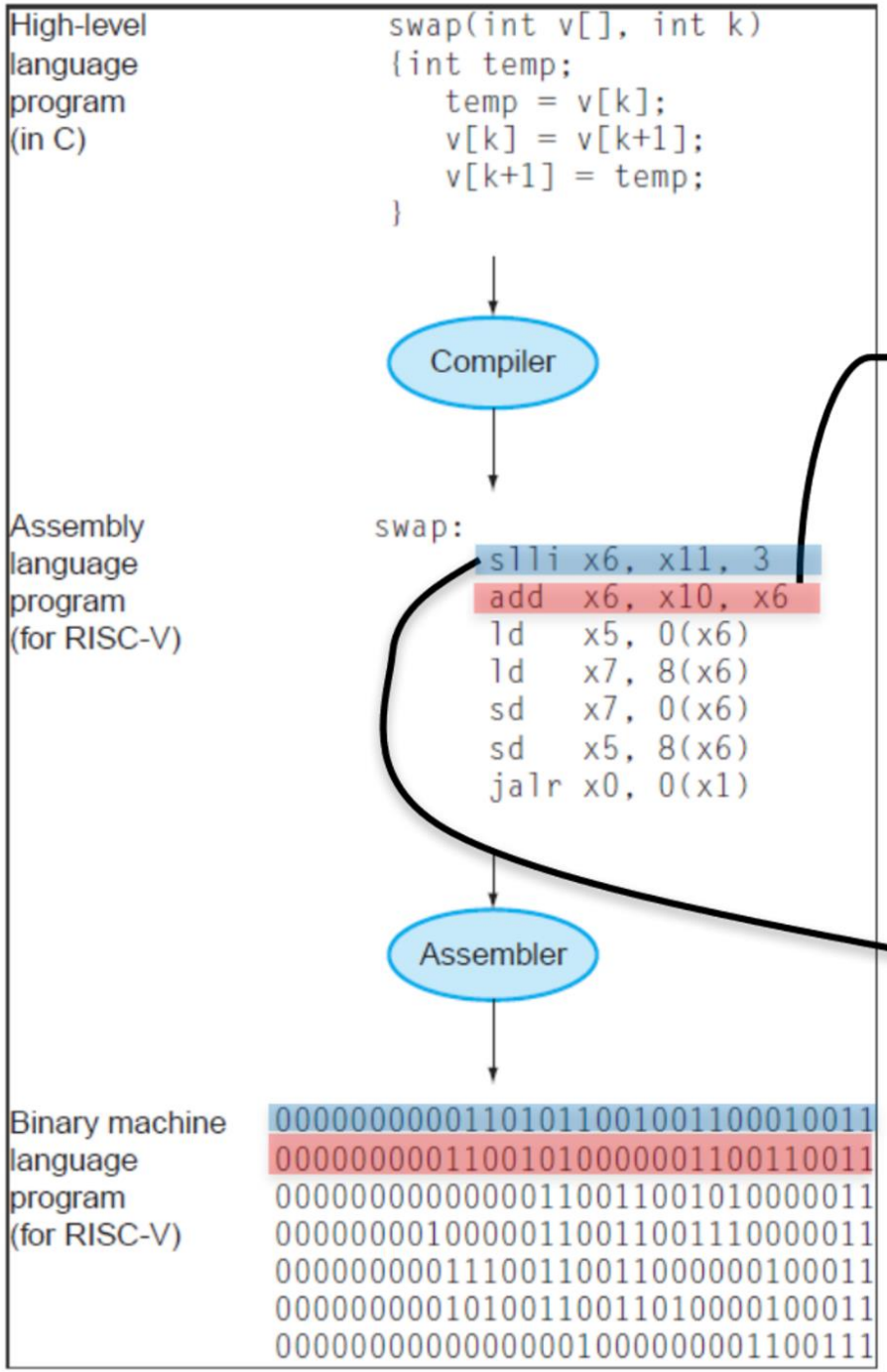
Архитектуры процессорных систем

Лекция 6. Программирование RISC-V

Цикл из 16 лекций о цифровой схемотехнике, способах построения и архитектуре компьютеров

План лекции

- Язык ассемблера RISC-V
- Условные переходы и циклы
- Вызовы подпрограмм
- Карта памяти
- Компиляция программ с языков высокого уровня



- RISC-V инструкция R-типа

funct7	rs2	rs1	funct3	rd	opcode
7 бит	5 бит	5 бит	3 бита	5 бит	7 бит

add x6, x10, x6

0	6	10	0	6	51

0000000	00110	01010	000	00110	0110011

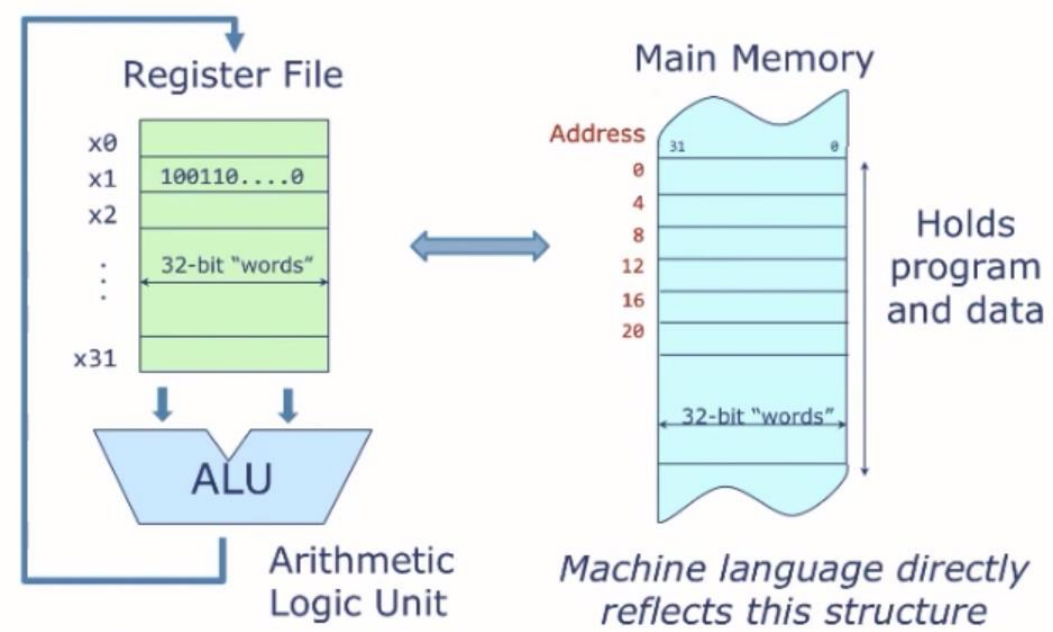
$0000\ 0000\ 0110\ 0101\ 0000\ 0011\ 0011\ 0011_{two} = 00650333_{16}$

