

05/set/2025



Adição de Instrução no RISC-V RS5



Fernando Gehm Moraes -fernando.moraes@pucrs.br

Willian Analdo Nunes -willian.nunes@edu.pucrs.br

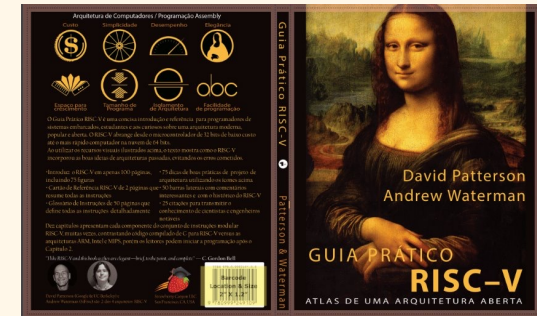
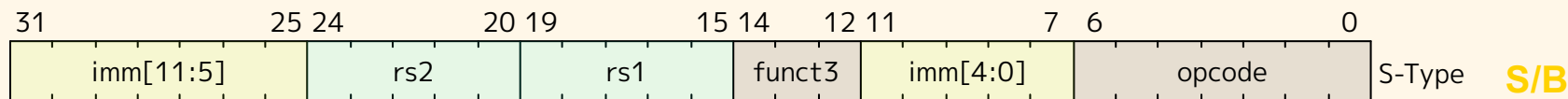
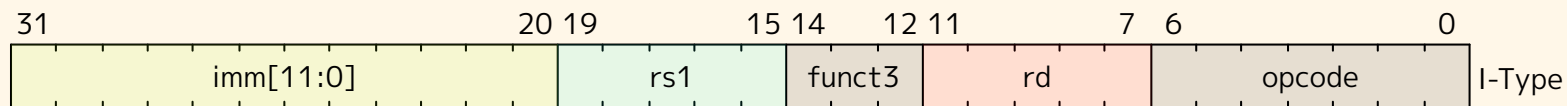
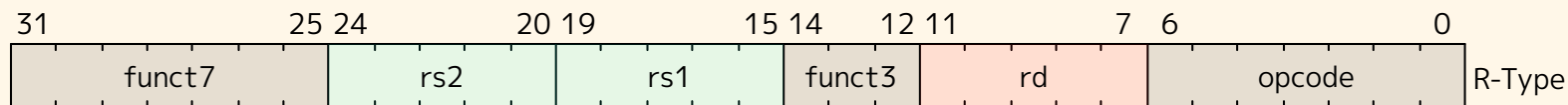
Angelo Dal Zotto -angelo.dalzotto@edu.pucrs.br



