АРБИТРЫ И РАЗДЕЛЕНИЕ ПАМЯТИ МЕЖДУ НЕСКОЛЬКИМИ ПРОЦЕССОРНЫМИ ЯДРАМИ

школа синтеза цифровых схем

ПРИ ПАРТНЕРСТВЕ

YADRO

YADRO · MP

Syntacore™

Занятие №9

12 ноября 2022

Дмитрий Смехов

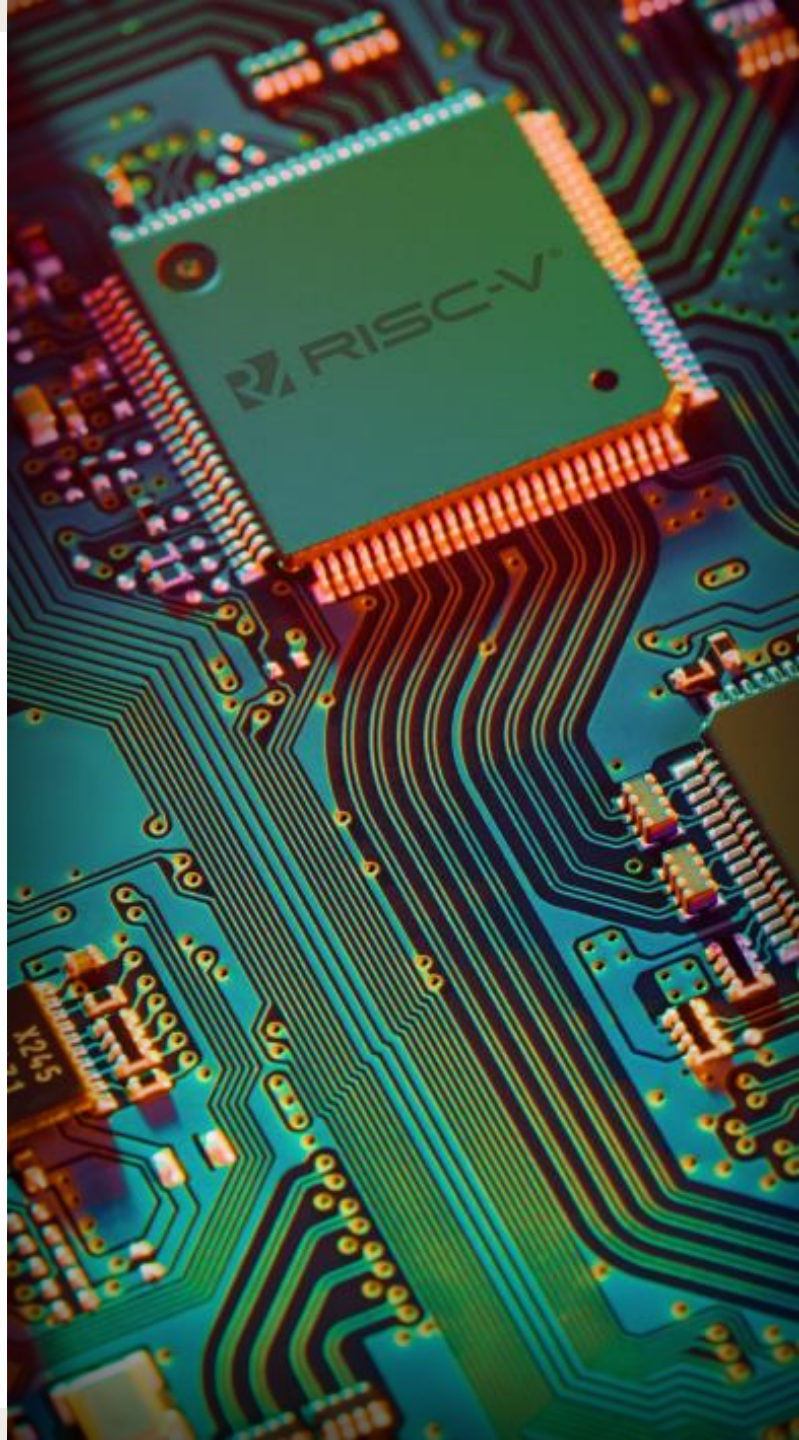
Инженер разработчик ПЛИС

2001 – 2020	«Инструментальные Системы»
2020 – н.в.	«IRQ»

ЩЕЛКНИТЕ, ЧТОБЫ ОТРЕДАКТИРОВАТЬ ТЕКСТ

ТЕМЫ ЗАНЯТИЯ

- Многопортовая память
- Арбитр
- Лабораторная работа – подключение многопортовой памяти к schoolRISCV



ЩЕЛКНИТЕ, ЧТОБЫ ОТРЕДАКТИРОВАТЬ ТЕКСТ

МАТЕРИАЛЫ К ЛАБОРАТОРНОЙ РАБОТЕ

Артём Воронов, Роман Воронов «Multibank memory: Создание многопортовой памяти, оптимальное количество банков и минимизация конфликтов». Доклад на ChipExpo 2021

https://docs.google.com/presentation/d/1fUaT1Cj00Atk1U16H0rVfGz-CQNtRglJ2jaHsq2IkZo/edit#slide=id.p

