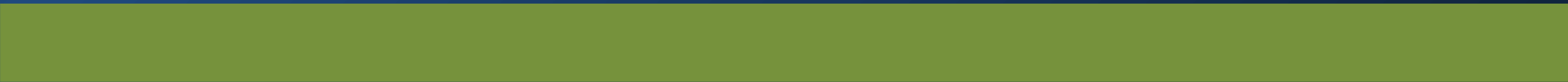


Архитектура вычислительных систем

Семинары №5, 6

Стек, подпрограммы и конвенции
относительно использования регистров



План семинарского занятия

Цель и задачи

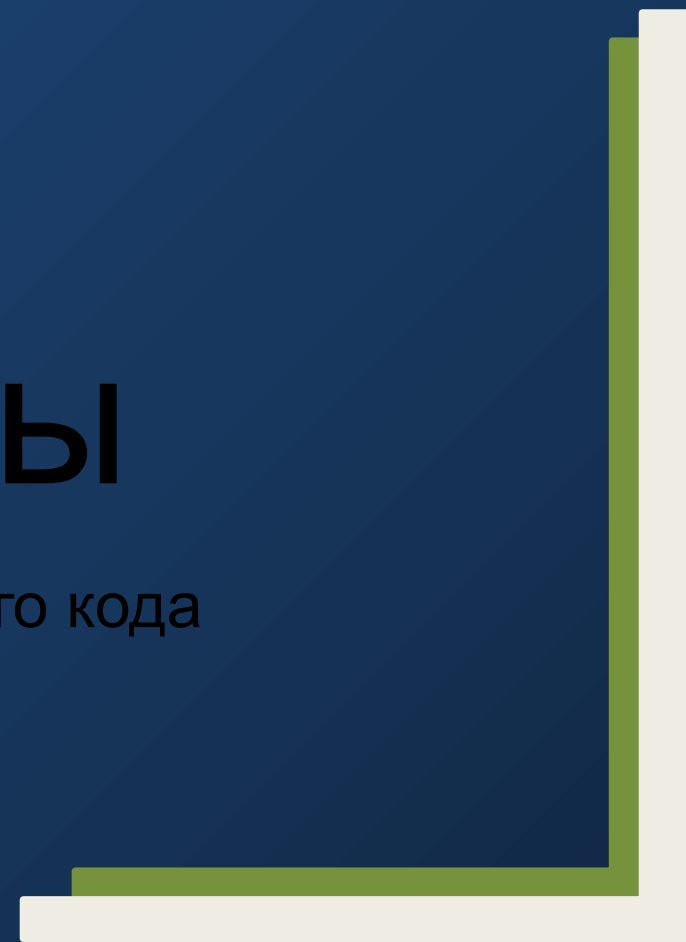
изучение использования подпрограмм на уровне системы команд

Основные вопросы

1. Особенности вызова подпрограмм в RISC-V и возврата из подпрограмм.
2. Использование подпрограмм с параметрами и без.
3. Соглашения о передаче фактических параметров и возврате результатов.
4. Использование стека для локальных переменных подпрограммы и дополнительных фактических параметров.
5. Разработка рекурсивных подпрограмм.
6. Выдача ИДЗ № 1. Установка сроков и регламента сдачи.