Lembrando da ISA

Livro: **GUIA PRÁTICO RISC-V: ATLAS DE UMA ARQUITETURA ABERTA**

Link: <http://riscvbook.com/portuguese/>



MonalSA

Lembrando da ISA

Registers

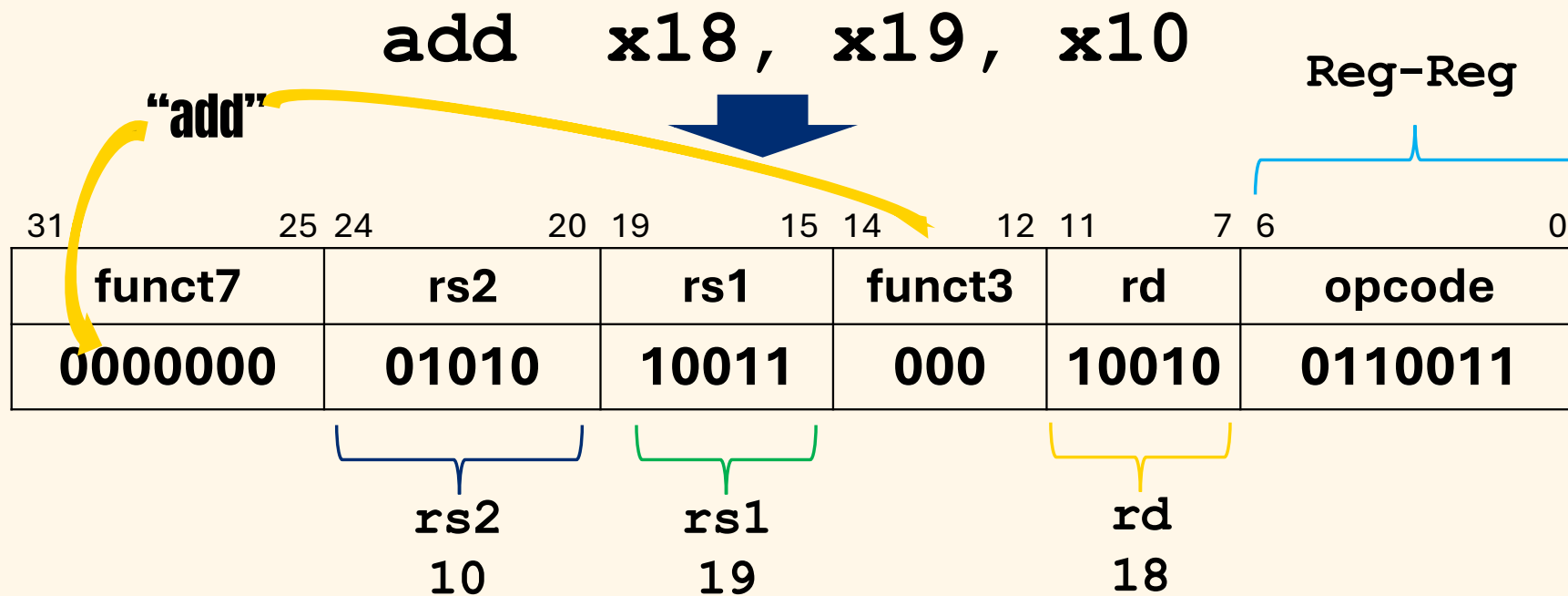
Register	ABI Name	Description	Saver	Register	ABI Name	Description	Saver
x0	zero	Zero constant	-	x10-x11	a0-a1	Fn args/return	Caller
x1	ra	Return Address	Callee	x12-x17	a2-a7	Fn args	Caller
x2	sp	Stack Pointer	Callee	x18-x27	s2-s11	Saved register	Callee
x3	gp	Global Pointer	-	x28-x31	t3-t6	Temporaries	Caller
x4	tp	Thread Pointer	-				
x5-x7	t0-t2	Temporaries	Caller				
x8	s0/fp	Saved/frame pointer	Callee				
x9	s1	Saved register	Callee				

Callee-saved: Os registradores classificados como "Callee" são preservados pela função chamada (callee)

Caller-saved: Os registradores classificados como "Caller" são responsabilidade da função chamadora (caller)

Application Binary Interface (ABI)

Lembrando da ISA - Instruções do tipo R do RV32



Lembrando da ISA - Todas as dez instruções do tipo R do RV32

Oito campos **funct3** para dez instruções

Faz compl.
de 2' do rs2

Extende
o sinal

funct7			funct3		opcode	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	010011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

Diferentes codificações **funct7** & **funct3** selecionam diferentes operações.

Lembrando da ISA - Exercício

Como codificar a instrução: **add x18, x19, x10**

Tabela de consulta:

funct7			funct3		opcode	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	111	rd	0110011	and

- A. 4021 8233_{hex}
- B. 0021 82b3_{hex}
- C. 4021 82b3_{hex}
- D. 0021 8233_{hex}
- E. 0021 8234_{hex}
- F. Outra coisa...

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	

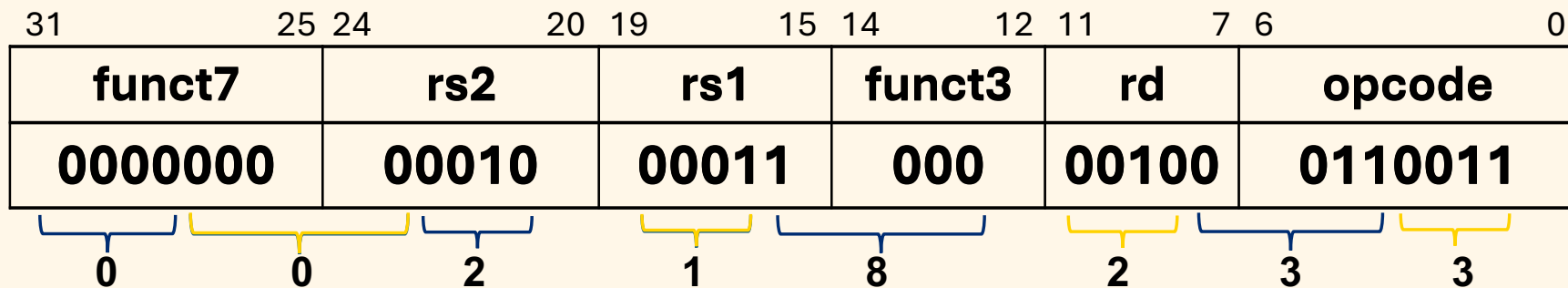
Lembrando da ISA - Solução

Como codificar a instrução: **add x18, x19, x10**

Tabela de consulta:

funct7	rs2	rs1	funct3	rd	opcode	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	111	rd	0110011	and