Презентация: «schoolRISCV + VGA»

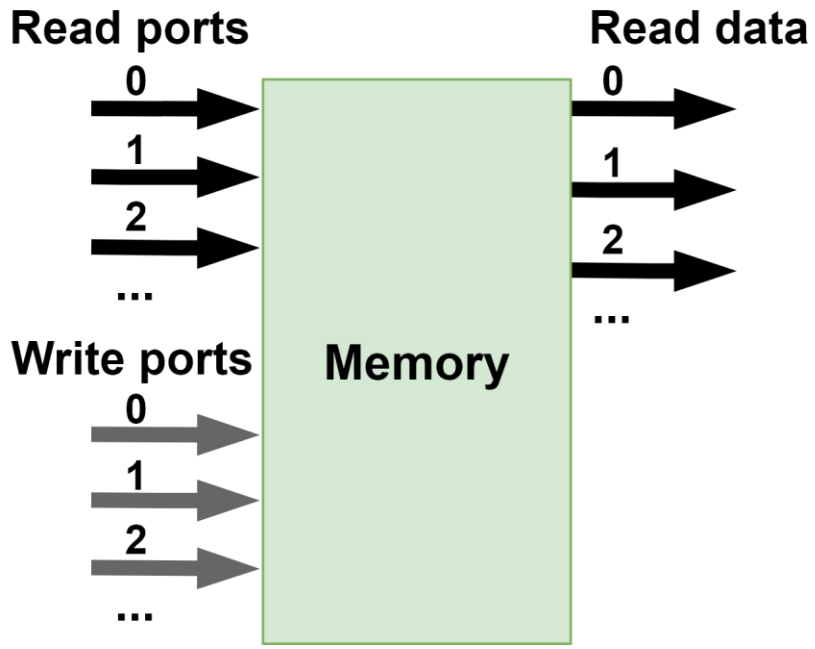
https://github.com/DigitalDesignSchool/ce2020labs/blob/master/day_4/doc/schoolRiscV_vga.pdf

Шаблон тестирования:

https://github.com/DigitalDesignSchool/ce2020labs/tree/master/next_step/dsmv/test_template/chip-expo-2021-template-1-param

