ГЛАВА 1

ВВЕДЕНИЕ

1.1. Система обозначений

Повсюду в книге при описании машинных команд используются выражения, отличающиеся от обычных арифметических выражений. В “компьютерной арифметике” операнды представляют собой битовые строки (или векторы) некоторой фиксированной длины. Выражения компьютерной арифметики похожи на выражения обычной арифметики, но в этом случае переменные обозначают содержимое регистров компьютера. Значение выражения компьютерной арифметики представляет собой строку битов без какой-либо специальной интерпретации; операнды же могут интерпретироваться поразному. Например, в команде сравнения операнды интерпретируются либо как двоичные целые числа со знаком, либо как беззнаковые целые числа. Поэтому, в зависимости от типа операндов, для обозначения оператора сравнения используются разные символы.

Основное отличие между обычными арифметическими действиями и компьютерными сложением, вычитанием и умножением состоит в том, что результат действий компьютерной арифметики приводится по модулю 2^n, где n — размер машинного слова. Еще одно отличие состоит в том, что компьютерная арифметика содержит большее количество операций. Кроме основных четырех арифметических действий, машина способна выполнить множество других команд: логические *и* (*and*), *исключающее или*, *сравнение*, *сдвиг влево* и др.

Если не оговорено иное, везде в книге предполагается, что длина слова равна 32 битам, а целые числа со знаком представляются в дополнительном к 2 коде. Если длина слова иная, такие случаи оговариваются особо.

Все выражения компьютерной арифметики записываются аналогично обычным арифметическим выражениям, с тем отличием, что переменные, обозначающие содержимое регистров компьютера, выделяются полужирным шрифтом (это обычное соглашение, принятое в векторной алгебре). Машинное слово интерпретируется как вектор, состоящий из отдельных битов. Константы также выделяются полужирным шрифтом, если обозначают содержимое регистра (в векторной алгебре аналогичного обозначения нет, так как там для записи констант используется только один способ — указание компонентов вектора). Если константа означает часть команды (например, непосредственно значение в команде сдвига), то она не выделяется.

Если оператор, например “+”, складывает выделенные полужирным шрифтом операнды, то он подразумевает машинную операцию сложения (“векторное сложение”). Если операнды не выделены, то подразумевается обычное арифметическое сложение скалярных величин (скалярное сложение). Переменная x обозначает арифметическое значение выделенной полужирным шрифтом переменной $\boldsymbol{x}$ (интерпретируемое как знаковое или беззнаковое, что должно быть понятно из контекста). Таким образом, если $\boldsymbol{x} = \mathbf{0x80000000}$, а $\boldsymbol{y} = \mathbf{0x80000000}$, то при знаковой интерпретации $x = y = -2^{31}$, $x + y = -2^{32}$ и $\boldsymbol{x} + \boldsymbol{y} = \mathbf{0}$. (Здесь **0x80000000** — шестнадцатеричная запись строки битов, состоящей из одного единичного бита и следующих за ним 31 нулевого бита.)

Биты нумеруются справа налево, причем крайний справа (младший) бит имеет номер 0. Длины бита, полубайта, байта, полуслова, слова и двойного слова равны соответственно 1, 4, 8, 16, 32 и 64 бит.

Небольшие и простые фрагменты кода представлены в виде алгебраических выражений с оператором присваивания (стрелка влево) и при необходимости с оператором `if`. Для таких задач использование математических выражений предпочтительнее, чем составление программы на языке ассемблера.

Более сложные и громоздкие алгоритмы представлены в виде программ на языке C++, причем свойства C++ как объектно-ориентированного языка программирования нигде в книге не используются. По сути, программы написаны на C, но с комментариями в стиле C++. Там, где различия между C и C++ не существенны, подразумевается просто язык C.

Полное описание языка C выходит за рамки данной книги. Однако для тех читателей, которые не знакомы с C, в табл. 1.1 приведено краткое описание основных элементов этого языка [32]. Это описание особенно полезно для читателей, которые не знакомы с C, но знают какой-либо иной процедурный язык программирования. В таблице перечислены также операторы, которые используются в алгебраических и арифметических выражениях этой книги. Все операторы языка C упорядочены по приоритетам: в начале таблицы расположены операторы, имеющие наиболее высокий приоритет, в конце — операторы с самым низким приоритетом. Буква L в столбце “Приоритет” означает, что данный оператор левоассоциативен, а именно:

$$a \bullet b \bullet c = (a \bullet b) \bullet c.$$

Буква R означает, что оператор правоассоциативен. Приведенные в таблице и используемые в книге алгебраические операторы следуют правилам ассоциативности и приоритетам операторов языка C.

