Архитектура вычислительных систем

Семинар №4

Целочисленная арифметика.
Одномерные и многомерные массивы.
Примеры использования.

План семинарского занятия

Цель и задачи

- изучение команд, обеспечивающих обработку целочисленных данных
- рассмотреть организацию управления в программах, использующих целочисленную арифметику.
- ознакомиться со способом распределения памяти под целочисленные одномерные и многомерные массивы, организацией доступа к элементам массивов различной размерности.

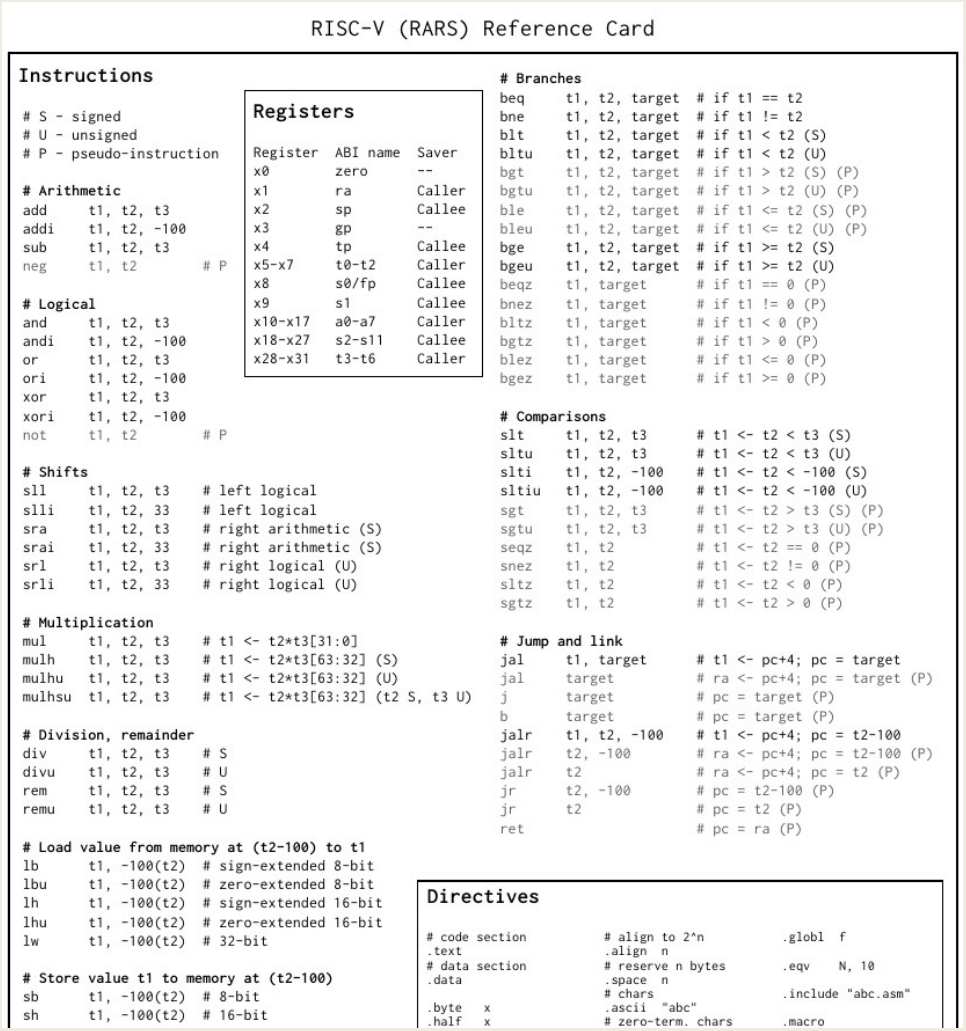
Основные вопросы

1. Обзор команд, обеспечивающих поддержку целочисленной арифметики.
2. Организация в памяти одномерных массивов и методы доступа к ним.
3. Примеры работы с одномерными массивами.