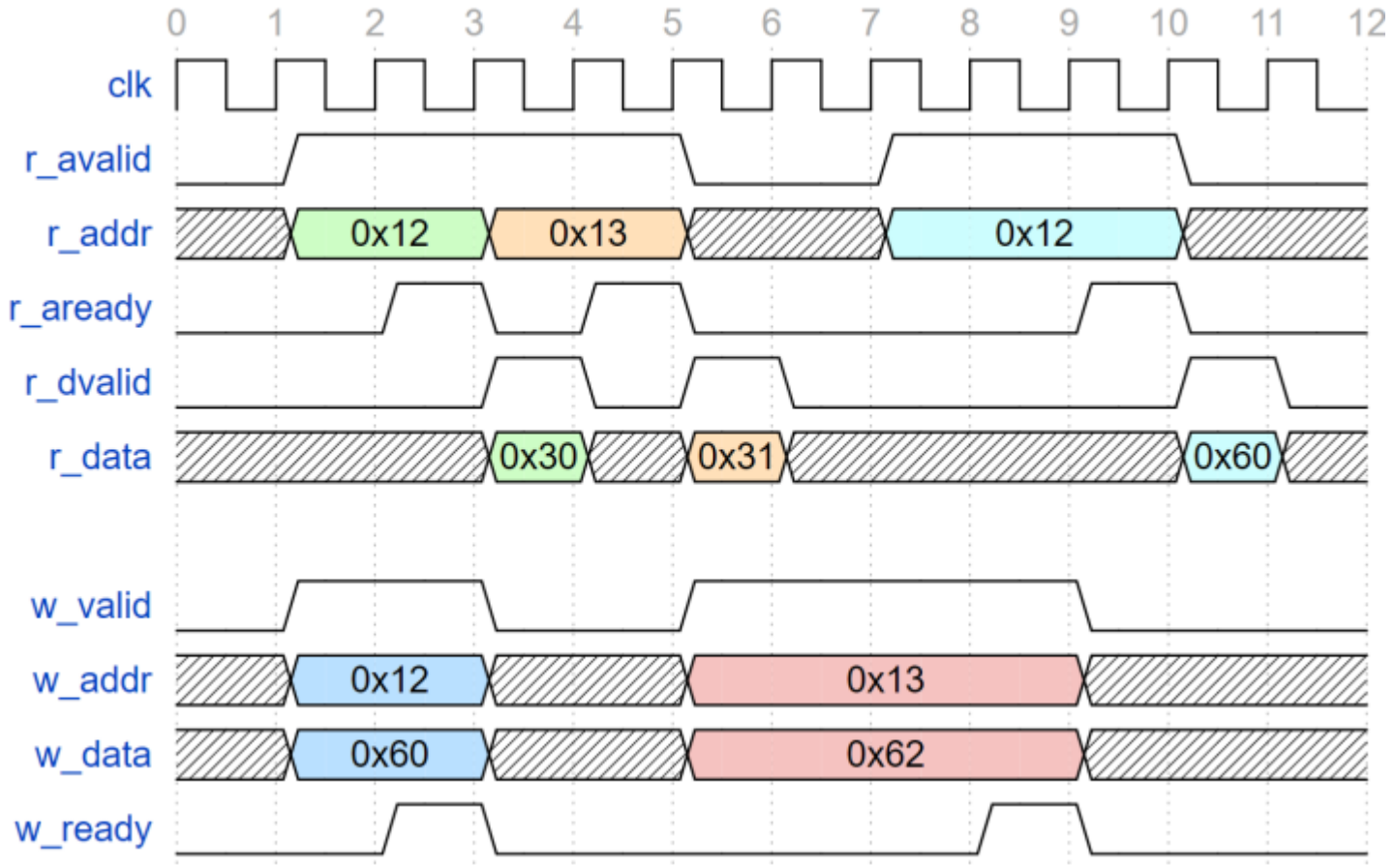
ЩЕЛКНИТЕ, ЧТОБЫ ОТРЕДАКТИРОВАТЬ ТЕКСТ

КОМПОНЕНТ МНОГОПОРТОВОЙ ПАМЯТИ



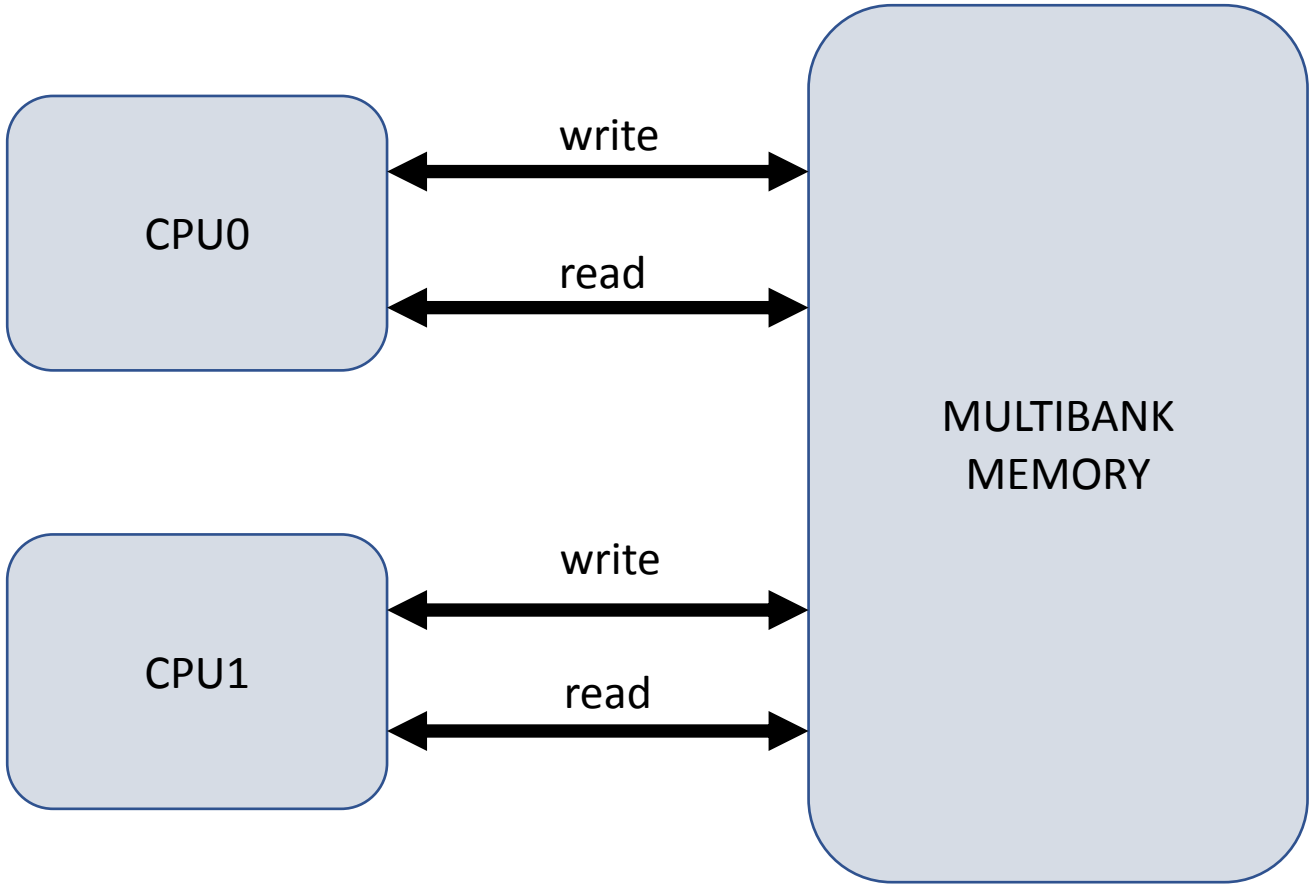
ЩЕЛКНИТЕ, ЧТОБЫ ОТРЕДАКТИРОВАТЬ ТЕКСТ