В дополнение к перечисленному в табл. 1.1 используется ряд обозначений из булевой алгебры и стандартной математики (соответствующие пояснения приводятся по мере необходимости).

Таблица 1.1. Элементы языка C и алгебраические операторы

Приоритет	**C**	**Алгебра**	**Описание**
	`0x...`	**0x**…, **0b**…	Шестнадцатеричные, двоичные константы
16	`a[k]`		Выбор k-го компонента
16		$\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots$	Различные переменные или выборка битов (зависит от контекста)
16	`f(x,...)`	$f(\boldsymbol{x},\ldots)$	Вычисление функции
16		$\mathrm{abs}(\boldsymbol{x})$	Абсолютное значение (однако $\mathrm{abs}\left(-2^{31}\right) = -2^{31}$)
16		$\mathrm{nabs}(\boldsymbol{x})$	Отрицательное абсолютное значение
15	`x++, x--`		Прибавление, вычитание единицы (постфиксная форма, постинкремент, постдекремент)
14	`++x, --x`		Прибавление, вычитание единицы (префиксная форма, преинкремент, предекремент)
14	(*type name*) `x`		Приведение типа
14 R		$\boldsymbol{x}^k$	*x* в степени *k*

Окончание табл. 1.1

Приоритет	**C**	**Алгебра**	**Описание**
14	`~x`	$\neg x, \overline{x}$	Побитовое *не* (побитовое дополнение до единицы)
14	`!x`		Логическое *отрицание* (`if` x = **0** `then` **1** `else` **0**)
14	`-x`	$-x$	Арифметическое отрицание
13 L	`x*y`	$x * y$	Умножение по модулю размера слова
13 L	`x / y`	$x \div y$	Знаковое целочисленное деление
13 L	`x / y`	$x \overset{u}{\div} y$	Беззнаковое целочисленное деление
13 L	`x % y`	$\mathrm{rem}(x, y)$	Остаток (может быть отрицательным) от деления ($x \div y$) знаковых аргументов
13 L	`x % y`	$\mathrm{remu}(x, y)$	Остаток от деления ($x \overset{u}{\div} y$) беззнаковых аргументов
		$\mathrm{mod}(x, y)$	Приведение x по модулю y к интервалу $[0,\ \mathrm{abs}(y)-1]$; знаковые аргументы
12 L	`x + y, x - y`	$x+y,\ x-y$	Сложение, вычитание
11 L	`x << y, x >> y`	$x \ll y, x \overset{u}{\gg} y$	Сдвиг влево, вправо с заполнением нулями (“логический” сдвиг)
11 L	`x >> y`	$x \overset{s}{\gg} y$	Сдвиг вправо со знаковым заполнением (“арифметический” или “алгебраический” сдвиг)
11 L		$x \overset{rot}{\ll} y,\ x \overset{rot}{\gg} y$	Циклический сдвиг влево, вправо
10 L	`x < y, x <= y` `x > y, x >= y`	$x < y,\ x \le y$ $x > y,\ x \ge y$	Сравнение знаковых аргументов
10 L	`x < y, x <= y` `x > y, x >= y`	$x \overset{u}{<} y,\ x \overset{u}{\le} y$ $x \overset{u}{>} y,\ x \overset{u}{\ge} y$	Сравнение беззнаковых аргументов
9 L	`x == y, x != y`	$x = y,\ x \ne y$	Равно, не равно
8 L	`x & y`	$x \& y$	Побитовое *и*
7 L	`x ^ y`	$x \oplus y$	Побитовое *исключающее или*
7 L		$x \equiv y$	Побитовая *эквивалентность* ($\neg(x \oplus y)$)
6 L	`x | y`	$x \mid y$	Побитовое *или*
5 L	`x && y`	$x \vec{\&} y$	Логическое *и* (`if` x = **0** `then` **0** `else if` y = **0** `then` **0** `else` **1**)
4 L	`x || y`	$x \vec{\mid} y$	Логическое *или* (`if` $x \ne$ **0** `then` **1** `else if` $y \ne$ **0** `then` **1** `else` **0**)
3 L		$x \parallel y$	Конкатенация
2 R	`x = y`	$x \leftarrow y$	Присваивание

Кроме функций “abs”, “rem” и прочих, в книге используется множество других функций, которые будут определены позже.

