Асемблер

Улаз асемблера је текстуална датотека, њу чине асемблерске инструкције и асемблерске директиве. Излаз асемблера је објектна датотека, бинарни фајл у `ELF` форату.

Садржај излазне датотеке је одређен садржајем улазне датотеке. Асемблер чита улазну датотеку линију по линију, и постепено генерише бинарни садржај излазне датотеке. Када асемблер у улазној датотеци прочита асемблерску инструкцију, у излазну датотеку њен мешиниски код. У зависности од директиве коју је прочитао у улазној датотеци, асемблер може емитовати неки садржај у излазну датотеку (садржај одређен самом директивом), или на неки начин изменити свој понашање.

Наредне информације се односе на ГНУ-ов асемблер.

Секције

Излазна датотека је подељена на секције. Секција представља континуалан и недељив низ бајтова у излазној датотеци. Свакој секцији је придружено име (низ карактера).

Током рада асемблер одржава вредност `location-counter`, која представља тренутни померај у текућој секцији.

Директива `.section name`, означава почетак секције са именом `name`. Ова директива за текућу секцију поставља секцију која је њом направљена, а `location-counter` поставља на 0.

Свака излазна датотека ће имати секције `.text`, `.data` и `.bss`, чак и ако нису уведене директивом `.section`.

Садржај улазне датотеке који је наведен пре прве `.section` директиве припада `.text` секцији.

Директива `.end`, означава крај улазне датотеке, асемблер ће игнорисати садржај наведен након `.end` директиве.

Директиве за дефинисање садржаја излазне датотеке

`.byte expr {, expr}`

У излазну датоеку емитује бајт за сваки од наведених израза, вредности тих бајтова одговарају вредностима наведених израза.

Директиве `.2byte`, `.4byte`, `.8byte`

Имају ефекат сличан директиви `.byte`, само је податак који емитују друге ширине.

`.ascii string, {, string}`

За сваки од наведених стринг литерала у излазну датотеку емитују његов саджај енкодован по `ASCII` табели.

`.asciz string, {, string}`

Ефекат сличан директиви `.ascii`, једина разлика је што не крају сваког стринга додат 0 бајта. Директива `.string` је синоним.

`.align value[, fill]`

Помера `location-counter` на прву следећу вредност дељиву са `value`. У излазну датотеку емитује онлико бајтова за колико је померен `location-counter`. Вредност сваког од тих бајтова је `fill`, ако `fill` није наведен њихова вредност је 0.

`.skip size[, fill]`

У излазну датотеку емитује `size` бајтова, вредност сваког од тих бајтова је `fill`. Ако `fill` није наведен њихова вредност је 0.

Остале директиве за дефинисање садржаја излазне датотеке

Постоји више инструкција за дефинисање садржаја излазне датотеке, њихов ефекат је сличан директиви `.byte`, али ширина вредности коју емитују зависи од архитектуре за коју је излазна датотека намењена. Следећа табеле дефинише ширине за процесор `x86-64`:

Директива	Ширина(у битима)
`.short`	16
`.int`	32
`.long`	32
`.hword`	16
`.word`	16
`.dword`	32
`.quad`	64

Симболи

Симболи представљају начин за означавање одређених мемориских локација излазне датотеке. Сваки симбол припада некој секцији. У објектној датотеци вредност симбола је померај меморијске локације коју означава од почетка секције којој симболе припада. У извршној датотеци вредност симбола је виртуелна адреса на коју ће меморијска локација коју симбол означава бити учитана.

Постоји више начина за дефинисање симбола:

`symbol:`

Овако дефинисан симбол припада текућој секцији, и има вредност једнаку `location-counter`-у.

`.equ symbol, expr`

Симбол дефинисан `.equ` директивом има вредност једанку вредности наведеног израза. Секција којој овако дефинисан симбол припада се одређује на основу наведног израза. Израз може да садржи друге симболе, константе и уобичајене аритметичке операције. Директива `.set` је синоним.

Видљивост симбола

Симбол је локални или глобални. Глобални симболи ће бити доступни у процесу повезивања (линковања), локални неће. Симбол је локални уколико није експлицитно означен као глобални коришћњем директиве `.global` праћеном именом симбола.