ПРИМЕР ЧТЕНИЯ И ЗАПИСИ



ЩЕЛКНИТЕ, ЧТОБЫ ОТРЕДАКТИРОВАТЬ ТЕКСТ

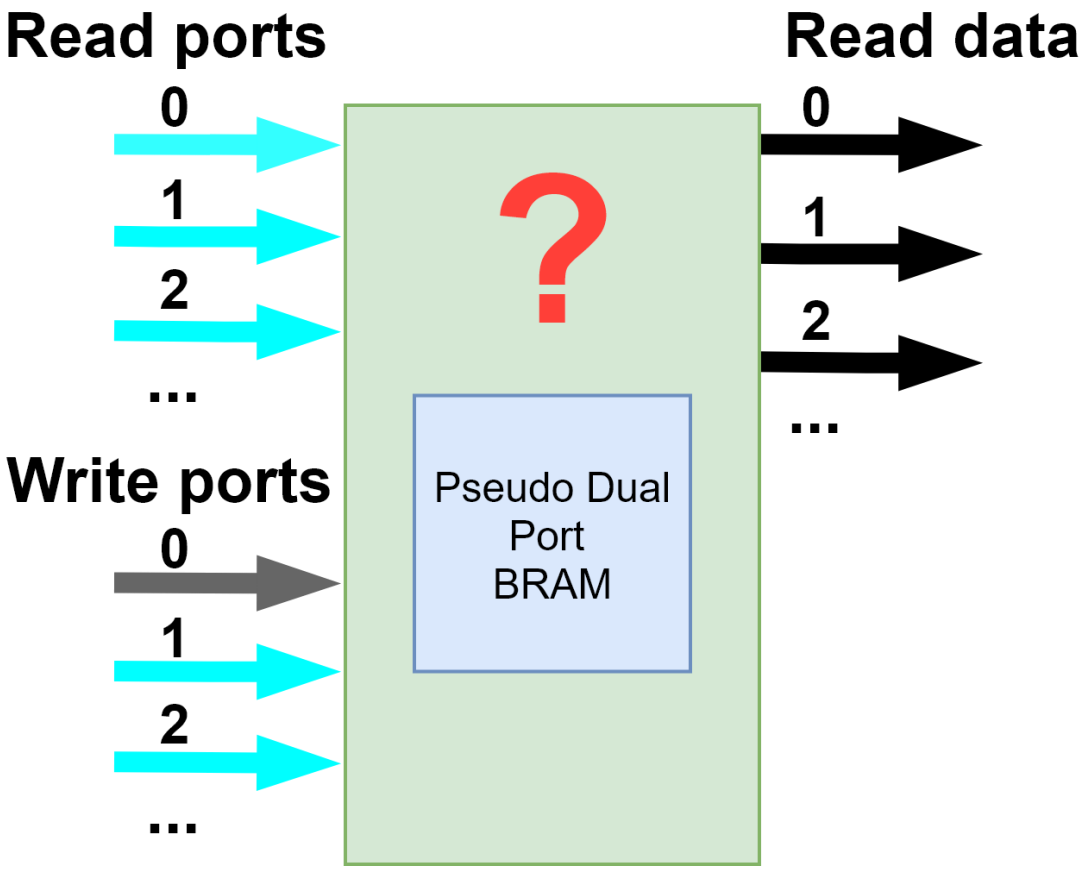
ПРИМЕР СИСТЕМЫ



ПРОБЛЕМА КОНФЛИКТОВ

Что делать при одновременном обращении к памяти по двум и более портам?

