AQUARIUS

Введение в микропроцессорные системы

Матвеев Борис

Старший инженер направления функциональной верификации

Микропроцессор

«Микропроцессором называется программно-управляемое устройство для обработки цифровой информации и управления процессом обработки, реализованное в виде сверхбольшой интегральной микросхемы (СБИС)»

И.И. Шагурин

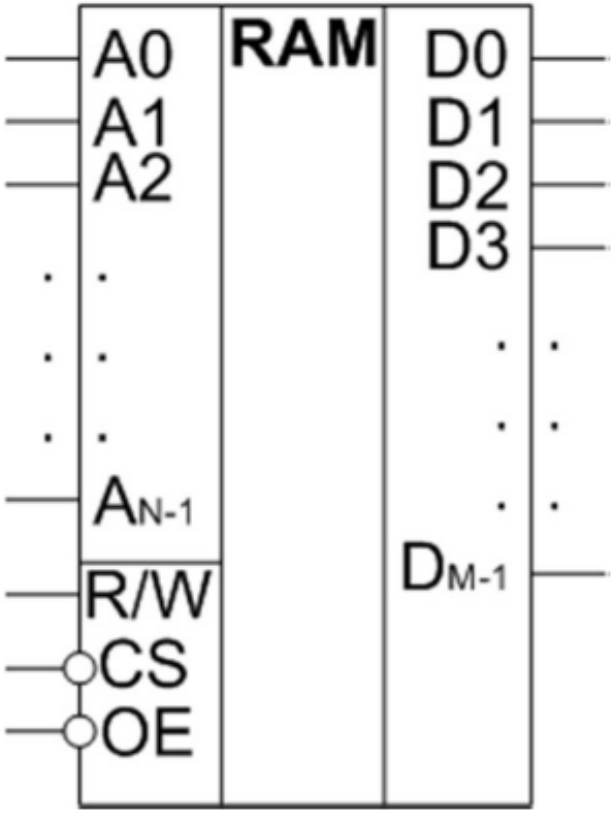
Цифровую информацию процессор может получать от внешних устройств и из памяти. Описание процесса обработки этой информации (программу) процессор также получает из памяти.

В зависимости от выбранной архитектуры процессора память может быть:

- Разделена на память инструкций и память данных (Гарвардская архитектура)
- Представлять собой единый массив данных и инструкций (Принстонская архитектура)

RAM

Условно-графическое обозначение



$A_0 \ldots A_{N-1}$ – адресные входы

$D_0 \ldots D_{M-1}$ – входы/выходы данных

R/W, CS, OE – управляющие входы

R/W
- 0 – запись
- 1 – считывание или хранение

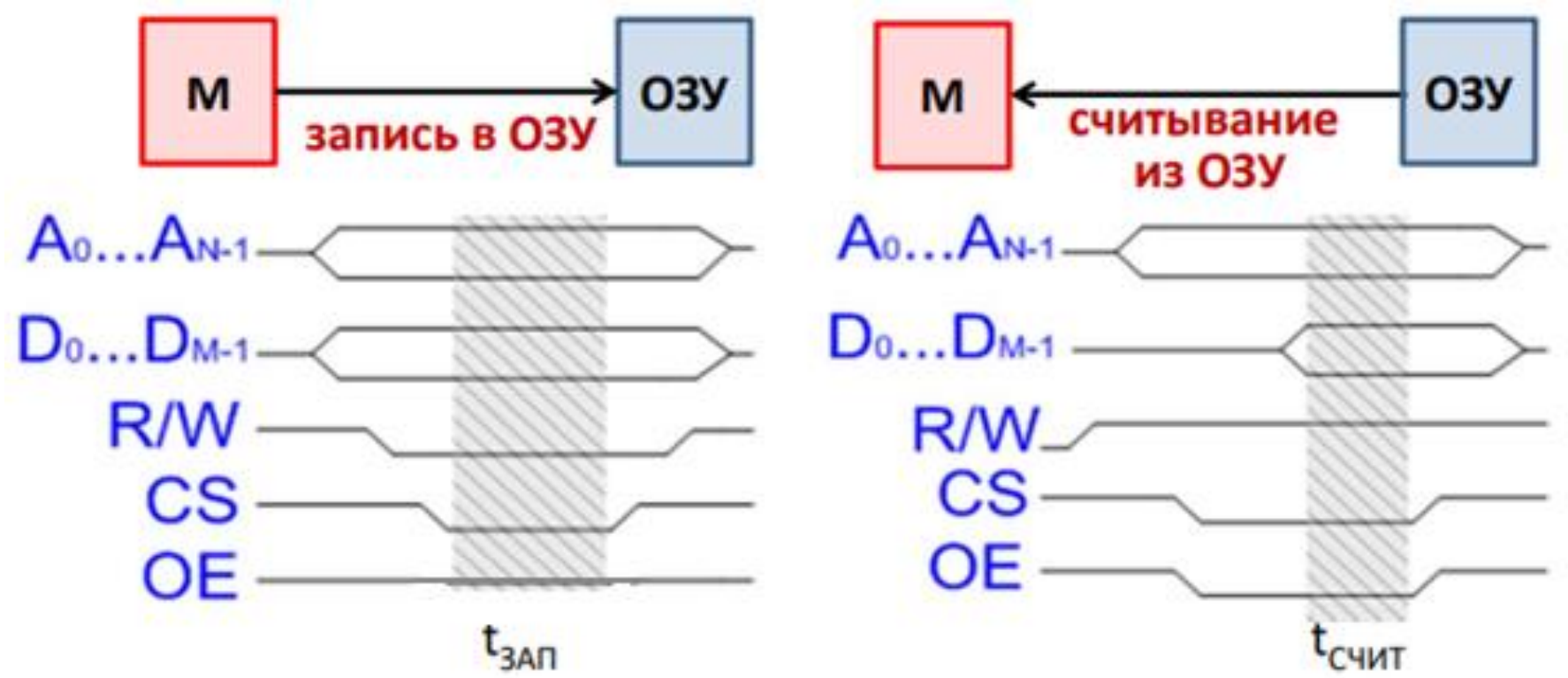
CS
- 1 – выходы 3-е состояние (хранение)
- 0 – кристалл выбран (ОЗУ)

OE
- 1 – выходы 3-е состояние (хранение)
- 0 – данные активные (ОЗУ)