При вычислении выражения `x < y < z` в языке C сначала вычисляется выражение `x < y`, (результат равен 1, если выражение истинно, и 0, если выражение ложно), затем полученный результат сравнивается с `z`. Выражение $x<y<z$ с операторами сравнения “<” вычисляется как $(x<y)\&(y<z)$.

В языке C есть три оператора цикла: `while`, `do` и `for`. Цикл `while` имеет вид:

`while(`*выражение*`)` *оператор*

Перед выполнением цикла вычисляется *выражение*. Если *выражение* истинно (не нуль), выполняется *оператор*. Затем снова вычисляется *выражение*. Цикл `while` завершается, когда *выражение* становится ложным (равным нулю).

Цикл `do` аналогичен циклу `while`, однако проверочное условие стоит после оператора цикла, а именно:

`do` *оператор* `while(`*выражение*`)`

В этом цикле сначала выполняется *оператор*, затем вычисляется *выражение*. Если выражение истинно, операторы цикла выполняются еще раз, если выражение ложно, цикл завершается.

Оператор `for` имеет вид

`for(`e_1`;`e_2`;`e_3`)` *оператор*

Сначала вычисляется выражение e_1 — как правило, это оператор присваивания (аналог выражения-инициализации). Затем вычисляется e_2 — оператор сравнения (или условное выражение). Если значение условного выражения равно нулю (условие ложно), цикл завершается, если не равно нулю (условие истинно), выполняется *оператор*. Затем вычисляется e_3 (тоже, как правило, оператор присваивания), и вновь вычисляется условное выражение e_2. Таким образом, знакомый всем цикл “do i=1 to n” запишется как `for(i=1; i<=n; i++)` (это один из контекстов использования оператора постинкремента).

1.2. Система команд и модель оценки времени выполнения команд

Чтобы можно было хотя бы грубо сравнивать алгоритмы, представим, что они кодируются для работы на машине с набором команд, подобных современным RISC-компьютерам общего назначения (типа Compaq Alpha, SGI MIPS и IBM RS/6000). Это трехадресная машина, имеющая достаточно большое количество регистров общего назначения — не менее 16. Если не оговорено иное, все регистры 32-разрядные. Регистр общего назначения с номером 0 всегда содержит нули, все другие регистры равноправны и могут использоваться для любых целей.

Для простоты будем считать, что в компьютере нет регистров “специального назначения”, в частности, слова состояния процессора или регистра с битами состояний, например “переполнение”. Не рассматриваются также команды для работы с числами с плавающей точкой, как выходящие за рамки тематики данной книги.

В книге описаны два типа RISC: “базовый RISC”, команды которого перечислены в табл. 1.2, и “RISC с полным набором команд”, в который кроме основных RISC-команд входят дополнительные команды, перечисленные в табл. 1.3.

