ƒ# Лабораторная работа №4. Эксперимент

Цель:

- экспериментальное знакомство с устройством процессоров через моделирование;
- получение опыта работы с компьютерной системой на нескольких уровнях организации;
- поиск компромиссов при совместном проектировании компьютерной системы на разных уровнях.

Данная лабораторная работа носит практический характер. Она включает разработку:

- языка программирования и транслятора;
- системы команд;
- модели процессора и его принципиальной схемы;
- нескольких алгоритмов (реализация и тестирование работы).

Дополнительно:

- работа с CI;
- средства автоматического контроля качества кода;
- автоматическое тестирование.

Пример отчёта и реализаций транслятора и модели процессора: Brainfuck.

Примечание 1: данная лабораторная работа не ставит перед собой задачу разработки промышленного компилятора, виртуальной машины, языка программирования или системы команд. Она подразумевает целый ряд упрощений, направленных на сокращение объёма работы. К примеру: отсутствие “внятных” сообщений об ошибках; отсутствие нормального тестирования на “некорректных” исходных и машинных кодах, отсутствие формальной проверки разработанного языка программирования, “архитектура программы по заветам ООП” и т.п.

Примечание 2: приведённый пример имеет очень много документации в коде, часть из которой продублирована в отчёте. Вам НЕ требуется иметь такой уровень документации в исходном коде как в примере.

Структура проекта

```
Язык программирования
    |                              ввод
    |                Система команд       |
    |                    |            v
    |   +------------+       |        +------------+
-----*---->| Транслятор |--------*--------->|   Модель   |----> журнал
 алгоритм  +------------+   машинный код    | процессора |      работы
                                +------------+
                                    |
                                    v
                                  вывод
```

<!-- markdown-toc start - Don't edit this section. Run M-x markdown-toc-refresh-toc -->

Table of Contents

<!-- markdown-toc end -->

Структура отчёта

В шапке необходимо указать:

- ФИО, группу;
- ваш вариант отформатированный как исходный код из ведомости.

Язык программирования

Раздел должен включать:

- Описание синтаксиса). Рекомендуется использовать форму Бэкуса-Наура.
- Описание семантики). В первую очередь:
  - стратегия вычислений,
  - области видимости,
  - типизация, виды литералов.
- Данного описания должно быть достаточно для выполнения программы “на листе бумаги”.

Организация памяти

Данный раздел является сквозным по отношению к работе и должен включать:

- модель памяти процессора, размеры машинного слова, варианты адресации;
- механику отображения программы и данных на процессор.

Модель памяти должна включать:

- Какие виды памяти и регистров доступны программисту?
- Где хранятся инструкции, процедуры и прерывания?
- Где хранятся статические и динамические данные?

```
    Registers
+-----------------------------+
| acc                         |
+-----------------------------+

    Instruction memory
+-----------------------------+
| 00  : jmp N                 |
|   ...                       |
| 10  : interruption vector 0 |
| 11  : interruption vector 1 |
|   ...                       |
```

```
| n   : program start         |
|   ...                       |
| i   : interruption handler 0 |
| i+1 : interruption handler 0 |
|   ...                       |
+-----------------------------+

        Data memory
+-----------------------------+
| 00  : constant 1            |
| 01  : constant 2            |
|   ...                       |
| l+0 : num literals          |
| l+1 : num literals          |
|   ...                       |
| c+0 : variable 1            |
|   ...                       |
+-----------------------------+
```

А также данный раздел должен включать в себя описание того, как происходит работа с 1) литералами, 2) константами, 3) переменными, 4) инструкциями, 5) процедурами, 6) прерываниями во время компиляции и исполнения. К примеру:

- В каких случаях литерал будет использован при помощи непосредственной адресации?
- В каких случаях литерал будет сохранён в статическую память?
- Как будут размещены литералы, сохранённые в статическую память, друг относительно друга?
- Как будет размещаться в память литерал, требующий для хранения несколько машинных слов?
- В каких случаях переменная будет отображена на регистр?
- Как будет разрешаться ситуация, если регистров недостаточно для отображения всех переменных?
- В каких случаях переменная будет отображена на статическую память?
- В каких случаях переменная будет отображена на стек?
- И так далее по каждому из пунктов в зависимости от варианта…