- RISC-V инструкция I-типа

immediate	rs1	funct3	rd	opcode
12 бит	5 бит	3 бита	5 бит	7 бит

Инструкция логического сдвига влево:
rs1 – сдвигаемый операнд
immediate – значение сдвига (первые 5 бит)

Модель процессора RISC-V



- Регистровый файл
  - 32 регистра общего назначения
  - Каждый регистр 32 бита
  - x0 = 0
- Память
  - Каждая ячейка памяти имеет ширину 32 бита (1 слово)
  - Память имеет побайтовую адресацию
  - Адреса соседних слов отличаются на 4
  - Адрес 32 бита
  - Может быть адресовано 2^32 байт или 2^30 слов

RISC-V инструкции

- Вычислительные
  - Register-register `op dest, src1, src2`
  - Register-immeliate `op dest, src1, const`
- Загрузки и сохранения
  - `lw dest, offset(base)`
  - `sw src, offset(base)`
- Управления
  - Безусловный переход `jal label` и `lalr register`
  - Условный переход `comp src1, src2, label`
- Псевдоинструкции

Instr	Название	Функция	Описание	Формат	Opcode	Func3	Func7	Пример использования
add	ADDition	Сложение	rd = rs1 + rs2			0x0	0x00	
sub	SUBtraction	Вычитание	rd = rs1 - rs2			0x0	0x20	
xor	eXclusive OR	Исключающее ИЛИ	rd = rs1 ^ rs2			0x4	0x00	
or	OR	Логическое ИЛИ	rd = rs1 | rs2			0x6	0x00	op rd, rs1, rs2
and	AND	Логическое И	rd = rs1 & rs2			0x7	0x00	
sll	Shift Left Logical	Логический сдвиг влево	rd = rs1 << rs2	R	0110011	0x1	0x00	xor x2, x5, x6
srl	Shift Right Logical	Логический сдвиг вправо	rd = rs1 >> rs2			0x5	0x00	sll x7, x11, x12
sra	Shift Right Arithmetic	Арифметический сдвиг вправо	rd = rs1 >>> rs2			0x5	0x20	
slt	Set Less Then	Результат сравнения A < B	rd = (rs1 < rs2) ? 1 : 0			0x2	0x00	
sltu	Set Less Then Unsigned	Беззнаковое сравнение A < B	rd = (rs1 < rs2) ? 1 : 0			0x3	0x00	
addi	ADDition Immediate	Сложение с константой	rd = rs1 + imm			0x0		
xori	eXclusive OR Immediate	Исключающее ИЛИ с константой	rd = rs1 ^ imm			0x4	-	
ori	OR Immediate	Логическое ИЛИ с константой	rd = rs1 | imm			0x6		
andi	AND Immediate	Логическое И с константой	rd = rs1 & imm			0x7		op rd, rs1, imm
slli	Shift Left Logical Immediate	Логический сдвиг влево	rd = rs1 << imm	I	0010011	0x1	0x00	
srli	Shift Right Logical Immediate	Логический сдвиг вправо	rd = rs1 >> imm			0x5	0x00	addi x6, x3, -12 ori x3, x1, 0x8F
srai	Shift Right Arithmetic Immediate	Арифметический сдвиг вправо	rd = rs1 >>> imm			0x5	0x20	
slti	Set Less Then Immediate	Результат сравнения A < B	rd = (rs1 < imm) ? 1 : 0			0x2	-	
sltiu	Set Less Then Immediate Unsigned	Беззнаковое сравнение A < B	rd = (rs1 < imm) ? 1 : 0			0x3		
lb	Load Byte	Загрузить байт из памяти	rd = SE(Mem[rs1 + imm][7:0])			0x0		
lh	Load Half	Загрузить полуслово из памяти	rd = SE(Mem[rs1 + imm][15:0])			0x1		op rd, imm(rs1)
lw	Load Word	Загрузить слово из памяти	rd = SE(Mem[rs1 + imm][31:0])	I	0000011	0x2	-	
lbu	Load Byte Unsigned	Загрузить беззнаковый байт из памяти	rd = Mem[rs1 + imm][7:0]			0x4		lh x1, 8(x5)
lbh	Load Half Unsigned	Загрузить беззнаковое полуслово из памяти	rd = Mem[rs1 + imm][15:0]			0x5		
sb	Store Byte	Сохранить байт в память	Mem[rs1 + imm][7:0] = rs2[7:0]			0x0		op rs2, imm(rs1)
sh	Store Half	Сохранить полуслово в память	Mem[rs1 + imm][15:0] = rs2[15:0]	S	0100011	0x1	-	
sw	Store Word	Сохранить слово в память	Mem[rs1 + imm][31:0] = rs2[31:0]			0x2		sw x1, 0xCF(x12)
beq	Branch if Equal	Перейти, если A == B	if (rs1 == rs2) PC += imm			0x0		
bne	Branch if Not Equal	Перейти, если A != B	if (rs1 != rs2) PC += imm			0x1		comp rs1, rs2, imm
blt	Branch if Less Then	Перейти, если A < B	if (rs1 < rs2) PC += imm			0x4		
bge	Branch if Greater or Equal	Перейти, если A >= B	if (rs1 >= rs2) PC += imm	B	1100011	0x5	-	beq x8, x9, offset
bltu	Branch if Less Then Unsigned	Перейти, если A < B беззнаковое	if (rs1 < rs2) PC += imm			0x6		bltu x20, x21, 0xFC
bgeu	Branch if Greater or Equal Unsigned	Перейти, если A >= B беззнаковое	if (rs1 >= rs2) PC += imm			0x7		
jal	Jamp And Link	Переход с сохранением адреса возврата	rd = PC + 4; PC += imm	J	1101111	-	-	jal x1, offset
jalr	Jamp And Link Register	Переход по регистру с сохранением адреса возврата	rd = PC + 4; PC = rs1	I	1100111	0x0		jalr x1, 0(x5)
lui	Load Upper Immediate	Загрузить константу в сдвинутую на 12	rd = imm << 12	U	0110111	-	-	lui x3, 0xFFFFF
auipc	Add Upper Immediate to PC	Сохранить счетчик команд в сумме с константой << 12	rd = PC + (imm << 12)		0010111			auipc x2, 0x000FF
ecall	Environment CALL	Передача управления операционной системе	Воспринимать как **nop**	I	1110011	-	-	-
ebreak	Environment BREAK	Передача управления отладчику						

