В.Э. Карпов

ТЕОРИЯ КОМПИЛЯТОРОВ

ЧАСТЬ 2.

ДВОИЧНАЯ ТРАНСЛЯЦИЯ

Москва 2014

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ

Задачей курса является создание учебного программного комплекса, демонстрирующего механизмы двоичной трансляции. Комплекс должен включать в себя следующие составляющие:

1) компилятор языка высокого уровня;
2) двоичный транслятор;
3) загрузчик;
4) виртуальную машину.

На вход комплекса поступает тест программы на некотором языке высокого уровня, затем компилятор превращает ее в некий модельный объектный код. Далее двоичный транслятор распараллеливает последовательный код и формирует т.н. широкие командные слова VLIW (Very Long Instruction Word). Загрузчик загружает их в память и затем виртуальная машина выполняет полученную объектную программу.

Виртуальная машина представляет собой модель некоторого многопроцессорного вычислительного устройства.

В упрощенной форме создаваемый программный комплекс выглядит так:

Здесь

ПЯВУ – программа на языке высокого уровня;
ЭОС – эмулятор операционной системы;
ВМ – виртуальная машина.

В настоящем пособии приведены некоторые теоретические сведения из области двоичной трансляции, а также алгоритмы и методы реализации ее основных элементов.

2. ОБЩАЯ СХЕМА РАСПАРАЛЛЕЛИВАНИЯ ПРОГРАММЫ

Пусть объектный код представляет собой набор тетрад. При этом будем полагать, что все этапы трансляции и компоновки были пройдены, в т.ч. была проведена оптимизация (как машинно-независимая, так и машинно-зависимая) и были настроены все адреса.

Последовательность тетрад образует *линейный список инструкций* $[w_1,w_2,\ldots,w_n]$. Из этого списка последовательно выбираются инструкции, которые и отправляются на выполнение. Это – стандартная процедура для фон-неймановской архитектуры.

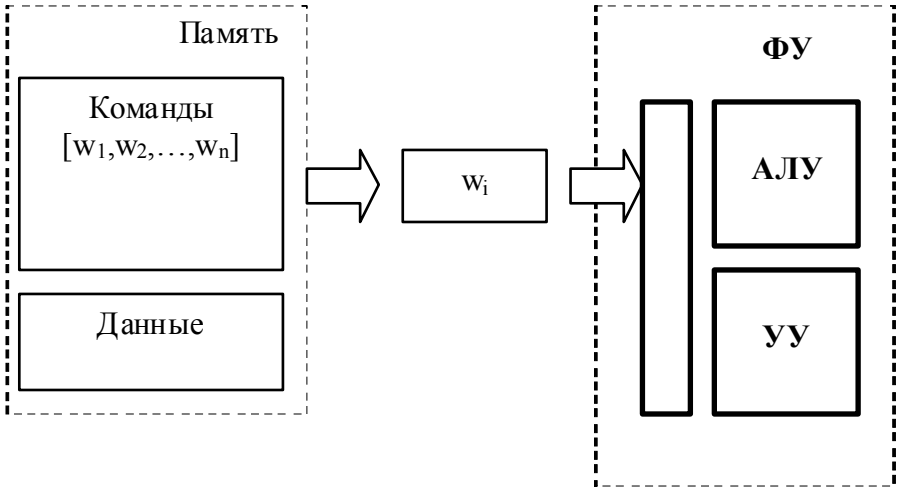


Рис. 1. Последовательное выполнение списка инструкций

Устройство управления (УУ), арифметико-логическое устройство (АЛУ), прочие узлы, необходимые для выполнения вычислений, объединим в некоторый узел, который будем называть *функциональным устройством* (ФУ).

Существует много путей решения задачи увеличения скорости исполнения программы. Например, можно сосредоточиться на создании эффективных оптимизирующих компиляторов, которые пытаются исключить избыточное исполнение инструкций или сохраняют исходные данные и промежуточные результаты в быстродействующей регистровой памяти, тем самым уменьшая затраты на вычисление адреса и доступ к памяти. Кроме того, можно повышать тактовую частоту процессора, увеличивать емкость сверхоперативной памяти и т.п. Но мы сейчас рассмотрим ситуацию, при которой повышение эффективности исполнения программы основано на наличии множества параллельно работающих функциональных узлов.

Представим теперь, что этих ФУ много. Очевидно, что чем больше ФУ мы сможем единовременно загрузить работой, тем большей будет производительность всего вычислительного комплекса. Для этого нам необходимо выбирать не по одной инструкции w_i из списка команд, а извлекать по несколько команд. При этом, разумеется, извлекать мы можем те команды, которые могут выполняться одновременно, не конфликтуя друг с другом. Такая группа одновременно выполняемых команд образует т.н. широкое командное слово VLIW. Чем больше команд будет содержаться в VLIW, тем больше ФУ мы сможем загрузить работой.

Таким образом, необходимо преобразовать линейный список инструкций [w_1,w_2,…,w_n] в список широких командных слов [$VLIW_1$,$VLIW_2$,…,$VLIW_m$].

$$[w_1, w_2, ..., w_n] \Rightarrow \begin{bmatrix} VLIV_1 \\ ... \\ VLIV_m \end{bmatrix} = \begin{bmatrix} [w_1^1, w_2^1, ..., w_k^1] \\ ... \\ [w_1^m, w_2^m, ..., w_l^m] \end{bmatrix}$$

Формировать из последовательного потока формируется множество VLIW можно на аппаратном уровне (на этапе выполнения). И тогда мы имеем дело с суперскаляром. Если же эта задача решается программно, то такой транслятор называется двоичным транслятором (иногда его называют двоичным компилятором).

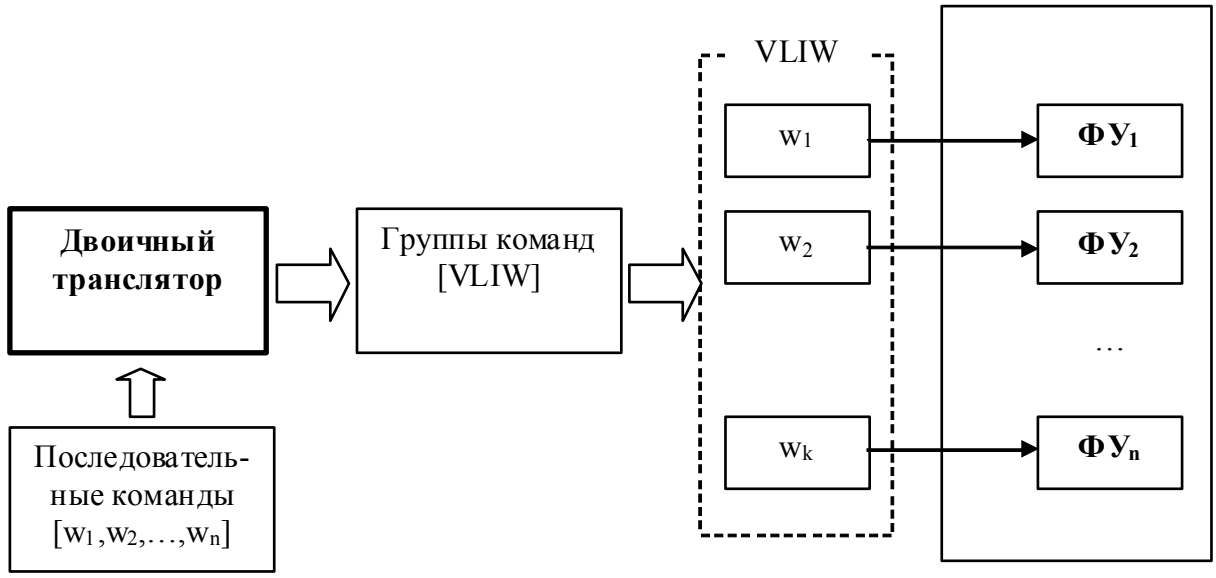


Рис. 2. Выполнение списка множеств команд – широких командных слов

Представим поток команд в виде графа, описывающего порядок выполнения инструкций. Это – т.н. *граф программы*.

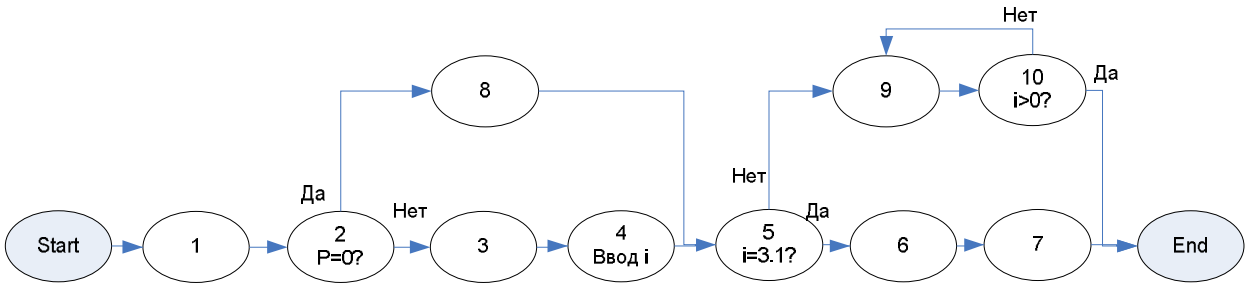


Рис. 3. Граф программы

В этом графе имеются вершины–источники (они определяют начало ветвления) и вершины (инструкции), на которые осуществляется переход. В нашем примере источниками являются вершины 2, 5 и 10. А управление передается на вершины 5 и 9.

Все эти вершины важны для анализа структуры программы, т.к. именно они определяют пути выполнения. Поэтому первым этапом является расстановка специальных меток для исходных инструкций. Эти метки определяют тип вершины (источник или приемник). Метки также могут определять и иные характеристики вершины, которые могут потребоваться дальше.

Следующим этапом является выделение трасс – фрагментов программы (графа), которые будут выполняться с наибольшей вероятностью. Планирование трасс – одна из наиболее проблемных задач. Здесь используются зачастую слабоформализуемые приемы и методы (эвристики), однако от того, насколько правильно спланированы трассы, в большой степени зависит эффективность распараллеливания.

Например, граф программы может быть представлен в следующем виде:

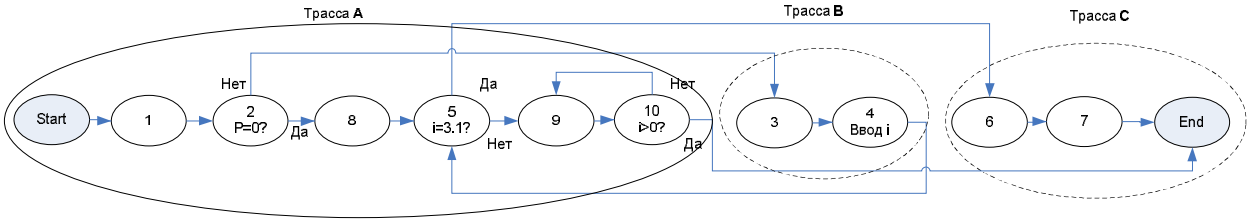


Рис. 4. Граф программы с выделенными участками

Здесь, исходя из некоторых соображений, о которых мы будем говорить ниже, было высказано предположение, что исходная программа будет разбита на 3 трассы – A, B и C.

После того, как были спланированы трассы, эти последовательности инструкций станут основным объектом оптимизации. После приведения трасс к приемлемому для обработки виду, будут образованы т.н. линейные участки.

Линейный участок – это некая последовательность инструкций, блок, имеющий один вход и не более чем два выхода.

Далее происходит обработка основного объекта для распараллеливания кода – линейных участков. Для них строятся свои модели, описывающих, например, то, как одни данные зависят от вычисления других.

После определения взаимного влияния формируются множества инструкций, которые способны исполняться параллельно, не мешая друг другу. Это – т.н. ярусно-параллельная форма.

Наконец, между командами распределяются такие ресурсы, как регистры, окончательно формируются широкие командные слова, настраиваются, если надо, адреса.

Таким образом, общая последовательность формирования множества параллельно исполняемых команд выглядит следующим образом:

1. Формирование модели макроуровня. Объект – исходный поток инструкций.
    1.1. Расстановка меток.
    1.2. Построение графа программы.
    1.3. Планирование трасс.
    1.4. Преобразование трасс.
    1.5. Формирование линейных участков.

2. Формирование модели микроуровня. Объект – линейные участки.
    2.1. Построение графа зависимости по данным (ГЗД).
    2.2. Преобразование ГЗД к ярусно-параллельной форме.
    2.3. Распределение регистров.