Команды 32-разрядного процессора RISC-V

реализованные в эмуляторе RARS

Таблица с командами RARS



RISC-V (RARS) Reference Card

Instructions

```
# S - signed
# U - unsigned
# P - pseudo-instruction

# Arithmetic
add     t1, t2, t3
addi    t1, t2, -100
sub     t1, t2, t3
neg     t1, t2          # P

# Logical
and     t1, t2, t3
andi    t1, t2, -100
or      t1, t2, t3
ori     t1, t2, -100
xor     t1, t2, t3
xori    t1, t2, -100
not     t1, t2          # P

# Shifts
sll     t1, t2, t3      # left logical
slli    t1, t2, 33      # left logical
sra     t1, t2, t3      # right arithmetic (S)
srai    t1, t2, 33      # right arithmetic (S)
srl     t1, t2, t3      # right logical (U)
srli    t1, t2, 33      # right logical (U)

# Multiplication
mul     t1, t2, t3      # t1 <- t2*t3[31:0]
mulh    t1, t2, t3      # t1 <- t2*t3[63:32] (S)
mulhu   t1, t2, t3      # t1 <- t2*t3[63:32] (U)
mulhsu  t1, t2, t3      # t1 <- t2*t3[63:32] (t2 S, t3 U)

# Division, remainder
div     t1, t2, t3      # S
divu    t1, t2, t3      # U
rem     t1, t2, t3      # S
remu    t1, t2, t3      # U

# Load value from memory at (t2-100) to t1
lb      t1, -100(t2)  # sign-extended 8-bit
lbu     t1, -100(t2)  # zero-extended 8-bit
lh      t1, -100(t2)  # sign-extended 16-bit
lhu     t1, -100(t2)  # zero-extended 16-bit
lw      t1, -100(t2)  # 32-bit

# Store value t1 to memory at (t2-100)
sb      t1, -100(t2)  # 8-bit
sh      t1, -100(t2)  # 16-bit
```

Registers

Register	ABI name	Saver
x0	zero	--
x1	ra	Caller
x2	sp	Callee
x3	gp	--
x4	tp	Callee
x5-x7	t0-t2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10-x17	a0-a7	Caller
x18-x27	s2-s11	Callee
x28-x31	t3-t6	Caller

```
# Branches
beq     t1, t2, target  # if t1 == t2
bne     t1, t2, target  # if t1 != t2
blt     t1, t2, target  # if t1 < t2 (S)
bltu    t1, t2, target  # if t1 < t2 (U)
bgt     t1, t2, target  # if t1 > t2 (S) (P)
bgtu    t1, t2, target  # if t1 > t2 (U) (P)
ble     t1, t2, target  # if t1 <= t2 (S) (P)
bleu    t1, t2, target  # if t1 <= t2 (U) (P)
bge     t1, t2, target  # if t1 >= t2 (S)
bgeu    t1, t2, target  # if t1 >= t2 (U)
beqz    t1, target      # if t1 == 0 (P)
bnez    t1, target      # if t1 != 0 (P)
bltz    t1, target      # if t1 < 0 (P)
bgtz    t1, target      # if t1 > 0 (P)
blez    t1, target      # if t1 <= 0 (P)
bgez    t1, target      # if t1 >= 0 (P)

# Comparisons
slt     t1, t2, t3      # t1 <- t2 < t3 (S)
sltu    t1, t2, t3      # t1 <- t2 < t3 (U)
slti    t1, t2, -100    # t1 <- t2 < -100 (S)
sltiu   t1, t2, -100    # t1 <- t2 < -100 (U)
sgt     t1, t2, t3      # t1 <- t2 > t3 (S) (P)
sgtu    t1, t2, t3      # t1 <- t2 > t3 (U) (P)
seqz    t1, t2          # t1 <- t2 == 0 (P)
snez    t1, t2          # t1 <- t2 != 0 (P)
sltz    t1, t2          # t1 <- t2 < 0 (P)
sgtz    t1, t2          # t1 <- t2 > 0 (P)

# Jump and link
jal     t1, target      # t1 <- pc+4; pc = target
jal     target          # ra <- pc+4; pc = target (P)
j       target          # pc = target (P)
b       target          # pc = target (P)
jalr    t1, t2, -100    # t1 <- pc+4; pc = t2-100
jalr    t2, -100        # ra <- pc+4; pc = t2-100 (P)
jalr    t2              # ra <- pc+4; pc = t2 (P)
jr      t2, -100        # pc = t2-100 (P)
jr      t2              # pc = t2 (P)
ret                     # pc = ra (P)
```

