ला एक-एक करून दिले जातात.

- **इन्स्ट्रक्शन पॉइंटर (IP-Instruction Pointer)**

इन्स्ट्रक्शन पॉइंटर मध्ये पुढील इंस्ट्रक्शनचा ॲड्रेस असतो,जो प्रोग्राममधील पुढील इंस्ट्रक्शन ॲक्सेस करण्यासाठी वापरला जातो.

1.3 पाइपलाइनिंगची संकल्पना (Concept of pipelining):

प्रत्येक क्लॉक सायकलमध्ये इंस्ट्रकशन पूर्ण करण्यासाठी वापरल्या जाणाऱ्या तंत्राला पाइपलाइनिंग म्हणतात. साधारणपणे, पाइपलाइन नसलेल्या प्रोसेसरवर (Non-pipeline processor), तीन इंस्ट्रकशनसाठी फेच (Fetch), डीकोड (Decode) आणि एक्झिक्युट (Execute) सायकलसाठी नऊ क्लॉक सायकल्स आवश्यक असतात जे खालील Fig.1.6 मध्ये दर्शवले आहे .

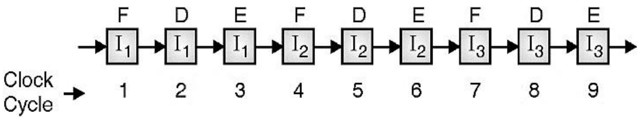


Fig. 1.6 : नॉन पाईपलाईन प्रोसेसर एक्सिक्युशन

परंतु, पाइपलाइन असलेल्या प्रोसेसरवर, फेच (Fetch), डीकोड (Decode) करणे आणि एक्सिक्युट (Execute) करणे ही क्रिया समांतरपणे केली जाते, त्याच तीन इंस्ट्रक्शन एक्सिक्युट करण्यासाठी फक्त पाच क्लॉक सायकल आवश्यक असतात जे खालील Fig.1.7 मध्ये दर्शविलेले आहे.

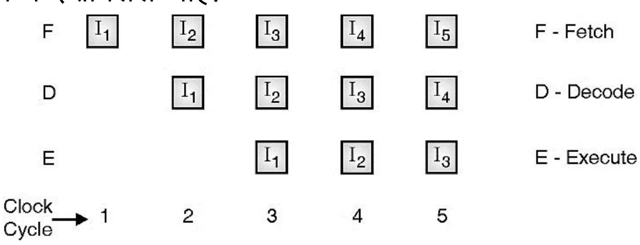


Fig. 1.7 : पाईपलाईन प्रोसेसर एक्सिक्युशन (तीन इंस्ट्रक्शन साठी)

पहिल्या इंस्ट्रक्शनला पूर्ण एक्सिक्युट करण्यासाठी करण्यासाठी तीन क्लॉक सायकल आवश्यक आहेत. नंतर पुढील इंस्ट्रक्शन, प्रति क्लॉक सायकल एक इंस्ट्रक्शन या दराने पूर्ण एक्सिक्युट करते. क्लॉक सायकल 5 दरम्यान प्रोसेसर I3 इंस्ट्रक्शन पूर्ण करते, I4 डीकोड केली जाते आणि I5 फेच केली जाते. सध्याची एक्सिक्युट करत असताना पुढील इंस्ट्रक्शन आणण्याच्या किंवा फेच करण्याचा वैशिष्ट्याला पाइपलाइनिंग म्हणतात ज्यामुळे एक्सिक्युशनचा वेळ कमी होतो, म्हणून, पाइपलाइनिंग प्रोसेसरच्या एक्सिक्युशनची गती वाढवते. 8086 मध्ये, पाइपलाइनिंग साठी 6 बाईट (6 BYTE) इंस्ट्रक्शन क्यू (Instruction Queue) दिलेली आहे, जीथे एक बाईटचा सहा इंस्ट्रक्शन वेळेचा आधी साठवता येते, जिथून एका मागे एक इंस्ट्रक्शन डिकोडिंग आणि एक्सिक्युशनसाठी जाते. म्हणून, क्यू मधील पहिली इंस्ट्रक्शन एक्सिक्युट करताना, प्रोसेसर दुसरी इंस्ट्रक्शन डीकोड करतो आणि मेमरीमधून ८वी इंस्ट्रक्शन क्यू मध्ये आणतो. अशा प्रकारे, एका क्लॉक सायकलेमध्ये 8086 इंस्ट्रक्शन्स समांतरपणे फेच, डिकोड आणि एक्सिक्युट करतो.

1.4 8086 मायक्रोप्रोसेसरमध्ये मेमरी सेगमेंटेशनची संकल्पना (Concept of Memory Segmentation in 8086 Microprocessor):

सेगमेंटेशन ही अशी प्रक्रिया आहे ज्यामध्ये संगणकाची मुख्य मेमरी लॉजिकली वेगवेगळ्या सेगमेंटमध्ये विभागली जाते आणि प्रत्येक सेगमेंटचा स्वतःचा बेस ॲड्रेस असतो. हे मूलत: संगणक प्रणालीच्या एक्सिक्युशनचा वेग वाढविण्यासाठी वापरला जातो, जेणेकरून प्रोसेसर मेमरीमधून डेटा सहजपणे आणि जलद आणण्यास आणि एक्सिक्युशन करण्यास सक्षम असेल. प्रत्येक सेगमेंटचा साईज 64 Kbytes असते आणि एका बेस सेगमेंट रजिस्टर द्वारे ॲड्रेस केले जाते जसे की CS, DS, ES किंवा SS. 16-बिट्स सेगमेंट रजिस्टरमध्ये त्या विशिष्ट सेगमेंटचा स्टार्टींग ॲड्रेस असतो. म्हणून, सेगमेंटमधील विशिष्ट मेमरी स्थानास ॲड्रेस करण्यासाठी ऑफसेट ॲड्रेस किंवा डिस्प्लेसमेंट किंवा इफेक्टिव्ह ॲड्रेस आवश्यक आहे. ऑफसेट ॲड्रेस 16-बिट आहे त्यामुळे कमाल ऑफसेट मूल्य FFFF H असेल आणि म्हणून कोणत्याही सेगमेंटचा कमाल साईज 2^{16} = 64k लोकेशन्स आहे. CPU 8086 फिजिकल मेमरी 1 Mbytes ॲड्रेस करण्यास सक्षम आहे. संपूर्ण 1 Mbytes मेमरी 16 सेगमेंटमध्ये विभागली जाऊ शकते, प्रत्येक सेगमेंट 64 Kbytes साईजचा असतो जे खालील Fig.1.8 मध्ये दर्शविलेले आहे.

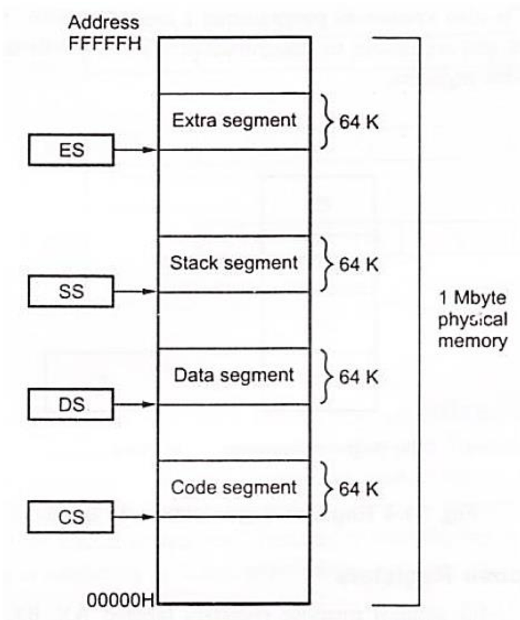


Fig.1.8 : 8086चे मेमरी सेगमेंटेशन

सेगमेंटेशन चे फायदे (Advantages of Segmentation):

1. कोणत्याही इंस्ट्रक्शन किंवा डेटाशी संबंधित ॲड्रेस फक्त 16-बिट्सचा आहे, जरी 8086 मध्ये 20-बिट्स फिजिकल ॲड्रेस आहे.
2. मल्टि युझर टाइम शेअर्ड सिस्टम मध्ये सेगमेंटेशन वापरले जाऊ शकते.
3. प्रोग्राम्स आणि डेटा सेगमेंटेशनमध्ये एकमेकांपासून स्वतंत्रपणे साठवला (Store) जाऊ शकतात.

4. आपण एकापेक्षा जास्त कोड किंवा डेटा सेगमेंट वापरून 64 Kbytes पेक्षा जास्त प्रोग्राम किंवा 64 Kbytes पेक्षा जास्त डेटा वापरू शकतो.
5. सेगमेंटेशनमुळे स्वतंत्र (Independent) किंवा डायनॅमिकली री-लोकॅटेबल असलेले प्रोग्राम लिहिणे शक्य होते.
6. सेगमेंटेशन दोन प्रोसेसना डेटा शेअर करण्यास अनुमती देते.
7. सेगमेंटेशन तुम्हाला प्रोसेसरची अॅड्रेसबिलिटी वाढवण्याची परवानगी देते.

फिझिकल मेमरी ऍड्रेस निर्मिती (Physical Memory Address Generation in 8086):

कोणत्याही सेगमेंटमध्ये कुठलेही मेमरी लोकेशन वापरण्यासाठी 20-बिट ऍड्रेस आवश्यकता आहे ज्याला फिजिकल ऍड्रेस म्हणतात जो ऍड्रेस लाईन्सवर पाठवला जातो. 8086 सेगमेंट रजिस्टर आणि त्याच्याशी संबंधित ऑफसेट रजिस्टरची व्हॅल्यू वापरून हा फिजिकल ऍड्रेस तयार करतो जे खालील Fig.1.9 मध्ये दर्शविलेले आहे.

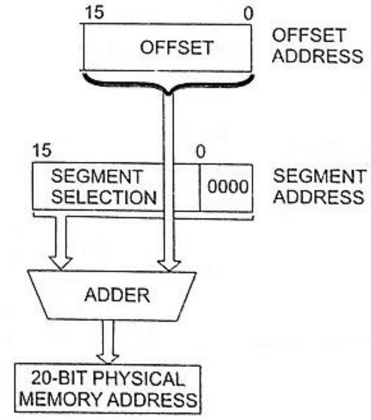


Fig. 1.9 : 8086 मधील फिजिकल मेमरी ऍड्रेस निर्मिती

BIU कडे फिझिकल ऍड्रेस तयार करण्यासाठी एक समर्पित अॅडर आहे ज्यामध्ये बेस आणि लॉजिकल ऑफसेट ऍड्रेसची बेरीज केली जाते. बेरीज करण्यापूर्वी बेस ऍड्रेस चार बिट्सनी डाव्या बाजूला शिफ्ट केला जातो आणि त्या चार बिट्स चा जागेवर चार झिरो (0) इन्सर्ट केले जातात ह्या प्रमाणे बेस ऍड्रेस 16 बिट्सचा 20 बिट्स केला जातो. ह्या 20 बिट्स बेस ऍड्रेस मध्ये 16 बिट्स ऑफसेट ऍड्रेस ऍड केला जातो आणि 20 बिट्स फिझिकल ऍड्रेस तयार होतो जो २० बिट्स ऍड्रेस लाईन्स वर पाठवला जातो.

आता आपण एक खालील उदाहरण बघू या:

आपल्याला माहित आहे की CS रजिस्टरमध्ये कोड सेगमेंटचा बेस ऍड्रेस असतो आणि इंस्ट्रक्शन पॉइंटर (IP) मध्ये कोड सेगमेंटचा पुढील कोड बाइटचा 16-बिट ऑफसेट ऍड्रेस असतो.

समजा CS रेजिस्टर मध्ये 3000H बेस ऍड्रेस आहि आणि IP रेजिस्टरमध्ये 0123H ऑफसेट ऍड्रेस आहे. तर 8086 मायक्रोप्रोसेसर चा BIU युनिट खालील Fig.1.10 मध्ये दर्शविल्या प्रमाणे फिझिकल ऍड्रेस निर्माण करतो.

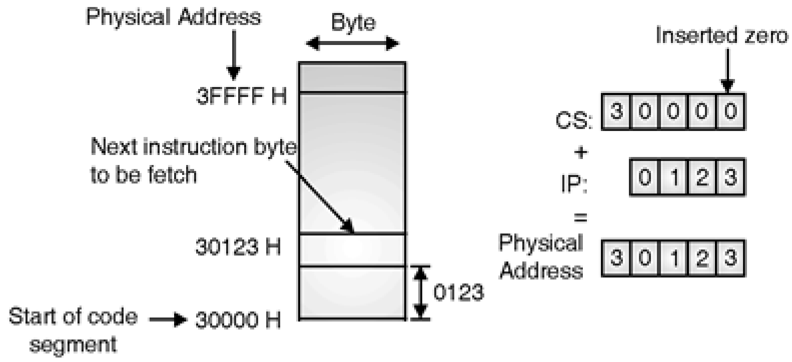


Fig. 1.10 : कोड सेगमेंटचा फिजिकल मेमरी ऍड्रेसची निर्मिती

CS आणि IP ची व्हॅलूची बेरीज करण्यापूर्वी, CS ची व्हॅलू चार बिट्स स्थानाद्वारे डावीकडे शिफ्ट केली जाते. आता CS मध्ये 3000 H आहे, जेव्हा चार बिट पोझिशन्सने डावीकडे शिफ्ट केले जाते, तेव्हा ती व्हॅल्यू 30000 H देते, जो कोड

सेगमेंटचा बेस ॲड्रेस आहे. जेव्हा ऑफसेट 0123 H ॲड करतो , तेव्हा 30123 H पुढील इंस्ट्रकशन बाइटचा 20-बिट फिजिकल ॲड्रेस तयार होतो.

References:

1. Microprocessor and Interfacing (Programming and Hardware) by Douglas V. Hall [McGraw Hill Education, New Delhi ISBN-13: 978-0070257429]
2. The 8088 and 8086 Microprocessors: Programming, Interfacing, Software, Hardware, and Applications by Walter A. Triebel, Avtar Singh [Pearson Publications, New Delhi ISBN-13: 978-0131228047]
3. Microprocessor 8086: Architecture, Programming and Interfacing by Sunil Mathur [PHI, New Delhi ISBN-13: 978-8120340879]
4. Microprocessor X86 Programming by K. R. Venugopal and Raj Kumar [BPB Publications, Delhi ISBN-13: 978-8170294580

युनिट - 2
असेम्ब्ली लँग्वेज प्रोग्रामिंगची कला
(The Art of Assembly Language Programming)

विषय निष्पत्ती (Course Outcome):

CO2: प्रोग्राम डेव्हलपमेंट टूल्स आणि असेंबलर डिरेकटिव्सचे वापर करा.

घटक निष्पत्ती (Theory Learning Outcome):

1. प्रोग्रॅम डेव्हलोपमेंट आणि एक्सिक्युशनच्या दिलेल्या स्टेप्सचे वर्णन करा.
2. असेंब्ली लँग्वेजचा वापर करून दिलेल्या समस्येसाठी कोड विकसित करण्यासाठी स्टेप्स लिहा.
3. निर्दिष्ट प्रोग्रामिंग त्रुटी दुरुस्त करण्यासाठी डीबगरची संबंधित कमांड वापरा.
4. दिलेल्या असेंबलर डिरेक्टीव्हसच्या उदाहरणासह कार्य वर्णन करा.

2.1 प्रोग्राम डेव्हलपमेन्टचे चरण (Program Development Steps)

असेंब्ली लँग्वेज प्रोग्रामिंग डेव्हलप करण्यासाठीचे चरण खाली दिले आहेत.

1. समस्या परिभाषित करणे (Problem definition)
2. अल्गोरिथम (Algorithm)
3. फ्लोचार्ट (Flowchart)
4. आरंभिक चेकलिस्ट (Initialization checklist)
5. इंस्ट्रक्शन्स निवडणे (Choosing instructions)
6. अल्गोरिथमला असेंब्ली लँग्वेज प्रोग्राममध्ये रूपांतरित करणे (Convert algorithm to assembly language program)

1. **समस्या परिभाषित करणे (Problem definition):** प्रोग्राम लिहिण्याची पहिली पायरी म्हणजे आपण प्रोग्राम सोडवायचा असलेल्या समस्येबद्दल परिभाषित करणे आणि याक्षणी आपण प्रोग्राम लिहू नये परंतु आपण काय करू इच्छिता हे आपल्याला माहित असले पाहिजे.
2. **अल्गोरिथम (Algorithm):** अल्गोरिदम ही एक चरण -दर -चरण पद्धत किंवा समस्या सोडवण्याच्या साध्या इंग्रजी भाषेत लिहिलेली विधान आहे.
3. **फ्लोचार्ट (Flowchart):** फ्लोचार्ट प्रोग्राम ऑपरेशन्स किंवा टास्कचे ग्राफिकरित्या प्रतिनिधित्व आहे. फ्लोचार्ट हे प्रोग्राम ऑपरेशन्स किंवा टास्कचे ग्राफिकरित्या प्रतिनिधित्व आहे जे वर्तुळ (Circle), आयत (Rectangle), कर्ण (Diagonal), चौरस (Square)आणि समांतरकरण (Parallelogram) इत्यादी ग्राफिकल चिन्हाद्वारे दर्शविले जाते.
4. **आरंभिक चेकलिस्ट (Initialization checklist):** सामान्यत: प्रोग्राममध्ये अनेक कॉन्स्टन्ट (Constant), व्हेरिएबल (Variable) आणि प्रोसेसरचे विविध भाग जसे कि सेगमेंट रजिस्टर (Segment Register), फ्लॅग्स (Flags), स्टॅक (Stack) इत्यादी असतात जे योग्यरित्या आरंभ करणे आवश्यक आहे.
5. **इंस्ट्रक्शन्स निवडणे (Choosing instructions):** ही स्टेप्स आपल्या समस्येचे ऑपरेशन्स किंवा कार्ये कार्यासाठी योग्य इंस्ट्रक्शन्स निवडणे आहे त्यासाठी आपल्याला मायक्रोप्रोसेसरचा संपूर्ण इंस्ट्रक्शन सेट माहित असणे आवश्यक आहे.
6. **अल्गोरिथमला असेंब्ली लँग्वेज प्रोग्राममध्ये रूपांतरित करणे (Convert algorithm to assembly language program):** एकदा आपण ऑपरेशन्स करण्याच्या इंस्ट्रक्शन्स निवडल्यानंतर अल्गोरिदमनुसार या इंस्ट्रक्शन्सची अनुक्रमे लावून प्रोग्राम तयार करा.

2.2 असेंब्ली लँग्वेज प्रोग्राम डेव्हलोपमेंटचे टूल्स (Assembly language program development tools)

असेंब्ली लँग्वेज प्रोग्राम डेव्हलोपमेंट मध्ये वापरली जाणारी विविध टूल्स खाली दिली आहेत.

a. एडिटर (Editor)
b. असेम्ब्लर (Assembler)
c. लिंकर (Linker)

d. डिबगर (Debugger)

a. **एडिटर (Editor):** असेंब्ली भाषा प्रोग्राम तयार करण्यासाठी एडिटरचा वापर केला जातो. एडिटर मध्ये प्रोग्राम टाईप केल्यानंतर फाईल्सचे एक्स्टेंशन **.asm** असायला पाहिजेएडिटरचे उदाहरण EDIT, नोटपॅड (NOTEPAD) इत्यादी आहेत.

b. **असेम्ब्लर (Assembler):** असेंबलरचा वापर असेंब्ली लँग्वेज प्रोग्राममधील प्रत्येक इंस्ट्रक्शन्सचे योग्य मशीन कोडमध्ये रूपांतरित करण्यासाठी आणि ऑब्जेक्ट फाइल तयार करण्यासाठी केला जातो ज्याचे फाईल एक्स्टेंशन **.obj** असते. असेंबलरची काही उदाहरणे म्हणजे TASM बोरलँडचे टर्बो असेंबलर आणि MASM मायक्रोसॉफ्ट मॅक्रो असेंबलर इत्यादि आहेत.

c. **लिंकर (Linker):** लिंकर बऱ्याच ऑब्जेक्ट फायलींचे मोठ्या ऑब्जेक्ट फायलींमध्ये सामील करण्यासाठी आणि नंतर एक्झिक्युटेबल फाइल तयार करण्यासाठी वापरला जातो ज्याचे एक्स्टेंशन **.exe** असते. लिंकरची काही उदाहरणे म्हणजे TLINK बोरलँडचा टर्बो लिंकर आणि LINK मायक्रोसॉफ्टचा लिंकर इत्यादी आहेत.

d. **डिबगर(Debugger):** डीबगर युझरच्या नियंत्रणाखाली सिंगल स्टेप मोडमध्ये प्रोग्राम एक्सिक्युट करण्यासाठी वापरला जातो. डीबगरची काही उदाहरणे म्हणजे बोरलँडचा टर्बो डीबगर TD, मायक्रोसॉफ्टचा डीबगर कोड व्ह्यू CV इत्यादि आहेत.

2.3 असेम्ब्लर डिरेक्टिव्हस (Assembler directives)

असेंब्ली लँग्वेज प्रोग्राम अनेक राखीव वर्डना सपोर्ट करतो ज्याला किवर्ड (Keyword) किंवा डिरेक्टिव्ह (Directive) म्हणतात आणि त्यांचा उपयोग तुम्हाला प्रोग्राम एकत्रित करण्याच्या आणि प्रोग्रॅम एक्सिक्युशनवर नियंत्रण ठेवण्यास सक्षम करतो. म्हणून डिरेक्टिव्ह हे विधान आहे जे असेंबलरला दिशा देते आणि मशीन कोडमध्ये भाषांतर केले जात नाही. प्रोग्रॅम लिहितांना लागणारे डिरेक्टिव्ह खाली नमूद केले आहे.

1. **db (Define byte):** हा डिरेक्टिव्ह बाईट म्हणजेच 8 -बिट टाईप व्हेरिएबल आहे आणि एक किंवा अनेक बाईट व्हेरिएबल साठी वापरला जातो.

   रचना (Syntax): variable name db initialization value(s)

उदाहरण:	num	db	?
	array	db	1,2,3,4,5,
	name	db	‘MSBTE$’
	earray	db	10 dup()

2. **dw (Define Word):** हा डिरेक्टिव्ह वर्ड म्हणजेच 16 -बिट किंवा दोन बाईट टाईप व्हेरिएबल आहे आणि एक किंवा अनेक वर्ड व्हेरिएबल साठी वापरला जातो.

   रचना(Syntax): variable name dw initialization value(s)

उदाहरण:	num	dw	?
	array	dw	1,2,3,4,5,
	earray	dw	10 dup()

3. **dd (Define Double Word):** हा डिरेक्टिव्ह डबल वर्ड म्हणजेच 32 -बिट किंवा दोन वर्ड किंवा चार बाईट टाईप व्हेरिएबल आहे आणि एक किंवा अनेक डबल वर्ड व्हेरिएबल साठी वापरला जातो.

   रचना(Syntax): variable name dd initialization value(s)

उदाहरण:	num	dd	?
	array	dd	1,2,3,4,5,
	earray	dd	10 dup()

4. **dq (Define Quad Word):** हा डिरेक्टिव्ह क्वाड वर्ड म्हणजेच 64 -बिट किंवा दोन डबल वर्ड किंवा चार वर्ड किंवा आठ बाईट टाईप व्हेरिएबल आहे आणि एक किंवा अनेक क्वाड वर्ड व्हेरिएबल साठी वापरला जातो.

   रचना(Syntax): variable name dq initialization value(s)

उदाहरण:	num	dq	?

array dq 1,2,3,4,5,
earray dq 10 dup()

5. **dt (Define Ten Byte):** हा डिरेक्टिव्ह दहा बाईट टाईप व्हेरिएबल आहे आणि एक किंवा अनेक टेन बाईट व्हेरिएबल साठी वापरला जातो.

   रचना(Syntax): variable name dq initialization value(s)

   उदाहरण: num dq ?
   array dq 1,2,3,4,5,
   earray dq 10 dup()

6. **equ (Equate to):** हा डिरेक्टिव्हचा वापर चिन्हे (Symbol) घोषित करण्यासाठी केला जातो ज्यांना काही स्थिर व्हॅल्यू असाईन्ड केले जाते. अशा चिन्हांना मॅक्रो चिन्हे (Micro Symbol) म्हणतात, म्हणून मॅक्रो असेंबलर प्रोग्राममधील वापरलेल्या चिन्हाच्या प्रत्येक जागी त्याचा व्हॅल्यूने बदलली जाते.

   रचना(Syntax): variable name equ expression or value

   उदाहरण: num equ 10H
   Inc_fact equ 2

7. **org (Originate):** हा डिरेक्टिव्ह डिरेक्टिव्हमध्ये निर्दिष्ट केलेल्या व्हॅल्यूचा उपयोग करून लोकशन काउंटर (Location counter) असाईन्ड करते. हे असेंबलरद्वारे मशीन कोडमध्ये इंस्ट्रक्शनचे भाषांतर करताना विशिष्ट ठिकाणी मशीन कोड ठेवण्यास मदत करते.

   रचना(Syntax): org [$+] numeric value

   उदाहरण: org 100
   org $
   org $+200

8. **dup (Duplicate memory location): dup** डिरेक्टिव्ह अनेक बाइट्स किंवा वर्ड जनरेट करण्यासाठी वापरला जाऊ शकतो.

   उदाहरण: array db 10 dup()

9. **ASSUME directive:** ASSUME डिरेक्टिव्ह असेंबलरला लॉजिकल सेगमेंटचे नाव सूचित करतो जो निर्दिष्ट सेगमेंटसाठी वापरला जातो.

   रचना(Syntax): ASSUME segment register : segment name………………

   उदाहरण: ASSUME CS:My_code, DS:My_data

10. **SEGMENT directive**: SEGMENT डिरेक्टिव्ह लॉजिकल सेगमेंटची सुरूवात दर्शविण्यासाठी वापरला जातो. SEGMENT आणि ENDS डिरेक्टिव्हच्या मध्ये सेगमेंट डेटा (data), कोड (code), एक्सट्रा(extra) किंवा प्रोग्रामचा स्टॅक (stack) संलग्न करणे आवश्यक असते.

