**Лекція № 1**

**Низькорівневе програмування: розуміння, мови, переваги та недоліки. Вступ у асемблер. Машинний код, компілятори та інтерпретатори.**

Низькорівневе програмування є однією з ключових областей в IT-сфері, оскільки це дозволяє безпосередньо взаємодіяти з апаратним забезпеченням комп'ютера. Мови низькорівневого програмування забезпечують максимальну ефективність використання ресурсів системи та управління ними.

**Що таке низькорівневе програмування?**

*Низькорівневе програмування - це процес створення програм з використанням мов програмування, які дозволяють розробникам мати прямий доступ до апаратного забезпечення комп'ютера, такого як процесор, пам'ять та інші пристрої.*

Ці мови дозволяють ефективніше керувати ресурсами системи та оптимізувати продуктивність програм. Вони використовуються для створення програм, які вимагають максимальної продуктивності та точного контролю апаратних ресурсів. Такі мови програмування мають менший рівень абстракції, ніж високорівневі мови, і надають розробникам більш прямий контроль над апаратурою.

Низькорівневе програмування застосовується в багатьох областях, де потрібний жорсткий контроль над пристроєм, таких як військова та космічна сфери, медичні додатки, інженерія та ін. Незважаючи на складність використання низькорівневих мов, вони є базовими для побудови комп'ютерних технологій та присутні у кожному комп'ютері.

На початку комп'ютерної ери програмування проводилося на машинному коді, який складався з комбінації двох цифр: «1» та «0». Це була найпримітивніша мова, яка забезпечувала контроль над залізом комп'ютера на найнижчому рівні. З розвитком технологій та збільшенням складності програмного забезпечення написання коду двійковою мовою стало неможливим. Тому були створені більш зручні для розробки програмного забезпечення низькорівневі мови програмування, які все ще дозволяли розробникам керувати апаратним забезпеченням комп'ютера.

**Які мови програмування належать до низькорівневих?**

Низькорівневе програмування передбачає більш прямий доступ до апаратури комп'ютера, що вимагає фахівця вищої кваліфікації. До таких мов відносяться, наприклад, Assembler, C і C++. До низькорівневих мов програмування також відносять: Rust, Ada, Fortran та Pascal.

Деякі з цих мов були розроблені в часи, коли комп'ютери були менш потужними та вимагали більш низькорівневого програмування. Незважаючи на це, вони все ще використовуються сьогодні у різних сферах, таких як автомобільна промисловість, мікроконтролери та промислове керування.

**Відмінності від високорівневих мов програмування**

Низькорівневе програмування вимагає від програміста розуміння, як працює комп'ютер на найнижчому рівні, тобто на рівні заліза. Для цього необхідно мати великий досвід та знання. Низькорівневі мови програмування дозволяють безпосередньо управляти ресурсами комп'ютера та робити програми швидше та ефективнішими. Ці мови знаходяться ближче до апаратного рівня та дозволяють програмісту точніше керувати апаратними компонентами.

Високорівневі мови програмування використовують абстрактніші конструкції, які дозволяють фахівцям сконцентруватися на логіці програми, а не на деталях реалізації, вони створені для людей, щоб розробка була легша і зрозуміліша. Вони не розуміються безпосередньо комп'ютером, тому використовуються спеціальні програми для конвертації в низькорівневу мову. Приклади високорівневих мов програмування включають Python, Java, JavaScript, PHP, C# та інші. Вони використовуються для створення різних продуктів, включаючи програми, ігри та веб-програми.

**Переваги**

Переваги низькорівневих мов програмування:

1. Висока швидкість виконання коду.
2. Максимальний контроль апаратного забезпечення, що робить їх ідеальними для написання системного програмного забезпечення, драйверів та операційних систем.
3. Робота із ресурсами. Розробники можуть мати прямий доступ до ресурсів комп'ютера, таких як пам'ять та процесор, що дозволяє їм працювати більш ефективно та швидко.
4. Підвищена портованість. Низькорівневі мови можуть бути скомпільовані на багато різних архітектур процесорів, що дозволяє використовувати код на різних платформах.
5. Можливість оптимізації. Розробники можуть написати код, який можна оптимізувати для конкретного апаратного забезпечення, що дозволяє досягти максимальної продуктивності програми.
6. Розробка низькорівневого програмного забезпечення надійніша і безпечніша, оскільки програма працює близько до апаратного рівня і не залежить від інших програмних шарів.