Подпрограммы

Задача повторного использования исходного кода



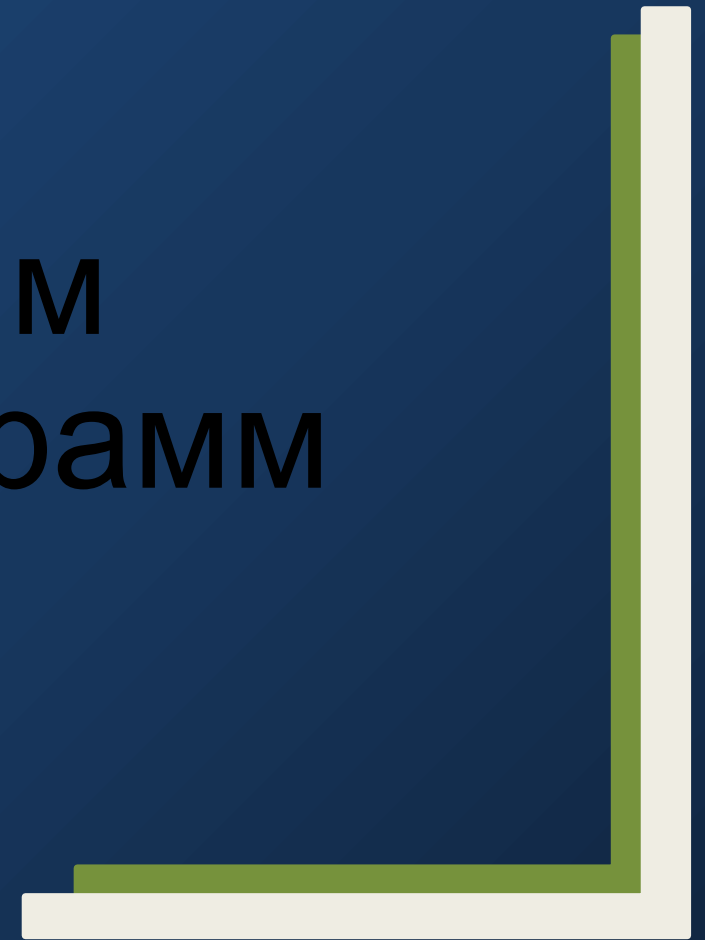
Подпрограмма

Часть программного кода, оформленная таким образом, что

- возможно выполнение этого участка кода более, чем один раз
- переход на этот участок кода возможен из произвольных мест кода
- после выполнения подпрограммы происходит переход «обратно»

Вызов подпрограмм и возврат из подпрограмм

Особенности реализации в RISC-V



Аппаратное решение

Вызов подпрограмм

Решение: **псевдокоманда** записи адреса возврата и перехода:

jal адрес

1. Адрес следующей ячейки записывается в **ra (x1)**
2. Происходит переход на *адрес*

Возврат из подпрограммы

псевдокоманда перехода на адрес, находящийся в регистре **ra**:

ret

Рассмотрим пример

```
1  .data
2  ping:  .asciz  "Ping\n"
3  .text
4          jal      subr
5          la        s1 subr
6          jalr      s1
7          li        a7 10
8          ecall
9  subr:   la        a0 ping
10         li        a7 4
11         ecall
12         ret
```

```

1  .data
2  ping:  .asciz  "Ping\n"
3  .text
4          jal      subr
5          la        s1 subr
6          jalr      s1
7          li        a7 10
8          ecall
9  subr:   la        a0 ping
10         li        a7 4
11         ecall
12         ret

```

Переход и связывание (Jump and Link)

JAL rd, offset $\# \text{rd} \leftarrow \text{PC} + 4, \text{PC} \leftarrow \text{PC} + \text{offset}$

Регистр перехода и связывания (Jump and Link Register)

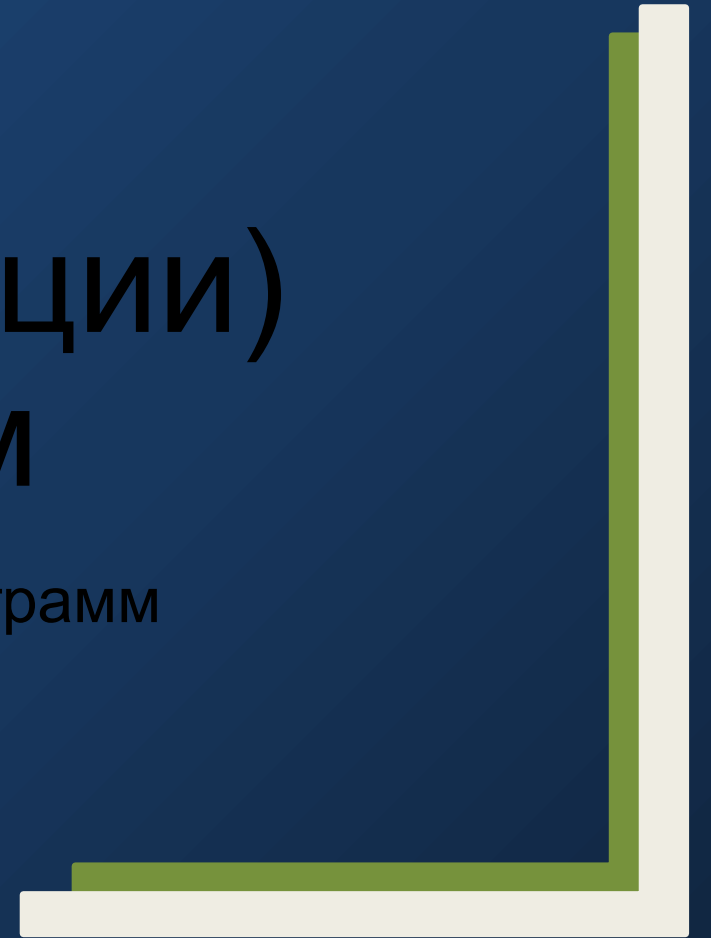
JALR rd, offset(rs1) $\# \text{rd} \leftarrow \text{PC} + 4, \text{PC} \leftarrow \text{rs1} + \text{offset}$