    रचना(Syntax): Segment_Name SEGMENT [WORD/PUBLIC]

    उदाहरण:

    (a) My_Data SEGMENT
    ……….
    प्रोग्रामचा डेटा…….
    ……….
    My_Data ENDS

    (b) My_Code SEGMENT
    ……….
    प्रोग्रामचा कोड……

……….

My_Code ENDS

11. **ENDS (End of the segment):** ENDS डिरेक्टिव्ह असेंबलरला सेगमेंटच्या शेवटी सूचित करतो. SEGMENT आणि ENDS डिरेक्टिव्हच्या मध्ये सेगमेंट डेटा (data), कोड (code), एक्सट्रा(extra) किंवा प्रोग्रामचा स्टॅक (stack) संलग्न करणे आवश्यक असते.

    रचना(Syntax): Segment_Name ENDS

    उदाहरण:

    (a) My_Data SEGMENT

    ……….

    प्रोग्रामचा डेटा…….

    ……….

    My_Data ENDS

    (b) My_Code SEGMENT

    ……….

    प्रोग्रामचा कोड……

    ……….

    My_Code ENDS

12. **END (End of the program):** END डिरेक्टिव्ह चा वापर असेंबलरला प्रोग्रॅमचा शेवट सूचित करण्यासाठी केला जातो.

    रचना(Syntax): END

13. **.CODE (Simplified CODE segment directive):** सर्व एक्झिक्युटेबल कोड या कोड सेगमेंट मध्ये ठेवणे आवश्यक आहे.
14. **.DATA (Simplified DATA segment directive):** सर्व डेटा डेफिनेशन्स आणि डिक्लेरेशन्स या सेगमेंटमध्ये ठेवणे आवश्यक आहे.
15. **.STACK (Simplified STACK segment directive):** हे सेगमेंट डिरेक्टिव्ह स्टॅक सेगमेंट परिभाषित करते आणि स्टॅकचा डीफॉल्ट आकार 1024 बाइट्स आहे.
16. **.MODEL (Memory model declaration for segments):** डिरेक्टिव्ह .MODEL डीफॉल्ट सेगमेंट तयार करतात.

    रचना(Syntax): .MODEL type of model

    Type of Models:

    - TINY : .COM प्रोग्रामसाठी वापरले जाते.
    - SMALL : एका सेगमेंटमधील सर्व डेटा आणि एका सेगमेंटमधील सर्व कोड.
    - MEDIUM : एका सेगमेंटमधील सर्व डेटा पण एकापेक्षा जास्त कोड सेगमेंट
    - COMPACT : एका सेगमेंटमधील सर्व कोड पण एकापेक्षा जास्त डेटा सेगमेंट
    - LARGE : दोन्ही डेटा आणि कोड सेगमेंट एकापेक्षा जास्त पण सेगमेंटची साईझ 64K पेक्षा जास्त नाही.
    - HUGE : दोन्ही डेटा आणि कोड सेगमेंट एकापेक्षा जास्त पण सेगमेंटची साईझ 64K पेक्षा जास्त असू शकते

17. **OFFSET Directive**: OFFSET डिरेक्टिव्ह सेगमेंटच्या बेस ॲड्रेस पासून निर्दिष्ट व्हेरिएबलचे डिस्प्लेसमेंट निर्धारित करण्यासाठी असेंबलरला सूचित करते आणि हे सहसा रजिस्टरमध्ये व्हेरिएबलचे ऑफसेट लोड करण्यासाठी वापरले जाते. या ऑफसेट व्हॅल्यूचा वापर करून, इंडेक्सड ॲड्रेसिंग मोड वापरून व्हेरिएबलचा संदर्भ दिला जाऊ शकतो.

    रचना(Syntax): OFFSET variable_name

उदाहरण: MOV SI, OFFSET ARRAY

18. **PTR (Pointer):** PTR डिरेक्टिव्ह मेमरी ऍक्सेसचा प्रकार दर्शविण्यासाठी वापरला जातो म्हणजे BYTE/WORD/DWORD.

    उदाहरण: MOV SI, BYTE PTR NUM

References:

1. Microprocessor and Interfacing (Programming and Hardware) by Douglas V. Hall [McGraw Hill Education, New Delhi ISBN-13: 978-0070257429]
2. The 8088 and 8086 Microprocessors: Programming, Interfacing, Software, Hardware, and Applications by Walter A. Triebel, Avtar Singh [Pearson Publications, New Delhi ISBN-13: 978-0131228047]
3. Microprocessor 8086: Architecture, Programming and Interfacing by Sunil Mathur [PHI, New Delhi ISBN-13: 978-8120340879]
4. Microprocessor X86 Programming by K. R. Venugopal and Raj Kumar [BPB Publications, Delhi ISBN-13: 978-8170294580

युनिट - 3
8086 चा इंस्ट्रक्शन सेट
(Instruction Set of 8086 Microprocessor)

विषय निष्पत्ती (Course Outcome):

CO3: वेगवेगळ्या ॲड्रेसिंग मोड्समध्ये इंस्ट्रक्शन्स वापरा.

घटक निष्पत्ती (Theory Learning Outcome):

1. दिलेल्या इंस्ट्रक्शनची लांबी निश्चित करा.
2. उदाहरणांसह दिलेल्या ॲड्रेसिंग मोड्सचे वर्णन करा.
3. दिलेल्या इंस्ट्रक्शनचे एक्सिक्युशन दरम्यान केलेल्या ऑपरेशनचे स्पष्टीकरण द्या.
4. दिलेल्या इंस्ट्रक्शनचा ॲड्रेसिंग मोड ओळखा.

3.1 मशीन लँग्वेज इंस्ट्रक्शन फॉरमॅट (Machine Language Instruction Format)

मशीन लँग्वेज इंस्ट्रक्शन फॉरमॅटमध्ये त्याच्याशी संबंधित फील्डची एक किंवा अधिक संख्या असते. पहिल्या फील्डला ऑपरेशन कोड फील्ड (Operation code field) किंवा ऑप-कोड फील्ड (Op-code field) म्हणतात, जे CPU द्वारे केले जाणारे ऑपरेशनचे प्रकार सूचित करते. दुसऱ्या फील्डला ऑपरेंड फील्ड (Operand field) म्हणतात, म्हणजे डेटा फील्ड ज्यावर CPU निर्देश ऑप-कोड द्वारे निर्दिष्ट केलेले ऑपरेशन करते. 8086 इंस्ट्रक्शन सेटमध्ये इंस्ट्रक्शनचे सहा सामान्य स्वरूप आहेत. इंस्ट्रक्शनची लांबी एक बाइट (One byte) ते सहा बाइट्स (Six byte) पर्यंत असू शकते. इंस्ट्रक्शन स्वरूपांची खाली दिलेली आहे.

1. **एक बाईट इंस्ट्रक्शन (One byte instruction)** : हे फॉरमॅट फक्त एक बाइट लांब आहे आणि त्यात इम्प्लिसिट डेटा (Implicit data) किंवा रेजिस्टर ऑपरेंड असू शकतात. रेजिस्टर ऑपरेंड निर्दिष्ट करण्यासाठी ऑप-कोडचे लोवर 3 बिट वापरले जातात, जर असेल तर. अन्यथा, सर्व 8 बिट्स एक ऑप-कोड बनवतात आणि ऑपरेंड्स इम्प्लाइड (Implied) असतात.
2. **रेजिस्टर टू रेजिस्टर (Register-to-register):** रेजिस्टर टू रेजिस्टर इंस्ट्रक्शनचे स्वरूप 2 बाइट लांब आहे. कोडचा पहिला बाइट इंस्ट्रक्शनचा ऑपरेशन कोड दर्शवतो, म्हणजे ऑप-कोड आणि डब्ल्यू बिटने निर्दिष्ट केलेल्या ऑपरेंडची रुंदी W बिटने दर्शविले जाते. इंस्ट्रक्शन कोडचा दुसरा बाइट खाली Fig 3.1 मध्ये दाखवल्याप्रमाणे रजिस्टर ऑपरेंड आणि R/M फील्ड सूचित करतो. REG फील्डमध्ये निर्दिष्ट केलेले रेजिस्टर हे रजिस्टर ऑपरेंडपैकी एक आहे. R/M फील्ड दुसरे ऑपरेंड सूचित करते जे कदाचित रजिस्टर किंवा मेमरी लोकशन असू शकते.

First Byte

D7	D6	D5	D4	D3	D2	D1	D0
Opcode							W

Second Byte

D7	D6	D5	D4	D3	D2	D1	D0
1	1	Register			R/M		

Fig. 3.1: रेजिस्टर टू रेजिस्टर फॉरमॅट

3. **रेजिस्टर ते/मधून मेमरी डिस्प्लेसमेंट रहित (Register to/from memory with no displacement):** या प्रकारच्या इंस्ट्रक्शनमध्ये, इंस्ट्रक्शनचे स्वरूप 2 बाइट्सचे असते. पहिला बाइट हा रजिस्टर-टू-रजिस्टर फॉरमॅट सारखाच आहे परंतु दुसऱ्या बाइटमध्ये खाली Fig. 3.2 मध्ये दाखवल्याप्रमाणे MOD फील्ड आहे. MODE, R/M, REG आणि W फील्ड Table 3.1, 3.2 आणि 3.3 मध्ये दिलेले आहेत.

First Byte

D7 D6 D5 D4 D3 D2 D1	D0
Opcode	W

Second Byte

D7 D6	D5 D4 D3	D2 D1 D0
Mode	Register	R/M

Fig.3.2: रेजिस्टर ते/मधून मेमरी डिस्प्लेसमेंट रहित

4. **रेजिस्टर ते/मधून मेमरी डिस्प्लेसमेंट सहित (Register to/from memory with displacement):** या प्रकारच्या इंस्ट्रक्शन फॉरमॅटमध्ये डिस्प्लेसमेंटसाठी एक किंवा दोन अतिरिक्त बाइट्स आणि मेमरी टू/फ्रॉम रजिस्टरचे दोन बाइट्स फॉरमॅट खालील Fig. 3.3 मध्ये दाखवले आहे.

First Byte

D7 D6 D5 D4 D3 D2 D1	D0
Opcode	W

Second Byte

D7 D6	D5 D4 D3	D2 D1 D0
Mode	Register	R/M

Third Byte

D7 D6 D5 D4 D3 D2 D1 D0
Lower byte displacement

Fourth byte

D7 D6 D5 D4 D3 D2 D1 D0
Higher byte displacement

Fig. 3.3: रेजिस्टर ते/मधून मेमरी डिस्प्लेसमेंट सहित

5. **ईमेडिएट ऑपरंड टू रेजिस्टर (Immediate operand to register):** या प्रकारात, इंस्ट्रक्शन फॉरमॅट मध्ये प्रथम बाइट्स तसेच दुसऱ्या बाइटमधील 3-बिट्स, जे REG फील्डसाठी वापरले जातात, जर ते रजिस्टर-टू-रेजिस्टर फॉरमॅट असेल, तर ऑप-कोडसाठी वापरले जातात. या प्रकारात, इंस्ट्रक्शन फॉरमॅटमध्ये खाली Fig. 3.4 मध्ये दाखवल्याप्रमाणे ईमेडिएट डेटाचे एक किंवा दोन बाइट्स समाविष्ट आहेत.

First Byte

D7 D6 D5 D4 D3 D2 D1 D0
Opcode

Second Byte

D7 D6	D5 D4 D3	D2 D1 D0
Mode	Opcode	R/M

Third Byte

D7 D6 D5 D4 D3 D2 D1 D0
Lower byte data

Fourth byte

D7 D6 D5 D4 D3 D2 D1 D0
Higher byte data

Fig. 3.4: ईमेडिएट ऑपरंड टू रेजिस्टर

6. **ईमेडिएट ऑपरंड टू मेमरी डिस्प्लेसमेंट सहित (Immediate operand to memory with 16-bit displacement):** या प्रकारच्या इंस्ट्रक्शनचा फॉरमॅटची लांबी पाच किंवा सहा बाइट इतकी असते. पहिल्या दोन सलग बाइट्समध्ये OP-CODE, MODE आणि R/M फील्डची माहिती असते. शेवटच्या चार बाइट्समध्ये पहिल्या दोन बाइट्स डिस्प्लेसमेंटचे असते आणि नंतरचे दोन बाइट्सचा डेटाचे असतात जे खालील Fig. 3.5 मध्ये दर्शविला आहे.

First Byte

D7 D6 D5 D4 D3 D2 D1 D0

Opcode

Second Byte

D7 D6 D5 D4 D3 D2 D1 D0

Mode	Opcode	R/M

Third Byte

D7 D6 D5 D4 D3 D2 D1 D0

Lower byte displacement

Fourth byte

D7 D6 D5 D4 D3 D2 D1 D0

Higher byte displacement

Fifth byte

D7 D6 D5 D4 D3 D2 D1 D0

Lower byte data

Sixth byte

D7 D6 D5 D4 D3 D2 D1 D0

Higher byte data

Fig. 3.5: ईमेडिएट ऑपरंड टू मेमरी डिस्प्लेसमेंट सहित

Table 3.1

W-bit	Register address bits	Registers (8/16 bit)	W-bit	Register address bits	Registers (8/16 bit)
0	000	AL	1	000	AX
0	001	CL	1	001	CX
0	010	DL	1	010	DX
0	011	BL	1	011	BX
0	100	AH	1	100	SP
0	101	CH	1	101	BP
0	110	DH	1	110	SI
0	111	BH	1	111	DI

Table 3.2

Segment register (2 bit code)	Segment register (16 bit)
00	ES
01	CS
10	SS
11	DS

Table 3.3

Operands	Memory operands			8 /16 bit Register operands	
	without offset	With 8-bits displacement	With 16-bits displacementt		
MOD R/M	00	01	10	11	
				W=0	W=1
000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16	AL	AX
001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16	CL	CX
010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16	DL	DX
011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16	BL	BX
100	(SI)	(SI) + D8	(SI) + D16	AH	SP
101	(DI)	(DI) + D8	(DI) + D16	CH	BP
110	D16 (Direct address)	(BP) + D8	(BP) + D16	DH	SI
111	(BX)	(BX) + D8	(BX) + D16	BH	DI

3.2 ॲड्रेसिंग मोड्स (Addressing Modes)

मेमरी, रजिस्टर किंवा I/O मध्ये डेटा किंवा ऑपरेंड शोधण्यासाठी ॲड्रेसिंग मोडचा वापर केला जातो. 8086 ची कोणतीही इंस्ट्रक्शन एक किंवा अधिक ॲड्रेसिंग मोडमध्ये वापरली जाऊ शकते परंतु काही इंस्ट्रक्शन कोणत्याही ॲड्रेसिंग मोडमध्ये वापरल्या जाऊ शकत नाहीत कारण त्याचा डेटा टाईपवर अवलंबून असते.

ॲड्रेसिंग मोडचे प्रकार (Type of addressing Mode)

1. ईमेडिएट अड्रेसिंग मोड (Immediate addressing mode)
2. डायरेक्ट अड्रेसिंग मोड (Direct addressing mode)
3. रेजिस्टर अड्रेसिंग मोड (Register addressing mode)
4. रेजिस्टर इंडिरेक्ट अड्रेसिंग मोड (Register Indirect addressing mode)
5. रेजिस्टर रिलेटिव अड्रेसिंग मोड (Register Relative addressing mode)
6. बेस इंडेक्स अड्रेसिंग मोड (Base Index addressing mode)
7. रिलेटिव बेस इंडेक्स अड्रेसिंग मोड (Relative base index addressing mode)

1. ईमेडिएट अड्रेसिंग मोड (Immediate addressing mode)

या मोडमध्ये, ईमेडिएट डेटा हा इंस्ट्रक्शनचा भाग असतो आणि ऑप-कोड बाइट्सनंतर बाइट किंवा बाइट्सच्या स्वरूपात दिसून येतो. त्यामुळे ईमेडिएट डेटा 8 बिट [बाइट] किंवा 16 बिट [वर्ड] लांबीचा असू शकतो. इंस्ट्रक्शन क्यू मध्ये उपलब्ध असल्याने ईमेडिएट डेटा त्वरीत ॲक्सेस केला जाऊ शकतो, त्यामुळे डेटा वाचण्यासाठी कोणत्याही अतिरिक्त बस सायकलची आवश्यकता नाही.

उदाहरण: MOV AL, 64H
MOV BX, 4321H

2. डायरेक्ट अड्रेसिंग मोड (Direct addressing mode)

या मोडमध्ये, ऑपरेंडचा 16-बिट मेमरी ॲड्रेस (ऑफसेट) त्याचा एक भाग म्हणून इंस्ट्रक्शनमध्ये थेट नमूद केला जातो. डिस्प्लेसमेंटचा ऑफसेट एकतर 8 बिट किंवा 16 बिट असू शकतो जो इंस्ट्रक्शन ऑप-कोड सोबत दिलेला असतो. तर, हा ऑफसेट बेस सेगमेंट रजिस्टर्स म्हणजेच CS, DS, ES, SS मध्ये जोडून फिझिकल ॲड्रेस काढला जातो.

उदाहरण: MOV AL, [4000H]

3. रेजिस्टर अड्रेसिंग मोड (Register addressing mode)

या मोडमध्ये, डेटा रजिस्टरमध्ये स्टोअर केला जातो आणि तो विशिष्ट रजिस्टर वापरून संदर्भित केला जातो म्हणजेच आयपी (IP) वगळता सर्व रजिस्टर या मोडमध्ये वापरले जाऊ शकतात. रेजिस्टर हे सोर्स ऑपरेंड (Source Operand),

डेस्टिनेशन ऑपरेंड (Destination Operand) किंवा दोन्ही असू शकतात. रेजिस्टर 8 बिट किंवा 16 बिट असू शकतात. या मोडमध्ये, ज्या इंस्ट्रक्शन मध्ये दोन ऑपरंड आहे तिथे सोर्स ऑपरेंड (Source Operand) आणि डेस्टिनेशन ऑपरेंड (Destination Operand) सारख्या साईझचे असायला पाहिजे.

उदाहरण: MOV AX, CX

4. रेजिस्टर इंडिरेक्ट अड्रेसिंग मोड (Register Indirect addressing mode)

या मोडमध्ये, BX, SI, DI रजिस्टर सारख्या ऑफसेट रजिस्टरचा वापर करून, डेटा किंवा ऑपरेंड असलेल्या मेमरी स्थानाचा ऍड्रेस अप्रत्यक्षपणे उपलब्ध असतो. वापरलेल्या इंस्ट्रक्शन नुसार, डीफॉल्ट सेगमेंट रजिस्टर हा एकतर DS किंवा ES असतो. जर BP वापरला असेल, तर SS हे डिफॉल्ट सेगमेंट रजिस्टर असतो.

उदाहरण: MOV AX, [BX]

इंडेक्स ॲड्रेसिंग मोड:

या मोडमध्ये, ऑपरेंडचा ऑफसेट कोणत्याही एका इंडेक्स रजिस्टरमध्ये जसे की SI आणि DI मध्ये संग्रहित केला जातो. DS आणि ES हे अनुक्रमे इंडेक्स रजिस्टर SI आणि DI साठी डीफॉल्ट सेगमेंट आहेत जे वापरलेल्या इंस्ट्रक्शनवर अवलंबून असते.

उदाहरण: MOV BL, [SI]

5. रेजिस्टर रिलेटिव अड्रेसिंग मोड (Register Relative addressing mode)

या मोडमध्ये, डीफॉल्ट DS आणि ES सेगमेंटमध्ये, BX, BP, SI आणि DI सारख्या कोणत्याही एका रजिस्टरच्या डेटासह 8-बिट किंवा 16-बिट डिस्प्लेसमेंट जोडून तयार केलेल्या इफेक्टिव्ह ऍड्रेस काढला जातो ज्यावर डेटा उपलब्ध आहे.

उदाहरण: MOV AX, 50[BX]

6. बेस इंडेक्स अड्रेसिंग मोड (Base Index addressing mode)

या मोडमध्ये, DS किंवा ES सह इंडेक्स रजिस्टर SI किंवा DI च्या डेटामध्ये बेस रजिस्टर BX किंवा BP ची डेटा जोडून डेटाच्या इफेक्टिव्ह ऍड्रेस काढला जातो.

उदाहरण: MOV AX, [BX][SI] or MOV AX, [BX+SI]

7. रिलेटिव बेस इंडेक्स अड्रेसिंग मोड (Relative base index addressing mode)

या मोडमध्ये, DS आणि ES या डिफॉल्ट सेगमेंट मधील बेस रजिस्टर BX किंवा BP आणि SI किंवा DI रजिस्टरच्या बेरीजसह 8-बिट किंवा 16-बिट्स डिस्प्लेसमेंट जोडून डेटाचा ऑफसेट ऍड्रेस काढला जातो.

उदाहरण: MOV AX, 60[BX][SI]

3.3 8086 चा इंस्ट्रक्शन सेट Instruction Set of 8086)

8086 चे इंस्ट्रक्शन सेट्स त्यांच्या कार्यांनुसार खालील प्रमाणे ग्रुप केलेले आहेत.

1. डेटा ट्रान्स्फर इंस्ट्रक्शन ग्रुप (Data transfer instruction group)
2. अरीथमेटिक इंस्ट्रक्शन ग्रुप (Arithmetic instruction group)
3. लॉजिकल इंस्ट्रक्शन ग्रुप (Logical instruction group)
4. प्रोग्राम कंट्रोल ट्रान्सफर किंवा ब्रांचिंग इंस्ट्रक्शन्स ग्रुप (Program control transfer or branching instructions group)
5. मशीन किंवा प्रोसेस कंट्रोल इंस्ट्रक्शन ग्रुप (Machine or process control instruction group)
6. फ्लॅग म्यानिप्युलेशन इंस्ट्रक्शन ग्रुप (Flag manipulation instruction group)
7. शिफ्ट आणि रोटेट इंस्ट्रक्शन ग्रुप (Shift and rotate instruction group)
8. स्ट्रिंग ऑपरेशन्स इंन्स्ट्रक्शन ग्रुप (String operation instruction group)

3.3.1 डेटा ट्रान्स्फर ग्रुप (Data transfer group)

या गटाच्या इंस्ट्रक्शनचा वापर सोअर्स कडून डेस्टिनिशनकडे डेटा ट्रान्स्फर करण्यासाठी केला जातो जेथे सोअर्स रेजिस्टर (Register), मेमरी लोकेशन (Memory locations)किंवा इमिडिएट डेटा (Immediate data) असू शकतो आणि डेस्टिनिशन रेजिस्टर, मेमरी लोकेशन असू शकते.

1. MOV destination, source

ही इंस्ट्रक्शन सोअर्स (Source) पासून डेस्टिनिशन (Destination) मध्ये डेटा कॉपी करण्यासाठी वापरली जाते, जिथे सोअर्स हे रेजिस्टर/मेमरी लोकेशन/इमिडिएट डेटा ह्या पैकी एक असू शकतो आणि डेस्टिनिशन हे दुसरे रजिस्टर/मेमरी लोकेशन असू शकते. सोअर्स आणि डेस्टिनिशनची साईझ समान असणे आवश्यक आहे. सोअर्स आणि डेस्टिनिशन दोन्ही मेमरी लोकेशन्स असायला नको. डेस्टिनिशन हा इमिडिएट डेटा असायला नको.

ऑपरेशन: destination ← source

उदाहरण: MOV AX, BX
MOV BX, [3000H]
MOV AL, [SI]

2. PUSH source

ही इंस्ट्रक्शन सोअर्सपासून स्टॅक (Stack) लोकेशनवर वर्ड स्टोअर करण्यासाठी वापरली जाते. स्टॅक पॉइंटर कमी केला जातो आणि नंतर स्टॅक सेगमेंटमध्ये जिथे स्टॅक पॉइंटर पॉइंट करतो त्या स्थानावर सोअर्सपासून शब्द वर्ड स्टोअर करतो. 16 बिट सोर्स हा जनरल पर्पज रजिस्टर, सेगमेंट रजिस्टर किंवा 16 बिट मेमरी लोकेशन असणे आवश्यक आहे.

ऑपरेशन: SP ← SP –2
SS:[SP] ← MSB of source
SS:[SP – 1] ← LSB of source

उदाहरण: PUSH BX

3. POP destination

ही इंस्ट्रक्शन स्टॅक पॉइंटरद्वारे निर्देशित केलेल्या स्टॅक लोकेशनमधून एक वर्ड निर्देशित केलेल्या डेस्टिनेशन मध्ये कॉपी करतो. 16 बिट डेस्टिनेशन हा जनरल पर्पज रजिस्टर, सेगमेंट रजिस्टर किंवा 16 बिट मेमरी लोकेशन असणे आवश्यक आहे. स्टॅकवर पुढील वर्ड निर्देशित करण्यासाठी, स्टॅक पॉइंटर आपोआप 2 ने वाढविला जातो आणि नंतर तो वर्ड निर्देशित डेस्टिनेशनमध्ये कॉपी केला जातो.

ऑपरेशन: LSB of destination ← SS:[SP].
MSB of destination ← SS:[SP+1].
SP ← SP + 2.

उदाहरण: POP DX

4. XCHG destination, source

ही इंस्ट्रक्शन रजिस्टरमधील डेटा दुसऱ्या रजिस्टर किंवा मेमरी स्थानाच्या डेटासह अदलाबदली करते. हि इंस्ट्रक्शन दोन मेमरी लोकेशन्सच्या डेटाची थेट अदलाबदली करू शकत नाही. सोअर्स आणि डेस्टिनेशन दोन्ही वर्ड असणे आवश्यक आहे किंवा ते दोन्ही बाइट असणे आवश्यक आहे. या इंस्ट्रक्शनमध्ये सेगमेंट रजिस्टर वापरले जाऊ शकत नाही.

