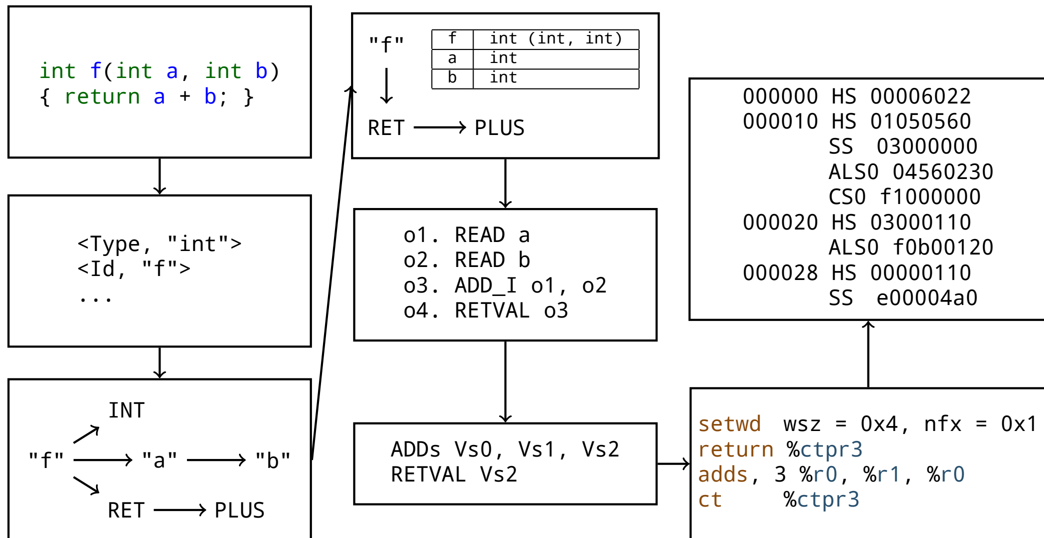
# Проектирование компиляторов

Лекция 7.
Создание целевого кода.

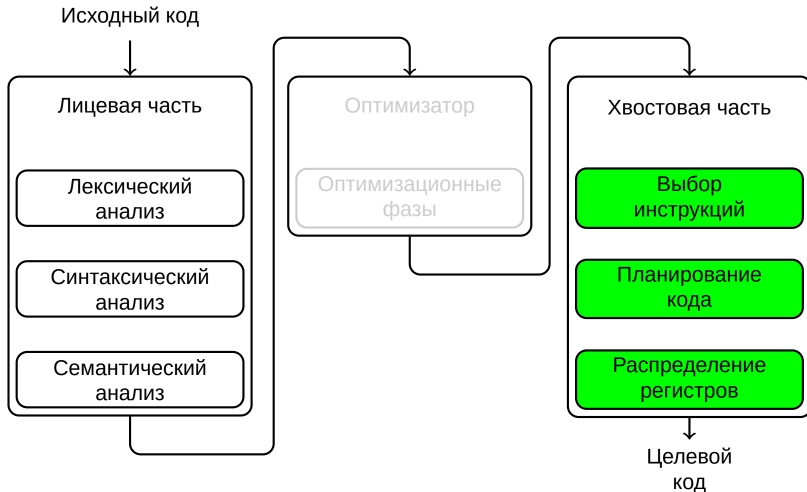
*Маркин А. Л.*
alexanius@gmail.com

2021

# Компиляция программы



# Компиляция программы



# Проблема

Для исполнения программы необходимо перевести код из промежуточного представления в целевой ассемблер:

```
o1. LDd   Vd1,  Vd2, Vd3
o2. LDd   Vd4,  Vd2, Vd5
o3. FMULd Vd3,  Vd5, Vd7
o4. LDd   Vd8,  Vd2, Vd10
o5. FADDd Vd10, Vd7, Vd11
o6. STd   Vd12, Vd2, Vd11
```

→

```
{
  ldd, 0  %r0, %r1, %r2
  ldd, 2  %r3, %r1, %r4
  ldd, 5  %r5, %r1, %r6
  nop 3
}
{
  fmuld, 5 %r2, %r4, %r2
  nop 8
}
{
  faddd, 5 %r2, %r6, %r2
  nop 8
}
{
  std, 5 %r7, %r1, %r2
}
```

# Исполнение инструкций

Между началом исполнение инструкции и выработкой её результата должно пройти определённое время. Это время называется **задержкой**.