Эти этапы образуют структуру двоичного транслятора:

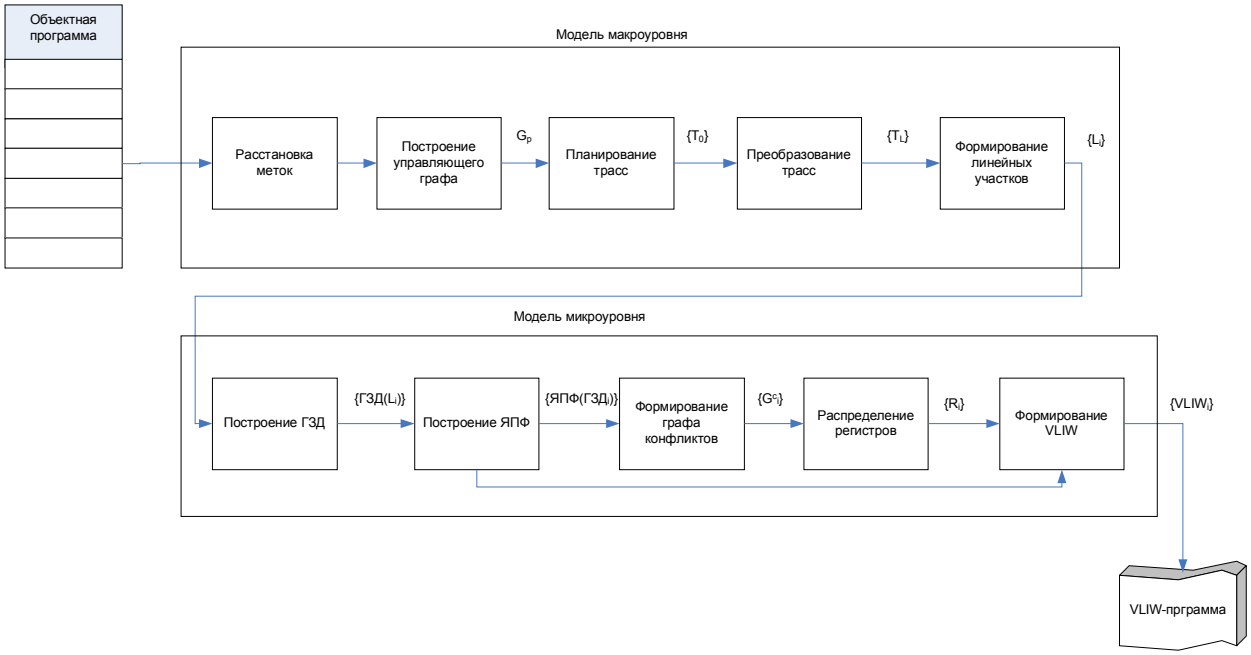


Рис. 5. Общая структура двоичного транслятора

Рассмотрим подсистемы двоичного транслятора более подробно.

3. ЭЛЕМЕНТЫ ДВОИЧНОГО ТРАНСЛЯТОРА

Расстановка меток

Для каждой тетрады определяются ее атрибуты, относящие тетраду к типу «развилка», «сток», «плохая инструкция».

Под «плохими инструкциями» мы будем понимать те, которые «мешают» распараллеливанию. Это, например, инструкции, имеющие выход на внешние (периферейные) устройства, прежде всего - операции ввода-вывода; вызовы подпрограмм; операции синхронизации и т.п.

Определение типа управления достаточно тривиально. Если мы имеем дело с операцией условного перехода, то это – «развилка». Если же адрес (номер) тетрады используется где-либо в качестве адреса перехода, то это – «сток». При этом, очевидно, что тетрада может иметь несколько подобных атрибутов (быть и «развилкой», и «стоком»).

Построение управляющего графа

Управляющий граф содержит описание линейных блоков программы и представляет собой орграф, вершинами которого являются линейные участки программы, а дуги указывают пути передачи управления. Управляющий граф имеет единственную входную и единственную выходную вершину. Каждая вершина имеет не более двух потомков.

Структура линейного блока:
Номер блока
Метка блока
Номер начальной инструкции
Номер конечной инструкции
Переход 1 (номер инструкции)
Переход 2 (номер инструкции)

Алгоритм формирования управляющего графа

Вход: размеченный поток тетрад.
Выход: управляющий граф в виде описания множества линейных блоков

```
Инициализация;
блок_готов := False;
Цикл Пока (поток тетрад не пуст)
    Считать тетраду;
    Если (тетрада с меткой) То
        Фиксация окончания очередного блока;
        блок_готов := True;
    Кесли
    Если (код операции = переход) То
        Фиксация окончания очередного блока;
        Фиксация метки перехода;
        блок_готов := True;
        окончание_блока_по_переходу = True;
    Кесли
    Если (блок_готов = True) То
```

```
        Добавить к графу вершину;
    Кесли
    Если (окончание_блока_по_переходу = True) То
        Добавить к графу дугу соответствующего перехода;
    Кесли
КонецЦикла
Конец алгоритма
```

Планирование трасс

Оптимизируемый участок разбивается на набор трасс, представляющих собой пути с высокой вероятностью исполнения. Трасса может содержать условные ветви, выходы из середины (side-exit) и входы из других трасс (side-entrance).

В процессе планирования и оптимизации наличие этих переходов не учитывается. Трасса оптимизируется как простой блок. После планирования необходимо выполнение специальной процедуры (bookkeeping) для сохранения корректности кода, не принадлежащего трассе.

Методы предсказания ветвлений

Как уже говорилось выше, основной единицей оптимизации по времени является простой блок. Чем больше (длиннее) блок, тем лучше. Таким образом, основной задачей является преобразование УГ для получения максимально длинных линейных блоков.

Линейные участки могут быть «хорошими» и «плохими». К «плохим» будем относить, очевидно, короткие линейные участки, а также участки, содержащие внутри себя «плохие» операции.

"Плохие" инструкции. Существуют группы инструкций, которые ухудшают возможности использования потенциального параллелизма:

- вызовы внешних подпрограмм;
- возвраты из подпрограмм;
- операции ввода-вывода;
- операции синхронизации по времени;
- переходы по вычисляемым адресам (т.к. адрес перехода неизвестен заранее, то оптимизировать нельзя);
- операции с данными, находящимися по вычисляемым адресам (невозможно проследить зависимости по данным на этапе трансляции).

Итак, важно сформировать блоки так, чтобы они были длинными и «хорошими». При наличии ветвлений очень важным становится определение наиболее пригодных для оптимизации участков, т.е. выбор *доминирующей ветви*. Таким образом, при анализе графа возникает задача выбора оптимизации – задача предсказания ветвлений.

Существует ряд правил, которые могут помочь в предсказании направления выполнения программы. Некоторые из этих правил носят неформальный характер, поэтому их вполне можно назвать эвристиками.

Предсказание на основе истории ветвлений. Сбор статистики об исходах операций ветвления и построение на ее основе строится предположение о результате выполнения текущей операции.

Предсказание на основе пробных прогонов программы. Для этого производится имитация выполнения программы на одном или нескольких наборах данных. Собирается

статистика. Недостаток очевиден: метод работает лишь при определенных обстоятельствах и исходных данных.

Использование эвристик. Метод работает быстро, но не всегда надежно.

Вот как могут выглядеть эвристики выбора доминирующей ветви.

1. Избегание «плохих» инструкций.
2. Условия с указателями. Обычно справедливы условия
   Ptr≠NULL
   Ptr1≠Ptr2
   if((ptr=malloc(…))!=NULL) …
   for(p=p0;p!=NULL;p=p–>next) …
3. Эвристика исполнения циклов. Если при ветвлении одна из ветвей содержит цикл, то обычно именно она и будет доминировать. По статистике исполнение циклов занимает до 90% времени выполнения программы в целом.
4. Эвристика направления ветвления. Возврат назад более вероятен (высока вероятность неявного цикла с постусловием).
5. Предсказание по коду операции:
   – при сравнении чисел с плавающей точкой более вероятно неравенство;
   – отрицательные числа менее вероятны.

После выбора доминирующих ветвей могут быть определены границы оптимизации.

Пример. Пусть дан следующий УГ:

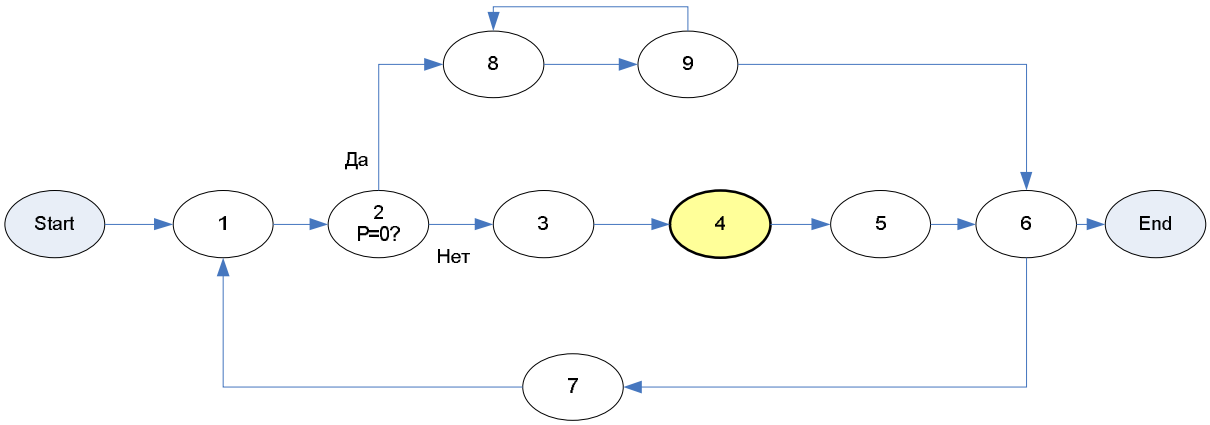


Рис. 6. Исходный управляющий граф

Спланируем трассы – вероятный путь выполнения программы. Наши правила расположены по убыванию приоритета.

В первую очередь, избегаем плохих инструкций У нас это (4). Если бы (4) была хорошей командой, то мы от (2) перешли на (3), т.к. чаще всего при выполнении сравнения числа не равны друг другу. После (6) идем на (7), т.к. вероятнее движение по циклу.

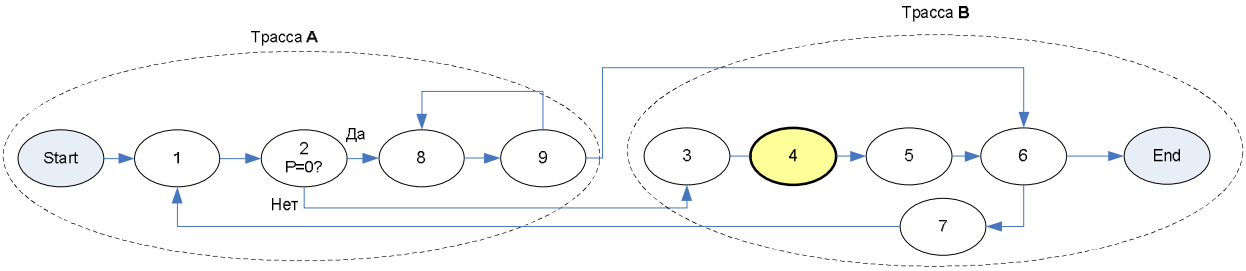


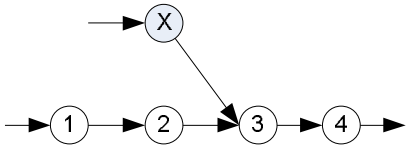
Рис. 7. Наиболее вероятные трассы

Теперь у нас есть трассы наиболее вероятного выполнения программы. Далее будем оптимизировать полученные участки A и B.

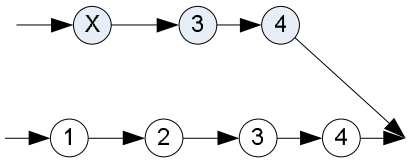
Преобразование трасс

Как видно, трасса включает в себя помимо последовательности линейных блоков различного рода внутренние переходы, входы и выходы из середины трассы. Однако нам крайне желательно свести ее к *линейному* виду. Классический подход – **метод дублирования остатка**. При этом производится эквивалентные преобразования трассы, что зачастую ведет к увеличению объема объектного кода. Несмотря на это, общий выигрыш во времени исполнения может оказаться все равно выше, т.к. мы будем иметь дело с более длинными участками.

Вход в трассу. Пусть имеется следующий фрагмент трассы:



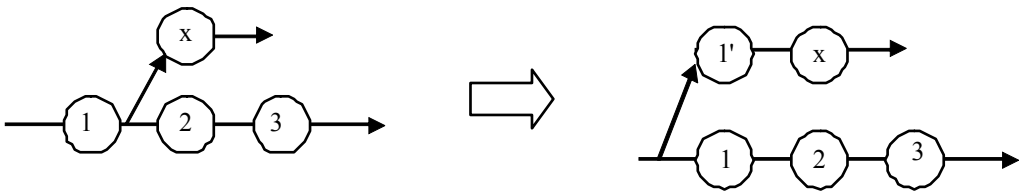
Нам нужно перенести точку входа из середины трассы в ее конец. Для этого продублируем инструкции (3) и (4).



Таким образом мы получаем два достаточно длинных линейных участка. Это – достаточно простое и естественное преобразование.

Выход из трассы. Здесь происходит перемещение инструкций позже выхода из середины трассы. Копии инструкций, значение которых используется после выхода, должны быть помещены между выходом и первым использованием их значений.

