Архитектура ARM Cortex-M и ее сравнение с другими ARM архитектурами

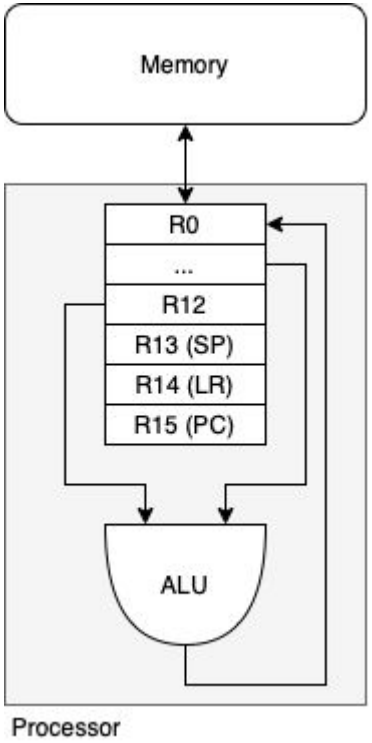
Где применяются знания архитектуры процессора ?

Где применяются знания архитектуры процессора ?

- Операционные системы и embedded firmware: Baremetal разработка, смена контекста (процессов,потоков) в ОС, userspace и syscalls и др.…
- Компиляторы: эффективно использовать регистры процессора, оптимизировать машинный код для эффективного заполнения pipeline, эффективная векторизация и loop unrolling, и др.…
- Корректность и Performance ваших программ: уменьшение кэш промахов, многопоточность - Out Of Order execution and memory model …

```
int calculate(uint32_t repeat,
              uint32_t x, uint32_t y) {
    int max = 10;
    int res = 0;
    for (int i = 0; i < repeat; ++i) {
        int sum = x + y;
        int mul = x * y;

        res += sum + mul;
        x = (sum < max) ? sum : max;
    }
    return res;
}

void main(void) {
    int res = calculate(10, 2, 4);
}
```

- Как процессор выполняет код ?
- Как функция *calculate* получает значения *repeat*, *x* и *y* ?
- Как возвращается значение из *calculate* ?
- Где хранятся переменные *max*, *res*, … ?
- Как из *calculate функции* вернуться и продолжить исполнение *main* ?
- Как выполняется вычисление sum и mul?
- Могут ли вычисления sum и mul выполняться параллельно ?



- SP - Stack Pointer Register (Вершина стека)
- LR - Link Register (Адрес возврата)
- PC - Program Counter Register (Адрес следующей выполняемой инструкции)

ALU (Arithmetic Logic Unit) Operations:
Add, Sub, And, Or, Shift, Mul, Div, ...

Зачем нужны регистры ?
Доступ к регистрам быстрее чем к памяти - в том числе быстрее чем к кэш памяти !!!

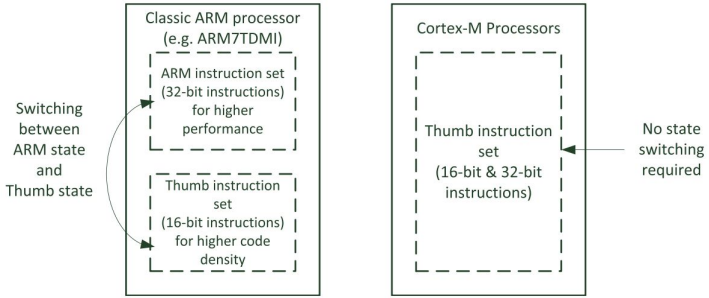
volatile ??

```
int c = a + b;
```

```
Load R1, A
Load R12, B
Add R0, R12, R1
Store R0, C
```

NOTE: Cortex M3/M4/M7 на основе Harvard архитектуры - используют разные шины для команд и данных.