Кодирование инструкций RISC-V

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Кодирование инструкций RISC-V

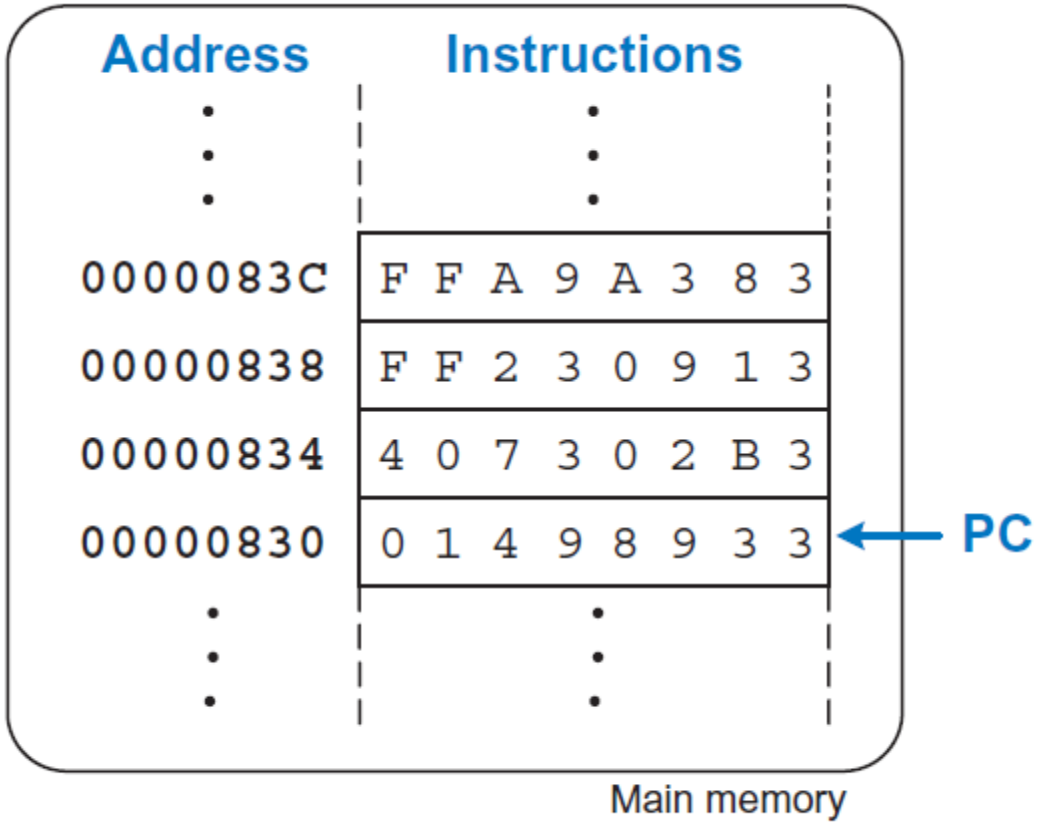
Assembly	funct7	rs2	rs1	funct3	rd	op	Machine Code						
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	funct7	rs2	rs1	funct3	rd	op	
`add s2, s3, s4` `add x18,x19,x20`	0	20	19	0	18	51	0000,000	10100	10011	000	10010	011,0011	(0x01498933)
`sub t0, t1, t2` `sub x5, x6, x7`	32	7	6	0	5	51	0100,000	00111	00110	000	00101	011,0011	(0x407302B3)

Machine Code	funct7	rs2	rs1	funct3	rd	op	Field Values						Assembly
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	funct7	rs2	rs1	funct3	rd	op	
(0x41FE83B3)	0100 000	11111	11101	000	00111	011 0011	32	31	29	0	7	51	`sub x7, x29,x31` `sub t2, t4, t6`

Machine Code	$imm_{11:0}$	rs1	funct3	rd	op	Field Values					Assembly
	12 bits	5 bits	3 bits	5 bits	7 bits	$imm_{11:0}$	rs1	funct3	rd	op	
(0xFDA48293)	1111 1101 1010	01001	000	00101	001 0011	-38	9	0	5	19	`addi x5, x9, -38` `addi t0, s1, -38`

Представление программы в памяти

Assembly code	Machine code
`add s2, s3, s4`	`0x01498933`
`sub t0, t1, t2`	`0x407302B3`
`addi s2, t1, -14`	`0xFF230913`
`lw t2, -6(s3)`	`0xFFA9A383`



Пример линейной программы

```
pc →  addi x13, x10, 3
pc →  li x14, 123456
pc →  add x14, x11, x14
pc →  or x11, x13, x14
pc →  slli x13, x10, 2
pc →  xor x12, x13, x11
```

Условные переходы

- Конструкция `if` может быть реализована с помощью инструкций передачи управления (`beq`, `bne` и др.)

C code

```
if (условие) {
    тело условия
}
```

```
int x, y;
…
if (x < y) {
    y = y – x;
}
```

RISC-V assembly

```
    (условие вычисляется в xN)
    beqz xN, endif
    (тело условия)
endif:
```

```
    // x: x10, y: x11
    slt x12, x10, x11
    beqz x12, endif
    sub x11, x11, x10
endif:
```

Иногда мы можем комбинировать вычисление условия и ветвление

```
    bge x10, x11, endif
    sub x11, x11, x10
endif:
```