RAM

Протоколы обмена

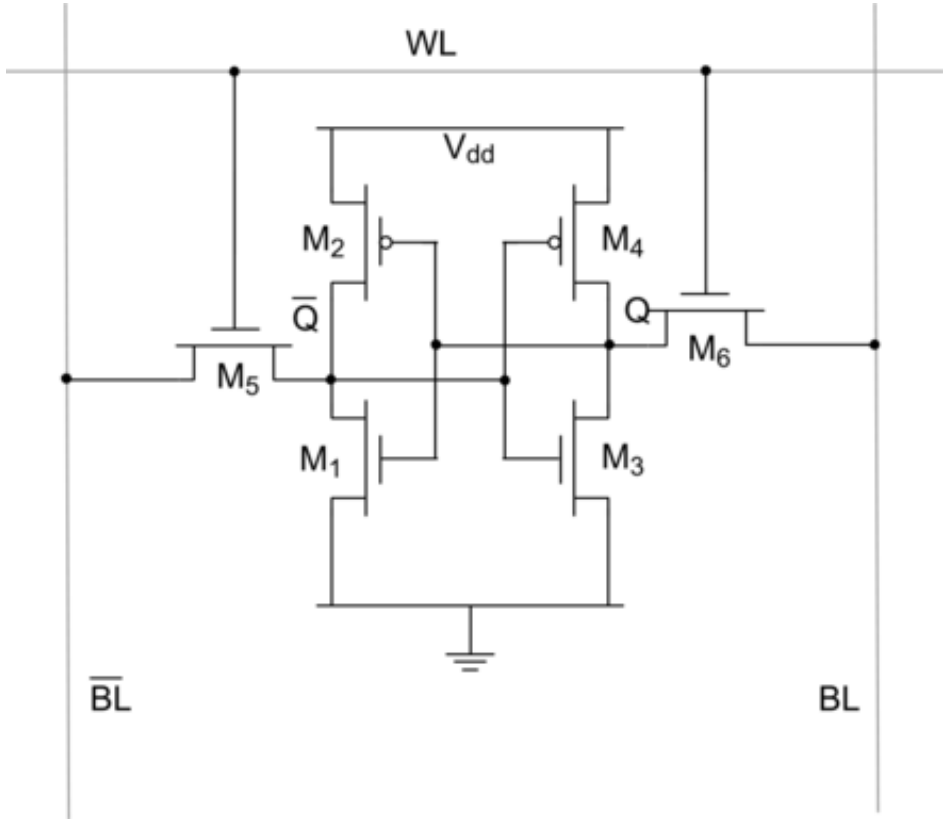
Циклы считывания и записи



Устройство SRAM памяти

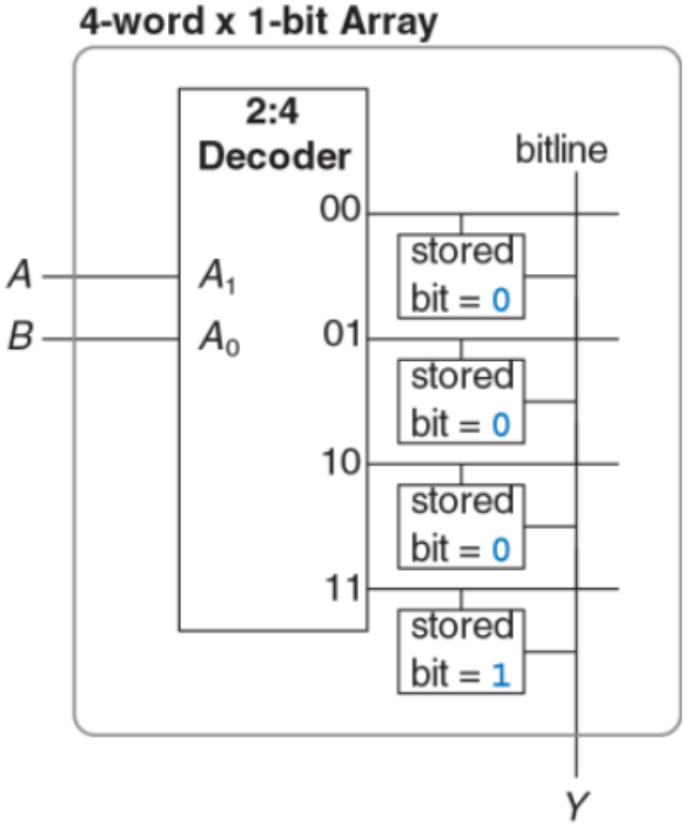
Протоколы обмена

Циклы считывания и записи

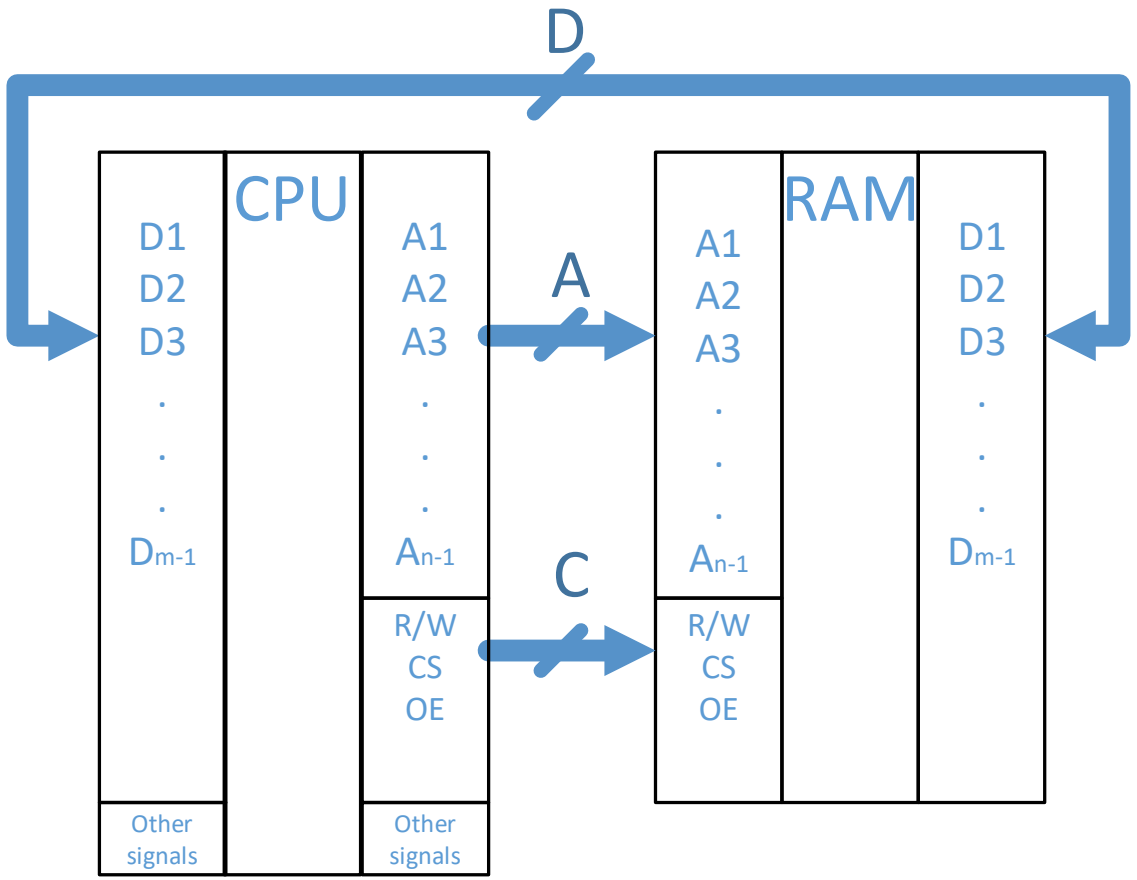


Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



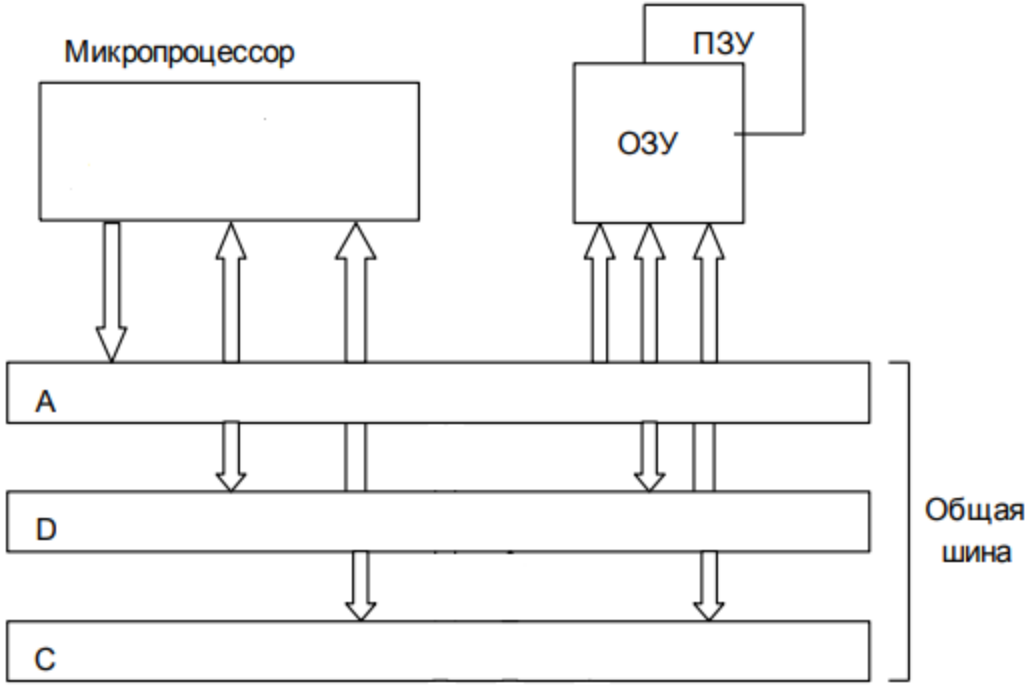
Соединение CPU и SRAM



CPU является ведущим устройством в связке CPU-RAM:

- **CPU формирует адрес обращения в память**
- **CPU может считывать данные/инструкции из памяти**
- **CPU может записывать данные в память**

