Стековая виртуальная машина и модельный ассемблер

Коновалов А.В.

14 февраля 2023 г.

Стековая виртуальная машина

Стековая виртуальная машина

Мы будем рассматривать компиляцию различных высокоуровневых абстракций на примере модельного компьютера, т.к. рассмотрение реальных процессоров сильно усложнит изложение.

Модельная виртуальная машина отражает интересующие нас характеристики реальных машин с точки зрения генерации кода: у неё есть основные команды (арифметика, переходы), аппаратная поддержка стека, и, соответственно, вызовы функций, минимальное количество регистров.

Система команд вдохновлена книгой *Свердлов С.З. Языки программирования и методы трансляции: Учебное пособие — СПб.: Питер, 2007. — 638 с.: ил.*, страница 352.

Чего здесь нет

Чего в данной машине намеренно нет:

- нет вещественной и беззнаковой арифметики,
- поддержки многозадачности и многопоточности,
- байтовой адресации памяти,
- ввод-вывод существенно абстрагирован,
- команд с операндами (все команды получают аргументы со стека),
- и многого другого.

Всё это только усложнит изложение и отвлечёт от центральной темы курса: представления высокоуровневых абстракций ЯП на уровне машины.

Память и регистры

Память машины состоит из слов, каждое из которых хранит знаковое целое число в диапазоне как минимум от $-2\,000\,000$ до $+2\,000\,000$. Размер памяти как минимум миллион слов. Адреса слов — неотрицательные целые числа. Первые 256 слов недоступны для обращения (чтения, записи и исполнения) — диапазон заблокирован для ловли нулевых указателей.

Инструкция процессора занимает ровно одно слово.

Адресное пространство машины обычно выглядит следующим образом:

- 256 недоступных для использования слов,
- код загруженной программы,
- свободная память,
- стек.

Стек растёт «вниз» — от старших адресов ко младшим, как на большинстве архитектур.

Регистры процессора

Машина имеет четыре регистра — два специальных и два общего назначения:

- **IP** — указатель инструкций (instruction pointer),
- **SP** — указатель стека (stack pointer),
- **FP** — указатель базы (frame pointer),
- **RV** — возвращаемое значение (return value).

Специальные регистры `IP` и `SP` меняются при выполнении команд, регистры общего назначения `FP` и `RV` пользователь может использовать по своему усмотрению.

Специальные регистры IP и SP

Регистр IP содержит адрес инструкции, которая будет выполнена после исполнения следующей инструкции (инструкция JMP — фактически инструкция присваивания этому регистру). При выполнении большинства инструкций он инкрементируется, инструкции перехода его меняют соответстсвующим способом.

Регистр SP содержит адрес слова, хранящего значение, добавленное на стек последним. Большинство инструкций его изменяют, т.к. либо снимают операнды со стека, либо добавляют что-либо в стек.

При добавлении значения в стек регистр SP декрементируется, при удалении — инкрементируется. Допустимо считать, что после удаления значения со стека значение $\mathtt{SP} - 1$ не определено.

Регистры общего назначения FP и RV

С точки зрения машины, содержимое этих регистров может быть произвольным, никакие команды, кроме GETFP, SETFP, GETRV, SETRV, к ним не обращаются. И при написании кода на ассемблере пользователь может ими распоряжаться, как угодно.

Однако, в сгенерированном коде у них будет своя, конкретная роль (которая отражена в названии):

- регистр FP мы будем использовать как регистр, хранящий адрес текущего фрейма (кадра) стека,
- регистр RV мы будем использовать для хранения возвращаемого значения (его можно передавать и через стек, но (а) это приводит к громоздкому коду и (б) возвращаемое значение зачастую передаётся через регистр и на практике).

Обозначения (1)

При описании команд машины мы будем использовать следующие обозначения:

- Размер памяти машины мы будем обозначать как $SIZE$,
- Саму память будем рассматривать как массив из M слов, допустимые к обращению ячейки — от $M[256]$ до $M[SIZE - 1]$.
- Операции $++$ и $--$ означают тоже самое, что и в языке Си.

В частности:

- Помещение значения V на стек описывается формулой $M[-- \texttt{SP}] := V$.
- Извлечение значения V со стека описывается как $V := M[\texttt{SP} ++]$.
- На каждом шаге машина считывает слово $M[\texttt{IP} ++]$ и его интерпретирует — на следующем шаге по умолчанию выполнится следующее слово (если сама команда не является командой перехода, меняющей IP.

Обозначения (2)

Инструкции мы будем описывать в нотации, принятой для языка FORTH:

`имя инструкции (код): … стек до → … стек после`

или

`имя инструкции (код): … стек до → … стек после (доп. действие)`

При записи мы считаем, что стек растёт слева-направо, многоточие означает ту часть стека, которая не меняется.

