Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Архитектура вычислительных систем

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе
на тему:

“Разработка простейшего 256-битного SIMD-АЛУ”

Выполнил: студент группы 953505
Басенко Кирилл Александрович

Проверил:
Леченко Антон Владимирович

Минск 2021

Введение

Время неумолимо, объем и сложность данных, обрабатываемых современными персональными компьютерами, растут в геометрической прогрессии, предъявляя невероятные требования к микропроцессорам. Между тем, производительность вычислений, которая может быть достигнута за счет увеличения тактовой частоты микропроцессора, приближается к физическим пределам, что делает архитектурные решения более заметными. В связи с этим умные люди создали важную архитектурную - SIMD (одна инструкция для нескольких данных), которая представляет собой набор инструкций, которые могут повысить производительность приложения, позволяя выполнять базовые операции над несколькими элементами данных параллельно с меньшим количеством инструкций.

Single instruction, multiple data

Устройство SIMD обеспечивает неконкурентную параллельность на уровне данных. Такое устройство будет выполнять одну команду для разных данных. Инструкции такого устройства будут обрабатывать входные данные за одну операцию.

К примеру: нам нужно соответственно сложить n чисел с другими n числами. Первое, что приходит в голову -- это сложить соответственные числа:

A0	+	B0	=	res0
A1	+	B1	=	res1
...	...	...	...	...
An	+	Bn	=	resn

Инструкция SIMD это выполнит так:

A0 A1 ... An	+	B0 B1 ... Bn	=	res0 res1 ... resn

Повторение -- мать учения

““Симулякр — это вовсе не то, что скрывает собой истину, — это истина, скрывающая, что её нет.
Симулякр есть истина.”
-Экклезиаст”
-Жан Бодрийяр

В качестве примеров:

Basic-SIMD-Processor-Verilog-Tutorial от zslwyuan

Данный пример представляет собой реализацию SIMD CPU.

Представленные типы данных: упакованные 4-битные числа, упакованные байты, упакованные слова.

На вход в ALU идут два 16-битных оператора и флаги типа данных: Q -- упакованные 4-битные числа, O -- упакованные байты, H -- упакованные слова. На выходе слово.

Реализованные операции: сложение/вычитание, умножение, правый/левый сдвиг.

Intel's MMX Technology

Данный пример представляет собой расширение архитектуры Intel.

Представленные типы данных: упакованные байты, упакованные слова, упакованные двойные слова, четверное слово.

На вход в инструкции идут два четверных слова, на выходе четверное слово.

Реализованные инструкции: сложение/вычитание, умножение/деление, правый/левый сдвиг, сравнение на больше/меньшее/равенство, etc.

Отличия от Basic-SIMD-Processor-Verilog-Tutorial

“Мой маршрут — это путь из точки ‘А’ в точку ‘А’.
Завершается петля и в середине нее я”
-Booker

Данная работа отличается от вышеназванного проекта:

- Языком разработки -- Python/Verilog
- Операции АЛУ выполняются за один контрольный сигнал
- Результатом (и частично целью) данного проекта служит АЛУ, а не процессор
- Другие размеры операндов -- 256/16
- Наличием/отсутствием некоторых команд
- Целью разработки

newMigen

Так как курсовая выполнена на языке Python с использованием стороннего модуля nmigen, то:

nMigen основан на Migen, языке описания аппаратного обеспечения. Предоставляет обширный уровень совместимости, который позволяет создавать и моделировать большинство проектов Migen без изменений, а также интегрировать модули, написанные для Migen и nMigen.

Несмотря на то, что проектирование оборудования с помощью Verilog и VHDL быстрее, чем ввод схемы, проектирование остается утомительным и неэффективным по нескольким причинам. Модель, управляемая событиями, вводит проблемы и ручное кодирование, которые не нужны для синхронных схем, которые составляют львиную долю современных логических схем. Так же поддержка процедурной генерации логики с помощью операторов «генерации» очень ограничена и ограничивает способы, которыми код может быть обобщен, повторно использован и организован.

Migen имеет недостатки, такие как:

- Migen сильно адаптирован к Verilog, но перевод Migen в Verilog не является простым, оставляет много неявной семантики (например, подписи, расширение ширины, комбинаторные назначения, назначения вспомогательных сигналов)
- Иерархические конструкции полезны для планирования и оптимизации этажей, но Migen их не поддерживает
- Синтаксис Migen нелегко составить, и что-то вроде конечного автомата требует расширения синтаксиса Migen неортогональными способами
- Migen требует неудобных специальных возможностей для некоторых функций FPGA, таких как асинхронный сброс

newMigen исправляет этим недостатки и добавляет удобность в разработке.