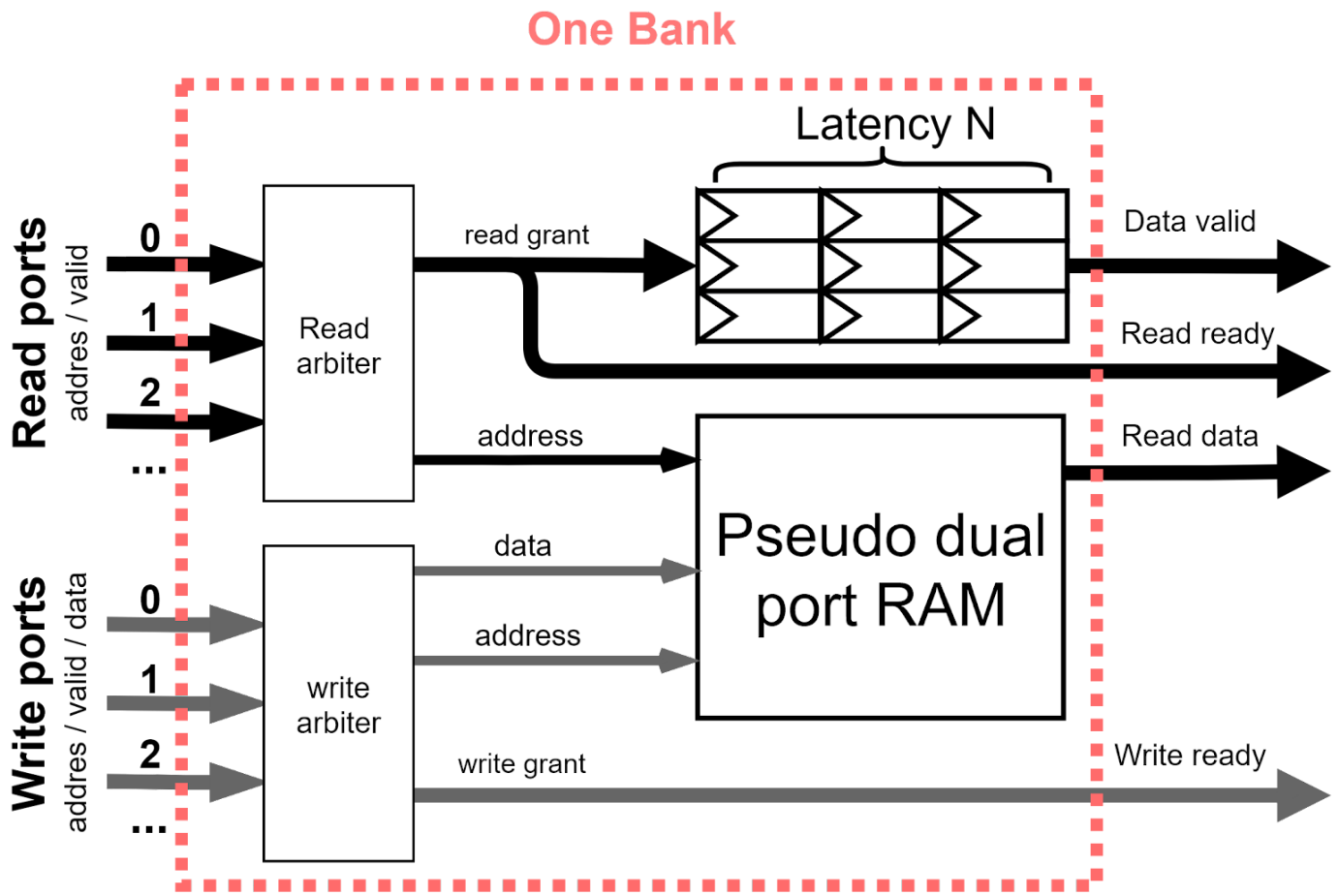
Стандартные блоки памяти позволяют обработать только один запрос.