Пример:

`ADD (-1) : … x y → … x + y`

Инструкция, которая в ассемблере обозначается `ADD`, имеет код операции -1, снимает со стека два числа и помещает на вершину их сумму.

Помещение констант на стек

Любое неотрицательное слово в памяти машина рассматривает как команду добавления этого слова на стек:

`x (x ≥ 0) : … → … x`

Для помещения на стек отрицательного значения следует сначала поместить его абсолютное значение, а затем поменять знак командой `NEG`.

Поэтому коды операций других команд будут отрицательными.

Арифметические и логические операции

Арифметические операции:

```
ADD (-1) : … x y → … x + y
SUB (-2) : … x y → … x - y
MUL (-3) : … x y → … x × y
DIV (-4) : … x y → … x / y
MOD (-5) : … x y → … x % y
NEG (-6) :   … x → … -x
```

Битовые операции:

```
BITAND  (-7) : … x y → … x & y
BITOR   (-8) : … x y → … x | y
BITNOT  (-9) :   … x → … ~x
LSHIFT (-10) : … x y → … x << y
RSHIFT (-11) : … x y → … x >> y
```

Команда `SUB` вычитает из подвершины стека вершину, аналогично `DIV` и `MOD`.

Операции для работы со стеком

```
DUP   (-12)  :           … x → … x x
DROP  (-13)  :           … x → …
SWAP  (-14)  :         … x y → … y x
ROT   (-15)  :       … x y z → … y z x
OVER  (-16)  :         … x y → … x y x
DROPN (-17)  : … x1 … xN N → …              (SP += N + 1)
PUSHN (-18)  :           … N → … x1 … xN    (SP -= N - 1)
```

Команда PUSHN кладёт на вершину стека N неопределённых значений. Она будет использоваться для резервирования памяти под локальные переменные.

Операции для работы с памятью

```
LOAD (-19) :   … a → … M[a]
SAVE (-20) : … a v → …        (M[a] := v)
```

Команда LOAD снимает со стека адрес и кладёт на стек слово, лежащее по данному адресу.

Команда SAVE снимает со стека сначала присваиваемое значение, потом адрес и слову с данным адресом присваивает значение.

Операции для работы с регистрами (1)

```
GETIP (-21) :    … → … IP
SETIP (-22) : … a → …          (IP := a)
GETSP (-23) :    … → … SP
SETSP (-24) : … a → ?          (SP := a)
GETFP (-25) :    … → … FP
SETFP (-26) : … a → …          (FP := a)
GETRV (-27) :    … → … RV
SETRV (-28) : … a → …          (RV := a)
```

Вспомним, что … в описаниях команд означает ту часть стека, которую команды не меняют. После выполнения команды SETSP указатель стека меняется, а значит содержимое стека может оказаться каким угодно, поэтому после стрелки изображён знак вопроса.

Операции для работы с регистрами (2)

Инструкция GETIP помещает на стек адрес следующей инструкции — точно также, как и инструкция CALL (см. далее).

Инструкция GETSP помещает на стек значение, которое имел SP до выполнения этой инструкции. Т.е. её можно описать псевдокодом

$$prev := \mathsf{SP}; M[--\ \mathsf{SP}] := prev$$

Команда сравнения CMP

```
CMP (-29) : … x y → sgn
```

Команда сравнения снимает со стека два числа и кладёт на вершину

- -1, если $x < y$,
- 0, если $x = y$,
- $+1$, если $x > y$.

Заменить её операцией вычитания нельзя, т.к. возможно арифметическое переполнение.

Инструкции переходов

```
JMP  (-22) :    … a → …     (IP := a)
JLT  (-30) : … x a → …      (IP := x < 0 ? a : IP + 1)
JGT  (-31) : … x a → …      (IP := x > 0 ? a : IP + 1)
JEQ  (-32) : … x a → …      (IP := x = 0 ? a : IP + 1)
JLE  (-33) : … x a → …      (IP := x ≤ 0 ? a : IP + 1)
JGE  (-34) : … x a → …      (IP := x ≥ 0 ? a : IP + 1)
JNE  (-35) : … x a → …      (IP := x ≠ 0 ? a : IP + 1)
```

Инструкции переходов

```
JMP  (-22) :    … a → …     (IP := a)
JLT  (-30) : … x a → …      (IP := x < 0 ? a : IP + 1)
JGT  (-31) : … x a → …      (IP := x > 0 ? a : IP + 1)
JEQ  (-32) : … x a → …      (IP := x = 0 ? a : IP + 1)
JLE  (-33) : … x a → …      (IP := x ≤ 0 ? a : IP + 1)
JGE  (-34) : … x a → …      (IP := x ≥ 0 ? a : IP + 1)
JNE  (-35) : … x a → …      (IP := x ≠ 0 ? a : IP + 1)
```

Внимательные студенты заметили, что код операции -22 уже встречался как код инструкции `SETIP`. Да, одну и ту же команду можно записывать двумя разными мнемониками.