Система команд

Раздел должен включать:

- Особенности процессора (всё необходимое для понимания системы команд):
  - типы данных и машинных слов;
  - устройство памяти и регистров, адресации;
  - устройство ввода-вывода;
  - поток управления и системы прерываний;
  - и т.п.
- Набор инструкций включая количество тактов на полный цикл исполнения.
- Детальное описание способа кодирования инструкций. <!-- FIXME:TODO for bf example -->
- Описания системы команд должно быть достаточно для её классификации (CISC, RISC, Acc, Stack).

Транслятор

Раздел подразумевает разработку консольного приложения:

- *Входные данные*:
  - Имя файла с исходным кодом в текстовом виде.
  - Имя файла для сохранения полученного машинного кода.
  - Другие аргументы командной строки (ключи, настройки, и т.п.).
- *Выходные данные*:
  - Имя выходного файла для машинного кода.

Раздел должен включать описание:

- Интерфейса командной строки.
- Принципов работы разработанного транслятора (этапы, правила и т.п.).

Модель процессора

Раздел подразумевает разработку консольного приложения:

- *Входные данные*:
  - Имя файла для чтения машинного кода.
  - Имя файла с данными для имитации ввода в процессор.
- *Выходные данные*:
  - Вывод данных из процессора.
  - Журнал состояний процессора, включающий:
    - состояния регистров процессора;
    - выполняемые инструкции (возможно, микрокод) и соответствующие им исходные коды;
    - ввод/вывод из процессора.

Раздел должен включать:

1. Схемы DataPath и ControlUnit, описание сигналов, регистров и флагов.
2. Особенности реализации процесса моделирования.

Обратите внимание, что схемы должны отражать аппаратную структуру процессора и его элементов, а не устройство вашей программной модели.

Требования/рекомендации:

- В случае, если схемы DataPath и ControlUnit совмещены, должна быть убедительная аргументация в тексте отчёта.
- Не стоит полностью отрисовывать сигнальные линии от ControlUnit ко всем элементам схемы, это загромождает схему и усложняет её чтение. Обозначьте их как сделано в примере.
- Если вы настаиваете на полной отрисовке сигнальных линий, то они должны визуально отличаться от линий передачи данных / адресов.
- Схемы должны помещаться на экран.
- В случае, если схемы не соответствуют данным требованиям, они могут быть признаны нечитаемыми, следовательно, непроверяемыми. Пример нечитаемой схемы: link.
- Реализация машинной арифметики на уровне схем не требуется, просто складывайте, вычитайте, умножайте и делите так, как будто это поддержано АЛУ за один такт;
- При моделировании процессов следует ориентироваться на схему процессора и её функционирование (а не писать отвлечённый код). Это поможет вам валидировать схему/модель относительно друг-друга, а также

лучше понять принципы работы цифровой схемостехники.

Тестирование

Раздел должен включать:

1. Краткое описание разработанных тестов.
2. Ссылки на golden тесты следующих алгоритмов:
   i. `hello` — напечатать hello world;
   ii. `cat` — печатать данные, поданные через ввод (размер ввода потенциально бесконечен);
   iii. `hello_user_name` — запросить у пользователя его имя, считать его, вывести на экран приветствие (`<` — ввод пользователя через файл ввода, `>` вывод симулятора):

   ```
   > What is your name?
   < Alice
   > Hello, Alice!
   ```

   iv. `sort` — пользователь загружает в систему список чисел (формат загрузки — по аналогии с типом строки вашего варианта), и выводит их в отсортированном формате;
   v. Арифметика двойной точности: если машинное слово — 32 бита, то необходимо продемонстрировать работу с числами в 64 бита.
   vi. алгоритм согласно варианту;
   vii. дополнительные алгоритмы, демонстрирующие особенности вашего варианта (демонстрация особенностей языка, работу специфических инструкций и т.п.).
3. Требования к golden тестам:

- Должны включать: алгоритм, машинный код и данные, ввод/вывод, журнал работы процессора.
- Если размер журнала модели процессора слишком большой (сотни килобайт), его полное включение в golden test нецелесообразно. Необходимо адаптировать журнал под каждый алгоритм, добившись достаточной репрезентативности для проверки задания.
- Все исходные коды должны быть отформатированы, быть читаемыми.