ret — это... jalr

Address	Code	Basic	
0x00400000	0x018000ef	jal x1,0x00000018	4: jal subr
0x00400004	0x00000497	auipc x9,0	5: la s1 subr
0x00400008	0x01448493	addi x9,x9,20	
0x0040000c	0x000480e7	jalr x1,x9,0	6: jalr s1
0x00400010	0x00a00893	addi x17,x0,10	7: li a7 10
0x00400014	0x00000073	ecall	8: ecall
0x00400018	0x0fc10517	auipc x10,0x0000fc10	9: subr: la a0 ping
0x0040001c	0xfe850513	addi x10,x10,0xffffffffe8	
0x00400020	0x00400893	addi x17,x0,4	10: li a7 4
0x00400024	0x00000073	ecall	11: ecall
0x00400028	0x00008067	jalr x0,x1,0	12: ret

Соглашения (конвенции) для подпрограмм

Простая конвенция для конечных подпрограмм



Необходимость соглашений

Решённые задачи:

атомарного вызова
произвольного вызова и возврата

Нерешённые задачи

1. «Прозрачности» повторного использования
 - сохранение регистров
 - передача параметров
 - возвращение значения
 - локальности меток (переменных/адресов перехода):
2. Вложенного вызова. В т. ч. рекурсивного вызова.

Подпрограмма (другой взгляд)

- **обычный код**, находящийся в памяти там, куда его поместил транслятор. **Выполняться** обязан **по инструкциям jal или jalr**.

1. Прозрачность требует
 - a) отдельной конвенции о протоколе передачи параметров*
 - b) механизма сокрытия локальных меток*
2. Проблема вложенного вызова возникает, когда подпрограмма вызывается из другой подпрограммы:
 - a) текущий адрес возврата надо сохранять перед вложенным вызовом и восстанавливать перед возвратом;*
 - b) конвенция должна предусматривать цепочку вложенных вызовов*
3. Проблема рекурсивного вызова возникает, когда в цепочке вызовов некоторая подпрограмма встречается более одного раза
4. Локальные данные могут быть изменены во вложенном вызове, поэтому их надо где-то динамически заводить/сохранять/освобождать

Простая конвенция для концевых подпрограмм

1. Подпрограмма вызывается с помощью инструкций `jal` / `jalr`
2. Подпрограмма не будет вызывать другую подпрограмму
3. Подпрограмма возвращает управление вызывающей программе с помощью инструкции `ret`
4. Регистры используются следующим образом:
 - ***t0 - t6:*** подпрограмма может изменить эти регистры.
 - ***s0 - s11:*** при выходе из подпрограммы эти регистры должны содержать те же данные, которые были в них при входе
 - ***a0 - a7:*** эти регистры содержат параметры для подпрограммы. Подпрограмма может изменить их.
 - ***a0, a1:*** эти регистры содержат значения, возвращаемые из подпрограммы

```

1 .data
2 yes: .asciz "It is a triangle\n"
3 no: .asciz "It is not a triangle\n"
4 # Основная программа
5 .text
6     jal    input
7     mv     s1 a0
8     jal    input
9     mv     s2 a0
10    jal    input
11    mv     s3 a0
12    jal    check
13    la     a0 no
14    li     a7 4
15    ecall
16    li     a7 10
17    ecall
18