Data Processing Instructions	ADD, SUB, AND, ORR, CMP, MUL, SDIV, UDIV, LSL, LSR, …
Data Transfer Instructions	MOV, LDR, LDRH, LDRB, STR, PUSH, POP, …
Branching Instructions	BEQ, BNE, B, BL, BX, …
Special	SVC, CPSID, CPSIE, MRS, MSR, WFI, ISB, …



Thumb2 ISA является смесью 16- и 32-битных инструкций. Инструкции Thumb2 дают улучшение плотности кода на 26% по сравнению с 32-битными инструкциями ARM и производительности на 25% по сравнению с 16-битными инструкциями Thumb.

https://developer.arm.com/documentation/ddi0337/h/CHDDIGAC
https://developer.arm.com/documentation/qrc0001/m

```
ADD R0, R1, R2      ; R0=R1+R2
SUB R0, R1, R2      ; R0=R1-R2

MOV R1, R2          ; R1=R2
MOV R1, #0x300      ; R1=0x300
STR R0, [R1]        ; *R1=R0 (Запись в память по адресу R1 значения R0)
LDR R2, [R1]        ; R2=*R1 (Запись в R2 значения из адреса R1)

CMP R0, R1          ; Сравнить R0 и R1, Результат сравнения в APSR регистр
BEQ func            ; Переход (Branch) к выполнению func,
                      если результат предыдущей операции был == (equal)
B func              ; Переход (Branch) к выполнению func
```

https://developer.arm.com/documentation/ddi0337/h/CHDDIGAC
https://developer.arm.com/documentation/qrc0001/m

- Как функция *calculate* получает значения *repeat*, *x* и *y* ?
- Как возвращается значение из *calculate* ?

```
int calculate(uint32_t repeat,
              uint32_t x, uint32_t y) {
    int max = 10;
    int res = 0;
    for (int i = 0; i < repeat; ++i) {
        int sum = x + y;
        int mul = x * y;

        res += sum + mul;
        x = (sum < max) ? sum : max;
    }
    return res;
}

void main(void) {
    int res = calculate(10, 2, 4);
}
```

https://godbolt.org/z/3n1q3Tbj7

- Как функция *calculate* получает значения *repeat*, *x* и *y* ?
- Как возвращается значение из *calculate* ?

R0, R1, R2, R3	Аргументы функции (Если функция имеет >4 аргументов - дополнительные аргументы передаются через стэк). Caller-save регистры: Вызывающая функция должна сохранить их перед вызовом функции. Вызываемая функция может менять эти значения.
R0, R1	Возвращаемое значение
R4 - R11	Callee-save регистры: Если *вызываемая функция* использует их, вызываемая функция должна сохранить их на стэк перед использованием и восстановить перед выходом. Вызывающая функция имеет гарантии что вызываемые функции не будут менять эти регистры. NOTE: R9 может иметь специальное назначение и не быть Callee-save.

https://godbolt.org/z/3n1q3Tbj7

- Как функция *calculate* получает значения *repeat*, *x* и *y* ?
- Как возвращается значение из *calculate* ?

```c
int calculate(uint32_t repeat,
              uint32_t x, uint32_t y) {
    int max = 10;
    int res = 0;
    for (int i = 0; i < repeat; ++i) {
        int sum = x + y;
        int mul = x * y;

        res += sum + mul;
        x = (sum < max) ? sum : max;
    }
    return res;
}

void main(void) {
    int res = calculate(10, 2, 4);
}
```

```
main:
 ...
 movs r2, #4
 movs r1, #2
 movs r0, #10
 bl calculate
 str r0, [r7, #4]
```

```
calculate:
 push {r7}
 sub sp, sp, #44
 add r7, sp, #0
 str r0, [r7, #12]
 str r1, [r7, #8]
 str r2, [r7, #4]
 ...
 adds r7, r7, #44
 mov sp, r7
 pop {r7}
 bx lr
```

Calling Convention является частью *ABI* (*Application Binary Interface*) и позволяет обеспечить совместимость между машинами поддерживающими один ABI.