Попытка использовать неготовый результат приведёт к блокировке конвейера и ожиданию выработки результата. Т.е. процессор будет простаивать вместо исполнения других инструкций.

# Исполнение инструкций

Типичные задержки для инструкций для процессора Эльбрус:

| Инструкция | Задержка (такты) | Назначение |
| --- | --- | --- |
| add[sd] | 1 | Целочисленное сложение |
| ld[sd] | 3-100 | Загрузка значения из памяти |
| mul[sd] | 4 | Целочисленное умножение |
| disp | 5 | Подготовка перехода |
| fmul[sd] | 8 | Плавающее умножение |

# Исполнение инструкций

Если процессор исполняет инструкции в порядке, предложенным компилятором, то такой процессор использует подход **поочерёдного исполнения** (In Order InO).

Большинство современных процессоров общего назначения умеет самостоятельно выбирать готовые к исполнению инструкции, даже если они идут не последовательно. Такой подход называется **внеочередное исполнение** (Out of Order, OoO).

Внеочередное исполнение инструкций позволяет уменьшить потери от блокировок исполнения, но оно усложняет проектирование процессора.

# Исполнение инструкций

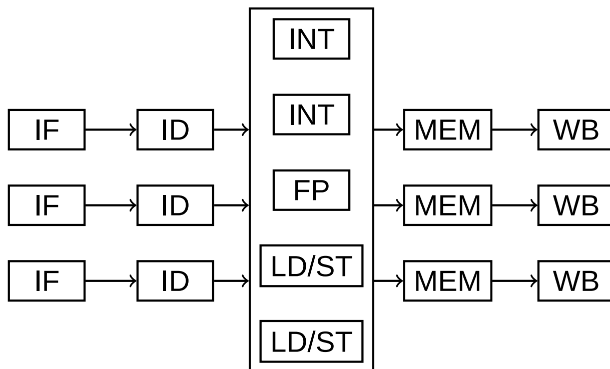
Производительность процессоров основывается не только и не столько на количестве исполняемых инструкций в секунду, сколько на возможности исполнять множество операций единовременно. Такая возможность называется **параллелизмом**.

Параллелизм бывает разных уровней, в данной лекции будет рассматриваться **параллелизм уровня инструкций** (ILP — instruction level parallelism).

Одной из характеристик такого параллелизма является среднее количество инструкций, исполняемое за один такт **Instructions per Cycle, IPC**.

# Исполнение инструкций

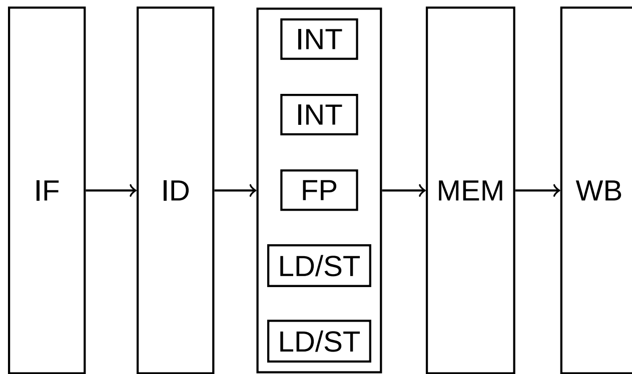
Параллелизм уровня инструкций достигается за счёт добавления в ядро процессора дополнительных исполняющих устройств, работающих параллельно:




Такие процессоры называются **суперскалярными**.