Спољашњи симболи

У улазној датотеци се могу корситити симболи који у њој нису дефинисани, такви симболи се називају спољашњим. Спољашњи симбол се уводи директивом `.extern` праћеном именом симбола. Спољашњи симболи припадају `UND` секцији, секцији без садржаја, којој припадају сви спољашњи симболи. Сваки симбол које се користи у улазној датотеци, а није у њој дефинисан је имплицитно спољшњни, није неопходно увести га `.extern` директивом.

Величина симбола

Симболу је придружена величина, која говори о ширини записа на који симбол реферише. Величина симбола се поставља директивом `.size` на следећи начин: `.size symbol, size`.

Вредност симбола

У објектној датоци вредност симбола је померај меморијске локације коју означава од почетка секције којој припада.

У извршној дататоци вредност симбола је виртуелна адреса на коју ће бити учитана меморијска локција коју означава.

Током процеса линковања вредност симбола се може променити.

Пример 1: Линкер спаја објектне фајлове `A.o` и `B.o`, и производи објектни фајл `C.o`. У објектном фајлу `A.o` дефинисан је глобални симбол `a` који припада `.text` секцији и има вредност `0` (меморијска локација коју симбол означава се налази на почетку секције). Величина `.text` секције у објектном фајлу`A.o` је `100`.

У `.text` секцији објектног фајла `B.o` дефинисан је глобални симбол `b` са вредношћу `0`.

Линкер `.text` секцију излазног објектног фајла добија спајањем .text секција улазних објектних фајлова, тако што прво наведе садржај `.text` секције објектног фајла `A.o` па затим садржај `.text` секције објектног фајла `B.o`.

Симболи `a` и `b` ће припади `.text` секцији објектног фајла `C.o`. Меморијска локација коју означава симбол `a`, ће бити на померају 0 у новој `.text` секцији, па он задржава своју вредност. Меморијска локација коју означава симбол `b` је на померају 100 у новој `.text` секцији, па ће он сада имати вредност 100.

Пример 2: Линкеру је дат задатак да од објектног фајла `A.o` направи извршну датотеку. У `.text` секцији ја дефинисан симбол `a` са вредношћу 100. Кроз линкерску скрипту је дефинисано да `.text` секцију треба учитина почевиши од виртуелне адресе `40000`. У извршној датотеци вредност симбола `a` ће бити `40100`.

Апсолутна секција (`ABS`)

Којој секцији припада симбол `.equ A, 10`? Симбол `A` не представља померај у некој секцији, његова вредност је констата. Ово се не уклата у дефиницију симбола. Да би се омогућили константе симболе, уведена ја апсолутна секција (`ABS`). `ABS` је секција без садржаја, којој припадају сви константи (апсолутни симболи). Вредност симбола апсолутне секције се не мења у процесу линковања.

Индекс класификације симбола

Индекс класификације предстаља механизам за одређивање којој секцији припада симбол дефинисан `.equ` директивом.

Неки изрази нису валидни као део `.equ` директиве.

Пример 1: Нека је `x` симболе секције `X` и нека је `y` симбол секције `Y`. И нека је симбол `z` дефинисан као `.equ z, x + y`. Ова `.equ` директива није валидна. Није валидна јер би вредност овакво дефинисаног симбола зависила од две скеције. Зависила би од положаја секције `X` и положаја секције `Y`. Другим речима: овакав симбол није могуће доделити ниједној секцији.

Пример 2: Нека су `x1` и `x2` симболи секције `X`. Тада ће симбол `y` дефинисан помоћу `.equ y, x1 - x2` бити апсолутни, јер његова вредност неће зависити од полажаја секције `X`, као ни од положаја било које друге секције. Јер његова вредност представља растојање између две меморијске локације једне секције, а секција је недељива у порцесу линковања.

Индекс класификације се користу у алгоритму којим се одређује којој секцији припада симбол дефинисан `.equ` директивом. Алгоритам се састоји од следећих корака (уведена је претпоставка да се у изразу појављују само сабирање и одузимање):

Корак 1: Израз који је наведен у `.equ` директиви се трансформише на следећи ничин: Свако појављивање константе или апсолутног симбола се мења `0`. Свако појављивање нормалног симбола се мења са `1.section`, где је `section` име секције којој симбол припада. Свако појављивање спољашњег симбола се мења са `1.unique(symbol)`, где је `symbol` име симбола.