Это – менее очевидное преобразование, т.к. в данной ситуации мы имеем дело с развилкой, определяющей ход выполнения программы.



Очевидно, что и здесь объем инструкций при этом возрастает, однако параллелизм увеличивается.

На практике выход из трассы реализуем крайне нетривиально, т.к. с этим преобразованием связано понятие спекулятивного исполнения программы. Как уже говорилось выше, спекулятивное исполнение подразумевает наличие весьма специфических механизмов, например, использования предикатных файлов.

Распараллеливание и оптимизация циклов

Львиная доля вычислительных ресурсов обычно приходится на циклы. Поэтому очевидно, что оптимизация циклов дает весьма ощутимый результат. Однако провести качественную их оптимизацию удается далеко не всегда. Речь идет, разумеется, о такой оптимизации циклов, как их распараллеливание, т.к. мы полагаем, что с классические способы, подобные вынесению инвариантов за пределы цикла, уже были применены заранее. Существуют три основных методов оптимизации/распараллеливания циклов:

1. Грубое распараллеливание. Распараллеливаем, не взирая на данные.
2. Предикатное исполнение. Задерживаем выполнение инструкции до времени выполнения итерации.
3. Точное распараллеливание – распараллеливание линейного участка внутри итерации (тело цикла).

Грубое распараллеливание (coarce-grain parallelism). Суть заключается в комбинировании отдельных итераций циклов. При этом выделяют два основных метода: *DoAll* и *DoAcross*. *DoAll* не учитывает связи по данным между итерациями и пытается начать следующую, как только появляются свободные вычислительные ресурсы. *DoAcross* изменяет порядок выполнения итераций для соблюдения зависимости по данным (например, задерживает начало выполнения некоторых итераций до завершения определенных инструкций в предыдущих).

Методы «точного» распараллеливания (fine-grain parallelism) используют параллелизм внутри цикла (на уровне инструкций).

Линейные участки

Линейным участком называют последовательность инструкций, у которой имеется вход и два выхода. Линейный участок заканчивается тогда, когда осуществляется переход или ставится метка.

Линейный участок – это основной объект оптимизации. Чем длиннее участок, тем больше возможностей для параллельных вычислений. По линейному участку строится граф зависимости по данным (ГЗД), который далее приводится к ярусно-параллельной форме (ЯПФ). Далее, после процедуры распределения регистров, формируются длинные командные слова (VLIW).

Граф зависимости по данным

Для описания линейных участков строится т.н. *граф зависимости по данным* (ГЗД).

Пусть имеется участок программы – список инструкций

$$A=(a_1,a_2,\ldots,a_n)$$

Каждая инструкция a_i представлена в машинно-независимой тетрадной форме

$$a_i=(OP_i, I_i^{(1)}, I_i^{(2)}, R_i),$$

где OP_i,– операция, $I_i^{(1)}$, $I_i^{(2)}$ – операнды, R_i – результат.

ГЗД участка A представляет собой граф (A,V) с вершинами $a_i \in A$ и дугами $(a_i, a_j) \in V$

Множество V определяется как

$$V=\{(a_i, a_j) : i<j, (R_i \cap I_j) \cup (R_j \cap I_i) \cup (R_i \cap R_j) \neq \varnothing\}$$

или:

$$V=\{(a_i, a_j): i<j, (R_i=I_j)\vee(R_j=I_i)\vee(R_i=R_j)=true\}$$

Пример. Пусть необходимо вычислить выражение

$$S = p*(p–a)*((p–b)*(p–c))$$

Обратите внимание на скобки. Они определяют порядок формирования и выполнения операций. Для этого выражения мы имеем следующее множество тетрад:

1. (-, p, a, T1)
2. (-, p, b, T2)
3. (-, p, c, T3)
4. (*, T2, T3, T4)
5. (*, p, T1, T5)
6. (*, T5, T4, T6)

По этому множеству строим граф:

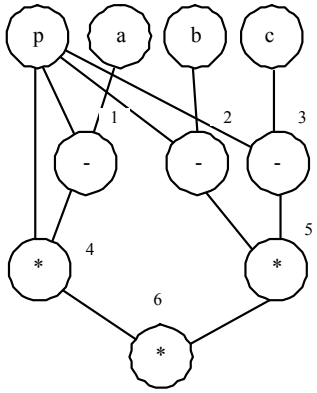


Рис. 8. Граф зависимости по данным

Вершины-операторы промаркированы номерами, соответствующими номерам тетрад.

Очевидно, что полученный граф – не самый удобный объект для распараллеливания. Дело в том, что неясно, какие операции могут выполняться одновременно (не мешая друг другу). К тому же в графе имеются вершины, которые одновременно участвуют в нескольких операциях (вершины ***p, a, b*** и ***c***).

В этом смысле удобнее работать с графом в виде дерева. В дереве у каждой вершины имеется лишь один родитель, что в нашей терминологии означает, что вершина участвует лишь в одной операции.

Ярусно-параллельная форма

Начнем с построения дерева по имеющемуся ГЗД. Для превращения графа в дерево используется *принцип однократного присваивания*. При таком подходе любая переменная может присутствовать в левой части оператора присваивания только один раз.

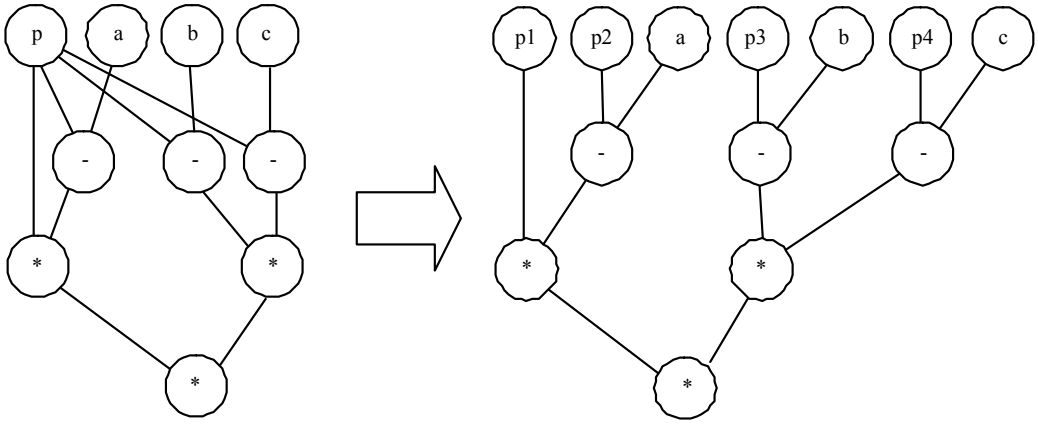
Построение дерева

Для преобразования ГЗД к дереву (лесу бинарных деревьев) используется *дублирование переменных*. Дублирование производится для вершин

$$\deg^-(a_i\in A)>1,$$

т.е. тех, у которых значение, соответствующее вершине используется более одного раза. В результате вершина дублируется $\deg^-(a_i) - 1$ раз и каждой из вершин сопоставляется одна исходящая дуга.

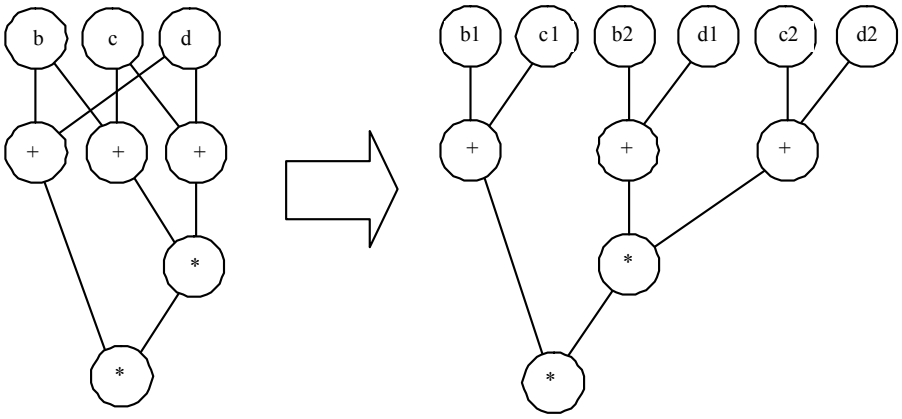
Пример 1. Для рассмотренного выше выражения *S = p*(p–a)*((p–b)*(p–c))* преобразованный граф будет выглядеть так:



Исходный ГЗД Преобразованный ГЗД

Пример 2. Дано выражение

a=(b+c)*(c+d)*(b+d)



Исходный ГЗД Преобразованный ГЗД

Построение ЯПФ

Ярусно-параллельная форма (ЯПФ) – эта некая форма разметки дерева. Эта разметка должна определить уровни (порядок) выполнения операций. В ЯПФ каждой вершине графа приписывается некое число – ранг (номер яруса). Вершины, имеющие один ранг (находящиеся на одном ярусе) могут исполняться одновременно.

Будем считать, что ГЗД оперирует двумя видами тетрад (вершин) – полными тетрадами вида

T4=(OP, A1, A2, R)

и неполными (вырожденными) тетрадами вида

T3=(OP, A, ,R)

Полные тетрады отвечают за выполнение операций с двумя операндами и аргументом-приемником результата. Таковыми являются, например, тетрады арифметических операций:

(+,A,B,C) -- *C:=A+B*

(*,A,B,C) -- *C:=A*B*

и т.д.

Вырожденные тетрады имеют один операнд и аргумент-приемник. Например, такой тетрадой может быть представлен оператор присваивания (если, конечно, этот оператор не возвращает значение, как, скажем, в языке C или C++):

(:=,A, , C) -- *C:=A*

Представление графа ЯПФ

Будем представлять граф ЯПФ в виде множества вершин. Каждая вершина описывает аргумент тетрады. Если с этим аргументом связана операция, то описание содержит адреса аргументов. Описание вершины представляет собой структуру вида

Name	OP
left	right
rank	id

Здесь Name – имя вершины, OP – операция, связанная с данной вершиной, left, right – аргументы (идентификаторы вершин) тетрады, rank – ранг (ярус) вершины, id – идентификатор.

Имя вершины – это имя аргумента тетрады. Идентификатор – внутреннее имя вершины (или ее адрес). Если вершин с одинаковыми именами может быть несколько, то идентификатор – уникальный.

Связи между вершинами определяются полями left и right. Если мы имеем дело с полной тетрадой, то поля left и right содержат адреса (идентификаторы) вершин для операндов A1 и A2 соответственно. Если тетрада вырожденная, то поле right (операнд A2) содержит нулевой указатель (NULL).

Алгоритм построения ЯПФ считывает тетрады из входного потока и, в зависимости от результата анализа аргументов, добавляет очередную вершину в множество вершин графа.

Алгоритм построения ЯПФ

Вход: поток тетрад {T}
Выход: граф G в ярусно-параллельной форме

Очистить список вершин графа G
Цикл по всем тетрадам T
    Выбрать очередную тетраду T.
    Если тип тетрады T соответствует T4 (T=(OP, A1, A2, R)), то
    1) -- *Анализируем аргумент A1.*
        Если ГЗД нет элемента с именем A1, то -- *добавляем новый элемент g1 в G*
            g1.name := A1;
            g1.rank:= 0;
            g1.left := NULL;
            g1.right := NULL.
            добавить элемент g1 в G
        иначе запомнить элемент g1 (g1.name=A1)

