وزارة التعليم العالي والبحث العلمي
الجامعة التقنية الشمالية
معهد الادارة التقني - نينوى

معلومات عامة

الحقيبة التعليمية

القسم العلمي:	انظمة الحاسوب
اسم المقرر:	معمارية الحاسوب
المرحلة / المستوى:	المرحلة الثانية
الفصل الدراسي:	الثاني
السنة الدراسية:	2024-2023

اسم المقرر:	معمارية الحاسوب					
القسم:	انظمة الحاسوب					
الكلية:	معهد الادارة التقني - نينوى					
المرحلة / المستوى	الثاني					
الفصل الدراسي:	الثاني					
عدد الساعات الاسبوعية:	نظري	1	عملي	2		
عدد الوحدات الدراسية:	3					
الرمز:	CST209					
نوع المادة	نظري		عملي		كلهما	نعم
هل يتوفر نظير للمقرر في الاقسام الاخرى	لا يوجد					
اسم المقرر النظير						
القسم						
رمز المقرر النظير						

معلومات تدريسي المادة

اسم مدرس (مدرسي) المقرر:	استبرق بسام يحيى
اللقب العلمي:	مدرس مساعد
سنة الحصول على اللقب	2023
الشهادة :	ماجستير
سنة الحصول على الشهادة	2017
عدد سنوات الخبرة (تدريس)	1 سنة

الوصف العام للمقرر

المقرر يتناول موضوع معمارية الحاسوب والبرمجة بلغة التجميع، حيث يركز على كيفية تصميم وبناء الأنظمة الحاسوبية وتفاعلها مع الأجهزة والبرمجيات. يتم تقديم المفاهيم الأساسية لمعمارية الحاسوب ووحدات المعالجة المركزية والذاكرة والإدخال/الإخراج والتحكم في النظام. كما يتضمن المقرر دروسًا عملية لبرمجة الأنظمة باستخدام لغة التجميع وتنفيذ البرامج على المحاكيات. يهدف المقرر إلى تعزيز فهم الطلاب لكيفية عمل الحواسيب وتحسين مهاراتهم في تطوير البرمجيات والتفاعل مع الأجهزة الحاسوبية.

الاهداف العامة

يجب على الطلاب أن يتعلموا ويحققوا الأهداف الرئيسية العامة للمقرر وهي:

- سيتعلم الطلاب مفاهيم معمارية الحاسوب وأجزاء الحاسوب المختلفة مثل وحدة المعالجة المركزية والذاكرة.
- سيتمكن الطلاب من فهم كيفية عمل الذاكرة وأنواعها مثل الذاكرة العشوائية والقراءة فقط.
- سيكتسب الطلاب المعرفة حول كيفية تنفيذ البرامج باستخدام لغة التجميع والتفاعل مع الأجهزة الحاسوبية.
- سيكون بإمكان الطلاب تطبيق المفاهيم التي تعلموها في تصميم وتطوير برامج تعمل على الحواسيب.
- سيتمكن الطلاب من فهم أهمية إدارة الذاكرة والتحكم في النظام في سياق معمارية الحاسوب.
- سيتعلم الطلاب كيفية التعامل مع المقاطعات والمقاطعات البرمجية والأجهزة الحاسوبية بشكل عام.

الأهداف الخاصة

الأهداف الخاصة للمقرر تتضمن:

- تمكين الطلاب من فهم مفاهيم معمارية الحاسوب وتطبيقها على تصميم الأنظمة الحاسوبية.
- تعليم الطلاب كيفية استخدام لغة التجميع لتطوير برامج تفاعلية تتفاعل مع وحدات الحاسوب.
- توضيح أهمية إدارة الذاكرة والتحكم في النظام في سياق تطوير البرمجيات.
- تعزيز مهارات الطلاب في التعامل مع المقاطعات والمقاطعات البرمجية لتحقيق أداء فعال في برمجة الحواسيب.
- تحقيق القدرة على تحليل وفهم تفاصيل تنفيذ البرامج والتعامل مع الأجهزة الحاسوبية بشكل شامل.

الأهداف السلوكية او نواتج التعلم

الأهداف السلوكية أو نواتج التعلم لمقرر معمارية الحاسوب تشمل:

- تمكين الطلاب من تطبيق مفاهيم معمارية الحاسوب في حل مشاكل واقعية باستخدام لغة التجميع.

- تعزيز قدرة الطلاب على تحليل وتفسير أداء وحدات الحاسوب مثل وحدة المعالجة المركزية والذاكرة.
- تطوير مهارات الطلاب في استخدام تعليمات التحويل البيانات وتعليمات تعديل البتات لتحقيق أداء فعال.
- تحقيق القدرة على تصميم وتنفيذ برامج تفاعلية تستفيد من مفاهيم معمارية الحاسوب.
- تعزيز الفهم العميق لأنواع الذاكرة وأهميتها في تشغيل البرامج وتخزين البيانات.
- تحقيق القدرة على التعامل مع وحدات الإدخال والإخراج وفهم كيفية تبادل البيانات بين الحاسوب والأجهزة الخارجية.

المتطلبات السابقة

- المتطلبات التخصصية (29) وحدة (25) وحدة اجباري +(4) وحدة اختياري

الأهداف السلوكية او مخرجات التعليم الأساسية		
ت	تفصيل الهدف السلوكي او مخرج التعليم	آلية التقييم
1	فهم مفاهيم ومبادئ معمارية الحاسوب	اختبارات نهائية تحتوي على أسئلة نظرية لقياس فهم الطلاب للمفاهيم
2	تطوير مهارات استخدام لغة التجميع في تصميم وتنفيذ برامج	مشاريع عملية يتم تقديمها للطلاب لتطبيق مفاهيم معمارية الحاسوب وتقييم قدراتهم في تطبيقها
3	تحليل وتقييم أداء البرامج على مستوى النظام	تقييم أداء الطلاب في مهام عملية تتضمن استخدام لغة التجميع وتحليل أداء البرامج
4	تعزيز الفهم لأهمية إدارة الذاكرة والتحكم في النظام في سياق تطوير البرمجيات	استخدام تقييم مستمر خلال الفصل الدراسي لمتابعة تقدم الطلاب وضمان تحقيق الأهداف السلوكية المحددة
	تعزيز مهارات البرمجة وتحليل الأداء للبرامج على مستوى النظام	اختبارات عملية تحتوي على أسئلة تطبيقية لقياس قدرات الطلاب في تحليل وتنفيذ البرامج.

أساليب التدريس **(حدد مجموعة متنوعة من أساليب التدريس لتناسب احتياجات الطلاب ومحتوى المقرر)**

الاسلوب او الطريقة	مبررات الاختيار
1. دروس نظرية مباشرة	يمكن استخدام هذا الأسلوب لتقديم المفاهيم النظرية الأساسية لمعمارية الحاسوب. يمكن للطلاب فهم المفاهيم الأساسية والنظريات من خلال شرح مباشر وتوضيح العلاقات بين المفاهيم.
2. دروس عملية تطبيقية	تقديم دروس عملية تطبيقية تتيح للطلاب التفاعل مع أجهزة الحاسوب وتطبيق المفاهيم النظرية على أنشطة عملية
3. استخدام تقنيات التعلم النشط	يمكن تشجيع الطلاب على المشاركة في أنشطة تعليمية تفاعلية ويمكن استخدام حلقات النقاش لتشجيع الطلاب على تبادل الأفكار والآراء حول مواضيع معمارية الحاسوب. يمكن لهذا الأسلوب تعزيز التفاعل الاجتماعي وتحفيز التفكير النقدي
4. مشاريع عملية	يمكن تنظيم مشاريع عملية تتضمن تصميم وتنفيذ برامج على مستوى النظام لتعزيز فهم الطلاب وتطبيق المفاهيم النظرية في بيئة عملية

الفصل الاول من المحتوى العلمي							
عنوان الفصل	الوقت						
التوزيع الزمني	النظري	العملي	العنوان الفرعي		طريقة التدريس	التقنيات	طرق القياس
الأسبوع 1	1	2	Introduction to the course, learning objectives, course content		محاضرة	عرض تقديمي	
الأسبوع 2	1	2	Introduction	▪ What is computer architecture	محاضرة	عرض تقديمي، شرح، أسئلة وأجوبة, مناقشة	Quiz
				▪ Install Emu8086 ▪ Learn the basics of programming in assembly language and become familiar with the Emu8086 environment	تطبيق عملي	شرح، تنفيذ عملي باستخدام الحاسبة , أسئلة وأجوبة, مناقشة	القدرة على التعامل مع برنامج Emu8086
الاسبوع 3	1	2	Computer architecture	▪ The structure of computer architecture ▪ The structure of Central Processing Unit (CPU)	محاضرة	عرض تقديمي، شرح، أسئلة وأجوبة, مناقشة	Discussion
				▪ Use Emu8086 to understand program execution step by step, monitor memory and	تطبيق عملي	شرح، تنفيذ عملي باستخدام الحاسبة , أسئلة وأجوبة, مناقشة	Discussion

			record contents during and after execution				
Quiz	عرض تقديمي، شرح، أسئلة وأجوبة, مناقشة	محاضرة	▪ Microprocessor ▪ Evolution of Microprocessor ▪ Memory System ▪ Input/Output System ▪ System Bus	Computer architecture components	2	1	الاسبوع 4-5
Discussion	شرح، تنفيذ عملي باستخدام الحاسبة , أسئلة وأجوبة, مناقشة	تطبيق عملي	▪ Understand the basic role of the components of the 8086 processor architecture, general registers , index registers, pointer registers , segment registers, and flag register when executing instructions in assembly (part1).language				
Discussion	عرض تقديمي، شرح، أسئلة وأجوبة, مناقشة	محاضرة	▪ Microprocessor - Classification ▪ 8086 Microprocessor ▪ Features of 8086	8086 Microprocessor	2	1	الاسبوع 6

Discussion	شرح، تنفيذ عملي باستخدام الحاسبة , أسئلة وأجوبة, مناقشة	تطبيق عملي	▪ Understand the basic role of the components of the 8086 processor architecture, general registers , index registers, pointer registers , segment registers, and flag register when executing instructions in assembly language. (part2)				
Discussion	عرض تقديمي، شرح، أسئلة وأجوبة, مناقشة	محاضرة	▪ Addressing Modes types	Addressing Modes	2	1	الاسبوع 7
Quiz	شرح، تنفيذ عملي باستخدام الحاسبة , أسئلة وأجوبة, مناقشة	تطبيق عملي	▪ Examples about Addressing Modes types				
Discussion, Quiz, Home Exercises	عرض تقديمي، شرح، تنفيذ عملي باستخدام الحاسبة , أسئلة وأجوبة, مناقشة	محاضرة , تطبيق عملي	▪ Data Transfer Instructions ▪ Arithmetic Instructions ▪ Bit Manipulation Instructions ▪ String Instructions	8086 Instruction Sets	2	1	الاسبوع 8-11

			▪ Processor Control Instructions ▪ Iteration Control Instructions ▪ Program Execution Transfer Instructions (Branch & Loop Instructions)				
Discussion	عرض تقديمي، شرح , أسئلة وأجوبة, مناقشة	محاضرة	▪ 8086 Hardware Architecture ▪ 8086 Pin Diagram	8086 Hardware set	2	1	الاسبوع 12
Discussion, Quiz, Home Exercises	عرض تقديمي، شرح , حل رياضي, أسئلة وأجوبة, مناقشة	محاضرة , تطبيق عملي	▪ Segmentation ▪ Memory Addressing ▪ Practical Examples	8086 Memory Management	2	1	الاسبوع 13-14
Discussion, Home Exercises	عرض تقديمي، شرح , أسئلة وأجوبة, مناقشة	محاضرة , تطبيق عملي	▪ Interrupts types ▪ Interrupt Instructions	8086 Interrupts	2	1	الاسبوع 15

خارطة القياس المعتمدة

المحتوى التعليمي	عناوين الفصول	الأهمية النسبية (1-5)	الأهداف السلوكية					عدد الفقرات
			المعرفة	الفهم	التطبيق	التحليل	التقييم	
المحاضرة 1	Introduction to the course, learning objectives, course content	2	✓					**1**
المحاضرة 2	Introduction	3	✓	✓	✓			**3**
المحاضرة 3	Computer architecture	4	✓	✓	✓			**3**
المحاضرة 5-4	Computer architecture components	4	✓	✓	✓	✓	✓	**6**
المحاضرة 6	8086 Microprocessor	5	✓	✓	✓			**4**
المحاضرة 7	Addressing Modes	5	✓	✓	✓	✓	✓	**2**
المحاضرة 11-8	Instruction 8086 Sets	5	✓	✓	✓	✓	✓	**7**
المحاضرة 12	8086 Hardware set	3	✓	✓				**2**
المحاضرة 13-14	8086 Memory Management	5	✓	✓	✓	✓	✓	**3**
المحاضرة 15	8086 Interrupts	4	✓	✓	✓	✓		**2**
المجموع								

المحتويات (لكل فصل في المقرر)

رقم المحاضرة:	**1**
عنوان المحاضرة:	Introduction to the course, learning objectives, course content
اسـم المدرس:	استبرق بسام يحيى
الفئة المستهدفة :	طلبة المرحلة الثانية
الهدف العام من المحاضرة :	تعريف الطلاب بمحتوى المقرر وأهداف التعلم.
الأهداف السلوكية او مخرجات التعلم:	- فهم أهداف المقرر. - التعرف على المحتوى والمفاهيم الأساسية.
استراتيجيات التيسير المستخدمة	محاضرة ، مناقشة جماعية.
المهارات المكتسبة	مهارات الفهم العام والتنظيم.
طرق القياس المعتمدة	مناقشة لتقييم الفهم.
الاسئلة قبلية	1. ما هي أهمية الحواسيب في حياتنا اليومية؟ 2. كيف تؤثر معمارية الحاسوب على أداء النظام؟
الاسئلة البعدية	1. اشرح كيف تعمل البرمجيات والأجهزة معًا في معمارية الحاسوب. 2. ما هي العناصر الأساسية التي تشكل معمارية الحاسوب؟

رقم المحاضرة:	**2**
عنوان المحاضرة:	Introduction
اسـم المدرس:	استبرق بسام يحيى
الفئة المستهدفة :	طلبة المرحلة الثانية
الهدف العام من المحاضرة :	تعريف الطلاب بمعمارية الحاسوب وتثبيت بيئة Emu8086
الأهداف السلوكية او مخرجات التعلم:	- تعريف الطلاب بمعمارية الحاسوب. - تثبيت Emu8086 وفهم أساسيات البرمجة بلغة التجميع.
استراتيجيات التيسير المستخدمة	محاضرة تفاعلية ، تطبيق عملي.
المهارات المكتسبة	مهارات تثبيت البرمجيات، أساسيات البرمجة
طرق القياس المعتمدة	تقييم عملي لتثبيت البرمجيات.
الاسئلة قبلية	1. ما هو تعريف معمارية الحاسوب؟ 2. ما هي المكونات الأساسية لمعمارية الحاسوب؟
الاسئلة البعدية	1. اشرح الفرق بين معمارية مجموعة التعليمات (ISA) ومعمارية النظام (HSA). 2. كيف تؤثر معمارية الحاسوب على سرعة المعالجة وكفاءة الطاقة؟

رقم المحاضرة:	**3**
عنوان المحاضرة:	Computer architecture
اسـم المدرس:	استبرق بسام يحيى
الفئة المستهدفة :	طلبة المرحلة الثانية
الهدف العام من المحاضرة :	فهم هيكل معمارية الحاسوب و CPU
الأهداف السلوكية او مخرجات التعلم:	- التعرف على مكونات معمارية الحاسوب. - فهم هيكل CPU.
استراتيجيات التيسير المستخدمة	محاضرة تفاعلية ، تطبيق عملي.
المهارات المكتسبة	مهارات تثبيت البرمجيات، أساسيات البرمجة
طرق القياس المعتمدة	اختبارات قصيرة، تقييم عملي.
الاسئلة قبلية	1. ما هي معمارية فون نيومان؟ 2. ما هي معمارية هارفارد؟
الاسئلة البعدية	1. ما هي مزايا وعيوب كل من معمارية فون نيومان وهارفارد؟ 2. كيف تستخدم الحواسيب الحديثة مزيجًا من كلا المعمارين؟

رقم المحاضرة:	5-4
عنوان المحاضرة:	Computer architecture components
اسـم المدرس:	استبرق بسام يحيى
الفئة المستهدفة :	طلبة المرحلة الثانية
الهدف العام من المحاضرة :	فهم مكونات معمارية الحاسوب.
الأهداف السلوكية او مخرجات التعلم:	- التعرف على الميكروبروسيسور ونظام الذاكرة. - فهم دور مكونات معمارية 8086.
استراتيجيات التيسير المستخدمة	محاضرة تفاعلية ، مناقشة جماعية وتطبيق عملي.
المهارات المكتسبة	مهارات الفهم العميق للمكونات.
طرق القياس المعتمدة	اختبارات قصيرة، نشاطات عملية.
الاسئلة قبلية	1. ما هي الأجزاء الرئيسية التي تتكون منها معمارية الحاسوب؟ 2. كيف يؤثر تصميم وحدة المعالجة المركزية (CPU) على أداء النظام؟ 3. ما هي وظيفة وحدة المعالجة المركزية (CPU)؟ 4. ما هي الأجزاء الرئيسية لوحدة المعالجة المركزية؟
الاسئلة البعدية	1. اشرح دور نظام الإدخال والإخراج (I/O) في معمارية الحاسوب. 2. كيف يتم تنظيم الذاكرة في معمارية الحاسوب؟ 3. كيف تؤثر سرعة الساعة وهرمية الذاكرة على أداء وحدة المعالجة المركزية؟ 4. اشرح كيفية عمل وحدة التحكم ووحدة الحساب والمنطق (ALU). 5. اشرح الفرق بين ذاكرة القراءة فقط (ROM) وذاكرة الوصول العشوائي (RAM).