**Недоліки**

Недоліки низькорівневих мов програмування:

1. Вони складніші у використанні, ніж високорівневі мови програмування, т.к. вимагають глибшого розуміння апаратного забезпечення. Відповідно програміст повинен контролювати кожну машинну команду, яку генерує його програма.
2. Недостатня переносимість. Через те, що кожен комп'ютер має унікальну апаратну конфігурацію, програми можуть працювати тільки на певному комп'ютері або апаратному забезпеченні.
3. Невелика економічна цінність. Низькорівневі мови програмування займають невелику нішу в програмному ринку і використовуються головним чином для розробки операційних систем, драйверів пристроїв та програмного забезпечення для систем, що вбудовуються.

Низькорівневі мови програмування зазвичай не мають такого широкого вибору готових бібліотек і фреймворків, як високорівневі мови. Однак, за рахунок більш прямого доступу до апаратних ресурсів, вони дозволяють написати більш продуктивний та оптимізований код, що також може вважатися їхньою перевагою.

Ви, напевно, помітили деякі протиріччя щодо підвищеної портованості і, навпаки, недостатньою переносимістю, пояснимо. З одного боку, можливість компіляції низькорівневих мов на різні архітектури процесорів може підвищити їх портованість, тобто можливість використання коду на різних платформах. З іншого боку, деякі програми, написані низькорівневими мовами, можуть містити інструкції, які прив'язані до конкретної апаратури та не можуть бути виконані на інших комп'ютерах чи апаратних конфігураціях. Це може обмежувати перенесення низькорівневих програм, особливо якщо вони написані для конкретної архітектури процесора або використовують специфічні апаратні можливості.

**Чи є перспектива у низькорівневих мов програмування**

Перспектива низькорівневих мов програмування досі викликає певні питання серед розробників. З одного боку, існує безліч додатків і систем, які все ще використовуються низькорівневими мовами програмування. З іншого боку, з розвитком технологій з'являються нові високорівневі мови програмування. Вони значно спрощують процес розробки та скорочують час написання коду.

Однак, не слід забувати, що низькорівневі мови програмування відіграють важливу роль у розробці операційних систем, драйверів, компіляторів та інших критично важливих компонентів. Низькорівневі мови програмування також широко використовуються в багатьох галузях, де продуктивність є критично важливим фактором, наприклад, в аерокосмічній та автомобільній промисловості.

**Асемблер. Вступ**

У сучасному світі програмування мова асемблера була витіснена навіть із такої традиційно «ассемблерної» області, як програмування мікроконтролерів — маленьких однокристалльних ЕОМ, призначених для вбудовування у всіляку техніку, від пральних машин і стільникових телефонів до літаків і турбін на електростанціях. У більшості випадків прошивки мікроконтролерів зараз пишуть мовою Сі, і лише невеликі вставки виконують мовою асемблера; те саме вірно і для ядер операційних систем, і для інших завдань, в яких необхідна прив'язка до можливостей конкретного процесора.

Звичайно, зовсім обійтися без фрагментів мовою асемблера поки що не виходить. Окремі асемблерні модулі, а також асемблерні вставки в текст іншими мовами присутні і в ядрах операційних систем, і в системних бібліотеках тієї ж мови, Сі (та інших мов високого рівня); в особливих випадках програмісти мікроконтролерів теж змушені відмовлятися від Сі та писати «на асемблері», щоб, наприклад, заощадити дефіцитну пам'ять. Так, досить популярний мікроконтролер АТtiny4 має всього 16 байт оперативної пам'яті та 512 байт псевдопостійної пам'яті для зберігання коду програми; за таких умов мові асемблера практично немає альтернативи. Однак такі випадки рідкісні навіть у світі мікроконтролерів, більшість з яких надає програмісту далеко не такі жорсткі умови. Мало кому з вас, що вивчають нині програмування, мовою асемблера, доведеться хоча б один раз за все життя вдатися до цих навичок на практиці.

Тож навіщо витрачати час на вивчення асемблера? Адже все одно це ніколи не стане в нагоді? Так це виглядає лише на перший погляд; при більш уважному розгляді питання вміння мислити в термінах машинних команд не просто «знадобиться», воно виявляється життєво необхідним будь-якому професійному програмісту, навіть якщо цей програміст ніколи не пише мовою асемблера. Якою б мовою ви не писали свої програми, потрібно хоча б приблизно уявляти, що саме робитиме процесор, щоб виконати вашу високу волю. Якщо такого уявлення немає, програміст починає бездумно застосовувати всі доступні операції, не знаючи, що насправді робить. Тим часом одне привласнення на добре знайомому нам Паскалі може виконатися за мільярдну частку секунди, але може й розтягнутися на помітний час, якщо, наприклад, нам прийде в голову надавати один одному великі масиви рядків.

