

Архитектура.

- Архитектура планируется стековая, безадресная.
- Как такового отдельного стека (операндов) не предполагается, это область памяти, на которую указывает пара регистров.
- В дальнейшем для работы со стеком операндов предполагается небольшой кэш ~ на пару десятков слов.
- Стек локальных переменных также расположен в памяти
- Предполагается что оба этих стека расположены в одной области памяти и растут навстречу друг другу, при встрече возникает аппаратная ошибка.
- Инструкции упаковываются алгоритмом Vluint7, каждая инструкция представлена опкодом и аргументами. И опкод и аргументы закодированы Vluint7. Это позволит не бояться, что внезапно закончатся опкоды, а также даст возможность плавного перехода с 32-х на 64-х разрядную архитектуру.
- Два младших разряда опкода - число аргументов, облегчим жизнь декодеру.
- Потоки управления и исполнения разделены. Т.е. есть два независимых декодера - потока управления и потока исполнения (strands). И два счетчика команд.
- Инструкция из потока управления может запустить новый strand, после чего дожидается конца его работы (когда потоку исполнения не встретится стоп-инструкция). После возврата, поток управления продолжает работу.

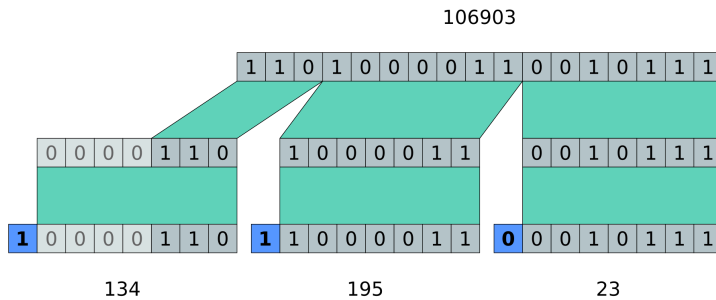
Подготовка.

Распаковщик Vluint7

Как мы условились, поток инструкций - это записанные подряд числа, упакованные Vluint7. В этом алгоритме число (не важно, 64-х, 32-х или 16-разрядное) записывается как последовательность байт, в каждом из которых 7 значащих разрядов и один управляющий, который означает- закончена запись числа или нет. Так, 32-х разрядное значение может потребовать от 1 до 5 байт. Но поскольку идентификаторы инструкций или сдвиги до данных (из которых предположительно состоит код) обычно небольшие числа, такая запись довольно компактна.

Есть два варианта записи - начиная с младших или со старших разрядов. Вторым вариантом показан на Фиг.5.2.1, но мы будем использовать другое, он представляется чуть более простым в реализации.

5.2. Практика. Постановка задачи.



Фиг.5.2.1 Кодирование методом Vuint7.([отсюда](#)),

Для тестирования [модуля](#) распаковки Vuint7 выберем следующие данные: три числа 0x4b6e, 0x58, 0x1ab1d, в бинарном виде это выглядит так:

00000000 00000000 01001011 01101110
00000000 00000000 00000000 01011000
00000000 00000001 10101011 00011101

цветами выделены блоки по 7 разрядов.

Всего получилось уложиться в 7 байтов, они показаны на Фиг.5.2.2

```
ee 96 01 58 9d d6 06 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```

Фиг.5.2.2 Содержимое памяти, для тестирования выбран блок в 64 байта.

Модуль vuint7 имеет интерфейс:

```
3 module vuint7 (
4     input wire clk,
5     input wire reset,
6     input wire beg,
7     input wire [`MEM_ADDR_WIDTH-1:0] addr,
8
9     output logic [`MEM_ADDR_WIDTH-1:0] addr_out,
10    output logic rd,
11    output logic [`WORD_WIDTH-1:0] data
12 );
```

Фиг.5.2.3 интерфейс модуля

Здесь:

- clk: синхроимпульс
- reset: сброс состояния
- beg: сигнал к распаковке
- addr: адрес начала распаковки
- addr_out: адрес на котором закончилась распаковка
- rd: сигнал об окончании распаковки
- data: распакованные данные

5.2. Практика. Постановка задачи.

Распаковка начинается с приходом сигнала beg