# Исполнение инструкций

Другим вариантом наращивания параллелизма уровня инструкций является появление архитектуры с **широким командным словом** (Very Long Instruction Word, VLIW):



# Исполнение инструкций

Парк устройств процессора Эльбрус:

| | |
|---|---|
| Int, FP, Vect, LD, Cmp | Int, FP, Vect, LD, Cmp |
| Int, FP, Vect, Cmp | Int, FP, Vect, Cmp |
| Int, LD, ST, FP* | Int, LD, ST, Div/Sqrt, FP* |

CT

PL | PL | PL

QP | QP | QP | QP | QP | QP

APB | APB | APB | APB

LIT32 | LIT32 | LIT32 | LIT32

# Исполнение инструкций

Суперскалярные процессоры:

- Динамическое планирование операций
- Сложное аппаратное исполнение планировщика
- Меньший размер кода
- Бинарная совместимость кода

Процессоры с широким командным словом:

- Статическое планирование операций
- Сложное исполнение компилятора
- Упрощённое исполнение процессора
- Ухудшенная бинарная совместимость кода

# Создание кода

# Создание кода

В задачу генерации кода входит:

- Подбор целевых инструкций
- Оптимальное планирование инструкций
- Обеспечение корректности спланированного кода
- Оптимальное распределение регистров

# Выбор инструкций

# Проблема

Необходимо преобразовать операции промежуточного представления в инструкции целевой машины:

```
o1. LDd   Vd1,  Vd2, Vd3              ldd   Vd1,  Vd2, Vd3
o2. LDd   Vd4,  Vd2, Vd5              ldd   Vd4,  Vd2, Vd5
o3. FMULd Vd3,  Vd5, Vd7              fmuld Vd3,  Vd5, Vd7
o4. LDd   Vd8,  Vd2, Vd10  ────────→  ldd   Vd8,  Vd2, Vd10
o5. FADDd Vd10, Vd7, Vd11             faddd Vd10, Vd7, Vd11
o6. STd   Vd12, Vd2, Vd11             std   Vd12, Vd2, Vd11
```

Таким преобразованием занимается стадия **выбора инструкций**.

# Выбор инструкций

После выбора инструкций в представлении уже появляются целевые инструкции, но при этом всё ещё сохраняются виртуальные регистры.

Не все операции промежуточного представления присутствуют в целевой машине, поэтому выбор инструкций должен уметь раскрывать их в существующие.

Иногда для конкретной машины или конкретной версии системы команд приходится создавать отдельные правила выбора с учётом особенностей платформы.

# Планирование

# Проблема

Необходимо переставить инструкции, упаковать в широкие команды, определить время исполнения и распределить их по исполняющим устройствам оптимальным образом:

```
ldd   Vd1,  Vd2, Vd3
ldd   Vd4,  Vd2, Vd5
fmuld Vd3,  Vd5, Vd7
ldd   Vd8,  Vd2, Vd10
faddd Vd10, Vd7, Vd11
std   Vd12, Vd2, Vd11
```

→

```
{
  ldd, 0   Vd1,  Vd2, Vd3
  ldd, 2   Vd4,  Vd2, Vd5
  ldd, 5   Vd8,  Vd2, Vd10
  nop 3
}
{
  fmuld, 5 Vd3,  Vd5, Vd7
  nop 8
}
{
  faddd, 5 Vd10, Vd7, Vd11
  nop 8
}
{
  std, 5   Vd12, Vd2, Vd11
}
```

# Зависимости

Возможность перестановки операций определяется наличием **зависимостей** между ними.

Если операция `o1` обязана исполняться перед операцией `o2`, то говорят что `o1` зависит от `o2`:

```
fmuld Vd3,  Vd5, Vd7
faddd Vd10, Vd7, Vd11
```

Зависимости бывают по данным, по управлению, межитерационные.