- A. 4021 8233_{hex}
- B. 0021 82b3_{hex}
- C. 4021 82b3_{hex}
- D. 0021 8233_{hex}**
- E. 0021 8234_{hex}
- F. Outra coisa...



Modificando Arquivos

- Ao longo deste laboratório vamos modificar/criar os seguintes arquivos.

RS5

```
| --app
| | --nor-test (novo)
| | | --Makefile
| | | --src
| | | -- nor-test.S
...
|--rtl
| |--decode.sv
| |--execute.sv
| |--RS5_pkg.sv
...
| |--sim
| | --testbench.sv
```


1

rtl/RS5_pkg.sv

Este arquivo é responsável por guardar os tipos de sinais que são utilizados no processador.

Sendo assim, seguindo o exemplo dos outros tipos de sinais(**typedef enum**), a partir da linha 55, iremos criar um novo tipo, a instrução **NOR**.

Adição da Instrução NOR

```
RS5_pkg.sv x
RS5 > rtl > RS5_pkg.sv
26 package RS5_pkg;
34
55 > typedef enum { ...
115 } iType_e;
```

```
55 typedef enum {
56     NOP,
57     LUI,
58     SRET,
59     MRET,
60     WFI,
61     ECALL,
62     EBREAK,
63     INVALID,
64     ADD,
65     SUB,
66     SLTU,
67     SLT,
68     XOR,
```

```
55 typedef enum {
56     NOR,
57     NOP,
58     LUI,
59     SRET,
60     MRET,
61     WFI,
62     ECALL,
```

Nova
Instrução

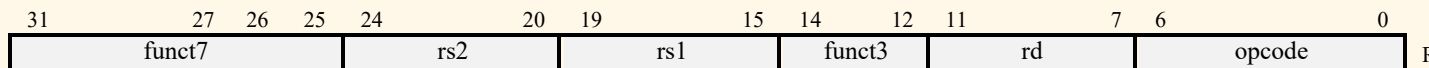
2

rtl/decode.sv

Adição da Instrução NOR

Decodificar a instrução:

- É possível definir qualquer valor de **opcode** contando que não haja conflito com nenhuma outra instrução. Entretanto, como o processador define grupos de instrução baseados no **opcode**, **não** devemos alterá-lo.
- Visto que é uma instrução do tipo **Registrador**, é modificada a região que inclui **funct7** e **funct3**



Sendo o **NOR** uma instrução derivada da **OR**, vamos assumir:

Inst	Name	FMT	Opcode	funct3	funct7
add	ADD	R	0110011	0x0	0x00
sub	SUB	R	0110011	0x0	0x20
xor	XOR	R	0110011	0x4	0x00
or	OR	R	0110011	0x6	0x00

- funct3** igual: **110** (3 bits)
- funct7**: **0100000** (7 bits)