Структурная схема соединения микропроцессора и памяти



*обратите внимание на направление шины адреса

AQUARIUS

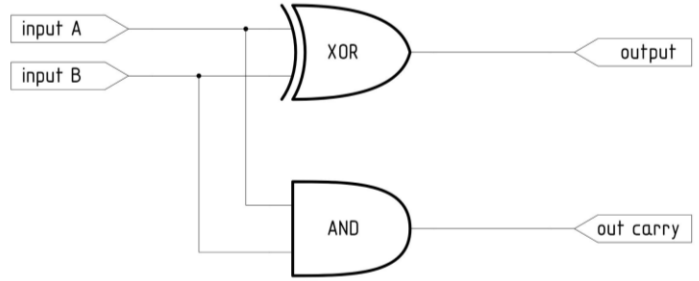
Программирование микропроцессора

Уровни представления вычислений

программа

аппаратура

Уровень абстракции	Средство	Пример
Алгоритм	Языки высокого уровня(C/C++)	z = x + y
Архитектура набора команд	Assembler – язык, доступный для чтения человеком	
Машинный код	Двоичный код	add x3, x2, x1
Микроархитектура	Блок-схемы и языки описания аппаратуры (Verilog, VHDL)	0x0012331232
Цифровой логический уровень	Логические элементы (синтез)	
Физическая реализация	Электрическая схема (расстановка и трассировка)	



Команды процессора

- Команды процессора – простейшие операции, которые он может выполнять
  - *Команды выполняются последовательно*
  - *Команда выполняет операцию над операндами*
  - *Некоторые команды могут менять последовательность выполнения программы*
- Последовательность команд, хранящаяся в памяти – программа

Структура команд

«Команда представляет собой многоразрядное двоичное число, которое состоит из двух частей – кода операции КОП и кода адресации операндов КАД.»

1 Код операции (КОП)

Код операции КОП задает вид операции, выполняемой данной командой

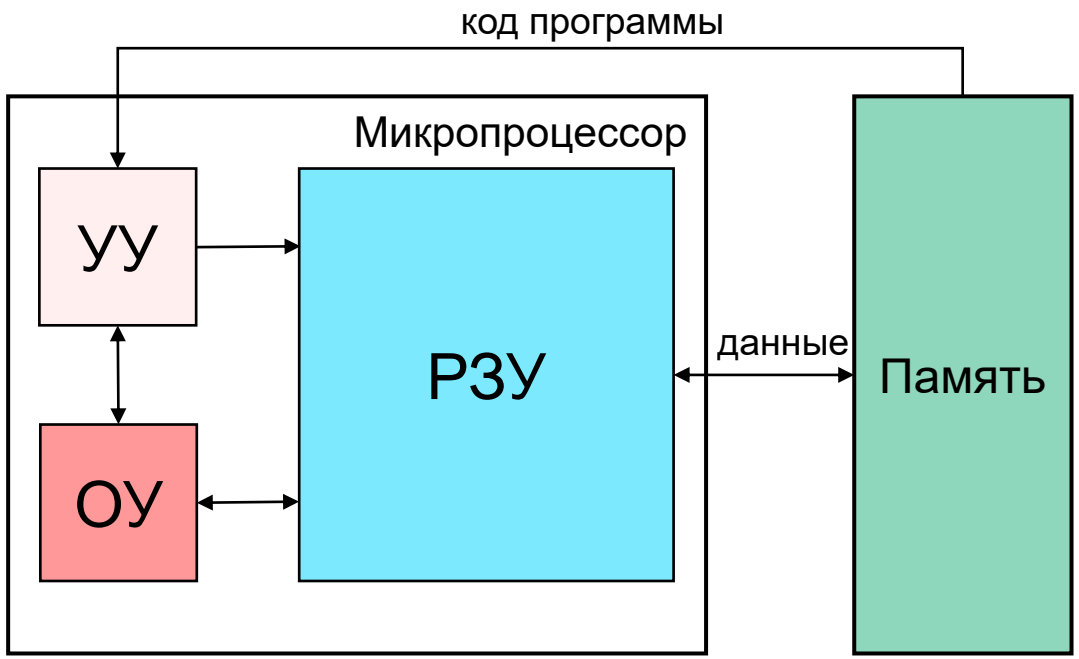
2 Код адресации операндов (КАД)

Код адресации КАД определяет выбор операндов (способ адресации), над которыми производится заданная операция.

**В зависимости от выбранной архитектуры код операции может разбиваться на дополнительные поля*

Операнды команд

Регистры в процессоре

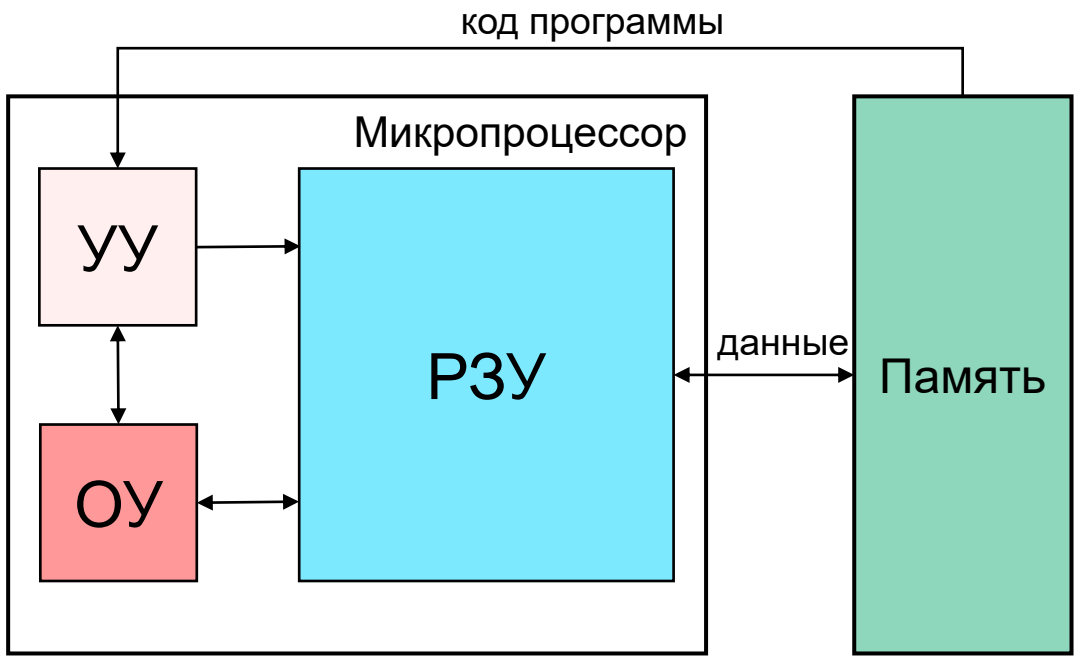


Микропроцессор в своем составе содержит:

УУ – управляющее устройство
ОУ – оперативное устройство
РЗУ – регистровое запоминающее устройство (регистровый файл)

Операнды команд

Регистры в процессоре



Регистры предназначены для хранения данных в процессоре.

В рамках RISC архитектур арифметические операции могут производиться только с регистрами.

Чтобы провести арифметическую операцию над некоторым числом, его необходимо сначала загрузить в регистр процессора.

Переменные в ассемблере - регистры

«Язык ассемблера (англ. assembly language) — представление команд процессора в виде, доступном для чтения человеком.»

- В отличие от языков высокого уровня в ассемблере отсутствуют переменные
- Вместо переменных команды оперируют с регистрами
  - Ограниченный набор ячеек хранения чисел, встроенных прямо в аппаратуру
  - В архитектурах RISC арифметические операции могут выполняться только с регистрами
  - С памятью возможны только операции записи и считывания (в отличие от CISC)
- Преимущество работы с регистрами – скорость доступна к ним
- Недостатки – ограниченное число регистров – 32 в RISC-V

Assembler и двоичное представление команд

«Язык ассемблера (англ. assembly language) — представление команд процессора в виде, доступном для чтения человеком.»