ऑपरेशन: destination ←→ source

उदाहरण: XCHG AX, BX

5. IN accumulator, port

IN इंस्ट्रक्शन पोर्टवरून दिलेल्या डेस्टिनेशन डेटा कॉपी करते जे AL किंवा AX (Accumulator) रेजिस्टर असू शकतो. पोर्टचा ऍड्रेस थेट किंवा अप्रत्यक्षपणे इंस्ट्रक्शनमध्ये निर्दिष्ट केला जाऊ शकतो. फिक्स्ड पोर्टसाठी (Fixed Port) पोर्टचा 8-बिट ऍड्रेस थेट इंस्ट्रक्शनमध्ये नमूद केला जातो. व्हेरिएबल पोर्टसाठी (Variable Port) पोर्टचा 16 बिट ऍड्रेस फक्त DX रजिस्टरमध्ये निर्दिष्ट केला जातो. म्हणून DX रजिस्टर मध्ये नेहमी IN इंस्ट्रक्शनपूर्वी 16-बिट्स पोर्ट ऍड्रेससह लोड केले जाणे आवश्यक आहे.

ऑपरेशन: AL ← [port] for byte.
AL ← [port] and AH ← [port+1] for word

उदाहरण: IN AL, 80H ; Fixed Port
MOV DX, 8000H ; Variable port
IN AL, DX

6. OUT port, accumulator

OUT इंस्ट्रक्शन AL मधील बाइट किंवा AX मधील वर्ड दिलेल्या पोर्टवर कॉपी करते. पोर्टचा ऍड्रेस थेट किंवा अप्रत्यक्षपणे इंस्ट्रक्शनमध्ये निर्दिष्ट केला जाऊ शकतो. फिक्स्ड पोर्टसाठी (Fixed Port) पोर्टचा 8-बिट ऍड्रेस थेट इंस्ट्रक्शनमध्ये नमूद केला जातो. व्हेरिएबल पोर्टसाठी (Variable Port) पोर्टचा 16 बिट ऍड्रेस फक्त DX रजिस्टरमध्ये निर्दिष्ट केला जातो. म्हणून DX रजिस्टर मध्ये नेहमी OUT इंस्ट्रक्शनपूर्वी 16-बिट्स पोर्ट ऍड्रेससह लोड केले जाणे आवश्यक आहे.

ऑपरेशन: [port] ← AL for byte.
[port] ← AL and [port+1] ← AH for word

उदाहरण:

```
OUT 80H, AL        ; Fixed Port
MOV DX, 8000H      ; Variable port
OUT DX, AL
```

7. XLAT

XLAT इंस्ट्रक्शन AL रजिस्टरमधील बाइटला मेमरीमधील लुकअप टेबलमधील बाइटसह बदलते. XLAT इंस्ट्रक्शन एक्सिक्युशन करण्यापूर्वी नवीन कोडची मूल्ये लुकअप टेबल मेमरीमध्ये ठेवणे आवश्यक आहे आणि लुकअप टेबलच्या सुरुवातीच्या ऑफसेट ऍड्रेस BX मध्ये लोड करणे आवश्यक आहे. लुकअप टेबलमध्ये इच्छित बाइट दर्शवण्यासाठी XLAT इंस्ट्रक्शन BX मधील टेबलच्या सुरूवातीच्या ऑफसेटमध्ये AL मधील बाइट जोडते आणि [BX + AL] ने पॉईंट केलेल्या ऍड्रेसवरील बाइट परत AL मध्ये कॉपी करते. XLAT कोणतेही फ्लॅग बदलत नाही.

ऑपरेशन: AL ← DS:[BX+AL]

उदाहरण:

```
.DATA
        LTABLE    DB    ‘0123456789ABCDEF’
        NUM       DB    10
.CODE
        .
    MOV BX, offset TABLE    ; BX ला लुकअप टेबलच्या सुरूवातीस पॉईंट करतो
    MOV AL, NUM
    XLAT                    ; लुकअप टेबलमधील कोडमधून AL मध्ये कोड 0AH कॉपी करतो
```

8. LEA 16-bit register, source

ही इंस्ट्रक्शन सोअर्स म्हणून व्हेरिएबल किंवा मेमरी लोकेशनचा ऑफसेट निर्धारित करते आणि दिलेल्या 16-बिट्स रजिस्टरमध्ये हा ऑफसेट लोड करते.

ऑपरेशन: 16 bit register ← effective address or offset address

उदाहरण: LEA BX, STRING ; STRING व्हेरिएबलचा ऑफसेट BX रेजिस्टरमध्ये लोड करतो

9. LDS / LES 16-bit register, memory address of first word

या इंस्ट्रक्शन्स सलग दोन मेमरी लोकेशन्स वरून इंस्ट्रक्शनमध्ये नमूद केलेल्या रजिस्टरमध्ये एक वर्ड कॉपी करतात. त्यानंतर ते पुढील सलग दोन मेमरी लोकेशन्स वरील वर्ड DS रेजिस्टरमध्ये LDS साठी आणि ES रेजिस्टरमध्ये LES साठी कॉपी करते.

ऑपरेशन:

For LDS instruction

16 bit register ← [memory address]
DS ← [memory address + 2]

For LES instruction

16 bit register ← [memory address]
ES ← [memory address + 2]

उदाहरण: LDS BX, [3000H]

मेमरी लोकेशन 3000H चा डेटा BL मध्ये, 3001H चा डेटा ते BH आणि 3002H आणि 3003H चा डेटा DS रजिस्टरमध्ये कॉपी करतो.

10. LAHF

ही इंस्ट्रक्शन 8086 च्या फ्लॅग रजिस्टरचे लोअर बाइट AH रजिस्टरमध्ये स्टोअर करते.

ऑपरेशन: AH ← Lower byte of 8086 flag register

उदाहरण: LAHF

11. SAHF

ही इंस्ट्रक्शन AH रजिस्टरचा डेटा कॉपी करते जी 8086 च्या फ्लॅग रजिस्टरच्या लोवर बाइटमध्ये फ्लॅग सेट किंवा रीसेट करण्यासाठी वापरली जाते.

ऑपरेशन: Lower byte of 8086 flag register ← AH

उदाहरण: SAHF

12. PUSHF

ही इंस्ट्रक्शन फ्लॅग रजिस्टरला स्टॅकवर (Stack) स्टोअर करण्यासाठी वापरली जाते. स्टॅक पॉइंटर (Stack Pointer) दोनने कमी केला जातो आणि फ्लॅग रजिस्टरमधील वर्ड स्टॅक रजिस्टरद्वारे दर्शविलेल्या मेमरी लोकेशन्सवर स्टोअर करतो.

ऑपरेशन: SP ←SP - 2
SP ← Higher byte of 8086 flag register
SP - 1 ← Lower byte of 8086 Flag register

उदाहरण: PUSHF

13. POPF

ही इंस्ट्रक्शन स्टॅकच्या शीर्षस्थानी (Top of Stack) असलेल्या मेमरी लोकेशन्समधून फ्लॅग रजिस्टर लोड करतो आणि स्टॅक पॉइंटर दोनने वाढवला जातो.

ऑपरेशन: SP ← Higher byte of 8086 flag register ← SP
SP + 1 ← Lower byte of 8086 Flag register ← SP + 1

उदाहरण: POPF

3.3.2 अरीथमेटिक इंस्ट्रक्शन ग्रुप (Arithmetic instruction group)

या इंस्ट्रक्शन्स बेरीज (Addition), वजाबाकी (Subtraction), गुणाकार (Multiplication) आणि भागाकार (Division) इत्यादी यांसारख्या अरीथमॅटिक ऑपरेशन्स करतात.

1. ADD / ADC destination, source

ADD इंस्ट्रक्शन सोअर्स आणि डेस्टिनेशन मधील डेटाची बेरीज करते. ADC इंस्ट्रक्शन सोअर्स आणि डेस्टिनेशन मधील डेटाची बेरीज करते आणि बेरजेच्या रिझल्ट मध्ये कॅरी सुद्धा जोडतो. सोअर्स (Source) हा ईमेडिएट डेटा, रजिस्टर किंवा मेमरी लोकेशनवरील डेटा असू शकते. डेस्टिनेशन (Destination) हा रेजिस्टर किंवा मेमरी लोकेशनमधील डेटा असू शकते. सोअर्स आणि डेस्टिनेशनची साईझ सारखी असणे आवश्यक आहे आणि दोन्ही मेमरी लोकेशन असायला नको. डेस्टिनेशन हा ईमेडिएट नंबर नसावा.

प्रभावित होणारे फ्लॅग: OF, CF, PF, AF, SF, ZF

ऑपरेशन: Destination ← source + destination for ADD.
Destination ← source + destination + CF for ADC.

उदाहरण: ADD AL, BL
ADC BX, 4000H

2. SUB / SBB destination, source

SUB इंस्ट्रक्शन मध्ये डेस्टिनेशनमधील डेटामधून सोअर्स मधील डेटा वजा केला जातो आणि रिझल्ट डेस्टिनेशन मध्ये स्टोअर केला जातो. SBB इंस्ट्रक्शन मध्ये डेस्टिनेशनमधील डेटामधून सोअर्स मधील डेटा वजा केला जातो आणि वजा केलेल्या रिझल्टमधून कॅरी सुद्धा वजा केला जातो आणि वजाबाकीचा रिझल्ट डेस्टिनेशन मध्ये स्टोअर केला जातो. सोअर्स (Source) हा ईमेडिएट डेटा, रजिस्टर किंवा मेमरी लोकेशनवरील डेटा असू शकते. डेस्टिनेशन (Destination) हा

रेजिस्टर किंवा मेमरी लोकेशनमधील डेटा असू शकते. सोअर्स आणि डेस्टिनेशनची साईझ सारखी असणे आवश्यक आहे आणि दोन्ही मेमरी लोकेशन असायला नको. डेस्टिनेशन हा ईमेडिएट नंबर नसावा.

प्रभावित होणारे फ्लॅग: OF, CF, PF, AF, SF, ZF

ऑपरेशन: Destination ← source - destination for SUB.

Destination ← source - destination - CF for SBB.

उदाहरण: SUB AL, BL

SBB BX, 4000H

3. INC destination

ही इंस्ट्रक्शन सूचित केलेल्या डेस्टिनेशनमध्ये 1 जोडते. डेस्टिनेशन (Destination) हा रेजिस्टर किंवा मेमरी लोकेशनमधील डेटा असू शकते. डेस्टिनेशन हा ईमेडिएट नंबर नसावा.

प्रभावित होणारे फ्लॅग: OF, CF, PF, AF, SF, ZF

ऑपरेशन: Destination ← Destination + 1

उदाहरण: INC CX

INC [BX]

4. DEC destination

ही इंस्ट्रक्शन सूचित केलेल्या डेस्टिनेशनमधून 1 वजा करते. डेस्टिनेशन (Destination) हा रेजिस्टर किंवा मेमरी लोकेशनमधील डेटा असू शकते. डेस्टिनेशन हा ईमेडिएट नंबर नसावा.

प्रभावित होणारे फ्लॅग: OF, CF, PF, AF, SF, ZF

ऑपरेशन: Destination ← Destination - 1

उदाहरण: DEC CX

5. CMP destination, source

CMP इंस्ट्रक्शन सोअर्स मधील बाइट/शब्द आणि डेस्टिनेशन मधील बाइट/शब्द यांच्यात तुलना करते. तुलना प्रत्यक्षात सोअर्स किंवा डेस्टिनेशन विना-विध्वंसक (Non-Destructive Subtraction) वजाबाकीद्वारे केली जाते, म्हणजेच सोअर्स आणि डेस्टिनेशन मधील नंबर बदलणार नाही, परंतु तुलनाचे परिणाम सूचित करण्यासाठी फ्लॅग सेट केले जातील. सोअर्स (Source) हा ईमेडिएट डेटा, रजिस्टर किंवा मेमरी लोकेशनवरील डेटा असू शकते. डेस्टिनेशन (Destination) हा रेजिस्टर किंवा मेमरी लोकेशनमधील डेटा असू शकते. सोअर्स आणि डेस्टिनेशनची साईझ सारखी असणे आवश्यक आहे आणि दोन्ही मेमरी लोकेशन असायला नको. डेस्टिनेशन हा ईमेडिएट नंबर नसावा.

प्रभावित होणारे फ्लॅग: OF, CF, PF, AF, SF, ZF

ऑपरेशन:

जर डेस्टिनेशन > सोअर्स तर CF = 0, ZF = 0, SF = 0.

जर डेस्टिनेशन < सोअर्स तर CF = 1, ZF = 0, SF = 1.

जर डेस्टिनेशन = सोअर्स तर CF = 0, ZF = 1, SF = 0.

उदाहरण: CMP AL, BL

6. DAA (Decimal adjust accumulator)

DAA इंस्ट्रक्शन दोन पॅक्ड बीसीडी (packed BCD) नंबर्सची बेरीज केल्यानंतर रिझल्टला पॅक्ड बीसीडी (packed BCD) नंबर्समध्ये रूपांतरित करण्यासाठी वापरला जातो. DAA फक्त AL रजिस्टरवर काम करते. म्हणून, ADD/ADC इंस्ट्रक्शननंतर DAA इंस्ट्रक्शन वापरणे आवश्यक आहे. ADD/ADC इंस्ट्रक्शन हेक्साडेसिमल (Hexadecimal) स्वरूपात दोन BCD नंबर्सची बेरीज करते आणि DAA इंस्ट्रक्शन या हेक्साडेसिमल रिझल्टला BCD नंबरमध्ये रूपांतरित करते.

प्रभावित होणारे फ्लॅग: OF is undefined and CF, PF, AF, SF, ZF

ऑपरेशन:

1. जर AL मधील रिझल्टचा लोवर निब्बल (Lower Nibble) 9 पेक्षा मोठा किंवा AF=1 असल्यास, तर AL=AL+06 करतो.
2. जर AL मधील रिझल्टचा हायर निब्बल (Higher Nibble) 9 पेक्षा मोठा किंवा CF=1 असल्यास, तर AL=AL+60 करतो.
3. वरील दोन्ही अटी पूर्ण झाल्यास AL=AL+66 करतो.

उदाहरण:

जर AL = 66 BCD आणि BL = 66 BCD

```
तर ADD AL, BL          0110 0110 = AL = 66 BCD
                  +    0110 0110 = BL = 66 BCD
                       ---------
                       1100 1100 = AL = CC H and CF=0, AF=0
```

आता, बेरजेनंतर वरील उदाहरणामध्ये, कॅरी (CF) आणि ऑक्झिलरी कॅरी (AF) फ्लॅग रिसेट झालेले आहेत, पण लोवर आणि हायर निबल 9 पेक्षा मोठे आहे. तर DAA इंस्ट्रक्शन योग्य BCD रिझल्ट मिळविण्यासाठी AL रेजिस्टरमधील लोवर (Lower) आणि हायर (Higher) निबलमध्ये 6 जोडते म्हणजे 132 BCD मधील 32 AL मध्ये मिळतो आणि CF =1 मध्ये खाली दिल्याप्रमाणे.

DAA इंस्ट्रक्शन एक्सिक्युट झाल्यानंतर, रिझल्ट खालील प्रमाणे मिळतो.

```
     1100 1100 = AL = CC H and AF = 0, Cy =0
  +  0110 0110
     ---------
     0011 0010 = AL = 32 in BCD form and AF=1, CF = 1
```

7. DAS (Decimal adjust after subtraction)

DAS इंस्ट्रक्शन दोन पॅक्ड बीसीडी (packed BCD) नंबर्सची वजाबाकी केल्यानंतर रिझल्टला पॅक्ड बीसीडी (Packed BCD) नंबर्समध्ये रूपांतरित करण्यासाठी वापरला जातो. DAS फक्त AL रजिस्टरवर काम करते. म्हणून, SUB/SBB इंस्ट्रक्शननंतर DAS इंस्ट्रक्शन वापरणे आवश्यक आहे. SUB/SBB इंस्ट्रक्शन हेक्साडेसिमल (Hexadecimal) स्वरूपात दोन BCD नंबर्सची वजाबाकी करते आणि DAS इंस्ट्रक्शन या हेक्साडेसिमल रिझल्टला BCD नंबरमध्ये रूपांतरित करते.

प्रभावित होणारे फ्लॅग: OF, CF, PF, AF, SF, ZF

ऑपरेशन:

1. जर AL मधील रिझल्टचा लोवर निब्बल (Lower nibble) 9 पेक्षा मोठा किंवा AF=1 असल्यास, तर AL=AL-06 करतो.
2. जर AL मधील रिझल्टचा हायर निब्बल (Higher Nibble) 9 पेक्षा मोठा किंवा CF=1 असल्यास, तर AL=AL-60 करतो.
3. वरील दोन्ही अटी पूर्ण झाल्यास AL=AL-66 करतो.

उदाहरण:

जर AL = 55 BCD आणि BL = 49 BCD

```
तर SUB AL, BL        0101 0101 = AL = 55 BCD
                  -  0100 1001 = BL = 49 BCD
                     ------------------------
                     0000 1100 = AL = 0C H and CF=0, AF=1
```

आता, वजाबाकीनंतर वरील उदाहरणामध्ये, ऑक्झिलरी कॅरी (AF) फ्लॅग सेट झालेले आहे. तर DAS इंस्ट्रक्शन योग्य BCD रिझल्ट मिळविण्यासाठी AL रेजिस्टरमधील लोवर (Lower) निबलमध्ये 06 वजा करून योग्य BCD रिझल्ट मिळवते म्हणजेच 06 BCD खाली दिल्याप्रमाणे. DAS इंस्ट्रक्शन एक्सिक्युट झाल्यानंतर, रिझल्ट खालील प्रमाणे मिळतो.

```
     0000 1100 = AL = 0CH, CF=0 आणि AF = 1
  -  0000 0110
     ---------
     0000 0110 = AL = 06 in BCD form
```

8. NEG destination

ही इंस्ट्रक्शन डेस्टिनेशनमधील नंबर बाइट/वर्ड सेकंड कॉम्प्लिमेंट (2nd Compliment) मध्ये रूपांतरित करते आणि डेस्टिनेशनमध्येच रिझल्ट स्टोअर करते जे रजिस्टर किंवा मेमरी लोकेशन असू शकते.

प्रभावित होणारे फ्लॅग: OF, CF, PF, AF, SF, ZF

ऑपरेशन: Destination ← 2nd Complement of destination

उदाहरण: NEG AX

9. MUL source

ही इंस्ट्रक्शन AL रजिस्टरमधील अन्साईन्ड (Unsigned) बाइटला सोअर्स अन्साईन्ड बाईट सोबत गुणाकार करण्यासाठी किंवा AX रजिस्टरमधील अन्साईन्ड वर्डला सोअर्स अन्साईन्ड वर्ड सोबत गुणाकार करण्यासाठी वापरण्यासाठी वापरली जाते. सोअर्स (Source) कोणतेही रजिस्टर किंवा मेमरी लोकशन असणे आवश्यक आहे. जेव्हा AL मधील बाइटसोबत सोअर्स बाइटचा गुणाकार केला जातो, तेव्हा रिझल्ट AX मध्ये स्टोअर केला जातो कारण दोन 8-बिट म्हणजेच बाइट नंबर्सच्या गुणाकाराचा रिझल्ट जास्तीत जास्त 16 बिट असतो. जेव्हा AX मधील वर्डचा सोअर्स वर्ड सोबत गुणाकार केला जातो, तेव्हा रिझल्टचा MSW DX मध्ये स्टोअर केला जातो आणि AX रजिस्टरमध्ये रिझल्टचा LSW स्टोअर केला जातो, कारण दोन 16-बिट नंबर्सच्या गुणाकाराचा रिझल्ट जास्तीत जास्त 32-बिट असतो. जर रिझल्टचा MSB किंवा MSW शून्य असेल, तर CF आणि OF दोन्ही सेट केले जातात.

प्रभावित होणारे फ्लॅग: OF, CF आणि PF, AF, SF, ZF अनडिफाइन्ड असतात

ऑपरेशन: जर सोअर्स बाइट असेल तर AX ← AL * अन्साईन्ड 8 बिट सोअर्स

जर सोअर्स वर्ड असेल तर DX:AX ← AX * अन्साईन्ड 16 बिट सोअर्स

उदाहरण: MUL BL

MUL BX

10. IMUL source

ही इंस्ट्रक्शन AL रजिस्टरमधील साईंड (Signed) बाइटला सोअर्स साईंड बाईट सोबत गुणाकार करण्यासाठी किंवा AX रजिस्टरमधील अन्साईन्ड वर्डला सोअर्स अन्साईन्ड वर्ड सोबत गुणाकार करण्यासाठी वापरण्यासाठी वापरली जाते. सोअर्स (Source) कोणतेही रजिस्टर किंवा मेमरी लोकशन असणे आवश्यक आहे. जेव्हा AL मधील बाइटसोबत सोअर्स बाइटचा गुणाकार केला जातो, तेव्हा रिझल्ट AX मध्ये स्टोअर केला जातो कारण दोन 8-बिट म्हणजेच बाइट नंबर्सच्या गुणाकाराचा रिझल्ट जास्तीत जास्त 16 बिट असतो. जेव्हा AX मधील वर्डचा सोअर्स वर्ड सोबत गुणाकार केला जातो, तेव्हा रिझल्टचा MSW DX मध्ये स्टोअर केला जातो आणि AX रजिस्टरमध्ये रिझल्टचा LSW स्टोअर केला जातो, कारण दोन 16-बिट नंबर्सच्या गुणाकाराचा रिझल्ट जास्तीत जास्त 32-बिट असतो. जर रिझल्टचा MSB किंवा MSW शून्य असेल, तर CF आणि OF दोन्ही सेट केले जातात.

प्रभावित होणारे फ्लॅग: OF, CF आणि PF, AF, SF, ZF अनडिफाइन्ड असतात

ऑपरेशन: जर सोअर्स बाइट असेल तर AX ← AL * साईंड 8 बिट सोअर्स

जर सोअर्स वर्ड असेल तर DX:AX ← AX * साईंड 16 बिट सोअर्स

उदाहरण: IMUL BL

IMUL BX

11. DIV source

ही इंस्ट्रक्शन 16/8 भागाकार दरम्यान अन्साईन्ड वर्डला (Unsigned Word) अन्साईन्ड बाइटने (Unsigned byte) विभाजित करते आणि 32/16 भागाकार दरम्यान अन्साईन्ड डबल वर्ड (Unsigned Double Word) म्हणजेच 32-बिट्सला अन्साईन्ड वर्डने (Unsigned Word) विभाजित करते. 16 /8 विभाजन करताना, वर्ड (डिव्हिडंड) AX रजिस्टरमध्ये असणे आवश्यक आहे आणि सोअर्स बाइट (डिवाईझर) कोणत्याही 8 बिट रजिस्टर किंवा मेमरी लोकेशनवर असू शकते. भागाकार झाल्या नंतर, 8 बिट कोशंट (Quotient) AL रजिस्टरमध्ये उपलब्ध असेल आणि 8 बिट रिमेंडर (Remainder) AH रजिस्टरमध्ये उपलब्ध असेल. बल वर्डला एका वर्डने भागाकार करताना म्हणजेच 32 /16, डबल वर्डचा MSW DX मध्ये असावा आणि डबल वर्डचा LSW AX मध्ये असावा. सोअर्स वर्ड (डिवाईझर) कोणत्याही 16

बिट रजिस्टर किंवा मेमरी लोकेशनवर असू शकते. भागाकार झाल्या नंतर, 16 बिट कोशंट (Quotient) AX रजिस्टरमध्ये उपलब्ध असेल आणि 16 बिट रिमेंडर (Remainder) DX रजिस्टरमध्ये उपलब्ध असेल. जर वर्ड किंवा डबल वर्डला 0 ने भागले किंवा भागाकार नंतर AL मधील (16/8 साठी) कोशंट FFH पेक्षा किंवा AX मधील (32/16 साठी) कोशंट FFFFH पेक्षा मोठा असला तर प्रोसेसर टाईप 0 इंटरप्ट जनरेट करतो ज्याला डिव्हाइड बाय 0 किंवा डिव्हाइड ओव्हरफ्लोव इंटरप्ट म्हणतात.

प्रभावित होणारे फ्लॅग: OF, CF, PF, AF, SF, ZF सगळे अनडिफाइन्ड असतात

ऑपरेशन:

जर सोअर्स बाइट असेल तर AL / अन्साईन्ड 8 बिट सोअर्स
AL = 8 बिट कोशंट (Quotient)
AH = 8 बिट रिमेंडर (Remainder)
जर सोअर्स वर्ड असेल तर DX:AX / अन्साईन्ड 16 बिट सोअर्स
AX = 16 बिट कोशंट (Quotient)
DX = 16 बिट रिमेंडर (Remainder)

उदाहरण: DIV BL
DIV BX

12. IDIV source

ही इंस्ट्रक्शन 16/8 भागाकार दरम्यान साईंड वर्डला (Signed Word) साईंड बाइटने (Signed Byte) विभाजित करते आणि 32/16 भागाकार दरम्यान साईंड डबल वर्ड (Signed Double Word) म्हणजेच 32-बिट्सला साईंड वर्डने (Signed word) विभाजित करते. 16 /8 विभाजन करताना, वर्ड (डिव्हिडंड) AX रजिस्टरमध्ये असणे आवश्यक आहे आणि सोअर्स बाइट (डिवाईझर) कोणत्याही 8 बिट रजिस्टर किंवा मेमरी लोकेशनवर असू शकते. भागाकार झाल्या नंतर, 8 बिट कोशंट (Quotient) AL रजिस्टरमध्ये उपलब्ध असेल आणि 8 बिट रिमेंडर (Remainder) AH रजिस्टरमध्ये उपलब्ध असेल. बल वर्डला एका वर्डने भागाकार करताना म्हणजेच 32 /16, डबल वर्डचा MSW DX मध्ये असावा आणि डबल वर्डचा LSW AX मध्ये असावा. सोअर्स वर्ड (डिवाईझर) कोणत्याही 16 बिट रजिस्टर किंवा मेमरी लोकेशनवर असू शकते. भागाकार झाल्या नंतर, 16 बिट कोशंट (Quotient) AX रजिस्टरमध्ये उपलब्ध असेल आणि 16 बिट रिमेंडर (Remainder) DX रजिस्टरमध्ये उपलब्ध असेल. जर वर्ड किंवा डबल वर्डला 0 ने भागले किंवा भागाकार नंतर AL मधील (16/8 साठी) कोशंट FFH पेक्षा किंवा AX मधील (32/16 साठी) कोशंट FFFFH पेक्षा मोठा असला तर प्रोसेसर टाईप 0 इंटरप्ट जनरेट करतो ज्याला डिव्हाइड बाय 0 किंवा डिव्हाइड ओव्हरफ्लोव इंटरप्ट म्हणतात.