رقم المحاضرة:	6
عنوان المحاضرة:	8086 Microprocessor
اسـم المدرس:	استبرق بسام يحيى
الفئة المستهدفة :	طلبة المرحلة الثانية
الهدف العام من المحاضرة :	التعرف على معالج 8086 وميزاته.
الأهداف السلوكية او مخرجات التعلم:	- فهم تصنيف الميكروبروسيسور. - التعرف على ميزات 8086.
استراتيجيات التيسير المستخدمة	محاضرة تفاعلية ، مناقشة جماعية وتطبيق عملي.
المهارات المكتسبة	مهارات الفهم التقني.
طرق القياس المعتمدة	اختبارات قصيرة
الاسئلة قبلية	1. ما هو المعالج الدقيق وما هي مكوناته؟ 2. كيف يختلف المعالج الدقيق عن وحدة المعالجة المركزية التقليدية؟ 3. ما هي بعض المعالجات الدقيقة الشهيرة التي تعرفها؟ 4. كيف تطورت المعالجات الدقيقة عبر الزمن؟
الاسئلة البعدية	1. اشرح خطوات عمل المعالج الدقيق في تنفيذ التعليمات. 2. ما هي فوائد استخدام المعالجات الدقيقة في الأجهزة الحديثة؟ 3. ما هي مميزات المعالجات الحديثة مقارنة بالمعالجات القديمة؟ 4. كيف تؤثر عدد الترانزستورات في المعالج على أدائه؟

رقم المحاضرة:	7
عنوان المحاضرة:	Addressing Modes
اسـم المدرس:	استبرق بسام يحيى
الفئة المستهدفة :	طلبة المرحلة الثانية
الهدف العام من المحاضرة :	فهم أنماط العنوان في 8086.
الأهداف السلوكية او مخرجات التعلم:	- التعرف على أنواع أنماط العنوان. - تطبيق أمثلة على أنماط العنوان.
استراتيجيات التيسير المستخدمة	محاضرة تفاعلية ، أمثلة وتطبيق عملي.
المهارات المكتسبة	مهارات تحليل العناوين.
طرق القياس المعتمدة	اختبارات قصيرة، تقييم عملي
الاسئلة قبلية	1. ما هي أهمية وضع العنوان في تنفيذ التعليمات في المعالج؟ 2. كيف يمكن أن تؤثر أوضاع العنوان على طريقة الوصول إلى البيانات في الذاكرة؟
الاسئلة البعدية	1. اشرح كيف يمكن استخدام أوضاع العنوان المختلفة لتحسين كفاءة البرمجة في 8086. 2. ما هي التحديات التي قد تواجهها عند استخدام أوضاع العنوان في البرمجة بلغة التجميع؟

رقم المحاضرة:	11-8
عنوان المحاضرة:	8086 Instruction Sets
اسـم المدرس:	استبرق بسام يحيى
الفئة المستهدفة :	طلبة المرحلة الثانية
الهدف العام من المحاضرة :	فهم مجموعة تعليمات 8086.
الأهداف السلوكية او مخرجات التعلم:	- التعرف على تعليمات نقل البيانات، التعليمات الرياضية، التعليمات المنطقية, وتعليمات التحكم.
استراتيجيات التيسير المستخدمة	محاضرة تفاعلية ، أمثلة وتطبيق عملي.
المهارات المكتسبة	مهارات البرمجة والتعامل مع مجموعة التعليمات لتنفيذ مهام محددة
طرق القياس المعتمدة	اختبارات قصيرة، تقييم عملي، واجبات لاصفية
الاسئلة قبلية	1. ما هي العلاقة بين التعليمات البرمجية والعمليات التي ينفذها المعالج؟ 2. كيف يمكن تصنيف التعليمات في معمارية الحاسوب؟
الاسئلة البعدية	1. اشرح كيفية تأثير مجموعة التعليمات على تصميم البرمجيات في معمارية 8086. 2. ما هي التعليمات الأساسية التي تدعمها مجموعة تعليمات 8086 وكيف يتم استخدامها في البرمجة؟

رقم المحاضرة:	12
عنوان المحاضرة:	8086 Hardware Architecture
اسـم المدرس:	استبرق بسام يحيى
الفئة المستهدفة :	طلبة المرحلة الثانية
الهدف العام من المحاضرة :	فهم بنية معالج 8086.
الأهداف السلوكية او مخرجات التعلم:	- التعرف على مكونات معالج 8086. - فهم كيفية عمل المعالج.
استراتيجيات التيسير المستخدمة	محاضرة تفاعلية.
المهارات المكتسبة	مهارات تحليل بنية المعالج.
طرق القياس المعتمدة	اختبارات قصيرة، نقاشات جماعية.
الاسئلة قبلية	1. ما هي الوظائف الأساسية لوحدة المعالجة المركزية (CPU) في نظام الحاسوب؟

	2. ما هي الاجزاء الاساسية للكيان المادي المكون للمعالج 8086؟
الاسئلة البعدية	1. ماذا نقصد بالوصف Dual Inline package للمعالج 8086؟ 2. كيف طور عمل Mode يستطيع المعالج 8086 العمل به؟

رقم المحاضرة:	**14-13**
عنوان المحاضرة:	8086 Memory Management
اسـم المدرس:	استبرق بسام يحيى
الفئة المستهدفة :	طلبة المرحلة الثانية
الهدف العام من المحاضرة :	فهم إدارة الذاكرة في معالج 8086.
الأهداف السلوكية او مخرجات التعلم:	- التعرف على نماذج إدارة الذاكرة. - فهم كيفية تقسيم الذاكرة في 8086. - فهم كيفية استخدام سجلات الذاكرة. - التعرف على تقنيات تحسين أداء الذاكرة.
استراتيجيات التيسير المستخدمة	محاضرة تفاعلية، تطبيقات عملية.
المهارات المكتسبة	مهارات إدارة الذاكرة.
طرق القياس المعتمدة	اختبارات قصيرة، تقييم عملي.
الاسئلة قبلية	1. ما هي إدارة الذاكرة ولماذا هي مهمة في الحاسوب؟ 2. كيف يتم تقسيم الذاكرة في معالج 8086؟
الاسئلة البعدية	1. اشرح كيفية استخدام سجلات الذاكرة في معالج 8086. 2. ما هي فوائد التجزئة في تحسين أداء النظام؟

رقم المحاضرة:	**15**
عنوان المحاضرة:	8086 Interrupts
اسـم المدرس:	استبرق بسام يحيى
الفئة المستهدفة :	طلبة المرحلة الثانية
الهدف العام من المحاضرة :	فهم المقاطعات في معالج 8086.
الأهداف السلوكية او مخرجات التعلم:	- التعرف على أنواع المقاطعات. - فهم كيفية التعامل مع المقاطعات في 8086.
استراتيجيات التيسير المستخدمة	محاضرة تفاعلية، تطبيقات عملية.
المهارات المكتسبة	مهارات التعامل مع المقاطعات.
طرق القياس المعتمدة	اختبارات قصيرة، تطبيقات عملي.
الاسئلة قبلية	1. ما هي المقاطعة في نظام الحاسوب؟ 2. كيف يتم التعامل مع المقاطعات في المعالج؟
الاسئلة البعدية	1. اشرح الفرق بين المقاطعات القابلة للتجاهل والمقاطعات غير القابلة للتجاهل. 2. كيف تؤثر المقاطعات على أداء النظام؟

المصادر الاساسية :

- Rector, Russell, and George Alexy. The 8086 Book. Osborne/McGraw-Hill, 1980. ISBN-13: 978-0079310293.
- Hall, Douglas V. Microprocessors and Interfacing: Programming and Hardware. 2nd ed., Tata McGraw Hill, 1992. ISBN-13: 978-0070257429.

المصادر الاضافية المقترحة:

https://www.tutorialspoint.com/microprocessor/microprocessor_8086_overview.htm

روابط مقترحة ذات صلة: **لايوجد**

المحتوى العلمي

Computer Architecture
معمارية الحاسوب

Part 1

Estabrak Bassam Yahya

Introduction

Computers have become an important part of our lives in this modern world. These high-tech machines have an effect on every part of society, from healthcare to entertainment to interacting with one another.

But the science of computer architecture, which is what makes modern computers work, is what lies behind the modern screens and easy-to-use interfaces.

What is computer architecture?

❑The architecture of a computer tells you how the software and hardware work together to provide services to the user.

❑On the other word, computer architecture decides how computers are put together and how they work, as well as what technologies they can work with. This includes the central processing unit (CPU), memory, input/output devices, and storage units.

❑It is very important to understand what computer architecture means because it helps you come up with new and useful ways to use computers.

What is computer architecture?

❑Understanding the meaning of computer architecture is crucial, as it forms the basis for designing innovative and efficient computing solutions. These design decisions can have a huge influence on factors like a **computer's processing speed**, **energy efficiency**, **and overall system performance**.

❑There are two essential parts of computer architecture:

1- Instruction Set Architecture (ISA)

2- Hardware System Architecture (HSA)

What is computer architecture?

❑Instruction Set Architecture (**ISA**): This is any software that makes a computer run, including the CPU's functions and capabilities, programming languages, data formats, processor register types, and instructions used by programmers. A computer is generally viewed in terms of its ISA, which determines the computational characteristics of the computer.

❑Hardware System Architecture (**HSA**): Deals with the computer's major hardware subsystems, including its central processing unit (CPU), its storage system and its input-output system (I/O) (which is the computer's interface to the world). The HSA includes both the logical design and the data flow organization of these subsystems, HSA determines how efficiently the machine will operate.

Types of computer architecture

Despite the rapid advancement of computing, many of the fundamentals of computer architecture remain the same. There are two main types of computer architecture:

❑Von Neumann architecture - Named after mathematician and computer scientist John von Neumann, this features a single memory space for both data and instructions, which are fetched and executed sequentially. Von Neumann architecture introduced the concept of stored-program computers, where both instructions and data are stored in the same memory, allowing for flexible program execution.

❑Harvard architecture - This, on the other hand, uses separate memory spaces for data and instructions, allowing for parallel fetching and execution.

Types of computer architecture

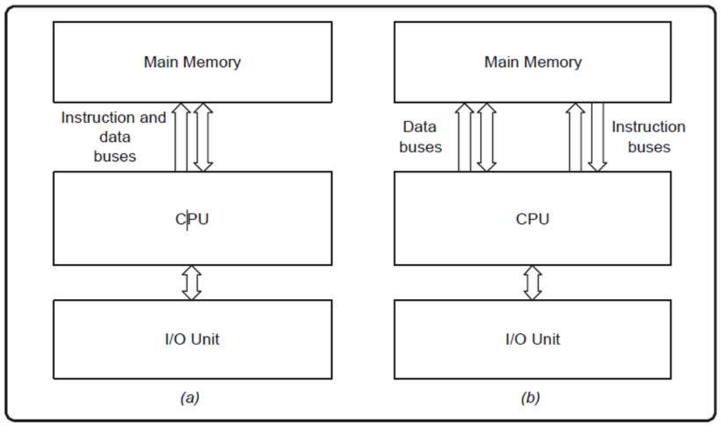


Diagram showing Von Neumann architecture (a) and Harvard architecture (b)

Both types of architecture have their own advantages and trade-offs, and modern computers often use a combination of both to get the best system performance.

The structure of computer architecture

Computer architectures can be very different based on what the computer is used for, but there are a few main parts that make up most computer architectures:

- ❑Central Processing Unit (CPU)
- ❑The Structure of Memory
- ❑Input/Output (I/O) System
- ❑System Bus

The structure of computer architecture

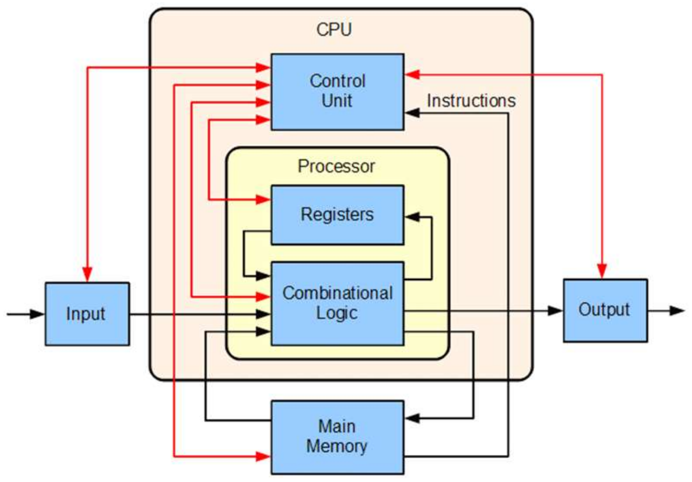


Diagram depicting the structure of basic computer architecture with a uniprocessor CPU.

The structure of Central Processing Unit (CPU)

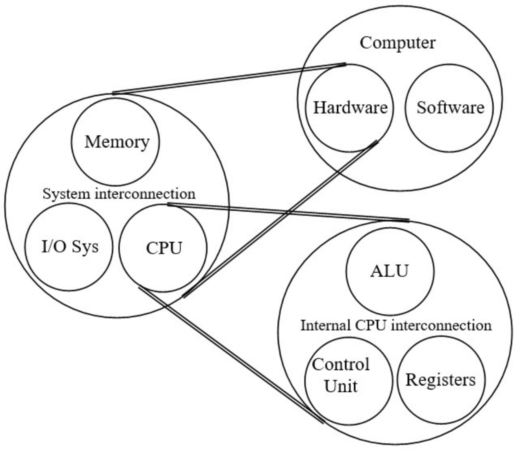
❑**Central Processing Unit (CPU)**

The CPU runs programs, does math, and keeps track of data. It is often called the "brain" of the computer. Its design controls things like the instruction set, clock speed, and cache hierarchy, all of which have a big effect on how well the system works as a whole.

The main parts of the CPU:

1. Control Unit: which controls the operation of the computer.
2. Arithmetic & Logic Unit (ALU): which performs arithmetic, logical and shift operations to produce results.
3. Register Set: which holds various values during the computer's operations.
4. Program Counter (PC) (Instruction Pointer IP): which holds the main memory address of an instruction.

The structure of Central Processing Unit (CPU)



Memory System

❑<u>The Structure of Memory</u>

shows the different kinds of memory, like cache memory, random access memory (RAM), and recording devices. The structure of memory is very important for accessing data quickly because data goes between levels of memory based on how close they are to the CPU and how often they are accessed.

The memory of a computer can be divided into three main groups:

1. <u>Internal processor memory:</u> This represent a small set of high-speed registers used as working memory for temporary storage of instructions and data.
2. <u>Main memory (Also called Primary memory):</u> This is a relatively large fast memory used for program and data storage during computer operation. It is characterized by the fact that locations in main memory can be accessed directly and rapidly by the CPU instruction set.