Decodificação do Conjunto I

func7				func3				opcode				
31	25	24	20	19	15	14	12	11	7	6		0
imm[31:12]								rd		0110111		U lui
imm[31:12]								rd		0010111		U auipc
imm[20 10:1 11 19:12]								rd		1101111		J jal
imm[11:0]				rs1		000		rd		1100111		I jalr
imm[12 10:5]		rs2		rs1		000		imm[4:1 11]		1100011		B beq
imm[12 10:5]		rs2		rs1		001		imm[4:1 11]		1100011		B bne
imm[12 10:5]		rs2		rs1		100		imm[4:1 11]		1100011		B blt
imm[12 10:5]		rs2		rs1		101		imm[4:1 11]		1100011		B bge
imm[12 10:5]		rs2		rs1		110		imm[4:1 11]		1100011		B bltu
imm[12 10:5]		rs2		rs1		111		imm[4:1 11]		1100011		B bgeu
imm[11:0]				rs1		000		rd		0000011		I lb
imm[11:0]				rs1		001		rd		0000011		I lh
imm[11:0]				rs1		010		rd		0000011		I lw
imm[11:0]				rs1		100		rd		0000011		I lbu
imm[11:0]				rs1		101		rd		0000011		I lhu
imm[11:5]		rs2		rs1		000		imm[4:0]		0100011		S sb
imm[11:5]		rs2		rs1		001		imm[4:0]		0100011		S sh
imm[11:5]		rs2		rs1		010		imm[4:0]		0100011		S sw
imm[11:0]				rs1		000		rd		0010011		I addi
imm[11:0]				rs1		010		rd		0010011		I slti
imm[11:0]				rs1		011		rd		0010011		I sltiu
imm[11:0]				rs1		100		rd		0010011		I xori
imm[11:0]				rs1		110		rd		0010011		I ori
imm[11:0]				rs1		111		rd		0010011		I andi
0000000		shamt		rs1		001		rd		0010011		I slli
0000000		shamt		rs1		101		rd		0010011		I srli
0100000		shamt		rs1		101		rd		0010011		I srai

0000000			rs2	rs1	000	rd	0110011	R add
0100000			rs2	rs1	000	rd	0110011	R sub
0000000			rs2	rs1	001	rd	0110011	R sll
0000000			rs2	rs1	010	rd	0110011	R slt
0000000			rs2	rs1	011	rd	0110011	R sltu
0000000			rs2	rs1	100	rd	0110011	R xor
0000000			rs2	rs1	101	rd	0110011	R srl
0100000			rs2	rs1	101	rd	0110011	R sra
0000000			rs2	rs1	110	rd	0110011	R or
0000000			rs2	rs1	111	rd	0110011	R and
0000	pred	succ	00000	000	00000	0001111	I fence	
0000	0000	0000	00000	001	00000	0001111	I fence.i	
000000000000			00000	000	00000	1110011	I ecall	
000000000001			00000	000	00000	1110011	I ebreak	
csr			rs1	001	rd	1110011	I csrrw	
csr			rs1	010	rd	1110011	I csrrs	
csr			rs1	011	rd	1110011	I csrrc	
csr			zimm	101	rd	1110011	I csrrwi	
csr			zimm	110	rd	1110011	I csrrsi	
csr			zimm	111	rd	1110011	I csrrci	

Figura 2.3: O mapa de opcode RV32I possui layout de instrução, opcodes, tipo de formato e nomes. (A Tabela 19.2 de [Waterman and Asanović 2017] é a base desta figura.)

Campo opcode: [6:0]

- Bits [1:0] iguais a “11” para instruções não comprimidas
- Em [decode.sv](#), há um “case” para relacionar opcode = instrução.

Decodificação do opcode:

Decodificação do Conjunto I

Na linha **168** do decode.sv, crie o sinal que decodifica **iType_e**, e insira o valor concatenado de **funct7** e **funct3**.

```
iType_e decode_op;
always_comb begin
    unique case ({funct7, funct3}) inside
        10'b0000000000: decode_op = ADD;
        10'b0100000000: decode_op = SUB;
        10'b0000000001: decode_op = SLL;
        10'b0000000010: decode_op = SLT;
        10'b0000000011: decode_op = SLTU;
        10'b0000000100: decode_op = XOR;
        10'b0000000101: decode_op = SRL;
        10'b0100000101: decode_op = SRA;

        10'b0000000110: decode_op = OR;
        10'b0100000110: decode_op = NOR; // bits escolhidos: 0100000 110

        10'b0000000111: decode_op = AND;
        10'b0000001000: decode_op = (MULEXT != MUL_OFF) ? MUL : INVALID;
        ...
        default: decode_op = INVALID;
    endcase
end
```

Instrução
Adicionada

Decodificação do Conjunto I

Decodificação do opcode:

O sinal criado no passo anterior será então identificado como umas das categorias abaixo de operação de instrução.