Directives

```
# code section          # align to 2^n          .globl  f
.text                   .align  n
# data section          # reserve n bytes       .eqv    N, 10
.data                   .space  n
                        # chars                 .include "abc.asm"
.byte   x               .ascii  "abc"
.half   x               # zero-term. chars      .macro
```

Отличается от списка команд RISC V

Open RISC-V Reference Card ①

Base Integer Instructions: RV32I and RV64I

Category	Name	Fmt	RV32I Base		+RV64I	
Shifts	Shift Left Logical	R	SLL	rd,rs1,rs2	SLLW	rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI	rd,rs1,shamt	SLLIW	rd,rs1,shamt
	Shift Right Logical	R	SRL	rd,rs1,rs2	SRLW	rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt	SRLIW	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2	SRAW	rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt	SRAIW	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2	ADDW	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm	ADDIW	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2	SUBW	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm		
	Add Upper Imm to PC	U	AUIPC	rd,imm		
Logical	XOR	R	XOR	rd,rs1,rs2		
	XOR Immediate	I	XORI	rd,rs1,imm		
	OR	R	OR	rd,rs1,rs2		
	OR Immediate	I	ORI	rd,rs1,imm		
	AND	R	AND	rd,rs1,rs2		
	AND Immediate	I	ANDI	rd,rs1,imm		
Compare	Set <	R	SLT	rd,rs1,rs2		
	Set < Immediate	I	SLTI	rd,rs1,imm		
	Set < Unsigned	R	SLTU	rd,rs1,rs2		
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm		
Branches	Branch =	B	BEQ	rs1,rs2,imm		
	Branch ≠	B	BNE	rs1,rs2,imm		
	Branch <	B	BLT	rs1,rs2,imm		
	Branch ≥	B	BGE	rs1,rs2,imm		
	Branch < Unsigned	B	BLTU	rs1,rs2,imm		
	Branch ≥ Unsigned	B	BGEU	rs1,rs2,imm		
Jump & Link	J&L	J	JAL	rd,imm		
	Jump & Link Register	I	JALR	rd,rs1,imm		
Synch	Synch thread	I	FENCE			
	Synch Instr & Data	I	FENCE.I			
Environment	CALL	I	ECALL			
	BREAK	I	EBREAK			

Control Status Register (CSR)

Read/Write	I	CSRRW	rd,csr,rs1
Read & Set Bit	I	CSRRS	rd,csr,rs1
Read & Clear Bit	I	CSRRC	rd,csr,rs1
Read/Write Imm	I	CSRRWI	rd,csr,imm
Read & Set Bit Imm	I	CSRRSI	rd,csr,imm
Read & Clear Bit Imm	I	CSRRCI	rd,csr,imm

Loads	Load Byte	I	LB	rd,rs1,imm	
	Load Halfword	I	LH	rd,rs1,imm	
	Load Byte Unsigned	I	LBU	rd,rs1,imm	
	Load Half Unsigned	I	LHU	rd,rs1,imm	
	Load Word	I	LW	rd,rs1,imm	
Stores	Store Byte	S	SB	rs1,rs2,imm	
	Store Halfword	S	SH	rs1,rs2,imm	
	Store Word	S	SW	rs1,rs2,imm	SD rs1,rs2,imm

RV Privileged Instructions

Category	Name	Fmt	RV mnemonic
Trap	Mach-mode trap return	R	MRET
	Supervisor-mode trap return	R	SRET
Interrupt	Wait for Interrupt	R	WFI
MMU	Virtual Memory FENCE	R	SFENCE.VMA rs1,rs2

Examples of the 60 RV Pseudoinstructions

Branch = 0 (BEQ rs,x0,imm)	J	BEQZ rs,imm
Jump (uses JAL x0,imm)	J	J imm
MoVe (uses ADDI rd,rs,0)	R	MV rd,rs
RETurn (uses JALR x0,0,ra)	I	RET