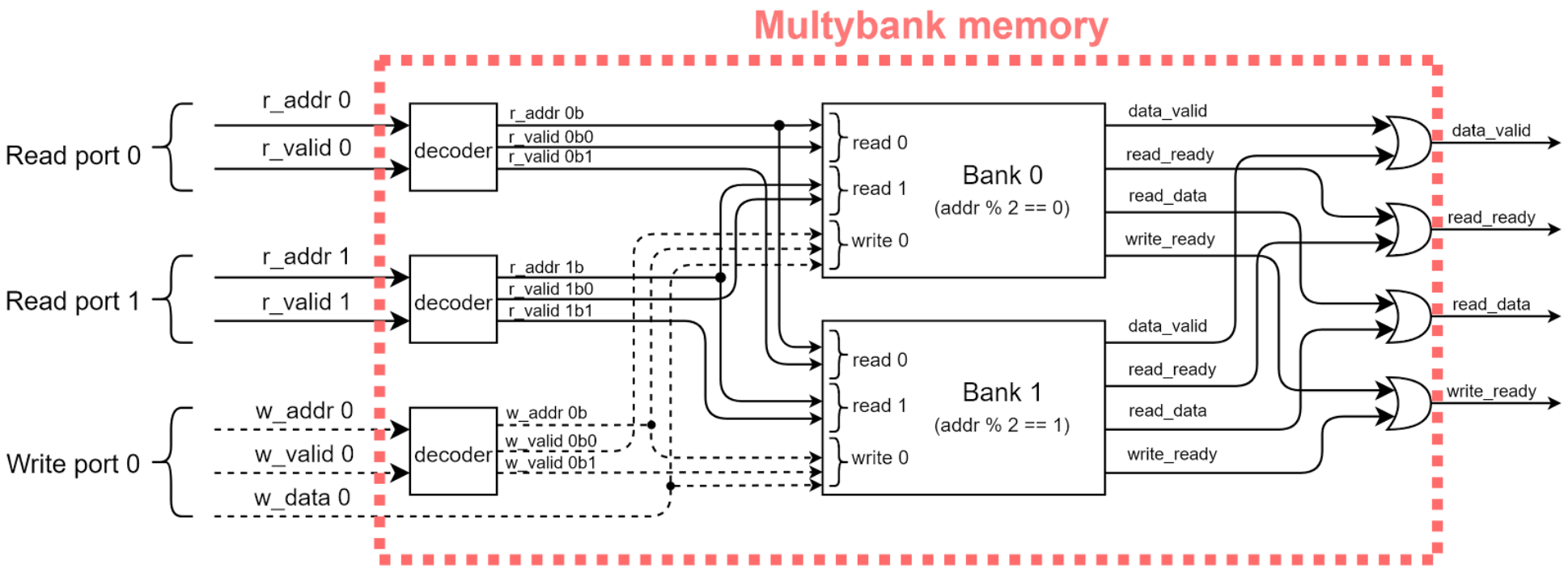
ЗАДЕРЖКА ЗАПРОСА: АРБИТРАЖ ЗАПРОСОВ

Плюсы:
- Только один блок памяти
- Малый размер на чипе

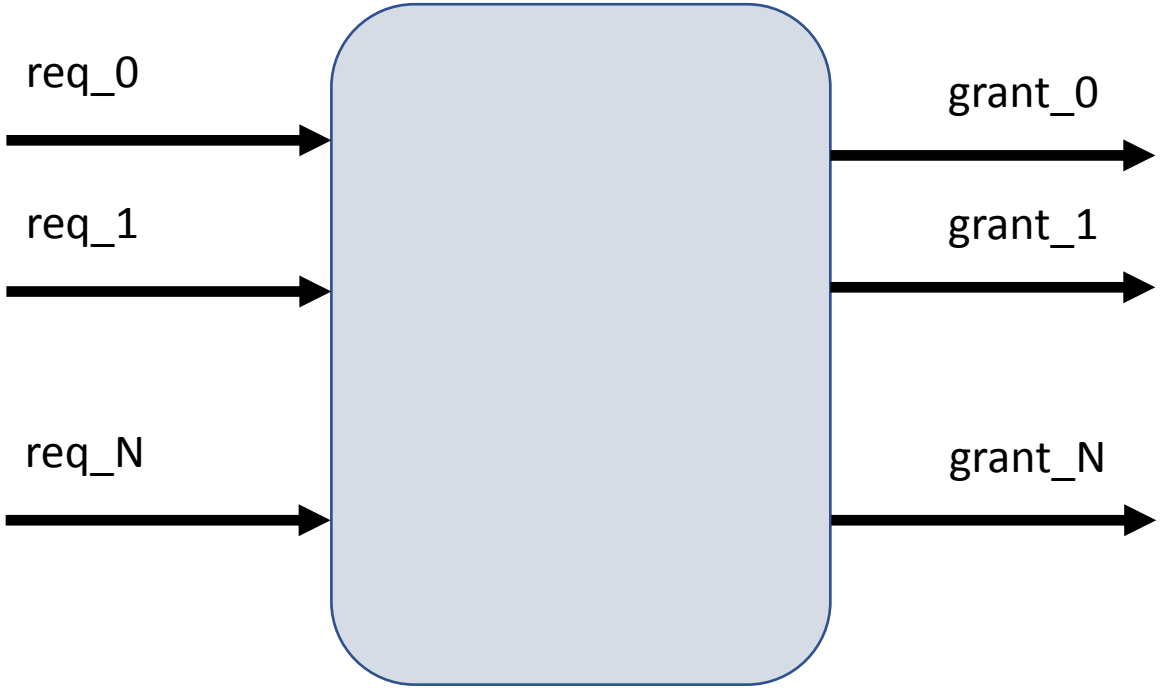
Минусы:
- Время транзакции зависит от количества запросчиков



МНОГОБАНКОВАЯ ПАМЯТЬ



АРБИТР



- несколько активных запросов
- только на один порт выдаётся разрешение
- арбитраж производится каждый такт

АРБИТР – НЕПОНЯТНЫЙ КОД

```
  reg  [N-1:0] pointer_req;
  //reg  [N-1:0] next_grant;

  wire [2*N-1:0] double_req = {req,req};
  wire [2*N-1:0] double_grant = double_req & ~(double_req - pointer_req);

  //Asynchronous grant update
//   assign grant = (rst)? {N{1'b0}} : double_grant[N-1:0] | double_grant[2*N-1:N];
  assign grant = double_grant[N-1:0] | double_grant[2*N-1:N];
```