https://godbolt.org/z/4o6hK1zdh

Register	ABI Name	Description	Saved by Calle-
x0	zero	hardwired zero	-
x1	ra	return address	-R
x2	sp	stack pointer	-E
x3	gp	global pointer	-
x4	tp	thread pointer	-
x5	t0	temporary register 0	-R
x6	t1	temporary register 1	-R
x7	t2	temporary register 2	-R
x8	s0 / fp	saved register 0 / frame pointer	-E
x9	s1	saved register 1	-E
x10	a0	function argument 0 / return value 0	-R
x11	a1	function argument 1 / return value 1	-R
x12	a2	function argument 2	-R
x13	a3	function argument 3	-R
x14	a4	function argument 4	-R
x15	a5	function argument 5	-R
x16	a6	function argument 6	-R
x17	a7	function argument 7	-R
x18	s2	saved register 2	-E
x19	s3	saved register 3	-E
x20	s4	saved register 4	-E
x21	s5	saved register 5	-E

```
main:
 ...
 li a0, 10
 li a1, 2
 li a2, 4

 call calculate
 sw a0, -16(s0)
```

```
calculate:
 addi sp, sp, -48
 sw ra, 44(sp)
 sw s0, 40(sp)
 addi s0, sp, 48
 sw a0, -12(s0)
 sw a1, -16(s0)
 sw a2, -20(s0)
 li a0, 10
...
 lw a0, -28(s0)
 lw ra, 44(sp)
 lw s0, 40(sp)
 addi sp, sp, 48
 ret
```

```
LI a0, 10        # load immediate 10 in a0
SW a0, -16(s0)   # store a0 in address: (s0-16)
LW s0, 40(sp)    # load from address (sp+40) into s0
ret              # jmp to ra
call func        # save next addr into ra and jump in func
```

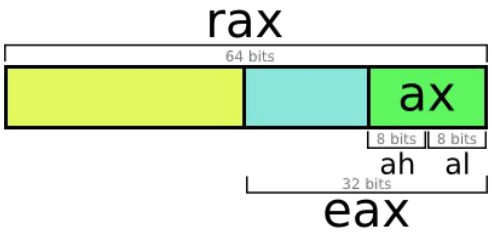
https://godbolt.org/z/q9vYcaE7x
https://habr.com/en/post/533272/

Register	Purpose	Saved across calls
%rax	temp register; return value	No
%rbx	callee-saved	Yes
%rcx	used to pass 4th argument to functions	No
%rdx	used to pass 3rd argument to functions	No
%rsp	stack pointer	Yes
%rbp	callee-saved; base pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10-r11	temporary	No
%r12-r15	callee-saved registers	Yes

```
main:
 ...
 mov edi, 10
 mov esi, 2
 mov edx, 4
 call calculate
 mov dword ptr [rbp - 8], eax
```

```
calculate:
 push rbp
 mov rbp, rsp
 mov dword ptr [rbp - 4], edi
 mov dword ptr [rbp - 8], esi
 mov dword ptr [rbp - 12], edx
...
 mov eax, dword ptr [rbp - 20]
 pop rbp
 ret
```

```
call func # Jump unconditionally to target and push return value (current PC + 1) onto stack
ret       # Pop the return address off the stack and jump unconditionally to this address
```



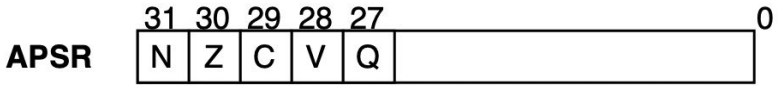

```
INC counter
```

```
LDR R0, =counter
LDR R1, [R0]
ADD R1, #1
STR R1, [R0]
```

https://godbolt.org/z/9xxrP5neY
http://6.s081.scripts.mit.edu/sp18/x86-64-architecture-guide.html

Calling Convention, X86-64, example

R13 (SP)	Stack Pointer	HW purpose
R14 (LR)	Link Register	
R15 (PC)	Program Counter	
xPSR registers	*Application PSR* (статус ALU: переполнение, отрицательное, ноль); *Interrupt PSR*; *Execution PSR*;	
R7, R11	Может использоваться как Frame Pointer, если код скомпилирован с -fno-omit-frame-pointer	ARM procedure call standard
R12 Intra-Procedure-call scratch register	Может использоваться для различных целей как дополнительный регистр который можно не сохранять внутри вызываемой функции (как R0-R3, но не для аргументов)	
R9	Может быть platform specific, Например: *Static Base* в Position Independent data model	