```

проверка неравенства треугольника
запись адреса no в регистр a0 (занимает две инструкции!!!)
вывод строки в a0 (сюда возврат в случае успеха: ra+8)

```

19 .data
20 prompt: .ascii "Enter triangle side: "
21 .text
22 # Подпрограмма ввода одной стороны треугольника
23 input:  la     a0 prompt
24         li     a7 4
25         ecall
26         li     a7 5
27         ecall
28         ret
29
30 # Подпрограмма проверки отрезков на треугольник
31 check:  add     t3 s1 s2
32         add     t1 s2 s3
33         add     t2 s1 s3
34         ble     t1 s1 notri
35         ble     t2 s2 notri
36         ble     t3 s3 notri
37         la     a0 yes
38         jalr    zero ra 8 # "Суровый" (зависимый от контекста) возврат
39 notri:  ret

```

```

1  .data
2  yes:      .asciz  "It is a triangle\n"
3  no:       .asciz  "It is not a triangle\n"
4  # Основная программа
5  .text
6          jal      input
7          mv       s1 a0
8          jal      input
9          mv       s2 a0
10         jal      input
11         mv       s3 a0
12         jal      check          # проверка неравенства треугольника
13         la       a0 no          # запись адреса no в регистр a0 (занимает две инструкции!!!)
14         li       a7 4           # вывод строки в a0 (сюда возврат в случае успеха: ra+8)
15         ecall
16         li       a7 10
17         ecall
18

```

```

19 .data
20 prompt: .ascii "Enter triangle side: "
21 .text
22 # Подпрограмма ввода одной стороны треугольника
23 input:  la      a0 prompt
24         li      a7 4
25         ecall
26         li      a7 5
27         ecall
28         ret
29
30 # Подпрограмма проверки отрезков на треугольник
31 check:  add      t3 s1 s2
32         add      t1 s2 s3
33         add      t2 s1 s3
34         ble      t1 s1 notri
35         ble      t2 s2 notri
36         ble      t3 s3 notri
37         la      a0 yes
38         jalr     zero ra 8 # "Суровый" (зависимый от контекста) возврат
39 notri:  ret

```

Стек

Реализация стека в машинных кодах



Абстракция «стек»

Для динамического хранения локальных переменных и адресов возврата нам нужен стек.

1. Хранилище «объектов»
2. Основные операции — положить на стек, снять со стека
3. Основной доступ — последовательный, к верхнему элементу стека
4. «Первым вошёл — последним вышел»
5. Может расти «бесконечно»
6. Поведение при исчерпании (снятии с пустого стека) не определено

Реализация стека в машинных кодах

- «Объект» — обычно ячейка
- **Хранилище** — область памяти, ограниченная только с одной стороны («дно» стека), а с другой — «бесконечно» растущая (пока не достигнет занятой области памяти)
- **Рост / снятие:** специальная ячейка памяти, хранящая адрес вершины стека + знание адреса дна сетка
 - *Используется косвенная адресация*
 - *⇒ Удобно хранить в регистре*
 - *Переполнение*
 - *Исчерпание*

Возможная аппаратная поддержка стека

- Отдельный небольшой стек на регистровой памяти
- Атомарные операции добавления/снятия
 - *например, автоувеличение / автоуменьшение указателя прямо в процессе взятия значения (в случае RISC-V — недопустимая двойная арифметическая операция в одной инструкции: вычисление смещения и сложение/вычитание)*
- Двойная косвенная адресация позволяет напрямую обращаться к данным, адреса которых лежат в стеке (в случае RISC-V — недопустимое «тяжёлое» двойное обращение к памяти)

Реализация стека в RARS

Регулируется соотв. конвенцией:

- выделенный регистр `sp` (x2)
- дно стека — `0x7ffffffc` (непосредственно под областью ядра)
- начальное значение `0x7ffeffc` отделено от дна буфером
- стек растёт вниз по одному слову (4 байта)
- предел стека — область кучи
- операции добавления и снятия — неатомарные:
 - *при добавлении сначала уменьшается указатель, затем записывается значение*
 - *при снятии сначала считывается значение, затем увеличивается указатель*

Пример

```
1  # Пример использования стека в RARS
2
3  li t1 123          # t1 = 123
4  addi sp sp -4      # сместить указатель стека вниз на 1 слово
5  sw t1 (sp)         # положить содержимое t1 на стек
6  addi t2 t1 100     # t2 = t1 + 100
7  addi sp sp -4      # сместить указатель стека вниз еще на 1 слово
8  sw t2 (sp)         # положить содержимое t2 на стек
9  lw t3 (sp)         # загрузить содержимое с вершины стека
10 lw t4 4(sp)        # загрузить содержимое первого загруженного значения
11 addi sp sp 4       # "убрать" со стека элемент (pop)
12 lw t0 (sp)         # загрузить содержимое с вершины стека
13 addi sp sp -4      # сместить указатель стека вниз на 1 слово
14 sw zero (sp)       # положить на стек значение = 0
```

```

1  # Пример использования стека в RARS
2
3  li t1 123      # t1 = 123
4  addi sp sp -4  # сместить указатель стека вниз на 1 слово
5  sw t1 (sp)     # положить содержимое t1 на стек
6  addi t2 t1 100  # t2 = t1 + 100
7  addi sp sp -4  # сместить указатель стека вниз еще на 1 слово
8  sw t2 (sp)     # положить содержимое t2 на стек
9  lw t3 (sp)     # загрузить содержимое с вершины стека
10 lw t4 4(sp)    # загрузить содержимое первого загруженного значения
11 addi sp sp 4    # "убрать" со стека элемент (pop)
12 lw t0 (sp)     # загрузить содержимое с вершины стека
13 addi sp sp -4  # сместить указатель стека вниз на 1 слово
14 sw zero (sp)   # положить на стек значение = 0

```

Address	Code	Basic	Source
0x00400000	0x07b00313	addi x6,x0,0x0000007b	3: li t1 123 # t1 = 123
0x00400004	0xffc10113	addi x2,x2,0xfffffffffc	4: addi sp sp -4 # сместить указатель стека вниз на 1 слово
0x00400008	0x00612023	sw x6,0(x2)	5: sw t1 (sp) # положить содержимое t1 на стек
0x0040000c	0x06430393	addi x7,x6,0x00000064	6: addi t2 t1 100 # t2 = t1 + 100
0x00400010	0xffc10113	addi x2,x2,0xfffffffffc	7: addi sp sp -4 # сместить указатель стека вниз еще на 1 слово
0x00400014	0x00712023	sw x7,0(x2)	8: sw t2 (sp) # положить содержимое t2 на стек
0x00400018	0x00012e03	lw x28,0(x2)	9: lw t3 (sp) # загрузить содержимое с вершины стека
0x0040001c	0x00412e83	lw x29,4(x2)	10: lw t4 4(sp) # загрузить содержимое первого загруженного значения
0x00400020	0x00410113	addi x2,x2,4	11: addi sp sp 4 # "убрать" со стека элемент (pop)
0x00400024	0x00012283	lw x5,0(x2)	12: lw t0 (sp) # загрузить содержимое с вершины стека
0x00400028	0xffc10113	addi x2,x2,0xfffffffffc	13: addi sp sp -4 # сместить указатель стека вниз на 1 слово
0x0040002c	0x00012023	sw x0,0(x2)	14: sw zero (sp) # положить на стек значение = 0

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7ffffefe0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x0000007b	0x00000000

Хранение данных в стеке

- Несколько более эффективно, чем в произвольном месте памяти (lw/sw не превращаются в псевдоинструкции)
- Оптимизированные версии процессора могут учитывать конвенцию по вызову и что-то делать с памятью
- Использует адресацию относительно меняющегося sp. Как следствие, требует аккуратного подсчёта текущей глубины стека
- Не требует явного указания адреса и заведения метки в программе на языке ассемблера
- Может привести к сбоям в работе при переполнении/исчерпании/неаккуратном использовании стека

Универсальные подпрограммы

Простая конвенция не поддерживает вложенного вызова подпрограмм:

- *⇒ надо сохранять **ra** (при повторном вызове он изменится)*
- *неправильное решение: выделить для каждой функции ячейку, в которую сохранять **ra** (не работает рекурсивный вызов)*

Рекурсивный вызов — сохранение в стеке

Динамически выделять память удобнее всего на стеке.