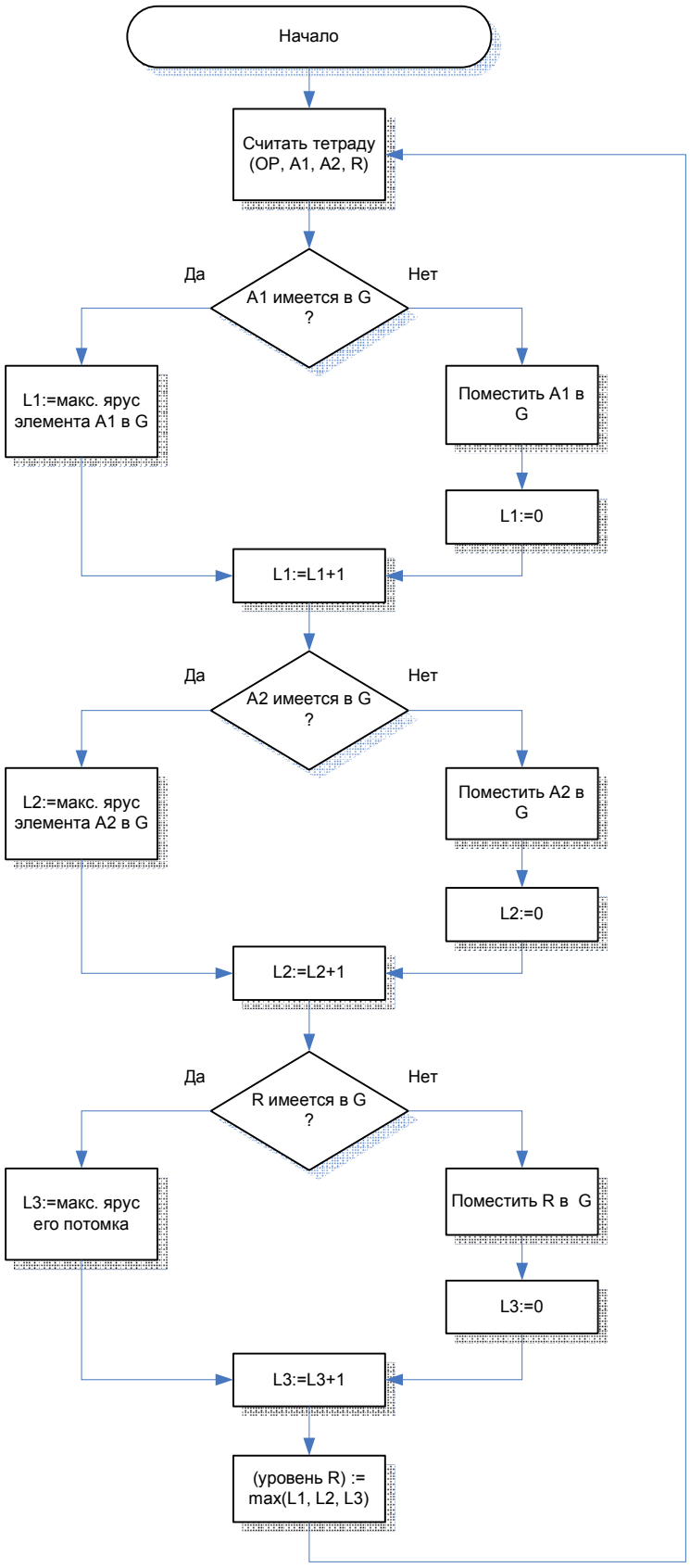
2) -- *Анализируем аргумент A2.*
   Если ГЗД нет элемента с именем A2, то -- *добавляем новый элемент g2 в G*
      g2.name := A2;
      g2.rank:= 0;
      g2.left := NULL;
      g2.right := NULL.
      добавить элемент g2 в G
   иначе запомнить элемент g2 (g2.name=A2)
-- *Анализируем аргумент R.*
3) Найти элемент g3 с максимальным рангом, использующий R в качестве аргумента A1 или A2.
4) Найти элемент g4 максимального ранга с именем R.
5) Выбираем максимальный из рангов среди найденных элементов g_i:
      r_{max} = max(g1.rank, g2.rank, g3.rank, g4.rank)
   (при этом если какой-либо из элементов g_i не был найден в G, то считаем его ранг равным нулю)
6) Помещаем элемент R на ярус со значением r_{max}+1.

КонецЕсли

Если тип тетрады T соответствует T3=(OP, A,, R), то
1) -- *Анализируем аргумент A.*
   Если ГЗД нет элемента с именем A, то -- *добавляем новый элемент g1 в G*
      g1.name := A;
      g1.rank:= 0;
      g1.left := NULL;
      g1.right := NULL.
      добавить элемент g1 в G
   иначе запомнить элемент g1 (g1.name=A)
-- *Анализируем аргумент R.*
2) Найти элемент g2 с максимальным рангом, использующий R в качестве аргумента A1 или A2.
   Найти элемент g3 максимального ранга с именем R.
3) Выбираем максимальный из рангов среди найденных элементов g_i:
      r_{max} = max(g1.rank, g2.rank, g3.rank)
   (при этом если какой-либо из элементов g_i не был найден в G, то считаем его ранг равным нулю)
4) Помещаем элемент R на ярус со значением r_{max}+1.

КонецЕсли

КонецЦикла



Пример. Рассмотрим следующее множество тетрад:

(+,a,b,c) -- *c:=a+b*

(+,a,b,c) -- *c:=a+b*

(+,a,b,b) -- *b:=a+b*

(+,d,e,f) -- *f:=d+e*
(:=,b,,a) -- *a:=b*
(:=,b,,f) -- *f:=b*
(:=,h,,g) -- *g:=h*

В результате последовательного добавления вершин получим следующий вид ГЗД в ярусно-параллельной форме:

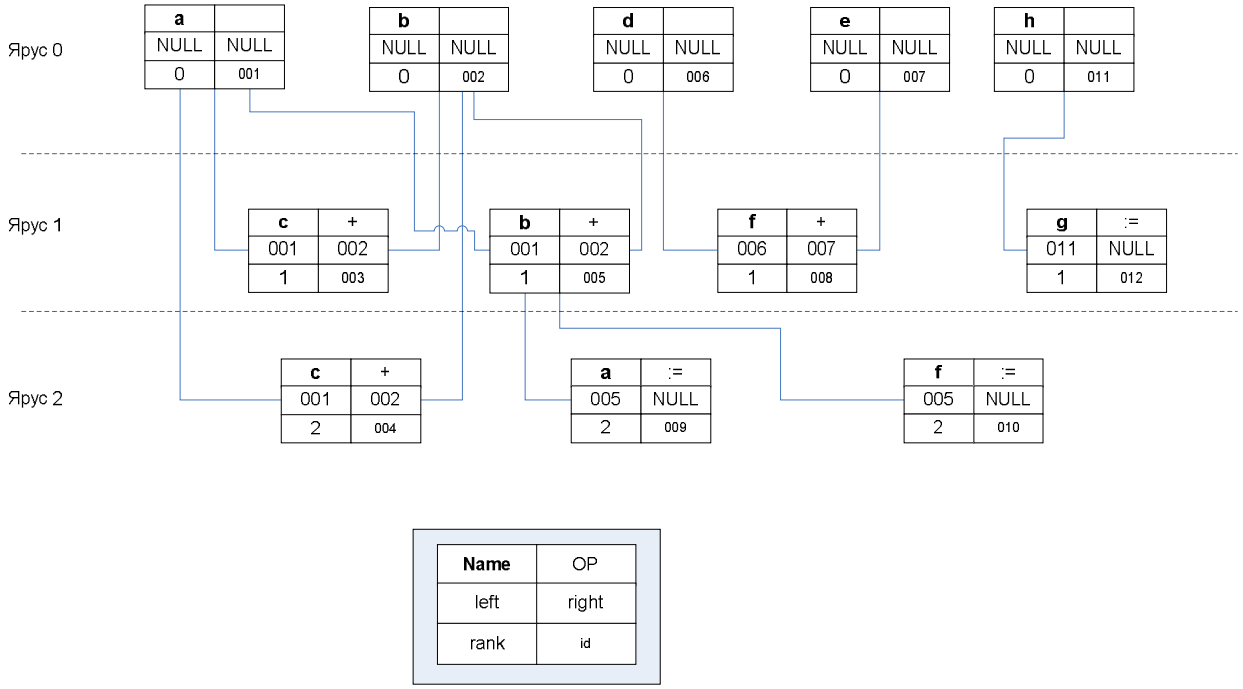


Рис. 9. Итоговая ярусно-параллельная форма

Разумеется, дерево, представляющее ЯПФ должно иметь минимальную высоту (при максимальной ширине). В этом случае мы будем иметь небольшое количество ярусов (командных слов), содержащих много слогов.

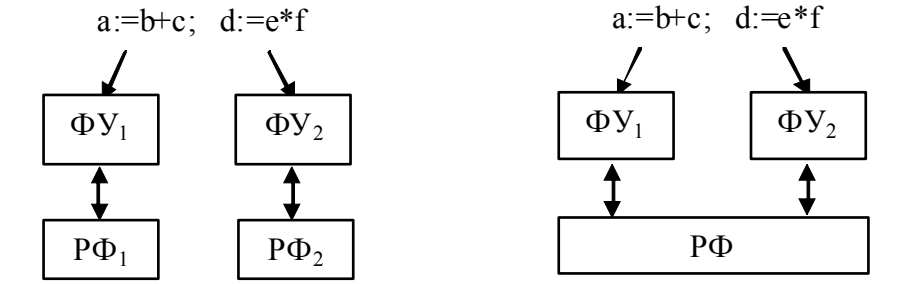
Построение максимальной ярусно-параллельной формы ГЗД представляет собой задачу *оптимальной укладки дерева.*

Распределение регистров

Пусть в нашей предварительной команде Cw^* имеются операции, которые вполне могут выполняться параллельно: [a:=b+c; d:=e*f]. Предполагаем, что a, b, c, d, e, f – адреса ячеек памяти (переменные). Пусть имеется два ФУ – $ФУ_1$ и $ФУ_2$. Будем считать, что мы придерживаемся следующего соглашения: все вычисления (операции) осуществляются исключительно и только с регистрами (своего рода схемотехническое ограничение). Вопрос заключается в том, как и какие регистры мы будем использовать для вычислений.

Начнем с того, что возможна схема, при которой каждое ФУ снабжено своим набором регистров общего назначения, т.е. своим регистровым файлом, Рис. 10а. В такой ситуации задача распределения регистров, вообще говоря, неактуальна. Правда, далее неизбежно возникают проблемы обмена данными между отдельными регистровыми файлами. Да и с точки зрения схемотехники такая схема может считаться излишне

расточительной: регистры – это "дорогие" устройства и не исключено, что их избыток приведет к их малой загрузке.



ФУ1	ФУ2
load R^1_1,b	load R^2_1,e
load R^1_2,c	load R^2_2,f
R^1_3:=R^1_1+R^1_2	R^2_3:=R^2_1*R^2_2
store R^1_3, a	store R^2_3, d

а)

ФУ1	ФУ2
load R_1,b	load R_4,e
load R_2,c	load R_5,f
R_3:=R_1+R_2	R_6:=R_4*R_5
store R_3, a	store R_6, d

б)

Рис. 10. Организация регистровых файлов. а) свой регистровый файл у каждого ФУ, б) общий регистровый файл

Более приемлемым и распространенным вариантом является организация общего регистрового файла, Рис. 10б. Но тогда неизбежно встает задача распределения регистров между ФУ. Необходимо, чтобы функциональные устройства не использовали занятые регистры. Кроме того, желательно также поменьше обращаться к памяти, а брать промежуточные результаты вычислений из регистров, а не хранить их в медленной оперативной памяти.

Рассмотрим следующий граф, полагая, что он описывает некую ЯПФ:

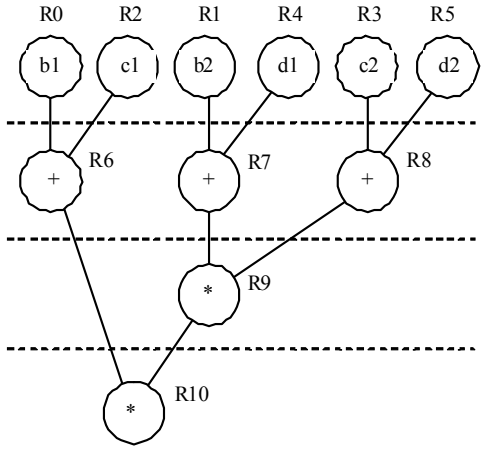


Рис. 11. Исходная ярусно-параллельная форма

Поскольку все операции производятся с использованием регистров, то припишем вершинам имена используемых регистров. Для нашей ЯПФ потребовалось 11 регистров (R0–R11).

Пусть имеется двухпортовый регистровый файл. Речь идет о том, что одновременно мы можем иметь доступ не более, чем к двум регистрам (считывать в

регистры из памяти или записывать значение регистров в память). Тогда из имеющейся ЯПФ мы сформируем следующие командные слова:

Cw_1: (load b1, R0), (load b2, R1)
Cw_2: (load c1, R2), (load c2, R3)
Cw_3: (load d1, R4), (load d2, R5)
Cw_4: (R6=R0+R2), (R7=R1+R4), (R8=R3+R5);
Cw_5: (R9=R7*R8)
Cw_6: (R10=R6*R9)
Cw_7: (store R10, a)

Оптимальная загрузка регистров. Получив ярусно-параллельную форму и сформировав команды, мы можем прийти к ситуации, когда для выполнения всех слогов не хватает регистров. Точнее, необходимо распределить регистры между командами так, чтобы хватило всем и кроме того так, чтобы регистров использовалось не слишком много (регистров обычно не хватает, т.к. регистр – это весьма дорогое аппаратное удовольствие). Разумеется, при этом не должно быть конфликтов из-за регистров.

Сведем эту задачу распределения регистров к хорошо известной задаче дискретной математики - раскраске графа.

Наша задача сводится к созданию графа, вершинами которого являются данные, а дуги определяют пересечение времен жизни (одновременность использования данных). Нам необходимо раскрасить граф – приписать каждой вершине графа свой цвет – используемый регистр. Количество цветов (красок) – это и есть количество регистров. Очевидно, что для начала надо найти минимальное число красок (регистров).

Но прежде следует определиться с тем, какие команды (вершины) вообще могут конфликтовать друг с другом из-за регистров.

Граф конфликтов

Граф конфликтов – это неориентированный граф, вершинами которого являются используемые переменные (данные), а ребра соединяют вершины с пересекающимися временами жизни. Строить граф конфликтов мы будем, опираясь на ЯПФ.