Optional Compressed (16-bit) Instruction Extension: RV32C

Category	Name	Fmt	RVC		RISC-V equivalent	
Loads	Load Word	CL	C.LW	rd',rs1',imm	LW	rd',rs1',imm*4
	Load Word SP	CI	C.LWSP	rd,imm	LW	rd,sp,imm*4
	Float Load Word SP	CL	C.FLW	rd',rs1',imm	FLW	rd',rs1',imm*8
	Float Load Word	CI	C.FLWSP	rd,imm	FLW	rd,sp,imm*8
	Float Load Double	CL	C.FLD	rd',rs1',imm	FLD	rd',rs1',imm*16
	Float Load Double SP	CI	C.FLDSP	rd,imm	FLD	rd,sp,imm*16
Stores	Store Word	CS	C.SW	rs1',rs2',imm	SW	rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP	rs2,imm	SW	rs2,sp,imm*4
	Float Store Word	CS	C.FSW	rs1',rs2',imm	FSW	rs1',rs2',imm*8
	Float Store Word SP	CSS	C.FSWSP	rs2,imm	FSW	rs2,sp,imm*8
	Float Store Double	CS	C.FSD	rs1',rs2',imm	FSD	rs1',rs2',imm*16
	Float Store Double SP	CSS	C.FSDSP	rs2,imm	FSD	rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD	rd,rs1	ADD	rd,rd,rs1
	ADD Immediate	CI	C.ADDI	rd,imm	ADDI	rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP	x0,imm	ADDI	sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN	rd',imm	ADDI	rd',sp,imm*4
	SUB	CR	C.SUB	rd,rs1	SUB	rd,rd,rs1
	AND	CR	C.AND	rd,rs1	AND	rd,rd,rs1
	AND Immediate	CI	C.ANDI	rd,imm	ANDI	rd,rd,imm
	OR	CR	C.OR	rd,rs1	OR	rd,rd,rs1
	eXclusive OR	CR	C.XOR	rd,rs1	AND	rd,rd,rs1
	MoVe	CR	C.MV	rd,rs1	ADD	rd,rs1,x0
	Load Immediate	CI	C.LI	rd,imm	ADDI	rd,x0,imm
	Load Upper Imm	CI	C.LUI	rd,imm	LUI	rd,imm
Shifts	Shift Left Imm	CI	C.SLLI	rd,imm	SLLI	rd,rd,imm
	Shift Right Ari. Imm.	CI	C.SRAI	rd,imm	SRAI	rd,rd,imm
	Shift Right Log. Imm.	CI	C.SRLI	rd,imm	SRLI	rd,rd,imm
Branches	Branch=0	CB	C.BEQZ	rs1',imm	BEQ	rs1',x0,imm
	Branch≠0	CB	C.BNEZ	rs1',imm	BNE	rs1',x0,imm
Jump	Jump	CJ	C.J	imm	JAL	x0,imm
	Jump Register	CR	C.JR	rd,rs1	JALR	x0,rs1,0
Jump & Link	J&L	CJ	C.JAL	imm	JAL	ra,imm
	Jump & Link Register	CR	C.JALR	rs1	JALR	ra,rs1,0
System	Env. BREAK	CI	C.EBREAK		EBREAK	

+RV64I	
LWU	rd,rs1,imm
LD	rd,rs1,imm

Optional Compressed Extention: RV64C

All RV32C (except C.JAL, 4 word loads, 4 word strores) plus:

ADD Word (C.ADDW)	Load Doubleword (C.LD)
ADD Imm. Word (C.ADDIW)	Load Doubleword SP (C.LDSP)
SUBtract Word (C.SUBW)	Store Doubleword (C.SD)
	Store Doubleword SP (C.SDSP)

Open RISC-V Reference Card ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV64M	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MULW	rd,rs1,rs2
	MULtiply High	R	MULH	rd,rs1,rs2		
	MULtiply High Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply High Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIVW	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REMW	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMUW	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV64A	
Load	Load Reserved	R	LR.W	rd,rs1	LR.D	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.D	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.D	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.D	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.D	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.D	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.D	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.D	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.D	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.D	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.D	rd,rs1,rs2