Корак 2: Трансформисани израз се своди на минимални облик припном следећа два правила, `a.section + b.section = (a+b).section` и `a.section - b.section = (a-b).section`.

Корак 3: Ако је минимални облик израза `0` симбол је апсолутни. Ако је минимални облик израза `1.section` симбол припада секцији `section`. Ако минимални облик израза има било коју другу вредност директива није валидна.

Примери: Нека су `x1` и `x2` симболи секције `X`, `y1` и `y2` симболи секције `Y`, а `z` спољашњи симбол.

`.equ A, x1-x2+10`, `1.X - 1.x + 0 = 0`, симбол `A` је апсолутни.

`.equ B, x1+x2`, `1.X + 1.X = 2.X`, симбол `B` није добро дефинисан.

`.equ C, x1-y1`, `1.X - 1.Y`, симбол `C` није добро дефинисан.

`.equ D, x1+y1-x2-y2`, `1.X + 1.Y - 1.X - 1.Y = 0`, симбол `D` је апсолутни.

`.equ E, x2-x1+z`, `1.X + 1.Y - 1.unique(z) = 1.unique(z)`, симбол `E` припада припада истој секцији као и симбол `z`.

Директива `.comm`

Директива `.comm symbol, size[, align]` дефинише `common` симбол величине `size` и поравнања `align`. За `common` симбол није алоцирана меморија ни у једној секцији. Када линкер у више објектиних фајлова пронађе више `common` симбола истог имена, неће пријавити грешку као у случају обичних симбола. Него ће тих више симбола спојити у један, чија ће величина бити једнака најећој величини од симбола датог имена, а његово поравнање ће ме бити максимално поравнање симбола датог имена.

Уколико линкер пронђе нормалан симбол (симбол који није `common`) и више `common` симбола истог имена. Све `common` симболе ће спојити са дефинисаним симболом. Величину и поравнање одређује нормални симбол.

Када линкер производи извршну датотеку, а у улазним објектним фајловим пронађе `common` симбол који се није везао за неки нормалан симбол, он у `.bss` секцији алоцира простор за тај симбол у складу са његовом величином и поравнањем.

Уколико се `C` изворни код преводи `gcc`-ом са опциојом `-fcommon`, глобалне променљиве који нису иницијализоване неће бити смештене у `.bss` секцију, већ ће бити `common` симболи одговарајуће величине и поравнања. Ово може бити извор проблема па је зато подразумевана опција `-fno-common`.

Употреба симбола у директивама и инструкцијама

Симболи се могу користити у директивама и инструкција навођењем њиховог имена. Нека је дефинисан сумбол `S`, њему је придржена вредност у складу са његовом дефиницијом. Примери употреба у инструкцијама и директивама:

```
mov rax, S
```

Ефекат ове инстукције биће читање у регистар `rax` садржаја меморије са адресе не коју реферше симбол `S` (адресе једнакој вредности симбола `S`).

```
mov rax, OFFSET S
```

Ефекат ове инструкције је учитавање вредности симбола `S` у регистар `rax`.

```
mov rax, OFFSET S[rbx]
```

Ефекат ове инструкције биће читање у регистар `rax` садржаја меморија са адресе добије када се сабере вредност симбола `S` са садржајем регистра `rbx`.

```
mov rax, OFFSET S[rip]
```

Ефекат ове инструкције биће читање садржаја мемријске локације на коју реферише симболе `S` у регистар `rax`. Али начин адресирања ће бити `rip`-релативно. У извршној датотеци у наведној инструкцији `OFFSET S` ће бити замењено померајем између меморијске локације на коју реферише `S` и адресе на којој се налази инструкција која следи након ове (вредност регистра rip у тренутку извршавања ове инструкције). Ова употреба `OFFSET S` није конзистента са предхдном, где је `OFFSET S` мењано вредношћу симбола.

```
.8byte S
```

Ефекат ове директиве је емитовање вредност симболе `S` (запис ширине 8 бајта) у излазну датотеку. `OFFSET` се може користит само у контексту инструкција.

```
.byte S
```

Ефекат ове директиве је емитовање вредности симболе `S` (запис ширине 1 бајта) у излазну датотеку.