Memory System

3. Secondary memory (Also called Auxiliary or Backing memory): This is generally much large in capacity but also much slower than main memory. It is used for storing system programs and large data files. This type of memory has the following groups:

a- Magnetic tape	b- Floppy disk
c- Hard disk	d- CD-Rom (Compact Disk ROM)

Memory System

Memory Device Characteristics:

✓ **Cost:**

The cost of a memory device is a critical characteristic influencing its affordability and suitability for specific applications. It is essential to consider the balance between cost and the desired storage capacity or performance.

✓ **Access Time:**

Access time is a crucial metric determining how quickly a memory device can provide requested data. It represents the time interval between issuing a read or write request and the availability of the data. Faster access times contribute to enhanced system performance.

Memory System

Memory Types:

- ✓ Read Only Memory (ROM)
- ✓ Random Access Memories (RAM)

Read Only Memory (ROM) Types:

ROM: Read Only Memory is a non volatile device that the CPU can read but cannot write. Computers use them for holding constants that specify the system's configuration. Many ROMs are factory programmed, and there is no way to alter their contents.

PROM: Field engineers can program this type of ROM memory by using special high-current device to destroy (burn) fuses that were manufactured into the devices. The result of burning a PROM is that certain bits are always 0s and the rest are always 1s. These values cannot be altered once written.

Memory System

EPROM: This type of ROM can be erased by ultraviolet light and reprogrammed many times. The components in the memory matrix of the EPROM complex electronic devices, these devices act like diodes that can be turned on or off by the presence or absence of minute amounts of electrical charge.

EEPROM: It uses components that are some what similar to those in the EPROM. However, the components in the EEPROM can be disconnected (thus erasing the memory) electrically rather than by exposure to ultraviolet light.

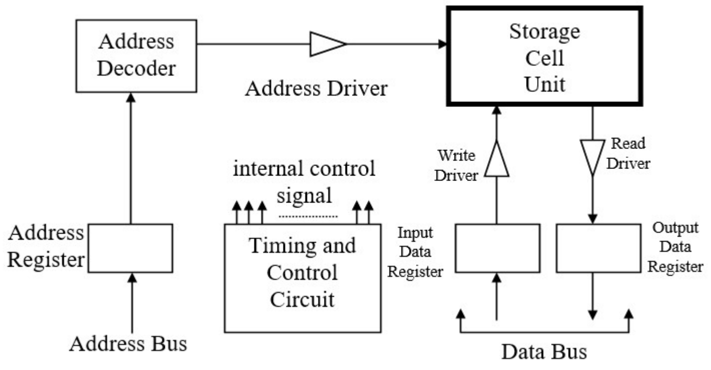
Memory System

Random Access Memories (RAM)

RAMs are characterized by the fact that every location can be accessed independently. The access and cycle times for every location are constant and independent of its position. RAM is a memory device that the CPU can read and write. Both the reading and writing are accomplished through the use of electrical signals. RAM is a volatile which mean that it lose their information content whenever the power to the system is turned off. Thus RAM can be used only as temporary storage.

Memory System

Main Components of a RAM Unit



The figure above shows the main components of a RAM unit. The storage cell unit comprises N cells, each of which can store 1 bit of information.

Memory System

The RAM operates as follow: The address of the required location is transferred via the address bus to the memory address register, the address is then processed by the address decoder which select the required location in the storage cell unit. A read-write select control line specifies the type of access to be performed. If read is requested, the contents of the selected location is transferred to the output data register. If write is requested, the word to be written is first placed in the memory input data register and then transferred to the selected cell.

Input/Output System

❑ **Input/Output (I/O) System**

The I/O system facilitates communication between the computer and external devices, encompassing keyboards, monitors, and storage devices. It is responsible for the meticulous design of efficient data transfer mechanisms, ensuring seamless interaction and data exchange.

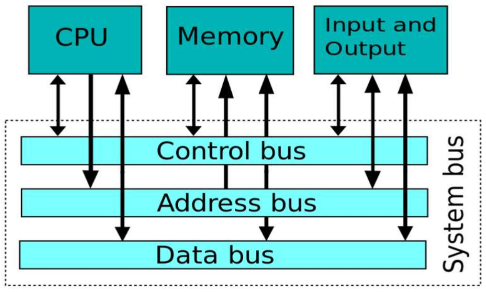
✓Inputs refer to signals or data entering the system, while outputs denote those leaving it. In a standard personal computer setup, peripheral devices include input tools like keyboards and mice, alongside output devices such as displays and printers.

✓Furthermore, certain devices, like hard drives, floppy drives, and optical drives, exhibit dual functionality as both input and output devices. They not only read data from external sources but also write data back, emphasizing their versatility in the I/O system.

System Bus

❑**System Bus Types**

All of the previous mentioned parts of computer architecture are connected through a system bus consisting of the address bus, data bus and control bus.



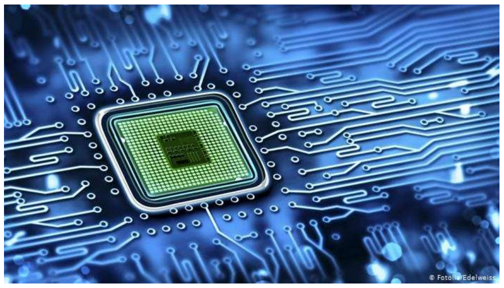
System Bus

Address Bus: Comprising a set of wires or lines, the address bus facilitates the transportation of addresses for memory or input/output devices. This unidirectional bus specifies the memory location where the processor reads or writes data. The number of memory locations is determined by 2 to the power of N address lines. For instance, a CPU with 16 address lines can address 2^16 or 65,536 memory locations.

Data Bus: As implied by its name, the data bus is responsible for bidirectional data transfer within microprocessors and between memory/input or output devices. This two-way communication enables the microprocessor to send or receive data as required.

Control Bus: The control bus orchestrates the information flow among components, indicating whether the operation is a read or write and ensuring the verification process occurs at the appropriate time.

Thanks



Computer Architecture
معمارية الحاسوب

Part 2

Estabrak Bassam Yahya

Microprocessor

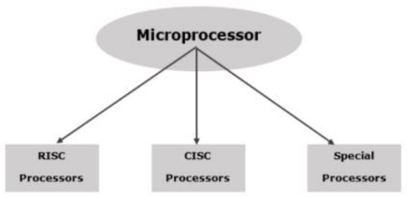
A Microprocessor can be classified into three categories:

❑**Reduced Instruction Set Computer (RISC)**

RISC processors execute a small, optimized set of instructions for simplicity and efficiency. Their instruction sets are smaller, thus each instruction executes faster. ARM processors and MIPS processors are RISC architectures

❑**Complex Instruction Set Computer (CISC)**

CISC processors support many complicated instructions, frequently completing many operations in one instruction. These processors decrease task instructions. CISC architectures include Intel and AMD x86 CPUs.



Microprocessor - Classification

❑Special Processors

Special processors refer to microprocessors designed for specific tasks or applications, such as digital signal processors (DSPs), graphics processing units (GPUs), and application-specific integrated circuits (ASICs). These processors are optimized for particular types of computations or tasks, offering high performance and efficiency within their specialized domains.

8086 Microprocessor

8086 Microprocessor was designed by Intel in 1976. It is a 16-bit Microprocessor having 20 address lines and 16 data lines that provides up to 1MB storage. It consists of powerful instruction set, which provides operations like multiplication and division easily.

It supports two modes of operation, i.e. Maximum mode and Minimum mode. Maximum mode is suitable for system having multiple processors and Minimum mode is suitable for system having a single processor.

Features of 8086

The most prominent features of a 8086 microprocessor are as follows –

- ✓8086 employs parallel processing.
- ✓It consists of 29,000 transistors
- ✓8086 CPU has two parts which operate at the same time
  - Bus Interface Unit
  - Execution Unit
- ✓It has an instruction queue, which is capable of storing six instruction bytes from the memory resulting in faster processing.
- ✓It was the first 16-bit processor having 16-bit ALU, 16-bit registers, internal data bus, and 16-bit external data bus resulting in faster processing.
- ✓It is available in 3 versions based on the frequency of operation (5MHz, 8MHz, and 10 MHz)

8086 Instruction Sets

The 8086 microprocessor supports 8 types of instructions –

- ✓Data Transfer Instructions
- ✓Arithmetic Instructions
- ✓Bit Manipulation Instructions
- ✓String Instructions
- ✓Processor Control Instructions
- ✓Iteration Control Instructions
- ✓Interrupt Instructions
- ✓Program Execution Transfer Instructions (Branch & Loop Instructions)

❑**8086 Microprocessor General Instructions Format**

The 8086 microprocessor's general instructions format encompasses variable length instructions, typically ranging from 1 to 6 bytes. Instructions start with optional prefix bytes followed by an opcode indicating the operation. The format accommodates various addressing modes and operands, allowing flexibility in memory and register access.

8086 Instruction Sets

1.Instruction (Opcode) - with No Operand
- Example: **NOP** (No Operation)
- Description: **NOP** is a common instruction found in many assembly languages. It performs no operation and is used for various purposes, such as inserting delays or placeholders in code.
- Syntax: **NOP**

2.Instruction (Opcode) - with One Operand
- Example: **INC** (Increment)
- Description: **INC** is used to increment the value of the operand by one.
- Syntax: **INC operand**
- Example: **INC AX** increments the value in register AX by one.

3.Instruction (Opcode) - with Two Operands
- Example: **MOV** (Move)
- Description: **MOV** is used to move data from one location to another.
- Syntax: **MOV destination, source**
- Example: **MOV AX, BX** moves the content of register BX to register AX.

8086 Instruction Sets

❑**Addressing Modes**

The different ways in which a source operand is denoted in an instruction is known as addressing modes. There are 8 different addressing modes in 8086 programming

➢**Immediate addressing mode**

The addressing mode in which the data operand is a part of the instruction itself is known as immediate addressing mode.

```
MOV CX, 4929 H, ADD AX, 2387 H,  MOV AL, FFH
```

➢**Register addressing mode**

It means that the register is the source of an operand for an instruction.

```
MOV CX, AX   ; copies the contents of the 16-bit AX register into
             ; the 16-bit CX register),
ADD BX, AX
```

8086 Instruction Sets

➢**Direct addressing mode**

The addressing mode in which the effective address of the memory location is written directly in the instruction.

```
MOV AX, [1592H], MOV AL, [0300H]
```

➢**Register indirect addressing mode**

This addressing mode allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI & SI.

```
MOV AX, [BX]  ; Suppose the register BX contains 4895H, then the contents
              ; 4895H are moved to AX
```

8086 Instruction Sets

➢**Based addressing mode**

In this addressing mode, the offset address of the operand is given by the sum of contents of the BX/BP registers and 8-bit/16-bit displacement.

```
MOV DX, [BX+04], ADD CL, [BX+08]
```

➢**Indexed addressing mode**

In this addressing mode, the operands offset address is found by adding the contents of SI or DI register and 8-bit/16-bit displacements.

```
MOV BX, [SI+16], ADD AL, [DI+16]
```

8086 Instruction Sets

➢**Based-index addressing mode**

In this addressing mode, the offset address of the operand is computed by summing the base register to the contents of an Index register.

```
ADD CX, [AX+SI], MOV AX, [AX+DI]
```

➢**Based indexed with displacement mode**

In this addressing mode, the operands offset is computed by adding the base register contents. An Index registers contents and 8 or 16-bit displacement.

```
MOV AX, [BX+DI+08], ADD CX, [BX+SI+16]
```

8086 Instruction Sets

❑**Data Transfer Instructions**

These instructions are used to transfer the data from the source operand to the destination operand. Following are the list of instructions under this group

✓**Instruction to transfer a word – 16 bit**

- **MOV** – Used to copy the byte or word from the provided source to the provided destination.
- **PPUSH** – Used to put a word at the top of the stack.
- **POP** – Used to get a word from the top of the stack to the provided location.
- **PUSHA** – Used to put all the registers into the stack.
- **POPA** – Used to get words from the stack to all registers.
- **XCHG** – Used to exchange the data from two locations.
- **XLAT** – Used to translate a byte in AL using a table in the memory.

8086 Instruction Sets

✓Instructions for input and output port transfer

- **IN** – Used to read a byte or word from the provided port to the accumulator.
- **OUT** – Used to send out a byte or word from the accumulator to the provided port.

✓Instructions to transfer the address

- **LEA** – Used to load the address of operand into the provided register.
- **LDS** – Used to load DS register and other provided register from the memory
- **LES** – Used to load ES register and other provided register from the memory.

8086 Instruction Sets

✓Instructions to transfer flag registers

- **LAHF** – Used to load AH with the low byte of the flag register.
- **SAHF** – Used to store AH register to low byte of the flag register.
- **PUSHF** – Used to copy the flag register at the top of the stack.
- **POPF** – Used to copy a word at the top of the stack to the flag register.

8086 Instruction Sets

❑**Arithmetic Instructions**

These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc. Following are the list of instructions under this group

✓**Instructions to perform addition**

- **ADD** – Used to add the provided byte to byte/word to word.
- **ADC** – Used to add with carry.
- **INC** – Used to increment the provided byte/word by 1.
- **AAA** – Used to adjust ASCII after addition.
- **DAA** – Used to adjust the decimal after the addition/subtraction operation.

8086 Instruction Sets

✓**Instructions to perform subtraction**

- **SUB** – Used to subtract the byte from byte/word from word.
- **SBB** – Used to perform subtraction with borrow.
- **DEC** – Used to decrement the provided byte/word by 1.
- **NPG** – Used to negate each bit of the provided byte/word and add 1/2's complement.
- **CMP** – Used to compare 2 provided byte/word.
- **AAS** – Used to adjust ASCII codes after subtraction.
- **DAS** – Used to adjust decimal after subtraction.

✓**Instruction to perform multiplication**

- **MUL** – Used to multiply unsigned byte by byte/word by word.
- **IMUL** – Used to multiply signed byte by byte/word by word.
- **AAM** – Used to adjust ASCII codes after multiplication.

8086 Instruction Sets

✓**Instructions to perform division**

- **DIV** – Used to divide the unsigned word by byte or unsigned double word by word.
- **IDIV** – Used to divide the signed word by byte or signed double word by word.
- **AAD** – Used to adjust ASCII codes after division.
- **CBW** – Used to fill the upper byte of the word with the copies of sign bit of the lower byte.
- **CWD** – Used to fill the upper word of the double word with the sign bit of the lower word.

8086 Instruction Sets

❑**Bit Manipulation Instructions**

These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc. Following is the list of instructions under this group –

✓**Instructions to perform logical operation**

- **NOT** – Used to invert each bit of a byte or word.
- **AND** – Used for adding each bit in a byte/word with the corresponding bit in another byte/word.
- **OR** – Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
- **XOR** – Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.
- **TEST** – Used to add operands to update flags, without affecting operands.

8086 Instruction Sets

✓Instructions to perform shift operations

- **SHL/SAL** – Used to shift bits of a byte/word towards left and put zero(S) in LSBs.
- **SHR** – Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.
- **SAR** – Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

✓Instructions to perform rotate operations

- **ROL** – Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].
- **ROR** – Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].
- **RCR** – Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.
- **RCL** – Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.

8086 Instruction Sets

❑String Instructions

String is a group of bytes/words and their memory is always allocated in a sequential order. Following is the list of instructions under this group –

- **REP** – Used to repeat the given instruction till CX ≠ 0.
- **REPE/REPZ** – Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **REPNE/REPNZ** – Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **MOVS/MOVSB/MOVSW** – Used to move the byte/word from one string to another.
- **COMS/COMPSB/COMPSW** – Used to compare two string bytes/words.
- **INS/INSB/INSW** – Used as an input string/byte/word from the I/O port to the provided memory location.
- **OUTS/OUTSB/OUTSW** – Used as an output string/byte/word from the provided memory location to the I/O port.
- **SCAS/SCASB/SCASW** – Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.
- **LODS/LODSB/LODSW** – Used to store the string byte into AL or string word into AX.