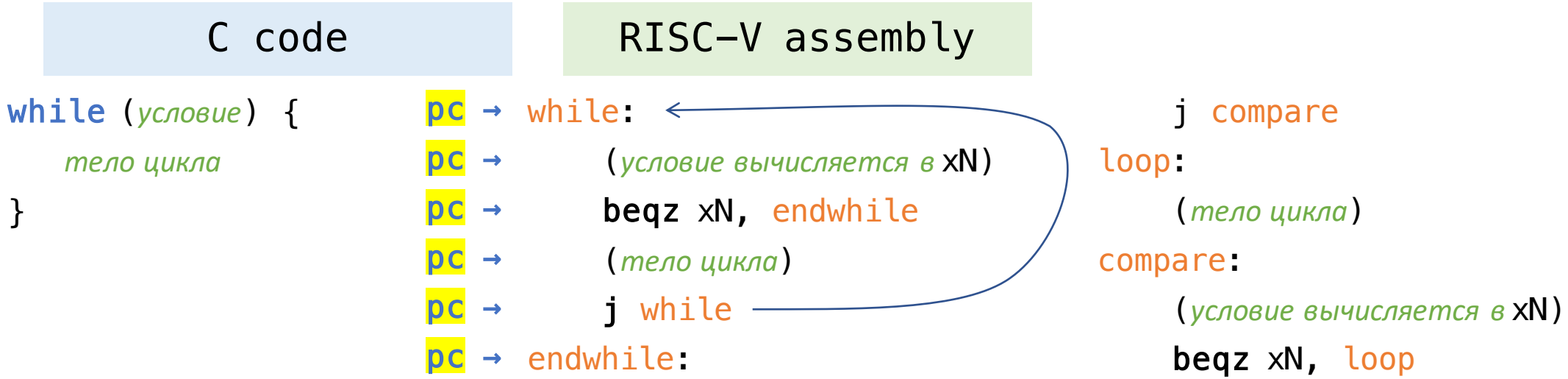
Условные переходы

- Конструкция **`if-else`** может быть реализована подобным образом

C code	RISC-V assembly
`if (`*условие*`) {`	`pc →` (*условие вычисляется в* `xN`)
тело условия если True	`pc → beqz xN, else`
`} else {`	`pc →` (*тело условия если True для if*)
тело условия если False	`pc → j endif`
`}`	`else:`
	(*тело условия если False для if*)
	`pc → endif:`

Циклы

- Циклы можно реализовать используя инструкции ветвления указывая назад



Циклы и условные переходы

C code

```c
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```

RISC-V assembly

```
// x: x10, y: x11
    j compare
loop:

    (тело цикла)

compare:
    bne x10, x11 loop
```

Циклы и условные переходы

C code

```c
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```

RISC-V assembly

```
// x: x10, y: x11
    j compare
loop:
    ble x10, x11, else
    sub x10, x10, x11
    j endif
else:
    sub x11, x11, x10
endif:
compare:
    bne x10, x11 loop
```

Процедуры

C code

```c
int gcd(int a, int b) {
    int x = a;
    int y = b;
    while (x != y) {
        if (x > y) {
            x = x - y;
        } else {
            y = y - x;
        }
    }
    return x;
}
```

RISC-V assembly

```
// x: x10, y: x11
    j compare
loop:
    ble x10, x11, else
    sub x10, x10, x11
    j endif
else:
    sub x11, x11, x10
endif:
compare:
    bne x10, x11 loop
```

Процедуры

- Процедуры (они же функции или подпрограммы) – это повторно используемые фрагменты кода реализующие вычисления определенной задачи
  - Использует имя как точку входа
  - Имеет ноль или более входных параметров
  - Использует локальное хранилище
  - Возвращает управление после того как закончит

- Использование процедур позволяет абстрагироваться и повторно использовать код
  - Большие программы состоят из простых процедур

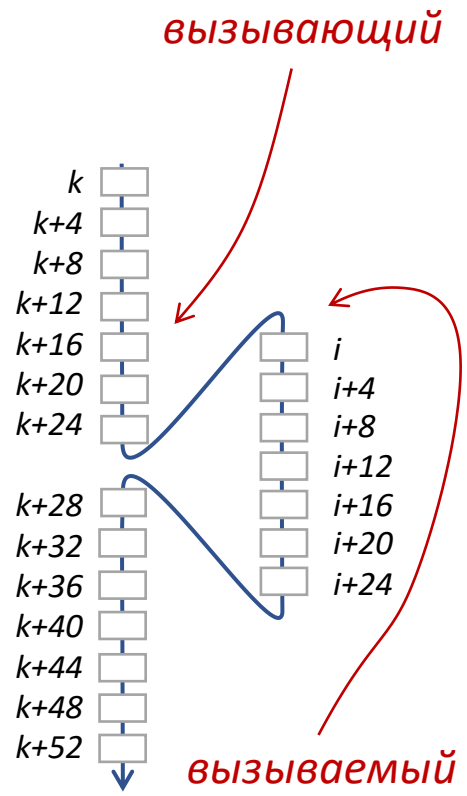
```
int gcd(int a, int b) {
    int x = a;
    int y = b;
    while (x != y) {
        if (x > y) {
            x = x - y;
        } else {
            y = y - x;
        }
    }
    return x;
}

bool coprimes(int a, int b) {
    return gcd(a, b) == 1;
}

coprimes(5, 10); // false
coprimes(9, 10); // true
```

Управление регистрами при вызове процедуры

- Вызывающий использует тот же набор регистров, что и вызываемая процедура
  - Вызывающий не должен полагаться на то, как вызываемая процедура управляет своим регистровым пространством
  - В идеале процедура должна иметь возможность использовать все регистры
- Либо вызывающий, либо вызываемый сохраняет регистры вызывающего в памяти и восстанавливает их, когда процедура завершает свое выполнение



Использование процедур

- Вызывающему необходимо передать параметры для вызываемой процедуры, так же как и вернуть результат обратно от вызываемой процедуры
  - Обе передачи происходят через регистры
- Процедура может быть вызвана из разных мест
  - Вызывающий может вызвать код процедуры просто выполнив безусловный переход к первой инструкции подпрограммы
  - Однако, для корректного возврата в то же место, откуда процедуру вызвали, необходимо знать адрес возврата

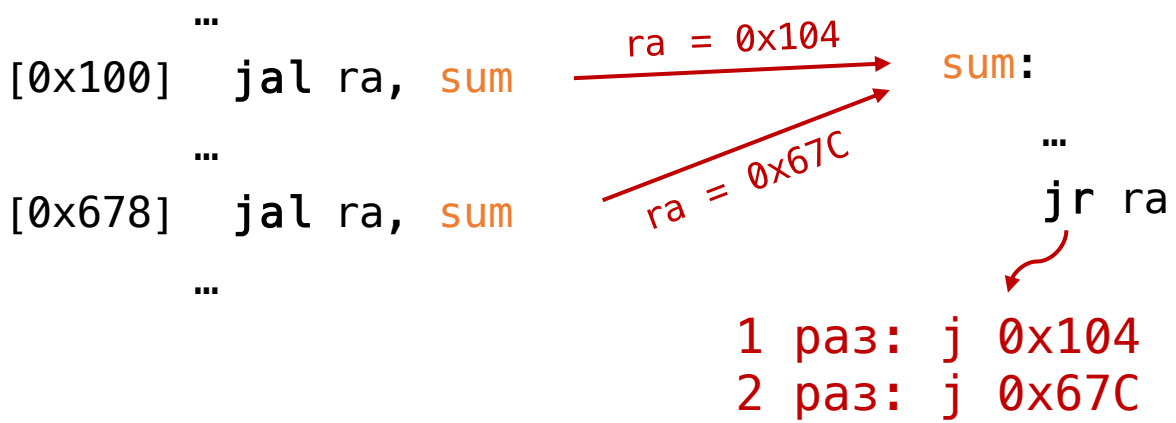
```
        …
[0x100]  j sum
        …
[0x678]  j sum
        …
```

```
sum:
    …
    j ?
```