Простейшее 256-битное SIMD-АЛУ

Представленные типы данных: упакованные байты, упакованные слова, упакованные двойные слова, упакованные четверные слова.

На вход в АЛУ идут сигналы: два 256-битных операнда, команда, тип данных. На выходе 256-битный результат. Выполняется за одни контрольный сигнал.

```
class ALU(Elaboratable):

    def __init__(self):

        super().__init__()

        self.op1      : Signal = Signal(256       , reset=0)

        self.op2      : Signal = Signal(256       , reset=0)

        self.data_type: Signal = Signal(DATA_TYPES, reset=0)

        self.func     : Signal = Signal(ALU_FUNCS , reset=0)

        self.res      : Signal = Signal(256       , reset=0)
```

В зависимости от операции выполняется соответствующая логика.

```
    def elaborate(self, platform) -> Module:

        m = Module()

        if platform is None:

            m.d.sync += Signal().eq(1)

        with m.Switch(self.func):

            with m.Case(ALU_FUNCS.ADD, ALU_FUNCS.SUB):

                m.d.comb += list(self.addsub_logic_gen())

            with m.Case(ALU_FUNCS.EQ):

                m.d.comb += list(self.equal_logic_gen())

            with m.Case(ALU_FUNCS.MORE, ALU_FUNCS.LESS):

                m.d.comb += list(self.moreless_logic_gen())

            with m.Case(ALU_FUNCS.SHR, ALU_FUNCS.SHL):

                m.d.comb += list(self.sh_logic_gen())

        return m

    def moreless_logic_gen(self) -> Assign:...
```

```
    def sh_logic_gen(self) -> Assign:...
    def equal_logic_gen(self) -> Assign:...
    def addsub_logic_gen(self) -> Assign:...
```

Определены константы для типов данных, команд:

```
from enum import Enum
class ALU_FUNCS(Enum):
    NONE = 0
    ADD  = 1
    SUB  = 2
    MUL  = 3
    DIV  = 4
    EQ   = 5
    MORE = 6
    LESS = 7
    SHL  = 8
    SHR  = 9
class DATA_TYPES(Enum):
    pckd_b  = 0
    pckd_w  = 1
    pckd_dw = 2
    pckd_qw = 3
```

Так же созданы тесты для команд АЛУ:

```
s: int = 0
f: int = 0
def alu_ut( alu: ALU,
            func: ALU_FUNCS,
            op1: str,
            op2: str,
            data_type: DATA_TYPES,
            expected: str) -> Assign:
```

```
    global s, f
    yield alu.op1      .eq(int(op1, 16))
    yield alu.op2      .eq(int(op2, 16))
    yield alu.data_type.eq(data_type)
    yield alu.func     .eq(func)
    yield Settle()
    res = to_formatted_hex((yield alu.res))
    if res == expected:
        s += 1
    else:
        print(f'WRONG:\n{op1 = }\n{op2 = }\n{func = }\n{res = }\n{data_type =
}\n{expected = }\n\n')
        f += 1
def alu_sh_test(alu: ALU) -> Assign: ...
def alu_moreless_test(alu: ALU) -> Assign: ...
def alu_equal_test(alu: ALU) -> Assign:...
def alu_addsub_test(alu: ALU) -> Assign:...
def alu_test(alu: ALU) -> Assign:
    global s, f
    yield from alu_moreless_test(alu)
    yield from alu_addsub_test(alu)
    yield from alu_equal_test(alu)
    yield from alu_sh_test(alu)
    print(f'{s = }\n{f = }')
def main():
    alu = ALU()
    sim = Simulator(alu)
    with sim.write_vcd(open('out.vcd', 'w')):
        sim.add_clock(1e-6)
        sim.add_sync_process(lambda: (yield from alu_test(alu)))
        sim.run()
if __name__ == '__main__': main()
```

Заключение

“*_,_,i = i[:] = 'я смотрю кино ', 'про себя, в котором ', [[]]*”
-Хаски

В результате выполнения курсовой работы:

- Рассмотрены статьи/проекты по текущей теме
- Повторены принципы SIMD архитектуры.
- Создано простейшее 256-битное SIMD-АЛУ на языке Python с использованием стороннего модуля nmigen.
- Проверена работа АЛУ на созданных тестах.

Список используемой литературы

1. SIMD: https://en.wikipedia.org/wiki/SIMD, лекции / теоретические сведения 1й ЛР.
2. Работа с nmigen: https://github.com/RobertBaruch/nmigen-tutorial, https://vivonomicon.com/2020/06/13/lets-write-a-minimal-risc-v-cpu-in-nmigen [Электронный доступ 19.12.2021]

Приложение

Исходный код: https://github.com/Lirosk/256b_SIMD_ALU [Электронный доступ 19.12.2021]