З більш складними мовами програмування справи ще цікавіше: присвоювання, записане мовою Сі++, може виконатися в одну машинну команду, а може спричинити мільйони команд". Два, таких присвоювання записуються в програмі абсолютно однаково (знаком рівності), але цей факт ніяк нам не допоможе: ми не зможемо адекватно оцінити ресурсомісткість тієї чи іншої операції, не розуміючи, як і що при цьому робить процесор. Програміст, що не має досвіду, роботи на рівні команд процесора, просто не знає, що насправді творить; вставляючи в програму мовою високого рівня ті чи інші операції, він часто не здогадується, наскільки складне завдання ставить перед процесором. На виході ми маємо величезні програми, що бентежать своєю низькою ефективністю — наприклад, додатки для автоматизації офісного документообігу, яким «тісно» в чотирьох гігабайтах оперативної пам'яті і для яких виявляється «надто повільним» процесор, що на багато порядків перевершує за швидкодією суперкомп'ютери вісімдесятих років.

Досвід показує, що професійний користувач комп'ютерів, чи то програміст чи системний адміністратор, може собі дозволити щось не знати, але в жодному разі не може дозволити собі не розуміти, як влаштована обчислювальна система на всіх її рівнях, від електронних логічних схем до громіздких прикладних програм. Не розуміючи чогось, ми залишаємо у своєму тилу місце для «відчуття магії»: на якомусь майже підсвідомому рівні нам продовжує здаватися, що щось там нечисто і без парочки чарівників з чарівними паличками не обійшлося. Таке відчуття для професіонала неприпустимо категорично: навпаки, професіонал зобов'язаний розуміти і відчувати, що пристрій, з яким він має справу, створено такими ж людьми, як він сам, і нічого «чарівного» або «непізнаваного» собою не представляє.

Якщо ставити за мету досягнення такого рівня розуміння, виявляється зовсім не важливо, яку конкретну архітектуру та мову якого конкретного асемблера вивчати. Знаючи одну мову асемблера, ви зможете почати писати будь-якою іншою, витративши дві-три години (а то й менше) на вивчення довідкової інформації; Але головне тут у тому, що, вміючи мислити в термінах машинних команд, ви завжди знатимете, що насправді відбувається при виконанні ваших програм.

Незважаючи на все вищесказане, слід пояснити вибір конкретної архітектури. Матеріал лекцій заснований на системі команд процесорів сімейства х86, причому ми будемо використовувати 32-бітний варіант цієї архітектури, так звану систему команд i386. 32-бітові комп'ютери сімейства х86 вже майже повністю витіснені комп'ютерами на основі 64-бітних процесорів?, але, на щастя, ці процесори можуть виконувати програми в 32-бітному режимі. Саму 64-бітну систему команд ми вивчати не будемо, тому є певна причина. Усі наявні описи цієї системи будуються за принципом перерахування її відмінностей від 32-бітового випадку; виходить, що нам у будь-якому випадку спочатку потрібно вивчити 32-бітну систему команд. Але вивчивши її, ми вже досягнемо своєї мети – отримаємо. досвід, працювати мовою асемблера; подальший перехід до 64-бітового випадку можливий, але для наших цілей дещо надмірний.

Доречно сказати кілька слів щодо вибору конкретного асемблера. *Як відомо, для роботи з процесорами сімейства, х86 використовується два основних підходи до синтаксису мови асемблера – це синтаксис АТ&Т та синтаксис Intel.* Одна і та ж команда процесора представляється в цих синтаксичних системах зовсім по-різному: наприклад, команда, яка в синтаксисі Intel виглядає як

mov еах, [а+еах]

у синтаксисі АТ&Т записуватиметься так:

mov1 а(%еdх), %еaх

Навчальні посібники, орієнтовані на програмування мовою асемблера в синтаксисі, все-таки існують, тоді як синтаксис АТ&Т описується виключно у спеціальній (довідковій) технічній літературі, яка має на меті навчання.