Таблица 1.2. Базовый набор RISC-команд

Мнемокод команды	Операнды	Описание команды
`add, sub, mul, div, divu, rem, remu`	`RT, RA, RB`	`RT←RA op RB` Здесь `op` — сложение (*add*), вычитание (*sub*), умножение (*mul*), знаковое деление (*div*), беззнаковое деление (*divu*), остаток от знакового деления (*rem*) или остаток от беззнакового деления(*remu*)
`addi, muli`	`RT, RA, I`	`RT←RA op I` Здесь `op` — сложение (*addi*) или умножение (*muli*), `I` — непосредственно заданное 16-битовое знаковое значение
`addis`	`RT, RA, I`	`RT←RA+(I || 0x0000)`
`and, or, xor`	`RT, RA, RB`	`RT←RA op RB` Здесь `op` — побитовое *и* (*and*), *или* (*or*) или *исключающее или* (*xor*)
`andi, ori, xori`	`RT, RA, Iu`	То же самое, но последний операнд является непосредственно заданным 16-битовым беззнаковым значением
`beq, bne, blt, ble, bgt, bge`	`RT, target`	Переход к метке `target`, если выполнено некоторое условие, т.е. если `RT=0`, `RT≠0`, `RT<0`, `RT≤0`, `RT>0` или `RT≥0` соответственно (`RT` — целое знаковое число)
`bt, bf`	`RT, target`	Переход к метке `target`, если выполнено некоторое условие (`true`/`false`); аналогичны командам `bne`/`beq` соответственно
`cmpeq, cmpne, cmplt, cmple, cmpgt, cmpge, cmpltu, cmpleu, cmpgtu, cmpgeu`	`RT, RA, RB`	`RT` содержит результат сравнения `RA` и `RB`; `RT` равен 0, если условие ложно, и 1, если условие истинно. Мнемокоды означают: сравнить на равенство, неравенство, меньше, не больше и т.д, как и в командах условного перехода. Суффикс “`u`” в названии обозначает сравнение беззнаковых величин
`cmpieq, cmpine, cmpilt, cmpile, cmpigt, cmpige`	`RT, RA, I`	Аналогичны командам серии `cmpeq`, но второй операнд представляет собой непосредственно заданное 16-битовое знаковое значение
`cmpiequ, cmpineu, cmpiltu, cmpileu, cmpigtu, cmpigeu`	`RT, RA, Iu`	Аналогичны командам серии `cmpltu`, но второй операнд представляет собой непосредственно заданное 16-битовое беззнаковое значение
`ldbu, ldh, ldhu, ldw`	`RT, d(RA)`	Загрузка беззнакового байта, знакового полуслова, беззнакового полуслова или слова в `RT` из ячейки памяти по адресу `RA + d`, где `d` — непосредственно заданное знаковое 16-битовое значение

Окончание табл. 1.2

Мнемокод команды	Операнды	Описание команды
mulhs, mulhu	RT,RA,RB	В RT помещаются старшие 32 бита результата умножения RA и RB (знакового и беззнакового)
not	RT,RA	В RT помещается побитовое дополнение RA до единицы
shl, shr, shrs	RT, RA, RB	В RT помещается значение RA, сдвинутое влево или вправо; величина сдвига задается шестью младшими битами второго операнда (RB). При выполнении команды shrs освободившиеся биты заполняются содержимым знакового разряда, в остальных командах освободившиеся биты заполняются нулями. (Значение величины сдвига вычисляется по модулю 64)
shli,shri,shrsi	RT, RA, Iu	В RT помещается значение RA, сдвинутое влево или вправо на величину, задаваемую пятью младшими битами непосредственно заданного значения I
stb,sth,stw	RS, d(RA)	Сохранение байта, полуслова или слова из регистра RS в ячейку памяти по адресу RA + d, где d — непосредственно заданное 16-битовое знаковое значение

Везде в описаниях команд исходные операнды RA и RB представляют собой содержимое регистров.

В реальной машине обязательно должны быть команды ветвления и обращения к подпрограммам, команды перехода по адресу, содержащемуся в регистре (для возврата из подпрограмм и обработки оператора выбора альтернативы типа switch), а также, возможно, ряд команд для работы с регистрами специального назначения. Конечно же, должны быть привилегированные команды и команды вызова служб супервизора. Кроме того, вероятно наличие команд для работы с числами с плавающей точкой.

Краткое описание некоторых дополнительных команд, которые может иметь RISC-компьютер, приведено в табл. 1.3.

Таблица 1.3. Дополнительные команды полного набора RISC-команд

Мнемокод команды	Операнды	Описание команды
abs, nabs	RT, RA	RT получает абсолютное значение (или отрицательное абсолютное значение) RA
andc, eqv, nand, nor, orc	RT, RA, RB	Побитовое *и с дополнением*, *эквивалентность*, *отрицание и*, *отрицание или* и *или с дополнением*
extr	RT, RA, I, L	Извлечение битов от I до I+L-1 из RA и размещение их в младших битах RT; остальные разряды заполняются нулями
extrs	RT, RA, I, L	Аналогична extr, но свободные биты заполняются содержимым бита знака