- RS5/rtl/decode.sv → linha 271

```
always_comb begin
    unique case (opcode)
        5'b01101: instruction_operation = LUI;
        5'b00101: instruction_operation = AUIPC;
        5'b11011: instruction_operation = JAL;
        5'b11001: instruction_operation = JALR;
        5'b11000: instruction_operation = decode_branch;    /* BRANCH */
        5'b00000: instruction_operation = decode_load;     /* LOAD */
        5'b01000: instruction_operation = decode_store;    /* STORE */
        5'b00100: instruction_operation = decode_op_imm;   /* OP-IMM */
        5'b01100: instruction_operation = decode_op;       /* OP */
        5'b00011: instruction_operation = decode_misc_mem; /* MISC-MEM */
        5'b11100: instruction_operation = decode_system;   /* SYSTEM */
        5'b10101: instruction_operation = VEnable ? VECTOR : INVALID; /* OP-V */
        5'b00001: instruction_operation = VEnable ? VLOAD : INVALID; /* LOAD-FP */
        5'b01001: instruction_operation = VEnable ? VSTORE : INVALID; /* STORE-FP */
        5'b01011: instruction_operation = decode_atomic;
        default: instruction_operation = INVALID;
    endcase
end
```

3

rtl/execute.sv

ULA

Neste arquivo, serão executadas as instruções decodificadas na parte anterior.

- Sendo assim, devemos primeiro acrescentar um sinal interno **nor_result** (linha 139)

Sinal Criado

```
logic [31:0] sum_result;  
logic [31:0] sum2_result;  
logic [31:0] and_result;  
logic [31:0] or_result;  
logic [31:0] nor_result;  
logic [31:0] xor_result;
```

- Em seguida, atribuir a este sinal a operação **NOR** (linha 180)

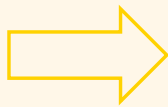
Sinal Criado

```
assign or_result    = first_operand | second_operand_i;  
assign nor_result   = ~(first_operand | second_operand_i);  
assign xor_result   = first_operand ^ second_operand_i;
```


rtl/execute.sv

- Atribuir o resultado no Demux (linha 640)

```
always_comb begin
    unique case (instruction_operation_i)
        CSRRW, CSRRS, CSRRRC,
        CSRRWI, CSRRSI, CSRRCI:    result = csr_data_read_i;
        JAL, JALR, SUB:           result = sum2_result;
        SLT:                       result = {31'b0, less_than};
        SLTU:                      result = {31'b0, less_than_unsigned};
        XOR:                       result = xor_result;
        OR:                        result = or_result;
        NOR:                       result = nor_result;
        AND:                       result = and_result;
```





Código Binário

Criaremos uma aplicação exemplo que usa **OR** ao invés **de NOR** em assembly.

- Para isto, crie uma pasta de trabalho nor-test a partir do fibonacci-asm:
- Sugestão, no terminal dê:

```
app % cp -r fibonacci-asm  
nor-test  
cd nor-test/src  
mv fibonacci.S nor-test.S
```

Copie o código ao lado no arquivo criado.

Ele possui um laço que percorre N elementos fazendo a soma de $\text{vet}[i] + \text{vet}[i+1]$:

```
i = 0;
while (i < N) {
    vet_i = vet[i];
    vet_i1 = vet[i + 1];
    result = vet_i | vet_i1;
    i += 2;
}
```

```
.section .init
.global _start
```

```
_start:
    csrw mstatus, zero
    csrw mie, zero
    csrw mip, zero
    li sp, MEM_SIZE
    jal main
```

```
.end:
    sw t0, 0(t4)
    li t0, 0x80000000
    sw zero, 0(t0) # End simulation
    j .end
```

```
.section .text
```

```
main:
    li t0, 0 # t0 = i (índice atual)
    la t1, N # Endereço de N
    lw t1, 0(t1) # t1 = N (tamanho do
vetor)
    la t2, vet # t2 = endereço inicial
do vetor
```

```
loop:
```

```
bge t0, t1, fim # Se i >= N, sai do laço
slli t3, t0, 2 # t3 = i * 4 (calcular offset
em bytes)
add t3, t3, t2 # t2 = endereço de vet[i]
lw a2, 0(t3) # a2 = vet[i]
lw a3, 4(t3) # a3 = vet[i+1]

or a4, a2, a3 # a4 = vet[i] or vet[i+1]
```

```
addi t0, t0, 2 # t0 = i + 2
li t4, 0x80002000 # imprime a soma
sw a4, 0(t4)
j loop
```

```
fim:
```

```
ret
```

```
.data
```

```
N: .word 10 #Número de elementos do vetor
vet: .word 0xFFFFFFFF, 0xFFFFFFFF, 0x0, 0x0,
0xAAAAAAAA, 0x55555555, 0x43, 0x34, 0xAAA,
0x775
```



Na pasta *nor-test*:

make

Assembling src/nor-test.S...

Linking nor-test.elf...