# Зависимости

Зависимости по данным бывают четырёх типов:

1. **Чтение после записи** (Read after Write, RAW), также называется **истинной** зависимостью:

   ```
   fmuld Vd0, Vd1, Vd2
   faddd Vd3, Vd2, Vd4
   ```

2. **Запись после записи** (Write after Write, WAW), также называется **выходной** зависимостью:

   ```
   fmuld Vd0, Vd1, Vd2
   faddd Vd3, Vd4, Vd2
   ```

# Зависимости

3. **Запись после чтения** (Write after Read, WAR), также называется **антизависимостью**:

```
fmuld Vd0, Vd1, Vd2
faddd Vd3, Vd4, Vd0
```

4. **Чтение после чтения** (Read after Read, RAW), также называется **входной** зависимостью, на практике не имеет смысла:

```
fmuld Vd0, Vd1, Vd2
faddd Vd0, Vd3, Vd4
```

# Зависимости

Зависимости бывают не только по данным:

```
std Vs0, Vs1, Vs2
ldd Vs3, Vs4, Vs5
```

# Зависимости

Зависимости бывают не только по данным:

```
std Vs0, Vs1, Vs2
ldd Vs3, Vs4, Vs5
```

Во время компиляции неизвестно будут ли чтение и запись работать по одному адресу, т.е. будет ли адрес `Vs0 + Vs1` равен адресу `Vs3 + Vs4`, а значит менять их местами нельзя.

# Планирование

Зачем менять порядок исполнения команд?

```
ldd   Vd1,  Vd2, Vd3
ldd   Vd4,  Vd2, Vd5
fmuld Vd3,  Vd5, Vd7
ldd   Vd8,  Vd2, Vd10
faddd Vd10, Vd7, Vd11
std   Vd12, Vd2, Vd11
```

Такты при последовательном исполнении

```
0:  ldd   Vd1,  Vd2, Vd3
1:  ldd   Vd4,  Vd2, Vd5
4:  fmuld Vd3,  Vd5, Vd7
5:  ldd   Vd8,  Vd2, Vd10
12: faddd Vd10, Vd7, Vd11
20: std   Vd12, Vd2, Vd11
```

Такты после планирования

```
0:  ldd   Vd1,  Vd2, Vd3
0:  ldd   Vd4,  Vd2, Vd5
0:  ldd   Vd8,  Vd2, Vd10
3:  fmuld Vd3,  Vd5, Vd7
11: faddd Vd10, Vd7, Vd11
19: std   Vd12, Vd2, Vd11
```

При этом если загрузки приводят к блокировке исполнения, фактическое время работы последовательного кода будет значительно больше.

# Планирование

**Планирование инструкций** — упорядочивание операций на линейном участке или в процедуре так чтобы были выдержаны все задержки.

На вход планировщику подаётся частично упорядоченный список операций целевой машины, на выходе он выдаёт упорядоченный список операций.

**Оптимальное планирование** — планирование инструкций таким образом чтобы все выходы из линейного участка стояли как можно раньше.

# Планирование

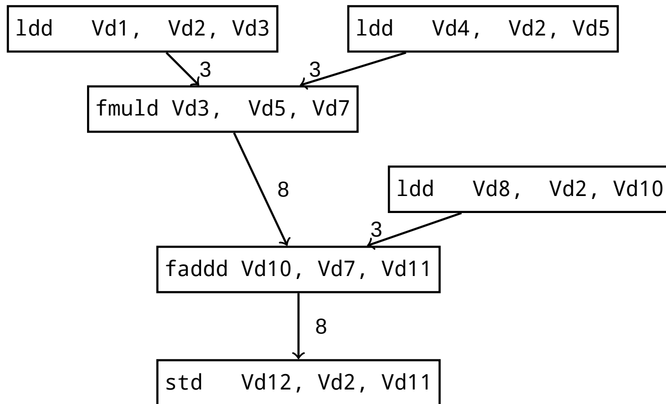
Для выполнения планирования необходимо описать последовательность исполнения операций. Это делается при помощи **графа зависимостей**.

Граф зависимостей в качестве узлов содержит операции линейного участка, а дугами являются зависимости одной операции от другой.

Дуги графа зависимостей имеют вес, соответствующий задержке между двумя операциями.