Окончание табл. 1.3

Мнемокод команды	Операнды	Описание команды
ins	RT, RA, I, L	Вставляет биты регистра RA от 0 до L-1 в биты от I до I+L-1 приемника RT
nlz	RT, RA	RT содержит количество старших нулевых битов RA (от 0 до 32)
pop	RT, RA	RT содержит количество единичных битов RA (от 0 до 32)
ldb	RT, d(RA)	Загрузка знакового байта из ячейки памяти по адресу RA + d в RT, где d — непосредственно задаваемое 16-битовое знаковое значение
moveq, movne, movlt, movle, movgt, movge	RT, RA, RB	В RT помещается значение RB, если RA=0 (RA≠0 и т.д.). Если условие не выполняется, содержимое RT не изменяется
shlr, shrr	RT, RA, RB	В RT помещается значение RA после циклического сдвига влево или вправо; величина сдвига задается пятью младшими битами RB
shlri, shrri	RT,RA,Iu	В RT помещается значение RA после циклического сдвига влево или вправо; величина сдвига задается пятью младшими битами непосредственно заданного значения
trpeq, trpne, trplt, trple, trpgt, trpge, trpltu, trpleu, trpgtu, trpgeu	RA, RB	Прерывание (*trap*), если RA=RB (RA≠RB и т.д.)
trpieq, trpine, trpilt, trpile, trpigt, trpige	RA, I	Аналогичны trpeq и остальным командам серии с тем отличием, что второй операнд представляет собой непосредственно заданное 16-битовое знаковое значение
trpiequ, trpineu, trpiltu, trpileu, trpigtu, trpigeu	RA, Iu	Аналогичны trpltu и другим командам серии с тем отличием, что второй операнд представляет собой непосредственно заданное 16-битовое беззнаковое значение

На практике в языке ассемблера удобно иметь ряд “расширенных мнемоник”, похожих на макросы, которые обычно выполняются как одна команда. Некоторые из них представлены в табл. 1.4.

Таблица 1.4. Расширенные мнемоники

Расширенная мнемоника	Расширение	Описание
`b target`	`beq R0, target`	Безусловный переход
`li RT, I`	См. пояснение в тексте	Загрузка непосредственно заданного числа, $-2^{31} \le I < 2^{32}$
`mov RT, RA`	`ori RT, RA, 0`	Пересылка регистра `RA` в `RT`
`neg RT, RA`	`sub RT, R0, RA`	Отрицание (побитовое дополнение до двух)
`subi RT, RA, I`	`addi RT, RA, -I`	Вычитание непосредственно заданного значения ($I \neq -2^{15}$)

Команда загрузки непосредственно заданного значения расширяется до одной или двух команд, в зависимости от непосредственного значения `I`. Например, если $0 \le I < 2^{16}$, можно применить команду `ori` с использованием регистра R0. Если $-2^{15} \le I < 0$, можно обойтись командой `addi` с регистром R0. Если младшие 16 битов `I` — нулевые, используем команду `addis`. Для остальных значений `I` выполняются две команды: например, после команды `ori` выполняется `addis`. (Иначе говоря, в последнем случае выполняется загрузка из памяти, но при оценке времени и места, которое требуется на выполнение этой макрокоманды, будем считать, что выполняются две элементарные арифметические команды.)

Существуют разные мнения о том, к какому набору RISC-команд следует отнести ту или иную команду — к базовому или к расширенному. Команды *беззнакового деления* и *получения остатка от деления* вполне можно было бы включить в расширенный набор RISC-команд. Команда *знакового сдвига вправо* — еще одна команда, редко используемая в тестах SPEC. Причина в том, что в C очень легко случайно использовать эти команды неправильно, например выполнив беззнаковое деление там, где используются знаковые операнды, или выполнить сдвиг вправо на величину со знаком (`int`), которая на самом деле является беззнаковой. Кстати, *не* следует выполнять деление целого знакового числа на степень 2 при помощи *знакового сдвига вправо*; так, вы должны добавить к результату 1, если делимое отрицательно и был выполнен сдвиг нескольких ненулевых битов.

Различия между основным и расширенным набором RISC-команд приводят ко множеству подобных спорных моментов, но не будем подробно останавливаться на этой теме.

Действие команд ограничивается двумя входными регистрами и одним выходным, что упрощает работу компьютера. Упрощается также работа оптимизирующего компилятора — ему не приходится обрабатывать инструкции с несколькими целевыми регистрами. Однако за такое упрощение приходится платить: если в программе требуется вычислить и частное, и остаток от деления двух чисел, то приходится выполнять две команды (деление и получение остатка). Стандартный алгоритм деления вместе с частным вычисляет и остаток от деления, поэтому на многих машинах и частное и остаток получаются в результате выполнения одной команды деления. Аналогичные замечания применимы и к случаю, когда при умножении двух слов получается двойное слово-произведение.