В начале подпрограммы все регистры, значения которых следует сохранить до выхода из подпрограммы, а также регистр возврата `ra` записываются в стек операцией `push`.

Перед выходом эти значения снимаются со стека операцией `pop`. Эти значения, как правило, не используются внутри подпрограммы, важна только последовательность сохранения и восстановления.

В самом простом случае сохранять надо только `ra`

```

1  # Рекурсивное вычисление факториала
2  # Вызывающая программа
3      li      a7 5
4      ecall
5      jal     fact      # Аргумент для вычислений
6      li      a7 1      # Параметр уже в a0 )
7      ecall      # Вывод результата
8      li      a7 10     # Параметр уже в a0 ))
9      ecall
10 # Подпрограмма вычисления факториала
11 fact: addi   sp sp -8   ## Запасаем две ячейки в стеке
12      sw     ra 4(sp)    ## Сохраняем ra
13      sw     s1 (sp)     ## Сохраняем s1
14
15      mv     s1 a0       # Запоминаем N в s1
16      addi   a0 s1 -1    # Формируем n-1 в a0
17      li     t0 1
18      ble    a0 t0 done  # Если n<2, готово
19      jal     fact      # посчитаем (n-1)!
20      mul    s1 s1 a0    # s1 пережил вызов
21      # Возврат из подпрограммы
22 done: mv     a0 s1      # Возвращаемое значение
23      lw     s1 (sp)     ## Восстанавливаем sp
24      lw     ra 4(sp)    ## Восстанавливаем ra
25      addi   sp sp 8     ## Восстанавливаем вершину стека
26      ret

```

Пролог и эпилог

— начальная и завершающая части подпрограммы, которые обслуживают соблюдение конвенции, а к решаемой задаче имеют только косвенное отношение.

Так, пролог в примере сохраняет на стеке два регистра, а использовалось бы их больше — сохранял бы больше; эпилог эти два регистра (`s0` и `ra`) восстанавливает

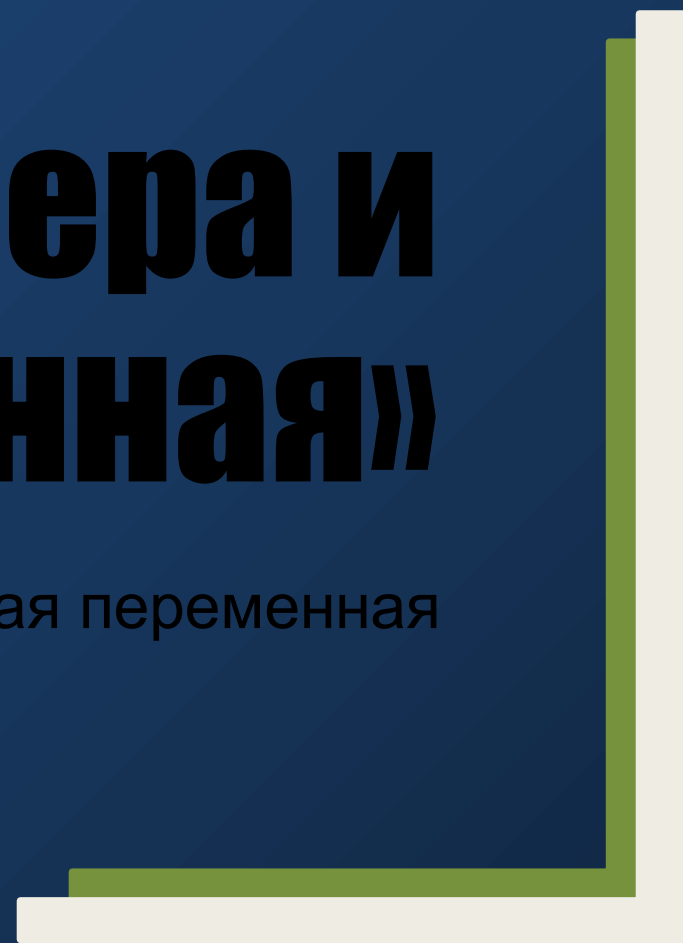
```

1  # Рекурсивное вычисление факториала без использования
2  # регистров  $s^*$ , сохраняемых на стеке
3      li      a7 5
4      ecall
5      jal     fact          # Параметр уже в a0 )
6      li      a7 1
7      ecall          # Параметр уже в a0 ))
8      li      a7 10
9      ecall
10
11 fact:  addi    sp sp -4
12        sw     ra (sp)      # Сохраняем ra
13        mv     t1 a0        # Запоминаем N в t1
14        addi   a0 t1 -1     # Формируем n-1 в a0
15        li     t0 1
16        ble    a0 t0 done   # Если n<2, готово
17        addi   sp sp -4     ## Сохраняем t1 на стеке
18        sw     t1 (sp)      ##
19        jal     fact        # Посчитаем (n-1)!
20        lw     t1 (sp)      ## Вспоминаем t1
21        addi   sp sp 4      ##
22        mul    t1 t1 a0     # Домножаем на (n-1)!
23 done:  mv     a0 t1        # Возвращаемое значение
24        lw     ra (sp)      # Восстанавливаем ra
25        addi   sp sp 4      # Восстанавливаем вершину стека
26        ret