Употреба `location-counter`-а у инструкција и директивама

Вредност `location-counter`-а је доступна коришћењем симбола `.`. На пример директива `.4byte .` за ефекат имати емитовање тренутне вредности `location-counter`-а. `.` се може посматрати као симбол који има вредност једнак локацији на којој је искоришћен.

Напомена: У инструкцији `mov rax, .`, симбол `.` ће бити замењен адресом на којој почиње машински код посматране инструкције.

Релкациони запис

Неке употребе симбола у директивама и инструкцијам асемблер може да разреши а неке не.

Примери случајева када асемблер може да разреши употребу симбола:

Пример 1: Нека је X симбол дефинисан у `.text` секцији улазне датотеке. У `.text` секцији је наведена инструкција `jmp X`. Како инструкције скока користе `rip`-релативно адресирање, на месту употребе симбола асемблер би требао да унесе померај између `X` и инструкције која следи, ову вредност асемблер може да израчуна, јер се обе локације налазе у истој секцији (њихово растојање се не може мењати у процесу линковања).

Пример 2: Нека је `X` симбол дефинисан следећом директивом `.equ X, 1`. У `.text` је наведена инструкција `mov rax, OFFSET X`. На месту употребе симбола `X` асемблер би требао да наведе његову вредност. Његова вредност је асемблеру позната, и не неће се мењати у процесу линковања, па је асемблер може навести у излазној датотеци.

Примери случајева када асемблер не може да разреши употребу симбола:

Пример 1: Нека је `X` спољашњи симбол, и нека је у `.text` секцији наведена инструкција: `call X`. Како `call` инструкција користи `rip`-релативно адресирање, на месту употребе симбола `X` асемблер би требао да наведе померај од адресе коју означава симбол `X` до наредне инструкције. Ова вредност асемблеру није позната па не може разрешиту ову употребу симбола `X`.

Пример 2: Нека је `X` симбол секције `.data`, и нека је у `.text` наведена инструкција `mov rax, OFFSET X`. Асемблер би требао да на месту употребе симбола `X` наведе виртуелну адресу на коју ће бити учитана меморијска адреса коју означава симбол `X`, која му није позната.

Када асемблер није у могућности да разреши употребу симбола (наведе одговрајућу вредност у излазној датотеци), он генерише релокациони запис. Релокацини запис је одређен следећим:

Место где треба унети вредност, секција и померај у секцији. Тип релокације, који говори о томе у ком облику треба унети вредност (нпр. вредност симбола или `rip`-релативна вредност симбола). Симбол на који се односи релокациони запис и `addend`. `addend` је нумеричка констата која се користи при одређивању вредности коју треба унети (корсти се на више различитих начина па зато има овако неодређен назив).

Неки од типова релокације за архитектуру `x86-64` су:

`R_X86_64_PC32`

Овај тип релокације означава 32-битни запис са вредношћу `S + A - P`, где је `S` вредност симбола (виртуелна адреса), `A` је `addend`, а `P` је адреса на којој се врши релоциони запис.

`R_X86_64_64`

Овај тип релокација означава 64-битни запис са вредносшћу `S + A`, где је `S` вредност симбола, а `A` је `addend`.

`R_X86_64_32`

Овај тип релокација означава 32-битни запис са вредносшћу `S + A`, где је `S` вредност симбола, а `A` је `addend`.

Бележење релокационих записа у излазној датотеци

Релокациони записи су забележени унутар посебних секција (чији садржај генерише сам асемблер), имена тих секција почињу са `.rela`. За сваку секцију за коју постоје релокациони записи биће генерисана одговарајућа секција која саджи податке о њима. На пример, подаци о релокационим записима унутар секције `XYZ`, биће забележени у секцији `.relaXYZ`.

Два примера употребе `addend`-а

Релокациони записи не могу реферисати локалне симболе.