प्रभावित होणारे फ्लॅग: OF, CF, PF, AF, SF, ZF सगळे अनडिफाइन्ड असतात

ऑपरेशन:

जर सोअर्स बाइट असेल तर AL / साईंड 8 बिट सोअर्स
AL = 8 बिट कोशंट (Quotient)
AH = 8 बिट रिमेंडर (Remainder)
जर सोअर्स वर्ड असेल तर DX:AX / साईंड 16 बिट सोअर्स
AX = 16 बिट कोशंट (Quotient)
DX = 16 बिट रिमेंडर (Remainder)

उदाहरण: IDIV BL
IDIV BX

13. CBW (Convert Byte to Word)

ही इंस्ट्रक्शन AL मधील बाइटचे साइन बिट (Sign bit) i.e. D_7 AH मधील सर्व बिट्सवर कॉपी करते. AH हे रेजिस्टर AL चे साईंड विस्तार असल्याचे म्हटले जाते. CBW ऑपरेशन AL मध्ये साईन्ड बाइटचे IDIV इंस्ट्रक्शनसह दुसऱ्या साईन्ड बाइटद्वारे विभाजन करण्यापूर्वी केले जाणे आवश्यक आहे.

ऑपरेशन: AH ← filled with 8^{th} bit of AL i.e. D_7

उदाहरण: जर AH = 00000000 आणि AL = 10011011

CBW ऑपरेशन नंतर

AH = 11111111 आणि AL = 10011011

14. CWD (Convert Word to Double word)

ही इंस्ट्रक्शन AX मधील वर्डचे साइन बिट (Sign bit) i.e. D_{15}, DX मधील सर्व बिट्सवर कॉपी करते. DX हे रेजिस्टर AX चे साईंड विस्तार असल्याचे म्हटले जाते. CWD ऑपरेशन AX मध्ये साईन्ड वर्डचे IDIV इंस्ट्रक्शनसह दुसऱ्या साईन्ड वर्डद्वारे विभाजन करण्यापूर्वी केले जाणे आवश्यक आहे.

ऑपरेशन: DX ← filled with 16^{th} bit of AX i.e. D_{15}

उदाहरण: जर DX = 0000000000000000 आणि AX = 1001101100000000

CWD ऑपरेशन नंतर

DX = 1111111111111111 आणि AX = 1001101100000000

15. AAA [ASCII adjust after addition]

ही इंस्ट्रक्शन AL रजिस्टरमधील डेटा अनपॅक्ड BCD रिझल्टमध्ये रूपांतरित करण्यासाठी वापरली जाऊ शकते. AL रजिस्टरचा हायर निबल शून्याने भरलेला जातो. ही इंस्ट्रक्शन ADD इंस्ट्रक्शननंतर वापरली जाते आणि रिझल्ट AL रजिस्टरमध्ये स्टोअर केला जातो. AAA इंस्ट्रक्शन एक्सिक्युट करण्यापूर्वी, AH 0 ने लोड केला पाहिजे. AAA इंस्ट्रक्शनचे कार्य खालीलप्रमाणे आहे.

प्रभावित होणारे फ्लॅग: CF, AF

ऑपरेशन:

AL चा हायर निबल क्लीअर करते म्हणजेच AL= AL AND 0FH

जर AL चा लोवर निबल > 9 किंवा AF = 1 असल्यास.

(a) AL = AL + 6. (b) AH = AH + 1.

(c) AF = CF = 1. (d) AL = AL AND 0FH.

उदाहरण: समजा AH = 00H आणि AL = 0AH आहे तर AAA एक्सिक्युट झाल्यानंतर AH = 01H आणि AL = 00H मिळेल.

16. AAS [ASCII adjust after subtraction]

ही इंस्ट्रक्शन AL रजिस्टरमधील डेटा BCD अनपॅक्ड रिझल्टमध्ये रूपांतरित करण्यासाठी वापरली जाऊ शकते. AL रजिस्टरचा हायर निबल शून्याने भरलेला जातो. ही इंस्ट्रक्शन SUB इंस्ट्रक्शननंतर वापरली जाते आणि रिझल्ट AL रजिस्टरमध्ये स्टोअर केला जातो. AAA इंस्ट्रक्शनचे कार्य खालीलप्रमाणे आहे.

प्रभावित होणारे फ्लॅग: CF, AF

ऑपरेशन:

AL चा हायर निबल क्लीअर करते म्हणजेच AL= AL AND 0FH

जर ALचा लोवर निबल > 9 किंवा AF = 1 असल्यास.

(a) AL = AL - 6. (b) AH = AH - 1.

(c) AF = CF = 1. (d) AL = AL AND 0FH.

उदाहरण:

समजा AH = 02H आणि AL = 0BH आहे तर AAS एक्सिक्युट झाल्यानंतर

AH = 01H आणि AL = 05H मिळेल

17. AAM [ASCII adjust after multiplication]

ही इंस्ट्रक्शन दोन वैध अनपॅक केलेल्या BCD नंबर्स च्या गुणाकाराचे अनपॅक्ड BCD रिझल्टमध्ये रूपांतरित करण्यासाठी वापरली जाते. ही इंस्ट्रक्शन MUL इंस्ट्रक्शननंतर वापरायला पाहिजे. अनपॅक्ड केलेल्या BCD नंबरवर MUL चे ऑपरेशन नेहमी 100 पेक्षा कमी असते; त्यामुळे, रिझल्ट AL रजिस्टरमध्ये असेल. AL रजिस्टरमधील बायनरी नंबरला

10 ने भागले जाते आणि कोशंट (Quotient) AH रजिस्टरमध्ये स्टोअर केला जातो आणि रिमेंडर (Remender) AL रजिस्टरमध्ये स्टोअर केला जातो.

प्रभावित होणारे फ्लॅग: PF, SF, ZF

ऑपरेशन:

(a) AL = AL MOD 10.

(b) AH = AL / 10 [केवळ इंटीजर भाग मानला जातो].

उदाहरण:

समजा AL = 08 आणि BL = 08 असेल तर

MUL BL AX = 40H (64 in decimal).

AAM AH = 06 and AL = 04.

18. AAD [ASCII adjust before division]

या इंस्ट्रक्शन AAD चा वापर AH आणि AL रजिस्टरमधील अनपॅक्ड (Unpacked) केलेला BCD अंक AL रजिस्टरमधील समतुल्य बायनरी (Binary) नंबरमध्ये रूपांतरित करण्यासाठी केला जाऊ शकतो. हि इंस्ट्रक्शन DIV इंस्ट्रक्शनचा आधी वापरायला पाहिजे. DIV इंस्ट्रक्शन AL रजिस्टरमध्ये कोशंट (Quotient) आणि रिमेंडर (Remainder) AH रजिस्टरमध्ये स्टोअर करतो. AH आणि AL चे हायर निबल शून्याने भरलेले जातात.

प्रभावित होणारे फ्लॅग: PF, SF, ZF

ऑपरेशन: (a) AL = AH * 10 + AL

(b) AH = 00

उदाहरण:

समजा AX = 19H (25 in decimal) आणि BL = 07 H (7 in decimal) असेल तर

AAD

DIV BL ; AL = 03 Quotient and AH = 04 Remainder

3.3.3 लॉजिकल इंस्ट्रक्शन ग्रुप (Logical instruction group)

1. AND destination, source

ही इंस्ट्रक्शन सोअर्स आणि डेस्टिनेशन ऑपरेंड सोबत बिट बाय बिट लॉजिकल अँड (AND) ऑपेरेशन करते आणि रिजल्ट निर्दिष्ट केलेल्या डेस्टिनेशनमध्ये स्टोअर केला जातो. प्रत्येक बिट पोझिशनचा रिझल्ट दोन इनपुट अँड (AND) गेटचा ट्रुथ टेबल अनुसरण करतो. सोअर्स हे रेजिस्टर/मेमरी लोकेशन/इमिडिएट डेटा ह्या पैकी एक असू शकतो आणि डेस्टिनिशन हे दुसरे रजिस्टर/मेमरी लोकेशन असू शकते. सोअर्स आणि डेस्टिनिशनची साईझ समान असणे आवश्यक आहे. सोअर्स आणि डेस्टिनिशन दोन्ही मेमरी लोकेशन्स असायला नको. डेस्टिनिशन हा इमिडिएट डेटा असायला नको.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : CF = 0, OF = 0, PF, SF, ZF.

ऑपरेशन: destination ← destination AND source

उदाहरण: AND AL, CL

AND BX, 0FF00H

2. OR destination, source

ही इंस्ट्रक्शन सोअर्स आणि डेस्टिनेशन ऑपरेंड सोबत बिट बाय बिट लॉजिकल ऑर (OR) ऑपेरेशन करते आणि रिजल्ट निर्दिष्ट केलेल्या डेस्टिनेशनमध्ये स्टोअर केला जातो. सोअर्स आणि डेस्टिनेशन हे बाईट (Byte) किंवा वर्ड (Word) असू शकते. प्रत्येक बिट पोझिशनचा रिझल्ट दोन इनपुट ऑर (OR) गेटचा ट्रुथ टेबल अनुसरण करतो. सोअर्स हे रेजिस्टर/मेमरी लोकेशन/इमिडिएट डेटा ह्या पैकी एक असू शकतो आणि डेस्टिनिशन हे दुसरे रजिस्टर/मेमरी लोकेशन असू शकते. सोअर्स आणि डेस्टिनिशनची साईझ समान असणे आवश्यक आहे. सोअर्स आणि डेस्टिनिशन दोन्ही मेमरी लोकेशन्स असायला नको. डेस्टिनिशन हा इमिडिएट डेटा असायला नको.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : CF = 0, OF = 0, PF, SF, ZF.

ऑपरेशन: destination ← destination OR source

उदाहरण: OR AL,CL
OR AX, 0FF00H

3. XOR destination, source

ही इंस्ट्रक्शन सोअर्स आणि डेस्टिनेशन ऑपरेंड सोबत बिट बाय बिट लॉजिकल एक्स-ऑर (XOR) ऑपेरेशन करते आणि रिजल्ट निर्दिष्ट केलेल्या डेस्टिनेशनमध्ये स्टोअर केला जातो. सोअर्स आणि डेस्टिनेशन हे बाईट (Byte) किंवा वर्ड (Word) असू शकते. प्रत्येक बिट पोझिशनचा रिझल्ट दोन इनपुट एक्स-ऑर (XOR) गेटचा ट्रुथ टेबल अनुसरण करतो. सोअर्स हे रेजिस्टर/मेमरी लोकेशन/इमिडिएट डेटा ह्या पैकी एक असू शकतो आणि डेस्टिनिशन हे दुसरे रजिस्टर/मेमरी लोकेशन असू शकते. सोअर्स आणि डेस्टिनिशनची साईझ समान असणे आवश्यक आहे. सोअर्स आणि डेस्टिनिशन दोन्ही मेमरी लोकेशन्स असायला नको. डेस्टिनिशन हा इमिडिएट डेटा असायला नको.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : CF = 0, OF = 0, PF, SF, ZF.

ऑपरेशन: destination ← destination XOR source

उदाहरण: XOR AH, AL
XOR BX, 00FFH

4. NOT destination

ही इंस्ट्रक्शन निर्दिष्ट डेस्टिनेशनमधील बाइट किंवा वर्डच्या प्रत्येक बिटला उलट करते, म्हणजेच 1's कॉम्प्लिमेंट (One's Complement). डेस्टिनिशन हे रजिस्टर किंवा मेमरी लोकेशन असू शकते.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : NON

ऑपरेशन: destination ← NOT destination

उदाहरण: NOT BX

5. TEST destination, source

ही इंस्ट्रक्शन सोअर्स आणि डेस्टिनेशन ऑपरेंड सोबत बिट बाय बिट लॉजिकल नॉन डिस्ट्रक्टिव्ह अँड (Non Destructive AND) ऑपेरेशन करते आणि फक्त फ्लॅग अद्ययावत केले जातात, परंतु कोणतेही ऑपरेंड बदललेले जात नाहीत. प्रत्येक बिट पोझिशनचा रिझल्ट दोन इनपुट अँड (AND) गेटचा ट्रुथ टेबल अनुसरण करतो. सोअर्स हे रेजिस्टर/मेमरी लोकेशन/इमिडिएट डेटा ह्या पैकी एक असू शकतो आणि डेस्टिनिशन हे दुसरे रजिस्टर/मेमरी लोकेशन असू शकते. सोअर्स आणि डेस्टिनिशनची साईझ समान असणे आवश्यक आहे. सोअर्स आणि डेस्टिनिशन दोन्ही मेमरी लोकेशन्स असायला नको. डेस्टिनिशन हा इमिडिएट डेटा असायला नको.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : CF, OF, PF, SF, ZF.

ऑपरेशन: Flags ← set for result of (destination AND source).

उदाहरण: TEST BH, CL
TEST BX, 0FFFH

3.3.4 प्रोग्राम कंट्रोल ट्रान्सफर किंवा ब्रांचिंग इंस्ट्रक्शन्स ग्रुप (Program control transfer or branching instructions group)

1. CALL procedure

कॉल (CALL) इंस्ट्रक्शनचा वापर प्रोग्रॅम कंट्रोल सब-प्रोग्राम (Sub-program) किंवा सब रूटीनमध्ये (Sub Routine) हस्तांतरित करण्यासाठी केला जातो. निअर (Near) आणि फार (Far) हे कॉलचे दोन मूलभूत प्रकार आहेत. निअर कॉल (NEAR CALL) म्हणजे एक प्रोसिजरसाठी कॉल आहे, जो कॉल इंस्ट्रक्शन प्रमाणेच त्याच कोड सेगमेंटमध्ये असतो. फार कॉल (FAR CALL) म्हणजे एका प्रोसिजरसाठी कॉल आहे, जो दुसऱ्या कोड सेगमेंटमध्ये असतो जिथे कॉल इंस्ट्रक्शन नसते. निअर कॉल साठी IP ची व्हॅलू स्टॅक वर स्टोअर केली जाते आणि फार कॉल साठी CS आणि IP ची व्हॅलू स्टॅक वर स्टोअर केली जाते.

ऑपरेशन:

1. जर निअर कॉल (NEAR CALL) असेल तर
SP ← SP – 2
Save IP on stack

IP ← address of procedure

2. जर फार कॉल (FAR CALL) असेल तर

SP ← SP — 2

SP ← CS i.e. Save CS on stack

CS ← New segment base address of the called procedure

SP ← SP – 2

SP ← IP i.e. Save IP on stack

IP ← New offset address of the called procedure

उदाहरण: CALL FACT

CALL FAR PTR FIBON

2. RET instruction

कॉलिंग प्रोग्राममधील CALL इंस्ट्रक्शननंतर लगेचच प्रोग्रॅम एक्झिक्यूशन कंट्रोलला प्रोग्रॅममधील पुढील इंस्ट्रक्शनवर परत करण्यासाठी RET इंस्ट्रक्शनचा वापर केला जातो. जर प्रोसिजर हि नियर प्रोसिजर (NEAR Procedure) असेल, तर स्टॅक टॉपवरून एक वर्ड IP ची व्हॅलू पुनर्संचयित करून प्रोग्रॅम कंट्रोल कॉलिंग प्रोग्रॅम मध्ये परत जाईल. कॉलिंग प्रोग्राममधील CALL इंस्ट्रक्शन नंतर स्टॅक वरचा वर्ड पुढील इंस्ट्रक्शनचा ऑफसेट ॲड्रेस आहे जो CALL इंस्ट्रक्शन द्वारे स्टॅकवर केलेला असतो. जर प्रोसिजर हि फार प्रोसिजर (FAR Procedure) असेल, तर स्टॅक टॉपवरून दोन वर्ड IP व CS ची व्हॅलू पुनर्संचयित करून प्रोग्रॅम कंट्रोल कॉलिंग प्रोग्रॅम मध्ये परत जाईल.

ऑपरेशन:

1. जर नियर रिटर्न (NEAR RETURN) असेल तर

IP ← content of top of stack.

SP ← SP + 2.

2. जर फार रिटर्न (FAR RETURN) असेल तर

IP ← contents of top of stack.

SP ← SP + 2.

CS ← contents of top of stack.

SP ← SP + 2.

3. INT N [N = type of interrupt]

या इंस्ट्रक्शनमुळे 8086 त्याच्या INTR किंवा NMI इनपुटवरील इंटरप्ट (Interrupt) सिग्नलला ज्या पद्धतीने प्रतिसाद देते त्याच पद्धतीने 8086 ला फार प्रोसिजर कॉल करण्यास प्रवृत्त करते. इंस्ट्रक्शनमधील दिलेले टर्म टाईप (N) हि 0 आणि 255 मधली संख्या दर्शवते जी इंटरप्ट ओळखते. जेव्हा 8086 INT इंस्ट्रक्शन एक्सिक्युट होते, तेव्हा ते पुढील ऑपरेशन करते.

1. स्टॅक पॉइंटर दोनने कमी करतो आणि स्टॅकवर फ्लॅग रेजिस्टरची व्हॅल्यू पुश करतो.
2. स्टॅक पॉइंटर पुन्हा दोनने कमी करतो आणि स्टॅकवर CS रेजिस्टरची व्हॅल्यू पुश करतो.
3. स्टॅक पॉइंटर पुन्हा दोनने कमी करतो आणि स्टॅकवर IP रेजिस्टरची व्हॅल्यू पुश करतो.
4. इंस्ट्रक्शनमध्ये नमूद केलेल्या प्रकाराच्या (Type N) 4 पट ॲबसॉल्यूट (Absolute) मेमरी ॲड्रेस वरून IP मध्ये नवीन व्हॅलू लोड करते.
5. इंस्ट्रक्शनमध्ये नमूद केलेल्या प्रकाराच्या (Type N) 4 पट ॲबसॉल्यूट (Absolute) मेमरी ॲड्रेस वरून CS मध्ये नवीन व्हॅलू लोड करते.
6. IF आणि TF दोन्ही फ्लॅग रीसेट करते, इतर फ्लॅगवर या इंस्ट्रक्शनचा परिणाम होणार नाही.

4. INTO instruction [Interrupt on overflow]

जर ओव्हरफ्लो फ्लॅग OF सेट केला असेल, तर ही इंस्ट्रक्शन प्रकार 4 इंटरप्ट निर्माण करेल आणि 8086 ला अप्रत्यक्ष फार कॉल प्रोसिजर करण्यास प्रवृत्त करेल जी युझर ओव्हरफ्लो स्थिती हाताळण्यासाठी लिहिलेली आहे. जेव्हा 8086 INTO इंस्ट्रक्शन एक्सिक्युट करते, तेव्हा ते पुढील ऑपरेशन करेल.

1. स्टॅक पॉइंटर दोनने कमी करतो आणि स्टॅकवर फ्लॅग रेजिस्टरची व्हॅल्यू पुश करतो.
2. स्टॅक पॉइंटर पुन्हा दोनने कमी करतो आणि स्टॅकवर CS रेजिस्टरची व्हॅल्यू पुश करतो.
3. स्टॅक पॉइंटर पुन्हा दोनने कमी करतो आणि स्टॅकवर IP रेजिस्टरची व्हॅल्यू पुश करतो.
4. इंस्ट्रक्शनमध्ये नमूद केलेल्या प्रकाराच्या (Type 4) 4 पट ऍबसॉल्यूट (Absolute) मेमरी ऍड्रेस 00010H वरून IP मध्ये नवीन व्हॅलू लोड करते.
5. इंस्ट्रक्शनमध्ये नमूद केलेल्या प्रकाराच्या (Type 4) 4 पट ऍबसॉल्यूट (Absolute) मेमरी ऍड्रेस 00012H वरून CS मध्ये नवीन व्हॅलू लोड करते.
6. IF आणि TF दोन्ही फ्लॅग रीसेट करते, इतर फ्लॅगवर या इंस्ट्रक्शनचा परिणाम होणार नाही.

5. IRET instruction

IRET इंस्ट्रक्शन इंटरप्ट सर्व्हिस रुटीन (Interrupt Service Routine) च्या शेवटी इंटरप्ट आणलेल्या प्रोग्रामवर एक्सिक्युशन परत करण्यासाठी वापरली जाते.

6. JMP destination address [Jump unconditional]

इंस्ट्रक्शन बिनशर्त (Unconditional) एक्सिक्युशनचे नियंत्रण 16-बिट डिस्प्लेसमेंट किंवा CS: IP वापरून निर्दिष्ट ऍड्रेसवर हस्तांतरित करते. JMP चे टार्गेट त्याच कोड सेगमेंटमध्ये असल्यास, टार्गेट लोकशन वर नियंत्रण हस्तांतरित करण्यासाठी केवळ IP बदलणे आवश्यक आहे आणि याला नियर (NEAR) JMP म्हणजेच इंट्रा सेगमेंट (Intra Segment) JMP म्हणतात. JMP चे टार्गेट दुसऱ्या कोड सेगमेंटमध्ये असल्यास, टार्गेट लोकशन वर नियंत्रण हस्तांतरित करण्यासाठी CS आणि IP बदलणे आवश्यक आहे आणि याला फार (FAR) JMP म्हणजेच इंटर सेगमेंट (Inter Segment) JMP म्हणतात.

उदाहरण: JMP UP
JMP FAT PRT NEXT

7. Conditional jump instructions

सशर्त JMP (Conditional JMP) इंस्ट्रक्शन काही निर्दिष्ट शर्त पूर्ण झाल्यास प्रोग्रॅम कंट्रोल टार्गेट लोकशनवर हस्तांतरित करते. सशर्त JMP सूचना सामान्यतः कंपेअर (CMP) इंस्ट्रक्शन किंवा अर्थमेटिक (Arithmetic) किंवा लॉजिकल (Logical) इंस्ट्रक्शन नंतर वापरल्या जातात. विविध प्रकारच्या सशर्त JMP इंस्ट्रक्शन टेबल 3.4 मध्ये दिल्या आहेत.

Table 3.4

Opertion	Description	Jump Condition
	Common Operations	
JC	Carry	**CF = 1**
JNC	Not Carry	CF = 0
JE/JZ	Equal or Zero	**ZF = 1**
JNE/JNZ	Not Equal or Not Zero	ZF = 0
JP/JPE	Parity or Parity Even	**PF = 1**
JNP/JPO	Not Parity or Parity Odd	PF = 0
	Signed Operations	roboticelectronics.in
JO	Overflow	**OF = 1**
JNO	Not Overflow	OF = 0
JS	Sign	**SF = 1**
JNS	Not Sign	SF = 0
JL/JNGE	Less	**(SF Ex-Or OF) = 1**
JGE/JNL	Greater or Equal	(SF Ex-Or OF) = 0
JLE/JNG	Less or Equal	**((SF Ex-Or OF) + ZF) = 1**
JG/JNLE	Greater	((SF Ex-Or OF) + ZF) = 0
	Unsigned Operations	
JB/JNAE	Below	**CF = 1**
JAE/JNB	Above or Equal	CF = 0
JBE/JNA	Below or Equal	**(CF Ex-Or ZF) = 1**
JA/JNBE	Above	(CF Ex-Or ZF) = 0

3.3.5 मशीन किंवा प्रोसेस कंट्रोल इंस्ट्रक्शन ग्रुप (Machine or process control instruction group)

1. HLT: Halt

HLT इंस्ट्रक्शन प्रोसेसरला हाल्ट स्थितीत प्रवेश करण्यास प्रवृत्त करते. CPU इंस्ट्रक्शन फेच करणे आणि एक्सिक्युशन करणे थांबवते. खालीलपैकी कोणतीही एक घटना घडल्यास CPU ला हाल्ट स्थितीतून बाहेर आणले जाऊ शकते.

1. INTR पिनवर इंटरप्ट सिग्नल आल्यास
2. NMI पिनवर इंटरप्ट सिग्नल आल्यास
3. RESET पिनवर रिसेट सिग्नल आल्यास

2. NOP: No Operation

ही इंस्ट्रक्शन तीन क्लॉक (Clock) सायकलची प्रतीक्षा स्थिती (Wait State) जोडण्यासाठी वापरली जाते आणि या तीन क्लॉक सायकल दरम्यान CPU कोणतेही ऑपरेशन करत नाही.

3. WAIT

इंस्ट्रक्शन WAIT प्रोसेसरला आयडल स्थितीत (Ideal State) किंवा प्रतीक्षा स्टेट (Wait State) मध्ये प्रवेश करण्यास प्रवृत्त करते आणि पुढीलपैकी एक सिग्नल येईपर्यंत प्रोसेसर प्राप्त स्थितीत राहते.

1. INTR पिनवर इंटरप्ट सिग्नल आल्यास
2. NMI पिनवर इंटरप्ट सिग्नल आल्यास
3. TEST पिनवर सिग्नल आल्यास

4. LOCK

ही इंस्ट्रक्शन इतर प्रोसेसर किंवा DMA कंट्रोलरला सामायिक संसाधनांवर (Shared resource) जसे कि ऍड्रेस बस, डेटा बस आणि कन्ट्रोल बस वर नियंत्रण ठेवण्यास प्रतिबंधित करते. ही इंस्ट्रक्शन 8086 प्रोसेसरवर असलेल्या LOCK पिन वर सिग्नल देते.

