Лекция 1. Система команд модельного компьютера

Александр Коновалов

10.02.2022

Общее описание машины

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

Модельная машина будет обладать стековой архитектурой, большинство команд будут безаргументыми, т.к. данные будут брать со стека.

Рассматриваемая машина будет примерно соответствовать описанию модельной машины из книги

Свердлов С.З. Языки программирования и методы трансляции: Учебное пособие — СПб.: Питер, 2007. — 638 с.: ил.

страница 352.

Память процессора для простоты будет не байтовая, а состоять из слов. Каждое слово хранит знаковое целочисленное значение в диапазоне как минимум от $-2\,000\,000$ до $+2\,000\,000$.

Процессор обладает следующими регистрами:

- **CP** — code pointer, указатель кода, содержит адрес инструкции, которая будет выполнена следующей,
- **SP** — stack pointer, указатель стека, содержит адрес слова, хранящего адрес последнего элемента, добавленного на стек,
- **BP** — base pointer, указатель базы, используется для организации стековых фреймов.

Машина однопоточная, средств для параллельного программирования не предусмотрено, т.к. в курсе этой темы мы касаться не будем.

Память организована в виде массива (обозначим его `M`) из N слов, ячеек памяти с адресами от 0 до $N-1$. Параметр N указывается пользователем при запуске виртуальной машины.

Стек растёт от старших адресов ко младшим. При помещении значения на стек сначала декрементируется SP, затем значение записывается в слово M[SP]. Операцию помещения значения на стек условно можно описать на Си следующей строчкой:

```
M[--SP] = x;
```

Операция извлечения значения со стека, соответственно, наоборот, считывает значение M[SP] и инкрементирует регистр SP:

```
x = M[SP++];
```

Считаем, что при запуске машины программа длиной P слов загружается в память по адресам от 0 до $P-1$, $CP=0$, $SP=N$, $BP=0$.

Каждая инструкция процессора занимает одно слово. Коды операций со значениями большими либо равными нулю кладут на стек свой код, отрицательные — являются кодами соответствующих инструкций.

Инструкции процессора

Инструкции мы будем описывать в нотации, традиционно используемой для стекового языка FORTH:

```
имя-команды (код операции) : ... стек до --> ... стек после
```

Верхушку стека мы будем располагать справа. Например, команду сложения мы опишем так:

```
ADD (-1) : ... x y --> ... x+y
```

Эта запись означает, что код инструкции сложения — -1, символически мы её будем записывать как ADD, со стека она снимает два числа и кладёт их сумму.

Арифметические команды

```
ADD (-1) : ... x y --> ... x+y
SUB (-2) : ... x y --> ... x-y
DIV (-3) : ... x y --> ... x/y
MOD (-4) : ... x y --> ... x%y
MUL (-5) : ... x y --> ... x*y
NEG (-6) : ... x --> ... -x

BITAND (-7) : ... x y --> ... x&y
BITOR  (-8) : ... x y --> ... x|y
BITNOT (-9) : ... x --> ... ~x
```

Команда вычитания SUB вычитает вершину стека из подвершины, аналогично деление и вычисление остатка.

Операции со стеком

```
DUP (-10)   : ... x       --> ... x x
DROP (-11)  : ... x       --> ...
SWAP (-12)  : ... x y     --> ... y x
ROT (-13)   : ... x y z   --> ... y z x
OVER (-14)  : ... x y     --> ... x y x
DROPN (-35) : ... x1…xN N --> ...       (SP += N + 1)
PUSHN (-36) : ... N       --> x1…xN     (SP -= N - 1)
```

Команда PUSHN просто смещает указатель стека, что выглядит как добавление на стек N неопределённых значений. Она будет использоваться для резервирования памяти под локальные переменные.

Операции работы с памятью

```
READ (-15)  : ... a   --> ... M[a]
WRITE (-16) : ... a v --> ...           (M[a] := v)
```

Здесь M[a] означает обращение к ячейке памяти по адресу a. Таким образом, команда READ снимает со стека адрес и кладёт на него значение слова по этому адресу, команда WRITE снимает со стека адрес (подвершина) и значение (вершина) и записывает в память по данному адресу данное значение.

Операции управления

```
CMP  (-17) : ... x y --> ... sgn
JMP  (-18) : ... a   --> ...          (CP := a)
JLT  (-19) : ... x a --> ...          (CP := x < 0 ? a : CP+1)
JGT  (-20) : ... x a --> ...          (CP := x > 0 ? a : CP+1)
JEQ  (-21) : ... x a --> ...          (CP := x == 0 ? a : CP+1)
JLE  (-22) : ... x a --> ...          (CP := x ⩽ 0 ? a : CP+1)
JGE  (-23) : ... x a --> ...          (CP := x ⩾ 0 ? a : CP+1)
JNE  (-24) : ... x a --> ...          (CP := x ≠ 0 ? a : CP+1)
CALL (-25) : ... a   --> ... CP+1     (CP := a)
RETN (-26) : ... x1…xN a N --> ...    (CP := a)
HALT (-32) : ... x   --> ...          (завершает программу)
```

Команда CMP используется для сравнения двух чисел, кладёт на стек

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

Команда JMP осуществляет безусловный переход на адрес, который она берёт со стека. Команды условного перехода Jcond сравнивают с нулём подвершину стека и если условие выполняется, осуществляют переход на адрес, лежащий на верхушке.

Команда вызова подпрограммы CALL осуществляет переход на команду, адрес которой снимает со стека, на стек вместо неё кладёт адрес инструкции, следующей за CALL.