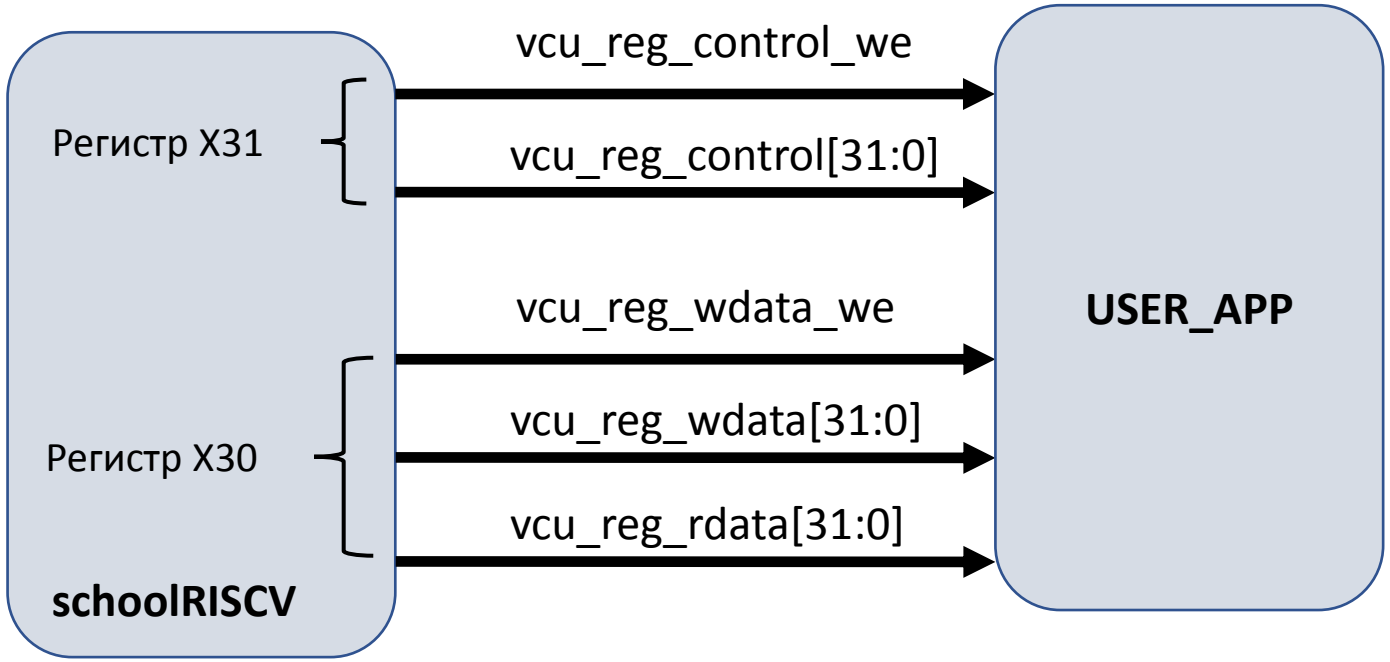
Описание работы арбитра представлено здесь:

https://docs.google.com/presentation/d/1fUaT1Cj00Atk1U16H0rVfGz-CQNtRgIJ2jaHsq2lkZo/edit#slide=id.p

lab_2/src_calc/arbiter_m2.sv - реализация на основе конечного автомата

Задание для самостоятельной работы – найти различия в поведении **arbiter** и **arbiter_m2**

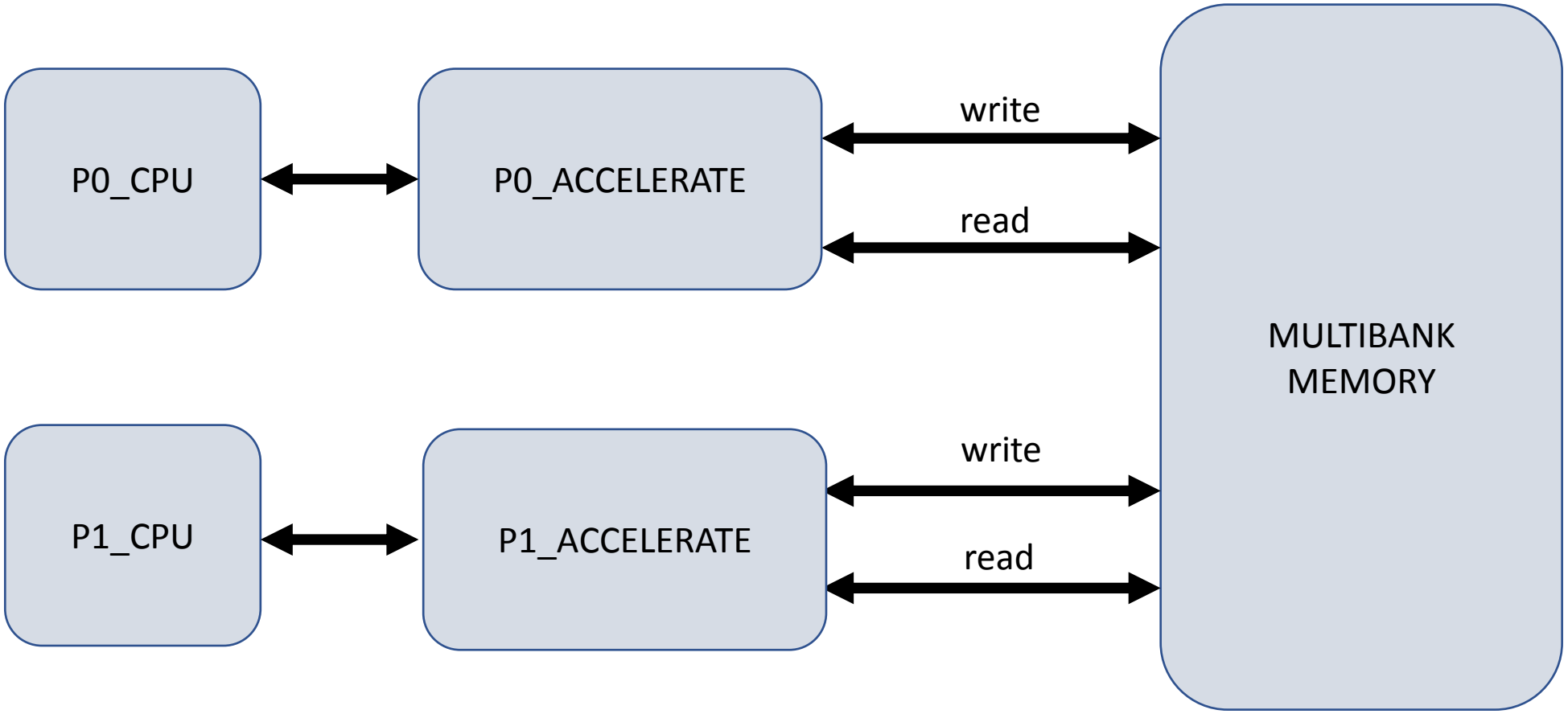
ПРОЦЕССОР schoolRISCV С ПОДКЛЮЧЕНИЕМ К VGA