Создается впечатление, что команда *условной пересылки* (`moveq`) имеет только два входных операнда, хотя на самом деле их три. Поскольку результат выполнения этой команды зависит от содержимого регистров `RT`, `RA` и `RB`, при неупорядоченном выполнении команд `RT` следует интерпретировать и как *используемый*, и как *устанавливаемый* ре-

гистр одновременно. Представьте ситуацию, когда за командой, устанавливающей значение RT, следует команда *условной пересылки* и при этом нельзя потерять результат первой команды. Таким образом, разработчик машины может убрать команду *условной пересылки* из набора допустимых команд, тем самым исключив обработку команд с тремя (логическими) входными операндами. С другой стороны, команда *условной пересылки* снижает количество ветвлений в программе.

В данной книге формат команд не играет большой роли. Полное множество RISC-команд, описанное выше, вместе с командами для работы с числами с плавающей точкой и рядом команд супервизора может быть реализовано при помощи 32-битовых команд на компьютере с 32 регистрами общего назначения (5-битовые поля регистров). Если размер непосредственно заданных полей в командах сравнения, загрузки, сохранения и прерывания снизить до 14 бит, то это же множество команд может быть реализовано на компьютере с 64 регистрами общего назначения (с использованием 6-битовых полей регистров).

Время выполнения

Предполагается, что все команды выполняются за один такт процессора, за исключением команд умножения, деления и получения остатка от деления, для которых невозможно точно определить время выполнения. Команды перехода занимают один такт, независимо от того, был выполнен переход или нет.

Команда *загрузки непосредственного значения* выполняется за один или два такта, в зависимости от того, сколько элементарных арифметических команд требуется для формирования константы в регистре.

Команды *загрузки* и *сохранения* не часто упоминаются в данной книге, но будем считать, что они выполняются за один такт, пренебрегая при этом задержкой при загрузке (промежутком времени между моментом завершения команды в арифметическом модуле и моментом, когда данные становятся доступны следующей команде).

Однако зачастую одних только знаний о количестве тактов, используемых всеми арифметическими и логическими командами, недостаточно для правильной оценки времени выполнения программы. Выполнение программы часто замедляется вследствие задержек, возникающих при загрузке и выборке данных. Задержки такого рода не обсуждаются в книге, хотя и могут существенно влиять на скорость выполнения программ (причем это влияние постоянно возрастает). Другим источником сокращения времени выполнения является возможность распараллеливания вычислений на уровне команд, которая реализована практически на всех современных RISC-компьютерах, особенно на специализированных быстродействующих машинах.

В таких машинах имеется несколько исполнительных модулей и возможность диспетчеризации команд, что позволяет параллельное выполнение независимых команд (независимыми команды считаются тогда, когда ни одна из них не использует результаты других команд, команды не заносят значения в один и тот же регистр или бит состояния). Поскольку такие возможности компьютеров теперь уже не редкость, в книге много внимания уделяется независимым командам. Так, можно сказать, что некая формула может быть закодирована таким образом, что для вычисления потребуется выполнение восьми инструкций за пять тактов на машине с неограниченными возможностями распараллеливания вычислений на уровне команд. Это означает: если команды расположены в правильном порядке и машина имеет достаточное количество регистров и арифметических модулей, то она в принципе способна выполнить такую программу за пять тактов.

Более подробно обсуждать этот вопрос не имеет смысла, так как возможности машин в плане распараллеливания вычислений на уровне команд существенно различаются. Например, процессор IBM RS/6000, созданный в 1992 году, имеет трехвходовой сумматор и может параллельно выполнять две следующие друг за другом команды сложения, даже если одна из них использует результаты другой команды (например, когда команда сложения использует результат команды сравнения или основной регистр команды загрузки). В качестве обратного примера можно рассмотреть простейший дешевый встраиваемый в различные системы компьютер, у которого регистр ввода-вывода имеет только один порт для чтения. Естественно, что такой машине для выполнения команды с двумя операндами-источниками потребуется дополнительный такт для повторного чтения регистра ввода-вывода. Однако если команда выдает операнд для команды, непосредственно следующей за ней, то этот операнд оказывается доступным и без чтения регистра ввода-вывода. На машинах такого типа возможно ускорение работы, если каждая команда предоставляет данные для следующей команды, т.е. если параллелизмы в коде отсутствуют.