```
39     always @ (posedge beg) begin
40         loc_rd <= 0;           // ожидание чтения памяти
41         loc_beg <= 1;         // начинаем читать память
42         data <= 0;            //
43         loc_shift <= 0;       // распаковка с младших разрядов
44         loc_addr <= addr;     //
45         rd <= 0;              // результат не готов
46         working <= 1;        // распаковываем
47     end
```

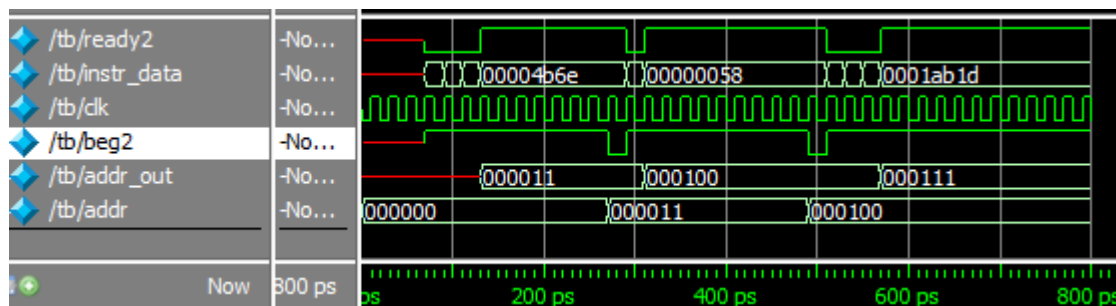
Фиг.5.2.4 начало работы

Собственно распаковка:

```
51     always @ (posedge loc_wrd) begin
52         if (working) begin
53             data <= data | (tmp_data[6:0] << loc_shift);
54             loc_addr ++;
55             if (tmp_data[7]) begin
56                 loc_shift += 7;
57                 loc_beg <= 1;
58             end else begin
59                 rd <= 1;
60                 working <= 0;
61                 addr_out = loc_addr;
62                 loc_beg <= 0;
63             end
64         end
65     end
```

Фиг.5.2.4 распаковка

Всё довольно просто, по окончании чтения памяти, сохраняем текущие 7 разрядов в их позицию, наращиваем адрес чтения, отдаём команду на чтение и наращиваем позицию сдвига данных.



Фиг.5.2.3 результат работы симулятора

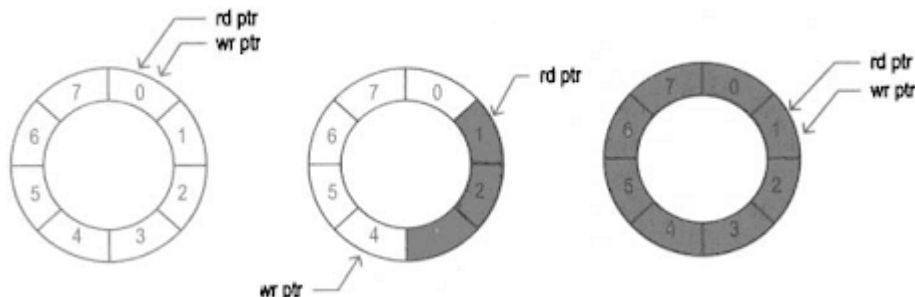
Эмулятор (в данном случае ModelSim от Altera, тут без него не обойтись) демонстрирует нам что распакованы три числа, прочитано 7 байтов памяти.

5.2. Практика. Постановка задачи.

Синхронное FIFO¹

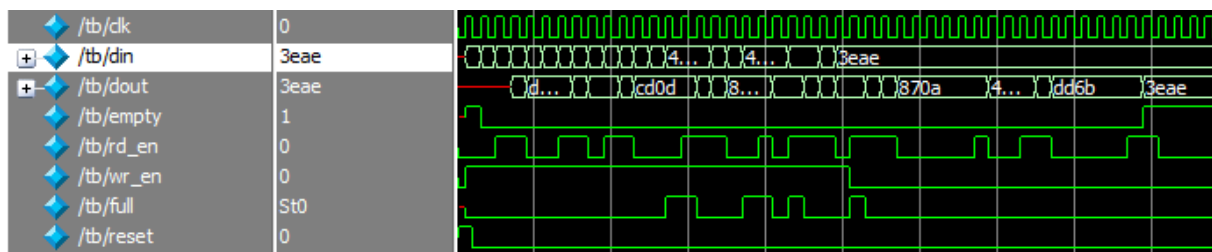
Прежде чем приступить к декодеру команд, нам потребуется буфер, в который декодер сможет записывать инструкции и из которого их можно будет вычитывать для исполнения. Дисциплина записи/чтения - очередь. Писатель и читатель работают на одной частоте, поэтому очередь синхронная.

Синхронная очередь - довольно простое устройство - это кольцевой буфер с указателями чтения / записи.



Фиг.5.2.4 состояния кольцевого буфера: пустой, частично заполненный, полный

Автор, не мудрствуя лукаво, взял готовый verilog [модуль \(sync_fifo\)](#) и использовал его с минимальными доработками. В силу особенностей реализации он способен работать только с буфером размером в степень двойки, но нас это вполне устроит.



Фиг.5.2.5 эмуляция модуля sync_fifo

На Фиг.5.2.5 показана эмуляция работы, чтение идёт со случайными задержками (/tb/rd_en), поэтому буфер время от времени переполняется (/tb/full), при этом приостанавливается запись (/tb/din, /tb/wr_en).

Декодер инструкций.

Для начала следует определиться с инструкциями, из опкодами и аргументами. Попробуем вычислить выражение $(a + 3) * b - c$. Для этого требуются следующие инструкции:

- stop - конец работы, opcode = 0, аргументов нет
- varpush - кладём на вершину стека адрес переменной, opcode = 1, один аргумент (адрес)

¹ First In First Out, очередь

5.2. Практика. Постановка задачи.

- **eval** - вычисляем значение переменной, берём адрес с вершины стека и вместо него помещаем значение по адресу, opcode=2, аргументов нет
- **imdpush** - помещаем на вершину стека аргумент - число, opcode=3, аргумент один (число)
- **pop** - удаляем элемент с вершины стека, opcode=4, аргументов нет
- **add** - удаляем из стека два элемента, складываем и сумму кладём в стек, opcode=5, аргументов нет
- **sub** - удаляем из стека два элемента, вычитаем и разность кладём в стек, opcode=6, аргументов нет
- **mul** - удаляем из стека два элемента, перемножаем и произведение кладём в стек, opcode=7, аргументов нет

Пусть адрес a=0, b=4, c=8.

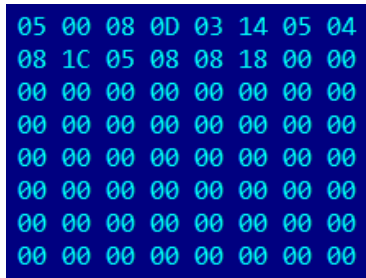
Опкоды и адреса на первых порах 16-разрядные, в нашей плате всего чуть больше 32 Кб памяти (15 разрядов).

Вышеприведённый пример порождает следующий набор инструкций:

Не забываем, что в младшие два разряда opcode помещается число аргументов.

мнемоника/ ассемблер	opcode	N args	args	код
varpush a	1	1	0	0005 0000
eval	2	0		0008
imdpush 3	3	1	3	000D 0003
add	5	0		0014
varpush b	1	1	4	0005 0004
eval	2	0		0008
mul	7	0		001C
varpush c	1	1	8	0005 0008
eval	2	0		0008
minus	6	0		0018

Поскольку все значения адресов и опкодов укладываются в 7 разрядов, упаковка в Vuint7 оказывается тривиальной, содержимое буфера памяти с кодом показано на Фиг.5.2.6.



Фиг.5.2.6 бинарный код, соответствующий выражению $(a + 3) * b - c$

В сущности, у нас всё есть для создания декодера инструкций: мы умеем буферизировать поток опкодов и операндов с помощью очереди, инициализировать и читать (синхронную) память, осталась малость - научиться вычитывать из очереди не просто значения, а именно инструкции - опкод и нужное число аргументов в одной транзакции. Впрочем, [это не сложно](#).

В цикле:

- читаем элемент из очереди

```
35      // Wait until there is data in fifo
36      while (fifo_empty) begin
37          fifo_rd <= 0;
38          $display("[%0t] FIFO is empty, wait for writes to happen", $time);
39          @(posedge clk);
40      end;
41
42      // Sample new values from FIFO
43      fifo_rd <= 1'b1;
44      @(posedge clk);
45      fifo_rd <= 1'b0;
46      @(posedge clk);
```

Фиг.5.2.7 ожидание и чтение из очереди

- делаем несколько итераций - по числу аргументов, которое хранится в двух младших разрядах инструкции