Адрес возврата необходимо сохранять и передавать вызываемой процедуре

Вызов процедур

- Как передать контроль вызываемой процедуре и вернуть его обратно?
  `proc_call: jal ra, label`
  1. Размещение адреса `proc_call + 4` в регистре `ra` (return address)
  2. Переход к инструкции по адресу `label` (название процедуры)
  3. После выполнения процедуры, `jr ra` возвращается управление к вызывающей процедуре и выполнение программы продолжается



Сложности вызова процедур

- Предположим, что процедура A вызывает процедуру B, которая вызывает процедуру C
  - Единственный регистр адреса возврата работать не будет – адрес возврата процедуры B уничтожит адрес возврата процедуры A
  - Аналогичное осложнение возникает в области памяти, где сохраняются регистры процедуры A - это пространство должно отличаться от места, где сохраняются регистры процедуры B

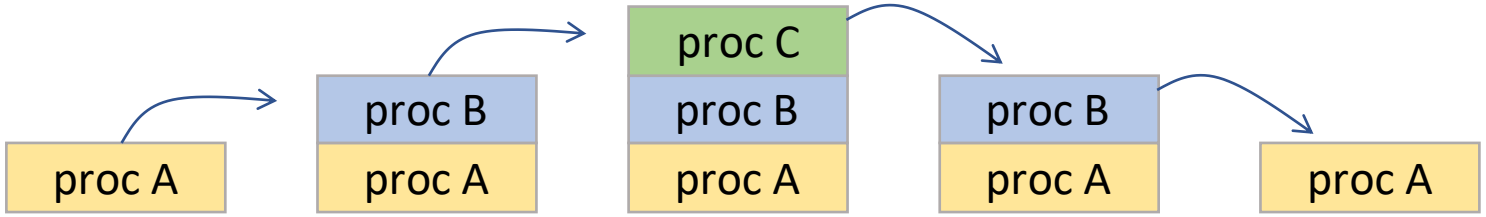
Необходимое хранилище для процедур

- Базовые требования для вызова процедур:
  - Входные аргументы
  - Адрес возврата
  - Результат
- Локальное хранилище:
  - Переменные, которые компилятор не смог поместить в регистровый файл
  - Пространство для сохранения значений регистров вызывающей процедуры, если они будут использоваться

Каждый вызов процедуры имеет свой собственный экземпляр всех этих данных, это называется активационная запись

Нужен Stack

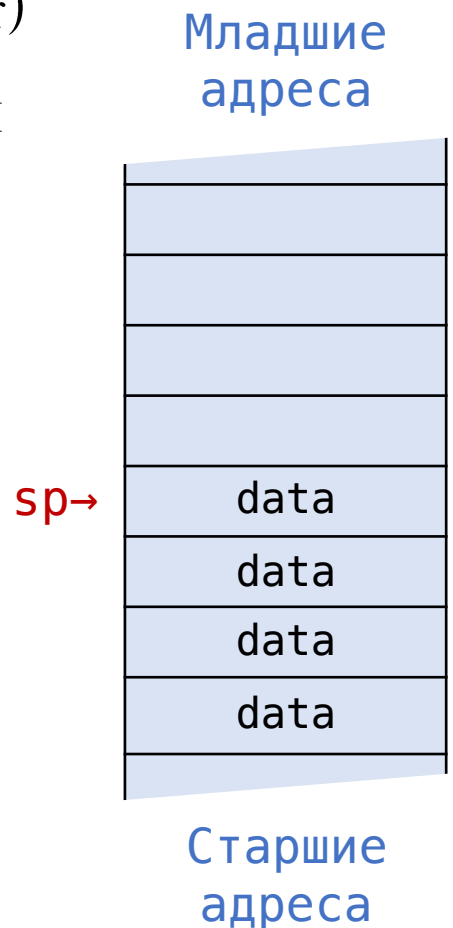
- Необходима структура данных для хранения активационных записей
- Под активационную запись память выделяется и освобождается по принципу last-in-first-out (LIFO)
- Stack: push, pop, доступ к верхнему элементу



- Нам необходим доступ только к активационной записи текущей процедуры

RISC-V Stack

- Стек хранится в памяти → нужен регистр указывающий на него
  - В RISC-V указатель на стек это `sp` (stack pointer)
- Стек растет снизу-вверх к младшим адресам
  - Push уменьшает адрес
  - Pop увеличивает адрес
- `sp` указывает на вершину стека
  (на последний записанный элемент)
- Стек можно использовать в любое время,
  но возвращать его нужно без изменений



Использование стека

- Пример размещения данных на стек

```
addi sp, sp, -N
sw ra, 0(sp)
sw a0, 4(sp)
```

- Пример освобождения стека

```
lw ra, 0(sp)
lw a0, 4(sp)
addi sp, sp, N
```

Соглашение о вызовах

- Соглашение о вызовах устанавливает правила использования регистров между процедурами
- В соглашении о вызовах RISC-V даются символические имена регистров `x0–x31` для обозначения их роли

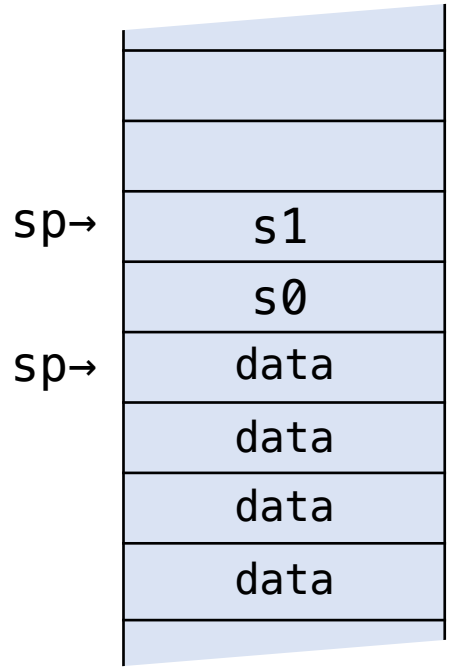
Имя	Регистр	Описание	Сохраняет
a0 – a7	x10 – x17	Аргументы для функции	Вызывающий
a0, a1	x10, x11	Возвращаемые значения	Вызывающий
ra	x1	Адрес возврата	Вызывающий
t0 – t6	x5 – x7, x28 – x31	Временные регистры	Вызывающий
s0 – s11	x8 – x9, x18 – x27	Сохраняемые (оберегаемые) регистры	Вызываемый
sp	x2	Указатель на вершину стека	Вызываемый
gp	x3	Указатель на глобальные переменные	---
tp	x4	Указатель потока	---
zero	x0	Аппаратный ноль	---