3.3.6 फ्लॅग म्यानिप्युलेशन इंस्ट्रक्शन ग्रुप (Flag manipulation instruction group)

1. CLC [Clear Carry] : ही इंस्ट्रक्शन कॅरी फ्लॅग क्लिअर करते. (CF← 0)
2. CMC [Complement Carry]: ही इंस्ट्रक्शन कॅरी फ्लॅगचा कॉम्प्लिमेंट करते. (CF←˜CF)
3. STC [Set Carry]: ही इंस्ट्रक्शन कॅरी फ्लॅग सेट करते. (CF ← 1)
4. CLD [Clear Direction flag]: ही इंस्ट्रक्शन डायरेक्शन फ्लॅग क्लिअर करते. (DF←0)
5. STD [Set Direction flag]: ही इंस्ट्रक्शन डायरेक्शन फ्लॅग सेट करते. (DF←1)
6. CLI [Clear Interrupt flag]: ही इंस्ट्रक्शन इंटरप्ट क्लिअर सेट करते. (IF←0)
7. STI [Set Interrupt flag]: ही इंस्ट्रक्शन इंटरप्ट फ्लॅग सेट करते. (IF←1)

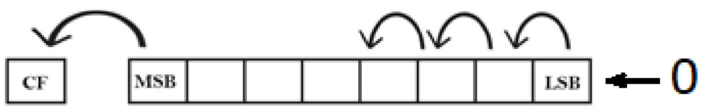
3.3.7 शिफ्ट आणि रोटेट इंस्ट्रक्शन ग्रुप (Shift and rotate instruction group)

1. SHL / SAL destination, count

SHL आणि SAL एकाच ऑपरेशनसाठी दोन इंस्ट्रक्शन्स आहेत. ह्या इंस्ट्रक्शन्स निर्दिष्ट डेस्टिनेशनमधील प्रत्येक बिट दिलेल्या काउंट (Count) वेळा डावीकडे शिफ्ट करते. जसजसे LSB पोझिशनमधून बिट डावीकडे शिफ्ट केले जातात, तसतसे LSB पोझिशनमध्ये 0 घातला जातो आणि MSB CF मध्ये शिफ्ट केला जातो. एका पेक्षा अधिक शिफ्टच्या बाबतीत, CF मध्ये MSB मधून सर्वात अलीकडे शिफ्ट केलेले बिट असतील आणि पूर्वी CF मध्ये शिफ्ट केलेले बिट गमावले जातील. डेस्टिनेशन ऑपरेंड हे बाइट किंवा वर्ड रजिस्टरमध्ये किंवा मेमरी लोकेशनमध्ये असू शकते. जर शिफ्ट्सचा काउन्ट एक असेल, तर हे इंस्ट्रक्शनच्या काउन्टच्या जागेवर 1 लिहिले जाऊ शकते. जर शिफ्ट्सचा काउन्ट एक पेक्षा जास्त असेल, तर काउन्टची व्हॅलू CL रेजिस्टरमध्ये असायला पाहिजे.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : CF, OF, PF, SF, ZF आणि AF अनडिफाइन्ड असतो

ऑपरेशन:



उदाहरण:

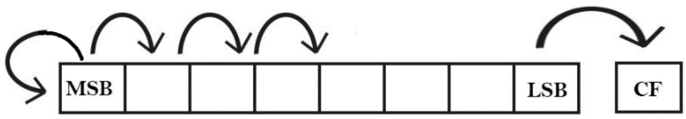
SHL BX, 1
SAL AL, CL

2. SAR destination, count

ही इंस्ट्रक्शन निर्दिष्ट डेस्टिनेशनमधील प्रत्येक बिट काउन्टमध्ये दिलेल्या नंबर वेळा उजवीकडे शिफ्ट करते. बिट MSB पोझिशनमधून बाहेर पडल्यावर, जुन्या MSB बिटची एक कॉपी MSB पोझिशनमध्ये ठेवली जाते म्हणजेच साइन बिट MSB मध्ये कॉपी केली जाते आणि LSB बिट CF मध्ये शिफ्ट केले जाते. एका पेक्षा अधिक शिफ्टच्या बाबतीत, CF मध्ये MSB मधून सर्वात अलीकडे शिफ्ट केलेले बिट असतील आणि पूर्वी CF मध्ये शिफ्ट केलेले बिट गमावले जातील. डेस्टिनेशन ऑपरेंड हे बाइट किंवा वर्ड रजिस्टरमध्ये किंवा मेमरी लोकेशनमध्ये असू शकते. जर शिफ्ट्सचा काउन्ट एक असेल, तर हे इंस्ट्रक्शनच्या काउन्टच्या जागेवर 1 लिहिले जाऊ शकते. जर शिफ्ट्सचा काउन्ट एक पेक्षा जास्त असेल, तर काउन्टची व्हॅलू CL रेजिस्टरमध्ये असायला पाहिजे.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : CF, OF, PF, SF, ZF आणि AF अनडिफाइन्ड असतो

ऑपरेशन:



उदाहरण:

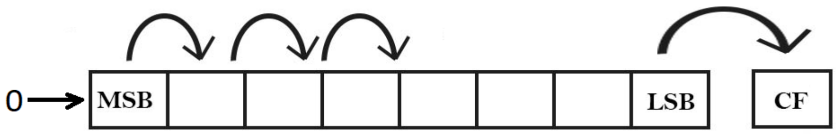
SAR BX, 1
SAR AL, CL

3. SHR destination, count

ही इंस्ट्रक्शन निर्दिष्ट डेस्टिनेशनमधील बाईट किंवा वर्ड चा प्रत्येक बिट काउन्ट वेळा उजवीकडे शिफ्ट करते. MSB पोझिशनच्या बिट उजवीकडे शिफ्ट केल्यावर, त्याच्या जागी 0 घातला जातो आणि LSB बिट CF मध्ये जाते. जर शिफ्ट्सचा काउन्ट एक असेल, तर हे इंस्ट्रक्शनच्या काउन्टच्या जागेवर 1 लिहिले जाऊ शकते. जर शिफ्ट्सचा काउन्ट एक पेक्षा जास्त असेल, तर काउन्टची व्हॅलू CL रेजिस्टरमध्ये असायला पाहिजे. पूर्वी CF मध्ये शिफ्ट केलेले बिट्स गमावले जातील. डेस्टिनेशन ऑपरेंड हे बाइट किंवा वर्ड रजिस्टरमध्ये किंवा मेमरी लोकेशनमध्ये असू शकते.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : CF, OF, PF, SF, ZF आणि AF अनडिफाइन्ड असतो

ऑपरेशन:



उदाहरण:

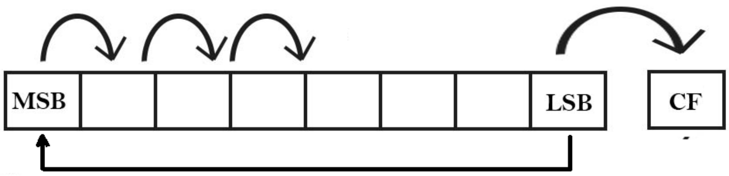
SHR BX, 1
SHR AL, CL

4. ROR destination, count [Rotate right without carry]

ही इंस्ट्रक्शन निर्दिष्ट बाइट किंवा वर्डचे सर्व बिट्स उजवीकडे कॉउंटमध्ये दिल्या संख्ये वेळा फिरवते. LSB मधून फिरवलेला बिट MSB मध्ये जातो आणि CF मध्ये कॉपी देखील केला जातो. एका पेक्षा अधिक वेळा बिट फिरवल्यानंतर CF मध्ये LSB मधून अलीकडे फिरवलेल्या बिटची कॉपी असेल. डेस्टिनेशन ऑपरेंड हे बाइट किंवा वर्ड रजिस्टरमध्ये किंवा मेमरी लोकेशनमध्ये असू शकते. रोटेशनची इच्छित संख्या एक असल्यास, इंस्ट्रक्शनमध्ये काउन्टचा जागी 1 लिहून हे करता येते. परंतु, एका पेक्षा अधिक रोटेशनसाठी, इच्छित रोटेशनची संख्या CL रजिस्टरमध्ये लोड केली जाते आणि CL रेजिस्टर काउन्टच्या ठिकाणी ठेवला जाते.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : CF, OF,

ऑपरेशन:



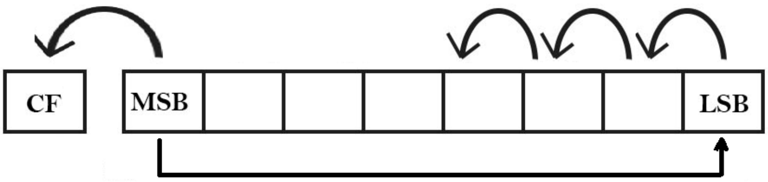
उदाहरण: ROR BX, 1
MOV CL, 4
ROR AL, CL

5. ROL destination, count [Rotate left without carry]

ही इंस्ट्रक्शन निर्दिष्ट बाइट किंवा वर्डचे सर्व बिट्स डावीकडे कॉउंटमध्ये दिल्या संख्ये वेळा फिरवते. MSB मधून फिरवलेला बिट LSB मध्ये जातो आणि CF मध्ये कॉपी देखील केला जातो. एका पेक्षा अधिक वेळा बिट फिरवल्यानंतर CF मध्ये LSB मधून अलीकडे फिरवलेल्या बिटची कॉपी असेल. डेस्टिनेशन ऑपरेंड हे बाइट किंवा वर्ड रजिस्टरमध्ये किंवा मेमरी लोकेशनमध्ये असू शकते. रोटेशनची इच्छित संख्या एक असल्यास, इंस्ट्रक्शनमध्ये काउन्टचा जागी 1 लिहून हे करता येते. परंतु, एका पेक्षा अधिक रोटेशनसाठी, इच्छित रोटेशनची संख्या CL रजिस्टरमध्ये लोड केली जाते आणि CL रेजिस्टर काउन्टच्या ठिकाणी ठेवला जाते.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : CF, OF,

ऑपरेशन:



उदाहरण:

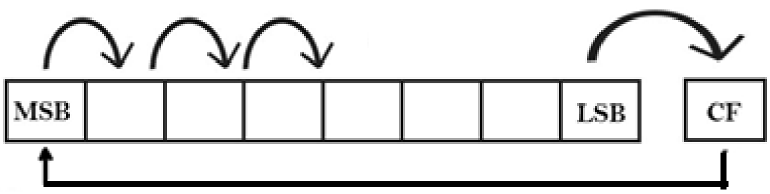
ROL BX, 1
MOV CL, 4
ROL AL, CL

6. RCR destination, count [Rotate right with carry]

ही इंस्ट्रक्शन निर्दिष्ट बाइट किंवा वर्डचे सर्व बिट्स उजवीकडे कॉउंटमध्ये दिल्या संख्ये वेळा फिरवते. LSB मधून फिरवलेला बिट CF मध्ये जातो आणि CF मधील बिट MSB मध्ये जाते. एका पेक्षा अधिक वेळा बिट फिरवल्यानंतर CF मध्ये LSB मधून अलीकडे फिरवलेल्या बिटची कॉपी असेल. डेस्टिनेशन ऑपरेंड हे बाइट किंवा वर्ड रजिस्टरमध्ये किंवा मेमरी लोकेशनमध्ये असू शकते. रोटेशनची इच्छित संख्या एक असल्यास, इंस्ट्रक्शनमध्ये काउन्टचा जागी 1 लिहून हे करता येते. परंतु, एका पेक्षा अधिक रोटेशनसाठी, इच्छित रोटेशनची संख्या CL रजिस्टरमध्ये लोड केली जाते आणि CL रेजिस्टर काउन्टच्या ठिकाणी ठेवला जाते.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : CF, OF,

ऑपरेशन:



उदाहरण:

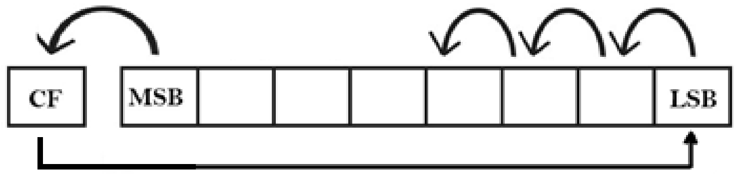
RCR BX, 1
MOV CL, 4
RCR AL, CL

7. RCL destination, count [Rotate left with carry]

ही इंस्ट्रक्शन निर्दिष्ट बाइट किंवा वर्डचे सर्व बिट्स डावीकडे कॉउंटमध्ये दिल्या संख्ये वेळा फिरवते. MSB मधून फिरवलेला बिट CF मध्ये जातो आणि CF मधील बिट LSB मध्ये जाते. एका पेक्षा अधिक वेळा बिट फिरवल्यानंतर CF मध्ये LSB मधून अलीकडे फिरवलेल्या बिटची कॉपी असेल. डेस्टिनेशन ऑपरेंड हे बाइट किंवा वर्ड रजिस्टरमध्ये किंवा मेमरी लोकेशनमध्ये असू शकते. रोटेशनची इच्छित संख्या एक असल्यास, इंस्ट्रक्शनमध्ये काउन्टचा जागी 1 लिहून हे करता येते. परंतु, एका पेक्षा अधिक रोटेशनसाठी, इच्छित रोटेशनची संख्या CL रजिस्टरमध्ये लोड केली जाते आणि CL रेजिस्टर काउन्टच्या ठिकाणी ठेवला जाते.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : CF, OF,

ऑपरेशन:



उदाहरण:

```
RCL BX, 1
MOV CL, 4
RCL AL, CL
```

3.3.8 स्ट्रिंग ऑपरेशन्स इंन्स्ट्रक्शन ग्रुप (String operation instruction group)

1. MOVS: Move String / MOVSB: Move String Byte /MOVSW: Move String Word.

इंस्ट्रक्शन MOVS सोअर्स स्ट्रिंगमधून डेस्टिनेशन स्ट्रिंगमध्ये बाइट किंवा वर्ड कॉपी करते. सोअर्स स्ट्रिंग डेटा सेगमेंट (Data Segment) मध्ये असणे आवश्यक आहे आणि डेस्टिनेशन स्ट्रिंग एक्सट्रा सेगमेंट (Extra Segment) मध्ये असणे आवश्यक आहे. सोअर्स बाइट किंवा वर्डचा ऑफसेट SI रजिस्टरमध्ये ठेवला जाणे आवश्यक आहे, जे DS:SI म्हणून दर्शविले जाते आणि डेस्टिनेशन बाइट किंवा वर्डचा ऑफसेट DI रजिस्टरमध्ये असणे आवश्यक आहे, जे ES : DI म्हणून दर्शविले जाते. या इंस्ट्रक्शन्स एक्सिक्युशन केल्यावर, SI आणि DI रजिस्टर आपोआप बाईट असेल तर एकने किंवा वर्ड असेल तर दोनने वाढवले जातात (जर DF=0 असेल) जे सोअर्स आणि डेस्टिनेशनच्या पुढील बाईट किंवा वर्ड ला ॲड्रेस करते. या इंस्ट्रक्शन्स एक्सिक्युशन केल्यावर, SI आणि DI रजिस्टर आपोआप बाईट असेल तर एकने किंवा वर्ड असेल तर दोनने कमी केले जातात (जर DF=1 असेल) जे सोअर्स आणि डेस्टिनेशनच्या आधीच्या बाईट किंवा वर्ड ला ॲड्रेस करते.

ऑपरेशन: ES:[DI] ← DS:[SI].

जर सोअर्स (Source) आणि डेस्टिनेशन (Destination) बाईट असेल तर

For DF = 0 SI ← SI + 1 and DI ← DI + 1.
For DF = 1 SI ← SI – 1 and DI ← DI – 1.

जर सोअर्स (Source) आणि डेस्टिनेशन (Destination) वर्ड असेल तर

For DF = 0 SI ← SI + 2 and DI ← DI + 2.
For DF = 1 SI ← SI – 2 and DI ← DI – 2.

उदाहरण:

```
MOV AX, @data
MOV DS, AX
MOV ES, AX
CLD
MOV SI, OFFSET STR_S
MOV DI, OFFSET STR_D
```

a. MOVS STR_S, STR_D
b. MOVSB
c. MOVSW

2. LODS: Load String / LODSB: Load String Byte/ LODSW: Load String Word.

LODS इंस्ट्रक्शन DS मधील SI द्वारे ऍड्रेस केलेल्या सोअर्स स्ट्रिंगमधून बाइट AL मध्ये किंवा वर्ड AX मध्ये कॉपी करते. इंस्ट्रक्शनचे एक्सिक्युशन केल्यावर, SI रजिस्टर आपोआप बाईट असेल तर एकने किंवा वर्ड असेल तर दोनने वाढवले जातात. जर DF = 0 असेल, तर रजिस्टर SI बाइटसाठी 1 ने वाढवला जातो आणि वर्डसाठी 2 ने वाढवला जातो. जर DF = 1 असेल, तर रजिस्टर SI बाइटसाठी 1 ने कमी केला जातो आणि वर्ड साठी 2 ने कमी केला जातो.

ऑपरेशन:

जर सोअर्स (Source) बाईट असेल तर AL ← DS:[SI]

For DF = 0 SI ← SI + 1

For DF = 1 SI ← SI – 1

जर सोअर्स (Source) वर्ड असेल तर AX ← DS:[SI]

For DF = 0 SI ← SI + 2

For DF = 1 SI ← SI – 2

उदाहरण:

```
    MOV AX, @data
    MOV DS, AX
    MOV ES, AX
    CLD
    MOV SI, OFFSET STR_S
a.  LODS STR_S
b.  LODSB
c.  LODSW
```

3. STOS: Store String/ STOSB: Store String Byte/ STOSW: Store String Word.

STOS इंस्ट्रक्शन ES मध्ये DI द्वारे ऍड्रेस केलेल्या लोकेशनवर AL मधून बाईट किंवा AX मधून वर्ड कॉपी करते. इंस्ट्रक्शनचे एक्सिक्युशन केल्यावर, DI रजिस्टर आपोआप बाईट असेल तर एकने किंवा वर्ड असेल तर दोनने वाढवले जातात. जर DF = 0 असेल, तर रजिस्टर DI बाइटसाठी 1 ने वाढवला जातो आणि वर्डसाठी 2 ने वाढवला जातो. जर DF = 1 असेल, तर रजिस्टर DI बाइटसाठी 1 ने कमी केला जातो आणि वर्ड साठी 2 ने कमी केला जातो.

ऑपरेशन:

जर डेस्टिनेशन (Destinetion) बाईट असेल तर ES:[DI] ← AL

For DF = 0 DI ← DI + 1

For DF = 1 DI ← DI – 1

जर डेस्टिनेशन (Destinetion) वर्ड असेल तर ES:[DI] ← AX

For DF = 0 DI ← DI + 2

For DF = 1 DI ← DI – 2

उदाहरण:

```
    MOV AX, @data
    MOV DS, AX
    MOV ES, AX
    CLD
    MOV SI, OFFSET STR_S
a.  STOS STR_S
b.  STOSB
c.  STOSW
```

4. CMPS: Compare String / CMPSB: Compare String Byte / CMPSW: Compare String Word.

CMPS इंस्ट्रक्शन सोअर्स स्ट्रिंगमधील बाइट किंवा वर्डची डेस्टिनेशन स्ट्रिंगमधील बाइट किंवा वर्डशी तुलना करते. सोअर्स स्ट्रिंग डेटा सेगमेंट (Data Segment) मध्ये असणे आवश्यक आहे आणि डेस्टिनेशन स्ट्रिंग एक्सट्रा सेगमेंट (Extra Segment) मध्ये असणे आवश्यक आहे. सोअर्स बाइट किंवा वर्डचा ऑफसेट SI रजिस्टरमध्ये ठेवला जाणे आवश्यक आहे, जे DS:SI म्हणून दर्शविले जाते आणि डेस्टिनेशन बाइट किंवा वर्डचा ऑफसेट DI रजिस्टरमध्ये असणे आवश्यक आहे, जे ES:DI म्हणून दर्शविले जाते. या इंस्ट्रक्शन्स एक्सिक्युशन केल्यावर, SI आणि DI रजिस्टर आपोआप बाईट असेल तर एकने किंवा वर्ड असेल तर दोनने वाढवले जातात (जर DF=0 असेल) जे सोअर्स आणि डेस्टिनेशनच्या पुढील बाईट किंवा वर्ड ला ॲड्रेस करते. या इंस्ट्रक्शन्स एक्सिक्युशन केल्यावर, SI आणि DI रजिस्टर आपोआप बाईट असेल तर एकने किंवा वर्ड असेल तर दोनने कमी केले जातात (जर DF=1 असेल) जे सोअर्स आणि डेस्टिनेशनच्या आधीच्या बाईट किंवा वर्ड ला ॲड्रेस करते.

एक्सिक्युशन नंतर फ्लॅगची स्थिती : AF, CF, OF, PF, SF, ZF

ऑपरेशन:

1. जर सोअर्स बाईट किंवा वर्ड डेस्टिनेशन बाईट किंवा वर्ड पेक्षा मोठा असला तर CF = 0, ZF = 0, SF = 0
2. जर सोअर्स बाईट किंवा वर्ड डेस्टिनेशन बाईट किंवा वर्ड पेक्षा लहान असला तर CF = 1, ZF = 0, SF = 1
3. जर सोअर्स बाईट किंवा वर्ड आणि डेस्टिनेशन बाईट किंवा वर्ड सारखे असले तर CF = 0, ZF = 1, SF = 0

जर सोअर्स (Source) आणि डेस्टिनेशन (Destinetion) बाईट असेल तर

For DF = 0 SI ← SI + 1 and DI ← DI + 1.
For DF = 1 SI ← SI – 1 and DI ← DI – 1.

जर सोअर्स (Source) आणि डेस्टिनेशन (Destinetion) वर्ड असेल तर

For DF = 0 SI ← SI + 2 and DI ← DI + 2.
For DF = 1 SI ← SI – 2 and DI ← DI – 2.

उदाहरण:

```
MOV AX, @data
MOV DS, AX
MOV ES, AX
CLD
MOV SI, OFFSET STR_S
MOV DI, OFFSET STR_D
```

a. CMPS STR_S, STR_D
b. CMPSB
c. CMPSW

5. SCAS: Scan String / SCASB: Scan String Byte / SCASW: Scan String Word

SCAS इंस्ट्रक्शन AL मधील स्ट्रिंग बाइट किंवा AX मधील स्ट्रिंग वर्ड डेस्टिनेशन स्ट्रिंगमध्ये स्कॅन करते. डेस्टिनेशन स्ट्रिंग ES मध्ये DI द्वारे ॲड्रेस केली जाते. इंस्ट्रक्शनचे एक्सिक्युशन केल्यावर, DI रजिस्टर आपोआप बाईट असेल तर एकने किंवा वर्ड असेल तर दोनने वाढवले जातात. जर DF = 0 असेल, तर रजिस्टर DI बाइटसाठी 1 ने वाढवला जातो आणि वर्डसाठी 2 ने वाढवला जातो. जर DF = 1 असेल, तर रजिस्टर DI बाइटसाठी 1 ने कमी केला जातो आणि वर्ड साठी 2 ने कमी केला जातो.

ऑपरेशन:

1. जर AL मधील बाईट किंवा AX मधील वर्ड डेस्टिनेशन बाईट किंवा वर्ड पेक्षा मोठा असेल तर CF = 0, ZF = 0, SF = 0

2. जर AL मधील बाईट किंवा AX मधील वर्ड डेस्टिनेशन बाईट किंवा वर्ड पेक्षा लहान असेल तर CF = 1, ZF = 0, SF = 1
3. जर AL मधील बाईट किंवा AX मधील वर्ड डेस्टिनेशन बाईट किंवा वर्ड सारखा असेल तर CF = 0,ZF = 1,SF = 0

जर डेस्टिनेशन (Destinetion) बाईट असेल तर

For DF = 0 DI ← DI + 1
For DF = 1 DI ← DI – 1

जर डेस्टिनेशन (Destinetion) वर्ड असेल तर

For DF = 0 DI ← DI + 2
For DF = 1 DI ← DI – 2

उदाहरण:

```
    MOV AX, @data
    MOV DS, AX
    MOV ES, AX
    CLD
    MOV SI, OFFSET STR_S
    MOV AL,'B'
a.  SCAS STR_S
b.  SCASB
c.  SCASW
```

6. REP: Repeat

हि इंस्ट्रक्शन स्ट्रिंग इंस्ट्रक्शन सोबत प्रीफिक्स इंस्ट्रक्शन म्हणून वापरली जाते आणि स्ट्रिंगचा अंतिम बाईट किंवा वर्ड येई पर्यंत स्ट्रिंग इंस्ट्रक्शन एक्सिक्युट करते. स्ट्रिंगची लांबी काउन्ट म्हणून CX रेजिस्टर मध्ये लोड केली जाते.

ऑपरेशन:

जर CX मधील काउन्ट झिरो (0) नसेल तर

1. स्ट्रिंग इंस्ट्रक्शन एक्सिक्युट करते
2. CX ← CX-1

उदाहरण:

```
MOV AX, @data
MOV DS, AX
MOV ES, AX
CLD
MOV CX, length of string
MOV SI, OFFSET S_STRING
MOV DI, OFFSET D_STRING
REP MOVSB or MOVSW
```

References:

1. Microprocessor and Interfacing (Programming and Hardware) by Douglas V. Hall [McGraw Hill Education, New Delhi ISBN-13: 978-0070257429]
2. The 8088 and 8086 Microprocessors: Programming, Interfacing, Software, Hardware, and Applications by Walter A. Triebel, Avtar Singh [Pearson Publications, New Delhi ISBN-13: 978-0131228047]
3. Microprocessor 8086: Architecture, Programming and Interfacing by Sunil Mathur [PHI, New Delhi ISBN-13: 978-8120340879]
4. Microprocessor X86 Programming by K. R. Venugopal and Raj Kumar [BPB Publications, Delhi ISBN-13: 978-8170294580

युनिट- 4
असेम्ब्ली लँग्वेज प्रोग्रामिंग
(Assembly Language Programming)

विषय निष्पत्ती(Course Outcome):

CO4: असेंबलर वापरुन दिलेल्या टास्कसाठी असेंब्ली लँग्वेज प्रोग्रॅम विकसित करा.

घटक निष्पत्ती(Theory Learning outcome):

1. दिलेल्या समस्येसाठी असेंब्ली लँग्वेज प्रोग्रामचे दिलेले मॉडेल वापरा.
2. दिलेल्या समस्येसाठी असेम्ब्ली लँग्वेज प्रोग्राम (ALP) विकसित करा.
3. दिलेल्या समस्येसाठी प्रोग्राममध्ये संबंधित नियंत्रण लूप लागू करा.
4. डेटाच्या दिलेल्या ब्लॉकच्या एलिमेंट्सना हाताळण्यासाठी स्ट्रिंग इंस्ट्रक्शन वापरा.

4.1 Models of 8086 assembly language program

8086 चे सर्वसाधारण असेंब्ली लँग्वेज प्रोग्रामिंगचे मॉडेल खाली दिले आहेत.