Одновременно "живут" (сосуществуют) те вершины, которые находятся на одном ярусе. Кроме того, сохраняются связи, полученные в ярусно параллельной форме, а также учитываются связи с вершинами предыдущего ярусами.

Иными словами, определяется связь со всеми вершинами, которые находятся выше и имеют потомков ниже или на том же уровне, что и текущая:
Для вершины a_k строится связь (a_k, a_i) с вершиной a_i такой, что: $L(a_i)<L(a_k)$ и $\exists\, a_j$: $L(a_j)\geq L(a_k)$
Здесь L – уровень вершины

Например, пусть дана следующая последовательность инструкций:

a:=b+c;
k:=a*d
e:=b+f
m:=b*g

Соответствующая ЯПФ может выглядеть следующим образом (без дублирования вершин):

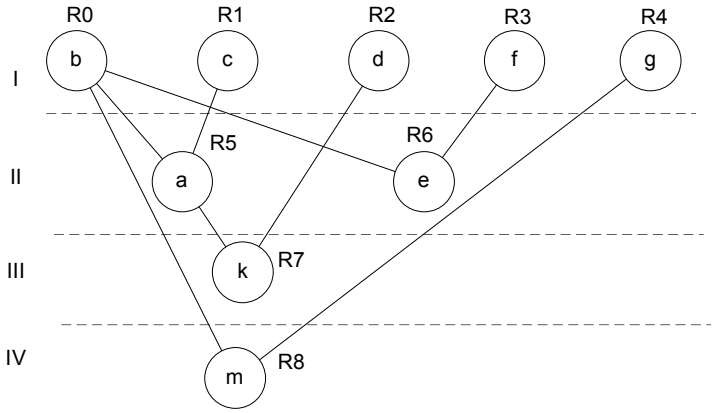


Рис. 12. Ярусно-параллельная форма с распределением регистров

Таким образом распределятся регистры без раскраски. При этом нам потребовалось 9 регистров.

Теперь посмотрим, как можно распределить регистры более целесообразно.

Задача распределения регистров, в которых хранятся данные, сводится к задаче раскраски графа конфликтов. При этом количество цветов определяется количеством регистров целевой системы.

Сформируем граф конфликтов на основе имеющейся ЯПФ.

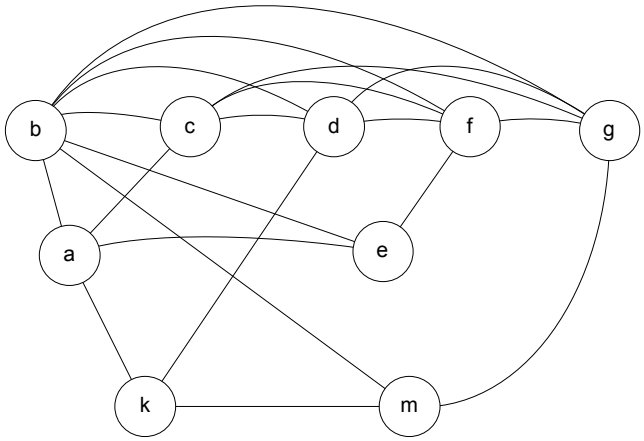


Рис. 13. Исходный граф конфликтов

Раскраска этого графа даст нам следующий результат:

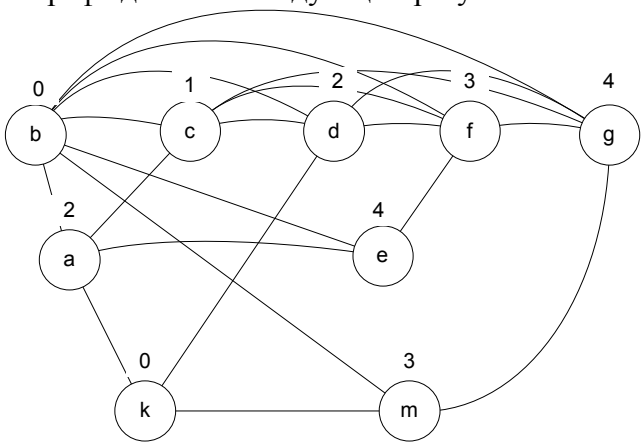


Рис. 14. Раскрашенный граф

Здесь использованы краски с именами 0-4. Итого было использовано 5 цветов, что означает, что мы можем использовать 5 регистров (вместо 9 регистров без раскраски). Возможно, что это – не самый показательный пример. Дело в том, что наш граф конфликтов сильносвязный. В более разреженных графах ситуация с раскраской может оказаться значительно лучше.

4. РЕАЛИЗАЦИЯ ДВОИЧНОГО ТРАНСЛЯТОРА

Общая структура программного комплекса приведена на Рис. 15.

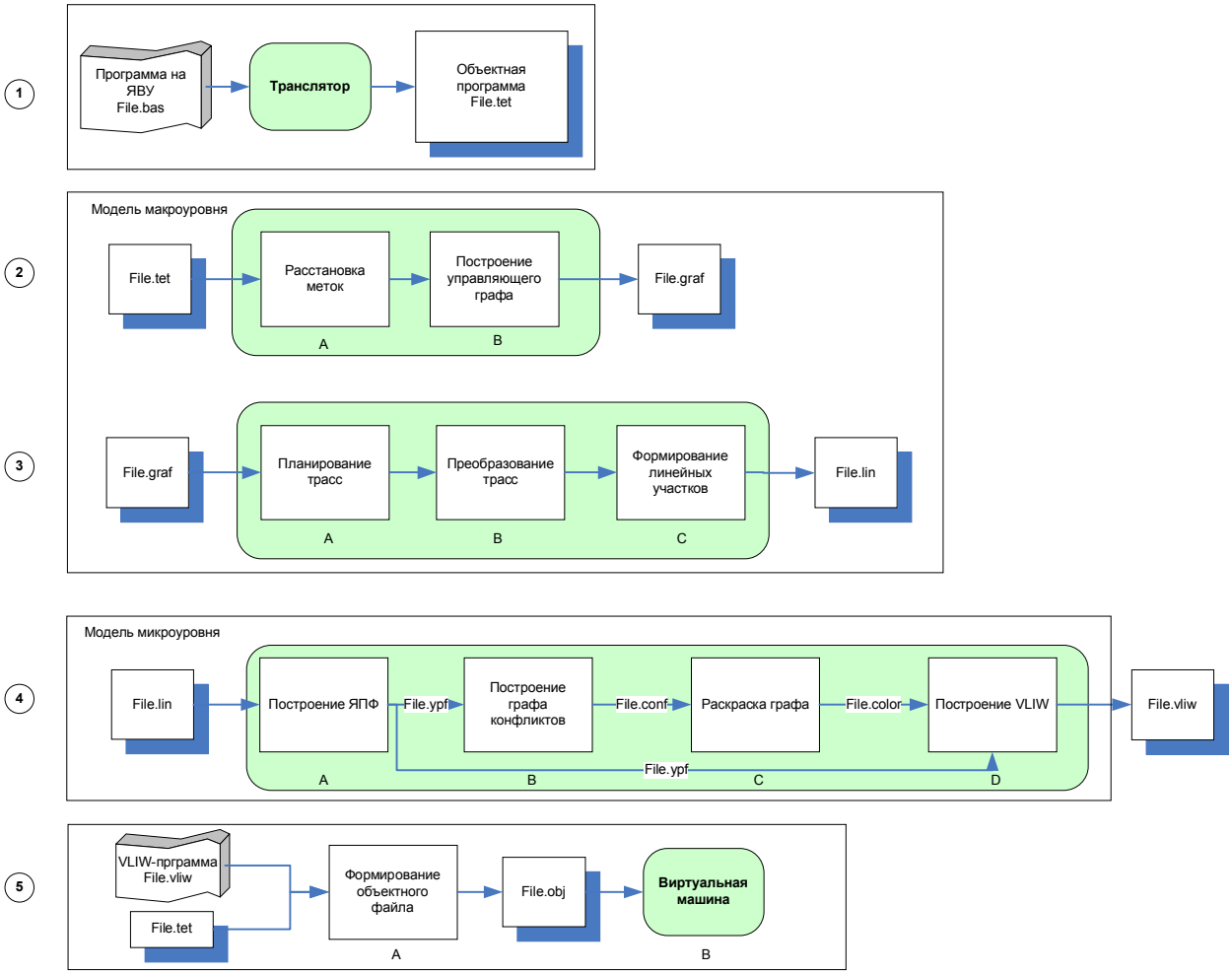
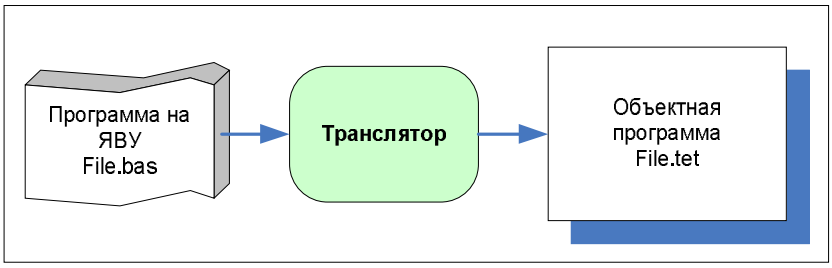


Рис. 15. Структура программного комплекса

Далее рассмотрим более детально его составляющие, полагая, что весь комплекс состоит из независимых модулей, обмен данными между которыми осуществляется с помощью фалов.

4.1. Создание объектного файла



Формат объектного файла

Поскольку мы имеем дело с учебным проектом, то сосредоточимся больше на удобстве и наглядности представления объектного кода. Очевидно, что в памяти виртуальной машины должны быть размещены 3 основных сегмента:

- сегмент текста (хранит исполняемые инструкции);
- сегмент данных (там располагаются глобальные данные);
- сегмент стека (хранит временные переменные).

Очевидно, что формат объектного файла (ОФ) должен быть таким, чтоб в памяти виртуальной машины были размещены и проинициализированы все эти сегменты. При этом будем полагать, что (простоты ради) ОФ представляет собой почти полный образ процесса в памяти.

Кроме того, целесообразно иметь такой формат, который позволил бы хранить как линейный поток инструкций, так и широкие командные слова. Сделано это для того, чтобы упростить работу виртуальной машины.

Исходя из этого, объектный файл будет состоять из 3-х разделов – заголовка, сегмента данных и сегмента текста. Хранить сегмент стека в ОФ не имеет смысла, т.к. память для этого сегмента может выделяться автоматически при загрузке в ВМ. Для этого в заголовке достаточно хранить лишь размер сегмента стека.

Заголовок <HEAD> DATA_LEN TEXT_LEN STACK_LEN Прочая информация </HEAD>
Сегмент данных <DATA> </DATA>
Сегмент текста <TEXT> </TEXT>

Заголовок. Описание заключено в теги <HEAD> и </HEAD> и содержит необходимую управляющую информацию:

- DATA_LEN – размер сегмента данных в словах.

- TEXT_LEN – количество командных слов в сегменте текста.
- STACK_LEN– размер сегмента стека в словах.

Кроме того, заголовок может содержать прочую служебную информацию, определяющую конфигурацию ВМ: количество ФУ, размер регистрового файла и т.п. Это как минимум позволит иметь гибкую систему, не отягощенную множеством файлов конфигурации и настроек.

Сегмент данных. Описание заключено в теги <DATA> и </DATA> и фактически содержит таблицу имен. Количество хранимых имен определено в заголовке (поле DATA_LEN).

Описание каждого имени содержит указание ее вида, типа и собственно значение. Если имя является константой, то припишем признак (0), если имя – переменная, то признак равен 1. Будем считать, что типов данных всего два: число (тип 0) и строка (тип 1). Это описание представляет собой тег (мы проектируем машину именно с теговой защитой).

Имена расположены в сегменте данных в естественном порядке и обращение к ним осуществляется по адресу, начиная с нуля.

Таким образом, запись в сегменте данных – это тройка

(вид, тип, значение)

Сегмент текста. Заключен в теги <TEXT> и </TEXT> и содержит командные слова. При этом в сегменте могут храниться командные слова с произвольным количеством слогов.

Командное слово представлено в виде множества инструкций (слогов) в виде тетрад:

$$(Тетрада_1; Тетрада_2; \ldots Тетрада_n)$$

где

$$Тетрада_i=(OP, A1, A2, R)$$

Пример объектного файла

```
<HEAD>
  DATA_LEN = 6
  TEXT_LEN = 4
  STACK_LEN = 20
  ; Описание архитектуры ВМ
  FU_NUM = 3            ; Количество функциональных устройств в ВМ
  RF_SIZE = 10          ; Количество регистров в регистровом файле
</HEAD>
<DATA>
  (1,0,50)              ; 000: переменная, число, 50
  (1,0,3.14)            ; 001: переменная, число, 3.14
  (0,1,"qwerty")        ; 002: константа, строка символов
  (1,0,1)               ; 003: переменная, 1
  (1,1,"any string")    ; 004: переменная, строка символов
  (0,0,0)               ; 005: константа, число, 0
</DATA>
<TEXT>
  ((+,0,1,6))
  ((+,1,6,6);(*,3,5,7);(:=,1,5,))
  ((out,4,,);(+,7,3,1))
  ((out,1,,))
</TEXT>
```

Примечание: после загрузки ОФ в память первый элемент сегмента данных DATA получает адрес DATA+0, т.е. для обращения к нему нужно указывать ячейку памяти 0 – например ((in,0,,)). Аналогично номера остальных ячеек уменьшаются на единицу.