```
48      if (nargs) begin
49          cur_args[cur_arg] <= fifo_data;
50          nargs <= nargs - 1;
51          cur_arg <= cur_arg + 1;
52      end else begin
53          cur_instr <= fifo_data;
54          has_smth <= 1'b1;
55          cur_arg <= 0;
56          nargs <= fifo_data[1:0];
57      end;
```

Фиг.5.2.8

5.2. Практика. Постановка задачи.

- распаковка завершена

```
59         if (has_smth && cur_instr == 0)
60             stop <= 1'b1;
61         else if (has_smth && nargs == 0) begin
62             case (cur_instr[7:2])
63                 6'b000001: $display("[%0t] VARPUSH %d", $time, cur_args[0]);
64                 6'b000010: $display("[%0t] EVAL", $time);
65                 6'b000011: $display("[%0t] IMDPUSH %d", $time, cur_args[0]);
66                 6'b000101: $display("[%0t] ADD", $time);
67                 6'b000111: $display("[%0t] MUL", $time);
68                 6'b000110: $display("[%0t] MINUS", $time);
```

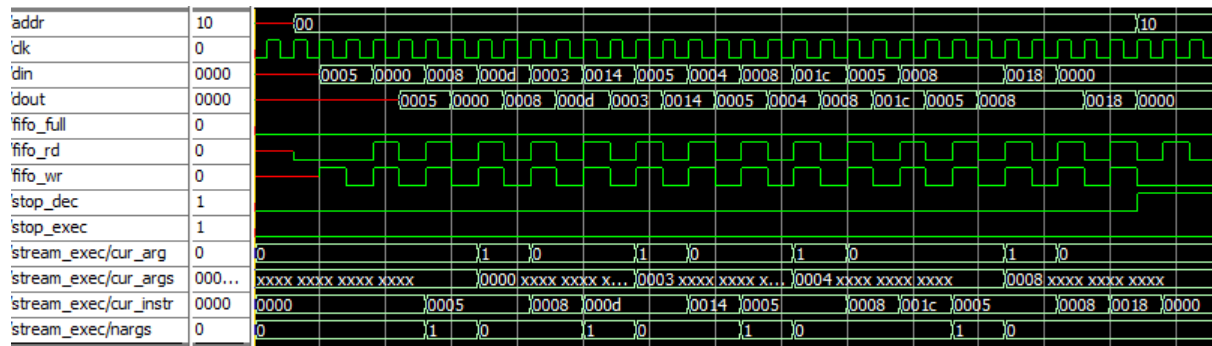
Фиг.5.2.9 отладочная печать распакованных инструкций

- Запускаем пример в отладчике и получаем заветные слова в консоли отладчика