```

Язык ассемблера и понятие «переменная»

понятие локальная переменная



Язык ассемблера и понятие «переменная»

В процессе вычислений на языке ассемблера постоянно происходит так, что **значение**, соответствующее некоторому объекту программы (*например, текущее посчитанное значение факториала в примерах*), в разное время **представлено различными аппаратными средствами.**

Такое отношение к данным резко отличается от более высокоуровневого понятия «переменная», в котором предполагается, что представление объекта и способ работы с ним всегда одинаковы.

Можно сказать, что **в языке ассемблера нет переменных, а есть только метки**, и это не одно и то же.

Локальная переменная

- Предположим, что в нашей подпрограмме используется так много объектов, что для всех них не хватает регистров, или мы по каким-то другим причинам хотим хранить некоторое значение не в регистре, а в памяти
- Понятно, что это значение надо хранить на стеке. И для того, чтобы не путаться, куда в данный момент указывает `sp`, выделение и инициализацию такой памяти на стеке удобно совмещать с прологом, а освобождение — с эпилогом.

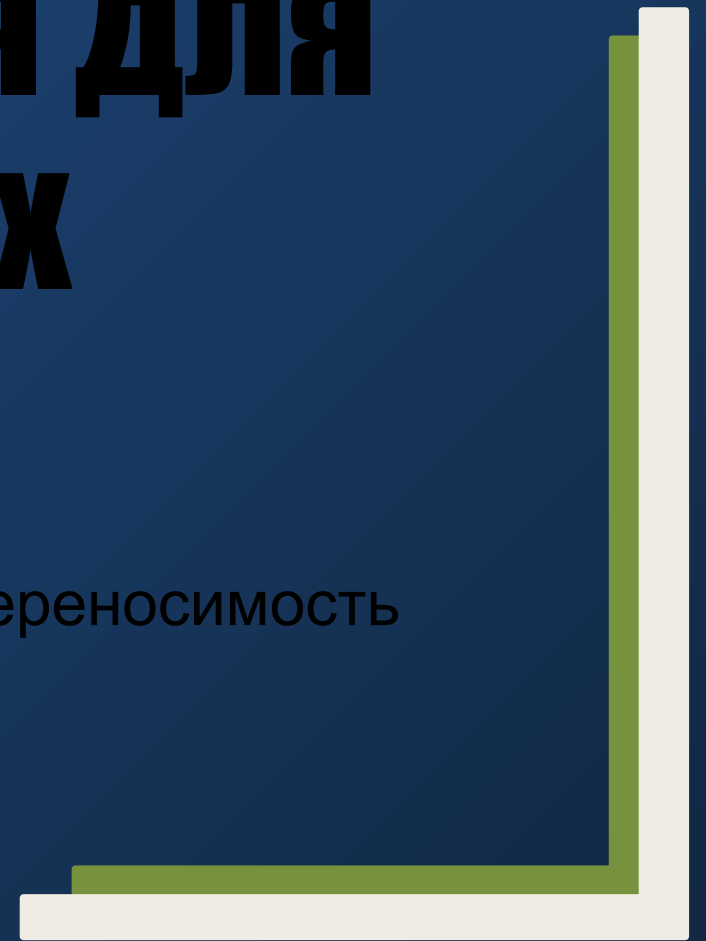
```
1  subr:  addi    sp sp -12      ## Выделяем три ячейки – под ra и две переменные
2          sw     ra 8(sp)      ## Сохраняем ra
3          sw     zero 4(sp)    ## Инициализируем первую переменную нулём
4          sw     zero (sp)     ## Инициализируем вторую переменную нулём
5          # ...
6          lw     t0 4(sp)      # Загружаем первую локальную переменную в t0
7          # ...
8          sw     t1 (sp)       # Записываем t1 во вторую локальную переменную
9          # ...
10         lw     ra 8(sp)      # Восстанавливаем ra
11         addi    sp sp 12     # Освобождаем две ячейки стека
12         ret
```