Assembler:

addi **rd rs1 data** ## rd = rs1 + data

rd – destination register – регистр результата

rs1 – source register – регистр-операнд

data - число

Пример:

addi **t0 t1 16** ## t0 = t1 + 16

Assembler и двоичное представление команд

Двоичное представление:

32’b 0000 0001 0000 0011 0000 0010 1001 0011

31 20	19 15	14 12	11 7	6 0
Imm[11:0]	rs1	funct3	rd	opcode
0000 0001 0000	00110	000	00101	0010011

Адрес команды

Формирование и хранение

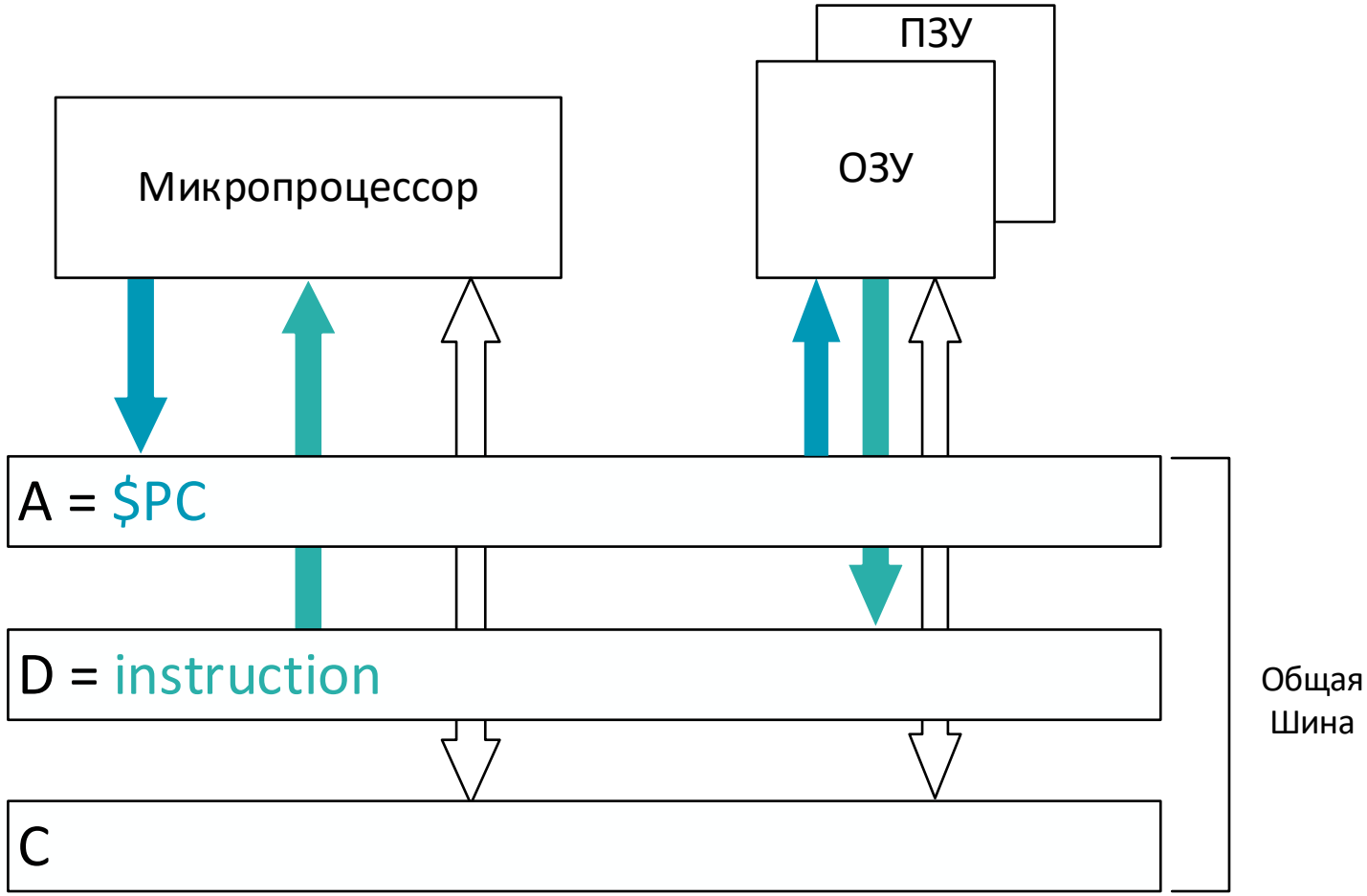
Program counter (PC) – регистр хранения адреса следующей инструкции.

Значение PC может быть изменено двумя способами:

1. PC автоматически увеличивается после исполнения команды, что позволяет исполнять программу последовательно.
2. PC изменяется специальными командами, что позволяет исполнять программу непоследовательно, то есть исполнять команды ветвления (if/case) и циклы (while/for).