4. Пример использования разработанной инструментальной цепочки.

Требования к реализации

1. Рекомендуемый язык разработки: Python (иные варианты требуют предварительного согласования).
   - Возможно использование других языков по согласованию[^dl]. Требования по CI и способу тестирования сохраняются, отсутствие библиотек — не оправдание.
2. Не стоит заниматься overengineering-ом: структура приложения по всем заветам ООП, десяток файлов кода раскиданным по 5 папкам, использование промышленного парсера/лексера. Не переусложняйте.
3. Помимо непосредственной реализации, необходимы:
   - разработка интеграционных тестов для алгоритмов в форме **golden test-ов**, которые включают:
     - входной алгоритм;
     - конфигурацию транслятора и/или модели (если необходима);
     - машинный код (с мнемониками, если ваш вариант подразумевает бинарное представление);
     - журнал моделирования;
     - вывод результатов моделирования;
     - формат хранения эталонов должен быть удобен для проверки (как в примере).
   - демонстрация реализации особенностей вашего варианта.
4. Тесты должны быть структурированными и читаемыми.
5. Исходный код должен быть отформатирован, включая код на вашем языке программирования.
6. Публикация кода на Github/Gitlab[^gl] с настроенным CI, включая lint tools и автоформаттер;
   - для Python: ruff, mypy;
   - настройки инструментария должны быть “суровыми”. Отключение проверок должно быть аргументированным.
7. Допускается использование шаблонов при настройке CI, при понимании его работы.

[^dl]: Рекомендуемые языки: Julia, Go, Rust, Haskell, Agda, Ocaml. Нерекомендуемые: Java, C#, C, C++.
[^gl]: Аккуратнее с GitLab-ом от ИТМО. У вас очень ограничены доступные образы для CI.

Варианты

Вариант определяется[^sv]:

- комбинацией особенностей языка программирования и архитектуры процессора;
- реализуемым алгоритмом;
- усложнение (опция для тех, кто хочет приключений).

[^sv]: Если у особенности только один вариант, то он является требованием ко всем реализациям.

Особенность				
ЯП. Синтаксис	alg	lisp	asm	forth
Архитектура	acc	cisc	risc	stack
Организация памяти	neum	harv		
Control Unit	hw	mc		
Точность модели	tick			
Представление маш. кода	bin			
Ввод-вывод	stream	trap		
Ввод-вывод ISA	mem	port		
Тип строк	cstr	pstr		
Алгоритм	alg1	alg2	alg3	alg4
Усложнение	pipeline	cache	vector	superscalar

Пример варианта: `alg | cisc | neum | hw | instr | binary | stream | port | pstr | prob2 | pipeline` .

<!-- Убрано вместе с вариантивностью сложности. Управление сложностью вашего варианта: 1. **Упрощённый вариант**. Необходимо выбрать опции справа от стрелок. Для приведённого примера получится: `asm | risc | neum | hw | instr | struct | stream | port | pstr | prob2 | pipeline` 2. **Базовый вариант**. Необходимо выбрать опции слева от стрелок. Для приведённого примера получится: `alg | cisc | neum | hw | instr | binary | stream | port | pstr | prob2 | pipeline`. 3. **Усложнённый вариант**. Необходимо взять базовый вариант и выполнить его с усложнением (последний пункт). -->

Как формировались варианты: lab4variants.py.