Пример: использование оберегаемых регистров

- При запуске `f` используются регистры `s0` и `s1` для хранения временных значений

```c
int f(int x, int y) {
    return (x + 3) | (y + 123456);
}
```

```
f:
pc →  addi sp, sp, -8        // выделить 2 слова (8 байт) на стеке
pc →  sw s0, 4(sp)           // сохранить s0
pc →  sw s1, 0(sp)           // сохранить s1
pc →  addi s0, a0, 3
pc →  li s1, 123456
pc →  add s1, a1, s1
pc →  or a0, s0, s1
pc →  lw s1, 0(sp)           // восстановить s1
pc →  lw s0, 4(sp)           // восстановить s0
pc →  addi sp, sp, 8         // освободить два слова на стеке
pc →  ret
```



Пример: использование необерегаемых регистров

Вызывающая

```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```

```
pc → li a0, 1
pc → li a1, 2
pc → addi sp, sp, -8
pc → sw ra, 0(sp)
pc → sw a1, 4(sp) // save y
pc → jal ra, sum
     // a0 = sum(s, y) = z
pc → lw a1, 4(sp) // restore y
pc → jal ra, sum
     // a0 = sum(z, y) = w
pc → lw ra, 0(sp)
pc → addi sp, sp, 8
```

Вызываемая

```
int sum(int a, int b) {
    return a + b;
}
```

```
  sum:
pc →  add a0, a0, a1
pc →  ret
```

Почему мы сохранили a1?

Вызываемая функция может изменить a1 (вызывающая функция не может знать произойдет это или нет)

Соглашения о вызовах

Вызывающая

Сохраняет регистр `ra` на стек перед тем как вызвать новую подпрограмму и стереть текущий адрес возврата. Также надо сохранить любые регистры `aN` или `tN` значения из которых планируется использовать в будущем

```
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp)
jal ra, func
lw ra, 0(sp)
lw a1, 4(sp)
addi sp, sp, 8
```

Вызываемая

Сохраняет оригинальные значения `sN` перед тем как использовать их в этой процедуре. Необходимо вернуть значения `sN` и стека до выхода из процедуры

```
func:
    addi sp, sp, -4
    sw s0, 0(sp)
    …
    lw s0, 0(sp)
    addi sp, sp, 4
    jr ra
```

Вложенные процедуры

- Если процедура вызывает другие процедуры, то необходимо сохранить адрес возврата
  - Потому, что `ra` должна сохранять вызываемая подпрограмма
- Пример:

```
bool coprimes(int a, int b) {
    return gcd(a, b) == 1;
}
```

```
coprimes:
    addi sp, sp, -4
    sw ra, 0(sp)
    call gcd            // перезаписываем ra
    addi a0, a0, -1
    sltiu a0, a0, 1
    lw ra, 0(sp)
    addi sp, sp, 4
    ret                 // требуется оригинальное ra
```

Передача больших структур данных

- Предположим мы хотим написать процедуру `vadd(a, b, c)` для сложения двух массивов `a` и `b` и записать результат в массив `c`
  - Массивы слишком большие для размещения в регистрах
- Мы будем размещать по одному элементу из `a` и `b` в регистры, складывать элементы и размещать результат в основной памяти
- Как нам передать массивы `a` и `b` как аргументы?
  - Передавать базовый адрес и размер каждого массива в виде аргументов

Передача больших структур данных

```
// Найти элемент массива
// с максимальным значением
int maximum(int a[], int size) {
    int max = 0;
    for (int i = 0; i < size; i++) {
        if (a[i] > max) {
            max = a[i];
        }
    }
    return max;
}

int main() {
    int ages[5] = {23, 4, 6, 81, 16};
    int max = maximum(ages, 5);
}
```

Передача больших структур данных

```
// Найти элемент массива
// с максимальным значением
int maximum(int a[], int size) {
    int max = 0;
    for (int i = 0; i < size; i++) {
        if (a[i] > max) {
            max = a[i];
        }
    }
    return max;
}

int main() {
    int ages[5] = {23, 4, 6, 81, 16};
    int max = maximum(ages, 5);
}
```

```
main:
    li a0, ages
    li a1, 5
    call maximum
    // max вернется в a0
```

```
ages:
    23
    4
    6
    81
    16
```

Передача больших структур данных

```c
// Найти элемент массива
// с максимальным значением
int maximum(int a[], int size) {
    int max = 0;
    for (int i = 0; i < size; i++) {
        if (a[i] > max) {
            max = a[i];
        }
    }
    return max;
}

int main() {
    int ages[5] = {23, 4, 6, 81, 16};
    int max = maximum(ages, 5);
}
```

```asm
maximum:
    mv t0, zero     // t0: i
    mv t1, zero     // t1: max
    j compare
loop:
    slli t2, t0, 2  // t2: i*4
    add t3, a0, t2  // t3: адрес a[i]
    lw t4, 0(t3)    // t4: a[i]
    ble t4, t1, endif
    mv t1, t4       // max = a[i]
endif:
    addi t0, t0, 1  // i++
compare:
    blt t0, a1, loop
    mv a0, t1       // a0 = max
    ret
```

Почему не стоит всегда использовать указатели

```c
// Найти периметр треугольника
int perimA(int a, int b, int c) {
    int res = a + b + c;
    return res;
}

int perimB(int sides[], int size) {
    int res = 0;
    for (int i = 0; i < size; i++) {
        if (a[i] > max) {
            res = res + sides[i];
        }
    }
    return res;
}
```

```asm
perimA:
    add t0, a0, zero            // t0: res
    add t0, t0, a1
    add t0, t0, a2
    mv a0, t0
    ret