Вызов процедур и возврат

```
CALL (-36) :              … a → … IP  (IP := a)
RETN (-37) : … x1 … xN a N → …        (IP := a)
```

Инструкция `CALL` фактически обменивает вершину стека и регистр `IP` — $M[\texttt{SP}] \leftrightarrow \texttt{IP}$.

Инструкция `RETN` увеличивает регистр стека на $N + 2$ и осуществляет переход на подвершину. Она будет использоваться при возврате из функции, имеющей формальные параметры.

Ввод-вывод, остановка

```
IN   (-38) :   … → … c
OUT  (-39) : … c → …
HALT (-40) : … a → …
```

Инструкция `IN` заставляет считать интерпретатор из стандартного ввода символ и положить его код (ASCII или Unicode) на вершину стека.

Инструкция `OUT` записывает символ с кодом `c` на стандартный вывод.

Инструкция `HALT` завершает работу интерпретатора с кодом возврата `a`.

Запуск виртуальной машины

Виртуальная машина стартует в следующем состоянии:

- Начиная с адреса 256 загружены слова пользовательской программы,
- Регистр `IP` хранит значение 256, т.е. будет выполнена самая первая инструкция.
- Регистр `SP` хранит значение $SIZE$ — прочитать со стека ничего нельзя, записать можно.
- Значение регистров `FP` и `RV` не определено.

Модельный ассемблер

Синтаксис модельного ассемблера

Программа на модельном ассемблере состоит из меток и слов.

- **Метки** определяют новые имена.
- **Слова** компилируются в слова целевой программы.

Метки и слова разделяются пробельными символами — пробелами, табуляциями и знаками перевода строк.

Комментарии начинаются со знака `;` и продолжаются до конца строки.

Синтаксис слов

Слово может быть записано как в виде целого числа, так и при помощи идентификатора.

- **Число** начинается с необязательного знака `+` или `-`, за которым следует непустая последовательность десятичных цифр. Примеры чисел: `10`, `+65`, `-40`.
- **Идентификатор** начинается с латинской буквы или знака `_` (прочерка) и состоит из латинских букв, цифр, знаков `_` (прочерк) и `-` (дефис). Примеры идентификаторов: `Loop`, `HALT`, `_Read_Number`, `write-line`.

Метки и предопределённые имена

Идентификатор является синонимом некоторого числа.

Идентификатор может быть встроенным, либо определённым при помощи метки.

Встроенными идентификаторами являются все мнемоники команд (`ADD`, `SETIP`, `JMP`, `HALT` и т.д.) и идентификатор `PROGRAM_SIZE`.

Метка записывается как знак `:`, после которого записывается идентификатор — значением идентификатора будет адрес в памяти следующего слова.

Уже определённый идентификатор переопределить нельзя, поэтому метки не могут повторяться и не могут совпадать с мнемониками и иметь имя `PROGRAM_SIZE`.

Алгоритм ассемблирования (1)

Ассемблирование выполняется в два прохода:

1. Вычисление адресов меток.
2. Генерация кода.

Алгоритм ассемблирования (2) — первый проход

- Инициализируется глобальная переменная — счётчик слов (назовём его $current$) значением 256.
- Для каждого обнаруженного слова счётчик $current$ инкрементируется.
- Каждой обнаруженной метке сопоставляется текущее значение счётчика $current$. Если эта метка уже получала значения (была встроенной или была определена ранее) — синтаксическая ошибка.
- Идентификатору `PROGRAM_SIZE` сопоставляется окончательное значение $current$.

Алгоритм ассемблирования (3) — второй проход

- Инициализируется глобальная переменная — счётчик слов (назовём его $current$) значением 256.
- Для каждого слова в память по адресу $M[current]$ записывается значение этого слова: числа записываются как есть, для идентификаторов выписываются их значения; переменная $current$ инкрементируется.
- Метки игнорируются.

Пример программы

Следующая программа печатает `Hello!`:

```
72 OUT 101 OUT 108 OUT 108 OUT 111 OUT 33 OUT 0 HALT
```

Вместо мнемоник можно использовать и константы:

```
72 -39 101 -39 108 -39 108 -39 111 -39 33 -39 0 -40
```