# Планирование

Пример графа зависимостей:



# Планирование

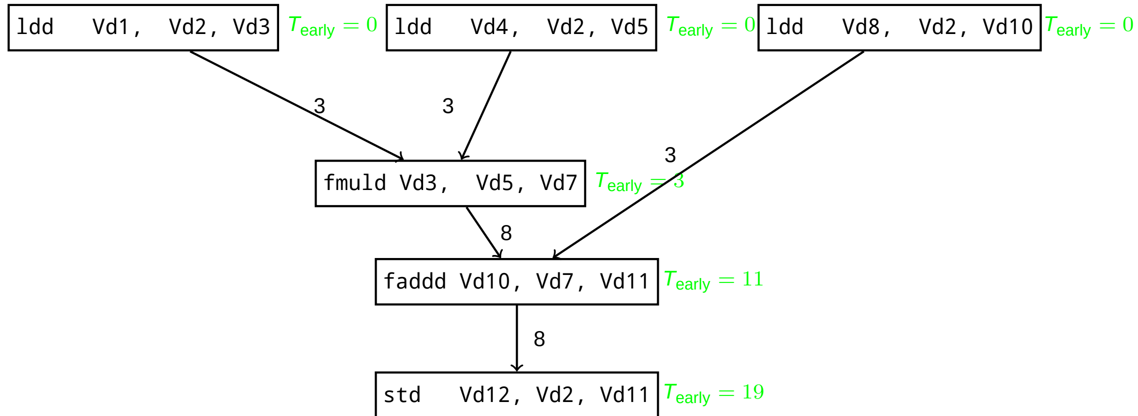
Время **раннего планирования** операции — минимальный номер такта в котором может быть спланирована операция, при котором не нарушается ни одна из длительностей в графе зависимостей.

$$T_{\text{early}} = \begin{cases} 0, \mathit{pred}(\mathit{op}) = \varnothing \\ \max(T_{\text{early}} + \mathit{latency}(\mathit{op}_{\text{cur}}, \mathit{op}_{\text{pred}})) \end{cases}$$

Попытка поставить операцию раньше времени раннего планирования приведёт к блокировке конвейера.

# Планирование

Раннее планирование:



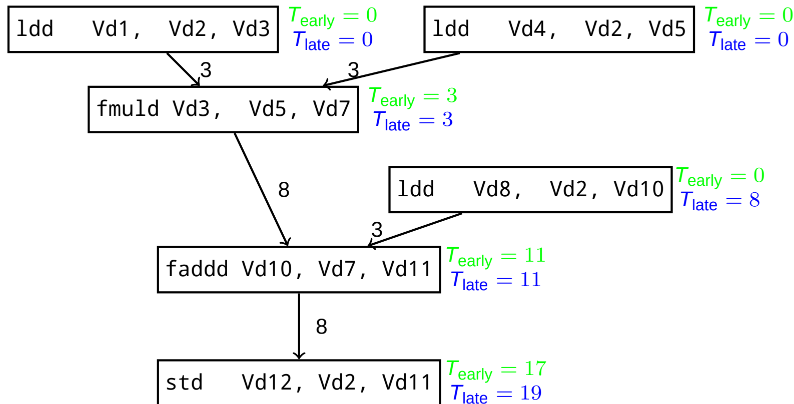
# Планирование

Время **позднего планирования** операции — максимальный номер такта, в котором может быть спланирована операция, при котором не нарушается ни одна из длительностей в графе зависимостей.

$$T_{\text{late}} = \begin{cases} T_{\text{early}}, succ(op) = \varnothing \\ \min(T_{\text{late}} - latency(op_{\text{cur}}, op_{\text{succ}})) \end{cases}$$