/soft64/util/.....elf has a LOAD segment with RWX permissions

Generating nor-test.bin...

Generating nor-test.lst...

Na pasta *sim*:

make

-- RUN -----

4294967295

0

4294967295

119 → **0x43** | **0x34 = 0x77 = 119**

4095

980 END OF SIMULATION

- testbench.sv:288: Verilog \$finish

- Simulation Report: Verilator 5.024 2024-04-05

-- DONE -----

Compilando e Simulando



Na pasta *nor-test*:

```
main:
    li t0, 0
    la t1, N
    lw t1, 0(t1)
    la t2, vet

loop:
    bge t0, t1, fim

    slli t3, t0, 2
    add t3, t3, t2
    lw a2, 0(t3)
    lw a3, 4(t3)

    or a4, a2, a3

    addi t0, t0, 2

    li t4, 0x80002000
    sw a4, 0(t4)

    j loop
fim:
    ret
```

Modificando o binário para uso da NOR

Abrir o arquivo *nor-test.lst*, e identificar a instrução **OR**.

```
00000020 <main>:
    20: 00000293          li    t0,0
    24: 05c00313          li    t1,92
    28: 00032303          lw    t1,0(t1)
    2c: 06000393          li    t2,96

00000030 <loop>:
    30: 0262d463          bge   t0,t1,58 <fim>
    34: 00229e13          slli  t3,t0,0x2
    38: 007e0e33          add   t3,t3,t2
    3c: 000e2603          lw    a2,0(t3)
    40: 004e2683          lw    a3,4(t3)
    44: 00d66733          or    a4,a2,a3
    48: 00228293          addi  t0,t0,2
    4c: 80002eb7          lui   t4,0x80002
    50: 00eea023          sw    a4,0(t4) # 80002000
    54: fddff06f          j     30 <loop>

00000058 <fim>:
    58: 00008067          ret
```

Inst	Name	FMT	Opcode	funct3	funct7	Description
or	OR	R	0110011	0x6	0x00	rd = rs1 rs2

“Desmontando” o Binário

44: 00d66733 or a4,a2,a3

0x00d66733 = 0000 0000 1101 0110 0110 0111 0011 0011

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
0000 000	0 1101	0110 0	110	0111 0	011 0011	
	13 (a3)	12 (a2)		14 (a4)		


- Escolhemos para NOR:

- **funct3** igual: **110** (3 bits)

44: **4**0d66733 nor a4, a2, a3

- **funct7** : **0100000** (7 bits).

Register	ABI Name
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5-x7	t0-t2
x8	s0/fp
x9	s1
x10-x11	a0-a1
x12-x17	a2-a7
x18-x27	s2-s11
x28-x31	t3-t6

	00	01	02	03
00000000	73	10	00	30
00000004	73	10	40	30
00000008	73	10	40	34
0000000C	37	01	01	00
00000010	EF	00	00	01
00000014	B7	02	00	80
00000018	23	A0	02	00
0000001C	6F	F0	9F	FF
00000020	93	02	00	00
00000024	13	03	C0	05
00000028	03	23	03	00
0000002C	93	03	00	06
00000030	63	D4	62	02
00000034	13	9E	22	00
00000038	33	0E	7E	00
0000003C	03	26	0E	00
00000040	83	26	4E	00
00000044	33	67	D6	00
00000048	93	82	22	00
0000004C	B7	2E	00	80
00000050	23	A0	EE	00
00000054	6F	F0	DF	FD

Modificando o Binário

- 1) No **Visual Studio** (code) instalar a extensão **HEX EDITOR** (ou usar qualquer outro editor hexadecimal)
- 2) Abrir o arquivo binário `nor-test.bin`
 - Note que os bytes estão de 00 a 03 (“invertidos”)
- 3) Modificar:

DE :

44: 00d66733 or a4, a2, a3

PARA

44: 40d66733 nor a4, a2, a3

Simulando e Verificando

-- RUN -----

0

4294967295

0

4294967176

4294963200

```
#          980 END OF SIMULATION
- testbench.sv:288: Verilog $finish
- Simulation Report: Verilator 5.024
2024-04-05
- Verilator: $finish at 985ns; walltime 0.002 s; speed
0.000 s/s
- Verilator: cpu 0.000 s on 1 threads; allocated 217 MB
```