Model 1. SEGMENT, ASSUME and ENDS डिरेकटिव्स वापरून

```
Data_Seg    SEGMENT
                :
        Program Data Declaration
        [Data Segment of the program]
                :
Data_Seg    ENDS
Code_Seg    SEGMENT
        ASSUME CS: Code_Seg, DS:Data_Seg
            MOV AX, Data_Seg  ;Initialization of the data segment
            MOV DS, AX
                :
        Program Codes [Code Segment of the program]
                :
Code_Seg    ENDS
            END
```

वरील मॉडेलमध्ये, Data _Seg हे डेटा सेगमेंटचे नाव आहे ज्याचा वापर प्रोग्रामचा डेटा त्याच्या डेटा प्रकारासह घोषित करण्यासाठी केला जातो, म्हणजेच DB, DW इत्यादि. Code_Seg हे प्रोग्राम कोड लिहिण्यासाठी वापरल्या जाणाऱ्या कोड सेगमेंटचे नाव आहे. END हा प्रोग्राम संपुष्टात आल्याचे सूचित करतो.

Model 2 : सिम्पलीफाईड .DATA आणि .CODE डिरेकटिव्स वापरून

```
.MODEL SMALL
.STACK 100
.DATA
        :
    Program Data Declaration
    [Data Segment of the program]
        :
.CODE
MOV AX, @DATA   ;Data Segment initialization
MOV DS,AX
        :
```

```
                    Program Codes [Code Segment of the program]
                                :
            ENDS
            END
```

वरील मॉडेलमध्ये, .मॉडेल स्मॉल प्रोग्राममध्ये वापरण्यासाठी मेमरी मॉडेल सूचित करते. **.STACK** 100 हे सूचित करते की 100 word मेमरी लकेशन्स स्टॅकसाठी राखीव आहे. **.Data** डेटा सेगमेंटची सुरुवात सूचित करतो जेथे प्रोग्रामचा डेटाची घोषणा केली जाते. **.CODE** प्रोग्रामच्या वास्तविक कोडद्वारे कोड सेगमेंटची सुरुवात सूचित करतो. **ENDS** निर्देशीत डेटा आणि कोड सेगमेंटचा शेवट सूचित करतात आणि **END** प्रोग्राम समाप्ती दर्शवतात.

4.1.1 सिम्बॉल्स, व्हेरिएबल्स आणि काँस्टंट्स (Symbols, Variables and Constants)

व्हेरिएबल्स ही सिम्बॉल्स आहेत ज्यांचे मूल्य रन टाइम दरम्यान डायनॅमिकली बदलू शकते. असेंबलर व्हेरिएबलला मेमरी लोकेशन/लोकेशन्स नियुक्त करतो, जे यूजरसाठी थेट दृश्यमान नसते आणि असेंबली ल्यान्गवेज विधाने मशीन भाषा विधानांमध्ये भाषांतरित करते. व्हेरिएबलचे नाव असे निवडले पाहिजे की ते नाव त्याच्या मूल्याचा अर्थ दर्शवेल. उदाहरणार्थ, व्यक्तीचे वय स्टोअर करण्यासाठी एक अर्थ व्हेरिएबल नाव म्हणजे 'age' किंवा 'person_age'. म्हणून, कोणतेही व्हेरिएबल वापरले जाऊ शकते परंतु अर्थपूर्ण (Meaningful) व्हेरिएबलचे नाव प्रोग्रामची वाचनीयता(Readability) आणि देखभाल (Maintenance) सुलभता वाढवतात. असेंबलर सिम्बॉल्स,व्हेरिएबल्स आणि काँस्टंट्समध्ये खालील वर्ण असतात.

Upper case alphabets: A – Z	Lower case alphabets: a – z
Digits: 0 – 9	Special Characters_ , @, $, ?

व्हेरिएबल, सिम्बॉल्स आणि काँस्टंट्स नावांसाठीचे नियम खाली दिलेले आहेत.

1. व्हेरिएबल नावात खालीलपैकी कोणतेही वर्ण असू शकतात A – Z , a – z, 0 – 9, _(अंडरस्कोर), @
2. व्हेरिएबलचे नाव अक्षर (अल्फाबेट) किंवा अंडरस्कोरने (_) सुरू होणे आवश्यक आहे. त्यामुळे न्यूमेरिक डिजिट हा पहिला वर्ण (Character) म्हणून वापरता येत नाही.
3. व्हेरिएबल नावाची लांबी असेंबलरवर अवलंबून असते, साधारणपणे, कमाल लांबी 32 वर्ण (Character) असते.
4. असेम्ब्ली लॅंग्वेज मध्ये अप्पर (Upper) आणि लोअर (Lower) केस अक्षरांमध्ये भेद नाही. म्हणून, व्हेरिएबलची नावे केस असंवेदनशील (Case insensitive) आहेत.

वैध (Valid) व्हेरिएबलची उदाहरणे:

num1, age, my_data, array, result_lsb, count, average, small, large, next_i, name, total_marks, rate, baud_rate etc.

न्यूमेरिक कॉन्स्टंट्स (Numeric Constant)

न्यूमेरिक कॉन्स्टंट्स बायनरी (Binary) किंवा डेसिमल (Decimal) किंवा हेक्साडेसिमल (Hexadecimal) पूर्णांक म्हणून दर्शविले जाऊ शकतात. बायनरी संख्यांचा शेवट 'B' अक्षराने झाला पाहिजे, डेसिमल संख्यांचा शेवट 'D' अक्षराने झाला पाहिजे आणि हेक्साडेसिमल संख्यांचा शेवट 'H' अक्षराने झाला पाहिजे. तथापि, ‘D’, ‘B’ किंवा ‘H’ या दोन्ही अक्षरांनी न दिसणारी संख्या डेसिमल संख्या मानली जाते. वैध बायनरी कॉन्स्टन्ट नंबर मध्ये फक्त अंक 0 किंवा 1 असणे आवश्यक आहे, वैध डेसिमल नंबर मध्ये फक्त 0 ते 9 अंक असणे आवश्यक आहे आणि हेक्साडेसिमल कॉन्स्टन्ट नंबर मध्ये 0 ते 9 आणि A ते F दोन्ही असणे आवश्यक आहे. ज्या हेक्साडेसिमल क्रमांकाचा पहिला अंक A ते F मधील वर्ण आहे तो अंक 0 ने सुरू झाला पाहिजे, कारण BA, A0, CD इत्यादी संख्या न्यूमेरिक कॉन्स्टंट्स म्हणून नव्हे तर सिम्बॉल्स म्हणून मानली जाते. म्हणून हेक्साडेसिमल नंबर्स BA, A0, CD चा ऐवजी 0BA, 0A0, 0CD इ. असे लिहावे.

वैध Valid) न्यूमेरिक कॉन्स्टंट्सची उदाहरणे:

```
10101010      ; Decimal constant
10101010D     ; Decimal constant
10101010B     ; Binary constant
10101010H     ; Hexadecimal constant
9561H         ; Hexadecimal constant
0FB6AH        ; Hexadecimal constant
```

अवैध (Invalid) न्यूमेरिक कॉन्स्टंट्सची उदाहरणे:

```
10102101B    ; 2 हा बायनरी डिजिट नाही
09A7         ; A हा डेसिमल डिजिट नाही
ABC8         ; सिम्बॉल मानले जाते
```

4.2 असेंबलर वापरून प्रोग्रामिंग (Programming using Assembler)

1. नंबर्सची बेरीज (Addition of Numbers)

Program 1: दोन 8 बिट नंबर्सची बेरीज (Addition of two 8-bit numbers)

```
.model small
.data
        num1    db      85H             ;First Number
        num2    db      0A8H            ;Second Number
        res     db      ?               ;Result Variable
.code
        mov ax,@data                    ;Initialization of data segment
        mov ds, ax
        mov al, num1                    ;load 1st number in AL
        add al, num2                    ;add 2nd number with 1st number in AL
        mov res, al                     ;Store result from al to memory location
        ends
        end
```

वरील प्रोग्राममध्ये, num1, num2 आणि res साठी मेमरी लोकेशन असेंबलरद्वारेच वाटप केले जाईल, त्यामुळे त्या डेटा सेगमेंट डिक्लेरेशन आवश्यक आहे. म्हणून **.data** डिरेक्टिव्ह वापरून, प्रोग्राममध्ये आवश्यक व्हेरिएबल्ससह डेटा सेगमेंट घोषित केला जाऊ शकतो.

Program 2: दोन 16 बिट नंबर्सची बेरीज (Addition of two 16-bit numbers)

```
.model small
.data
        num1    dw      8A64H           ;First Number
        num2    dw      5F98H           ;Second Number
        res     dw?                     ;Result Variable
.code
        mov ax,@data                    ;Initialization of data segment
        mov ds,ax
        mov ax, num1                    ;load 1st number in AL
        add ax, num2                    ;add 2nd number with 1st number in AX
        mov res, ax                     ;Store result from AL to memory location
        ends
        end
```

Program 3 : दोन 16 बिट नंबरची बेरीज जेथे रिझल्ट 16 पेक्षा मोठा असू शकतो. (Addition of two 16 bit numbers where result may be more than 16 bit.)

```
.model small
.data
        num1            dw      0ffffh
        num2            dw      0ffffh
        res_lsb         dw      0
```

```
        res_msb     db    0
.code
        mov ax,@data    ;Initialize data segment
        mov ds,ax
        mov ax,num1     ;load num1 to AX
        add ax,num2     ;Add num1 and num2
        jnc exit        ; if result > 16 bit
        inc res_msb     ; increment carry counter
exit:   mov res_lsb,ax  ;store result
        ends
        end
```

वरील प्रोग्रामच्या रिझल्ट 16 बिटपेक्षा जास्त असू शकते, म्हणून डेटा सेगमेंटमध्ये res_lsb याचा डेटा टाईप DW आहे आणि res _msb ज्याचा डेटा टाईप DB आहे.

2. नंबर्सची वजाबाकी (Subtraction of Numbers)

Program 1: दोन 8 बिट नंबर्सची वजाबाकी (Subtraction of two 8-bit numbers)

```
.model small
.data
        num1    db    99H   ;First Number
        num2    db    09H   ;Second Number
        res     db    ?     ;Result Variable
.code
        mov ax,@data        ;Initialization of data segment
        mov ds,ax
        mov al, num1        ;load 1st number in AL
        sub al, num2        ;subtract 2nd number from 1st number
        mov res, al         ;Store result from al to  memory location
        ends
        end
```

Program 2: दोन 16 बिट नंबर्सची वजाबाकी (Subtraction of two 16-bit numbers)

```
.model small
.data
        num1    dw    0FFFFH    ;First Number
        num2    dw    0AAAAH    ;Second Number
        res     dw    ?         ;Result Variable
.code
        mov ax,@data        ;Initialization of data segment
        mov ds,ax
        mov ax, num1        ;load 1st number in AL
        sub ax, num2        ;subtract 2nd number from 1st number
        mov res, ax         ;Store result from al to memory location
        ends
        end
```

3. साईन्ड आणि अनसाईन्ड नंबर्सचा गुणाकार (Multiplication of Signed and Unsigned Numbers)

Program 1: दोन 8 बिट अनसाईन्ड नंबर्सचा गुणाकार (Multiplication of two 8 bit unsigned numbers)

```
.model small
.data
        num1        db  0FFh
        num2        db  0FFh
        result      dw  0
.code
        mov ax,@data        ;Initialize data segment
        mov ds,ax
        mov al,num1         ;Multiply num1 by num2
        mul num2
        mov result,ax       ;Store result
        ends
        end
```

वरील प्रोग्राममध्ये, num1 आणि num2 हे दोन 8 बिट अनसाईन्ड नंबर्स आहेत आणि त्याचा रिझल्ट 16 बिट असेल आणि AX रजिस्टरमध्ये रिझल्ट (FE01H) मिळेल जे 16 बिट आहे.

Program 2: दोन 8 बिट साईन्ड नंबर्सचा गुणाकार (Multiplication of two 8 bit signed Numbers)

```
.model small
.data
        num1        db  0FFh
        num2        db  0FFh
        result      dw  0
.code
        mov ax,@data        ;Initialize data segment
        mov ds,ax
        mov al,num1         ;Multiply num1 by num2
        imul num2
        mov result,ax       ;Store result
        ends
        end
```

वरील प्रोग्राममध्ये, num1=FFH(-1) आणि num2=FFH(-1) हे दोन 8 बिट साईन्ड नंबर्स आहेत आणि त्याचा रिझल्ट 16 बिट असेल आणि AX रजिस्टरमध्ये रिझल्ट (0001H) मिळेल जे 16 बिट आहे.

Program 3: दोन 16 बिट अनसाईन्ड नंबर्सचा गुणाकार (Multiplication of two 16 bit unsigned numbers)

```
.model small
.data
   num1      dw     0ffffh
   num2      dw     0ffffh
   res_lsb   dw     0
   res_msb   dw     0
.code
   mov ax,@data        ;Initialize data segment
   mov ds,ax
```

```
   mov ax,num1          ;Multiply num1 by num2
   mul num2
   mov res_lsb,ax       ;Store LSB of result
   mov res_msb,dx       ;Store MSB of result
   ends
   end
```

वरील प्रोग्राममध्ये, num1 आणि num2 हे दोन 16 बिट अनसाईन्ड नंबर्स आहेत आणि रिझल्ट कमाल 32 बिट मोठा असेल. आपल्याला DX:AX रजिस्टरमध्ये रिझल्ट (FFFE0001H)मिळेल जे दोन 16 बिट रेजिस्टर आहेत.

Program 4: दोन 16 बिट साईन्ड नंबर्सचा गुणाकार (Multiplication of two 16 bit signed numbers)

```
.model small
.data
   num1      dw    0ffffh
   num2      dw    0ffffh
   res_lsb   dw    0
   res_msb   dw    0
.code
   mov ax,@data         ;Initialize data segment
   mov ds,ax
   mov ax,num1          ;Multiply num1 by num2
   imul num2
   mov res_lsb,ax       ;Store LSB of result
   mov res_msb,dx       ;Store MSB of result
   ends
   end
```

वरील प्रोग्राममध्ये, num1=FFFFH(-1) आणि num2=FFFFH(-1) हे दोन 16 बिट साईन्ड नंबर्स आहेत आणि रिझल्ट कमाल 32 बिट मोठा असेल. आपल्याला DX:AX रजिस्टरमध्ये रिझल्ट (00000001H) मिळेल जे दोन 16 बिट रेजिस्टर आहेत.

4. साईन्ड आणि अनसाईन्ड नंबर्सचा भागाकार (Division of Signed and Unsigned Numbers)

Program 1: 16 बिट / 8 बिट अनसाईन्ड नंबर्सचा भागाकार (Division of 16 bit by 8 bit unsigned numbers)

```
model small
.data
        dividend     dw    0123h
        divisor      db    12h
        quo          db    0
        rem          db    0
.code
        mov ax,@data         ;Initialize data segment
        mov ds,ax
        mov ax, dividend     ;Divide word by byte
        div divisor
        mov quo,al           ;Store Quotient
        mov rem,ah           ;Store Remainder
        ends
```

```
        end
```

वरील प्रोग्राममध्ये, डिविडेंड हा वर्ड आहे म्हणजे 16 बिट नंबर आणि डिवाइजर हा बाइट म्हणजे 8 बिट नंबर आहे. भागाकारानंतर, कोशंट AH मध्ये 10H आणि रिमेंडर AL मध्ये 03H असेल.

Program 2: 32 बिट / 16 बिट अनसाईन्ड नंबर्सचा भागाकार (Division of 32 bit by 16 bit unsigned numbers)

```
.model small
.data
        div_lsw     dw      0ffffh      ;Lower word of dividend
        div_msw     dw      0001h       ;Higher word of dividend
        divisor     dw      0fh
        quo         dw      0
        rem         dw      0
.code
        mov ax,@data ;Initialize data segment
        mov ds,ax
        mov ax,div_lsw          ;load LSW of dividend
        mov dx,div_msw          ;load MSW of dividend
        div divisor             ;Divide double word by word
        mov quo,ax              ;Store Quotient
        mov rem,dx              ;Store Remainder
        ends
        end
```

OR

```
.model small
.data
        dividend    dd      0001FFFFH   ;Dividend as a single   variable
        divisor     dw      0FH
        quo         dw      0
        rem         dw      0
.code
        mov ax, @data                   ;Initialize data segment
        mov ds, ax
        mov ax, word ptr dividend       ;load LSW of Dividend
        mov dx, word ptr dividend+2     ;Load MSW of Dividend
        div divisor                     ;Divide Double Word by word
        mov quo, ax                     ;Store quotient
        mov rem, dx                     ;Store remainder
        ends
        end
```

वरील प्रोग्राममध्ये, डिविडेंड हा डबल वर्ड आहे म्हणजे 32 बिट नंबर ज्याचा LSW AX रेजिस्टर मध्ये स्टोअर केल्या जातो आणि MSW DX रेजिस्टर मध्ये स्टोअर केल्या जातो आणि डिवाइजर हा वर्ड म्हणजे 16 बिट नंबर आहे. भागाकारानंतर, कोशंट AX मध्ये 2222H आणि रिमेंडर DX मध्ये 0001H आहे.

Program 3: 16 बिट / 8 बिट साईन्ड नंबर्सचा भागाकार (Division of 16 bit by 8 bit signed numbers)

```
.model small
.data
        dividend        dw      –123h
        divisor         db      12h
        quo             db      0
        rem             db      0
.code
        mov ax,@data                    ;Initialize data segment
        mov ds,ax
        mov ax, dividend                ;Divide word bit by byte
        idiv divisor
        mov quo,al                      ;Store Quotient
        mov rem,ah                      ;Store Remainder
        ends
        end
```

वरील प्रोग्राममध्ये, डिविडेंड हा निगेटिव्ह वर्ड आहे म्हणजे 16 बिट निगेटिव्ह नंबर आणि डिवाइजर हा बाइट म्हणजे 8 बिट नंबर आहे. भागाकारानंतर, रिमेंडर FDH हा AH मध्ये 03 [– 03H] ची 2nd कॉम्प्लिमेंट आणि कोशंट F0H आहे AL मधील 10 [–10H] ची 2nd कॉम्प्लिमेंट आहे. 03 चा 2nd कॉम्प्लिमेंट FDH आहे आणि 10 2nd कॉम्प्लिमेंट F0H आहे कारण निगेटिव्ह नंबर्स बायनरीमध्ये त्या संख्येचा 2nd कॉम्प्लिमेंट म्हणून दर्शविल्या जातात.

Program 4: 32 बिट / 16 बिट साईन्ड नंबर्सचा भागाकार (Division of 32 bit by 16 bit signed numbers)

```
.model small
.data
        dividend        dd      – 12345h
        divisor         dw      12h
        quo             dw      0
        rem             dw      0
.code
        mov ax,@data                    ;Initialize data segment
        mov ds,ax
        mov ax, word ptr dividend       ;load LSW of Dividend
        mov dx, word ptr dividend+2     ;Load MSW of Dividend
        idiv divisor
        mov quo,ax                      ;Store Quotient
        mov rem,dx                      ;Store Remainder
        ends
        end
```

वरील प्रोग्रॅमचा एक्सिक्युशन नंतर, कोशंट AX मध्ये EFD2H आहे जो 102EH चे 2nd कॉम्प्लिमेंट म्हणजेच –102E आहे आणि रिमेंडर भाग DX मध्ये 0009H असायला पाहिजे परंतु आपल्याला FFF7H सापडेल जो 0009H चे 2nd कॉम्प्लिमेंट आहे म्हणजेच - 0009H .

5. BCD नंबर्सची बेरीज (Addition of BCD Numbers)

Program 1: दोन 8 बिट BCD नंबर्सची बेरीज (Addition of two 8-bit BCD numbers)

```
.model small
.data
        num1    db      09H             ; First Number
        num2    db      09H             ; Second Number
        res     db      ?               ; Result Variable
.code
        mov ax,@data                    ;Initialization of data segment
        mov ds, ax
        mov al, num1                    ; load 1st number in AL
        add al, num2                    ; add 2nd number with 1st number in AL
        daa
        mov res, al                     ; Store result from al to memory location
        ends
        end
```

वरील प्रोग्राम मध्ये बेरजेनंतर AL मध्ये 12H रिझल्ट मिळेल पण जेव्हा DAA एक्सिक्युट होईल, तेव्हा रिझल्ट डेसिमल मध्ये 18 मिळेल.

Program 2: दोन 16 बिट BCD नंबर्सची बेरीज (Addition of two 16-bit BCD numbers)

```
.model small
.data
        num1            dw      1234h
        num2            dw      4321h
        res             dw      0
.code
        mov ax,@data                    ; Initialize data segment
        mov ds,ax
        mov al,byte ptr num1            ; Add LSB first
        add al,byte ptr num2            ; Convert result to BCD
        daa
        mov byte ptr res, al            ; Store LSB of result
        mov al,byte ptr num1+1
        adc al,byte ptr num2+1          ; Add MSB  next
        daa                             ; Convert result to BCD
        mov byte ptr res +1,al          ; Store result
        ends
        end
```

6. BCD नंबर्सची वजाबाकी (Subtraction of BCD Numbers)

Program 1: दोन 8 बिट BCD नंबर्सची वजाबाकी (Subtraction of two 8-bit BCD numbers)

```
.model small
.data
        num1   db      90H     ; First BCD number
        num2   db      33H     ; Second BCD number
        res    db      0       ; Result variable
```

```
.code
        mov ax, @data           ; Initialize data segment
        mov ds, ax
        mov al, num1            ; load first number in AL
        sub al, num2            ; Subtract second no. from first
        das                     ; Adjust result to Correct BCD
        mov res, al             ; Store result
        ends
        end
```

Program 2: दोन 16-बिट BCD नंबर्सची वजाबाकी (Subtraction of two 16-bit BCD numbers)

```
.model small
.data
        num1    dw      8800h
        num2    dw      0088h
        result  dw      0
.code
        mov ax,@data                    ;Initialise data segment
        mov ds,ax
        mov al,byte ptr num1            ;Sub LSB first
        sub al,byte ptr num2            ;convert result to BCD
        das
        mov byte ptr result,al          ;Store result
        mov al,byte ptr num1+1
        sbb al,byte ptr num2+1          ;Sub MSB next
        das                             ;Convert result to BCD
        mov byte ptr result+1,al        ;Store result
        ends
        end
```

7. BCD नंबर्सचा गुणाकार (Multiplication of BCD Numbers)

Program 1: दोन 8 बिट BCD नंबर्सचा गुणाकार (Multiplication of two 8-bit BCD numbers)

बीसीडी (BCD) नंबर्सचा गुणाकार थेट करता येत नाही कारण गुणाकारानंतर बीसीडीमध्ये रिझल्ट समायोजित करण्यासाठी कोणतीही इंस्टक्शन उपलब्ध नाही. तर, बीसीडी गुणाकार करण्यासाठी सक्सेसिव्ह ॲडीशन (Successive Addition) पद्धत वापरली जाऊ शकते जिथे आपण ADD किंवा ADC आणि DAA इंस्ट्रक्शन वापरू शकतो. उदाहरणार्थ, समजा, 9 चा 2 ने गुणाकार करायचा असेल तर खाली दिलेल्या प्रमाणे 9 ची 2 वेळा किंवा 2 ची 9 वेळा बेरीज करू शकता.

9+9 किंवा 2+2+2+2+2+2+2+2+2

हे करण्यासाठी, बाइट गुणाकारासाठी CL रेजिस्टर मध्ये बेरजेच्या काउंटर म्हणून 9 किंवा 2 घेऊ शकता आणि वर्ड गुणाकारासाठी CX रेजिस्टर घेऊन अंमलबजावणी करू शकतो. जर 9 हा बेरीज काउंटर म्हणून घेतला, तर खालील दिलेल्या प्रोग्रॅमप्रमाणे गुणाकाराचा योग्य रिझल्ट मिळवण्यासाठी 2 ची 9 वेळा करता येईल.

```
.model small
.data
        num1            db      9H
        num2            db      2h
        res             db      0
```

```
.code
        mov ax, @data          ; Initialize data segment
        mov ds, ax
        mov cl, num2           ; multiplicand as a addition counter
        mov al, 0              ; initialize AL with 0
up : add al, num1              ; Add num2 to AL num1 times
        daa                    ; Adjust result to BCD
        dec cl                 ; decrement addition counter
        jnz up                 ; if addition counter not zero then go to up
        mov res, al            ;else store LSB of result
        ends
        end
```

Program 2: दोन 16 बिट BCD नंबर्सचा गुणाकार (Multiplication of two 16-bit BCD numbers)

```
.model small
.data
        num1       dw    9h          ; Multiplier
        num2       dw    9999h       ; Multiplicand
        res_lsw    dw    0
.code
        mov ax,@data           ; Initialize data Segment
        mov ds,ax
        mov cx,num1            ;Load multiplier as a Counter
        mov ax,0
up :
        add al,byte ptr num2   ; Multiply two BCD numbers
        daa
        mov byte ptr res_lsw,al  ; Successive Addition method
        mov al,ah
        adc al,byte ptr num2+1
        daa
        mov byte ptr res_lsw+1,al
        mov ax, res_lsw
        loop  up               ; IF addition counter not 0 go  to Up
        ends
        end
```

8. BCD नंबर्सचा भागाकार (Division of BCD Numbers)

बीसीडी नंबर्सच्या गुणाकाराच्या प्रमाणेच, बीसीडी भागाकारासाठी **8086** च्या इंस्ट्रक्शन सेट मध्ये कोणत्याही थेट इंस्ट्रक्शन्स उपलब्ध नाहीत. म्हणून**, SUB** किंवा **SBB** आणि **DAS** इंस्ट्रक्शन्स आवश्यक असलेल्या बीसीडी भागाकार ऑपरेशन करण्यासाठी सक्सेशिव वजाबाकी (Successive Subtraction) मेथड वापरली जाऊ शकते. या मेथडमध्ये, डिविडेंड मधून डिवाईझर वजा करतो जोपर्यंत डिवाईझर डिविडेंडपेक्षा समान किंवा मोठा असतो. तर, सक्सेशिव वजाबाकी (Successive Subtraction) मोजण्यासाठी एक काउंटर (Counter) आवश्यक आहे जे तुम्हाला भागाकार ऑपरेशनचे कोशंट (Quotient) देते आणि सलग वजाबाकीच्या शेवटी, रिमेंडर (Remender) मिळेल. उदाहरणार्थ, समजा **11** ला **2** ने भागायचे असेल तर ते खाली दिल्याप्रमाणे करता येईल.