Крім того, необхідно враховувати й багаторічне панування середовища MS DOS як платформу для аналогічних навчальних курсів; все це дозволяє назвати синтаксис Intel суттєво звичним і краще підтримуваним.

**Машинний код, компілятори та інтерпретатори**

Практично всі сучасні цифрові обчислювальні машини працюють за одним і тим самим принципом. Обчислювальний пристрій (власне сам комп'ютер) складається з центрального процесора, оперативної пам'яті та периферійних пристроїв. Найчастіше всі ці компоненти підключаються до загальної шини.

*Оперативна пам'ять складається з однакових комірок пам'яті, кожна з яких має свій унікальний номер, який називається адресою.*

Комірка містить кілька (найчастіше — вісім) двійкових розрядів, кожен із яких може бути у одному з двох станів, зазвичай позначених як «нуль» і «одиниця». Це дозволяє комірці як єдиному цілому перебувати в одному з 2n станів, де n - кількість розрядів в комірці; так, якщо розрядів вісім, то можливих станів комірки буде 28 = 256, або, інакше кажучи, комірка може "пам'ятати" число від 0 до 255.

У центральному процесорі є кілька регістрів - схем, що нагадують комірки пам'яті; оскільки регістри знаходяться безпосередньо в процесорі, вони працюють дуже швидко, але їхня кількість обмежена, так що використовувати регістри слід для зберігання самої необхідної інформації. Процесор має здатність копіювати дані з оперативної пам'яті в регістри і назад, проводити над вмістом регістрів арифметичні та інші операції; у деяких випадках операції можна проводити і безпосередньо з даними в комірках пам'яті, не копіюючи їх вміст у регістри.

Кількість інформації, що може обробити процесор в один прийом (за одну команду), називається машинним словом. Розмір більшості регістрів точно дорівнює машинному слову. У сучасних системах машинне слово, як правило, більше, ніж комірка пам'яті; так, машинне слово процесора Pentium становить 32 біти, тобто чотири восьмибітові комірки пам'яті.

Процесор Pentium та його сучасні «спадкоємці», з якими ми збираємося працювати, є черговими представниками лінійки процесорів х86, і ранні представники цієї лінійки (до 80286 включно) були 16-розрядними, тобто їхнє машинне слово становило 16 біт. Програмісти, які працюють із цими процесорами на рівні машинних команд, звикли називати «словом» саме два байти інформації, а чотири байти називали «подвійним словом». Коли з виходом чергового процесора розмір слова подвоївся, програмісти стали змінювати звичну термінологію, що породжує певну плутанину. До цього питання нам доведеться повернутися, коли ми дістанемося програмування на рівні команд центрального процесора.

Програма записується в оперативну пам'ять у вигляді цифрових кодів, що позначають ті чи інші операції, а спеціальний регістр процесора, який називається лічильником, команд визначає, з якого місця пам'яті потрібно брати інструкції до виконання. Процесор виконує цикл обробки команд - тобто витягує з пам'яті код чергової команди, збільшує лічильник команд, дешифрує вилучений код, виконує приписані дії, знову витягує з пам'яті чергову команду, і так до нескінченності.

Подання програми, що складається з кодів машинних команд і, як наслідок, «зрозуміле» центральному процесору, називається машинним кодом. Процесор легко може дешифрувати такі коди команд, але людині їх запам'ятати дуже важко, тим більше, що в багатьох випадках потрібне число доводиться обчислювати, підставляючи в певні місця кодові ланцюжки двійкових бітів. Ось, наприклад, два байти, що записуються в шістнадцятковій системі як 01 D8 (відповідні десяткові значення - 1,216) позначають на процесорах Pentium команду «взяти число з регістру ЕАХ, додати до нього число з регістра ЕВХ, результат додавання помістити назад в регістр ЕАХ.». Запам'ятати два числа 01 D8 нескладно, але різних команд на процесорі Pentium — кілька сотень, та до того ж тут сама команда — тільки перший байт (01), а другий (D8) нам доведеться обчислити в розумі, згадавши, що молодші три біти в цьому байті позначають перший регістр, наступні три біти позначають другий регістр, а найстарші два біти тут повинні дорівнювати одиницям, що означає, що обидва операнда є регістрами. Знаючи, що номер регістру EАХ - 0, а номер регістра EВХ - 3, ми тепер можемо записати двійкове уявлення нашого байта: 11011000, що дає в десятковому записі 216 , а у шістнадцятковій системі — шукане D8.

