ИПР 2. Разработка функционального симулятора.

В данной работе вам предстоит разработать функциональный симулятор архитектуры RISC-V. Архитектура создана в 2010 в Калифорнийском университете Беркли, распространяется по открытой лицензии и любой может использовать ее для реализации своих вычислительных устройств. Архитектура разрабатывалась по принципу RISC, является модульной. Мотивация разрабоки архитектуры приведена ее авторами в статье [1]. Актуальная спецификация находится в открытом доступе [2]. В данной работе предстоит реализовать выполнение набора целочисленных команд.

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.
  - Cpu.h — модуль ЦПУ.
  - Decoder.h — модуль декодирования инструкции.

    - RegisterFile.h — модуль регистров общего назначения.
    - CsrFile.h — модуль служебных регистров.
    - Executor.h — модуль выполнения инструкции.
- CMakeLists.txt — cmake-файл для сборки проекта.
- test.sh — скрипт для запуска тестов.
- units — директория для юнит-тестов

В main.cpp вызывается Cpu::ProcessInstruction(). Эта функция выполняет один цикл тракта данных виртуальной машины, исполняющей RISC-V код. В рамках этого цикла происходит: получение слова инструкции типа Word из модуля памяти Memory по указателю инструкции _ip, декодирование инструкции в структуру типа Instruction, чтение требуемых инструкцией регистров из регистровых файлов RegisterFile и CsrFile, исполнение инструкции в модуле Executor, обращение в память, запись результата в регистровые файлы RegisterFile и CsrFile. Завершается цикл обновлением регистров в CsrFile и вычислением нового _ip для следующего цикла тракта данных.

Задание.

- Написать один юнит-тест для функции Decoder::Decode() для инструкции, приведенной в таблице 1, по вариантам. Юнит-тест должен выполнять три шага.
    - Составить 32-битное слово, соответствующее закодированной инструкции. Коды инструкций приведены в спецификации [2], с. 130.
    - Декодировать инструкцию функцией Decoder::Decode.
    - Проверить корректность заполнения структуры Instruction.
- Реализовать функцию Executor::Execute(). Функция должна менять состояние структуры Instruction в соответствии со спецификацией, предложенной в разделе «Спецификация Executor». Реализовать необходимо исполнение всех инструкций, поддержаных декодером.

- Написать один юнит-тест для функции Executor::Execute(). Для теста взять ту же инструкцию, что и в тесте Decoder::Decode().
- Дописать функцию Cpu::ProcessInstruction(), чтобы она реализовывала корректный цикл тракта данных.
- Необязятельное задание на 10. Предложенная программная архитектура хоть и является простой, недостаточно красива, т.к. инструкции разных типов распаковываются в общую структуру Instruction и использует слишком много switch-case. Предложите свою архитектуру программы и отрефакторите код в соответствие с ней.

Реализовав Executor::Execute() и Cpu::ProcessInstruction(), у вас должен получиться готовый функциональный симулятор, который может исполнять небольшие программы. Для тестирования работы вам надо запустить bash-скрипт test.sh и через командную строку передать путь к исполняемому файлу симулятора. Пример: «test.sh cmake-build-debug/riscv-sim».

Замечание. Вполне возможно сделать рабочую версию симулятора меньше чем за 1000 строк кода. Из них, примерно 600 вам уже предоставлено.

Спецификация Executor.

Для начала, рассмотрим структуру Instruction, которая представляет из себя декодированную инструкцию. Поля структуры и их назначение представлены ниже:

- IType _type; // тип инструкции.
- BrFunc _brFunc; // тип ветвления.
- AluFunc _aluFunc; // тип перехода.
- std::optional<RId> _dst; // индекс выходного регистра.
- std::optional<RId> _src1; // индекс первого входного регистра.
- std::optional<RId> _src2; // индекс второго входного регистра.
- std::optional<CsrIdx> _csr; // индекс системного регистра.
- std::optional<Word> _imm; // константа, закодированная в инструкции.
- Word _src1Val; // значение первого входного регистра.
- Word _src2Val; // значение второго входного регистра.
- Word _csrVal; // значение системного регистра.
- Word _data = 0xdeadbeaf; // данные для записи в регистровый файл.

- Word _addr = 0xdeadbeaf; // адрес для обращения в память.
- Word _nextIp = 0xdeadbeaf; // адрес следующей инструкции.

Поля от _type до _imm инициализируются на этапе декодирования. Тип optional у полей означает, что поле остается не проинициализированным, если оно не используется данной инструкцией. Поля _src1Val, src2Val и _csrVal инициализируются значениями соответствующих регистров. Если инструкция использует хотя бы одно из этих полей, то на этапе исполнения в функции Execute они уже известны. В функции Execute происходит вычисление полей _data, _addr и _nextIp.

Executor состоит из блока АЛУ, блока ветвления и блока логики, определяющей, какое значение должно быть записано в поле _data.

АЛУ выполняет арифметическо-логические операции над двумя операндами. Значение первого операнда берется из _srcVal1 в случае, если инструкция определяет валидный _src1. Значение второго операнда определяется либо значением _imm, если оно определено инструкцией, либо _srcVal2. Возможно, что один или оба операндов не определены в инструкции, это значит, что вычисления АЛУ или не происходят, или игнорируются. Результат вычисления ALU всегда записывается в поле _addr для инструкций типа Itype::Ld или Itype::St. Обозначим первый операнд А, а второй Б. Операции реализуемые АЛУ:

- AluFunc::Add — А + Б.
- AluFunc::Sub — А - Б.
- AluFunc::And — побитовое А и Б.
- AluFunc::Or — побитовое А или Б.
- AluFunc::Xor — побитовое А xor Б.
- AluFunc::Slt — А < Б, где А и Б считаются числами со знаком.
- AluFunc::Sltu — А < Б, где А и Б считаются беззнаковыми.
- AluFunc::Sll — сдвиг А на В влево, где В = Б % 32.
- AluFunc::Srl — беззнаковый сдвиг А на В вправо, где В = Б % 32.
- AluFunc::Sra — знаковый сдвиг А на В вправо, где В = Б % 32.