स्टेप 1: 0 सह कोशंट काउंटर Q सुरू करा

स्टेप 2 : R = 11 – 02 = 09 ; R=Dividend -divisor

Increment Q = Q +1 = 1

Compare Divisor with R,

If R > Divisor perform next subtraction

स्टेप 3 : R = 09 – 02 = 07

Increment Q = Q +1 = 2

Compare Divisor with R,

If R > Divisor perform next subtraction

स्टेप 4 : R = 07 – 02 = 05

Increment Q = Q +1 = 3

Compare Divisor with R,

If R > Divisor perform next subtraction

स्टेप 5 : R = 05 – 02 = 03

Increment Q = Q +1 = 4

Compare Divisor with R,

If R > Divisor perform next subtraction

स्टेप 6 : R = 03 – 02 = 01

Increment Q = Q +1 = 5

Compare Divisor with R,

R < Divisor stop subtraction

Q = 5 आहे जे कोशंट आहे आणि R = 1 जे रिमेंडर आहे.

Program 1: दोन 8 बिट BCD नंबर्सचा भागाकार (Division of two 8-bit BCD numbers)

```
.model small
.data
        divisor     db      02H
        dividend    db      11H
        quo         db      0
        rem         db      0
.code
        mov ax,@data ; Initialize data segment
        mov ds,ax
        mov al,dividend         ; division using  successive
next:  sub al,divisor           ; Subtraction method
        das
        inc quo
        cmp al,divisor
        jnc next
        mov rem,al              ; Store remainder
        ends
        end
```

Program 2: दोन 16 बिट BCD नंबर्सचा भागाकार (Division of two 16-bit BCD numbers)

```
.model small
.data
```

```
        num1    dw      0009h           ;DIVISOR
        num2    dw      0999h           ;DIVIDEND
        quo     db     0
        rem     dw      0
.code
        mov ax,@data                    ;Initialize data Segment
        mov ds,ax
        mov bh,0
        mov ax,num2
up:
        sub al,byte ptr num1            ;Divide two BCD numbers
        das
        mov byte ptr rem,al             ;Successive Subtraction Methods
        mov al,ah
        sbb al,byte ptr num1+1
        das
        mov byte ptr rem+1,al
        mov al,bl                       ;Convert quotient to BCD
        add al,1
        daa
        mov bl,al
        mov quo,al
        mov ax,rem
        cmp ax,num1                      ;Compare result with divisor
        jge up
        ends
        end
```

9. सिरीझमधील (अ‍ॅरे) नंबर्सची बेरीज (Sum of Series (Array))

Program 1: मेमरीमध्ये साठवलेल्या N नंबर्सच्या अ‍ॅरेमधील 8-बिट नंबर्सची बेरीज करणे. (Sum of Series of 8 bit numbers in the array of N numbers)

इथे, बाइट किंवा वर्ड काउंटर (Counter) जे बाइट किंवा वर्डच्या टर्ममध्ये अ‍ॅरेची लांबी (Length) दर्शवतात, अ‍ॅरेमधून नंबर्स वाचण्यासाठी आवश्यक असते. **n** नंबर्सच्या अ‍ॅरेमधून बाइट किंवा वर्ड वाचण्यासाठी मेमरी पॉइंटर वापराने आवश्यक आहे.

```
.model small
.data
        array           db  99h,56h,78h,32h,46h
        sum_lsb         db  0
        sum_msb         db  0
.code
        mov ax,@data            ;Initialize data segment
        mov ds,ax
        mov cx,5                ;Initialize byte counter
        mov si,offset array     ;Initialize memory pointer
up:
```

```
        mov al,[si]             ;Read byte from memory
        add sum_lsb,al          ;Add with sum
        jnc next                ;If Sum>8 bit
        inc sum_msb             ;Increment msb counter
next:
        inc si                  ;Increment memory pointer
        loop up                 ;decrement byte counter
                                ;if byte counter = 0 then exit else read next number
        ends
        end
```

Program 2: मेमरीमध्ये साठवलेल्या n नंबर्सच्या ॲरेमधील 16-बिट नंबर्सची बेरीज करणे. (Sum of Series of 16 bit numbers)

```
.model small
.data
        array    dw  9999h,1111h,6666h,8888h,4444h
        sum_lsw         dw  0
        sum_msb         db   0
.code
        mov ax,@data            ;Initialize data segment
        mov ds,ax
        mov cx,5                ;Initialize word counter
        mov si,offset array     ;Initialize memory pointer
up:     mov ax,[si]             ;Read word from memory
        add sum_lsw,ax          ;Add with sum
        jnc next                ;If Sum >16 bit
        inc sum_msb             ;Increment msb counter next:
        inc si                  ;Increment memory pointer
        inc si
        loop up                 ;decrement word counter if word counter = 0 then exit
                                ;else read next number
        ends
        end
```

10. ॲरे मधील सर्वात लहान नंबर्स शोधणे (Find smallest Number from the Array)

ॲरे N बाइट किंवा वर्ड नंबरचा संच आहे आणि ॲरे मधील मेमरी लोकेशन वरून नंबर्स रीड करण्यासाठी मेमरी पॉईंटर आणि काउंटर आवश्यक आहे.

Program 1: पाच 8 बिट नंबरच्या ॲरेमधून सर्वात लहान संख्या शोधणे (Find Smallest number from the array of five 8 bit numbers)

```
.model small
.data
      array      db      12h,31h,02h,45h,65h
      small      db       0
.code
      mov ax,@data              ;Initialize data segment
      mov ds,ax
```

```
        mov cx,5                ;Initialize byte counter to read  numbers from  array
        mov si,offset array     ;Initialize memory pointer to read number
        mov al,[si]             ;read number from the array
        dec cx                  ;decrement byte counter by 1
up:     inc si                  ;increment memory pointer to point next number in array
        cmp al,[si]             ;compare numbers to find smallest ;number
        jc next                 ;if first number < second go to up
        mov al,[si]             ;compare it with next number decrement byte counter
next: loop up                   ;if it is NOT ZERO, compare  with next number in array
        mov small,al            ;Store smallest number from AL memory variable small
        ends
       end
```

Program 2: पाच 16 बिट नंबरच्या ॲरेमधून सर्वात लहान संख्या शोधणे (Find Smallest number from the array of five 16 bit numbers)

```
.model small
.data
        array       dw      12h,31h,02h,45h,65h
        small       dw      0
.code
        mov ax,@data            ;Initialize data segment
        mov ds,ax
        mov cx,5                ;Initialize word counter to read numbers from array
        mov si,offset array     ;Initialize memory pointer to read  number
        mov ax,[si]             ;read number from the array
        dec cx                  ;decrement word counter by 1
up:    inc si                   ;increment memory pointer to point next
        inc si                  ;number in array
        cmp ax,[si]             ;compare numbers to find smallest number
        jc next                 ;if it is smallest then compare it with
        mov ax,[si]             ;next number and decrement word counter
next:  loop up                  ;if it is NOT ZERO, compare with next number in array
        mov small, ax           ;Store smallest  number from AX  to
        ends                    ;memory variable small
       end
```

11. ॲरे मधील सर्वात मोठा नंबर शोधणे (Find largest Number from the Array)

ॲरे N बाइट किंवा वर्ड नंबरचा संच आहे आणि ॲरे मधील मेमरी लोकेशन वरून नंबर्स रीड करण्यासाठी मेमरी पॉईंटर आणि काउंटर आवश्यक आहे.

Program 1: पाच 8 बिट नंबरच्या ॲरेमधून सर्वात मोठा नंबर शोधणे (Find largest number from the array of five 8 bit numbers)

```
.model small
.data
        array      db      12h,31h,02h,45h,65h
        larg       db       0
.code
        mov ax,@data            ;Initialize data segment
```

```
        mov ds,ax
        mov cx,5                ;Initialize byte counter to read  numbers from  array
        mov si,offset array     ;Initialize memory pointer to read number
        mov al,[si]             ;read number from the array
        dec cx                  ;decrement byte counter by 1
up:     inc si                  ;increment memory pointer to point next number in array
        cmp al,[si]             ;compare numbers to find largest number
        jnc next                ;if first number > second go to up
        mov al,[si]             ;compare it with next number decrement byte counter
next: loop up                   ;if it is NOT ZERO, compare  with next number in array
        mov larg,al             ;Store largset number from AL memory variable larg
        ends
        end
```

Program 2: पाच 16 बिट नंबरच्या ॲरेमधून सर्वात मोठा नंबर शोधणे (Find largest number from the array of five 16 bit numbers)

```
.model small
.data
        array   dw      12h,31h,02h,45h,65h
        larg    dw      0
.code
        mov ax,@data            ;Initialize data segment
        mov ds,ax
        mov cx,5                ;Initialize word counter to read numbers from array
        mov si,offset array     ;Initialize memory pointer to read  number
        mov ax,[si]             ;read number from the array
        dec cx                  ;decrement word counter by 1
up:    inc si                   ;increment memory pointer to point next
        inc si                  ;number in array
        cmp ax,[si]             ;compare numbers to find largest number
        jnc next                ;if it is largest then compare it with
        mov ax,[si]             ;next number and decrement word counter
next:  loop up                  ;if it is NOT ZERO, compare with next number in array
        mov larg, ax            ;Store largest  number from AX  to memory variable larg
        ends
       end
```

11. N नंबर्सच्या ॲरेमध्ये चढत्या क्रमाने नंबर्सची क्रमवारी लावा. (Sort numbers in array of N numbers in Ascending Order)

```
.model small
.data
        array    dw     10h,56h,41h,36h,99h
.code
        mov ax,@data            ;Initialize data segment
        mov ds,ax
        mov bx,5                ;Initialize pass counter
up1:    mov si,offset array     ;Initialize memory pointer
        mov cx,4                ;Initialize counter
```

```
up:    mov ax,[si]
       cmp ax,[si+2]         ;Compare two number
       jc dn                 ;if number < next number then go to dn
       xchg ax,[si+2]        ;interchange numbers
       xchg ax,[si]
dn:  add si,2                ;increment memory pointer
       loop up               ;decrement counter if not equal 0 then up
       dec bx                ;decrement pass counter if not equal 0 then up1
       jnz up1
       ends
       end
```

12. N नंबर्स च्या ॲरेमध्ये उतरत्या क्रमाने नंबर्सची क्रमवारी लावा. (Sort numbers in array of N numbers in Descending Order)

```
.model small
.data
       array    dw     10h,56h,41h,36h,99h
.code
       mov ax,@data          ;Initialize data segment
       mov ds,ax
       mov bx,5              ;Initialize pass counter
up1:   mov si,offset array   ;Initialize memory pointer
       mov cx,4              ;Initialize counter
up:    mov ax,[si]
       cmp ax,[si+2]         ;Compare two number
       jnc dn                ;if number > next number then go to dn
       xchg ax,[si+2]        ;interchange numbers
       xchg ax,[si]
dn:  add si,2                ;increment memory pointer
       loop up               ;decrement counter if not equal 0 then up
       dec bx                ;decrement pass counter if not equal 0 then goto up1
       jnz up1
       ends
       end
```

13. दिलेल्या नंबर ऑड किंवा इव्हन आहे हे तपासा. (Check the given number is ODD or EVEN)

D_0 बिटचे डेसिमल वेटेज 1 ऑड (ODD) आहे म्हणून D_0 बिट ठरवते कि दिलेला नंबर ऑड किंवा इव्हन आहे आणि उरलेल्या बिट्सचे वेटेज 2, 4, 8… इत्यादी इव्हन (EVEN) आहेत. म्हणूनच आपण दिलेला नंबर एका बिटने उजवीकडे फिरवून फक्त D_0 बिट तपासू. जर बिट 1 असले तर नंबर ऑड आहे अन्यथा इव्हन आहे.

```
.model small
.data
        num          db      54H
        odd_num      db      ?
        eve_num      db      ?
.code
        mov ax,@data
```

```
        mov ds,ax
        mov al,num
        rol al,1
        jnc dn
        ror al,1
        mov odd_num,al
        jmp exit
dn:     ror al,1
        mov eve_num,al
exit:   ends
        end
```

14. दिलेल्या नंबर पॉझिटिव्ह किंवा निगेटिव्ह आहे हे तपासा. (Check given number is positive or negative)

8 बिट किंवा 16 बिट साईड नंबरमध्ये, सर्वात शेवटची बिट म्हणजे D_7 किंवा D_{15} हि बिट नंबर्सचे साइन दर्शवते. म्हणूनच आपण दिलेला नंबर एका बिटने डावीकडे फिरवून फक्त D_7 किंवा D_{15} बिट तपासू. जर बिट 1 असले तर नंबर निगेटिव्ह आहे अन्यथा पॉजिटीव्ह आहे.

```
.model small
.data
        num         db      -20H
        pos_num     db      ?
        neg_num     db      ?
.code
        mov ax,@data
        mov ds,ax
        mov al,num
        rol al,1
        jnc dn
        mov neg_num,al
        jmp exit
dn:     mov pos_num,al
exit:   ends
        end
```

15. ब्लॉक ट्रान्फर ऑपरेशन (Block Transfer operation)

Program 1: N नंबरच्या सोअर्स ॲरे मधून डेस्टिनेशन अरेमध्ये स्ट्रिंग इंस्ट्रक्शन न वापरता नंबर्स कॉपी करा म्हणजेच ब्लॉक ट्रान्सफर ऑपरेशन. (Copy numbers from source array of N numbers to destination array i.e. Block Transfer without using string instruction)

```
.model small
.data
        src_arr     dw      4545h, 5656h, 3232h, 9898h, 7474h
        dst_arr     dw      5 dup(0)        ;Declare empty array
.code
        mov ax, @data               ;Initialize data segment
        mov ds,ax
        mov cx,5                    ;Initialize word counter
        mov si, offset src_arr      ;Initialize memory pointer for source
```

```
        mov di, offset dst_arr          ;Initialize memory pointer for destination
up:     mov ax,[si]                     ;read number from source array
        mov [di],ax                     ;write number to destination array
        add si,2                        ;increment source memory pointer
        add di,2                        ;increment destination memory ;pointer
        loop next                       ;check word counter for zero, if not zero then up
        ends
        end
```

Program 2: N नंबरच्या सोअर्स ॲरे मधून डेस्टिनेशन ॲरेमध्ये स्ट्रिंग इंस्ट्रक्शन वापरून नंबर्स कॉपी करा म्हणजेच ब्लॉक ट्रान्सफर ऑपरेशन. (Copy numbers from source array of N numbers to destination array i.e. Block Transfer using string instruction)

```
.model small
.data
        src_arr     dw      4545h, 5656h, 3232h, 9898h, 7474h
        dst_arr     dw      5 dup(0)        ;Empty array
.code
        mov ax, @data
        mov ds,ax                       ;Initialize data segment
        mov es,ax                       ;Initialize extra segment
        mov cx,5                        ;Initialize word counter
        mov si, offset src_arr          ;Initialize memory pointer for source
        mov di, offset dst_arr          ;Initialize memory pointer for destination
up :    movsw                           ;Transfer word from source to destination
        loop up
        ends
        end
```

16. स्ट्रिंग ऑपरेशन्स (String Operations)

स्ट्रिंगमध्ये नंबर्स किंवा वर्ण असतात. लँग्वेज प्रोग्राममध्ये, स्ट्रिंग सिंगल कोट ' ' मध्ये घोषित करणे आवश्यक आहे आणि स्ट्रिंगचा शेवट '**$**' चिन्हाने केला पाहिजे. स्ट्रिंगचा डेटा प्रकार नेहमी बाइट टाईप असतो कारण असेंबलर मेमरीमध्ये स्ट्रिंगच्या प्रत्येक कॅरेक्टरचे ASCII व्हॅल्यू स्टोअर करतो जे 8 बिट असतात.

Program 1: स्ट्रिंगची लांबी शोधणे (Find length of string)

```
.model small
.data
        str_s    db     'MSBTE$'
        length   db     0
.code
        mov ax,@data            ;Initialize data segment
        mov ds,ax
        mov si,offset str_s     ;initialize memory pointer
next:   mov al,[si]             ;read character
        cmp al,'$'              ;check for end of string
        je exit                 ; if not end of string then
        inc si                  ;increment memory pointer
        inc length              ;increment length counter
```

```
        jmp next                ;jump to read next character
exit:
        ends
        end
```

Program 2: स्ट्रिंग कॉपी करणे (String copy)

```
.model small
.data
        str_s     db      'MSBTE$'
        str_d     db      10 dup('$')
        length   db      0
.code
        mov ax,@data            ;Initialize data segment
        mov ds,ax
        mov si,offset str_s     ;initialize memory pointers
        mov di,offset str_d
next:   mov al,[si]             ;read character
        cmp al,'$'              ;check for end of string
        je exit                 ; if not end of string then
        mov [di],al             ; Copy character to destination string
        inc si                  ;increment memory pointers
        inc di
        inc length              ;increment length counter
        jmp next                ;jump to read next character
exit:   ends
        end
```

Program 3: स्ट्रिंग रिवर्स करणे (String reverse)

```
.model small
.data
        str_s    db   MSBTE$'
        str_d    db  50 dup('$')
        count   db   0
.code
        mov ax,@data            ;initialize data segment
        mov ds,ax               ;Calculate length of source string
        mov si,offset str_s     ;Initialize memory pointer for source string
next: mov al,[si]               ;read first character from source string
        cmp al,'$'              ;check for end of string, if yes then exit
        je exit                 ;else
        inc si                  ;Increment memory pointer
        inc count               ;increment length counter
        jmp next                ;jump to read next character
exit :                          ;Copy Source String to Destination in reverse order
        mov di,offset str_d     ;initialize memory pointer for destination string
up :    dec si                  ;decrement memory pointer for source string
```

```
        mov al,[si]          ;read character from source string in reverse order
        mov [di],al          ;copy it to destination string in forward order
        inc di               ;increment memory pointer for destination string
        dec count            ;decrement length counter
        jnz up               ;if length counter ≠ 0 then jump up to copy next
        ends
        end
```

Program 4: दोन स्ट्रिंग जोडणे (String concatenate)

```
.model small
.data
        str_s   db      'MSBTE$'
        str_d   db      'MUMBAI$'
.code
        mov ax,@data         ;Initialize data segment
        mov ds,ax
        mov si,offset str_s  ;Initialize memory pointer for source string
 next:  mov al,[si]
        cmp al,'$'           ;Is memory pointer is at last character
        je exit              ;if yes then jump to concatenate string
        inc si
        jmp next
exit :  mov di,offset str_d  ;initialize memory pointer for destination string
up :    mov al,[di]          ;read character
        cmp al,'$'           ;check end of string
        je exit1             ;if yes then exit
        mov [si],al          ;else copy character to source string
        inc si               ;increment memory pointers for
        inc di               ;source and destination
        jmp up               ;repeat process till end of  destination string
exit1:
        ends
        end
```

Program 5: स्ट्रिंग कॅरेक्टर लोवर केस मधून अप्पर केस मध्ये बदलणे (Convert string from lower case to upper case)

```
.model small
.data
                str_l   db      'msbte$'
                str_u   db      20 dup('$')
.code
                mov ax,@data         ;Initialize data segment
                mov ds,ax
                mov si,offset str_l  ;initialize memory pointers
                mov di,offset str_u
next:           mov al,[si]          ;Read character from array
```

```
            cmp al,'$'          ;If end of string then exit
            je exit
            sub al,20H          ;Convert to upper case
            mov [di],al         ;store into memory
            inc si              ;increment memory pointers
            inc di
            jmp next            ; goto next character
exit:       ends
            end
```

Program 6: स्ट्रिंग कॅरेक्टर अप्पर केस मधून लोवर केस मध्ये बदलणे (Convert string from upper case to lower case)

```
.model small
.data
            str_l   db      'MSBTE$'
            str_u   db      20 dup('$')
.code
            mov ax,@data        ;Initialize data segment
            mov ds,ax
            mov si,offset str_l ;initialize memory pointers
            mov di,offset str_u
next:       mov al,[si]         ;Read character from array
            cmp al,'$'          ;If end of string then exit
            je exit
            sub al,20H          ;Convert to lower case
            mov [di],al         ;store into memory
            inc si              ;increment memory pointers
            inc di
            jmp next            ; go to next character
exit:       ends
            end
```

Program 7: स्ट्रिंगची तुलना (String Compare)

```
.model small
.data
        str_s   db      'MSBTE$'
        str_d   db      'MSBTE$'
        count_s         db      0
        count_d         db      0
        msg1            db      'Strings are Equal$'
        msg2            db      'Strings are Not Equal$'
.code
        mov ax,@data            ;Initialize data segment
        mov ds,ax
        mov si,offset str_s     ;Initialize memory pointer for source string
next:   mov al,[si]             ;read character from the source string
```

```
        cmp al,'$'          ;compare with $
        je exit             ;if equal then go to exit to read
        ;destination ;string
        inc si              ;else increment memory pointer
        inc count_s         ;increment counter to count length of string
        jmp next            ;jump to read next character
        exit:
        mov si,offset str_d ;Initialize memory ptr for destination string
next1:  mov al,[si]         ;read character from the destination string
        cmp al,'$'          ;compare with $
        je exit1            ;if equal then go to exit1 to compare length of both strings
        inc si              ;else increment memory pointer
        inc count_d         ;increment counter to count  length  of string
        jmp next1           ;jump to read next character exit1:
        mov al,count_s
        cmp al,count_d
        jne exit2           ;If length of both strings are not same then go to exit2 else compare
                            ;strings character by charecter
        mov ah,09h          ;Display string are same message
        lea dx,msg1
        int 21h
        jmp exit3
exit2:  mov ah,09h          ;Display string are not same message
        lea dx,msg2
        int 21h
exit3:  mov ah,4ch          ;Terminate program & Exit to DOS
        int 21h
        ends
        end
```

17. 16-बिट नंबरमध्ये बायनरी '1' आणि '0' ची संख्या मोजा (Count numbers of 'l' and '0' in 16-bit number)

कोणत्याही 16 बिट नंबरमध्ये बायनरी **'1'** किंवा **'0'** मोजण्यासाठी आपण नंबरला 16 वेळा उजवीकडे किंवा डावीकडे रोटेट करून मोजू शकतो. त्यासाठी आपण ROR/RCR किंवा ROL/RCL मधील कोणतेही एक इंस्ट्रक्शन वापरू शकतो.

```
.model small
.data
      num    dw     55aah
      zeros  db     0
      ones   db     0
.code
      mov ax,@data          ;Initialize data segment
      mov ds,ax
      mov cx,16             ; initialize rotation counter
      mov ax,num            ;load number in AX
```

```
up :   ror ax,1          ;Rotate number by 1 bit right
       jc dn             ;if bit ≠ 0 then go to dn
       inc zeros
       jmp    next       ;else increment zeros by one
dn:    inc ones
next:  loop up           ;decrement rotation counter
       ends              ;if rotation counter ≠ 0 then to up
      end
```

References:

5. Microprocessor and Interfacing (Programming and Hardware) by Douglas V. Hall [McGraw Hill Education, New Delhi ISBN-13: 978-0070257429]
6. The 8088 and 8086 Microprocessors: Programming, Interfacing, Software, Hardware, and Applications by Walter A. Triebel, Avtar Singh [Pearson Publications, New Delhi ISBN-13: 978-0131228047]
7. Microprocessor 8086: Architecture, Programming and Interfacing by Sunil Mathur [PHI, New Delhi ISBN-13: 978-8120340879]
8. Microprocessor X86 Programming by K. R. Venugopal and Raj Kumar [BPB Publications, Delhi ISBN-13: 978-8170294580

युनिट- 5
प्रोसिजर आणि मॅक्रो
(Procedure and Macro)

विषय निष्पत्ती(Course Outcome):

CO5: दिलेल्या समस्येसाठी असेंब्ली लँग्वेज प्रोग्राम विकसित करण्यासाठी प्रोसिजर आणि मॅक्रो वापरा.

घटक निष्पत्ती(Theory Learning Outcome):

1. दिलेल्या परिस्थितीत संबंधित 'पॅरामीटर-पासिंग' पद्धत लागू करा.
2. दिलेल्या समस्येसाठी संबंधित प्रोसिजर वापरून असेंब्ली लँग्वेज प्रोग्राम विकसित करा.
3. दिलेल्या समस्येसाठी संबंधित मॅक्रो वापरून असेंब्ली लँग्वेज प्रोग्राम विकसित करा.
4. दिलेल्या पॅरामीटरच्या आधारावर प्रोसिजर आणि मॅक्रोची तुलना करा.

5.1 प्रोसिजरचा परिचय (Introduction to Procedure)

एखाद्या प्रोग्राममध्ये, आम्हाला अशा परिस्थितीत सामोरे जावे लागते जेथे पुन्हा पुन्हा समान टास्क करण्याची आवश्यकता आहे. तर, इंस्ट्रक्शन्सचा समान क्रम पुन्हा पुन्हा लिहिण्याऐवजी, ते सब-प्रोग्राम (Sub Program) मध्ये स्वतंत्रपणे लिहिले जातात ज्याला प्रोसिजर (Procedure) म्हणतात. प्रोसिजरच्या मदतीने आपण आपल्या प्रोग्राममध्ये मॉड्यूलर प्रोग्रामिंगची (Modular Programming) संकल्पना फारच चांगली अंमलात आणू शकतो. तसेच, जेव्हा जेव्हा आपल्याला प्रोसिजरमध्ये नमूद केलेल्या इंस्ट्रक्शन्स एक्सिक्युट करण्याची आवश्यकता असते, तेव्हा आपण त्यास फक्त कॉल (CALL) करू शकतो. म्हणूनच, प्रोसिजरच्या मदतीने, इंस्ट्रक्शन्स मधील डुप्लिसिटी (Duplicity) टाळता येते.

