ИПР 3. Исследование производительности ЦПУ.

В данной работе будет исследоваться быстродействие процессора, реализующего RISC-V архитектуру. В предыдущей работе необходимо было закончить функциональный симулятор, основная задача которого — симуляция исполнения кода сторонней архитектуры на рабочей машине. В этой работе симулятор будет переписан в потактовый, основная задача которого — исследование скорости исполнения кода сторонней архитектуры. Основная метрика производительности исполнения кода — количество выполненных инструкций в такт. Потактовый симулятор может моделировать все задержки, возникающие при выполнении инструкций в разрабатываемой микроархитектуре. Однако, чем точнее модель, тем больше ресурсов требуется на ее разработку и тем дольше длится время симуляции. Поэтому обычно моделируют только те части, которые больше всего влияют на производительность. В данном случае, потактовая модель будет очень простой и близкой к модели функционального симулятора. Исключение составляет подсистема памяти. Теперь подсистема памяти будет разбита на функциональную и потактовую модель. Функциональная модель содержит значения памяти и имеет простой интерфейс. Потактовая модель моделирует временные характеристики работы с памятью и использует функциональную модель для хранения данных.

Структура проекта

В качестве задания вам предлагается проект симулятора, который необходимо закончить. Структура следующая:

- programms — директория с тестами.
  - assembly — директория с исходниками коротких ассемблерных тестов.
  - build/assembly — директория с собранными объектными файлами.
    - bin — директория с выполняемыми elf-файлами.
    - dump — директория с дампами elf-файлов.

- src — директория с исходными файлами симулятора.
  - main.cpp — точка входа в программу.
  - BaseTypes.h — основные типы программы.
  - Instruction.{h, cpp} — описание декодированной инструкции.
  - Memory.h — модуль подсистемы памяти. Содержит функциональную и потактовую модель памяти.
  - Cpu.h — модуль ЦПУ. Функция ProcessInstruction() переименована в Clock(), которая вызывает CsrFile::Clock().
  - Decoder.h — модуль декодирования инструкции.
  - RegisterFile.h — модуль регистров общего назначения.
  - CsrFile.h — модуль служебных регистров. Интерфейс переписан, появилась функция Clock, которую надо вызвать в самом начале Cpu::Clock()
  - Executor.h — модуль выполнения инструкции.
- CMakeLists.txt — cmake-файл для сборки проекта.
- test.sh — скрипт для запуска тестов.
- units — директория для юнит-тестов

Собрать проект и запустить тесты можно из bash-подобного терминала следующими командами:

```
〉cd /path/to/project/directory
〉mkdir build
〉cd build
〉cmake ..
〉make -j8
〉cd ..
〉./test.sh build/risv_sim
```

Интерфейс IMem

Функция Cpu::ProcessInstruction() должна быть переименована в Cpu::Clock() и переписана так, чтобы поддержать интерфейс подсистемы памяти IMem. В интерфейсе IMem есть две пары функций Request и Response — для чтения из памяти новой команды и для чтения/записи данных в процессе выполнения команд. Новый запрос в подсистему памяти начинается вызовом функции Request. Далее должен следовать вызов функции Response, который возвращает ответ. В общем случае, ответ может быть возвращен в том же или в любом другом последующем такте. std::optional<Word> Response() вернет пустой std::optional, если ответ не готов, и валидное значение, в случае завершения чтения. bool Response(InstructionPtr &instr) вернет true в случае успешного завершения обращения в память командой, и false, если обращение в память не завершено. Если возвращаемое любой из функций Response значение свидетельствует об успешном завершении обращения в память, то цикл тракта данных продолжает исполнение команды в этом такте. Иначе, Cpu прекращает исполнение цикла тракта данных в этом такте и выходит из функции Clock(). При этом, надо сохранить состояние тракта данных таким образом, чтобы при следующем вызове Clock() продолжить его выполнении с последнего незавершенного Response(). Изменение статуса запроса выполняется внутри функции IMem::Clock(), которая вызывается из main. Иными словами, если запрос в подсистему памяти длится несколько тактов, то после вызова Request(), несколько последующих вызовов Cpu::Clock() будут вызывать только функцию Response(), получать ответ о том, что операция не завершилась и сразу выходить из функции Cpu::Clock().