Пример 1: Нека је `X` локални симбол који припада `.data` секцији, и нека је његова вредност `100` (померај меморијске локације коју означава од почетка секције којој припада). И нека је у `.text` секцији наведена инструкција `mov rax, OFFSET X`. На месту употребе симбола `X`, асемблер би требао да унесе виртуелну адресу меморијске локације коју симбол `X` означава, она асемблеру није позната па он генерише релокациони запис. Како релокациони записи не могу реферисати локалне симболе, послужићемо се следећим триком при генерисању овог релокационог записа. У табелу симбола се додаје симбол `.data` који припада `.data` секцији и има вредност `0` (означава почетак `.data` секције). Генерише се релокациони запис `R_86_64_32` који реферише симбол `.data`, а `addend` је постављен на померај симбола `X`, тј. `100`. Вредност која ће бити унета када се овај релокациони запис буде разрешавао биће `S + A` где је `S` виртелна адреса почетка `.data` секције, а `A` је `addend`. Како је за вредност `addend`-а постављен померај симбола `X` унутар `.data` секције, биће унета адреса симбола `X`.

Пример 2: Нека је `X` глобални симбол (може бити реферисан у релокационом запису) `.data` секције. И нека је `.text` секцији наведена инструкција `mov rax, OFFSET X[rip]`. На месту употребе симбола `X` асемблер би требао да наведе растојање између адресе наредне инструкције и адресе коју означава симбол `X`, како ове две локације не припадају истој секцији асемблер не може да одреди ову вредност (јер се растојање између секција може мењати у процесу линковања) он генерише релокациони запис типа `R_86_64_PC32` који реферише на симбол `X`. На месту посматране инструкције асемблер генерише следећи садржај (овде записан у хексадецималном облику): `48 8b 05 00 00 00 00`. Прва три бајта представљају операциони код инструкције `mov`, код дестинационог регистера, и начина адресирања за изворишни операнда, док последња 4 бајта би требала да представљају померај симбола `X` од наредне инструкције. На месту помераја асемблер је емитовао нуле јер му померај није познат. Место на које ће се односити релокациони запис биће адреса првог нула бајата од ова четири. По дефиницији релокационог записа `R_X86_64_PC32`, вредност која ће бити унета у тренутку разрешавања је `S + A - P`. Вредност коју је потребно унети је `S - address(next-instruction)`, важи `address(next-instruction) = P + 4`. Ако се `addend` постави на `-4`, онда израз из дефиниције релокационог записа постаје `S + A - P = S - 4 - P = S - (P + 4) = S - address(next-instruction)`, што је тачно оно што је потребно.

Имплементација асемблера

Двопролазни асемблер

Двопролазни асемблер пролази два пута кроз садржај улазне датотеке.

Током првог пролаза формира табелу симболу, табелу секција, одређује величину сваке од секција. Не генрише садржај секција, само рачуна њихову величину (нпр. када прочита асемблерску инструкцију, израчунање ширину записа машинске инструкције која јој одговара, и за ту вредност увећати `location-counter`).

Током другог пролазка генрише бинарни садржај секција, и потребне релокационе записе користећи табелу симбола која је формирана током првог пролаза.

Једнопролазни асемблер

Једнопролазни асемблер само једном пролази кроз садржај улазне датотеке. Истоврено формира табелу симбола и генерше садржај секција. То доводи до проблема обраћања унапред. У улазној датотеци се може наћи инструкција која користи неку лабелу пре него што она дефинисана. Асемблер у том тренутку не може да одреди саджај илазне датотеке, јер му није позната вредност искоришћеног симбола, али не жели ни да постави релокациони запис, јер ће му вредност симбола бити позната у каснијем тренутку.

Овај проблем је решен увођењем табеле обраћања унапред. Свака употреба симбола који се не налази у табели симбола у том тренутку генерише нови улаз у табели обраћа унапред, кроз који је забележено који симбол је реферисан, у којој секцији, на ком офсету у секцији, и коју вредност треба уписати на наведну локацију (сви подаци потребни за генерисање релокационог записа).

Након пролазка кроз улазну датотеку, асемблер пролази кроз табелу обраћања унапред. Сваки улаз у табели обраћања унапред ће бити разрешен (уношењем одговарајуће вредности у излазну датотеку), или ће бити генерисан релокациони запис који одговара том обраћању.