perimB:
    mv t0, zero                 // t0: i
    mv t1, zero                 // t1: res
    j compare
loop:
    slli t2, t0, 2              // t2: i*4
    add t3, a0, t2              // t3: адрес sides[i]
    lw t4, 0(t3)                // t4: sides[i]
    add t1, t1, t4
    addi t0, t0, 1              // i++
compare:
    blt t0, a1, loop
    mv a0, t1
    ret
```

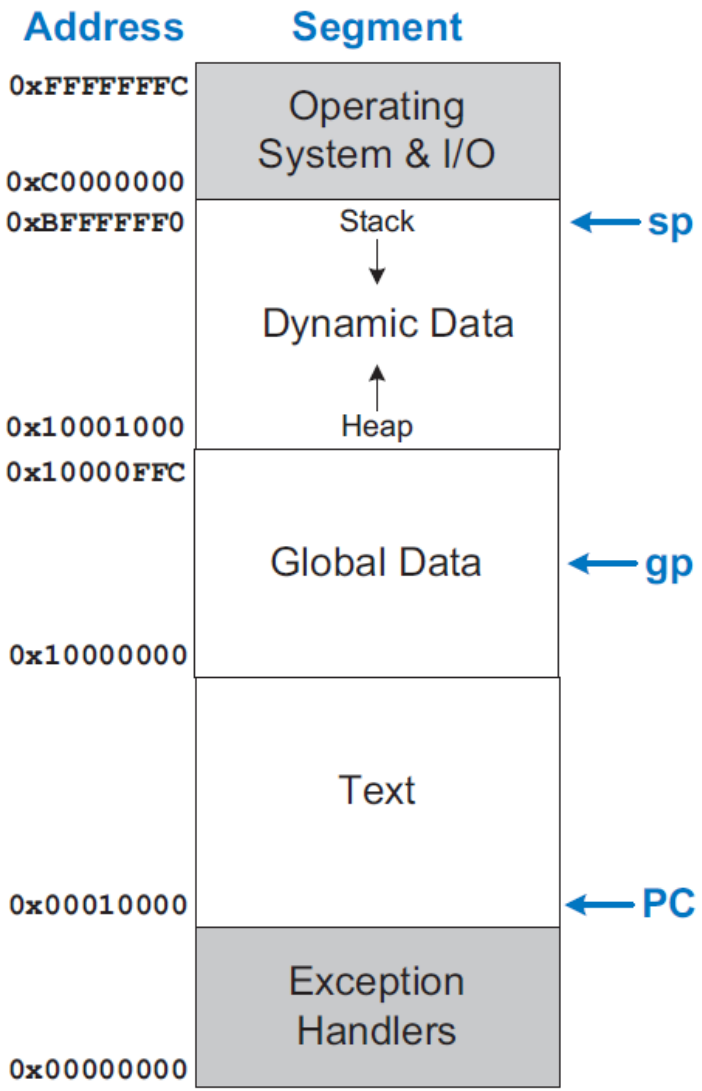
Передача больших структур данных

- Другие сложные структуры данных, такие как словари, структуры, связанные списки и т. д., будут следовать той же методологии передачи указателя на структуру данных в качестве аргумента процедуры вместе с любой дополнительной необходимой информацией, такой как количество элементов и т. д.
- Когда возвращаемое значение представляет собой сложную структуру данных, тогда эта структура данных сохраняется в памяти, а процедурой возвращается указатель на нее

Карта памяти

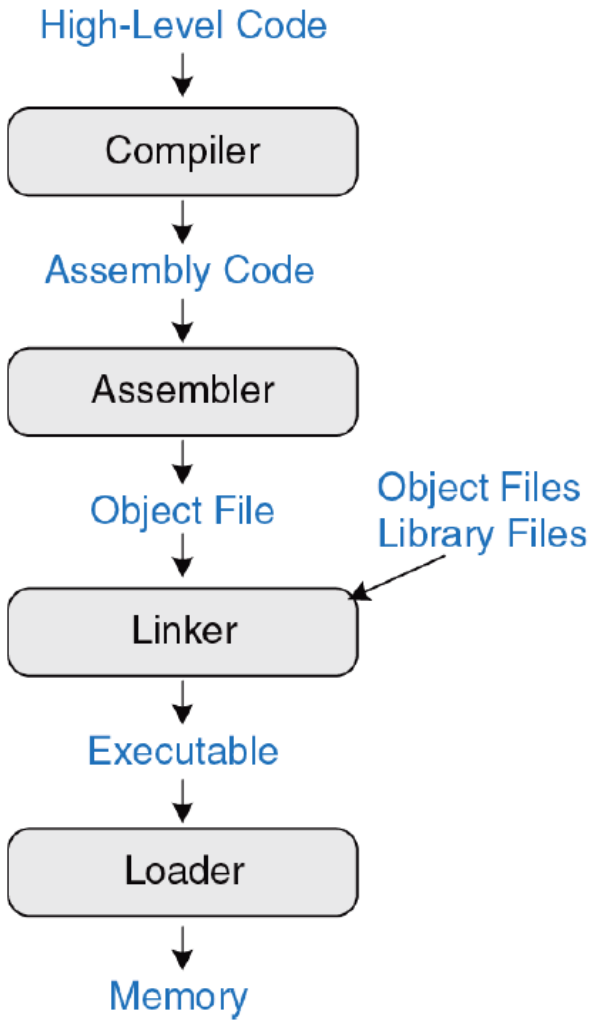
- Большинство языков программирования (в том числе C) используют три отдельных области памяти для данных:
  - `Stack`: Содержит данные используемые процедурными вызовами
  - `Static`: Содержит глобальные переменные, которые существуют в течении всего времени жизни программы
  - `Heap`: Содержит динамически-распределяемые данные
    - В C программист управляет кучей в ручную, размещая новые данные с помощью malloc() и освобождая с помощью free()
    - В Python, Java, и большинстве современных языков, куча управляется автоматически
- `Text`: область памяти содержащая программный код

Карта памяти RISC-V



- Области `Text`, `Static` и `Heap` и располагаются последовательно, начиная с младших адресов
- `Heap` растет в сторону старших адресов
- `Stack` начинается со старших адресов и растет к младшим адресам
- `sp` (stack pointer) – указатель на вершину стека
- `gp` (global pointer) – указатель на область `Static (Global Data)`

Компиляция кода



High-Level Code

```
int f, g, y;

int func(int a, int b) {
  if (b < 0)
    return (a + b);
  else
    return(a + func(a, b - 1));
}

void main() {
  f=2;
  g=3;
  y=func(f,g);

  return;
}
```

RISC-V Assembly Code

```
        .text
        .globl     func
        .type func,@function
func:
        addi  sp,sp,-16
        sw    ra,12(sp)
        sw    s0,8(sp)
        mv    s0,a0
        add   a0,a1,a0
        bge   a1,zero,.L5
.L1:
        lw    ra,12(sp)
        lw    s0,8(sp)
        addi  sp,sp,16
        jr    ra
.L5:
        addi  a1,a1,-1
        mv    a0,s0
        call  func
        add   a0,a0,s0
        j     .L1

        .globl         main
        .type main,    @function
main:
        addi  sp,sp,-16
        sw    ra,12(sp)
        lui   a5,%hi(f)
        li    a4,2
        sw    a4,%lo(f)(a5)
        lui   a5,%hi(g)
        li    a4,3
        sw    a4,%lo(g)(a5)
        li    a1,3
        li    a0,2
        call  func
        lui   a5,%hi(y)
        sw    a0,%lo(y)(a5)
        lw    ra,12(sp)
        addi  sp,sp,16
        jr    ra
        .comm y,4,4
        .comm g,4,4
        .comm f,4,4
```