Язык программирования. Синтаксис

1. `alg` — синтаксис языка должен напоминать java/javascript/lua. Должен поддерживать математические выражения.
   - Необходимо объяснить, как происходит отображение сложных выражений на регистры и память.
   - В тестах необходимо осуществить проверку AST (абстрактного синтаксического дерева, полученного в процессе трансляции). Оно должно быть человекочитаемым.
2. `lisp` — синтаксис языка Lisp. S-exp.
   - Требуется поддержка рекурсивных функций.
   - Необходимо объяснить, как происходит отображение сложных выражений на регистры и память.
   - Любое выражение (statement) — expression, что должно быть продемонстрировано в тестах. Примеры корректного кода (с точностью до ключевых слов):
     - `(print (if (= 1 x) "T" "F"))`
     - `(setq x (if (= 1 x) 2 3))`
     - `(setq x (loop ...))`
     - `(print (setq x 13))`
3. `asm` — синтаксис ассемблера. Необходима поддержка label-ов, секций и директивы `.org`. Поддержка пользовательских макроопределений (макросы, константы, условная компиляция).
4. `forth` — синтаксис языка Forth с обратной польской нотацией.
   - Требуется поддержка процедур.
   - Требуется поддержка `execution token`.

Примечание. Язык надо реализовывать по принципам:

- минимальной реализации;
- минимального удивления;
- с оглядкой на референсные реализации (Lisp, Forth — обязательно посмотреть и попробовать существующий диалект).

К примеру (ваш вариант `lisp`):

- (хорошо) Вам нужен цикл `for` и не нужен цикл `while`. Варианты:
  - Вы реализуете цикл через конструкцию `loop` (см. Common Lisp) и поддерживаете только `for`.
  - Вы реализуете синтаксис `loop - recur` (см. Clojure).
  - Вы реализуете рекурсивные вызовы с пониманием, как это работает и оптимизируется. Блистаете оптимизацией хвостовой рекурсии.
  - Иные варианты.
- (хорошо) Традиционно языки семейства `lisp` поддерживают переменное кол-во аргументов `(+ 1 2 3)`, вы пишете в коде `(+ 1 (+ 2 3))`.
- (плохо) У вас есть код: `(setq a (if (= x 0) 1 2))`.
  - В качестве предиката в вашем коде используются только выражения, транслятор не позволяет сделать вызов для переменной: `(setq a (if p 1 2))`.

Архитектура

- `acc` — система команд должна быть выстроена вокруг аккумулятора.
  - Инструкции — изменяют значение, хранимое в аккумуляторе.
  - Ввод-вывод осуществляется через аккумулятор.
- `cisc` — система команд должна содержать сложные инструкции:
  - Арифметические операции, работающие с регистрами и памятью за одну операцию.
  - Работа со специальными регистрами.
  - Инструкции с переменным числом аргументов и, соответственно, с переменным количеством машинных слов для кодирования (к примеру, расчёт многочлена: $ `c_0 + c_1x_1 + c_2x_2 + ... + c_nx_n` $). Выборка инструкции должна явно производить смену машинного слова.
- `risc` — система команд должна быть упрощенной, в духе RISC архитектур:

    - Стандартизированная длина команд.
    - Операции над данными осуществляются только в рамках регистров.
    - Доступ к памяти и ввод-вывод — отдельные операции (с учётом специфики вашего варианта `mem/port`).
- `stack` — система команд должна быть стековой:
    - Вместо регистров используется стек.
    - Это не исключает и не заменяет наличие памяти команд и памяти данных.

Архитектура организации памяти

Тип памяти — однопортовая.

- `neum` — фон Неймановская архитектура.
- `harv` — Гарвардская архитектура:
    - В тестах необходимо привести/проверить как память команд, так и память данных.

Control Unit

- `hw` — hardwired. Реализуется как часть модели.
- `mc` — microcoded.
    - В отчёте необходимо задокументировать уровень микроинструкций.
    - Моделирование должно выполняться с точностью до такта.
    - Микрокод должен быть сохранён в отдельной памяти для микропрограмм.
    - Модель процессора должна исполнять микрокод.

Точность модели

- `tick` — процессор необходимо моделировать с точностью до такта, процесс моделирования может быть приостановлен на любом такте.