Поле _data инициализируется результатом вычислений ALU, за исключением инструкций следующих типов:

- IType::Csrr — записать _csrVal.
- IType::Csrw — записать _src1Val.
- IType::St — записать _src2Val.
- IType::J и IType::Jr — записать адрес текущей инструкции увеличенный на 4.

- IType::Auipc — записать адрес текущей инструкции увеличенный на _imm.

Последний блок — блок вычисления переходов. Он вычисляет условие перехода и адрес следующей инструкции, на которую надо перейти. Условие перехода вычисляется по двум операндам _srcVal1 и _srcVal2 и типу функции перехода _brFunc. Оба операнда должны быть валидны. Существуют следующие функции:

- BrFunc::Eq — равенство.
- BrFunc::Neq — неравенство.
- BrFunc::Lt — знаковое сравнение операндов на меньше.
- BrFunc::Ltu — беззнаковое сравнение операндов на меньше.
- BrFunc::Ge — знаковое сравнение операндов на больше или равно.
- BrFunc::Geu — беззнаковое сравнение операндов на больше или равно.
- BrFunc::AT — всегда истинно.
- BrFunc::NT — всегда ложно.

Вычисление адреса зависит от типа инструкции. Для IType::Br и IType::J — следующий адрес вычисляется как текущий плюс смещение, записанное в _imm. Для IType::Jr — смещение из _imm добавляется к значению _srcVal1. Если условие перехода истинно, то полученный адрес записывается в _nextIp, иначе в _nextIp записывается адрес текущей инструкции, увеличенный на 4.

Таблица 1. Варианты заданий

	Инструкция	**Название**	**Псевдокод**
1	LUI rd,imm	Load Upper Immediate	rd ← imm
2	AUIPC rd,offset	Add Upper Immediate to PC	rd ← pc + offset
3	JAL rd,offset	Jump and Link	rd ← pc + length(inst) pc ← pc + offset
4	BEQ rs1,rs2,offset	Branch Equal	if rs1 = rs2 then pc ← pc + offset
5	BNE rs1,rs2,offset	Branch Not Equal	if rs1 ≠ rs2 then pc ← pc + offset
6	BLT rs1,rs2,offset	Branch Less Than	if rs1 < rs2 then pc ← pc + offset
7	BGE rs1,rs2,offset	Branch Greater than Equal	if rs1 ≥ rs2 then pc ← pc + offset
8	BLTU rs1,rs2,offset	Branch Less Than Unsigned	if rs1 < rs2 then pc ← pc + offset
9	BGEU rs1,rs2,offset	Branch Greater than Equal Unsigned	if rs1 ≥ rs2 then pc ← pc + offset
10	LW rd,offset(rs1)	Load Word	rd ← s32[rs1 + offset]
11	SW rs2,offset(rs1)	Store Word	u32[rs1 + offset] ← rs2
12	ADDI rd,rs1,imm	Add Immediate	rd ← rs1 + sx(imm)
13	SLTI rd,rs1,imm	Set Less Than Immediate	rd ← sx(rs1) < sx(imm)
14	SLTIU rd,rs1,imm	Set Less Than Immediate Unsigned	rd ← ux(rs1) < ux(imm)
15	XORI rd,rs1,imm	Xor Immediate	rd ← ux(rs1) ⊕ ux(imm)

16	ORI rd,rs1,imm	Or Immediate	rd ← ux(rs1) ∨ ux(imm)
17	ANDI rd,rs1,imm	And Immediate	rd ← ux(rs1) ∧ ux(imm)
18	SLLI rd,rs1,imm	Shift Left Logical Immediate	rd ← ux(rs1) « ux(imm)
19	SRLI rd,rs1,imm	Shift Right Logical Immediate	rd ← ux(rs1) » ux(imm)
20	SRAI rd,rs1,imm	Shift Right Arithmetic Immediate	rd ← sx(rs1) » ux(imm)
21	ADD rd,rs1,rs2	Add	rd ← sx(rs1) + sx(rs2)
22	SUB rd,rs1,rs2	Subtract	rd ← sx(rs1) - sx(rs2)
23	SLL rd,rs1,rs2	Shift Left Logical	rd ← ux(rs1) « rs2
24	SLT rd,rs1,rs2	Set Less Than	rd ← sx(rs1) < sx(rs2)
25	SLTU rd,rs1,rs2	Set Less Than Unsigned	rd ← ux(rs1) < ux(rs2)
26	XOR rd,rs1,rs2	Xor	rd ← ux(rs1) ⊕ ux(rs2)
27	SRL rd,rs1,rs2	Shift Right Logical	rd ← ux(rs1) » rs2
28	SRA rd,rs1,rs2	Shift Right Arithmetic	rd ← sx(rs1) » rs2
29	OR rd,rs1,rs2	Or	rd ← ux(rs1) ∨ ux(rs2)
30	AND rd,rs1,rs2	And	rd ← ux(rs1) ∧ ux(rs2)

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.