Two Optional Floating-Point Instruction Extensions: RVF & RVD

Category	Name	Fmt	RV32{F|D} (SP,DP Fl. Pt.)		+RV64{F|D}	
Move	Move from Integer	R	FMV.W.X	rd,rs1	FMV.D.X	rd,rs1
	Move to Integer	R	FMV.X.W	rd,rs1	FMV.X.D	rd,rs1
Convert	ConVerT from Int	R	FCVT.{S|D}.W	rd,rs1	FCVT.{S|D}.L	rd,rs1
	ConVerT from Int Unsigned	R	FCVT.{S|D}.WU	rd,rs1	FCVT.{S|D}.LU	rd,rs1
	ConVerT to Int	R	FCVT.W.{S|D}	rd,rs1	FCVT.L.{S|D}	rd,rs1
	ConVerT to Int Unsigned	R	FCVT.WU.{S|D}	rd,rs1	FCVT.LU.{S|D}	rd,rs1
Load	Load	I	FL{W,D}	rd,rs1,imm		
Store	Store	S	FS{W,D}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{S|D}	rd,rs1,rs2		
	SUBtract	R	FSUB.{S|D}	rd,rs1,rs2		
	MULtiply	R	FMUL.{S|D}	rd,rs1,rs2		
	DIVide	R	FDIV.{S|D}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{S|D}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D}	rd,rs1,rs2,rs3		
	Multiply-SUBtract	R	FMSUB.{S|D}	rd,rs1,rs2,rs3		
	Negative Multiply-SUBtract	R	FNMSUB.{S|D}	rd,rs1,rs2,rs3		
	Negative Multiply-ADD	R	FNMADD.{S|D}	rd,rs1,rs2,rs3		
Sign Inject	SiGN source	R	FSGNJ.{S|D}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{S|D}	rd,rs1,rs2		
	MAXimum	R	FMAX.{S|D}	rd,rs1,rs2		
Compare	compare Float =	R	FEQ.{S|D}	rd,rs1,rs2		
	compare Float <	R	FLT.{S|D}	rd,rs1,rs2		
	compare Float ≤	R	FLE.{S|D}	rd,rs1,rs2		
Categorize	CLASSify type	R	FCLASS.{S|D}	rd,rs1		
Configure	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		

Calling Convention

Register	ABI Name	Saver
x0	zero	---
x1	ra	Caller
x2	sp	Callee
x3	gp	---
x4	tp	---
x5-7	t0-2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10-11	a0-1	Caller
x12-17	a2-7	Caller
x18-27	s2-11	Callee
x28-31	t3-t6	Caller
f0-7	ft0-7	Caller
f8-9	fs0-1	Callee
f10-11	fa0-1	Caller
f12-17	fa2-7	Caller
f18-27	fs2-11	Callee
f28-31	ft8-11	Caller
zero	Hardwired zero	
ra	Return address	

Optional Vector Extension: RVV

Name	Fmt	RV32V/R64V	
SET Vector Len.	R	SETVL	rd,rs1
MULtiply High	R	VMULH	rd,rs1,rs2
REMainder	R	VREM	rd,rs1,rs2
Shift Left Log.	R	VSLL	rd,rs1,rs2
Shift Right Log.	R	VSRL	rd,rs1,rs2
Shift R. Arith.	R	VSRA	rd,rs1,rs2
LoaD	I	VLD	rd,rs1,imm
LoaD Strided	R	VLDS	rd,rs1,rs2
LoaD indeXed	R	VLDX	rd,rs1,rs2
STore	S	VST	rd,rs1,imm
STore Strided	R	VSTS	rd,rs1,rs2
STore indeXed	R	VSTX	rd,rs1,rs2
AMO SWAP	R	AMOSWAP	rd,rs1,rs2
AMO ADD	R	AMOADD	rd,rs1,rs2
AMO XOR	R	AMOXOR	rd,rs1,rs2
AMO AND	R	AMOAND	rd,rs1,rs2
AMO OR	R	AMOOR	rd,rs1,rs2
AMO MINimum	R	AMOMIN	rd,rs1,rs2
AMO MAXimum	R	AMOMAX	rd,rs1,rs2
Predicate =	R	VPEQ	rd,rs1,rs2
Predicate ≠	R	VPNE	rd,rs1,rs2
Predicate <	R	VPLT	rd,rs1,rs2
Predicate ≥	R	VPGE	rd,rs1,rs2
Predicate AND	R	VPAND	rd,rs1,rs2
Pred. AND NOT	R	VPANDN	rd,rs1,rs2
Predicate OR	R	VPOR	rd,rs1,rs2
Predicate XOR	R	VPXOR	rd,rs1,rs2
Predicate NOT	R	VPNOT	rd,rs1
Pred. SWAP	R	VPSWAP	rd,rs1
MOVe	R	VMOV	rd,rs1
ConVerT	R	VCVT	rd,rs1
ADD	R	VADD	rd,rs1,rs2
SUBtract	R	VSUB	rd,rs1,rs2
MULtiply	R	VMUL	rd,rs1,rs2
DIVide	R	VDIV	rd,rs1,rs2
SQuare RooT	R	VSQRT	rd,rs1,rs2
Multiply-ADD	R	VFMADD	rd,rs1,rs2,r
Multiply-SUB	R	VFMSUB	rd,rs1,rs2,r
Neg. Mul.-SUB	R	VFNMSUB	rd,rs1,rs2,r
Neg. Mul.-ADD	R	VFNMADD	rd,rs1,rs2,r
SiGN inJect	R	VSGNJ	rd,rs1,rs2
Neg SiGN inJect	R	VSGNJN	rd,rs1,rs2
Xor SiGN inJect	R	VSGNJX	rd,rs1,rs2
MINimum	R	VMIN	rd,rs1,rs2
MAXimum	R	VMAX	rd,rs1,rs2
XOR	R	VXOR	rd,rs1,rs2
OR	R	VOR	rd,rs1,rs2
AND	R	VAND	rd,rs1,rs2
CLASS	R	VCLASS	rd,rs1
SET Data Conf.	R	VSETDCFG	rd,rs1
EXTRACT	R	VEXTRACT	rd,rs1,rs2