8086 Instruction Sets

❑Processor Control Instructions

These instructions are used to control the processor action by setting/resetting the flag values. Following is the list of instructions under this group –

- **STC** – Used to set carry flag CF to 1
- **CLC** – Used to clear/reset carry flag CF to 0
- **CMC** – Used to put complement at the state of carry flag CF.
- **STD** – Used to set the direction flag DF to 1
- **CLD** – Used to clear/reset the direction flag DF to 0
- **STI** – Used to set the interrupt enable flag to 1, i.e., enable INTR input.
- **CLI** – Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

8086 Instruction Sets

❑Iteration Control Instructions

These instructions are used to execute the given instructions for number of times. Following is the list of instructions under this group –

- **LOOP** – Used to loop a group of instructions until the condition satisfies, i.e., CX = 0
- **LOOPE/LOOPZ** – Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0
- **LOOPNE/LOOPNZ** – Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0
- **JCXZ** – Used to jump to the provided address if CX = 0

8086 Instruction Sets

❑**Interrupt Instructions**

These instructions are used to call the interrupt during program execution. Following is the list of instructions under this group –

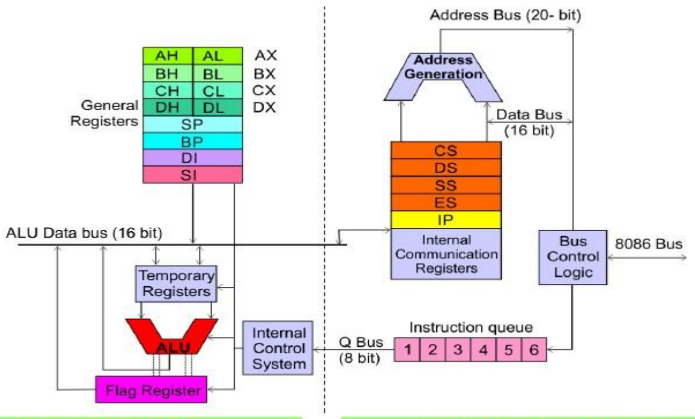
- **INT** – Used to interrupt the program during execution and calling service specified.
- **INTO** – Used to interrupt the program during execution if OF = 1
- **IRET** – Used to return from interrupt service to the main program

❑**Program Execution Transfer Instructions (Branch and Loop Instructions)**

These instructions are used to transfer/branch the instructions during an execution either with or without certain conditions. Examples : CALL , JC, … etc.

- **CALL** – Used to call a procedure and save their return address to the stack.
- **JC** – Used to jump if carry flag CF = 1

8086 Hardware Architecture



Execution Unit (EU)

EU executes instructions that have already been fetched by the BIU.

BIU and EU functions separately.

Bus Interface Unit (BIU)

BIU fetches instructions, reads data from memory and I/O ports, writes data to memory and I/ O ports.

8086 Hardware Architecture

EU decodes and executes instructions.

A decoder in the EU control system translates instructions.

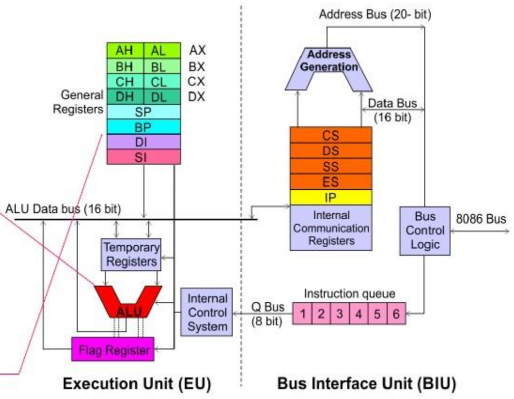
16-bit ALU for performing arithmetic and logic operation

Four general purpose registers(AX, BX, CX, DX);

Pointer registers (Stack Pointer, Base Pointer);

and

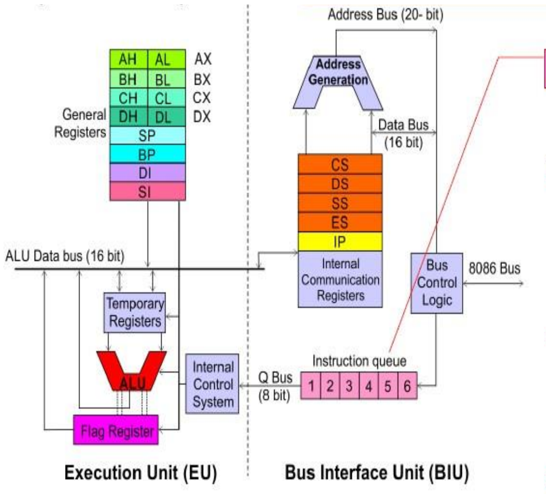
Index registers (Source Index, Destination Index) each of 16-bits



Some of the 16 bit registers can be used as two 8 bit registers as :

AX can be used as AH and AL
BX can be used as BH and BL
CX can be used as CH and CL
DX can be used as DH and DL

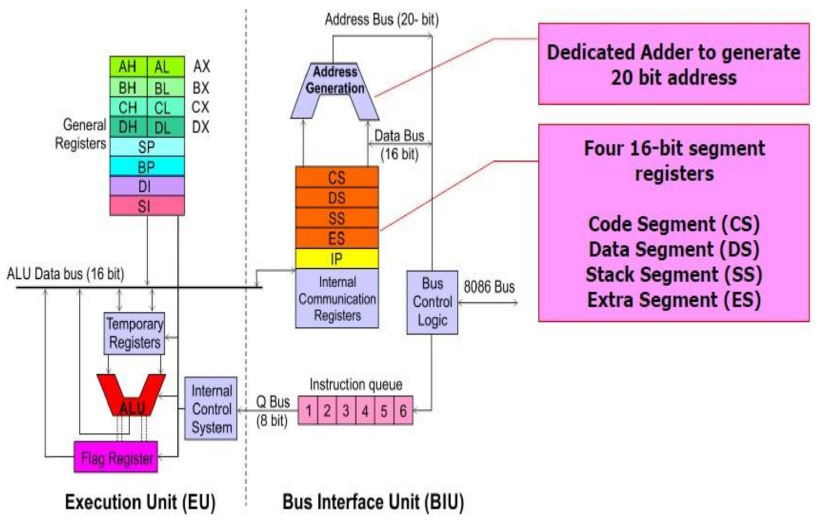
8086 Hardware Architecture



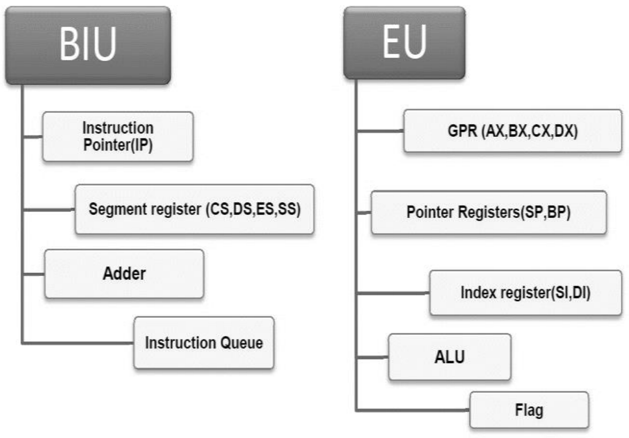
Instruction queue

- A group of First-In-First-Out (FIFO) in which up to 6 bytes of instruction code are pre fetched from the memory ahead of time.
- This is done in order to speed up the execution by overlapping instruction fetch with execution.
- This mechanism is known as <u>pipelining</u>.

8086 Hardware Architecture



8086 Hardware Architecture



8086 Memory Management

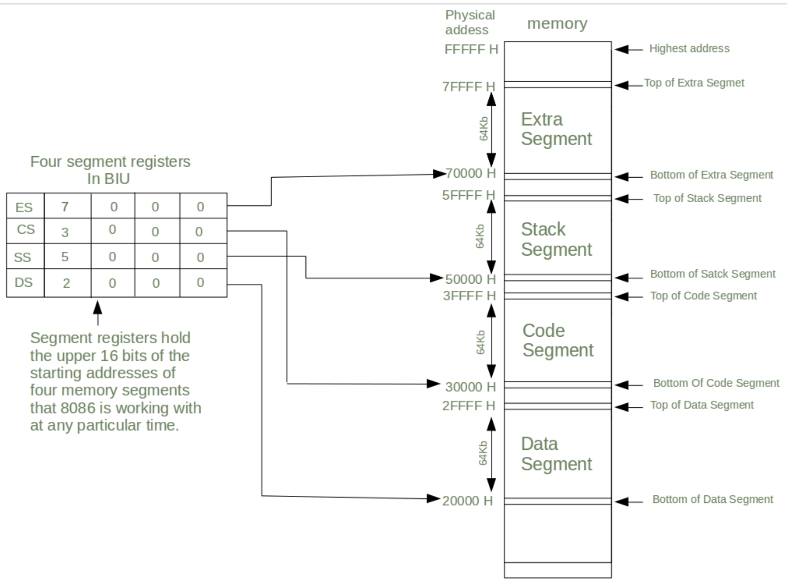
Memory management in the 8086 microprocessor involves the organization of memory into segments and the utilization of a segmented memory model.

Segmentation is the process in which the main memory of the computer is logically divided into different segments and each segment has its own base address. It is basically used to enhance the speed of execution of the computer system, so that the processor is able to fetch and execute the data from the memory easily and fast.

Memory Segments

The 8086 microprocessor has a 20-bit address bus, which means it can access up to 1 MB (2^20 bytes) of memory. The memory is divided into segments, each of which is 64 KB (2^16 bytes) in size. The 8086 has four segment registers – CS, DS, SS, and ES – which are used to access different segments of memory.

8086 Memory Management



8086 Memory Management

Advantages of the Segmentation

- It provides a powerful memory management mechanism.
- Data related or stack related operations can be performed in different segments.
- It allows to processes to easily share data.
- It allows to extend the address ability of the processor, i.e. segmentation allows the use of 16 bit registers to give an addressing capability of 1 Megabytes. Without segmentation, it would require 20 bit registers.

Disadvantages of the Segmentation

- Increased Complexity: Memory Segmentation increases processor instruction execution overhead.
- Inefficient Memory Use: Segmentation allows flexibility, but determining segment sizes needs expertise and precision to avoid wasting memory space.
- Compatibility Issues: Non-segmented programs may not work on segmented architectures.

8086 Memory Management

Memory Addressing

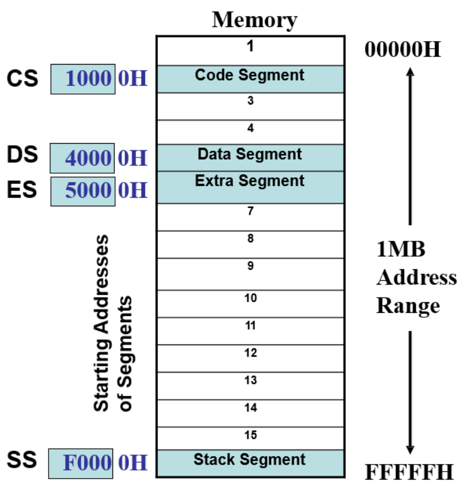
In 8086, there are four types of addresses frequently mentioned: the Segment Address , offset address, logical address, and physical address. The 8086 has a memory management unit (MMU) that maps logical addresses (segment and offset) to physical addresses in memory. The MMU performs address translation and protection, and it is responsible for mapping segments to physical memory.

✓ **Segment Address:** The segment address is a 16-bit address indicating a segment block in memory. Each segment in the 8086 microprocessor represents a block of 64 KB of memory space.

✓ **Offset Address:** Also known as the effective address, the offset address refers to the location within a 64K-byte segment range. It is contained in 16-bit registers like IP, BP, SP, BX, SI, or DI. The offset address in 8086 ranges from 0000H – FFFFH.

8086 Memory Management

- Address of a segment is of 20-bits. A segment register stores only upper 16-bits.
- BIU always inserts zeros for the lowest 4-bits of the 20-bit starting address.
- E.g. if CS = 348AH, then the code segment will start at 348A0H.
- *Note:* A 64-KB segment can be located anywhere in the memory, but will start at an address with zeros in the lowest 4-bits.

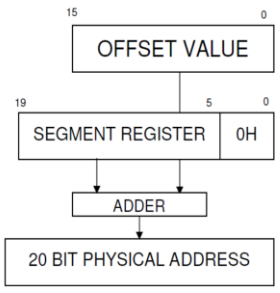


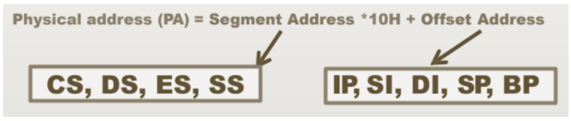
8086 Memory Management

✓**Logical Address:** A logical address consists of a segment value and an offset address. The segment value identifies a segment of memory, while the offset address points to a location within that segment.

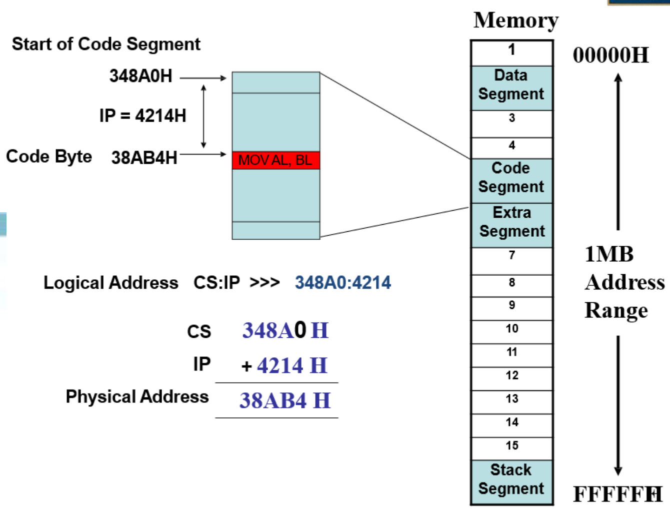
✓**Physical Address**: A physical address represents the actual location in the memory hardware where data is stored. It is a 20-bit address formed by combining the segment address and the offset address. In the 8086 microprocessor, the physical address has a range of 00000H – FFFFFH.

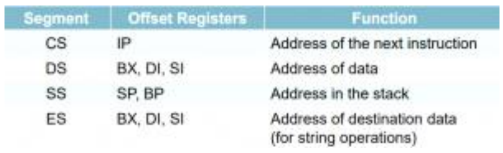
Segment value : Offset address




Physical address (PA) = Segment Address *10H + Offset Address
CS, DS, ES, SS
IP, SI, DI, SP, BP

8086 Memory Management





Segment	Offset Registers	Function
CS	IP	Address of the next instruction
DS	BX, DI, SI	Address of data
SS	SP, BP	Address in the stack
ES	BX, DI, SI	Address of destination data (for string operations)

8086 Memory Management

- ***Addressing in Code Segment***

- To execute a program, the 8086 fetches the instructions from the ***code segment***.
- The ***logical address*** of an instruction consists **CS** (Code Segment) and **IP**(instruction pointer).
- ***Logical Address* in Code segment** is represented by using **segment address in CS** register and **Offset Address in IP** register as follows:

CS:IP (16 bit CS and 16 bit IP making total of 32 bits)

Example: If CS register contains 2500H and IP register contains 95F3H. What is the ***Locical Adress*** in the code segment?

CS:IP → **2500:95F3** **(default in adressing is hex. You don't need H)**

8086 Memory Management

Example: If CS register contains 1980H and IP register contains 78FEH. What is the ***Physical Adress*** in the code segment?

Logical address: CS:IP → **1980:78FE**

1.	**Start with CS**	1980	
2.	**Shift left CS**	1980**0**	
3.	**Add IP**	78FE	(1980**0**+ 78FE =210FE)

Physical address: The microprocessor will retrieve the instruction from the memory locations starting from **210FE** ***(20 bit address).***

8086 Memory Management

Example: If CS register contains 1980H and IP register contains 78FEH. What is the ***Physical Adress*** in the code segment?

Logical address: CS:IP → **1980:78FE**

1.	**Start with CS**	1980	
2.	**Shift left CS**	1980**0**	
3.	**Add IP**	78FE	(1980**0**+ 78FE =210FE)

Physical address: The microprocessor will retrieve the instruction from the memory locations starting from **210FE** ***(20 bit address).***

8086 Memory Management

Example: If CS=24F6H and IP=634AH, determine:

a) The logical address
b) The offset address
c) The physical address
d) The lower range of the code segment
e) The upper range of the code segment

Solution:

a) The logical address is;	**24F6:634A**
b) The offset address is;	**634A**
c) The Physical address is;	**24F60+634A= 2B2AA**
d) The lower range of the code segment:	**24F6:0000 → 24F60+0000 = 24F60**
e) The upper range of the code segment:	**24F6:FFFF → 24F60+FFFF = 34F5F**

8086 Memory Management

- ***Addressing in Data Segment***

- The area of memory allocated strictly for data is called ***data segment.***
- ***Data segment*** contains ***variables*** containing single values and arrays of values, where ***code segment*** only contain program ***instructions***.
- ***Logical Address*** **in Data Segment** is represented by using **segment address in DS r**egister and **Offset Address in BX, SI or DI** registers.