Локальная переменная

- Можно использовать сколько угодно локальных переменных
- Переменные рекомендуется инициализировать:
 - *В этом месте стека наверняка лежит какой-нибудь мусор, оставшийся от предыдущих вызовов подпрограмм*
 - *Популярная ошибка — не проинициализировать переменную, а потом решить, что в ней лежит 0*
- Если не изменять стек в процессе работы, у них всегда будут фиксированные смещения относительно вершины стека

Общие соглашения для универсальных подпрограмм

цель соглашений — удобство, эффективность, переносимость



1. Передаваемые подпрограмме значения надо заносить в регистры a^*
2. Вызов подпрограммы должен производиться командой `jal` или `jalr`
3. Никакая вызываемая подпрограмма не модифицирует содержимое стека выше того указателя, который она получила в момент вызова
4. Вызываемая подпрограмма обязана сохранить на стеке значение `ra`
5. Подпрограмма обязана сохранить на стеке все используемые ею регистры s^*
6. Подпрограмма может хранить на стеке произвольное количество переменных. Количество этих переменных и занимаемого ими места на стеке не оговаривается и может меняться в процессе работы подпрограммы.
7. Возвращаемое подпрограммой значение надо заносить в регистры `a0`, `a1`
8. Подпрограмма должна освободить все занятые локальными переменными ячейки стека
9. Подпрограмма обязана восстановить из стека сохранённые значения s^* и `ra`
10. Подпрограмма обязана при возвращении восстановить значение `sp` в исходное. Это случится автоматически при соблюдении всех предыдущих требований конвенции
11. Возврат из подпрограммы должен производиться командой `ret`



Сара Л. Харрис, Дэвид Харрис
Цифровая схемотехника и архитектура компьютера:
RISC-V –
М.: ДМК Пресс, 2021. – 810 с.

UNIH

LecturesCMC/ArchitectureAssembler2022/03_StackSubroutines

*Стек, подпрограммы и конвенции относительно
использования регистров*

[http://uneex.org/LecturesCMC/
ArchitectureAssembler2022/03_StackSubroutines](http://uneex.org/LecturesCMC/ArchitectureAssembler2022/03_StackSubroutines)

Домашнее задание

Оценка до 8 баллов

Разработать программу, определяющую максимальное значение аргумента, при котором результат вычисления факториала размещается в 32-х разрядном машинном слове. Вычисление факториала организовать как подпрограмму с циклом, которая возвращает найденный аргумент в регистре **a0**. Вывод результатов должна осуществлять главная функция.

Опционально до +2 баллов

Дополнительно реализовать решение предыдущей задачи с использованием рекурсивной подпрограммы вычисления максимального значения аргумента, при котором результат вычисления факториала размещается в 32-х разрядном машинном слове.