Inputs		Output
A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

```
-- DONE 0xFFFFFFFF nor 0xFFFFFFFF = 1 nor 1 = 0
         0x0 nor 0x0 = 0 nor 0 = 1
         0xAAAAAAAA nor 0x55555555 = 0 nor 1 = 0
         0x43 nor 0x34 = FFFF FF88 = 4294967176
         0xAAA nor 0x775 = FFFF F000 = 4294963200
```

Simulando e Verificando

Simulando no QUESTA, observando o caso:

$$0x43 \text{ nor } 0x34 = \text{FFFF FF88} = 4294967176$$

	Msgs								
pc_i	32'h00000...	000000...	0000003c	00000040	00000044	00000048	0000004c	00000050	00000054
instruction_operat...	NOP	ADD	LW	NOP	NOR	ADD	LUI	SW	JAL
rs1_data_i	32'h00000...	000000...	00000078	00000043	00000006	00000000	80002000	00000000	00000014
rs2_data_i	32'h00010...	000000...	00000000	0000007c	00000034	00010000	00000000	ffffff88	80002000
second_operand_i	32'h00000...	000000...	00000000	00000004	0000007c	00000034	00000002	80002000	00000000
rd_i	5'h1c	1c	0c	0d	0e	05	1d	00	
result	32'h00000...	00000078	0000007c	000000bf	ffffff88	00000008	80002000	00000058	00000014
regfile[14]	32'hffff000	00000000					ffffff88		
regfile[13]	32'h00000...	55555555			00000034				
regfile[12]	32'h00000...	aaaaaaaa		00000043					
regfile	32'h00000...	000000...	00000000...	00000000...	00000000...	00000000 80...	00000000...	00000000 00000000 80002000 0000...	

- Observa o hazard após o segundo lw, com a inserção de um NOP:

lw

lw

or

a2, 0(t3)
a3, 4(t3)
a4, a2, a3

 (não há como fazer forward de um load)

Soma Saturada de Bytes em RISCV com Extensão P

A **Extensão P (Packed SIMD)** no conjunto de instruções RISC-V adiciona suporte para operações **SIMD (Single Instruction, Multiple Data)**, que são utilizadas em processamento de sinais digitais, gráficos e aplicações que exigem alto desempenho. Essa extensão introduz instruções para operar sobre subcampos dos registradores, permitindo realizar várias **operações em paralelo** dentro de um único registrador.

Neste exercício, será implementada uma operação de **soma saturada de bytes** utilizando o formato de 32 bits dos registradores do RISCV. Para a implementar esta operação, que consiste em somar os bytes de dois operandos, armazenados em registradores de 32 bits, e aplicar saturação no caso de overflow, será necessário realizar os seguintes passos (etapa de execução da instrução):

- 1. **Operação Byte-a-Byte:** Cada byte dos operandos é somado individualmente. A soma de cada byte é tratada como uma **operação de 9 bits** para capturar possíveis valores além de 255 (overflow)
- 2. **Saturação:** Se o resultado de qualquer byte ultrapassar 255, ele será ajustado para o valor máximo permitido (255, ou 0xFF)
- 3. **Concatenar Resultados:** Os resultados ajustados são concatenados para formar a palavra de 32 bits final.

Exercício

7.143. UKADD8 (SIMD 8-bit Unsigned Saturating Addition)											
Type: SIMD											
Format:											
31	25	24	20	19	15	14	12	11	7	6	0
UKADD8		Rs2		Rs1		000		Rd		OP-P	
0011100										1110111	

Exercício

Objetivo: modificar o processador RS5 para implementar a soma saturada de bytes, baseada nos princípios da extensão P. Utilizar os passos da inserção da instrução **NOR**

➤ **Criar a Instrução:** ADD8S, usando funct7: **0001000** e funct3: **000**

Exemplo 1

operand1 = 0x12345678
operand2 = 0x87654321

Passo 1: Soma byte-a-byte com overflow:

Byte 0: $0x78 + 0x21 = 0x99$
Byte 1: $0x56 + 0x43 = 0x99$
Byte 2: $0x34 + 0x65 = 0x99$
Byte 3: $0x12 + 0x87 = 0x99$

Passo 2: Não há saturação, pois nenhum byte excede 0xFF

Resultado:

Saída: 0x99999999

Exemplo 2

operand1 = 0xFF123456
operand2 = 0x01FF0000

Passo 1: Soma byte-a-byte com overflow:

Byte 0: $0x56 + 0x00 = 0x56$
Byte 1: $0x34 + 0x00 = 0x34$
Byte 2: $0x12 + 0xFF = 0x111$ (overflow)
Byte 3: $0xFF + 0x01 = 0x100$ (overflow)

Passo 2: Aplicar saturação:

Byte 2 e Byte 3 excederam 0xFF e são ajustados para 0xFF.