<!-- - `instr` -- процессор необходимо моделировать с точностью до каждой инструкции (наблюдается состояние после каждой инструкции). -->

Представление машинного кода

- `binary` — бинарное представление.
  - Требуются настоящие бинарные файлы, а не текстовые файлы с `0` и `1`.
  - Требуется отладочный вывод в текстовый файл вида:

    1. **<address>** - **<HEXCODE>** - **<mnemonic>**
    2. 20 - 03340301 - add #01 <- 34 + #03

<!-- - `struct` -- в виде высокоуровневой структуры данных. Считается, что одна инструкция укладывается в одно машинное слово, за исключением CISC архитектур. -->

Ввод-вывод

`stream`

Ввод-вывод осуществляется как поток токенов (как во Wrench). Есть в примере. Логика работы:

- При старте модели у вас есть буфер, в котором представлены все данные ввода (`['h', 'e', 'l', 'l', 'o']`).
- При обращении к вводу (выполнение инструкции) модель процессора получает “токен” (символ) информации.
- Если данные в буфере кончились — останавливайте моделирование.
- Вывод данных реализуется аналогично, по выполнении команд в буфер вывода добавляется ещё один символ.
- По окончании моделирования показать все выведенные данные.
- Логика работы с буфером реализуется в рамках модели на Python.

`trap`

Ввод-вывод осуществляется токенами через систему прерываний. Логика работы:

- При старте модели у вас есть расписание ввода (`[(1, 'h'), (10, 'e'), (20, 'l'), (25, 'l'), (100, 'o')]` , где число — такт когда будет вызвано прерывание, символ — значение доступное в порту начиная с указанного такта).
- Процессор имеет систему прерываний:
  - прерывания считаются внутренними;
  - обработчик прерывания реализуется программистом на вашем языке.
- Из журнала работы процессора должно быть ясно, работаете вы в прерывании или нет.
- Вывод данных реализуется посимвольно, как в варианте `stream` , по выполнении команд в буфер вывода добавляется ещё один символ.
- По окончании моделирования показать все выведенные данные.
- Ситуация наступления прерывания во время обработки прерывания должна быть проработана (способ — на ваше усмотрение) в стиле “реализм”.
- Логика работы с буферами реализуется в рамках модели на Python.
- Не стоит путать “вызов прерывания” и “получение данных”.
- Нет магическим очередям.

Ввод-вывод ISA

Поддержка ввода-вывода с точки зрения системы команд.

- `mem` — memory-mapped (порты ввода-вывода отображаются в память и доступ к ним осуществляется штатными командами),
  - отображение портов ввода-вывода в память должно конфигурироваться (можно hardcode-ом).
- `port` — port-mapped (специальные инструкции для ввода-вывода)
  - адресация портов ввода-вывода должна присутствовать.

Поддержка строк

Варианты:

- `cstr` — Null-terminated (C string)
- `pstr` — Length-prefixed (Pascal string)

Общие требования:

- Статические строки должны храниться в памяти (секции) данных.
- Один символ может храниться в одном машинном слове (несмотря на явную неэффективность).
- Работа со строками реализуется процедурами или функциями на разработанном вами языке.

Алгоритм

- Входные данные должны подаваться через ввод.
- Результат должен быть подан на вывод.
- Формат ввода/вывода данных — на ваше усмотрение.

Алгоритмы:

- `alg1` — Euler problem 4 link
- `alg2` — Euler problem 6 link
- `alg3` — Euler problem 9 link
- `alg4` — Euler problem 16 link

Усложнение

- `pipeline` — конвейерная организация работы процессора.
  - Количество стадий конвейера — не менее 3.
  - Необходимо показать влияние конвейера на производительность написанных программ.
  - В схеме необходимо явно отразить стадии работы конвейера. Пример — см. в слайдах лекций.
- `cache` — работа с памятью реализуется через кеш.
  - Скорость доступа к кешу — 1 такт, к памяти — 10 тактов.
  - Необходимо реализовать алгоритм, демонстрирующий работу с кеш памятью и её влияние на производительность.
  - В журнале процессора должно быть видно, как работает кеш и как процессор ожидает получение данных.
- `vector` — векторная организация работы процессора.
  - Необходимо реализовать векторные регистры и инструкции для работы с ними.
  - Минимальный набор операций: сложение, вычитание, умножение, деление, сравнение.