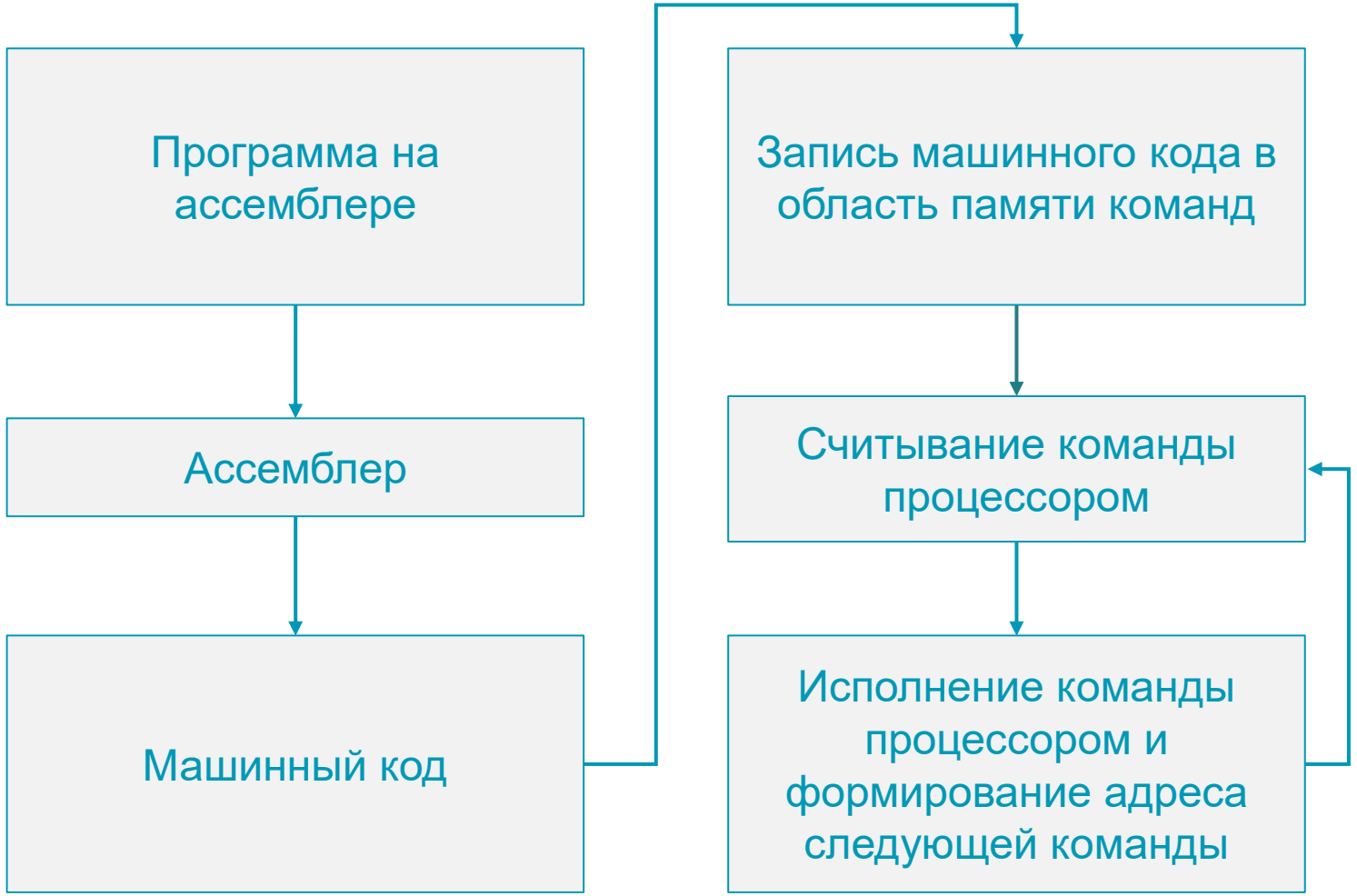
Адрес команды

Формирование и хранение

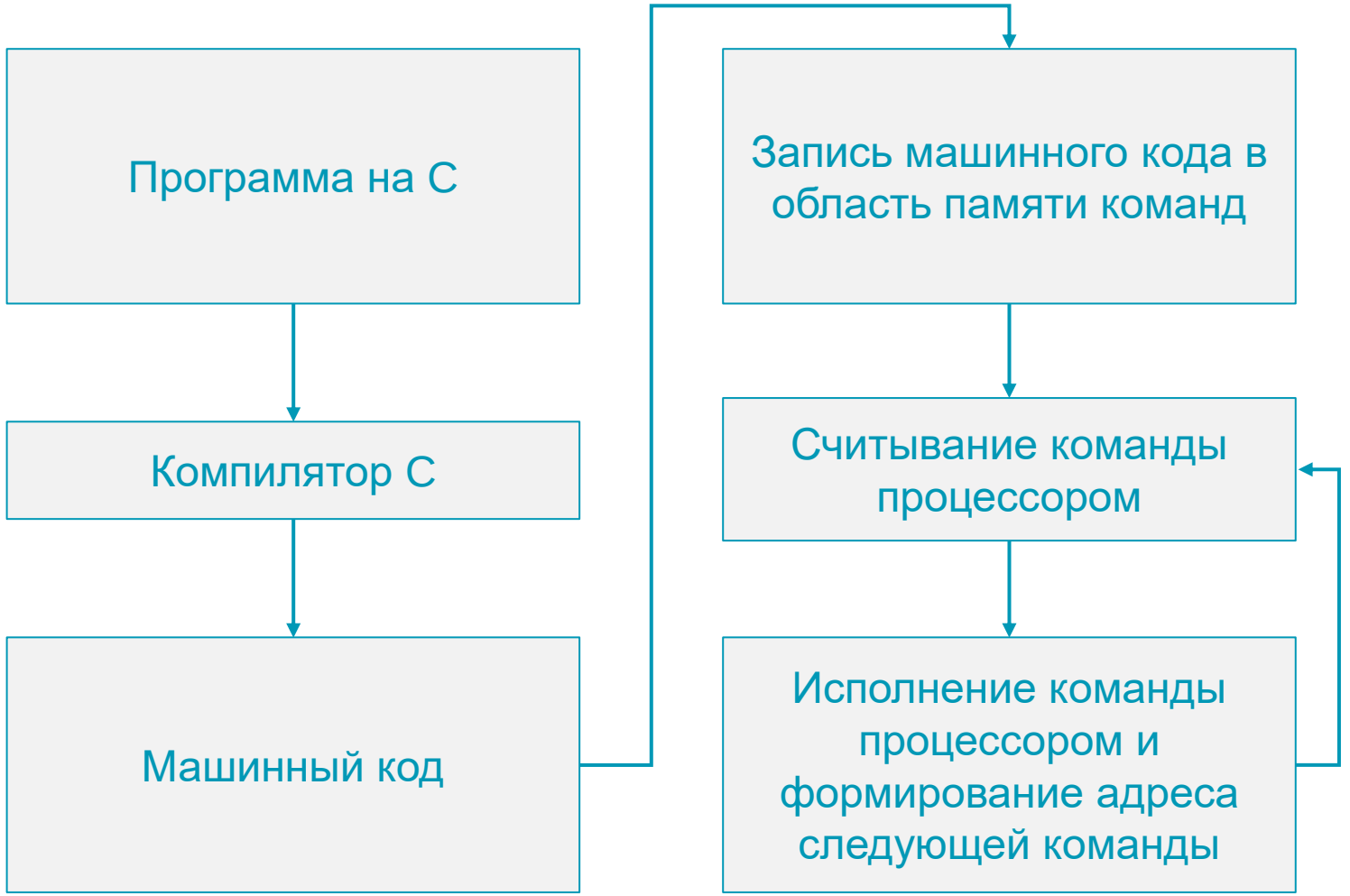


Ассемблер

От программы к исполнению



C
От программы к исполнению



AQUARIUS

Архитектура и типы команд

Архитектура набора команд

- Принято разделение процессоров в зависимости от архитектуры набора команд (Instruction Set Architecture, ISA), которую они реализуют. Примеры ISA:
  - ARM, Intel x86, MIPS, RISC-V, IBM Power и т.д.
- Процессоры одной архитектуры могут выполнять одинаковые программы.
- Язык ассемблера (англ. Assembly Language) – представление команд процессора в виде, доступном для чтения человеком. Программы, написанные на языке ассемблера, однозначным образом переводятся в инструкции конкретного процессора.

Регистры RISC-V

- 32 регистра для основного набора команд
- Каждый регистр имеет размер 32 бита = слово (word)
- x0 всегда равен 0
- 32 регистра для вещественных операций в расширении “F”
- В версии RV64 регистры имеют размер 64 бита (double word)

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Команды RISC-V

- Каждая команда имеет код операции (opcode) и операнды

```
add x1, x2, x3          # x1 = x2 + x3
                        # add – код операции сложение
                        # x1 – регистр результата
                        # x2, x3 – регистры-операнды
                        # # - используется для комментариев
```

- Регистр результата условно обозначают rd (destination register), регистры с операндами – rs1, rs2 (source register):

```
add rd, rs1, rs2
```

- Эквивалент в языке C:

```
a = b + c
```

Арифметические команды

- **add rd, rs1, rs2**

  сложение rd = rs1 + rs2
- **sub rd, rs1, rs2**

  вычитание rd = rs1 – rs2
- **and rd, rs1, rs2**

  побитовое И rd = rs1 & rs2
- **or rd, rs1, rs2**

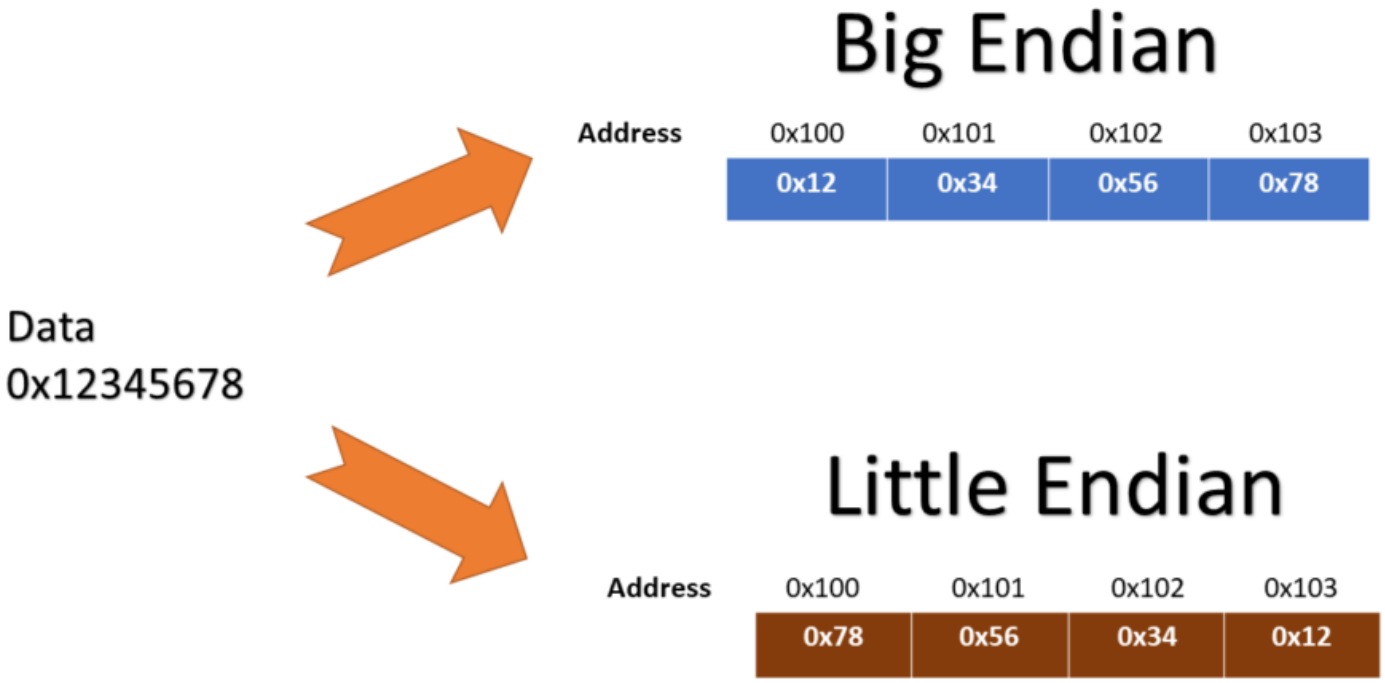
  побитовое ИЛИ rd = rs1 | rs2
- **xor rd, rs1, rs2**

  побитовое исключающее ИЛИ rd = rs1 xor rs2
- **sll rd, rs1, rs2**

  сдвиг влево rd = rs1 << rs2

Обращение к памяти

- 1 байт = 8 бит
- 4 байта = 1 слово (word)
- Адреса в памяти – адреса в байтах
- В RISC-V байты в словах расположены в соответствии с **little endian**. Т.е. младшие байты помещаются в меньший адрес.