Образ программы в памяти

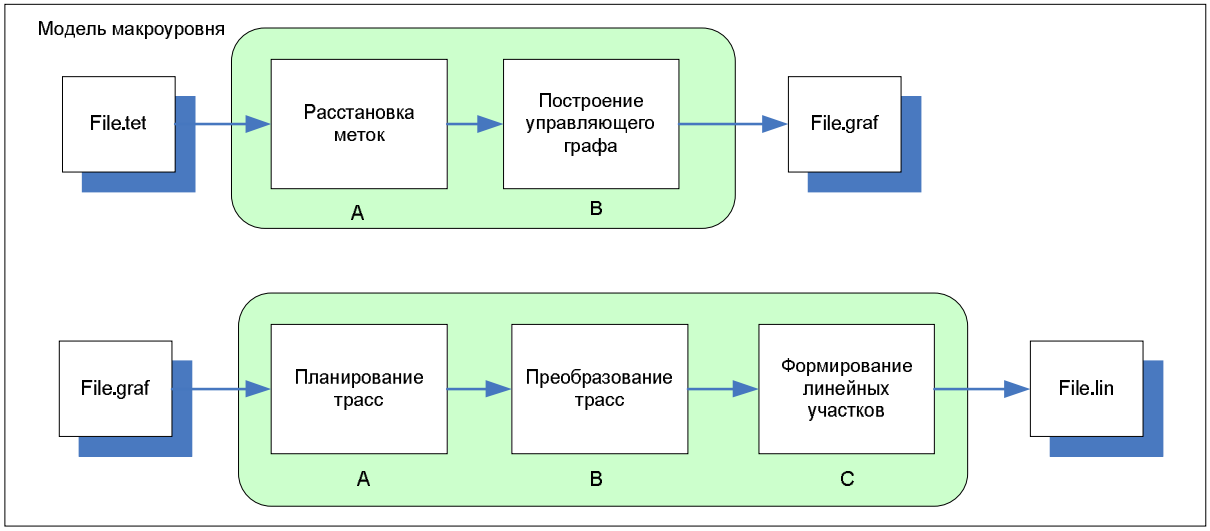
В памяти ВМ мы будем иметь следующее представление объектного кода:

Адрес	**Тег**	
000 001 002 … DATA_LEN-1		**Сегмент данных** DATA_LEN ячеек
DATA_LEN DATA_LEN+1 DATA_LEN+2 … DATA_LEN+STACK_LEN-1		**Сегмент стека** STACK_LEN ячеек
DATA_LEN+STACK_LEN		**Сегмент текста** TEXT_LEN командных слов

Память для сегмента стека будет выделяться автоматически при загрузке программы. Это – задача загрузчика, который может быть выполнен как отдельная компонента. Кроме того, загрузчик должен знать, по какому адресу находится первая исполняемая команда. Это тоже несложно, т.к. вся необходимая информация содержится в заголовке программы.

Организация памяти у нашей ВМ – теговая. Это значит, что загрузчик, помимо формирования значений тегов для сегмента данных, должен будет определить теги для сегмента текста (установив соответствующие атрибуты).

4.2. Модель макроуровня



Алгоритм расстановки меток

Суть алгоритма заключается в том, что просматривается поток тетрад и каждой тетраде приписывается ее вес и тип. Вес определяет вычислительные затраты на выполнение инструкции, а тип (метка) – это либо «ПлохаяИнструкция» (инструкция, которая не может выполняться параллельно с другими), либо развилка (передача управления).

```
-- Расстановка меток для тетрад
Цикл ДляКаждого в СписокТетрад: ТекущаяТетрада
  Если ТекущаяТетрада.Оператор = (OUT | IN | POP | PUSH | POPR | PUSHR) То
    ТекущаяТетрада.Метка := ПлохаяИнструкция
  Иначе
    Если ТекущаяТетрада.Оператор = (BREQ | BRNEQ | ... | BRNE | BRPL) То
      ТекущаяТетрада.Метка := Развилка
    Иначе
      Если ТекущаяТетрада.Оператор = JMP То
        ТекущаяТетрада.Метка := БезусловныйПереход
      Кесли
    Кесли
  КЕсли
КЦикла
```

Алгоритм построения управляющего графа и линейных участков

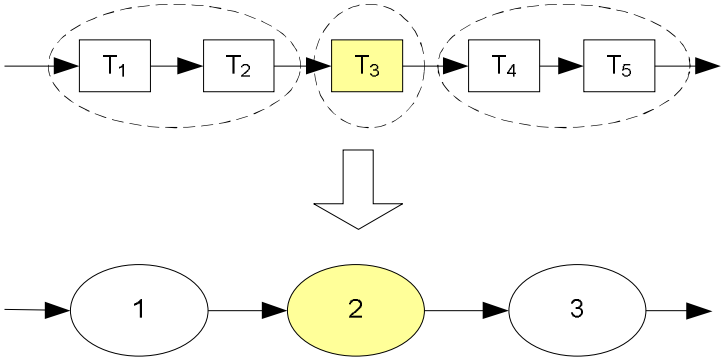
```
-- Формирование управляющего графа
-- Вход: список размеченных тетрад СписокТетрад
-- Выход: управляющий граф УправляющийГраф и множество линейных участков
Цикл ДляКаждого в СписокТетрад: ТекущаяТетрада
  Если (ТекущаяТетрада является стоком) И
       (ТекущаяТетрада.Метка = ХорошаяИнстркция) То
    УправляющийГраф.ДобавитьВершину
    УправляющийГраф.ПоследняяВершина.Дуга1 := номер_следующей_тетрады
  Кесли

  -- Граница линейного участка
  Начало_блока_последней_вершины_УГ := номер_текущей_тетрады
  ТекущаяТетрада.НомерЛинейногоУчастка := номер_последней_вершины_УГ

  Если ТекущаяТетрада.Метка = ПлохаяИнстркция То
    УправляющийГраф.ДобавитьВершину
    УправляющийГраф.ПоследняяВершина.Дуга1 := номер_следующей_тетрады
  Иначе
    Если ТекущаяТетрада.Метка = Развилка То
      УправляющийГраф.ДобавитьВершину
      УправляющийГраф.ПоследняяВершина.Дуга1 := номер_следующей_тетрады
    Иначе
      Если ТекущаяТетрада.Метка = БезусловныйПереход То
        УправляющийГраф.ДобавитьВершину
        УправляющийГраф.ПоследняяВершина.Дуга1 := -1
      КЕсли
    КЕсли
  КЕсли
КЦикла

-- Расстановка дуг управляющего графа
Цикл ДляКаждого в СписокТетрад: ТекущаяТетрада
  Если ТекущаяТетрада.Метка = (Развилка | БезусловныйПереход) То
    Сдвиг := ТекущаяТетрада.Аргумент1
    ТекущаяТетрада.Дуга2 := ТекущаяТетрада.НомерЛинУчастка + Сдвиг
  КЕсли
КЦикла
```

Обратите внимание на то, как алгоритм реагирует на появление «плохой» инструкции. Пусть имеется последовательность тетрад (T_1,…, T_5), причем в этой последовательности встретилась «плохая» тетрада T_3. В этом случае УГ будут сформированы три вершины (и три линейных участка соответственно). Действительно, исходя из смысла линейного участка, тетрады T_1 и T_2, а также T_4 и T_5 потенциально смогут выполняться параллельно, а «плохая» тетрада T_3, по определению, - нет.



Алгоритм построения трасс

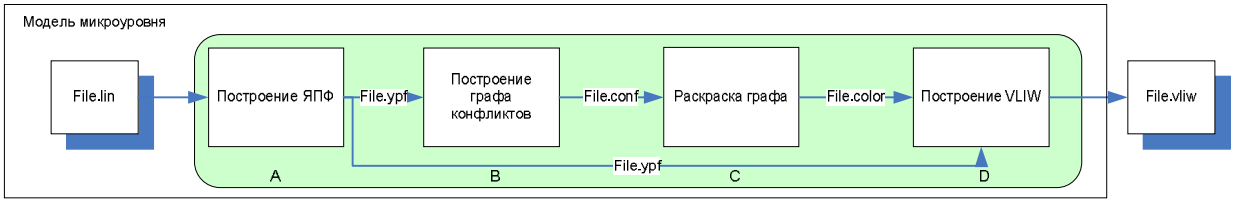
```
Цикл ДляКаждого в УправляющийГраф: ТекущаяВершина
  Если ТекущаяВершина есть в трассе То
    СписокТрасс.ДобавитьТрассу
    Вернуться по списку возврата
  КЕсли

  ТекущаяТрасса.ДобавитьТекущуюВершину
  ТекущаяВершина = есть в трассе

  Если ТекущаяВершина.Дуга2 != -1 То   // переход
    Если ТекущаяВершина.Дуга1 != -1 То // условный переход
      Если ТекущаяВершина.ПерваяТетрада.Оператор = (BREQ | BRNEQ)
        СчетчикЦикла := ТекущаяВершина.Дуга2
      КЕсли
      Если текущий вершины нет в списке возврата То
        Добавить в список возврата
      КЕсли
    Иначе // т.е. безусловный переход
      СчетчикЦикла := ТекущаяВершина.Дуга2
  Иначе
    Если ТекущаяВершина.Дуга1 = -1 То // конец трассы
    СписокТрасс.ДобавитьТрассу
  КЕсли
КЦикла
```

4.3. Модель микроуровня

Преобразование множества линейных участков в множество VLIW.



Вход: множество линейных участков (file.lin)
Выход: множество VLIW (file.vliw)

1. Построение ЯПФ.
   Вход: множество линейных участков (file.lin)
   Выход: ЯПФ (file.ypf);
2. Формирование графа конфликтов.
   Вход: file.ypf,
   Выход: граф конфликтов (file.conf)
3. Раскраска графа конфликтов.
   Вход: file.conf
   Выход: ЯПФ с раскраской графа по регистрам (file.color)
4. Формирование множества VLIW.
   Вход: file.color
   Выход: file.vliw

1. Файл множества линейных участков file.lin

Формат файла:

```
(OP,A1,A2,R)
…
#
```

Пример файла file.lin:

```
(+,,b,c)
(+,a,b,c)
(+,a,b,b)
(+,d,e,f)
(:=,b, ,a)
(:=,b,,f)
(:=,h, ,g)
#
(+,,b,c)
(+,a,b,c)
(+,a,b,b)
(+,d,e,f)
(:=,b, ,a)
(:=,b,,f)
(:=,h, ,g)
#
```

2. Файл ЯПФ file.ypf

Формат файла:

```
(NAME,OP,LEFT,RIGHT,RANK,ID,)
…
#
```

Пример файла file.ypf:

```
(b,,NULL,NULL,0,1,)
(a,,NULL,NULL,0,3,)
(d,,NULL,NULL,0,6,)
(e,,NULL,NULL,0,7,)
(h,,NULL,NULL,0,11,)
(c,+,NULL,1,1,2,)
(f,+,6,7,1,8,)
(g,:=,11,NULL,1,12,)
(c,+,3,1,2,4,)
(b,+,3,1,2,5,)
(a,:=,5,NULL,3,9,)
(f,:=,5,NULL,3,10,)
#
(b,,NULL,NULL,0,1,)
(a,,NULL,NULL,0,3,)
```

```
(d,,NULL,NULL,0,6,)
(e,,NULL,NULL,0,7,)
(h,,NULL,NULL,0,11,)
(c,+,NULL,1,1,2,)
(f,+,6,7,1,8,)
(g,:=,11,NULL,1,12,)
(c,+,3,1,2,4,)
(b,+,3,1,2,5,)
(a,:=,5,NULL,3,9,)
(f,:=,5,NULL,3,10,)
#
```

3. Граф конфликтов file.conf

Содержит множество графов конфликтов, представленными матрицами смежности.