DS:BX
DS:SI
DS:DI

- At any time ***three locations*** in the data segment are pointed with ***DS:BX***, ***DS:SI*** and ***DS:DI*** respectively.

8086 Memory Management

- ***Addressing in Data Segment***

Example: If DS=7FA2H and the offset is 438EH, determine:

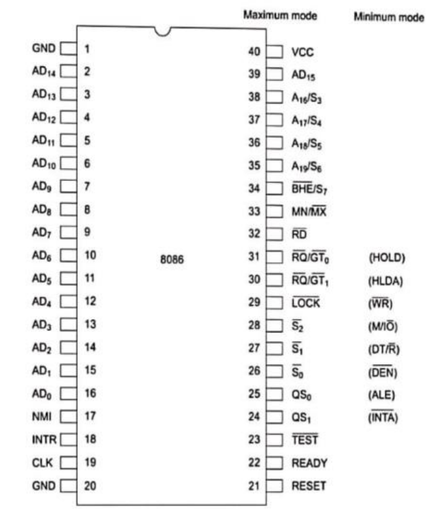
a) The physical address
b) The lower range of the data segment
c) The upper range of the data segment
d) Show the logical address

Solution:

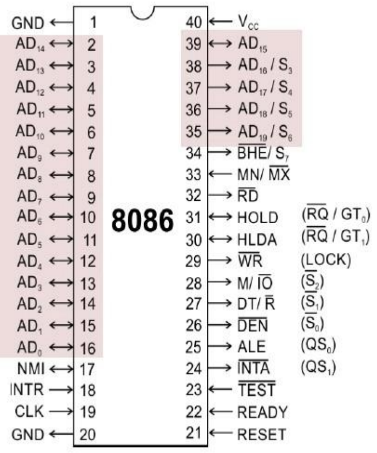
a) The Physical address is;	**7FA20+438E = 83DAE**
b) The lower range:	**7FA20+0000= 7FA20**
c) The upper range:	**7FA20+FFFF = 8FA1F**
d) The logical address is;	**7FA2:438E**

8086 Pin Diagram

8086 was the first 16-bit microprocessor available in 40-pin DIP (Dual Inline Package) chip. Let us now discuss in detail the pin configuration of a 8086 Microprocessor.



8086 Pin Diagram



AD_0-AD_{15} (Bidirectional)

Address/Data bus

Low order address bus; these are multiplexed with data.

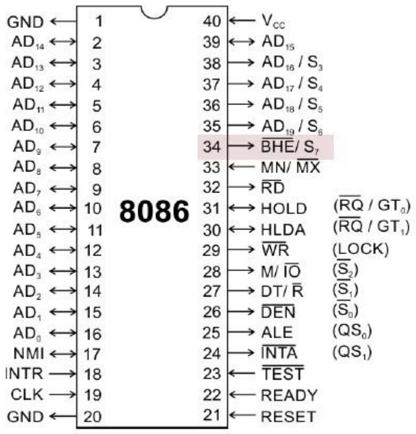
When AD lines are used to transmit memory address the symbol A is used instead of AD, for example A_0-A_{15}.

When data are transmitted over AD lines the symbol D is used in place of AD, for example D_0-D_7, D_8-D_{15} or D_0-D_{15}.

A_{16}/S_3, A_{17}/S_4, A_{18}/S_5, A_{19}/S_6

High order address bus. These are multiplexed with status signals

8086 Pin Diagram



BHE (Active Low)/S_7 (Output)

Bus High Enable/Status

It is used to enable data onto the most significant half of data bus, D_8-D_{15}. 8-bit device connected to upper half of the data bus use BHE (Active Low) signal. It is multiplexed with status signal S_7.

MN/ MX

MINIMUM / MAXIMUM

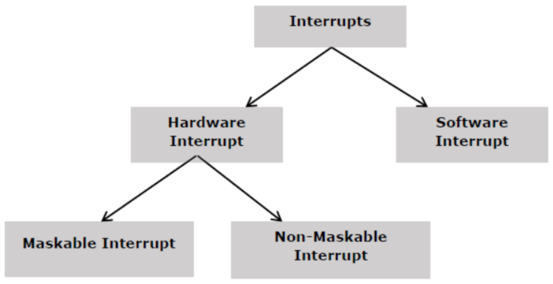
This pin signal indicates what mode the processor is to operate in.

RD (Read) (Active Low)

**The signal is used for read operation.
It is an output signal.
It is active when low.**

8086 Interrupts

Interrupt is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor. The microprocessor responds to that interrupt with an ISR (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt.



8086 Interrupts

Software Interrupts

These are instructions inserted within the program to generate interrupts. There are 256 software interrupts in the 8086 microprocessor.

Hardware Interrupts

Hardware interrupt is caused by any peripheral device by sending a signal through a specified pin to the microprocessor.

The 8086 has two hardware interrupt pins, i.e. **NMI** and **INTR**. NMI is a non-maskable interrupt and INTR is a maskable interrupt having lower priority. One more interrupt pin associated is **INTA** called interrupt acknowledge.

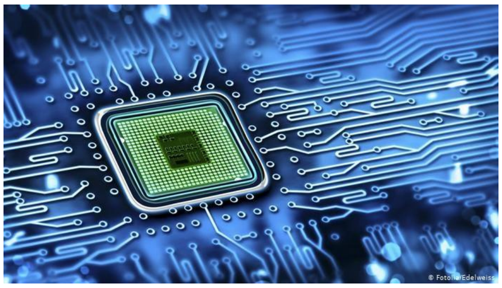
- ✓**NMI (Non-Maskable Interrupt):** This interrupt is non-maskable, meaning it cannot be ignored or disabled by the processor. It typically handles critical system events that require immediate attention.

8086 Interrupts

✓**INTR (Interrupt Request):** Unlike NMI, INTR is a maskable interrupt with lower priority. The processor can enable or disable this interrupt based on program requirements, allowing for greater flexibility in interrupt handling.

✓**INTA (Interrupt Acknowledge):** This pin is activated by the processor to acknowledge an interrupt request. It communicates with external devices to confirm that an interrupt has been recognized and is being processed.

These interrupt pins play crucial roles in managing and responding to external events and signals, ensuring proper interaction between the microprocessor and peripheral devices.



Thanks

ASSEMBLY LANGUAGE

INTRODUCTION

1.1 What is Assembly Language?

Assembly language is a low-level programming language designed for direct communication with a computer's hardware. It consists mostly of symbolic equivalents representing specific machine instructions for a particular computer architecture. Assembly code is converted into executable machine code by an assembler utility program

1.2 Why learn Assembly Language?

1. To understand computer architecture and operating system.

2. To master assembly language for practical use:

a. Some types of programming are challenging or impossible in high-level languages. Direct OS communication, for example. Therefore, a fast program may require a small memory area.

b. Create an assembly language subroutine program and call it from the high-level language. (this is to remove restrictions where there are certain operations which are not allowed in a high-level language)

1.3 What is an Assembler?

An assembler is a type of computer program that interprets software programs written in assembly language into machine language, code and instructions that can be executed by a computer.
An assembler enables software and application developers to access, operate and manage a computer's hardware architecture and components.

Assembly Language **Assembler** **Machine Language**

(source code) $\longrightarrow$ **(object code)**

1.4 What is a Compiler?

A compiler is a program, that translate the high-level language programming into machine language.

A single command of high-level language is translated into many machine instructions by the compiler.

High-level language Compiler Machine Language

(source code) $\longrightarrow$ (object code)

1.5 Editor

An assembly language can be written using any text editor and save in a computer as a text file. Example: EDIT, notepad, etc.

1.6 Debugger

A debugger is a software program used to test and find bugs (errors) in other programs.

1.7 EMU8086 – The Microprocessor Emulator

EMU8086 is a microprocessor emulator that allows users to emulate 8086 processors, providing a platform to run software designed for these processors on modern computers. Users can download EMU8086 to simulate the features of the 16-bit iAPX 86 chip from Intel, offering a virtual environment to execute and test code written for the 8086 architecture. The software includes features such as a code editor and a compiler, making it a comprehensive tool for developers working with legacy software designed for 8086 processors.

Procedures for writing and executing an assembly program

1. Install emu8086 and open it
2. Click new on the emulator, then chose .COM code template
3. Write your code on the given editor
4. Click [Compile and Emulate] button (or press F5 hot key).
5. Click [Single Step] button (or press F8 hot key), and watch how the code is being executed.

Lab Objectives

In this lab we will learn

- Basic of Assembly Language programming and Familiarization with Emu8086 environment
- Using Emu8086 to understand the step by the step execution of a program, and observe the memory and register contents during and after execution

SYSTEMS ARCHITECTURE

1. Central Processing Unit (CPU)

Microcomputers use a single CPU, as shown in figure 1.

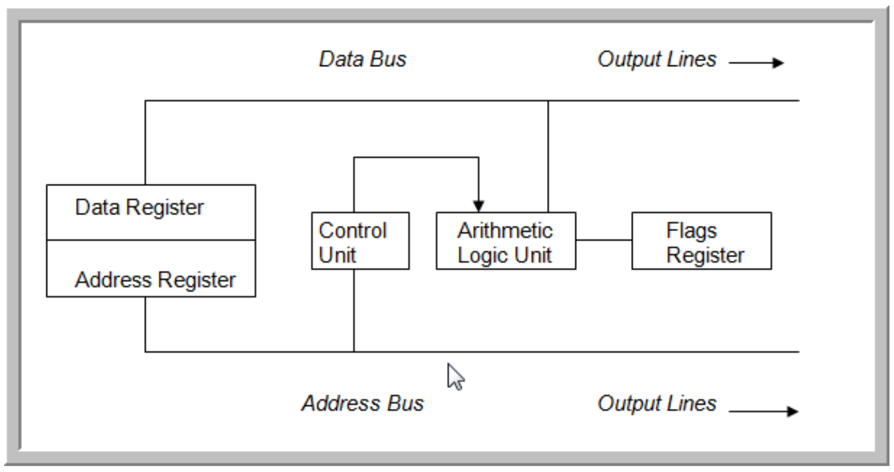


Figure 1.0 CPU parts

The CPU is divided into two parts: the arithmetic logic unit (ALU) and the control unit (Figure 1.0).

ALU functions : carries out arithmetic, logical, and shifting operations.

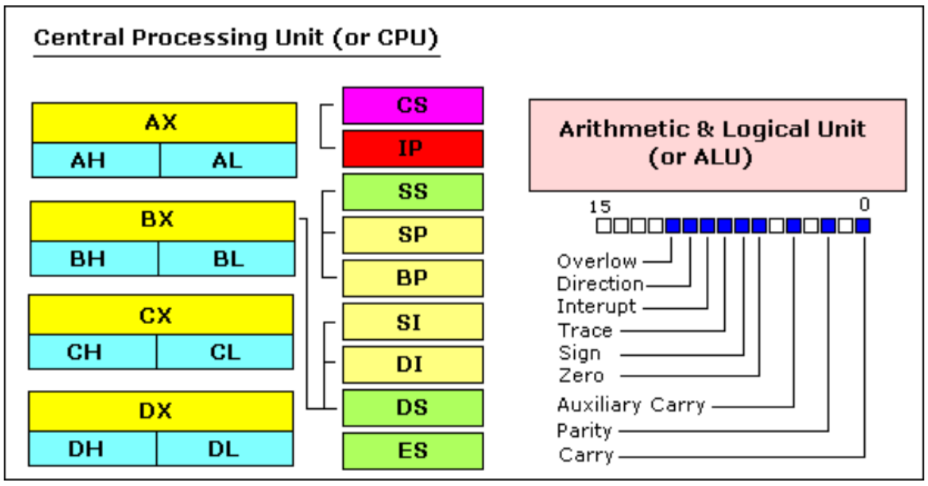
Control unit functions: fetches data and instructions and decodes addresses for the ALU.

2. DATA BUS

The bus is a series of parallel wires that transmit data between the various parts of the CPU. Both control signals and data bits are used when fetching a memory word and placing it in a register.

3. REGISTERS

They are fast storage units inside the central processing unit that are directly connected to the control unit and the arithmetic logic unit. This makes it faster to execute instructions. The registers are 16 bits long, but you have the option of accessing the upper or lower halves of the four data registers: AX, BX, CX and DX.



Type of registers

1. Data Registers

AX, BX CX and DX

- Data registers or general-purpose registers, are used for arithmetic and data movement.
- Each register may be addresses as either a 16-bit or 8-bit value.

Example:

16-bit AX register	
AH register (8 bit) Upper bits	AL register (8 bits) lower bits

Instructions may address either 16-bit or 8-bit data registers from the following list:

AX		BX		CX		DX	
AH	AL	BH	BL	CH	CL	DH	DL

For example, if AX = 126Fh, AL will equal to 6Fh.

Each general-purpose register has special attributes:

AX (accumulator). AX is called the accumulator register because it is favored by the CPU for arithmetic operations.

BX (base). BX register can perform arithmetic and data movement, and it has special addressing abilities.

CX (counter). The CX register acts as a counter for repeating or loping instructions. These instructions automatically repeat and decrement CX and quit when it equals 0.

DX (data) . The DX register has a special role in multiply and divide operations.

2. **Segment registers**
   - The segment registers are as follows:

     CS (code segment). The CS register holds the base location of all executable instructions (code) in a program.

     DS (data segment). The DS register is the default base location for variables. The CPU calculates their locations using the segment value in DS.

     SS (stack segment). The SS register contains the base location of the stack.

     ES (extra segment). The ES register is an additional base location for memory variables.

3. **Index Registers**

   Index registers contain the offsets of variables. The term offsets refers to the distance of a variable, label, or instruction from its base segment. Index registers speed up processing of strings, array, and other data structures containing multiple elements. The index registers are:

   SI (source index). This register takes its name from the string movement instructions, in which the source string is pointed to by the SI register. SI usually contains an offset value from the DS register, but it can address any variable.

   DI (destination index). The DI register acts as the destination for string movement instructions.

   BP (base pointer) . The BP register contains an assumed offset from the SS register, as does the stack pointer. The BP register is often used by a subroutine to locate variables that were passed on the stack by a calling program.

4. **Special Registers**.

   IP (instruction pointer). The IP register always contains the offset of the next instruction to be executed. CS and IP combine to form the complete address of the next instruction about to be executed.

   SP (stack pointer). The SP register contains the *offset*, or distance from the beginning of the stack segment to the top of the stack. The SS and SP registers combine to form the complete top-of-stack address.

5. **Flags Register**

   The flags register is a special 16-bit register with individual bit positions assigned to show the status of the CPU or the results of arithmetic operations. Each relevant bit position is given a name; other positions are undefined.

Bit Position

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	O	D	I	T	S	Z	x	A	x	P	x	C

O	=	Overflow	S	=	Sign
D	=	Direction	Z	=	Zero
I	=	Interrupt	A	=	Auxiliary Carry
T	=	Trap	P	=	Parity
x	=	undefined	C	=	Carry

A flag or bit is *set* when it equals 1 ; it is *clear* (or reset) when it equals 0. The CPU sets flags by turning on individual bits in the Flags register. There are two basic types of flags: *control flags* and *status flags*.

4 Clock

Each of the individual operations that place within the CPU must be synchronized by a clock. The most basic unit of time for machine instructions is called the machine cycle.

Writing your first Assembly codes:

1- Write the following code in emulator and examine the contents of registers by single stepping:

```
org 100H
mov ax,2
mov bx,2
add ax,bx
mov cx,ax
ret
```

2- Write the following code in emulator and examine the contents of registers by single stepping

```
ORQ100H
MOV AL, 57
MOV DH.69
MOV DL, 72
MOV BX, DX
MOV BH, AL
MOV BL, 9F
MOV AH, 20
ADD AX, DX
ADD CX, BX
ADD AX, 1F35
RET
```

ASSEMBLY INSTRUCTIONS

Definitions:

Variables: Variables are named memory locations that store data values. They are used to represent changing or mutable data in a program.

Constants: Constants are fixed values that do not change during the execution of a program. They are typically used for values that remain constant throughout the program.