Команды обращения к памяти

lw rd, addr

Считывание слова в регистр rd из памяти по адресу addr

sw rd, addr

Запись слова из регистра rd в память по адресу addr

Также доступны обращения меньшими размерами:

- Halfword – 2 байта: **lh**, **sh**
- Byte – 1 байт **lb**, **sb**

Адрес **addr** может быть указан несколькими способами, самый простой:

addr = offset(r1)

r1 – регистр, содержащий адрес обращения

offset – дополнительное смещение, указанное в виде числа

lw x2, 4(x3) #считывание слова из адреса = x3 + 4

Числовые константы

В коде ассемблера в качестве операндов можно использовать числа или непосредственные значения (immediate operand) или константы

Для использования констант в архитектуре предусмотрены специальные команды

addi rd, rs1, imm

сложение rd = rs1 + imm

Пример:

addi x2, x3, -4

x2 = x3 - 4

Запись константы в регистр

li rd, imm

rd = imm

- Числа по умолчанию в десятичной системе и со знаком
- Для использований шестнадцатиричных чисел нужно добавить “0x”

  например 0x10 = 16

Команды ветвления

- Ветвления подразумевают, что в зависимости от результатов некоторых вычислений нужно выполнять разные действия
- В языках программирования используется оператор if
- Аналог оператора if в ассемблере – операции условного перехода (branch)

beq rs1, rs2, label # branch if equal
  - если rs1 == rs2, сделать переход на участок кода, помеченный label, иначе выполнить следующую команду

bne rs1, rs2, label # branch if not equal
  - если rs1 != rs2, сделать переход на участок кода, помеченный label, иначе выполнить следующую команду

j label
  - безусловный переход (jump)

Где посмотреть кодировку команды?

- Файл RISCV_Green_Card.pdf (“RISC-V Reference”)
- RISC-V ISA (riscv-spec.pdf) chapter 25 Instruction Set Listings

Псевдоинструкции

Команды ассемблера, которые упрощают читаемость, но при сборке в двоичный код заменяются на другие

- nop ➡ addi x0,x0, 0
  # no operation
- mv rd, rs ➡ addi rd,rs, 0
  # copy register
- beqz rs, offset ➡ beq rs, x0, offset
  # branch if = zero

AQUARIUS

Выполнение лабораторной работы

Выполнение лабораторных заданий, не касающихся исполнения подпрограмм

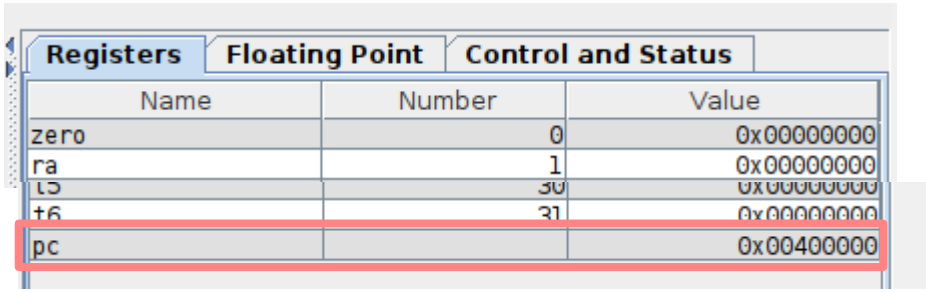
Лабораторная работа

1. Перейдите в директорию ~/pratise/basic-graphics-music/labs/30_schoolriscv
2. Запустите скрипт bash 10_run_instruction_set_simulator.bash
3. Откройте файл program.s: file -> open -> program.s
4. Запустите симулятор: run -> Assemble
5. Обратите внимание на превращение псевдо-инструкций в обычные

Basic	Source
add x10,x0,x0	9: mv a0, zero
addi x10,x10,4	10: addi a0, a0, 4
add x11,x0,x0	11: mv a1, zero
addi x11,x11,3	12: addi a1, a1, 3
add x6,x0,x0	13: mv t1, zero
add x7,x0,x0	14: mv t2, zero
add x6,x6,x10	16: mul: add t1, t1, a0
addi x7,x7,1	17: addi t2, t2, 1
bne x7,x11,0xfffffff8	18: bne t2, a1 mul

6. Для пошаговой отладки программы используйте клавишу

7. Обратите внимание, как изменяется Program Counter (PC) справа на вкладке registers.



Лабораторная работа

Самостоятельно

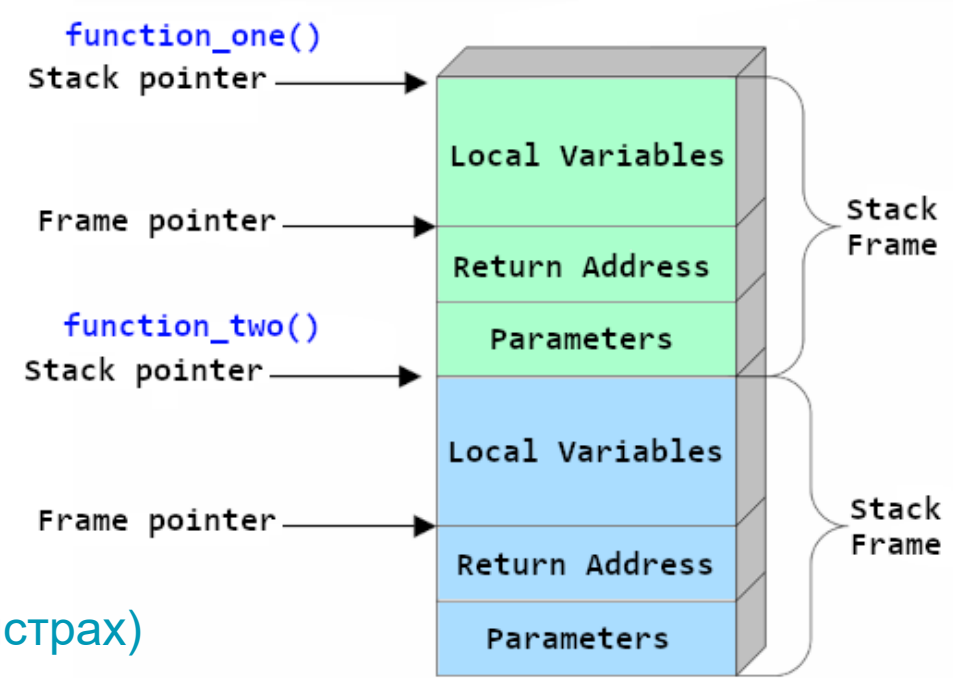
Реализуйте умножение с помощью цикла и команд сложения

AQUARIUS

Выполнение функций
Stack

Стэк вызова

Стэк вызова (call stack) – структура данных, хранящая информацию для возврата управления из подпрограмм (процедур, функций) в программу (или подпрограмму, при вложенных или рекурсивных вызовах) и/или для возврата в программу из обработчика прерываний.