- Запись в регистр X30 – формирование **vcu_reg_wdata[31:0]** и **vcu_reg_data_we**
- Чтение регистра X30 – чтение с шины **vcu_reg_rdata[31:0]**
- Запись в регистр X31 – формирование **vcu_reg_control[31:0]** и **vcu_reg_control_we**

Регистры X30, X31 имеют альтернативные названия T5 и T6 по соглашению о вызовах функций для компилятора

УЧЕБНАЯ СИСТЕМА



КОМПОНЕНТ MEMORY_ACCELERATE

Операции:

- Запись возрастающей последовательности в память
- Чтение последовательности из памяти и подсчёт суммы чисел

Входные сигналы:

- **reg_control_we** - 1:запись в регистр **REG_CONTROL** (это регистр T6 или X31)
- **reg_control[31:0]** – значение регистра
- **reg_mode_adr[31:0]** – начальный адрес
- **reg_mode_data[31:0]** – начальное значение слова данных
- **reg_mode_cnt[31:0]** – число слов для обращения

Выходные сигналы

- **w_done** – 1: завершён цикл записи
- **r_done** – 1: завершён цикл чтения
- **reg_calculate[31:0]** – подсчитанная сумма чисел

КАРТА РЕГИСТРОВ

REG_CONTROL:

- биты [3:0] – адрес косвенного регистра
- бит 8 – 1: - запуск цикла записи
- бит 9 – 1: - запуск цикла чтения

Косвенные регистры

- 0x00 –
- 0x01 – **REG_HEX** – вывод цифры на дисплей
- 0x02 – **REG_KEY0** – опрос кнопки S1 для P0 и S3 для P1
- 0x03 – **REG_KEY1** – опрос кнопки S2 для P0 и S4 для P1
- 0x04 – **REG_W_DONE** – завершение цикла записи
- 0x05 – **REG_R_DONE** – завершение цикла чтения
- 0x06 – **REG_CALCULATE** – результат суммирования
- 0x07 –
- 0x08 – **REG_MODE_ADR** – начальный адрес
- 0x09 – **REG_MODE_DATA** – начальное значение данных
- 0x0A – **REG_MODE_CNT** – число циклов

АЛГОРИТМ РАБОТЫ

Процессоры P0 и P1:

- При нажатии на **KEY0**
  - сбросить флаг нажатия на **KEY0**
  - записать последовательность длиной **N** слов с текущего адреса
  - увеличить текущий адрес на **N**
  - увеличить текущее значение данных на 1
- При нажатии на **KEY1**
  - сбросить флаг нажатия на **KEY1**
  - прочитать последовательность длиной N слов с текущего адреса
  - вывести младшую цифру суммы на дисплей

Процессор P0:

- **N=4**
- KEY0 это кнопка S1 (key_sw_p[3])
- KEY1 это кнопка S2 (key_sw_p[2])

Процессор P1:

- **N=8**
- KEY0 это кнопка S3 (key_sw_p[1])
- KEY1 это кнопка S4 (key_sw_p[0])

Система разработки

Основная система: Visual Studio Code с расширением для SystemVerilog (рекомендую TerosHDL)

Система моделирования: ModelSim, Questa

Система разработки ПЛИС: Quartus Lite

Компилирование, запуск моделирования, запуск сборки производится из встроенного терминала Visual Studio Code. Используется до 4-х терминалов.

Основные команды:

- **vlib_init.sh** - инициализация библиотеки моделирования
- **compile.sh** – компиляция проекта для моделирования
- **c_run_0.sh** – запуск моделирования в консольном режиме
- **g_run_0.sh** – запуск моделирования в режиме GUI
- **p_build.sh** – сборка программ для процессоров P0 и P1
- **x_synthesize.h** – запуск сборки проекта и загрузка проекта на плату