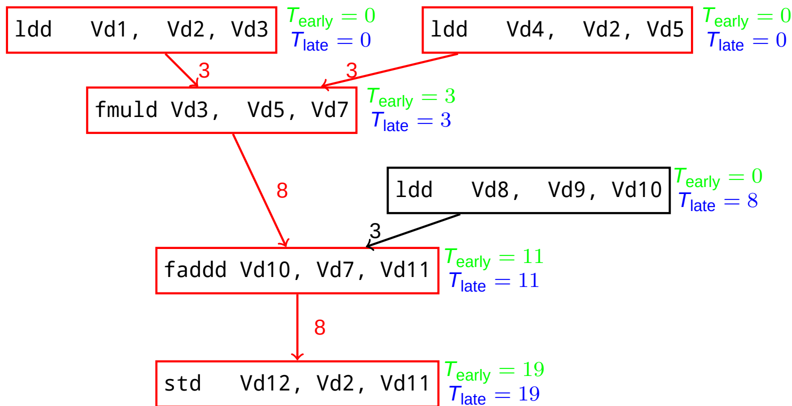
# Планирование

Позднее планирование:



# Планирование

**Критический путь** — операции, для которых $T_{\text{early}} = T_{\text{late}}$




Критический путь определяет время работы линейного участка.

# Планирование

Для выполнения планирования необходимо:

- Построить граф зависимостей
- Избавиться от антизависимостей (при помощи переименования регистров):

```
fmuld Vd0, Vd1, Vd2        ⟶        fmuld Vd0, Vd1, Vd2
faddd Vd3, Vd4, Vd0                 faddd Vd3, Vd4, Vd5
```

- Присвоить операциям приоритеты (наиболее важные планируются раньше)

# Планирование

Алгоритм планирования использует 2 списка:

1. Список **готовых** операций — операции, аргументы которых уже готовы, и они могут исполняться на текущем такте
2. Список **активных** операций — операции, вызванные в предыдущем такте, но всё ещё неготовые

В начале алгоритма все операции из корней графа зависимостей помещаются в список готовых операций

# Планирование

Прямой алгоритм планирования:

1. Из списка готовых операций выбираем те, которые будем исполнять на данном такте и помещаем их в список активных
2. Смещаем счётчик такта на единицу
3. Если операция из списка активных завершилась - она спланирована. Если в узле больше нет операций — завершаем алгоритм
4. Среди потомков спланированных операций в графе зависимостей ищем такие, аргументы которых уже вычислены и помещаем в список готовых
5. Переходим в п. 1

# Планирование

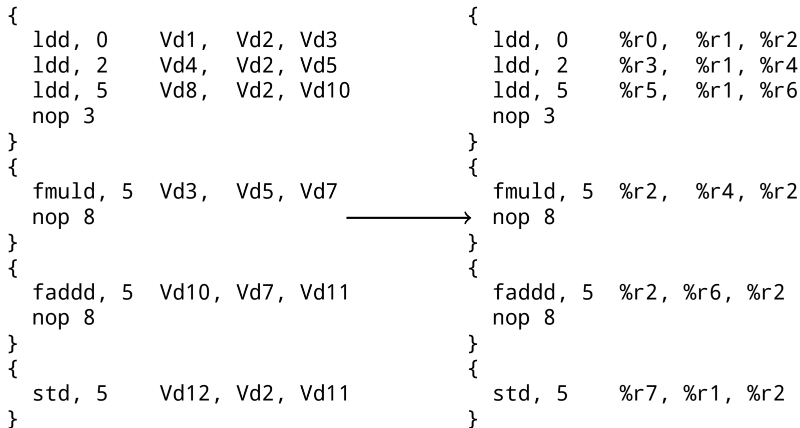
В плохих случаях код линейного участка имеет низкий IPC и длинный критический путь.

Чтобы избавиться от данной проблемы можно брать операции с критического пути и поднимать их на предшествующие линейные участки. Такая техника называется **глобальным планированием**.