1- Data Transfer Instructions

Data transfer instructions in assembly language are responsible for moving data between registers, memory locations, and other data storage locations. Here are some common data transfer instructions:

MOV (Move):

Syntax: **MOV** destination, source

Description: Copies the value from the source operand to the destination operand.

```
MOV AX, BX ; Copy the value in BX to AX
```

LEA (Load Effective Address):

Syntax: **LEA** destination, source

Description: Loads the effective address of the source operand into the destination operand.

```
LEA DI, array ; Load the effective address of the 'array' into DI
```

XCHG (Exchange):

Syntax: **XCHG** operand1, operand2

Description: Exchanges the values of operand1 and operand2.

```
XCHG AX, BX ; Exchange the values in AX and BX
```

PUSH and POP:

PUSH Syntax: **PUSH** source

POP Syntax: **POP** destination

Description: PUSH pushes the source operand onto the stack, and POP retrieves the top value from the stack into the destination operand.

```
PUSH AX ; Push the value in AX onto the stack
POP BX ; Pop the top value from the stack into BX
```

2- Bit Manipulation Instructions

Bit manipulation instructions in assembly language are used to manipulate individual bits within registers or memory. Here are some common bit manipulation instructions:

AND (Bitwise AND):

Syntax: **AND** destination, source

Description: Performs a bitwise AND operation between the destination and source operands, storing the result in the destination.

```
AND AL, 0FH ; Clears the upper 4 bits of AL (sets them to 0)
```

OR (Bitwise OR):

Syntax: **OR** destination, source

Description: Performs a bitwise OR operation between the destination and source operands, storing the result in the destination.

```
OR BH, 80H ; Sets the most significant bit of BH to 1
```

XOR (Bitwise XOR):

Syntax: **XOR** destination, source

Description: Performs a bitwise XOR (exclusive OR) operation between the destination and source operands, storing the result in the destination.

```
XOR DL, 0FFH ; Inverts all bits in DL
```

NOT (Bitwise NOT):

Syntax: **NOT** operand

Description: Performs a bitwise NOT operation on the operand, inverting all its bits.

```
NOT CX ; Inverts all bits in CX
```

SHL/SHR (Shift Left/Shift Right):

Syntax: **SHL** destination, count / **SHR** destination, count

Description: Shifts the bits in the destination operand left (SHL) or right (SHR) by the specified count.

```
SHL AX, 1 ; Shifts all bits in AX one position to the left (multiply by 2)
SHR CX, 3 ; Shifts all bits in CX three positions to the right (divide by 8)
```

ROL/ROR (Rotate Left/Rotate Right):

Syntax: **ROL** destination, count / **ROR** destination, count

Description: Rotates the bits in the destination operand left (ROL) or right (ROR) by the specified count.

```
ROL DX, 1 ; Rotates all bits in DX one position to the left
ROR SI, 2 ; Rotates all bits in SI two positions to the right
```

3- Arithmetic Instructions

Arithmetic instructions in assembly language are used to perform mathematical operations on data. Here are some common arithmetic instructions:

1. Increment & decrement
2. Addition (`add`) & Subtraction (`sub`)
3. Multiplication (`mul`) & Division (`div`)

Increment and decrement

INC (Increment):

Syntax: **INC** operand

Description: Increments the value of the operand by 1.

```
INC CX ; CX = CX + 1
```

DEC (Decrement):

Syntax: **DEC** operand

Description: Decrements the value of the operand by 1.

```
DEC SI ; SI = SI - 1
```

Addition and subtraction

ADD (Addition):

Syntax**: ADD** destination, source

Description: Adds the source operand to the destination operand and stores the result in the destination.

```
ADD AX, BX ; AX = AX + BX
```

SUB (Subtraction):

Syntax: **SUB** destination, source

Description: Subtracts the source operand from the destination operand and stores the result in the destination.

```
SUB AX, BX ; AX = AX - BX
```

The destination and source can be a combination of the following:

1. Register to Register

```
mov ax, 10 # move 10 to register 'ax'
mov bx, 5  # move 5 to register 'bx'
add ax, bx
```

2. Memory to Register

```
mov ax, 10  # move 10 to register 'ax'
add ax, [num1] # access value stored at location num1 and store sum
               # again in the register 'ax'
```

3. Register to Memory

```
mov ax, 10
add [num1], ax
```

4. Constant to Register

```
mov ax, 10
add ax, 5 # add 5 to the register 'ax' value and store the sum in 'ax'
```

5. Constant to Memory

```
add [num1], 5 # add 5 to the value stored at num1 and replace the value
              # there with the sum
```

Multiplication and division

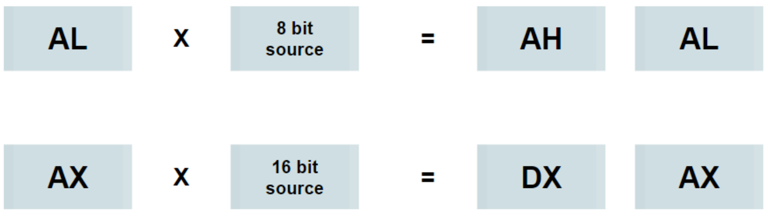
MUL (Multiplication):

Syntax: **MUL/IMUL** source
Description: Multiplies the accumulator register (EAX) by the source operand, and the result is stored in the accumulator.

```
MUL BX ; EAX = EAX * BX
```

If the source operand is a byte (8 bits), we multiply it by the value stored in the register AL. The result is returned in the registers AH and AL. The higher half of the 16-bit result is stored in AH and the lower half in AL. This means that if the result is small enough to be represented in 8 bits, AH would contain 0.

If the source operand is a word (16 bits), then we multiply it by the value stored in the register AX and the result is returned in the registers DX and AX. The higher half of the 32-bit result is stored in DX and the lower half in AX.



Variations of the mul instruction

DIV (Division):

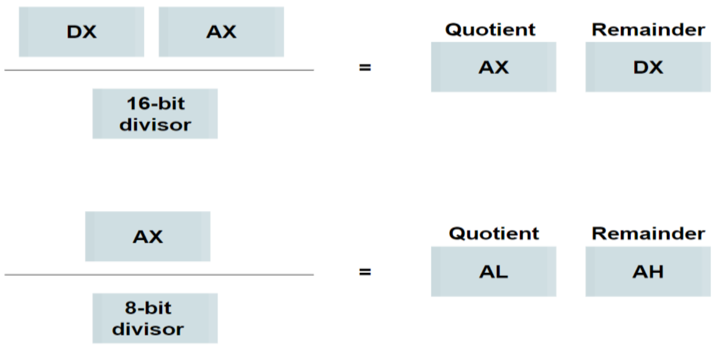
Syntax: **DIV/IDIV** divisor
Description: Divides the double-word in the accumulator (EDX:EAX) by the divisor operand. The quotient is stored in EAX, and the remainder is stored in EDX.

```
MOV EAX, 100 ; Set the dividend to 100
MOV ECX, 5   ; Set the divisor to 5
DIV ECX      ; EAX = EAX / ECX, EDX = EAX % ECX
```

div performs an integer division of the accumulator and the source operand. The result consists of an integer quotient and remainder.

If the source operand is a byte, the 16-bit number stored in AX is divided by the operand. The 8-bit quotient is stored in AL and the 8-bit remainder in AH.

If the source operand is a word, the 32-bit number stored in DX : AX is divided by the operand. The higher half of the number is stored in DX and the lower in AX. The 16-bit quotient is stored in AX and the remainder in DX.



Variations of the div instruction

Code

The following code shows how we can implement arithmetic instructions in assembly language:

```
# a program to show arithmetic instructions in AL

# addition
mov ax, 5   # load number in ax
mov bx, 2   # load number in bx
add ax, bx  # accumulate sum in ax

# subtraction
mov cx, 10
mov dx, 3
sub cx, dx  # accumulate difference in cx

# refreshing registers
mov ax, 0
mov bx, 0
mov cx, 0
mov dx, 0

# multiplication - 8 bit source
mov al, 5
mov bl, 10
mul bl      # result in ax

# refreshing registers
mov ax, 0
mov bx, 0

```

```
# multiplication - 16 bit source
mov ax, 5
mov bx, 10
mul bx      # result in dx:ax

# refreshing registers
mov ax, 0
mov bx, 0

# divison - 8 bit source
mov al, 23
mov bl, 4
div bl      # quotient in al, remainder in ah

# refreshing registers
mov ax, 0
mov bx, 0

# divison - 16 bit source
mov ax, 23
mov bx, 4
div bx      # quotient in ax, remainder in dx

```

Simple Assembly Program Structure in Emu8086

In emu8086, a simple assembly program follows a standard structure. Here's a basic template for a simple assembly program:

```
org 100H   ; Set the origin to 100H (standard for .COM files)

section .data
    ; Declare data (constants, initialized variables)


section .bss
    ; Declare uninitialized variables (if any)


section .text
    global _start
_start:
    ; Code segment - Entry point of the program
    ; Your assembly code goes here

    ; Example: Display a message
    mov     ah, 09h             ; DOS function to display a string
    lea     dx, message         ; Load the offset address of the message
    int     21h                 ; Call the DOS interrupt

    ; Example: Exit the program
    mov     ah, 4Ch             ; DOS function to exit the program
    int     21h                 ; Call the DOS interrupt

section .data
    message db 'Hello, World!', 0 ; Null-terminated string
```

Explanation:

- **org 100H**: Sets the origin to 100H, indicating that the program should be loaded at memory offset 100H. This is a convention for .COM files in emu8086.
- **section .data**: Declares the data section, where constants and initialized variables are defined.
- **section .bss**: Declares the BSS section, where uninitialized variables can be declared. In this simple example, it is kept empty.
- **section .text**: Declares the code section, where the actual assembly code is written.
- **global _start**: Declares a global symbol _start as the entry point of the program.
- **_start**: Marks the beginning of the code segment. Execution starts from here.
- **MOV, LEA, INT**: These are instructions for moving data, loading effective addresses, and invoking software interrupts, respectively. The example shows a simple code segment that displays a message and exits the program.
- **section .data**: Additional data section where the message is declared.

This is a basic structure, and real-world assembly programs may have additional sections or directives depending on the requirements.

Lab Tasks

1. For all assembly programs made in emu8086, the first line of code, org 100H, must be in place. Why should this be the first thing you write?
2. Write a program that exchanges two values stored in two consecutive memory locations using the MOV instruction.
3. Develop a program that toggles (flips) all the bits in a byte.
4. Implement an x86 assembly program to add two 16-bit integers and store the result.
5. Develop a program that multiplies two 8-bit unsigned integers using the MUL instruction.
6. Explain the role of the Data Transfer Instructions in assembly language programming. Provide examples.
7. What is the purpose of the Bit Manipulation Instructions in assembly language? Illustrate with examples.
8. Write an assembly program that uses a combination of data transfer, bit manipulation, and arithmetic instructions to solve a practical problem of your choice.

8086 Instruction Sets

Addressing Modes

Exercise 1: Register Addressing Mode

- **Task**: Move the value of register BX into register AX.
- **Code**:

```
MOV BX, 1234H; Initialize BX register with a value
MOV AX, BX ; Move the content of register BX into register AX
```

- **Answer**: This code initializes the BX register with a value and then moves its content into register AX.

Exercise 2: Immediate Addressing Mode

- **Task**: Move the immediate value 1234H into register AX.
- **Code**:

```
MOV AX, 0; Initialize AX register with 0
MOV AX, 1234H; Move immediate data 1234H into register AX
```

- **Answer**: This code initializes the AX register with 0 and then moves the immediate data 1234H into register AX.

Exercise 3: Direct Addressing Mode

- **Task**: Move the value stored at memory location 2000H into register AX.
- **Code**:

```
MOV AX, 1234H; Initialize AX register with 1234H
MOV AX, [2000H]; Move the content of the memory location 2000H into AX
```

- **Answer**: This code initializes the AX register with 1234H, This instruction moves the content of the memory location 1234H into the AX register.

Exercise 4: Register Indirect Addressing Mode

- **Task**: Move the value stored at memory location 1234H into register AX.
- **Code**:

```
MOV AX, 0; Initialize AX register with 0

MOV SI, 1234H; Initialize SI register with the memory location 1234H

MOV AX, [SI]; Move the content of the memory location pointed to by SI into AX
```

- **Answer**: This code initializes the AX register with 0, initializes the SI register with the memory location 1234H, and then moves the content of that memory location into register AX.

Exercise 5: Based-Index Addressing Mode

- **Task**: Move the value stored at memory location addressed by the sum of BX and SI registers into register AX.
- **Code**:

```
MOV BX, 1000H; Initialize BX register with 1000H

MOV SI, 200H; Initialize SI register with 200H

MOV AX, [BX+SI]; Move the content of the memory location addressed by BX+SI into AX
```

- **Answer**: This code initializes the BX register with 1000H, the SI register with 200H, and then moves the content of the memory location addressed by the sum of BX and SI into register AX.

Exercise 6: Based Addressing Mode

- **Task:** Move the value stored at memory location addressed by the sum of a base register and a constant offset into a register.
- **Code:**

```
MOV BX, 1000H; Initialize BX register with base address

MOV AL, [BX+10]; Move the content of memory location (1000H + 10) into AL
```

Answer: This code initializes the BX register with a base address of 1000H and moves the content of the memory location addressed by (1000H + 10) into the AL register.

Exercise 7: Indexed Addressing Mode

Task: Move the value stored at memory location addressed by the sum of a base register and an index register into a register.

Code:

```
MOV BX, 2000H ; Initialize BX register with base address

MOV SI, 0200H ; Initialize SI register with index value

MOV AL, [BX+SI] ; Move the content of memory location (2000H + 0200H) into AL
```

Answer: This code initializes the BX register with a base address of 2000H, the SI register with an index value of 0200H, and then moves the content of the memory location addressed by (2000H + 0200H) into the AL register.

Exercise 8: Based Indexed with Displacement Mode

Task: Move the value stored at memory location addressed by the sum of a base register, an index register, and a displacement value into a register.

Code:

```
MOV BX, 3000H; Initialize BX register with base address

MOV SI, 0300H; Initialize SI register with index value

MOV DI, 0010H; Initialize DI register with displacement value

MOV AL, [BX+SI+DI]; Move the content of memory location (3000H + 0300H + 0010H) into AL
```

Answer: This code initializes the BX register with a base address of 3000H, the SI register with an index value of 0300H, the DI register with a displacement value of 0010H, and then moves the content of the memory location addressed by (3000H + 0300H + 0010H) into the AL register.