```
# [210] VARPUSH    0
# [250] EVAL
# [330] IMDPUSH    3
# [370] ADD
# [450] VARPUSH    4
# [490] EVAL
# [530] MUL
# [610] VARPUSH    8
# [650] EVAL
# [690] MINUS
```

Фиг.5.2.10 отладочная печать вышеописанного примера

На сладкое - диаграмма состояния



Фиг.5.2.11 временная диаграмма

Что же, мы готовы к созданию калькулятора (вычислителя выражений).

Калькулятор.

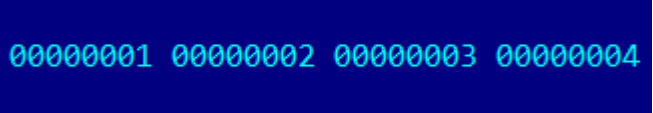
После предыдущих экспериментов у нас есть всё, что нужно для создания [калькулятора](#). Нам потребуется стек и стековые инструкции, которые с ним будут работать.

5.2. Практика. Постановка задачи.

Стек

Стек - это область памяти с дисциплиной записи/чтения LIFO². Мы уже использовали одну область памяти для хранения исполняемого кода (модуль `vuint7`), теперь модуль памяти придётся параметризовать т.к. изменится размер элемента памяти и размер области а также загрузка данных (для отладки) из разных файлов.

- Размер слова в стеке выберем в 32 разряда,
- стек будет размером в 256 слов (1K), т.е. ширина адреса - 8 разрядов
- начинаться стек будет не с начала области памяти, а с отступа в 16 байт, которые будут использованы под локальные переменные.



Фиг.5.2.12 область, занятая переменными, всего 4 слова (инициализационный файл), остальное неопределено

- вершина стека определяется значением регистра ***executer.stack_top*** с адресацией в байтах, регистр смотрит на слово, следующее за последним заполненным (словом)

Инструкции.

Минимальный набор инструкций описан в разделе “декодер инструкций”, разберем реализацию какой-нибудь одной, пусть ***eval***, в ней есть и чтение стека и запись в него.

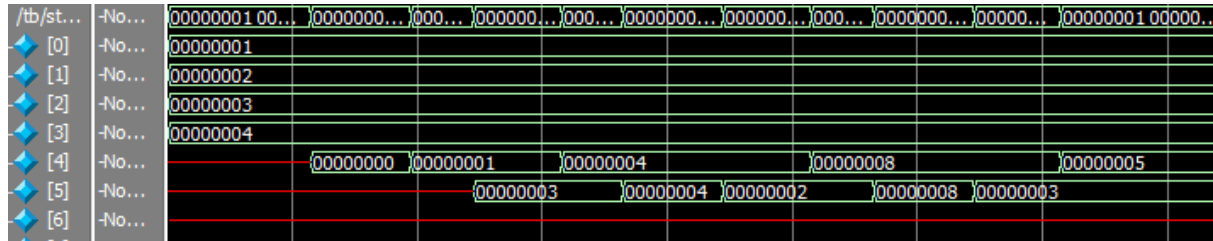
```
125         6'b000010: begin // EVAL
126             tmp_addr <= stack_top - stack_step;
127             loc_we <= 0;
128             while (!loc_wrd) begin
129                 @(posedge clk);
130             end;
131             @(posedge clk);
132             @(posedge clk);
133
134             tmp_addr <= tmp_data_reg_in;
135
136             @(posedge clk);
137             @(posedge clk);
138
139             tmp_addr <= stack_top - stack_step;
140             tmp_data_reg_out <= tmp_data_reg_in;
141
142             @(posedge clk);
143             loc_we <= 1;
144             while (!loc_wrd) begin
145                 @(posedge clk);
146             end;
147             @(posedge clk);
148             loc_we <= 0;
149
150             $display("[%0t] EVAL", $time);
151         end
```

Фиг.5.2.13 инструкция eval.

² Last In First Out, последним пришел, первым уйдешь

5.2. Практика. Постановка задачи.

- 6'b000010 - это 2, опкод инструкции eval
- stack_top - stack_step вычисляем адрес вершины стека и начинаем чтение по этому адресу. Должны вычитать адрес переменной в tmp_data_reg_in
- устанавливаем адрес загрузки в tmp_addr и ожидаем окончания чтения
- записываем прочитанное значение переменной и записываем его в вершину стека через регистр tmp_data_reg_out
- loc_we (write enabled)- в случае нулевого значения читаем данные иначе пишем



Фиг.5.2.14 временная диаграмма содержимого стека

Разберём диаграмму 5.2.14.

- изначально инициализированы первые 4 слова, это переменные a=1 (адрес 0), b=2 (... 4), c=3 (... 8), d=4 (... 12)
- инструкция *varpush a* помещает адрес a (т.е. 0) на вершину стека (адрес 16)
- инструкция *eval* замещает на вершине стека адрес переменной значением (т.е. 1)
- *imdpush 3* дописывает в стек значение 3
- инструкция *add* складывает последние два значения на вершине стека, удаляет их из стека и записывает в стек сумму $3 + 1 \Rightarrow 4$
- *varpush b* записывает в стек адрес b т.е. 4
- *eval* замещает на вершине стека адрес переменной значением (т.е. 2)
- инструкция *mul* перемножает два аргумента, удаляет их и записывает произведение $2 * 4 \Rightarrow 8$
- *varpush c* записывает в стек адрес c т.е. 8
- *eval* превращает его в 3
- инструкция *sub* вычитает аргументы, удаляет их из стека и записывает разность $8 - 3 \Rightarrow 5$
- на этот раз переменная d не пригодилась

Результат на вершине стека, $(a + 3) * b - c = 5$

Итого, имеем работающий вычислитель выражений с двухстадийным конвейером и базовой арифметикой. Наш следующий шаг - инструкции потока управления и их собственный конвейер.

Поток управления.