5.1.1 प्रोसिजर परिभाषित करणे (Defining Procedure)

PROC आणि ENDP डिरेक्टिव्हसचा उपयोग प्रोसिजर परिभाषित करण्यासाठी केला जातो जेथे डिरेक्टिव्ह PROC प्रोसिजरची सुरूवात सांगते आणि डिरेक्टिव्ह ENDP असेंबलरला प्रोसिजरचा शेवट सांगते. डिरेक्टिव्ह PROC आणि ENDP ने सबरुटीन परिभाषित करणाऱ्या प्रोसिजरचा कोड संलग्न करणे आवश्यक आहे. नियर (NEAR) प्रोसिजर केवळ त्याच कोड सेगमेंटमध्येच परिभाषित केल्या पाहिजेत जिथे कॉलिंग प्रोग्राम (Calling program) असतो आणि फार (FAR) प्रोसिजर दुसऱ्या कोड सेगमेंटमध्ये परिभाषित केली पाहिजे. प्रोसिजरसाठीची वाक्यरचना (Syntax) खालीलप्रमाणे आहे:

```
procedure_name PROC [NEAR / FAR]
            :
      code of procedure
            :
procedure_name ENDP
```

5.1.2 CALL आणि RET इंस्ट्रक्शन्स

1. CALL procedure

कॉल (CALL) इंस्ट्रक्शनचा वापर प्रोग्रॅम कंट्रोल सब-प्रोग्राम (Sub-program) किंवा सब रूटीनमध्ये (Sub Routine) हस्तांतरित करण्यासाठी केला जातो. निअर (Near) आणि फार (Far) हे कॉलचे दोन मूलभूत प्रकार आहेत. निअर कॉल (NEAR CALL) म्हणजे एक प्रोसिजरसाठी कॉल आहे, जो कॉल इंस्ट्रक्शन प्रमाणेच त्याच कोड सेगमेंटमध्ये असतो. फार कॉल (FAR CALL) म्हणजे एका प्रोसिजरसाठी कॉल आहे, जो दुसऱ्या कोड सेगमेंटमध्ये असतो जिथे कॉल इंस्ट्रक्शन नसते. निअर कॉल साठी IP ची व्हॅलू स्टॅक वर स्टोअर केली जाते आणि फार कॉल साठी CS आणि IP ची व्हॅलू स्टॅक वर स्टोअर केली जाते

ऑपरेशन:

1. जर निअर कॉल (NEAR CALL) असेल तर

   SP ← SP – 2
   Save IP on stack
   IP ← address of procedure

2. जर फार कॉल (FAR CALL) असेल तर

SP ← SP — 2
SP ← CS i.e. Save CS on stack
CS ← New segment base address of the called procedure
SP ← SP – 2
SP ← IP i.e. Save IP on stack
IP ← New offset address of the called procedure

उदाहरण: CALL FACT
CALL FAR PTR FIBON

2. RET instruction

कॉलिंग प्रोग्राममधील CALL इंस्ट्रक्शननंतर लगेचच प्रोग्रॅम एक्झिक्यूशन कंट्रोलला प्रोग्रॅममधील पुढील इंस्ट्रक्शनवर परत करण्यासाठी RET इंस्ट्रक्शनचा वापर केला जातो. जर प्रोसिजर हि नियर प्रोसिजर (NEAR Procedure) असेल, तर स्टॅक टॉपवरून एक वर्ड IP ची व्हॅलू पुनर्संचयित करून प्रोग्रॅम कंट्रोल कॉलिंग प्रोग्रॅम मध्ये परत जाईल. कॉलिंग प्रोग्राममधील CALL इंस्ट्रक्शन नंतर स्टॅक वरचा वर्ड पुढील इंस्ट्रक्शनचा ऑफसेट ऍड्रेस आहे जो CALL इंस्ट्रक्शन द्वारे स्टॅकवर स्टोअर केलेला असतो. जर प्रोसिजर हि फार प्रोसिजर (FAR Procedure) असेल, तर स्टॅक टॉपवरून दोन वर्ड IP व CS ची व्हॅलू पुनर्संचयित करून प्रोग्रॅम कंट्रोल कॉलिंग प्रोग्रॅम मध्ये परत जाईल.

ऑपरेशन:

1. जर नियर रिटर्न (NEAR RETURN) असेल तर

IP ← content of top of stack.
SP ← SP + 2.

2. जर फार रिटर्न (FAR RETURN) असेल तर

IP ← contents of top of stack.
SP ← SP + 2.
CS ← contents of top of stack.
SP ← SP + 2.

5.1.3 प्रोसिजरमध्ये पॅरामीटर पासिंगच्या पद्धती (Parameter passing methods in Procedure)

पॅरामीटर्स खालील चार पद्धतीचा वापर करून मुख्य प्रोग्राममधून प्रोसिजरला पास केले जाऊ शकतात.

1. रजिस्टरचा वापर करून पॅरामीटर्स पासिंग (Passing parameters using registers)
2. मेमरी वापरुन पॅरामीटर्स पासिंग (Passing parameters using memory)
3. पॉईंटर्स वापरुन पॅरामीटर्स पासिंग (Passing parameters using pointers)
4. स्टॅक वापरुन पॅरामीटर्स पासिंग (Passing parameters using stack)

1. रजिस्टरचा वापर करून पॅरामीटर्स पासिंग (Passing parameters using registers)

प्रोसिजरमध्ये पास करणारा डेटा रजिस्टरमध्ये स्टोअर केला जाणे आवश्यक आहे आणि डेटावर प्रक्रिया करण्यासाठी या रजिस्टरमधून त्याचा वापर केला जातो.

उदाहरण:

```
.model small
.data
    num1 dw  1234h
    num2 dw  4232h
.code
    mov ax,@data
    mov ds,ax
    mov ax, num1
```

```
        mov bx, num2
        call mul_num
    mul_num proc near
        mul bx          ; प्रोसिजर डेटा AX आणि BX रजिस्टर मधून घेतो
        ret
    mul_num endp
        ends
        end
```

पॅरामीटर्स पास करण्यासाठी रजिस्टरचा वापर करण्याचा गैरसोय आहे म्हणजे रजिस्टरची संख्या पाहिली तर पॅरामीटर्सची संख्या मर्यादित होते. रजिस्टरचा वापर करून 100 नंबरचा अॅरे प्रोसिजरला पास केला जाऊ शकत नाही.

2. मेमरी वापरुन पॅरामीटर्स पासिंग (Passing parameters using memory)

ज्या प्रकरणांमध्ये मोजकेच पॅरामीटर्स प्रोसिजरकडे पास करावे लागतात अशा प्रकरणांमध्ये, रेजिस्टर्स सोयीस्कर असतात. परंतु, जेव्हा आपल्याला प्रोसिजरसाठी मोठ्या संख्येने पॅरामीटर्स पास करण्याची आवश्यकता असते तेव्हा आपण मेमरी वापरू शकतो.

उदाहरण:

```
    .model small
    .data
        num1  dw    1234h
        num2  dw    4232h
        result  dw    ?
    .code
        mov ax, @data
        mov ds, ax
        call mul_num
    mul_num proc near
        mov ax, num1
        mul num2
        mov word ptr result, ax
        mov word ptr result, dx
        ret
    mul_num endp
        end
```

3. पॉईंटर्स वापरुन पॅरामीटर्स पासिंग (Passing parameters using pointers)

या पॅरामीटर पासिंग पद्धतीत, आपण इच्छित डेटा वापर करण्यासाठी मेमरी पॉईंटर्स वापरू शकतो जे प्रोसिजर मध्ये थेट डेटा व्हेरिएबलची नावे वापरण्याच्या गैरसोयवर मात करू शकतो.

उदाहरण:

```
    .model small
    .data
        num1  dw    1234h
        num2  dw    4232h
        result  dw    ?
    .code
        mov ax, @data
        mov ds, ax
```

```
        mov si, offset num1
        mov di, offset num2
        mov bx, offset result
        call mul_num
    mul_num proc near
        mov ax, [si]            ; SI मेमरी पॉईंटर वापरून मल्टीप्लिकेन्ड AX रजिस्टरमध्ये आणतो
        mul [di]                ; DI मेमरी पॉईंटर वापरून मल्टीप्लायर आणतो
        mov [bx], ax            ; BX मेमरी पॉईंटर वापरून रिझल्ट स्टोअर करतो
        mov [bx+2], dx
        ret
    mul_num   endp
        ends
        end
```

4. स्टॅक वापरुन पॅरामीटर्स पासिंग (Passing parameters using stack)

स्टॅक वापरुन पॅरामीटर्स पास करण्यासाठी आपण मुख्य प्रोग्राममधील प्रोसिजरसाठी कॉल करण्यापूर्वी त्यांना स्टॅकवर स्टोअर (PUSH) करतो. प्रोसिजर मध्ये वापरल्या जाणाऱ्या इंस्ट्रक्शन्स स्टॅकमधून रीड (READ) करून हे पॅरामीटर्स वापरतात. जेव्हा जेव्हा स्टॅक पॅरामीटर्स पास करण्यासाठी वापरला जातो तेव्हा स्टॅकवर काय स्टोअर (PUSH) केले जाते आणि मुख्य प्रोग्राममधील स्टॅकवरून काय रीड (READ) केले याचा मागोवा ठेवणे महत्वाचे आहे.

उदाहरण:

```
    .model small
    .data
        num1   dw      1234h
        num2   dw      4232h
        res            dw      ?
    .code
        mov ax, @data
        mov ds, ax
        push num1
        push num2
        call mul_num
        mov word ptr res,ax
        mov word ptr res+2,dx
mul_num proc near
        push bp
        mov bp, sp            ;copies offset of sp into bp
        mov ax, [bp + 6]      ;num1 value is available at [bp + 6] and is passed to ax
        mul word ptr [bp + 4]       ;num2 value is passed
        pop bp
        ret
mul_num endp
        ends
        end
```

5.1.4 री-एण्ट्रन्ट प्रोसिजर (Re-entrant Procedure)

काही परिस्थितींमध्ये असे होऊ शकते की प्रोसिजर 1 मुख्य प्रोग्राममधून कॉल केली जाते, प्रोसिजर 2 प्रोसिजर 1 मधून कॉल केली जाते आणि प्रोसिजर 1 पुन्हा प्रोसिजर 2 वरून कॉल केली जाते. या परिस्थितीत प्रोग्राम प्रोसिजर 1 मध्ये एक्सीक्युशनचा प्रवाह पुन्हा प्रवेश करते. या प्रकारच्या प्रोसिजरला री-एण्ट्रन्ट प्रोसिजर (Re-entrant procedure) म्हणतात. री-एण्ट्रन्ट करण्याच्या प्रोसिजरसाठी प्रोग्राम एक्सीक्युशनचा प्रवाह Fig. 5.1 मध्ये दर्शविले आहे.

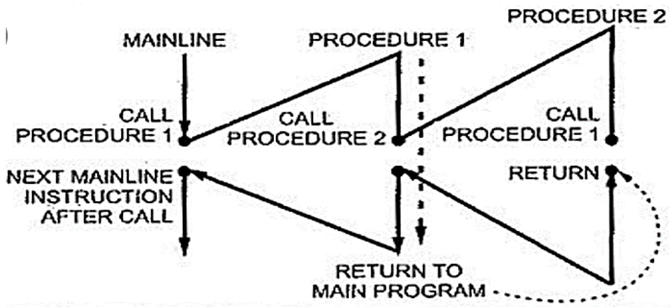


Fig. 5.1: री-एण्ट्रन्ट प्रोसिजर (Re-entrant Procedure)

5.1.5 रिकर्सिव्ह प्रोसिजर (Recursive Procedure)

रिकर्सिव्ह प्रोसिजर ही एकप्रोसिजर आहे जी स्वतःला कॉल करते. रिकर्सिव्ह प्रोसिजरचा वापर ट्री (Tree) नावाच्या जटिल डेटा स्ट्रक्चर्सवर कार्य करण्यासाठी केला जातो. जर प्रोसिजरला N (रिकरशन डेप्थ) = 3 सह कॉल केले असेल तर नंतर प्रत्येक प्रोसिजरच्या कॉलनंतर N एकाने कमी केले जाते आणि प्रोसिजरला N = 0 होई पर्यंत कॉल केली जाते. Fig. 5.2 रिकर्सिव्ह प्रोसिजरसाठी फ्लो डायग्राम आणि स्यूडो-कोड (pseudo-code) दर्शविते.

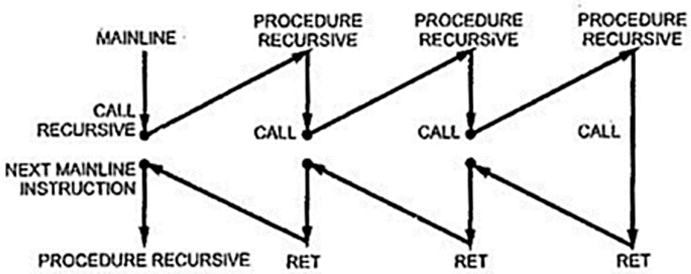


Fig. 5.2: रिकर्सिव्ह प्रोसिजर (Recursive Procedure)

5.1.6 प्रोसिजर वापरुन असेंब्ली लँग्वेजचे प्रोग्राम्स (Assembly language programs using procedure)

Program 1: बेरीज, वजाबाकी, गुणाकार आणि भागाकार साठी प्रोसिजर वापरुन असेंब्ली लँग्वेज प्रोग्राम. (An assembly language program using procedure to perform for addition, subtraction, multiplication and division.)

```
.model small
.data
        num1        dw  0FFFh
        num2        dw  FFh
        res_add     dw  ?   ; result of addition
        res_sub     dw  ?   ; result of subtraction
        res_mul     dd  ?   ; result of multiplication
        res_quo     dw  ?   ; Quotient of division
        res_rem     dw  ?   ; Remainder of division
.code
       mov ax,@data          ;Initialize data segment
       mov ds,ax
       call add_num          ;Call procedure for addition
```

```
        call sub_num          ;Call procedure for  subtraction
        call mul_num          ;Call procedure for multiplication
        call div_num          ;Call procedure for division
 add_num proc                 ; Procedure for addition
        mov ax,num1
        add ax,num2
        mov res_add,ax
        ret
        endp
sub_num proc                  ; Procedure for subtraction
        mov ax,num1
        sub ax,num2
        mov res_sub,ax
        ret
        endp
mul_num proc                  ; Procedure for multiplication
        mov ax,num1
        mul num2
        mov word ptr res_mul,ax
        mov word ptr res_mul+2,dx
        ret
        endp
div_num proc                  ; Procedure for division
        mov ax,num1
        cwd
        div num2
        mov res_quo,ax
        mov res_rem,dx
        ret
        endp
        ends
        end
```

Program 2: अरेमधून सर्वात लहान नंबर शोधण्यासाठी प्रोसिजर वापरून असेंब्ली लँग्वेज प्रोग्राम. (An assembly language program using procedure to find smallest number from the Array.)

```
.model small
        .data
                arr_s   dw      1,9,6,7,2
                sml_no   dw      0
        .code
                mov ax,@data            ;Initialize data segment
                mov ds,ax
                call small_num          ;call procedure to find smallest number
                mov sml_no, ax
        smallest proc
                mov cx,4                ; Initialize comparison counter
```

```
        mov si,offset arr_s     ;Initialize memory pointer
        mov ax,[si]             ;read number from array
        dec cx
up :    inc si
        inc si
        cmp ax,[si]             ;compare it with next number
        jc next                 ;if number is smallest then compare it
        mov ax,[si]             ;it next number
next:   loop up
        ret                     ;return to calling program
        endp
        ends
        end
```

Program 3: असेंब्ली लँग्वेज प्रोग्राम जसे की Z=(A+B)*(C+D) समीकरण सोडविण्यासाठी प्रोसिजर वापरणे.

(An assembly language program using procedure to solve equation such as Z = (A+B)*(C+D).)

```
model small
.data
    a       db      10H
    b       db      20H
    c       db      30H
    d       db      40H
    z       dw      ?
    sum     db      ?
.code
    mov ax,@data ; Initialize data segment
    mov ds,ax
    mov al,a        ; load al with a
    mov bl,b        ; load bl with b
    call add_byte   ; call procedure for addition
    mov asum,al     ; store result of addition
    mov al,c        ; load al with c
    mov bl,d        ; load bl with d
    call add_byte   ; call procedure for addition
    mul asum        ; multiply the result of 2 sum
    mov z,ax        ; store final result
add_byte        proc
    add al,bl
    ret
    endp
    ends
    end
```

5.2 मॅक्रोचा परिचय (Introduction to MACRO)

मॅक्रो (MACRO) हा एक युनिट आहे ज्यामध्ये गटबद्ध इंस्ट्रक्शन्सचा एक संच आहे. 8086 मायक्रोप्रोसेसरमध्ये मॉड्यूलर प्रोग्रामिंगची अंमलबजावणी करण्याची ही आणखी एक पद्धत आहे. मॅक्रो प्रोसिजरपेक्षा अशा प्रकारे भिन्न आहे की

प्रोसिजरप्रमाणे कॉल करणे आणि नियंत्रण परत करणे यासारखे नाही. जेव्हा जेव्हा मॅक्रोला कॉल केला जातो तेव्हा तेव्हा प्रोसेसर प्रत्येक वेळी प्रोग्राममध्ये कोड तयार करतो. असेंबलर डिरेक्टिव्हसचा वापर करून प्रोग्राममध्ये मॅक्रोची व्याख्या केली जाऊ शकते जसे कि **MACRO** (मॅक्रोच्या नावानंतर मॅक्रोचा मुख्य कोड सुरू करण्यापूर्वी वापरला जातो) आणि **ENDM** (मॅक्रोच्या शेवटी). मॅक्रोशी संबंधित असलेल्या सर्व इंस्ट्रक्शन्स या दोन असेंबलर डिरेक्टिव्हसमध्ये असायला पाहिजे.

5.2.1 मॅक्रो परिभाषित करणे (Define a MACRO)

8086 मायक्रोप्रोसेसरमध्ये मॅक्रो परिभाषित करण्यासाठी खालील सिंटॅक्स (Syntax) आहे:

```
        macro_name  MACRO  [ list of parameters ]
                        :
                Code of MACRO
                        :
                  ENDM
```

5.2.2 पॅरामीटर्ससह मॅक्रो (MACRO with parameters)

मॅक्रोमध्ये पॅरामीटर्स पास करणे पर्यायी आहे. जर आपण त्यांना आपल्या मॅक्रोमध्ये पास करू इच्छित असल्यास, डिरेक्टीव्ह MACRO नंतरच आपण मॅक्रोच्या अगदी पहिल्या विधानात त्या सर्वांचा उल्लेख करू शकतो. आपण पॅरामीटर्सला आपल्या मॅक्रोमध्ये पास करू इच्छित असल्यास, डिरेक्टीव्ह (MACRO) नंतरच आपण मॅक्रोच्या अगदी पहिल्या विधानात त्या सर्वांचा उल्लेख करू शकता जे खाली दिलेले आहे.

```
                add_num MACRO n1, n2, res
                            :
                     Code of MACRO
                            :
                        ENDM
```

वरील उदाहरणामध्ये n1, n2 आणि res हे मॅक्रोमध्ये पास केलेले पॅरामीटर्स आहेत.

5.2.3 मॅक्रो वापरुन असेंब्ली लँग्वेज प्रोग्राम्स (Assembly language programs using macro)

Program 1: बेरीज, वजाबाकी, गुणाकार आणि भागाकार साठी मॅक्रो वापरुन असेंब्ली लँग्वेज प्रोग्राम. (An assembly language program using macro to perform for addition, subtraction, multiplication and division.)

```
.model small
add_num macro n1, n2, r_add
        mov al, n1
        add al, n2
        mov r_add, al
        endm
sub_num macro n1, n2, r_sub
        mov al, n1
        sub al, n2
        mov r_sub, al
        endm
mul_num macro n1, n2, r_mul
        mov al, n1
        mul n2
        mov r_sub, ax
        endm
```

```
div_num macro n1, n2, q, r
        mov al, n1
        div n2
        mov q, al
        mov r, ah
        endm
.data
        rum1        db      45h
        rum2        db      10h
        res_add     db      ?
        res_sub     db      ?
        res_mul     dw      ?
        res_quo     db      ?
        res_rem     db      ?
.code
        mov ax, @data
        mov ds, ax
        add_num num1, num2, res_add
        sub_num num1, num2, res_sub
        mul_num num1, num2, res_mul
        div_num num1, num2, res_quo, res_rem
        ends
        end
```

Program 2: असेंब्ली लँग्वेज प्रोग्राम जसे की Z=(A+B)*(C+D) समीकरण सोडविण्यासाठी मॅक्रो वापरणे.

(An assembly language program using macro to solve equation such as Z = (A+B)*(C+D).)

```
.model small
add_num     macro  n1,n2, res
            mov al, n1
            add al, n2
            mov res, al
            endm
.data
            a       db      10H
            b       db      20H
            c       db      30H
            d       db      40H
            r1      db       ?
            r2      db       ?
            z       dw       ?
            sum     db       ?
.code
            mov ax,@data ;Initialize data segment
            mov ds,ax
            sum_num a,b,r1          ; Solve (a+b)
            sum_num c,d,r2          ; Solve (c+d)
```

```
mov al, r1
mul r2          ; Solve (a+b)*(c+d)
mov z, ax
ends
end
```

5.3 प्रोसिजर आणि मॅक्रोची तुलना (Compare procedures and macros)

Sr. No.	मॅक्रो (MACRO)	प्रोसिजर (PROCEDURE)
1	मॅक्रो परिभाषामध्ये इंस्ट्रक्शन्सचा एक संच आहे जे मॉड्यूलर प्रोग्रामिंगला समर्थन देते.	प्रोसिजरमध्ये इंस्ट्रक्शन्सचा एक संच असतो ज्यास विशिष्ट कार्य करण्यासाठी परत परत कॉल केली जाऊ शकते.
2	हे इंस्ट्रक्शन्सच्या छोट्या संचासाठी वापरले जाते.	हे इंस्ट्रक्शन्सच्या मोठ्या संचासाठी वापरले जाते
3	मेमरीची आवश्यकता जास्त असते.	मेमरीची आवश्यकता कमी असते
4	मॅक्रोमध्ये CALL आणि RET इंस्ट्रक्शन्स आवश्यक नाही.	प्रोसिजरमध्ये CALL आणि RET इंस्ट्रक्शन्स आवश्यक आहे.
5	असेंबलर डिरेक्टिव्ह MACROचा वापर मॅक्रो परिभाषित करण्यासाठी केला जातो आणि ENDMचा वापर कोड संपला आहे हे दर्शविण्यासाठी केला जातो.	असेंबलर डिरेक्टिव्ह PROCचा वापर प्रोसिजर परिभाषित करण्यासाठी केला जातो आणि ENDPचा वापर कोड संपला आहे हे दर्शविण्यासाठी केला जातो.
6	मॅक्रोची एक्सिक्युशनची वेळ कमी आहे कारण ती प्रोसिजरपेक्षा वेगवान एक्सिक्युट करते.	प्रोसिजरची एक्सिक्युशनची वेळ जास्त आहे कारण ती मॅक्रोपेक्षा हळू एक्सिक्युट करते.
7	येथे मशीन कोड अनेक वेळा तयार केला जातो कारण प्रत्येक वेळी मॅक्रो कॉल केला जातो तेव्हा मशीन कोड तयार होतो.	येथे मशीन कोड फक्त एकदाच तयार केला जातो, जेव्हा प्रोसिजर परिभाषित केली जाते तेव्हा ते फक्त एकदाच तयार होते.
8	मॅक्रोमध्ये पॅरामीटर स्टेटमेंटचा एक भाग म्हणून पास केले जाते ज्याला मॅक्रो पॅरामीटर म्हणतात.	प्रोसिजरमध्ये पॅरामीटर्स रेजिस्टर्स किंवा मेमरी लोकेशन्स किंवा स्टॅकमध्ये पास केले जातात.
9	कॉलिंग (CALL) आणि रिटर्निंग (RET) नसल्यामुळे वेळेचा ओव्हरहेड होत नाही.	कॉलिंग प्रोसिजर आणि कॉलिंग प्रोग्रामला नियंत्रण परत देताना वेळेचा ओव्हरहेड होतो.

References:

1. Microprocessor and Interfacing (Programming and Hardware) by Douglas V. Hall [McGraw Hill Education, New Delhi ISBN-13: 978-0070257429]
2. The 8088 and 8086 Microprocessors: Programming, Interfacing, Software, Hardware, and Applications by Walter A. Triebel, Avtar Singh [Pearson Publications, New Delhi ISBN-13: 978-0131228047]
3. Microprocessor 8086: Architecture, Programming and Interfacing by Sunil Mathur [PHI, New Delhi ISBN-13: 978-8120340879]
4. Microprocessor X86 Programming by K. R. Venugopal and Raj Kumar [BPB Publications, Delhi ISBN-13: 978-8170294580

HEAD OFFICE

Secretary,
Maharashtra State Board of Technical Education
49, Kherwadi, Bandra (East), Mumbai - 400 051
Maharashtra (INDIA)
Tel: (022)26471255 (5 -lines)
Fax: 022 - 26473980
Email: -secretary@msbte.com
Web -www.msbte.org.in

REGIONAL OFFICES:

MUMBAI

Deputy Secretary (T),
Mumbai Sub-region,
2nd Floor, Govt. Polytechnic Building,
49, Kherwadi, Bandra (East)
Mumbai - 400 051
Phone: 022-26473253 / 54
Email: rbtemumbai@msbte.com

PUNE

Deputy Secretary (T),
M.S. Board of Technical Education,
Regional Office,
412-E, Bahirat Patil Chowk,
Shivaji Nagar, Pune
Phone: 020-25656994 / 25660319
Fax: 020-25656994
Email: rbtepn@msbte.com

NAGPUR

Deputy Secretary (T),
M.S. Board of Technical Education
Regional Office,
Mangalwari Bazar, Sadar, Nagpur - 440 001
Phone: 0712-2564836 / 2562223
Fax: 0712-2560350
Email: rbteng@msbte.com

AURANGABAD

Deputy Secretary (T),
M.S. Board of Technical Education,
Regional Office,
Osmanpura, Aurangabad -431 001.
Phone: 0240-2334025 / 2331273
Fax: 0240-2349669
Email: rbteau@msbte.com