Организация в памяти одномерных массивов

…и методы доступа к этим массивам

Работа с одномерными массивами

```
#include <stdio.h>

int array[16];

int main()
{
    fill:
    for(int i = 0; i < 16; ++i) {
        array[i] = i+1;
    }
    printf("--------\n");
    out:
    for(int i = 0; i < 16; ++i) {
        printf("%d\n", array[i]);
    }
    return 0;
}
```

Если нужно обработать *массив* данных, косвенная адресация — единственный способ.
Массив — это адрес в памяти и длина

В ассемблере **предпочтительнее манипулировать адресным пространством** нужной величины, **а не умножать** каждый раз на величину слова.

Основной режим работы - использование косвенной адресации с индексацией относительно начала массива.

Как выделить память под массив:

- выделить пространство требуемого размера;
- при наличии заранее предопределенных значений элементов можно их перечислить;
- **при неизвестном числе элементов можно зарезервировать некоторый *большой кусок памяти*, а количество элементов задавать числом, меньшим выделенного размера, которое (как и в программе на Си) может служить ограничителем цикла при формировании массива;**
- можно выделить память под массив на куче после получения числа элементов в массиве.

Пример 1

Массив слов расписывается последовательными значениями:

```
 1 .data
 2 array:   .space   64
 3 arrend:
 4 .text
 5           la      t0 array
 6           la      t1 arrend
 7           li      t2 1
 8 loop:     sw      t2 (t0)
 9           addi    t2 t2 1
10           addi    t0 t0 4
11           bltu    t0 t1 loop
```

«адресная арифметика» — на каждом проходе цикла для доступа к следующему элементу массива к адресу надо прибавлять размер элемента

Адреса можно сравнивать на > / <, но **сравнение должно быть беззнаковое**: мало ли, в какую область памяти будет загружена программа (в RARS, где адрес загрузки фиксирован)

Пример 2

```
1  .data
2  sep:    .asciz  "--------\n"    # Строка-разделитель (с \n и нулём в конце)
3  .align  2                       # Выравнивание на границу слова
4  array:  .space  64              # 64 байта
5  arrend:                         # Граница массива
6  .text
7          la      t0 array                # Счётчик
8          la      s1 arrend
9          li      t2 1            # Число, которое мы будем записывать в массив
10 fill:   sw      t2 (t0)         # Запись числа по адресу в t0
11         addi    t2 t2 1         # Изменим число
12         addi    t0 t0 4         # Увеличим адрес на размер слова в байтах
13         bltu    t0 s1 fill      # Если не вышли за границу массива
14         la      a0 sep          # Выведем строку-разделитель
15         li      a7 4
16         ecall
17         la      t0 array
18 out:    li      a7 1
19         lw      a0 (t0)         # Выведем очередной элемент массива
20         ecall
21         li      a7 11           # Выведем перевод строки
22         li      a0 10
23         ecall
24         addi    t0 t0 4
25         blt     t0 s1 out
26         li      a7 10           # Останов
27         ecall
```