```
00000000 <func>:
int f, g, y;
int func(int a, int b) {
   0:  ff010113                addi  sp,sp,-16
   4:  00112623                sw    ra,12(sp)
   8:  00812423                sw    s0,8(sp)
   c:  00050413                mv    s0,a0
  if (b<0) return (a+b);
  10:  00a58533                add   a0,a1,a0
  14:  0005da63                bgez  a1,28 <.L5>
00000018 <.L1>:
  else return(a + func(a, b-1));
}
  18:  00c12083                lw    ra,12(sp)
  1c:  00812403                lw    s0,8(sp)
  20:  01010113                addi  sp,sp,16
  24:  00008067                ret
00000028 <.L5>:
  else return(a + func(a, b-1));
  28:  fff58593                addi  a1,a1,-1
  2c:  00040513                mv    a0,s0
  30:  00000097                auipc ra,0x0
  34:  000080e7                jalr  ra # 30 <.LVL5+0x4>
  38:  00850533                add   a0,a0,s0
  3c:  fddff06f                j     18 <.L1>

00000040 <main>:
void main() {
  40:  ff010113                addi  sp,sp,-16
  44:  00112623                sw    ra,12(sp)
  f=2;
  48:  000007b7                lui   a5,0x0
  4c:  00200713                li    a4,2
  50:  00e7a023                sw    a4,0(a5) # 0 <func>
  g=3;
  54:  000007b7                lui   a5,0x0
  58:  00300713                li    a4,3
  5c:  00e7a023                sw    a4,0(a5) # 0 <func>
  y=func(f,g);
  60:  00300593                li    a1,3
  64:  00200513                li    a0,2
  68:  00000097                auipc ra,0x0
  6c:  000080e7                jalr  ra # 68 <main+0x28>
  70:  000007b7                lui   a5,0x0
  74:  00a7a023                sw    a0,0(a5) # 0 <func>
  return;
}
  78:  00c12083                lw    ra,12(sp)
  7c:  01010113                addi  sp,sp,16
  80:  00008067                ret
```

SYMBOL TABLE:

Address			Size	Symbol Name
00000000	l	d .text	00000000	.text
00000000	l	d .data	00000000	.data
00000000	g	F .text	00000040	func
00000040	g	F .text	00000044	main
00000004		O *COM*	00000004	f
00000004		O *COM*	00000004	g
00000004		O *COM*	00000004	y

```
00000000 <func>:
int f, g, y;
int func(int a, int b) {
   0: ff010113                addi  sp,sp,-16
   4: 00112623                sw    ra,12(sp)
   8: 00812423                sw    s0,8(sp)
   c: 00050413                mv    s0,a0
  if (b<0) return (a+b);
  10: 00a58533                add   a0,a1,a0
  14: 0005da63                bgez  a1,28 <.L5>
00000018 <.L1>:
  else return(a + func(a, b-1));
}
  18: 00c12083                lw    ra,12(sp)
  1c: 00812403                lw    s0,8(sp)
  20: 01010113                addi  sp,sp,16
  24: 00008067                ret
00000028 <.L5>:
  else return(a + func(a, b-1));
  28: fff58593                addi  a1,a1,-1
  2c: 00040513                mv    a0,s0
  30: 00000097                auipc ra,0x0
  34: 000080e7                jalr  ra # 30 <.LVL5+0x4>
  38: 00850533                add   a0,a0,s0
  3c: fddff06f                j     18 <.L1>

00000040 <main>:
void main() {
  40: ff010113                addi  sp,sp,-16
  44: 00112623                sw    ra,12(sp)
  f=2;
  48: 000007b7                lui   a5,0x0
  4c: 00200713                li    a4,2
  50: 00e7a023                sw    a4,0(a5) # 0 <func>
  g=3;
  54: 000007b7                lui   a5,0x0
  58: 00300713                li    a4,3
  5c: 00e7a023                sw    a4,0(a5) # 0 <func>
  y=func(f,g);
  60: 00300593                li    a1,3
  64: 00200513                li    a0,2
  68: 00000097                auipc ra,0x0
  6c: 000080e7                jalr  ra # 68 <main+0x28>
  70: 000007b7                lui   a5,0x0
  74: 00a7a023                sw    a0,0(a5) # 0 <func>
  return;
}
  78: 00c12083                lw    ra,12(sp)
  7c: 01010113                addi  sp,sp,16
  80: 00008067                ret
```

SYMBOL TABLE:

Address				Size	Symbol Name
00000000	l	d	.text	00000000	.text
00000000	l	d	.data	00000000	.data
00000000	g	F	.text	00000040	func
00000040	g	F	.text	00000044	main
00000004		0	*COM*	00000004	f
00000004		0	*COM*	00000004	g
00000004		0	*COM*	00000004	y

```
00010144 <func>:
int f, g, y;

int func(int a, int b) {
   10144: ff010113            addi  sp,sp,-16
   10148: 00112623            sw    ra,12(sp)
   1014c: 00812423            sw    s0,8(sp)
   10150: 00050413            mv    s0,a0
  if (b<0) return (a+b);
   10154: 00a58533            add   a0,a1,a0
   10158: 0005da63            bgez  a1,1016c <func+0x28>
  else return(a + func(a, b-1));
}
   1015c: 00c12083            lw    ra,12(sp)
   10160: 00812403            lw    s0,8(sp)
   10164: 01010113            addi  sp,sp,16
   10168: 00008067            ret
  else return(a + func(a, b-1));
   1016c: fff58593            addi  a1,a1,-1
   10170: 00040513            mv    a0,s0
   10174: fd1ff0ef            jal   ra,10144 <func>
   10178: 00850533            add   a0,a0,s0
   1017c: fe1ff06f            j     1015c <func+0x18>

00010180 <main>:
void main() {
   10180: ff010113            addi  sp,sp,-16
   10184: 00112623            sw    ra,12(sp)
  f=2;
   10188: 00200713            li    a4,2
   1018c: c4e1a823            sw    a4,-944(gp) # 11a30 <f>
  g=3;
   10190: 00300713            li    a4,3
   10194: c4e1aa23            sw    a4,-940(gp) # 11a34 <g>
  y=func(f,g);
   10198: 00300593            li    a1,3
   1019c: 00200513            li    a0,2
   101a0: fa5ff0ef            jal   ra,10144 <func>
   101a4: c4a1ac23            sw    a0,-936(gp) # 11a38 <y>

  return;
}
   101a8: 00c12083            lw    ra,12(sp)
   101ac: 01010113            addi  sp,sp,16
   101b0: 00008067            ret
```