Задание: написать Cpu::Clock(), чтобы она исполняла цикл тракта данных и была совместима с интерфейсом класса IMem. Проверьте работоспособность на наборах тестов «asm tests», «small benchmarks» и «big benchmarks». Вычислите количество инструкций в такт и запишите в таблицу.

Кэширование

После выполнения предыдущего задания, вы могли обратить внимание, что количество тактов на исполнение тестов значительно выросло. Очевидно, причина этому большое время обращения в память. Для решения данной проблемы используется кэш. В этом задании необходимо реализовать кэширующую модель подсистемы памяти с тем же интерфейсом что и UncachedMem. В этой модели должны быть отдельные блоки кэша для данных и для кода. Ниже дано краткое описание работы полностью-ассоциативного кэша с LRU-замещением.

Наличие каждого слова должно проверяться в кэше в самом начале обработки запросов в память. В случае отсутствия (промаха), делается запрос в основную память, то есть обработка почти ничем не отличается от обрбаотки запроса в UncachedMem. Но, из памяти кэш получает не одно слово, а целую линия определенного размера — несколько слов, идущих подряд. Эта линия данных сохраняется в кэше и тегируется адресом линии. В дальнейшем, поиск линии при следующих запросах происходит по этому тегу. В случае наличия запрашиваемого слова в кэше (попадание), кэш кода возвращает слово сразу, кэш данных вернет слово через 3 такта. Так как размер кэша меньше размера общей памяти, то обязательно возникнет ситуация, когда для новых линий нет места. В случае отсутствия свободного места в кэше, надо выбрать линию-жертву на перезапись. Для этого можно использовать линию с самым давним использованием (least recently used, LRU). Линия с кодом перезаписывается сразу. Старая линия данных должна быть сначала записана в память, если она менялась.

Константа lineSizeBytes в Memory.h задает размер линии в байтах. Там же определена функция ToLineAddr, которая используется для вычисления адреса линии по запрашиваемому адресу в памяти. Эта функция должна быть использована для вычисления тега, по которому надо найти ячейку с запрашиваемыми данными в кэше, так же эта функция используется для вычисления адреса линии, которая должна быть запрошена в памяти в случае промаха. Смещение слова внутри линии можно получить функцией ToLineOffset. Кроме этого, каждый кэш должен обладать следующими параметрами времени компиляции: размер кэш-линии, количество кэш-линий, время обработки запроса в случае промаха и время обработки запроса в случае попадания.

Задание: написать кэширующую подсистему памяти с одтельными кэшами для данных и кода. Размеры кэш линий, размер каждого кэша в байтах и время доступа в память при промахе приведены в таблице по вариантам.

Варианты задания

Вариант	Линия, байт	Кэщ кода, байт	Кэше данных, байт	Время доступа
1	128	512	1024	152
2	64	512	1024	136
3	32	512	1024	128
4	128	1024	1024	152
5	64	1024	1024	136
6	32	1024	1024	128
7	128	2048	1024	152
8	64	2048	1024	136
9	32	2048	1024	128
10	128	512	2048	152
11	64	512	2048	136
12	32	512	2048	128
13	128	1024	2048	152
14	64	1024	2048	136
15	32	1024	2048	128
16	128	2048	2048	152
17	64	2048	2048	136
18	32	2048	2048	128
19	128	512	4096	152
20	64	512	4096	136
21	32	512	4096	128
22	128	1024	4096	152
23	64	1024	4096	136
24	32	1024	4096	128
25	128	2048	4096	152
26	64	2048	4096	136
27	32	2048	4096	128
28	128	1024	8096	152
29	64	1024	8096	136
30	32	1024	8096	128

Дополнительные источники

1. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf; перевод: https://habr.com/en/post/234047/.
2. https://riscv.org/specifications/isa-spec-pdf/, https://github.com/riscv/riscv-isa-manual.
3. https://ru.wikipedia.org/wiki/Кэш_процессора