# Распределение регистров

# Проблема

Для спланированного кода необходимо определить с какими физическими регистрами будут работать инструкции:



```
{                                          {
  ldd, 0    Vd1,  Vd2, Vd3                   ldd, 0    %r0,  %r1, %r2
  ldd, 2    Vd4,  Vd2, Vd5                   ldd, 2    %r3,  %r1, %r4
  ldd, 5    Vd8,  Vd2, Vd10                  ldd, 5    %r5,  %r1, %r6
  nop 3                                      nop 3
}                                          }
{                                          {
  fmuld, 5  Vd3,  Vd5, Vd7                   fmuld, 5  %r2,  %r4, %r2
  nop 8                        ------>       nop 8
}                                          }
{                                          {
  faddd, 5  Vd10, Vd7, Vd11                  faddd, 5  %r2, %r6, %r2
  nop 8                                      nop 8
}                                          }
{                                          {
  std, 5    Vd12, Vd2, Vd11                  std, 5    %r7, %r1, %r2
}                                          }
```

# Распределение регистров

**Виртуальный регистр** — абстрактная переменная, с которой работают операции внутри представления.

**Физический регистр** — реальный регистр в аппаратуре, с которым будет работать инструкция.

**Распределение регистров** (register allocation) — отображение множества виртуальных регистров на множество физических регистров микропроцессора.

# Распределение регистров

**Время жизни регистра** — время от момента определения регистра до конца его использования.

Если время жизни двух виртуальных регистров не пересекается, то их можно распределить на один физический регистр.

**Максимальная степень перекрытия** в такте — максимальное количество физических регистров, которые потребуются для распределения регистров.

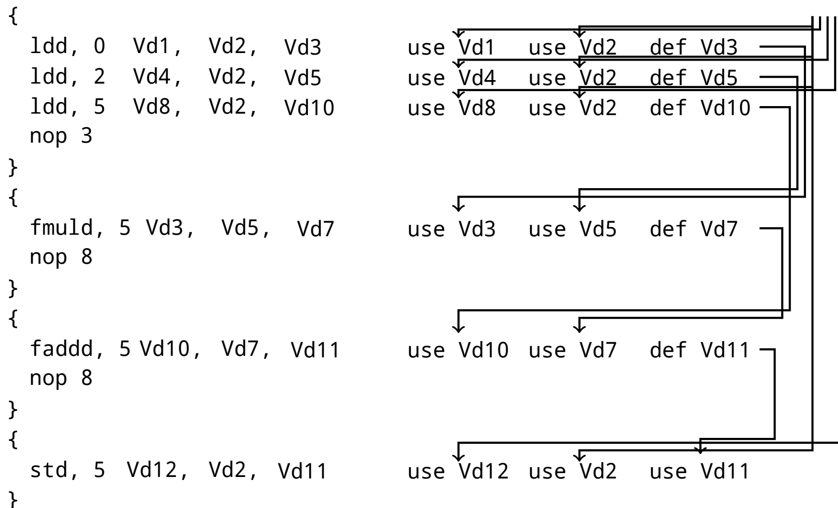
# Проблема

Как определить времена жизни регистров?

```
{
  ldd, 0   Vd1,  Vd2, Vd3
  ldd, 2   Vd4,  Vd2, Vd5
  ldd, 5   Vd8,  Vd2, Vd10
  nop 3
}
{
  fmuld, 5 Vd3,  Vd5, Vd7
  nop 8
}
{
  faddd, 5 Vd10, Vd7, Vd11
  nop 8
}
{
  std, 5   Vd12, Vd2, Vd11
}
```

# Граф определений и использований

Для определения времени жизни регистров строится **граф определений и использований**:



# Граф определений и использований

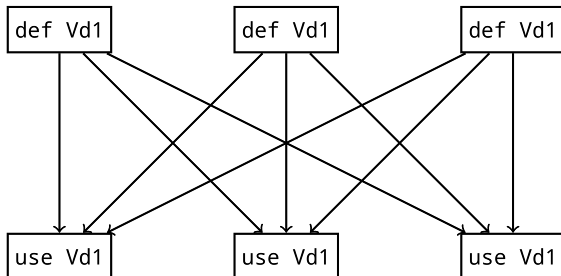
Граф определений и использований состоит из узлов двух типов: узлы **определения**, который обозначает запись значения в объект и узлы **использования**, которые обозначают чтение значения объекта.

Узел графа содержит операнды операции, а дуги обозначают отношение использования.