8086 Instruction Sets

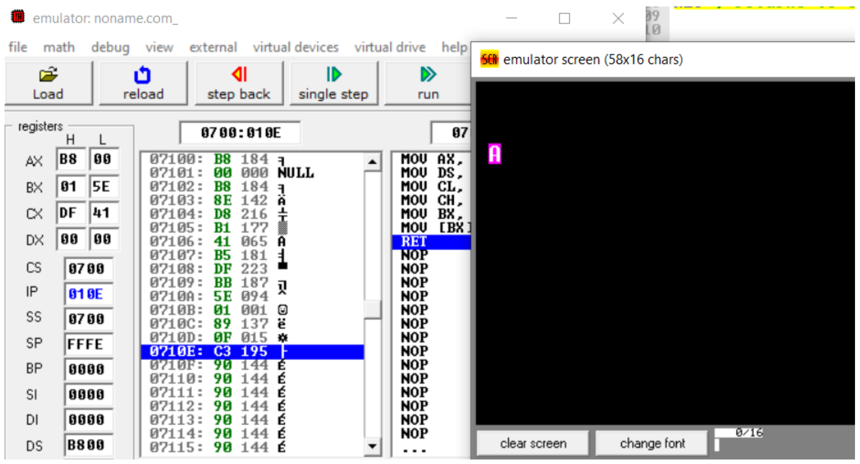
Data Transfer Instructions

Exercise 1: MOV (Move) Instruction

- **Task**: Write an assembly program that demonstrates the use of MOV instruction.
- **Code**:

```
ORG 100h ; this directive required for a simple 1 segment .com program.
MOV AX, 0B800h ; set AX to hexadecimal value of B800h.
MOV DS, AX ; copy value of AX to DS.
MOV CL, 'A' ; set CL to ASCII code of 'A', it is 41h.
MOV CH, 00DFh ; set CH to 00DFh value.
MOV BX, 15Eh ; set BX to 15Eh.
MOV [BX], CX ; copy contents of CX to memory at B800:015E
RET ; returns to operating system.
```

- **Code Result**:



```
registers
      H   L
AX   B8  00
BX   01  5E
CX   DF  41
DX   00  00
CS   0700
IP   010E
SS   0700
SP   FFFE
BP   0000
SI   0000
DI   0000
DS   B800

0700:010E

07100: B8 184        MOV AX,
07101: 00 000 NULL   MOV DS,
07102: B8 184        MOV CL,
07103: 8E 142        MOV CH,
07104: D8 216        MOV BX,
07105: B1 177        MOV [BX]
07106: 41 065 A      RET
07107: B5 181        NOP
07108: DF 223        NOP
07109: BB 187        NOP
0710A: 5E 094        NOP
0710B: 01 001        NOP
0710C: 89 137 ë      NOP
0710D: 0F 015        NOP
0710E: C3 195        NOP
0710F: 90 144 É      NOP
07110: 90 144 É      NOP
07111: 90 144 É      NOP
07112: 90 144 É      NOP
07113: 90 144 É      NOP
07114: 90 144 É      NOP
07115: 90 144 É      ...
```

A

Exercise 2: LEA (Load Effective Address) Instruction

- **Task**: Write an assembly program to calculate the effective address.
- **Code**:

```
ORG 100h ;
MOV BX, 005h;
LEA SI, [BX+010h]; Calculate the effective address of (BX + 010h)
RET ; returns to operating system.
```

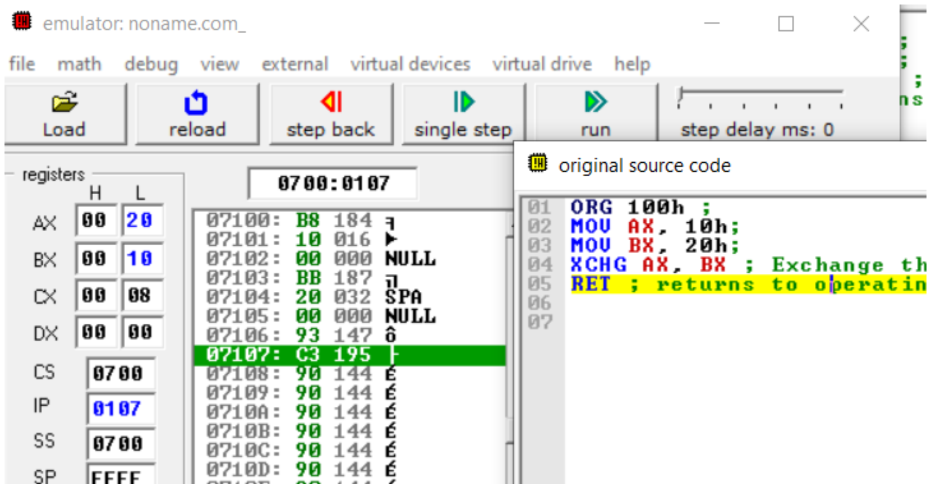
- **Code Result**: This code initializes the BX register with 05h and then Load Effective Address into register SI.

Exercise 3: XCHG (Exchange) Instruction

- **Task**: Using single instruction swap the contents of ax ,with bx
- **Code**:

```
ORG 100h ;
MOV AX, 10h;
MOV BX, 20h;
XCHG AX, BX; Exchange the values of AX and BX
RET ; returns to operating system.
```

- **Code Result**:



```
     H   L
AX   00  20
BX   00  10
CX   00  08
DX   00  00
CS   0700
IP   0107
SS   0700
SP   FFFF

0700:0107

07100: B8 184 ╕
07101: 10 016 ►
07102: 00 000 NULL
07103: BB 187 ╗
07104: 20 032 SPA
07105: 00 000 NULL
07106: 93 147 ô
07107: C3 195 ├
07108: 90 144 É
07109: 90 144 É
0710A: 90 144 É
0710B: 90 144 É
0710C: 90 144 É
0710D: 90 144 É
```

```
ORG 100h ;
MOV AX, 10h;
MOV BX, 20h;
XCHG AX, BX ; Exchange th
RET ; returns to operatin
```

Exercise 4: PUSH and POP Instructions

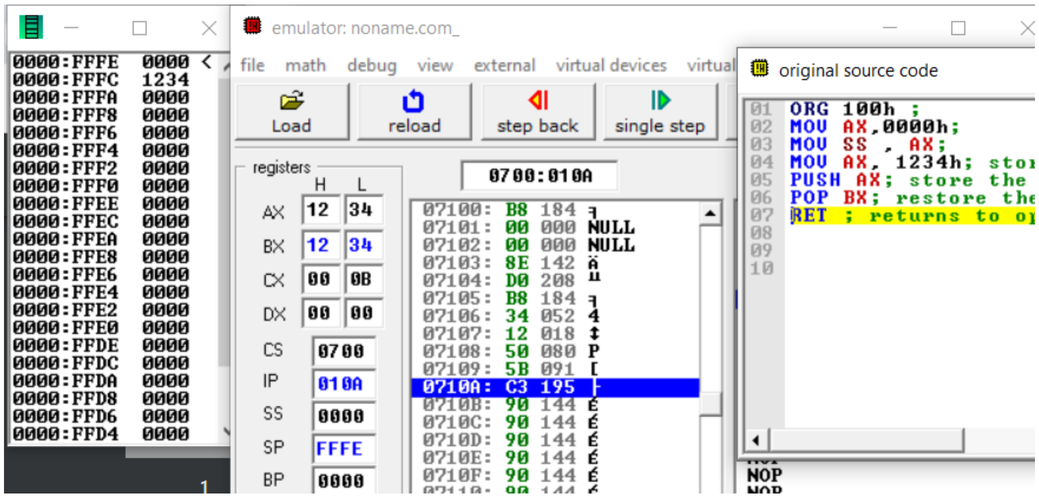
- **Task**: Write a 8086 program to do the following :

*store the value 1234h to the register ax .
*store the register ax to the stack.
*restore the value of stack into BX.

- **Code**:

```
ORG 100h ;
MOV AX,0000h;
MOV SS, AX;
MOV AX, 1234h; store the value 1234h to the register AX.
PUSH AX; store the register AX to the stack
POP BX; restore the value of stack into BX.
RET; returns to operating system.
```

- **Code Result**:



```
0000:FFFE  0000
0000:FFFC  1234
0000:FFFA  0000
0000:FFF8  0000
0000:FFF6  0000
0000:FFF4  0000
0000:FFF2  0000
0000:FFF0  0000
0000:FFEE  0000
0000:FFEC  0000
0000:FFEA  0000
0000:FFE8  0000
0000:FFE6  0000
0000:FFE4  0000
0000:FFE2  0000
0000:FFE0  0000
0000:FFDE  0000
0000:FFDC  0000
0000:FFDA  0000
0000:FFD8  0000
0000:FFD6  0000
0000:FFD4  0000
```

emulator: noname.com_

```
registers
     H   L
AX   12  34
BX   12  34
CX   00  0B
DX   00  00
CS   0700
IP   010A
SS   0000
SP   FFFE
BP   0000
```

```
0700:010A

07100:  B8 184 ╕
07101:  00 000 NULL
07102:  00 000 NULL
07103:  8E 142 Ä
07104:  D0 208 ╨
07105:  B8 184 ╕
07106:  34 052 4
07107:  12 018 ↕
07108:  50 080 P
07109:  5B 091 [
0710A:  C3 195 ├
0710B:  90 144 É
0710C:  90 144 É
0710D:  90 144 É
0710E:  90 144 É
0710F:  90 144 É
```

original source code

```
ORG 100h ;
MOV AX,0000h;
MOV SS , AX;
MOV AX, 1234h; stor
PUSH AX; store the
POP BX; restore the
RET ; returns to o
```

```
NOP
```

8086 Instruction Sets

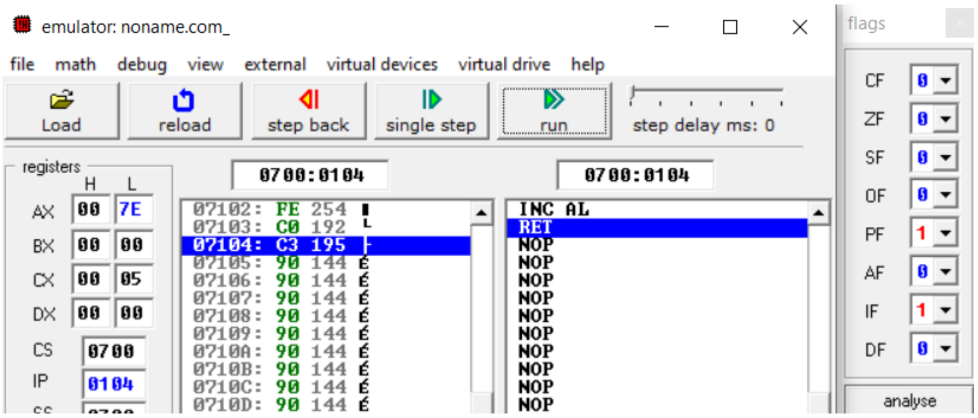
Arithmetic Instructions

Exercise 1: INC (Increment) Instruction

- **Task**: Write an assembly program that demonstrates the use of INC instruction.
- **Code**:

```
ORG 100h ;
MOV AL, 7DH; Sets AL to 7DH
INC AL; AL=AL+1
RET ; returns to operating system.
```

- **Code Result**:



```
registers      H  L
AX            00 7E
BX            00 00
CX            00 05
DX            00 00
CS            0700
IP            0104

0700:0104                 0700:0104
07102: FE 254             INC AL
07103: C0 192             RET
07104: C3 195             NOP
07105: 90 144 É           NOP
07106: 90 144 É           NOP
07107: 90 144 É           NOP
07108: 90 144 É           NOP
07109: 90 144 É           NOP
0710A: 90 144 É           NOP
0710B: 90 144 É           NOP
0710C: 90 144 É           NOP
0710D: 90 144 É           NOP

flags
CF 0
ZF 0
SF 0
OF 0
PF 1
AF 0
IF 1
DF 0
```

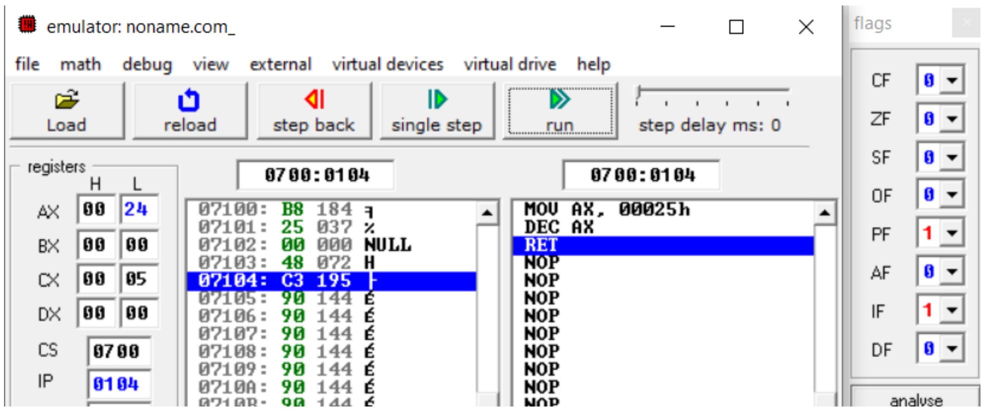
NOTE: The INC instruction adds 1 to the contents of destination operand. It can affect AF, OF, PF, SF and ZF flags.

Exercise 2: DEC (Decrement) Instruction

- **Task**: Write an assembly program that demonstrates the use of DEC instruction.
- **Code**:

```
ORG 100h ;
MOV AX, 25h; Sets AX to 25h
DEC   AX; AX=AX-1
RET ; returns to operating system.
```

- **Code Result**:



```
registers
     H   L
AX   00  24
BX   00  00
CX   00  05
DX   00  00
CS   0700
IP   0104

0700:0104
07100:  B8 184 ╕
07101:  25 037 %
07102:  00 000 NULL
07103:  48 072 H
07104:  C3 195 ├
07105:  90 144 É
07106:  90 144 É
07107:  90 144 É
07108:  90 144 É
07109:  90 144 É
0710A:  90 144 É

0700:0104
MOV AX, 00025h
DEC AX
RET
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP

CF 0
ZF 0
SF 0
OF 0
PF 1
AF 0
IF 1
DF 0
```

Exercise 3: ADD (Addition without Carry) Instructions

- **Task**: Find the result of Y=0EH+0FH and store it in the AX
- **Code**:

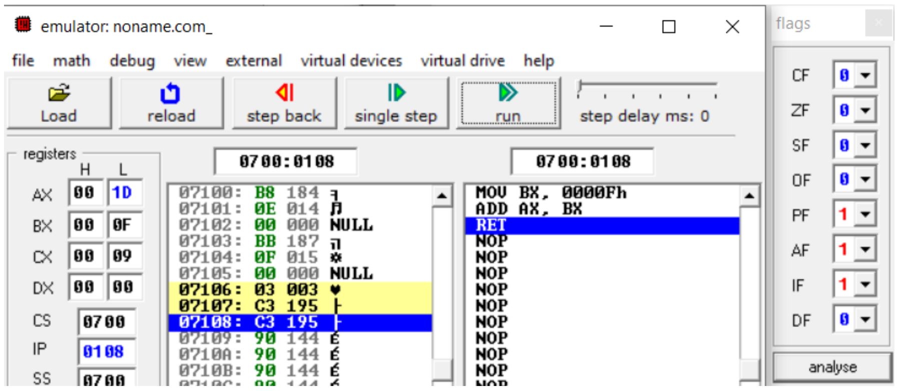
```
ORG 100h ;
MOV AX, 0Eh;

MOV BX, 0Fh;

ADD AX, BX  ; AX now contains 1D

RET; returns to operating system.
```

- **Code Result**:



```
registers
     H   L
AX   00  1D
BX   00  0F
CX   00  09
DX   00  00
CS   0700
IP   0108
SS   0700

0700:0108
07100:  B8 184 ╕
07101:  0E 014 ♫
07102:  00 000 NULL
07103:  BB 187 ╗
07104:  0F 015 ☼
07105:  00 000 NULL
07106:  03 003 ♥
07107:  C3 195 ├
07108:  C3 195 ├
07109:  90 144 É
0710A:  90 144 É
0710B:  90 144 É

0700:0108
MOV BX, 0000Fh
ADD AX, BX
RET
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP

CF 0
ZF 0
SF 0
OF 0
PF 1
AF 1
IF 1
DF 0
```

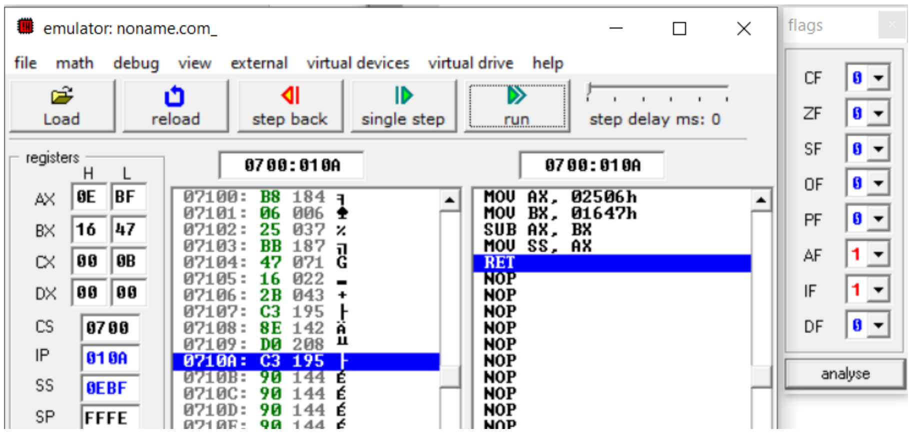
NOTE: If the sum of two operands exceeds the size of destination operand, then it would set the carry flag to 1. The ADD instruction can affect AF, CF, OF, PF, SF, ZF flags depending upon the result. If the result is zero, the ZF=1. Negative result sets SF to 1.