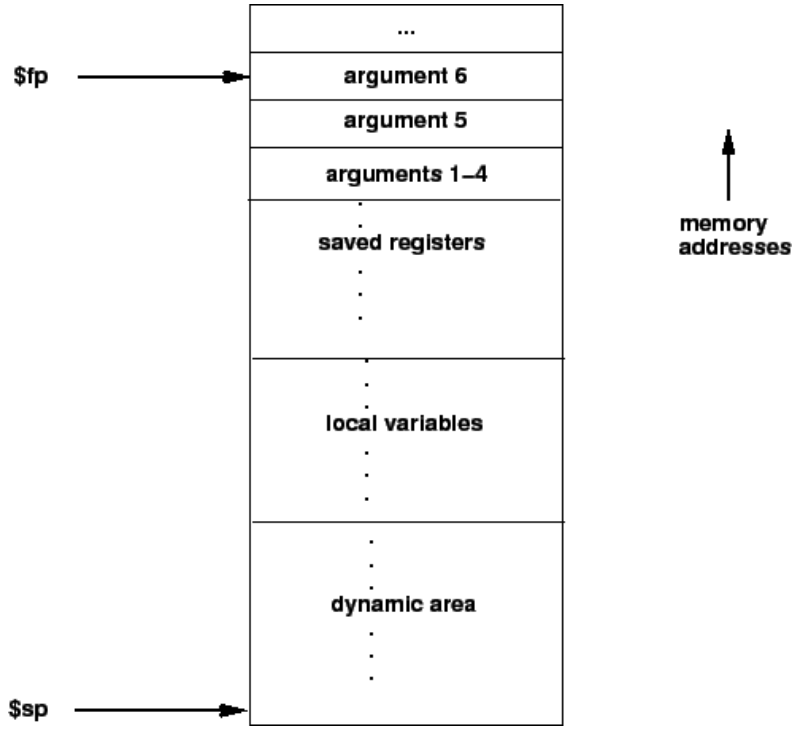


Стэк обычно хранится в памяти и в нем сохраняются:

- аргументы вызванной функции (если не помещаются в регистрах)
- адрес возврата или другие указатели
- локальные переменные самой функции (если не помещаются в регистрах)
- значение регистров вызывающей функции

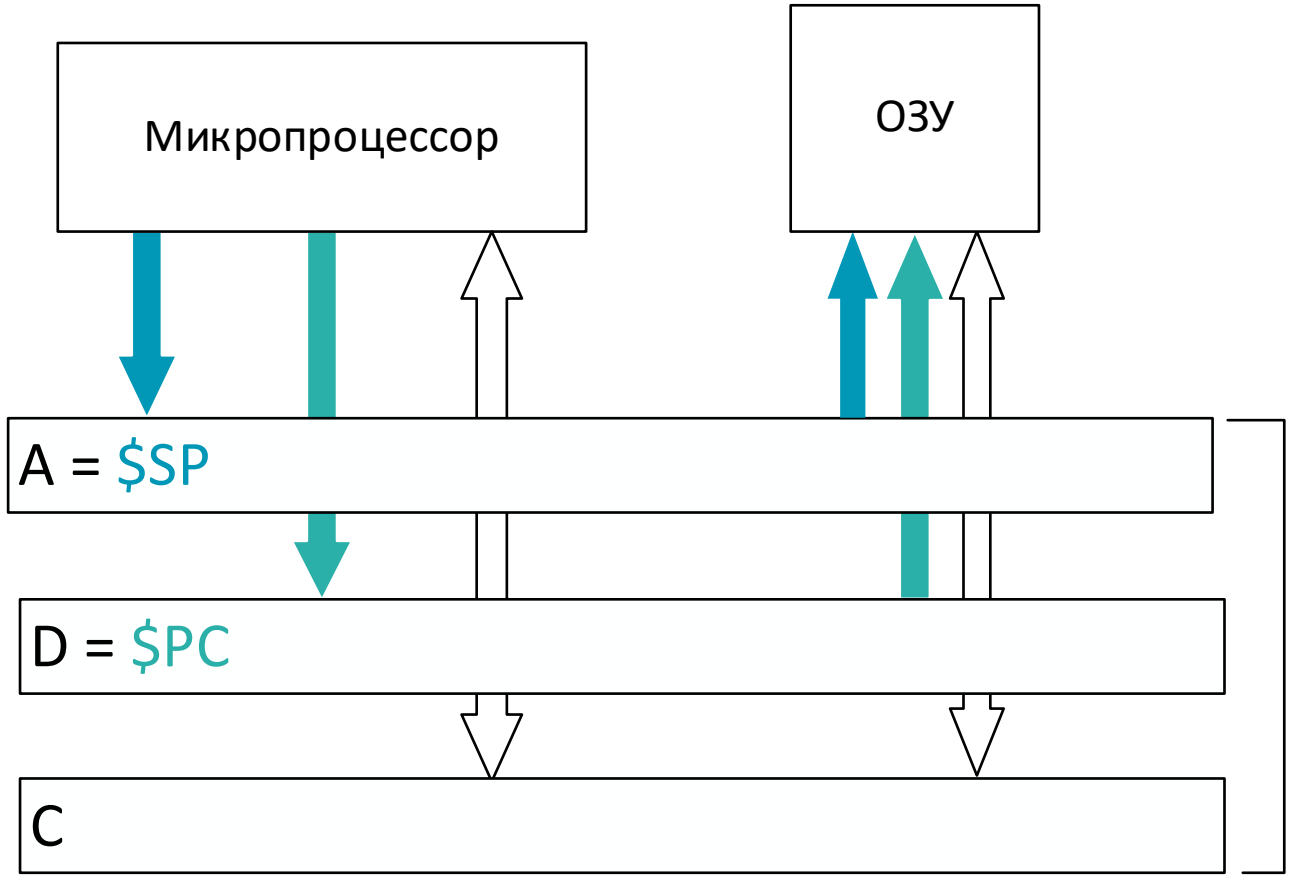
```
int function _two(int a, …)
{
    function_one(b, c, …);
}
```