Граф определений и использований позволяет оценивать время жизни значения объекта и определять его происхождение.

# Проблема

Определение значения объекта может приходить из разных узлов графа потока управления:




Такая ситуация усложняет анализ графа определений и использований. Кроме того, слишком большое количество дуг увеличивает расход памяти компилятора.

# SSA-форма

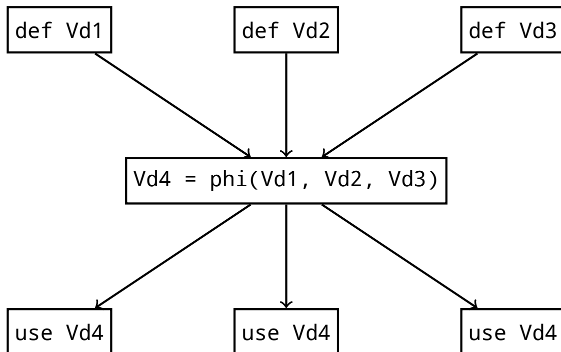
Для решения этих проблем представление преобразуется к **форме со статически однократным присваиванием** (Static single assignment, SSA) — представлению, в котором каждый объект имеет только одно определение.

Если в оригинальной программе объект имеет несколько возможных вариантов определения, для него создаются копии, определяемые единожды.

Для объединения возможных значений используются специальные $\phi$-функции.

# SSA-форма

Преобразованный к SSA-форме граф определений и использований:



# Алгоритм распределения регистров

В конце 80-х годов был выведен качественный алгоритм распределения регистров методом раскраски графа.

Строится граф $G = (N, E)$, где:

- каждый узел $N$ соответствует виртуальному регистру.
- если время жизни двух регистров пересекается, то соответствующие узлы графа соединяются дугой из $E$

Такой граф называется **графом несовместимости регистров** (RIG — Register Interference Graph).

Два графа могут быть распределены на один и тот же регистр, если отсутствует дуга, соединяющая их

# Алгоритм распределения регистров

Для распределения виртуального регистра на конкретный физический используется **раскраска графа**.

**Раскраска графа** — присвоение цветов узлам графа таким образом, что узлы связанные дугами имеют разные цвета.

Граф называется $k$**-цветным** если его возможно раскрасить $k$ цветами.

В раскрашенном графе цвета соответствуют физическим регистрам.

# Алгоритм распределения регистров

Раскраска графа является *NP*-сложной задачей. Не существует эффективного алгоритма её решения.

Бывает ситуация что заданный граф невозможно раскрасить $k$ цветами.

# Алгоритм распределения регистров

Алгоритм раскраски графа основан на эвристике и состоит из двух частей:

Извлекаем узлы из графа:

1. Выберем узел *t*, который имеет менее *k* соседей.
2. Удаляем его и его дуги из графа и кладём узел на стек
3. Повторяем 1 до тех пор пока не закончатся узлы в графе

Раскручиваем стек, присваивая цвета:

1. Берём узел с вершины стека
2. Присваиваем ему цвет, отличный от тех что были у соседей

# Сохранение-загрузка регистров

Если в линейном участке максимальная степень перекрытия превышает количество регистров на процессоре, раскрасить граф не удастся.

Для уменьшения степени перекрытия часть значений необходимо сохранить значение регистра в память, освободив его для использования другими значениями.

- Процесс сохранения регистра в память называется **spill**
- Процесс загрузки регистра из памяти называется называется **fill**

# Литература


- *А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман* Компиляторы: принципы, технологии и инструменты.
- *S. S. Muchnick* Advanced Compiler Design and Implementation.
- *K. D. Cooper, L. Torczon* Engineering a compiler.
- *R. Allen, K. Kennedy* Optimizing Compilers for Modern Architectures: A Dependence-based Approach.
- *A. Aiken* CS143 Compilers. Stanford lectures.
- *Д. Л. Хеннеси, Д. А. Паттерсон* Компьютерная архитектура. Количественный подход.
- *G. H. Blindell* Instruction Selection. Principles, Methods, and Applications