Формат файла:

Матрица смежности графа

```
Количество вершин
Матрица смежности графа
#
…
Количество вершин
Матрица смежности графа
#
```

Пример фрагмента файла file.conf:

```
12
0 1 1 1 1 1 1 0 0 0 1 0
1 0 0 0 0 0 0 1 0 0 0 1
1 0 0 1 1 1 1 0 0 0 1 0
1 0 1 0 1 0 0 0 0 0 0 0
…
1 0 1 0 0 1 1 0 0 0 0 1
0 1 0 0 0 0 0 1 0 0 1 0
#
12
0 1 1 1 1 1 1 0 0 0 1 0
1 0 0 0 0 0 0 1 0 0 0 1
1 0 0 1 1 1 1 0 0 0 1 0
…
1 0 1 0 0 1 1 0 0 0 0 1
0 1 0 0 0 0 0 1 0 0 1 0
#
```

4. Файл ЯПФ с раскраской графа по регистрам file.color

Формат файла:

```
(NAME,OP,LEFT,RIGHT,RANK,ID,COLOR)
…
#
```

Пример файла file.color:

```
(b,,NULL,NULL,0,1,0)
(a,,NULL,NULL,0,3,1)
(d,,NULL,NULL,0,6,2)
(e,,NULL,NULL,0,7,3)
(h,,NULL,NULL,0,11,4)
(c,+,NULL,1,1,2,1)
(f,+,6,7,1,8,0)
(g,:=,11,NULL,1,12,2)
(c,+,3,1,2,4,2)
(b,+,3,1,2,5,3)
(a,:=,5,NULL,3,9,0)
(f,:=,5,NULL,3,10,1)
#
(b,,NULL,NULL,0,1,0)
(a,,NULL,NULL,0,3,1)
(d,,NULL,NULL,0,6,2)
(e,,NULL,NULL,0,7,3)
(h,,NULL,NULL,0,11,4)
(c,+,NULL,1,1,2,1)
(f,+,6,7,1,8,0)
(g,:=,11,NULL,1,12,2)
(c,+,3,1,2,4,2)
(b,+,3,1,2,5,3)
(a,:=,5,NULL,3,9,0)
(f,:=,5,NULL,3,10,1)
#
```

5. Файл множества VLIW file.vliw

Формат файла:

```
[(OP,A1,A2,R) (OP,A1,A2,R)… (OP,A1,A2,R)]
…
#
```

Пример файла file.vliw:

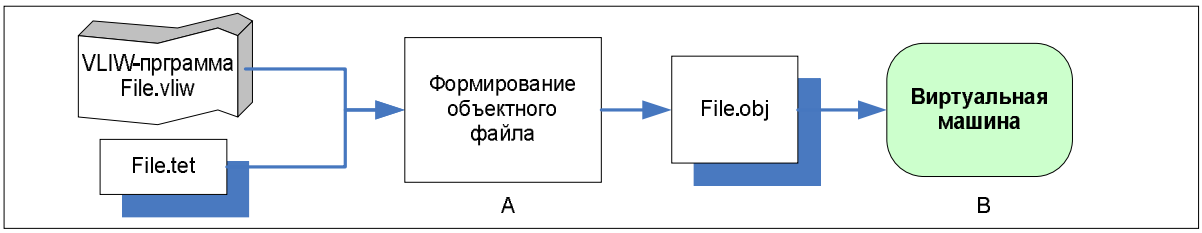
```
[(load,b,,R0)(load,a,,R1)(load,d,,R2)(load,e,,R3)(load,h,,R4)]
[(+,,R0,R1)(+,R2,R3,R0)(:=,R4,,R2)]
[(store,R1,,c)(+,R1,R0,R2)(+,R1,R0,R3)(store,R0,,f)(store,R2,,g)]
[(store,R2,,c)(:=,R3,,R0)(:=,R3,,R1)]
[(store,R0,,a)(store,R1,,f)]
#
```

```
[(load,b,,R0)(load,a,,R1)(load,d,,R2)(load,e,,R3)(load,h,,R4)]
[(+,,R0,R1)(+,R2,R3,R0)(:=,R4,,R2)]
[(store,R1,,c)(+,R1,R0,R2)(+,R1,R0,R3)(store,R0,,f)(store,R2,,g)]
[(store,R2,,c)(:=,R3,,R0)(:=,R3,,R1)]
[(store,R0,,a)(store,R1,,f)]
#
```

4.4. Виртуальная машина. Выполнение программы

Архитектура виртуальной машины

Виртуальная машина (ВМ) должна уметь выполнять как «обычный» поток тетрад, так и поток VLIW.



Особенности ВМ:

1. ВМ содержит N функциональных устройств.
2. Каждое ФУ имеет M регистров общего назначения и регистр флагов.
3. ВМ использует микропрограммное управление.
4. Организация памяти ВМ – теговая
5. Существует общий для всех ФУ регистровый файл.

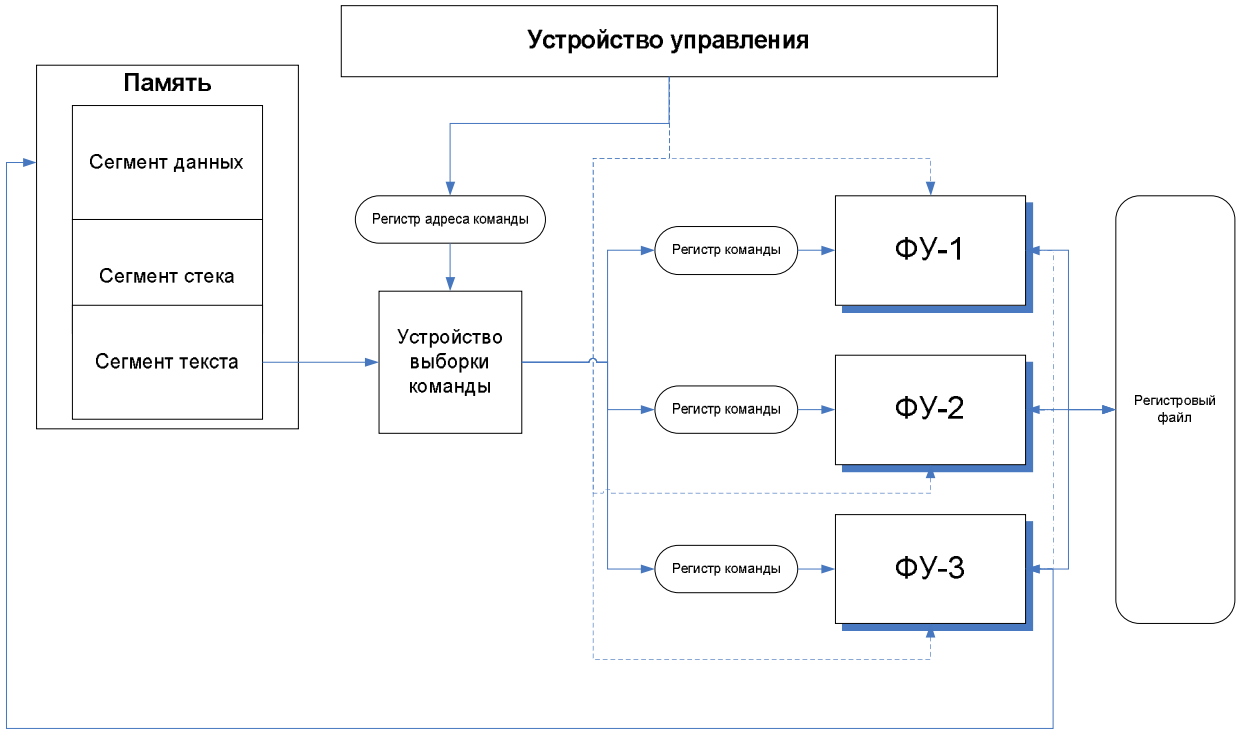


Рис. 16. Архитектура ВМ

Входными данными для работы ВМ являются:

1) Объектный файл, содержащий как линейную последовательность команд (результат работы «обычного» транслятора), так и последовательность широких командных слов (результат работы двоичного транслятора).
2) Конфигурационные файлы виртуальной машины.

Базовые инструкции ВМ

Ниже приведен примерный перечень базовых команд виртуальной машины.

Операции регистр-память		
LOAD A, R	A→R	Загрузка аргумента, находящегося по адресу A, в регистр R
STORE R, A	R→A	Сохранение регистра R в ячейке памяти по адресу A
Операции регистр-регистр		
SET R1, R2	R2 := R1	
Операции память-память		
MOV A1, A2	A2 := A1	Поместить содержимое ячейки памяти A1 в ячейку с адресом A2
CMP A1, A2, R	R := A1?A2	Сравнить содержимое ячеек памяти с адресами A1 и A2. Результат поместить в регистр R
Арифметические операции		
SUM R1, R2, R3	R3 := R1 + R2	
SUB R1, R2, R3	R3 := R1 - R2	
MUL R1, R2, R3	R3 := R1 * R2	
DIV R1, R2, R3	R3 := R1 / R2	
Операции ввода-вывода		
OUT R		Вывод R на “терминал”
IN R		Ввод значения R с “терминала”
Операции управления		
BR ADDR		Безусловный переход по адресу ADDR
BRZ R, ADDR		Условный переход (равно 0)
BRP R, ADDR		Условный переход (больше 0)
BRM R, ADDR		Условный переход (меньше 0)
Прочие операции		
EXIT		Завершить выполнение программы
NOP		Пустая инструкция

Кроме того, бываю полезны (и используются на практике) операции управления, подобные следующим:

jmp <арг1> - безусловный переход на <арг1>
jb <арг1> - условный переход на <арг1> (если Flags[2]=1)
ja <арг1> - условный переход на <арг1> (если Flags[1]=1)
je <арг1> - условный переход на <арг1> (если Flags[0]=1)

jbe <арг1> - условный переход на <арг1> (если Flags[0]=1 или Flags[2]=1)
jae <арг1> - условный переход на <арг1> (если Flags[0]=1 или Flags[1]=1)
jne <арг1> - условный переход на <арг1> (если Flags[0]=0)

mja <арг1>,<арг2>,<арг3> - перейти на метку <арг3>, если значение регистра <арг1> больше значения регистра <арг2>
mjb <арг1>,<арг2>,<арг3> - перейти на метку <арг3>, если значение регистра <арг1> меньше значения регистра <арг2>
mje <арг1>,<арг2>,<арг3>> - перейти на метку <арг3>, если значение регистра <арг1> равно значению регистра <арг2>
mjn <арг1>,<арг2>,<арг3> - перейти на метку <арг3>, если значение регистра <арг1> не равно значению регистра <арг2>
mj <арг1> - Безусловный переход на метку <арг1>

Микропрограммное управление

На вход ВМ могут поступать команды, которые представляют собой имя микропрограммы, т.е. разворачивающиеся в последовательность базовых инструкций ВМ (или микрокоманд). Файл с описанием таких команд загружается во время инициализации ВМ.

Формат описания микрокоманд:

mc <имя микропрограммы> <Арг1>, <Арг2>, <Арг3>

Например, введем микропрограмму сложения двух чисел, находящихся в памяти по заданным адресам. Описание такой микропрограммы может выглядеть так:

```
mc "+" A1 A2 A3
  load A1,RA
  load A2,RB
  sum RA, RB, RC
  store RC, A3
#
```

Если в потоке тетрад встретится команда «(+, 101, 240, 012)», то она будет развернута в следующую последовательность базовых инструкций:

(+, 101, 240, 012):

```
  load 101,RA
  load 240,RB
  sum RA, RB, RC
  store RC, 012
```

ПРИЛОЖЕНИЯ

1. ИСПОЛЬЗОВАНИЕ ПРОЛОГА

Многие компоненты проекта целесообразно реализовывать на языке Пролог. Достоинства Пролога очевидны: на этом языке достаточно просто (для программиста, разумеется) реализуются многие поисковые процедуры. Недостатки же заключаются в том, что на Прологе плохо реализуются процедурные, императивные механизмы. Это проистекает из самой природы Пролога, как декларативной системы. Отсюда заключаем, что, например, сканер лучше писать на процедурном языке, а синтаксический анализатор - на Прологе.

Синтаксический анализатор и генерация объектного кода

О создании синтаксического анализатора говорится в первой части курса – в классической теории компиляторов. Там приведен пример анализатора арифметических выражений. В том случае, когда мы решаем задачу создания транслятора программ с более сложной грамматикой, все происходит абсолютно аналогично. Для этого мы создаем предикаты, соответствующие правилам грамматики более высокого уровня:

Например, пусть нам необходимо реализовать такую конструкцию, как арифметический цикл. Правило грамматики для цикла может выглядеть так:

$$op \rightarrow FOR\ i:=N0\ to\ Nk\ do\ OP$$

Реализация анализатора на Прологе достаточно очевидна:

```
% op → FOR i:=N0 to Nk do OP
op(L,PF) :-
     a8(["for"], L2, [":="], L3, ["to"], L4, ["do"],L5, L),
     isitname (L2,L2PF),
     E(L3,L3PF),
     E(L4,L4PF),
     op(L5,L5PF).
```

Однако также очевидно, что этого явно не достаточно. Нам необходимо сформировать промежуточную форму (например, польскую), в которой имеются адреса переходов. Поэтому сначала распишем цикл через *if* и *goto*.

```
          i:=N0
     L1:  if i>Nk goto L2
          OP
          i:=i+1
          goto L1:
     L 2:
```

Именно эту конструкцию и следует реализовывать на Прологе. Соответствующие тетрады будут выглядеть так:

```
     :=, i, N0,,
     L1:,
     -, i, Nk, T1
     T1, L2, $BRP
     OP
     INC, i, 1,
```

```
$BR, L1, ,
L2:
```

Еще одно замечание касается **формирования имен меток**.