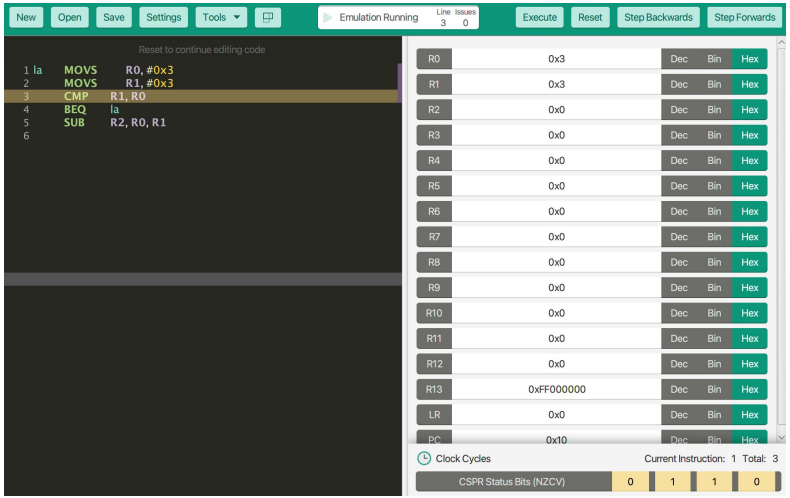


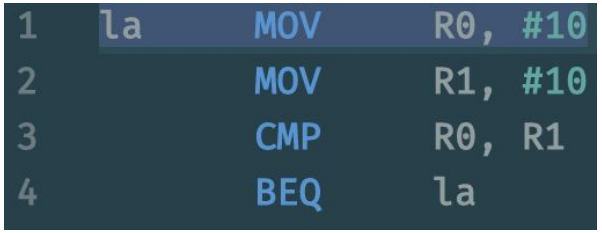

```
CMP R1, R0   ; Compare R0 and R1, set APSR
BEQ func2    ; Branch if equal

MRS R0, APSR  ; Read APSR into R0
MSR APSR, R0  ; Write APSR from R0
```

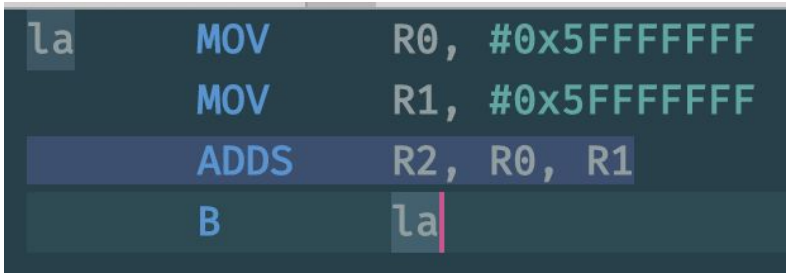
NOTE RISC V: beq rs, x0, offset
https://erik-engheim.medium.com/arm-x86-and-risc-v-microprocessors-compared-92bf0d46fd52

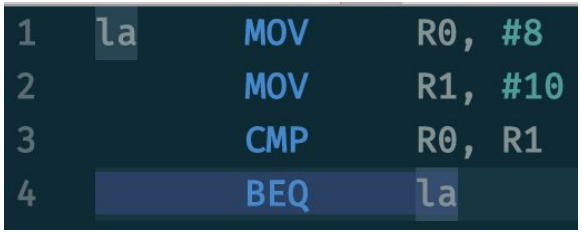
N (Negative)	1: Результат предыдущей операции был отрицательный, или при сравнение результат был меньше ((result >> 31)&1)
Z (Zero)	1: Результат предыдущей операции был == 0 0: Результат предыдущей операции был != 0
C (Carry)	1: Операция сложения **привела** к переносу (result > 2^32)]; Операция вычитания **не привела** к заимствованию (result >=0)
V (Overflow)	1: Операция привела к переполнению ((a >> 31 == b >> 31) & (a >> 31 != (result>>31)&1))
Q	1: Saturated операции (qadd, qsub, …) привели к переполнению: *int saturate_add_int(int x, int y)* *{* *if (y > INT_MAX - x) return INT_MAX;* *return x + y;* *}*