    - Необходимо продемонстрировать эффективность векторизации.
    - В отчете необходимо привести сравнение производительности векторной и скалярной реализаций алгоритма.
- `superscalar` — суперскалярная организация работы процессора.
    - Необходимо реализовать возможность параллельного исполнения нескольких инструкций (не менее 2-х).
    - Должны быть реализованы механизмы обнаружения и разрешения зависимостей по данным.
    - Необходимо показать влияние суперскалярности на производительность написанных программ.
    - В журнале процессора должно быть видно, какие инструкции выполняются параллельно.

Оценка

- 0-20 баллов если 1-2 пункта характеризуют вашу работу:
    - У вас не проходит CI, нельзя посмотреть отчёт о работе CI прямо в репозитории.
    - Нет golden test-ов.
    - Нет существенных разделов отчёта.
    - Есть существенные ошибки в реализации вашего варианта.
    - Работа выполнена с существенными ошибками и недостатками.
    - Учащийся не может объяснить ключевые принципы работы схем, модели, транслятора.
    - Работа частично соответствует варианту.
- 30 баллов:
    - Работа выполнена в соответствии с вариантом.
    - Работа выполнена с незначительными ошибками и недостатками, которые могут быть объяснены учащимся.
    - Если приняты “странные решения”, то они могут быть аргументированы.
    - Учащийся может объяснить принципы работы схем, модели, транслятора.
    - Лабораторная работа не переусложнена вами.
- 40 баллов:
    - Учащийся достойно справился с задачей, включая усложнение.

Типичные проблемы, которые не позволили получить высокий балл:

1. Интеграционные тесты реализованы альтернативным способом (не golden tests, что затрудняет проверку).
2. Данные программы являются частью кода инструкций, а не данных (к примеру: строка `hello` хранится как 5 инструкций записи константы в регистр).
3. Небрежные и не читаемые схемы, дублирующиеся входы/выходы из схем (к примеру 3 входа `data_addr` у памяти).
4. Игнорирование рекомендаций lint tools без убедительной аргументации.
5. Вариант реализован не в полной мере или с ошибкой, к примеру:
   - “Это CISC процессор, но нет инструкций переменной длины”.
   - “Это микрокод, но микрокодирование реализовано через словари микроинструкций без аппаратно-реализуемой логики”.
   - “Это система прерываний, но нет обработчиков прерываний”.
6. Наличие в репозитории лишних файлов (файлы ОС, файлы текстового редактора и т.п.), мёртвого кода.
7. Непонимание назначения мультиплексоров, процессов передачи данных между элементами схемы.
8. Непонимание, как устроены элементы структурных схем (кроме арифметики).
9. Код программ не отформатирован должным образом (включая код на вашем языке).
10. Схема построена на вентилях, а не на мультиплексорах и защёлках.
11. Ответ на вопрос: так захотелось и я уже не помню. Нужна аргументация.

Общие рекомендации по началу работы

К сожалению, нет универсального ответа на вопрос о том, как лучше делать эту лабораторную работу. Всё сильно зависит от того, какие у вас навыки и знания, какой вариант.

Я бы предложил вам делать её так:

1. Написать 1-2 синтетические программы на вашем (пока ещё воображаемом) ЯП, в которых будет:
   - ветвление,
   - цикл,
   - математика,
   - строки,
   - ввод/вывод,
   - что-то ещё, необходимое по вашему варианту.

2. Написать для них машинный код.
3. Нарисовать схемы процессора в первом приближении, чтобы убедиться, что ваш машинный код работоспособен.
4. Написать транслятор.
5. Написать симулятор процессора.
6. Реализовать ваши алгоритмы.