Пример 2

```
1  .data
2  sep:    .asciz  "--------\n"     # Строка-разделитель (с \n и нулём в конце)
3  .align  2                        # Выравнивание на границу слова
4  array:  .space  64               # 64 байта
5  arrend:                          # Граница массива
6  .text
7          la      t0 array                 # Счётчик
8          la      s1 arrend
9          li      t2 1             # Число, которое мы будем записывать в массив
10 fill:   sw      t2 (t0)          # Запись числа по адресу в t0
11         addi    t2 t2 1          # Изменим число
12         addi    t0 t0 4          # Увеличим адрес на размер слова в байтах
13         bltu    t0 s1 fill       # Если не вышли за границу массива
14         la      a0 sep           # Выведем строку-разделитель
15         li      a7 4
16         ecall
17         la      t0 array
18 out:    li      a7 1
19         lw      a0 (t0)          # Выведем очередной элемент массива
20         ecall
21         li      a7 11            # Выведем перевод строки
22         li      a0 10
23         ecall
24         addi    t0 t0 4
25         blt     t0 s1 out
26         li      a7 10            # Останов
27         ecall
```

ЗАДАНИЕ 1
На примере этой программы модифицируйте код, в котором заполнение осуществляется побайтно

Пример 2

```
1  .data
2  sep:    .asciz  "--------\n"    # Строка-разделитель (с \n и нулём в конце)
3  .align  2                       # Выравнивание на границу слова
4  array:  .space  64              # 64 байта
5  arrend:                         # Граница массива
6  .text
7          la      t0 array                # Счётчик
8          la      s1 arrend
9          li      t2 1            # Число, которое мы будем записывать в массив
10 fill:   sw      t2 (t0)         # Запись числа по адресу в t0
11         addi    t2 t2 1         # Изменим число
12         addi    t0 t0 4         # Увеличим адрес на размер слова в байтах
13         bltu    t0 s1 fill      # Если не вышли за границу массива
14         la      a0 sep          # Выведем строку-разделитель
15         li      a7 4
16         ecall
17         la      t0 array
18 out:    li      a7 1
19         lw      a0 (t0)         # Выведем очередной элемент массива
20         ecall
21         li      a7 11           # Выведем перевод строки
22         li      a0 10
23         ecall
24         addi    t0 t0 4
25         blt     t0 s1 out
26         li      a7 10           # Останов
27         ecall
```

ЗАДАНИЕ 2
На примере этой программы модифицируйте код, в котором заполнение осуществляется полусловами

```c
#include <stdio.h>

int array[16];
int n;

int main()
{
    in:
    printf("n = ? ");
    scanf("%d", &n);
    if(n > 16 || n < 1) {
        printf("n = %d is incorrect!\n", n);
        return 1;
    }
    fill:
    for(int i = 0; i < n; ++i) {
        array[i] = i;
    }
    printf("--------\n");
    out:
    for(int i = 0; i < n; ++i) {
        printf("%d\n", array[i]);
    }
    return 0;
}
```

Задание 3

Модифицировать первую программу так, чтобы вводить в память ограниченное число элементов, заданных максимально допустимым числом.

По примеру следующей программы на Си

Домашнее задание

Оценка до 8 баллов

Разработать программу, осуществляющую суммирование целочисленных элементов со знаком одномерного массива. Количество элементов в массиве может варьироваться от 1 до 10. Числа вводятся с клавиатуры. Необходимо контролировать, чтобы количество вводимых чисел не превышало максимально допустимое.

Суммирование осуществлять после размещения массива в памяти. Сумма вычисляется после ввода заданного количества чисел. Значение полученной суммы также выводится в консоль эмулятора RARS. В случае, когда возникает положительное или отрицательное переполнение, необходимо вывести последнее корректное значение суммы и число просуммированных при этом элементов.

Опционально до +2 баллов

Подсчитать количество четных и нечетных элементов во введенном массиве. Подсчет осуществлять в массиве, который уже расположен в памяти после ввода чисел. Подсчет осуществляется по всем введенным элементам независимо от того происходило переполнение при суммировании или нет.