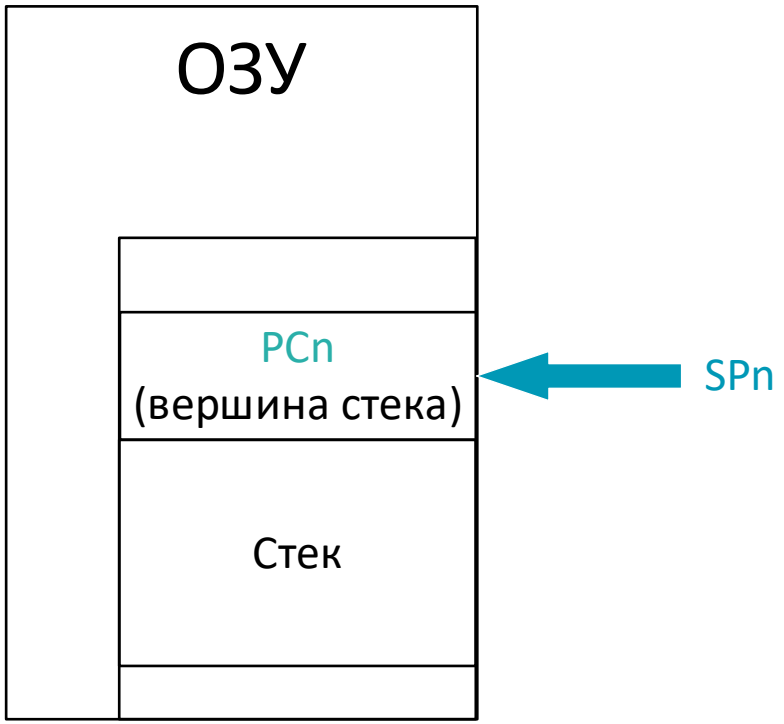
Регистры стэка вызова



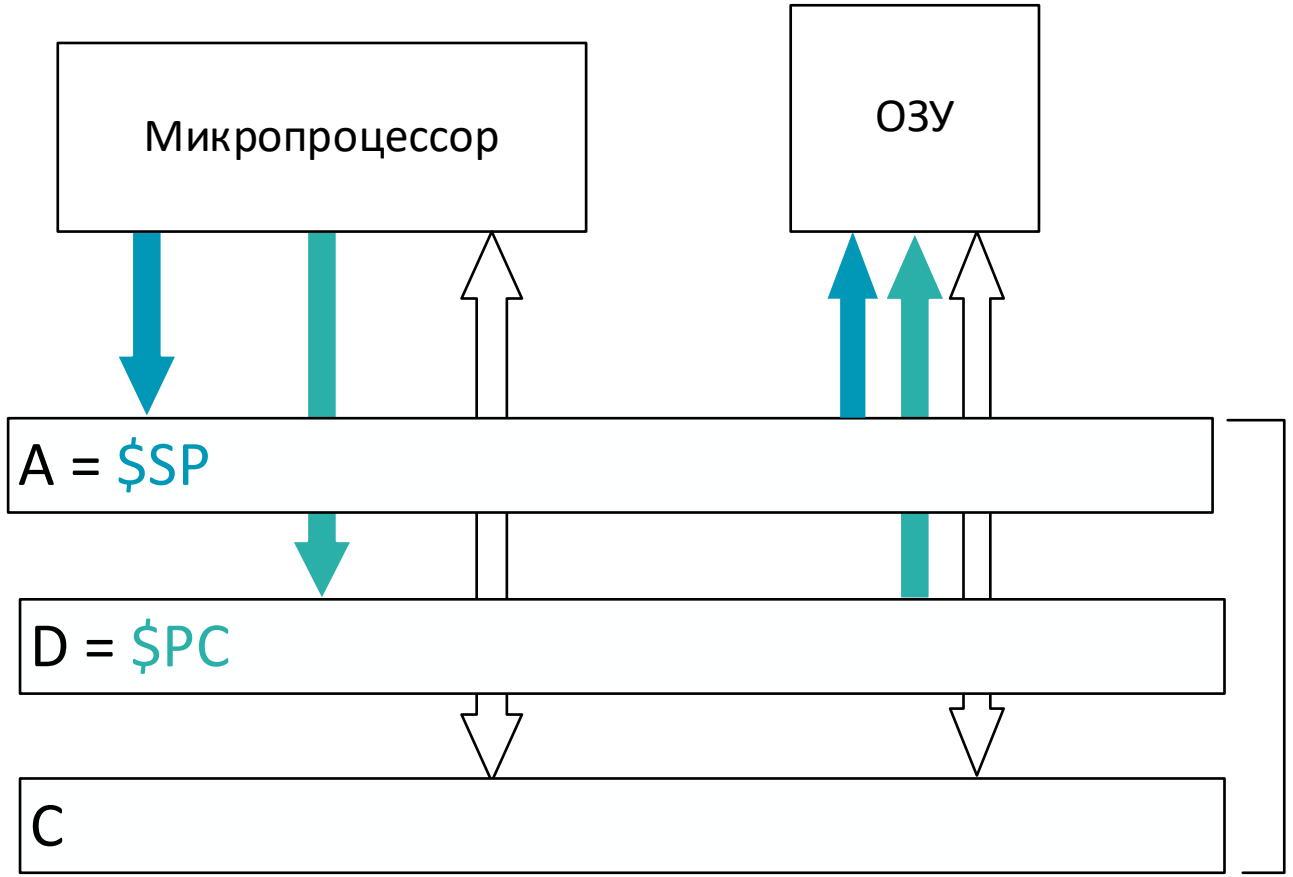
Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

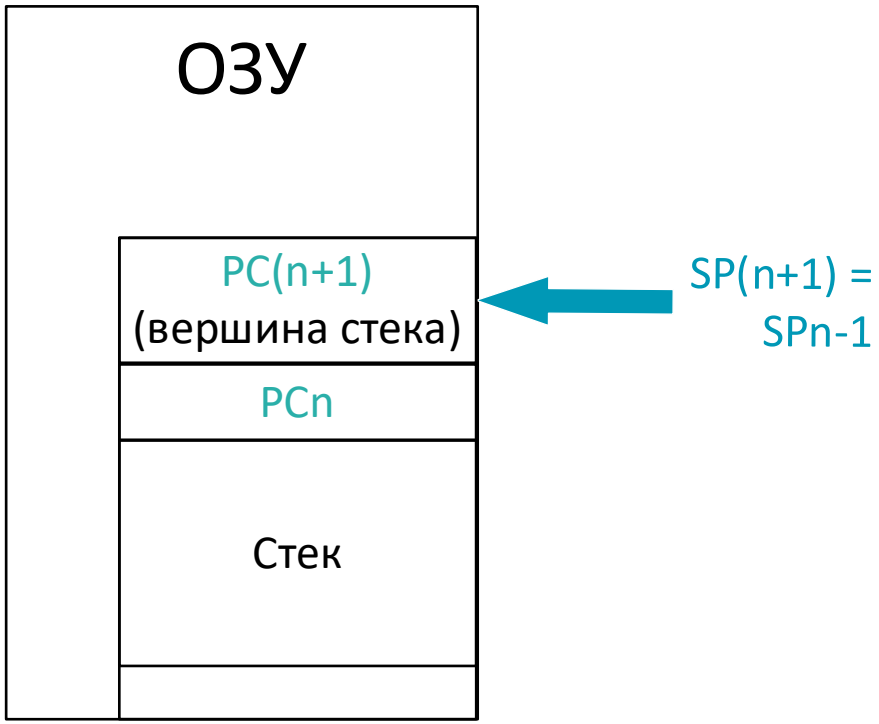
Вызов CALL функции



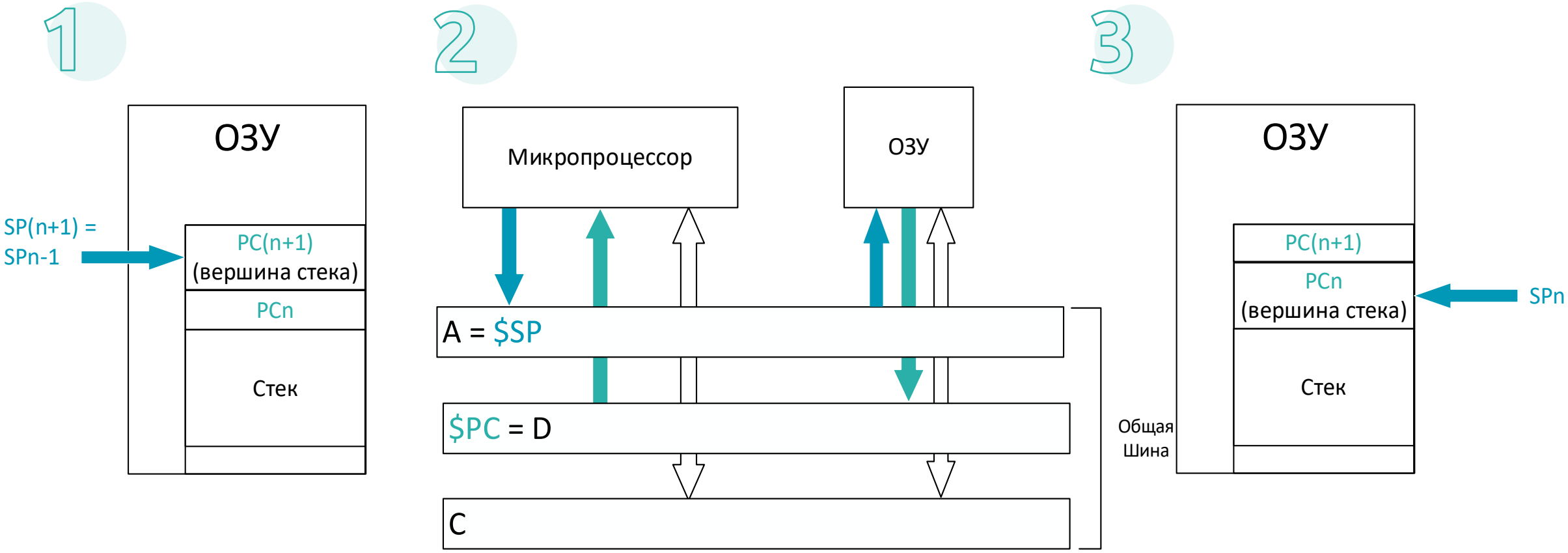


Повторный вызов CALL функции





Вызов Return функции



Команды JAL, JALR, RET

Команды JAL, JALR выполняют безусловный переход с сохранением текущего PC для возможности возврата в тот же участок программы

- **jal rd, label** # jump and link

  сделать переход на другой участок кода, помеченным label (immediate address), и записать (pc + 4) в регистр rd
- **jalr rd, offset(rs1)** # jump and link register

  сделать переход на участок кода по адресу (rs1 + offset), и записать (pc + 4) в регистр rd
- **ret jalr x0, x1, 0** # return from subroutine