Якщо нам потрібно буде освіжити в пам'яті шматочок нашої програми, написаний два дні тому, то щоб його прочитати, нам доведеться вручну розкладати байти на складові їх біти і, звіряючись з довідником, згадувати, що ж яка команда робить. Якщо програміста змусити складати програми ось таким способом, нічого корисного він не напише за все своє життя, тим більше що в будь-якій, навіть найменшій, але практично застосовній програмі таких команд буде кілька тисяч, ну а найбільші програми складаються з десятків мільйонів машинних команд.

Працюючи з мовами програмування високого рівня, такими як Паскаль, Сі, Лисп та інших., програмісту надається можливість написати програму у вигляді, зрозумілому і зручному для людини, а не для центрального процесора. Процесор таку програму виконати вже не може, і щоб таки змусити виконатися програмі, написаної мовою високого рівня, доводиться вдатися до одного з двох можливих способів трансляції програми. *Ці два способи називаються компіляцією та інтерпретацією.*

У першому випадку застосовують *комлілятор - програму, що приймає на вхід текст програми мовою програмування високого рівня та видає еквівалентний машинний код.*

Слід підкреслити, що *компілятор - це теж програма, написана якоюсь мовою програмування*; зокрема, наш компілятор Паскаля сам, як це не дивно, написаний на Паскалі, а для компіляції кожної наступної версії його творці використовують попередню версію свого компілятора.

Другий спосіб виконання програм, написаних мовами високого рівня, називається інтерпретацією. *Програма-інтерпретатор прочитує з файлу вихідний текст і виконує запропоновані цим текстом дії крок за кроком, нічого нікуди взагалі не перекладаючи.* Сучасні інтерпретатори зазвичай для зручності та збільшення швидкості виконання створюють якесь своє внутрішнє уявлення нашої програми, але з машинним кодом таке уявлення не має нічого спільного.

Зазначимо, що з одного шкільного підручника до іншого кочує абсолютно маячне слово, що нібито інтерпретатор перекладає програму в машинний код покроково і тут же виконує цей код. Якщо вам щось подібне говорили або ви самі таке прочитали в якійсь книжці — не вірте. На сучасних машинах і при роботі під управлінням сучасних операційних систем так взагалі неможливо зробити, оскільки навіть якщо ми сформуємо якийсь машинний код, записати його ми можемо тільки в область пам'яті, яка призначена для зберігання даних, а передати управління на цей код ми не зможемо, оскільки виконання команд з області, призначеної для даних, також заборонено. Загалом, навіть якби ми хотіли так зробити, то нічого не вийшло б — але найцікавіше, що так робити зовсім не потрібно! Інтерпретатор сам по собі є програмою, і він може просто виконати потрібні дії, нікуди, ні в який код їх не переводячи. Та дивна людина, яка першою вигадала фразу про покрокове переведення в машинний код з негайним виконанням, сама, очевидно, ніколи жодних програм не писала, інакше така божевільна ідея просто не спала б їй на думку.

Цікаво зауважити, що в наші часи межі між компіляцією та інтерпретацією поступово розмиваються. Багато компіляторів переводять програму не в машинний код, а в якесь проміжне уявлення, яке зазвичай називають «байт-кодом». саме так працюють компілятори мов Java та С#. У багатьох випадках компілятори породжують такий код, який вже під час виконання інтерпретує якусь частину проміжного представлення програми. З іншого боку, інтерпретатори з метою підвищення ефективності роботи теж переводять програму на проміжне уявлення, яке, щоправда, потім виконують самі. Існують компілятори, які начебто створюють окремий файл, але при уважному розгляді цього файлу виявляється, що в нього повністю включений інтерпретатор внутрішнього подання програми плюс саме це уявлення.

Так чи інакше, при компілюваному виконанні часто використовуються елементи інтерпретації, а при інтерпретованому - елементи компіляції, і постає питання про те, де між цими двома підходами провести кордон і чи існує такий кордон взагалі. Ми ризикнемо запропонувати досить просту відповідь на це питання, що дозволяє в кожному конкретному випадку безперечно сказати, чи має місце компілюване або інтерпретоване виконання. Інтерпретатор під час виконання інтерпретованої програми сам змушений перебувати у пам'яті, тоді як компілятор потрібен лише на етапі компіляції, а виконуватися програма може без його участі.