https://github.com/cornell-brg/pydgin/blob/master/arm/isa.py









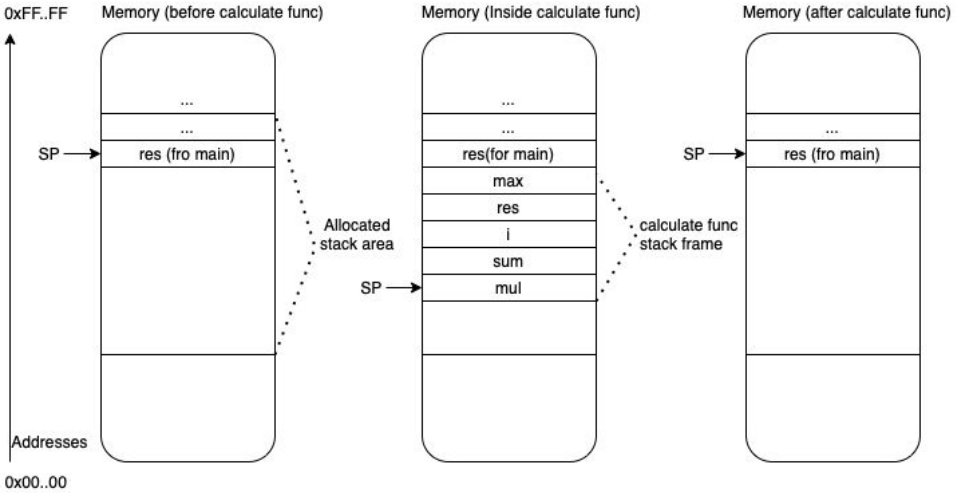
https://github.com/tomcl/V2releases/releases
https://tomcl.github.io/visual2.github.io/

```
int calculate(uint32_t repeat,
              uint32_t x, uint32_t y) {
    int max = 10;
    int res = 0;
    for (int i = 0; i < repeat; ++i) {
        int sum = x + y;
        int mul = x * y;

        res += sum + mul;
        x = (sum < max) ? sum : max;
    }
    return res;
}

void main(void) {
    int res = calculate(10, 2, 4);
}
```

Где хранятся переменные *max*, *res*, … ?

Они используются только внутри calculate - значит нам нужно какая то временная область памяти под них - **stack** !!!

SP регистр - указатель стэка;



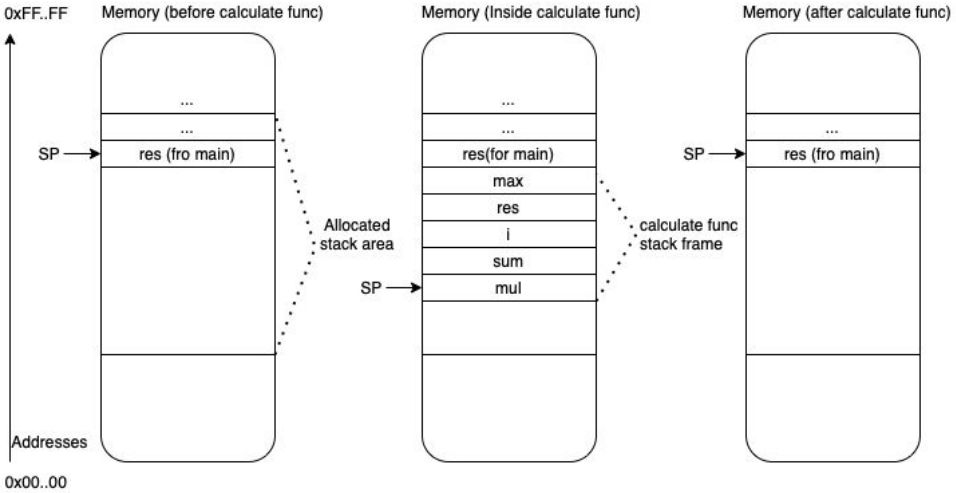
https://godbolt.org/z/3n1q3Tbj7

```c
int calculate(uint32_t repeat,
              uint32_t x, uint32_t y) {
    int max = 10;
    int res = 0;
    for (int i = 0; i < repeat; ++i) {
        int sum = x + y;
        int mul = x * y;

        res += sum + mul;
        x = (sum < max) ? sum : max;
    }
    return res;
}

void main(void) {
    int res = calculate(10, 2, 4);
}
```

```
calculate:
 push {r7}
 sub sp, sp, #44
 add r7, sp, #0
 ...
 adds r7, r7, #44
 mov sp, r7
 pop {r7}
 bx lr
```

NOTE:

- RISC-V: SP register
- x86-64: RSP register



https://godbolt.org/z/3n1q3Tbj7

- Если входных аргументов > 4 ?

```
uint32_t calculate(int a, int b, int c,
                   int d, int e, int f){
  return (a+b+c+d+e+f);
}

int main(void) {
   int res = calculate(10, 2, 4,
                       7, 3, 5);
   return res;
}
```

```
main:
 ...
 movs r3, #5
 str r3, [sp, #4]
 movs r3, #3
 str r3, [sp]
 movs r3, #7
 movs r2, #4
 movs r1, #2
 movs r0, #10
 bl calculate
 mov r3, r0
```

https://godbolt.org/z/egnP7acKa

- Как процессору узнать какой код продолжить выполнять, после выполнения функции calculate ?

```c
uint32_t calculate(int a, int b, int c,
                   int d, int e, int f){
  return (a+b+c+d+e+f);
}

int main(void) {
   int res = calculate(10, 2, 4,
                       7, 3, 5);
   return res;
}
```

```
main:
 ...
 bl calculate
 // next op
 ...

calculate:
 push {r7}
 sub sp, sp, #20
 ...
 adds r7, r7, #20
 mov sp, r7
 pop {r7}
 bx lr
```

LR регистр: используется для хранения адреса возврата при вызове функции;

bl (Branch with Link) - копирует адрес следующей (за bl) инструкции в регистр LR и выполняет branch to label;

https://godbolt.org/z/egnP7acKa

- Если у нас только один LR регистр, куда сохранять адрес возврата при вызове еще одной функции ?

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

int calculate(int a, int b) {
    return sum(a, b);
}

void main(void) {
    int res = calculate(2, 4);
}
```

```
main:
 movs r1, #4
 movs r0, #2
 bl calculate
 str r0, [r7, #4]
```

```
calculate(int, int):
 push {r7, lr}
 sub sp, sp, #8
 ...
 bl sum
 ...
 pop {r7, pc}
```

```
sum(int, int):
 push {r7}
 sub sp, sp, #12
 add r7, sp, #0
 ...
 adds r7, r7, #12
 mov sp, r7
 pop {r7}
 bx lr
```

При вызове еще одной функции, LR сохраняется на стэк.

Почему не использовать стэк всегда - зачем нужен LR ?
Доступ к регистрам быстрее чем к памяти - в том числе быстрее чем к кэш памяти !!!

https://godbolt.org/z/hscqrofK5

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

int calculate(int a, int b) {
    return sum(a, b);
}

void main(void) {
    int res = calculate(2, 4);
}
```

```
main:
 mov edi, 2
 mov esi, 4
 call calculate(int, int)
 mov dword ptr [rbp - 4], eax

calculate(int, int):
 push rbp
 mov rbp, rsp
 sub rsp, 16
 mov dword ptr [rbp - 4], edi
 mov dword ptr [rbp - 8], esi
 mov edi, dword ptr [rbp - 4]
 mov esi, dword ptr [rbp - 8]
 call sum(int, int)
 add rsp, 16
 pop rbp
 ret
```

```
sum(int, int):
 push rbp
 mov rbp, rsp
 mov dword ptr [rbp - 4], edi
 mov dword ptr [rbp - 8], esi
 mov eax, dword ptr [rbp - 4]
 add eax, dword ptr [rbp - 8]
 pop rbp
 ret
```

call func # Jump unconditionally to target and push return value (current PC + 1) onto stack

ret # Pop the return address off the stack and jump unconditionally to this address

https://godbolt.org/z/dv81Kfxb7

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

int calculate(int a, int b) {
    return sum(a, b);
}

void main(void) {
    int res = calculate(2, 4);
}
```

```
ret                 # jmp to ra
call func           # save next addr into ra and jump
in func
```

Задание: используя godbolt https://godbolt.org/ сгенерить ассемблер данного кода для архитектуры RISC-V и прислать мне shared link через godbolt на ваш сгенеренный код !

Что вынести из лекции:

- Simple CPU model = ALU + Registers
- ALU работает только с регистрами
- Компилятор использует РОН по своему усмотрению - calling convention
- R0-R3 входные аргументы функции, R0 - возвращаемое значение
- R4-R11 - Callee save registers
- APSR регистра - ALU status
- Stack, Stack Pointer
- Link Register

Задание:

- Установить симулятор https://github.com/tomcl/V2releases/releases и написать в нем код который делает тоже самое что и код:

```
int x = 4; // можете не читать из памяти а сразу использовать регистры
int y = 0; // можете не читать из памяти а сразу использовать регистры
while (x != 0) {
   if (x == 2) {
     y = 10;
   }
   --x;
}
```

- Подключиться к вашей плате в режиме дебага через IDE и посмотреть ассемблер, найти вызов любой функции и убедиться что входные параметры функции передаются через регистры в ассемблере - прислать мне скрин отладчика;