Команда RETN снимает со стека адрес возврата, число N, затем N слов и осуществляет переход на адрес возврата. Команда будет использоваться для возврата из подпрограммы с параметрами. При вызове подпрограммы на стек будут класться аргументы, затем она будет вызываться инструкцией CALL, которая положит на верхушку адрес возврата. Для возврата удобно иметь отдельную инструкцию, которая и вернёт управление в точку вызова, и удалит из стека лишние значения.

Инструкция HALT завершает работу виртуальной машины. Значение на верхушке стека соответствует коду возврата программы (т.е. возвращаемому значению `main()` языка Си).

Операции работы с регистрами

```
GETSP (-27) : ...    --> ... SP
SETSP (-28) : ... a --> ...              (SP := a)
GETBP (-29) : ...    --> ... BP
SETBP (-30) : ... a --> ...              (BP := a)
GETCP (-31) : ...    --> ... CP
```

Инструкции используются для получения и записи значений соответствующих регистров. Команда SETCP называется JMP :-)

Ввод-вывод

```
IN  (-33) : ...    --> ... c
OUT (-34) : ... c --> ...
```

Команда IN считывает кодовую точку Юникода со стандартного ввода. Команда OUT выводит на стандартный вывод символ Юникода с кодовой точкой, снятой со стека.

Язык ассемблера для модельного компьютера

Нотация для описания грамматики

Для записи синтаксиса языка ассемблера мы будем использовать РБНФ в нотации Вирта. Правила в грамматике имеют вид

```
Грамматика = Правило { Правило }.
Правило = НЕТЕРМИНАЛ "=" ПраваяЧасть ".".
```

где ПраваяЧасть — последовательность одной или нескольких альтернатив, разделённых знаком |, правило заканчивается на точку:

```
ПраваяЧасть = Альтернатива { "|" Альтернатива }.
```

Альтернатива — последовательность нуля или нескольких термов:

```
Альтернатива = { Терм }.
```

Терм — имя нетерминала, имя терминала, вложенное правило, необязательная или повторяющаяся часть:

```
Терм = ТЕРМИНАЛ | НЕТЕРМИНАЛ | ВложенноеПравило
     | НеобязательнаяЧасть | ПовторяющаясяЧасть.
```

Вложенное правило, необязательная и повторяющаяся части отличаются лишь скобками:

```
ВложенноеПравило = "(" ПраваяЧасть ")".
НеобязательнаяЧасть = "[" ПраваяЧасть "]".
ПовторяющаясяЧасть = "{" ПраваяЧасть "}".
```

Терминальные символы записываются либо КАПСЛОКОМ, либо в двойных кавычках, нетерминальные символы записываются в ВерблюжьемРегистре (CamelCase).

Синтаксис языка ассемблера

Синтаксис языка ориентирован на простую генерацию кода из программ на Рефале.

Программы пишутся в свободном синтаксисе, т.е. переводы строк считаются такими же пробельными символами, как и пробелы.

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

Программа на языке ассемблера представляет собой последовательность инструкций и определений констант:

```
Программа := { Инструкция | ОпределениеКонстанты }.
```

Инструкция записывается как арифметическое выражение, его значение вычисляет программа ассемблера и соответствующее число записывает в память виртуальной машины по очередному адресу. Запись `@` в выражении означает количество ранее скомпилированных инструкций (или, что тоже самое, адрес следующей инструкции).

Чаще всего, инструкции будут либо символическими константами, либо целочисленными литералами.

```
Инструкция = Слагаемое.
Слагаемое = ЧИСЛО | ИМЯ | "(" Выражение ")" | "@".
Выражение = ["-"] Слагаемое { ("+" | "-") Слагаемое }.
```

Любая реализация ассемблера должна поддерживать выражения со знаками + и −, в качестве расширения можно допустить и операции умножения и деления с более высоким приоритетом.

Определение константы начинается со знака двоеточия, после которого располагается имя константы и необязательное указание её значения.

```
ОпределениеКонстанты = ":" ИМЯ ["=" Слагаемое].
```

Если значение константы не указано, то подразумевается количество ранее скомпилированных инструкций. Таким образом, определение константы без явного значения определяет метку в коде. Можно сказать, что определение константы вида `:имя` эквивалентно `:имя=@`.

Имена кодов операций — это предопределённые константы с соответствующими значениями. Т.е. можно считать, что любая программа неявно начинается с преамбулы вида

```
:ADD = -1     :SUB = -2     :DIV = -3    :MOD = -4    :MUL = -5
:BITAND = -6  :BITOR = -7   :BITNOT = -8 :DUP = -9    :DROP = -10
....
```

Целые числа записываются как последовательности десятичных цифр с необязательным знаком:

```
ЧИСЛО = ["+"|"-"] ЦИФРА { ЦИФРА }.
ЦИФРА = "0" | "1" | ... | "9".
```

Имена — последовательности латинских букв и цифр, начинающихся с буквы, чувствительны к регистру, прочерк считается буквой:

```
ИМЯ = БУКВА { БУКВА | ЦИФРА }.
БУКВА = "A" | ... | "Z" | "a" | ... | "z" | "_".
```

Семантика

Компиляция программы на ассемблере осуществляется в два прохода:

- вычисление констант,
- порождение кода.

На первом проходе вычисляются значения всех констант, при этом, если значение константы определяется выражением (имеет вид `:ИМЯ = выражение`), то в выражении можно ссылаться только на константы, определённые ранее по тексту. Код при этом не генерируется, только подсчитывается количество инструкций для вычисления значений меток и значений `@`.

На втором проходе вычисляются значения инструкций и записываются в память виртуальной машины. Т.к. к этому моменту значения всех констант вычислены, можно в инструкциях использовать константы, определённые ниже по тексту.