Програмування на мовах високого рівня зручне, але, на жаль, не завжди можна застосувати. Причини можуть бути різні. Наприклад, мова високого рівня може не враховувати деякі особливості конкретного процесора або програміста може не влаштовувати той конкретний спосіб, яким компілятор реалізує ті чи інші конструкції вихідної мови за допомогою машинних кодів. У цих випадках доводиться відмовитися від мови високого рівня та скласти програму у вигляді конкретної послідовності машинних команд. Однак, як ми вже бачили, складати програму безпосередньо в машинних кодах дуже складно. І тут на допомогу приходить програма, яка називається асемблером.

*Асемблер - це окремий випадок компілятора: програма, яка приймає на вхід текст, що містить умовні позначення машинних команд, зручні для людини, і перекладає ці позначення в послідовність відповідних кодів машинних команд, зрозумілих процесору.* На відміну від самих машинних команд, їх умовні позначення, які називають також мнемоніками, запам'ятати порівняно легко. Так, команда з наведеного раніше прикладу, код якої, як ми з деякими труднощами з'ясували, дорівнює 01 D8, в умовних позначеннях виглядає так:

add eax, ebx

Тут нам уже не треба заучувати числовий код команди і обчислювати в умі позначення операндів, досить запам'ятати, що словом аdd позначається додавання, причому в таких випадках завжди першим після позначення команди стоїть перший доданок (не обов'язково регістр, це може бути область пам'яті), другим — другий доданок (це може бути і регістр, і область пам'яті, і число), а результат завжди заноситься на місце першого доданку. Мова таких умовних позначень (мнемонік) називається мовою асемблера.

Програмування мовою асемблера докорінно відрізняється від програмування мовами високого рівня. На мові високого рівня (на тому ж Паскалі) ми задаємо лише загальні вказівки, а компілятор вільний сам вибирати, яким саме способом їх виконати - наприклад, якими регістрами та комірками пам'яті скористатися для зберігання проміжних результатів, який застосувати алгоритм для виконання якоїсь нетривіальної інструкції та т.д. З метою оптимізації швидкодії компілятор може переставити інструкції місцями, замінити одні на інші, аби результат залишився незмінним. У програмі на мові асемблера ми абсолютно однозначно і недвозначно вказуємо, з яких машинних команд складатиметься наша програма, і ніякої свободи асемблер (на відміну від компілятора високого рівня) не має.

На відміну від машинних кодів, мнемоніки доступні для людини, тобто програміст може працювати з мнемоніками без особливих зусиль, але це не означає, що програмувати мовою асемблера просто. Дія, на опис якої ми витратили б один оператор мови високого рівня, може вимагати десятка, якщо не сотні рядків на мові асемблера, а в деяких випадках і більше. Справа тут у тому, що компілятор мови високого рівня містить більший набір готових «рецептів» вирішення невеликих завдань, що часто виникають, і надає всі ці «рецепти» програмісту у вигляді зручних високорівневих конструкцій; асемблер же нічого подібного не містить, так що в нашому розпорядженні виявляються лише можливості процесора.

Цікаво, що для одного процесора може існувати декілька різних асемблерів. На перший погляд це здається дивним, адже не може один і той же процесор працювати з різними системами машинних кодів (так званими системами команд). Насправді нічого дивного тут немає, досить згадати, що таке насправді асемблер.

Система команд процесора, зрозуміло, не може змінитися (якщо не взяти інший процесор). Однак для одних і тих же команд можна придумати різні позначення; так, вже знайома нам команда add eax, ebx в позначеннях, прийнятих у компанії АТ&Т, буде виглядати як add1 %ebx, %eax — і менімоніка інша, і регістри не так позначені, і операнди не в тому порядку, хоча машинний код, що отримується, зрозуміло, суворо той самий - 01 D8. Крім того, при програмуванні мовою асемблера ми зазвичай пишемо не тільки мнемоніки машинних команд, а й директиви, що є прямими наказами асемблеру. Дотримуючись таких вказівок, асемблер може зарезервувати пам'ять, оголосити ту чи іншу мітку видимої з інших модулів програми, перейти до генерації іншої секції програми, обчислити якийсь вираз і навіть сам (слідуючи, зрозуміло, нашим вказівкам) «написати» фрагмент програми мовою асемблера, який сам потім і обробить. Набір таких директив, що підтримуються асемблером, також може бути різним, як за можливостями, так і за синтаксисом.

Оскільки асемблер — це програма, написана цілком звичайними програмістами, ніхто не заважає іншим програмістам написати свою програму-асемблер, що часто і відбувається.