Exercise 4: SUB (Subtraction without Carry) Instruction

- **Task**: Find the result of Y=2506H-1647H and store it in the SS
- **Code**:

```
ORG 100h ;
MOV AX, 2506H ; Sets AX to 2506
MOV BX, 1647H  ; Sets BX to 1647
SUB AX, BX      ; AX=AX-BX
MOV SS, AX     ; SS= AX
RET ; returns to operating system.
```

Code Result:


emulator: noname.com_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
flags
analyse

```
registers
      H   L
AX    0E  BF
BX    16  47
CX    00  0B
DX    00  00
CS    0700
IP    010A
SS    0EBF
SP    FFFE

0700:010A
07100: B8 184 ╕
07101: 06 006 ♠
07102: 25 037 %
07103: BB 187 ╗
07104: 47 071 G
07105: 16 022 ▬
07106: 2B 043 +
07107: C3 195 ├
07108: 8E 142 Ä
07109: D0 208 ╨
0710A: C3 195 ├
0710B: 90 144 É
0710C: 90 144 É
0710D: 90 144 É
0710E: 90 144 É

0700:010A
MOV AX, 02506h
MOV BX, 01647h
SUB AX, BX
MOV SS, AX
RET
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP

CF 0
ZF 0
SF 0
OF 0
PF 0
AF 1
IF 1
DF 0
```

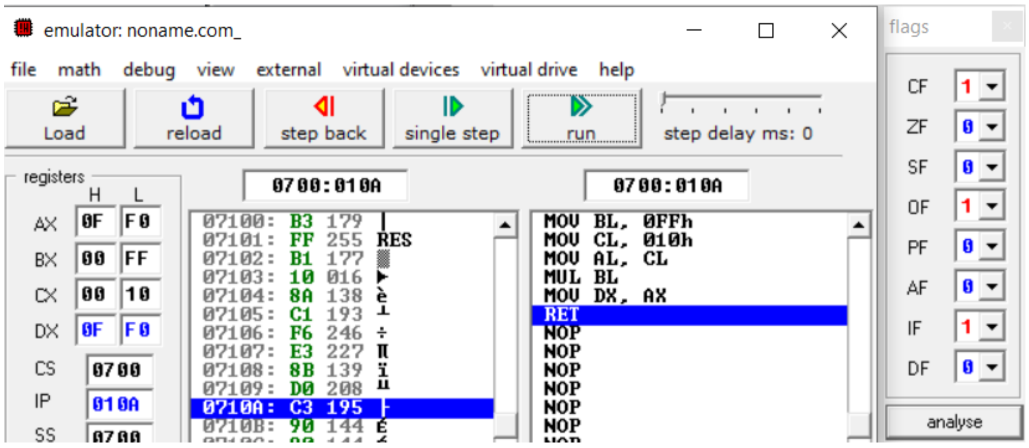
NOTE: The SUB instruction can affect AF, CF, OF, PF, SF and ZF flags depending upon the result obtained from difference.

Exercise 5: MUL (Unsigned Multiplying) Instruction

- **Task**: Multiply two 8 bit unsigned number the first in the register BL, and the second in CL, the result is stored in DX (Assumed BL=0FFH, CL=10H)
- **Code**:

```
ORG 100h ;
MOV BL, 0FFH ; Sets BL to 0FFH
MOV CL, 10H  ; Sets CL to 10H
MOV AL,CL
MUL BL      ; AX=AL*BL
MOV DX, AX    ; DX= AX
RET ; returns to operating system.
```

Code Result:



```
registers
     H   L
AX   0F  F0
BX   00  FF
CX   00  10
DX   0F  F0
CS   0700
IP   010A
SS   0700
```

```
0700:010A
07100:  B3 179 |
07101:  FF 255 RES
07102:  B1 177 ▒
07103:  10 016 ►
07104:  8A 138 è
07105:  C1 193 ┴
07106:  F6 246 ÷
07107:  E3 227 π
07108:  8B 139 ï
07109:  D0 208 ╨
0710A:  C3 195 ├
0710B:  90 144 É
```

```
0700:010A
MOV  BL, 0FFh
MOV  CL, 010h
MOV  AL, CL
MUL  BL
MOV  DX, AX
RET
NOP
NOP
NOP
NOP
NOP
NOP
```

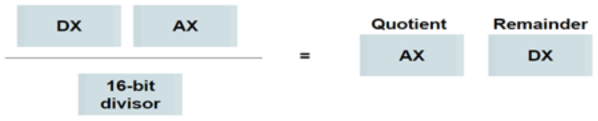
```
CF 1
ZF 0
SF 0
OF 1
PF 0
AF 0
IF 1
DF 0
```

NOTE: The MUL instruction can affect AF, CF, OF, PF, SF and ZF flags depending upon the result obtained from Multiplying.

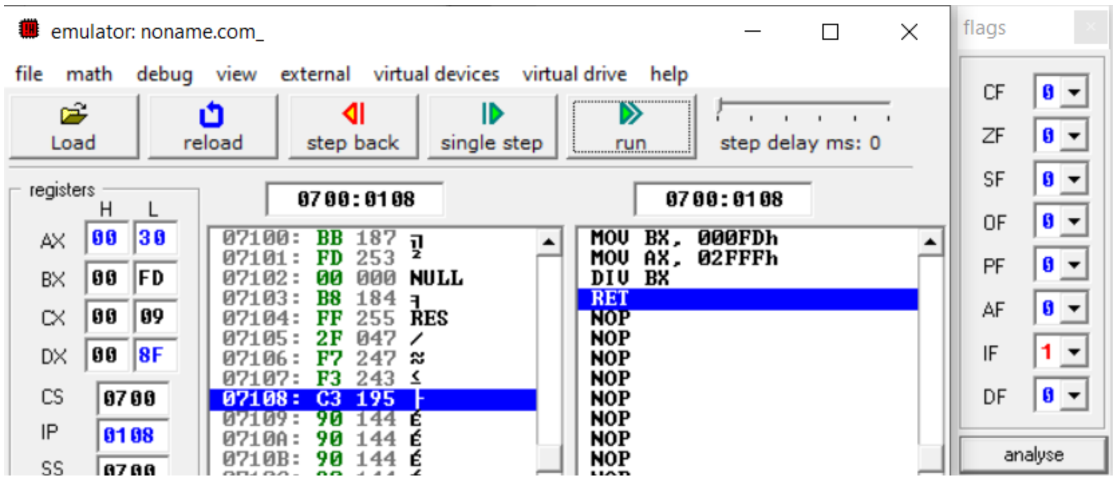
Exercise 6: DIV (Unsigned Division) Instruction

- **Task**: Divide two 16 bit unsigned number the first in the register BX, and the second in AX (Assumed BX=00FDH, AX=2FFFH)
- **Code**:

```
ORG 100h ;
MOV BX, 00FDH ; Sets BX to 00FDH
MOV AX, 2FFFH  ; Sets AX to 2FFFH
DIV BX      ; AX=   0030 (quotient), DX=8F (Remainder)
RET ; returns to operating system.
```



Code Result:



```
registers
     H   L
AX   00  30
BX   00  FD
CX   00  09
DX   00  8F
CS   0700
IP   0108
SS   0700
```

```
0700:0108
07100: BB 187 ╗
07101: FD 253 ²
07102: 00 000 NULL
07103: B8 184 ╕
07104: FF 255 RES
07105: 2F 047 /
07106: F7 247 ≈
07107: F3 243 ≤
07108: C3 195 ├
07109: 90 144 É
0710A: 90 144 É
0710B: 90 144 É
```

```
0700:0108
MOV BX, 000FDh
MOV AX, 02FFFh
DIV BX
RET
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
```

```
CF 0
ZF 0
SF 0
OF 0
PF 0
AF 0
IF 1
DF 0
```

8086 Instruction Sets

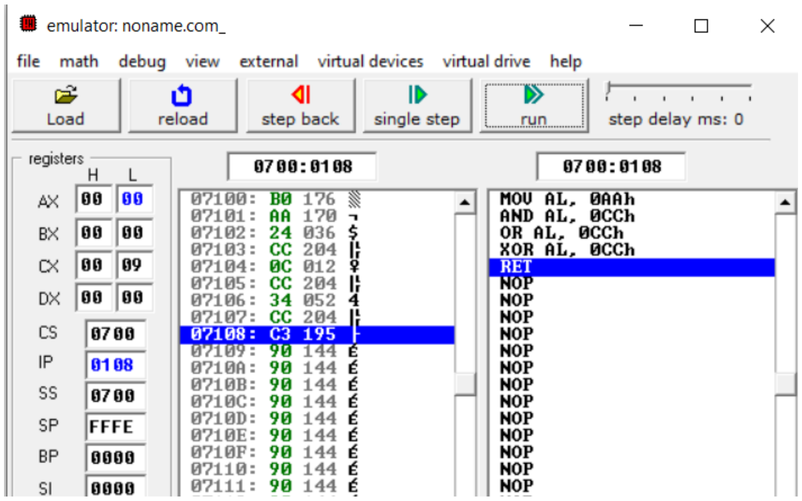
Bit Manipulation Instructions

Exercise 1: Performing Bitwise AND, OR, and XOR Operations

- **Task**: How to perform bitwise AND, OR, and XOR operations in assembly language?
- **Code**:

```
ORG 100h ; this directive required for a simple 1 segment .com program.
MOV AL, 10101010b; Load binary value into AL register
AND AL, 11001100b; Perform bitwise AND operation
OR AL, 11001100b; Perform bitwise OR operation
XOR AL, 11001100b; Perform bitwise XOR operation
RET ; returns to operating system.
```

- **Code Result**:



```
     H   L
AX   00  00
BX   00  00
CX   00  09
DX   00  00
CS   0700
IP   0108
SS   0700
SP   FFFE
BP   0000
SI   0000
```

```
0700:0108
07100: B0 176 ▒
07101: AA 170 ¬
07102: 24 036 $
07103: CC 204 ╠
07104: 0C 012 ♀
07105: CC 204 ╠
07106: 34 052 4
07107: CC 204 ╠
07108: C3 195 ├
07109: 90 144 É
0710A: 90 144 É
0710B: 90 144 É
0710C: 90 144 É
0710D: 90 144 É
0710E: 90 144 É
0710F: 90 144 É
07110: 90 144 É
07111: 90 144 É
```

```
0700:0108
MOV AL, 0AAh
AND AL, 0CCh
OR AL, 0CCh
XOR AL, 0CCh
RET
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
```

After executing the code:

- The AL register will contain the result of the bitwise AND operation, which is **10001000b**.
- The AL register will contain the result of the bitwise OR operation, which is **11101110b**.
- The AL register will contain the result of the bitwise XOR operation, which is **00000000b**.

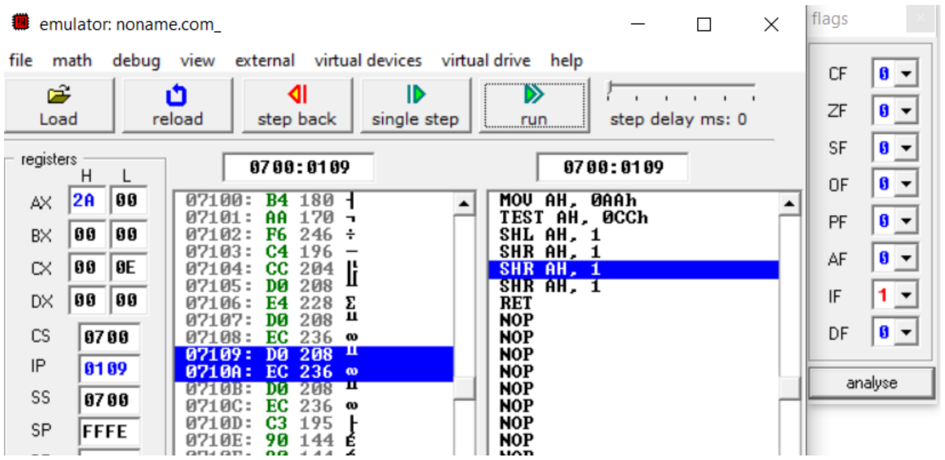
Effect on Flags:
The AND, OR, and XOR instructions may modify the Carry, Zero, Sign, Overflow, and Parity flags depending on the result of the operations.

Exercise 2: Performing Bitwise TEST and Shift Left (SHL)/Shift Right(SHR) Operations

- **Task**: How to perform bitwise TEST and SHL/SHR operations in assembly language?
- **Code**:

```
ORG 100h ;
MOV AH, 10101010b; Load binary value into AH register
TEST AH, 11001100b; Perform bitwise TEST operation
SHL AH, 1 ; Perform bitwise shift left operation
SHR AH, 3 ; Perform bitwise shift right operation
RET ; returns to operating system.
```

- **Code Result**:



```
registers
    H   L
AX  2A  00
BX  00  00
CX  00  0E
DX  00  00
CS  0700
IP  0109
SS  0700
SP  FFFE
```

```
0700:0109
07100: B4 180 ┤
07101: AA 170 ¬
07102: F6 246 ÷
07103: C4 196 ─
07104: CC 204 ╠
07105: D0 208 ╨
07106: E4 228 Σ
07107: D0 208 ╨
07108: EC 236 ∞
07109: D0 208 ╨
0710A: EC 236 ∞
0710B: D0 208 ╨
0710C: EC 236 ∞
0710D: C3 195 ├
0710E: 90 144 É
```

```
0700:0109
MOV AH, 0AAh
TEST AH, 0CCh
SHL AH, 1
SHR AH, 1
SHR AH, 1
SHR AH, 1
RET
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
```

```
CF 0
ZF 0
SF 0
OF 0
PF 0
AF 0
IF 1
DF 0
```

After executing the code:

- The TEST instruction updates the flags register based on the result of the bitwise AND operation between AH and the immediate value.
- The AH register will contain the result of the bitwise shift left operation and shift right operation, which is 00101010b.

Effect on Flags:

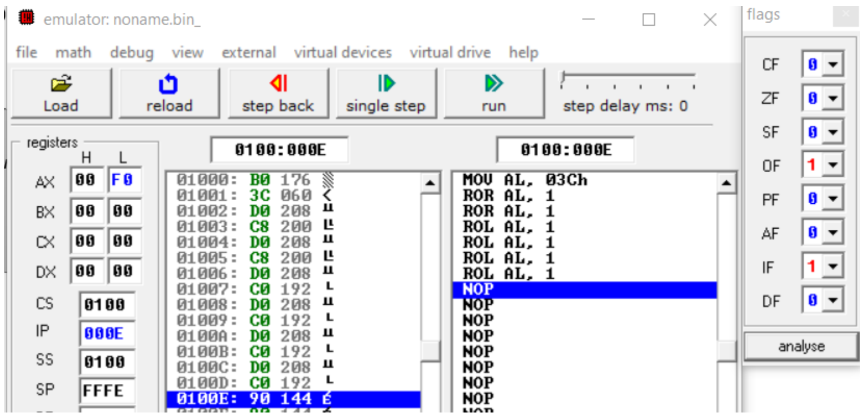
The TEST instruction may modify the Zero and Sign flags based on the result of the AND operation. The SHL instruction may modify the Carry, Zero, Sign, and Overflow flags depending on the result of the operation.

Exercise 3: Performing Bitwise Rotate Right (ROR) and Rotate Left (ROL) Operations

- **Task**: How to perform bitwise rotate right (ROR) and rotate left (ROL) operations in assembly language?
- **Code**:

```
ORG 100h ;
MOV AL, 00111100b; Load binary value into AL register
ROR AL, 2; Perform bitwise rotate right operation
ROL AL, 4 ; Perform bitwise rotate left operation
RET ; returns to operating system.
```

- **Code Result**:



```
registers
     H   L
AX  00  F0
BX  00  00
CX  00  00
DX  00  00
CS  0100
IP  000E
SS  0100
SP  FFFE
```

```
0100:000E

01000:  B0 176 ░
01001:  3C 060 <
01002:  D0 208 ╨
01003:  C8 200 ╚
01004:  D0 208 ╨
01005:  C8 200 ╚
01006:  D0 208 ╨
01007:  C0 192 └
01008:  D0 208 ╨
01009:  C0 192 └
0100A:  D0 208 ╨
0100B:  C0 192 └
0100C:  D0 208 ╨
0100D:  C0 192 └
0100E:  90 144 É
```

```
0100:000E

MOV AL, 03Ch
ROR AL, 1
ROR AL, 1
ROL AL, 1
ROL AL, 1
ROL AL, 1
ROL AL, 1
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
```

```
CF 0
ZF 0
SF 0
OF 1
PF 0
AF 0
IF 1
DF 0
```

After executing the code:

- The AL register will contain the result of the bitwise rotate right operation, which is 00001111b.
- The AL register will contain the result of the bitwise rotate left operation, which is 11110000b.

Effect on Flags:

The ROR and ROL instructions may modify the Carry, Zero, Sign, and Overflow flags depending on the result of the operations.