Для каждого перехода должна быть своя уникальная метка. "Глобальность" Пролог, вообще говоря, не поддерживает. Ввести же понятие глобальной переменной можно путем использования базы фактов. Например, например, некоторый глобальный счетчик можно реализовать в виде факта базы данных Пролога. Для работы с базой данных в Прологе используются предикаты `assert` и `retract`:

```
assert(cnt(0)),
…
retract(cnt(OLD)),
NEW=OLD+1,
assert(cnt(NEW)).
```

Возможен и другой способ: снабдить соответствующие предикаты дополнительным аргументом Label:

```
op(L, PF, Label):- …
```

Однако этот путь приводит к ухудшению читаемости программы и увеличению ее громоздкости.

Поиск путей в графе

Рассмотри классическую задачу поиска путей в ориентированном графе. К этой задаче сводится множество прикладных задач совершенно разного характера – от анализа систем до их синтеза. Среди множества существующих методов и алгоритмов поиска путей мы рассмотрим реализацию этой задачи на языке Пролог.

Итак, пусть дан некоторый орграф:

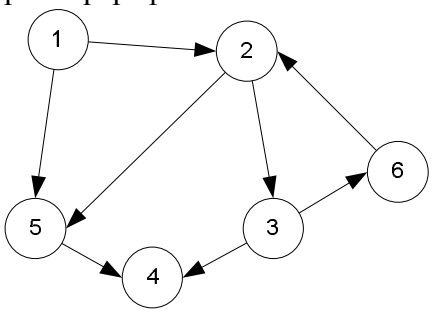


Рис. 17. Орграф

Основной предикат поиска путей может выглядеть так:

```
way(X,Y) :- a(X,Y).
way(X,Y) :- a(X,Z), way(Z,Y).
```

Это – сугубо "теоретический" предикат, отвечающий на вопрос, существует ли путь вообще. Нас же интересует получение пути в виде, скажем, списка вершин. Для этого введем еще один аргумент: way(from,to,**WL**).

```
way(X,Y,[Y]) :- a(X,Y).
way(X,Y,[Z|WZ]) :- a(X,Z), way(Z,Y,WZ).
```

Итоговая программа приведена ниже:

```
goal
   fwall.
clauses
   fwall :- finways(X),fail.
   fwall :- !.
```

```
findways(FROM) :- way (FROM,X,W), write(FROM), print(W),fail.
print ([ ]) :- nl.
print ([H|]) :- write (H), print (T).

way(X,Y,[Y]) :- a(X,Y).
way(X,Y,[Z|WZ]) :- a(X,Z), way(Z,Y,WZ).

a(1,2).
a(1,5).
a(2,3).
a(3,4).
a(2,5).
a(5,4).
a(3,6).
a(6,2).
```

Очевидно, что такая программа работать не будет, т.к. в графе присутствует цикл и программа будет осуществлять бесконечный перебор вариантов. Проблему можно решить, отслеживая повторяющиеся вершины. Для этого будем использовать предикат *isin()*, проверяющий наличие некоторого элемента в списке.

```
isin(E,[E|_]).
isin(E,[_|Tail]) :- isin(E,Tail).
```

Переписываем основной предикат поиска пути, добавляя в него еще один аргумент, хранящий старый путь, т.е. список вершин, по которым мы уже проходили:

и переписываем основной предикат:

```
way2(X,Y,[Y],OLD) :- a(X,Y),not(isin(Z,OLD)).
way2(X,Y,[Z|WZ],OLD) :- a(X,Z),not(isin(Z,OLD)),
ways(Z,Y,WZ,[Z|OLD]).
```

Таким образом, получаем почти окончательный вариант программы:

```
goal
      fwall.

clauses
      fwall :- findways(X), fail.
      fwall :- !.

      findways(FROM) :-
            way2(FROM,X,W,[FROM]), write(FROM," "), print(W), fail.
            findways(_).

      isin(E,[E|_]).
      isin(E,[_|Tail]) :- isin(E,Tail).

      print ([]) :- nl.
      print ([H|Tail]) :- write(H," "), print (Tail).

      way2(X,Y,[Y],OLD) :- a(X,Y), not(isin(Y,OLD)).

      way2(X,Y,[Z|W2],OLD) :-
            a(X,Z),
            not(isin(Z,OLD)),
            way2(Z,Y,W2,[Z|OLD]).
```

```
way(X,Y,[Y]) :- a(X,Y).
way(X,Y,[Z|W2]) :-
      a(X,Z),
      way(Z,Y,W2),
      not(isin(Z,W2)).

a(1,2).
a(1,5).
a(2,3).
a(3,4).
a(2,5).
a(5,4).
a(3,6).
a(6,2).
```

Взяв эту программу за основу, можно отыскивать в управляющем графе интересующие нас пути – трассы. Эти трассы, применяя различного рода эвристики, уже можно будет оценивать. Для этого, скажем, можно не только указывать наличие пути между вершинами, но и добавить еще один аргумент в предикат *a()* – вес пути.

2. РАСКРАСКА ГРАФА

Определения

Правильной k–раскраской графа G=(V,E) в k цветов называется такое разбиение множества V его вершин на k непересекающихся подмножеств $V_1,\ldots,V_k$, что никакие 2 вершины из одного и того же подмножества не смежны.

Хроматическим числом $\chi(G)$ графа G называется наименьшее k, при котором G имеет правильную k–раскраску.

Граф G *k–раскрашиваем*, если $\chi(G)\le k$ и *k–хроматичен*, если $\chi(G)=k$.

Если n есть число вершин графа, то граф, очевидно, имеет n–раскраску.

Хроматическое число $\chi(G)$ полного графа равно n.

Граф бихроматичен (2–хроматичен) тогда и только тогда, когда он не содержит нечетных простых циклов.

Следствие 1. Любое дерево бихроматично.

Следствие 2. Любой двудольный граф бихроматичен.

Если наибольшая из степеней вершин графа G равна p, то этот граф p+1–раскрашиваем.

Каждый планарный граф 5–раскрашиваем.

Приближенные методы раскраски графа

Нахождение оптимальной раскраски – это NP–полная задача. Поэтому чаще всего реализуют алгоритмы поиска субоптимального решения.

Последовательная раскраска

Пусть дано упорядоченное множество вершин графа $v_1,\ldots,v_n$.

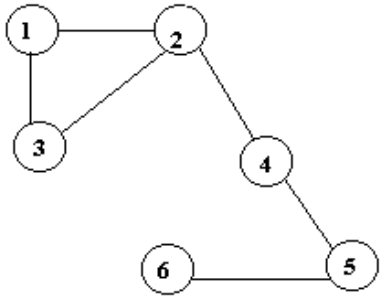
1) вершине v_1 приписываем цвет c_1;
2) если подграф $H(v_1,\ldots,v_{i-1})$, порожденный вершинами $v_1,\ldots,v_{i-1}$ k'–раскрашен, $k\le i-1$, то вершина v_i получает цвет c_m, где $m\le k+1$, т.е. цвет с наименьшим номером, не встречающимся на смежных с v_i вершинах.

Число цветов k при этом заранее не фиксируется. Этот алгоритм дает точную k–раскраску только для полных k–дольных графов.

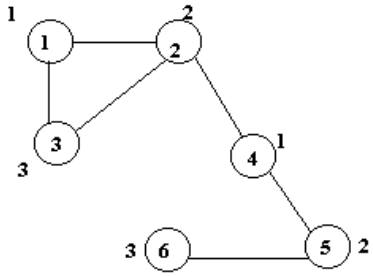
k–дольным называется граф, множество вершин которого можно разбить на k непересекающихся подмножеств $X_1,\ldots,X_k$ так, что никакие 2 вершины из подмножества X_i, i=1,..,n, не смежны.

k–дольный граф называется *полным k–дольным*, если каждая вершина из множества X_i смежна с каждой вершиной из X_j, $i \neq j$.

Пусть дан следующий граф:



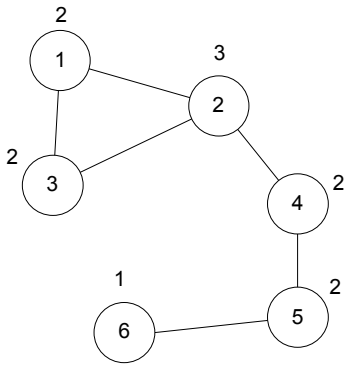
Проведя последовательную раскраску, получим следующее распределение цветов:



Стратегии последовательных раскрасок

Существуют 2 основные стратегии раскраски: НП– и ПН – стратегии. Рассмотрим их на примере раскраски приведенного ранее графа конфликтов:

НП–стратегия («Наибольшие–Первыми»). Упорядочить вершины $v_1,\ldots,v_n$ по убыванию их степеней связности, т.е. сначала раскрашиваются вершины с максимальными степенями.

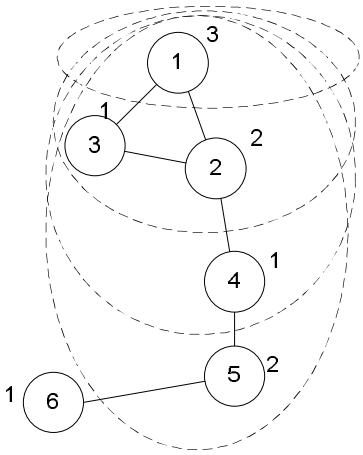


В данном случае упорядочивание может выглядеть так: {2,1,3,4,5,6}. Поэтому раскраску начнем с вершины $V_1=2$.

ПН–стратегия («Последними–Наименьшие»)

a) для n=|V| в качестве v_n выбирается вершина минимальной степени в G;
b) для i=n–1,n–2,…,2,1 в качестве v_i выбирается вершина минимальной степени в подграфе $H(V\backslash\{v_n,\ldots,v_{i+1}\})$.

Выберем вершину минимальной связности: $V_6=6$. Далее рассматриваем граф, где нет 6-й вершины. В этом графе $V_5=5$. Далее в оставшемся графе определим $V_4=4$, затем $V_3=1$, $V_2=2$ и $V_1=3$.


1
3
3
1
2
2
4
1
5
2
6
1

СПИСОК ЛИТЕРАТУРЫ

1. Babayan B.A. Main principles of E2k architecture // Free Software Magazine. 2002, Vol. 1, No. 2
2. Baraz L. et al. IA-32 Execution Layer: a Two Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-based Systems. Proceedings of the 36th International Symposium on Microarchitecture, 2003
3. Dehnert J.C., Grant B.K., Banning J.P., Johnson R., Kistler T., Klaiber A. and Mattson J. The transmeta code morphing software: using speculation, recovery and adaptive retranslation to address real-life challenges. Proceedings of the International Symposium on Code Generation and Optimization, 2003
4. Волконский В.Ю., Ким А.К. Развитие идей параллелизма в архитектуре вычислительных комплексов серии «Эльбрус» // PACO'2008, Труды четвертой международной конференции
5. Воронов Н.В., Гимпельсон В.Д., Маслов М.В., Рыбаков А.А., Сюсюкалов Н.С. (ЗАО «МЦСТ»), Система динамической двоичной трансляции x86 → «Эльбрус». Новости ВПК. – http://www/vpk.name/preview/?!=wys4ru3s
6. Гимпельсон В.Д. Конвейеризация циклов в двоичном динамическом трансляторе. – «Вопросы радиоэлектроники», серия ЭВТ, 2009, вып. 3
7. Гимпельсон В.Д. Сокращение длины критического пути циклических и ациклических участков в динамическом двоичном оптимизирующем трансляторе для архитектуры «Эльбрус». // Научные труды XXXIV Международной молодёжной научной конференции «Гагаринские чтения», М., МАТИ, 2008
8. Дроздов А.Ю., Новиков С.В., Шилов В.В. Эффективный алгоритм преобразования потока управления в поток данных. – «Информационные технологии», Приложение, 2005, №2, с. 24-31
9. Евстигнеев В.А. Применение теории графов в программировании /Под ред. А.П.Ершова –М.: Наука, 1985 –352с.
10. Ким А.К. Развитие архитектуры вычислительных комплексов серии «Эльбрус» // Сб. научных трудов ИТМ и ВТ / Под ред. Калинина С.В. – М: ИТМ и ВТ им. С.А. Лебедева РАН. 2008. № 1. С. 22-27
11. Королев Л.Н. Структуры ЭВМ и их математическое обеспечение. –М.: Наука, 1978.
12. Кузьминский М. Отечественные микропроцессоры: Elbrus E2k // Открытые системы. 1999. № 05-06
13. Липаев В.В. Документирование и управление конфигурацией программных средств: Методы и стандарты. М.: СИНТЕГ, 1998. -212 с.
14. Липаев В.В. Надежность программных средств. -М.: СИНТЕГ, 1998. -232 с.
15. Масич Г.Ф. МВК "Эльбрус-2" URL: http://www.icmm.ru/~masich/win/lexion/elbrus2/elbrus2.htm
16. Таненбаум Э. Архитектура компьютера 4-е издание М., 2003, 697 с.
17. Шувиков С.В. Методы и алгоритмы распараллеливания объектного кода для процессоров с программным управлением функциональными устройствами. Рязань, 2000. Рязанская государственная радиотехническая академия.

Серверы в Internet:

- www.mcst.ru
- www.el2000.ru