Resultado:

Saída: 0xFFFF3456

1. Usar o mesmo código da NOR:

```
cp -r nor-test/ add8s
cd add8s/src/
mv nor-test.S add8s.S
```

2. Alterar a área de dados:

```
.data
N: .word 10 # Numero de elementos do vetor
vet: .word 0x12345678, 0x87654321, 0xFF123456,
0x01FF0000, 0xAAAAAAAA, 0x56555453, 0xABCDEFFF,
0x1, 0xBAFECAFE, 0xCAFE1234
```

3. Ajustar código da instrução **nor** para usar **soma (add)**

```
24 loop:
25     bge t0, t1, fim # Se i >= N, sai do laço
26     slli t3, t0, 2 # t3 = i * 4 (calcular offset em bytes)
27     add t3, t3, t2 # t2 = endereço de vet[i]
28     lw a2, 0(t3) # a2 = vet[i]
29     lw a3, 4(t3) # a3 = vet[i+1]
30
31     or a4, a2, a3 # a4 = vet[i] or vet[i+1]
32
33     addi t0, t0, 2 # t0 = i + 2
34     li t4, 0x80002000 # imprime a soma
35
36     sw a4, 0(t4)
37     j loop
```

```
23 loop:
24     bge t0, t1, fim # Se i >= N, sai do laço
25     slli t3, t0, 2 # t3 = i * 4 (calcular offset em bytes)
26     add t3, t3, t2 # t2 = endereço de vet[i]
27     lw a2, 0(t3) # a2 = vet[i]
28     lw a3, 4(t3) # a3 = vet[i+1]
29
30     add a4, a2, a3 # a4 = vet[i] or vet[i+1]
31
32     addi t0, t0, 2 # t0 = i + 2
33     li t4, 0x80002000 # imprime a soma
34     sw a4, 0(t4)
35     j loop
```


Simulando com Soma

RS5/sim/testbench.sv:

Modificar trecho abaixo, usando **8X**, para imprimir valores em hexa:

```
else if (mem_address == 32'h80002000 && mem_write_enable != '0) begin
    $write(    "%8x\n",mem_data_write);
    $fwrite(fd, "%8x\n",mem_data_write);
    $fflush();
end
```

Resultado simulação com verilator:

-- RUN -----

99999999

01113456

00fffffd

abcdf000

85fcdd32

980 END OF SIMULATION

- testbench.sv:289: Verilog \$finish

-

-- DONE -----

Somas realizadas:

0x12345678	0xFF123456	0xAAAAAAAA	0xABCDEFFF	0xBAFECAFE
0x87654321	0x01FF0000	0x56555453	0x00000001	0xCAFE1234
-----	-----	-----	-----	-----
0x99999999	0x01113456	0x00FFFEFD	0xABCDF000	0x85FCDD32

Abrindo o binário e Ist

Modificar o binário para corresponder ao ADD8S:

44: 33 07 D6 00 add a4, a2, a3

0x00D60733 = 0000 0000 1101 0110 0000 0111 0011 0011

Add a4,a2,13

.data

0x12345678
0x87654321
0xFF123456
0x01FF0000
0xAAAAAAAA
0x56555453
0xABCDEF
0x1
0xBAFECAFE
0xCAFE1234

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
0000 000	0 1101	0110 0	000	0111 0	011 0011	

Sugestão:

func7 = 0001 000 (altera)

func3 = 000 (mantém)

Simulando com Soma Saturada

Resultado simulação com verilator:

```
-- RUN -----  
99999999  
ffff3456  
fffffeff  
abcdefff  
ffffdcff  
# 980 END OF SIMULATION  
-- DONE -----
```

Somas realizadas:

0x12 34 56 78	0xFF 12 34 56	0xAA AA AA AA	0xAB CD EF FF	0xBA FE CA FE
0x87 65 43 21	0x01 FF 00 00	0x56 55 54 53	0x00 00 00 01	0xCA FE 12 34
-----	-----	-----	-----	-----
0x99 99 99 99	0xFF FF 34 56	0xFF FF FE FD	0xAB CD EF FF	0xFF FF DC FF

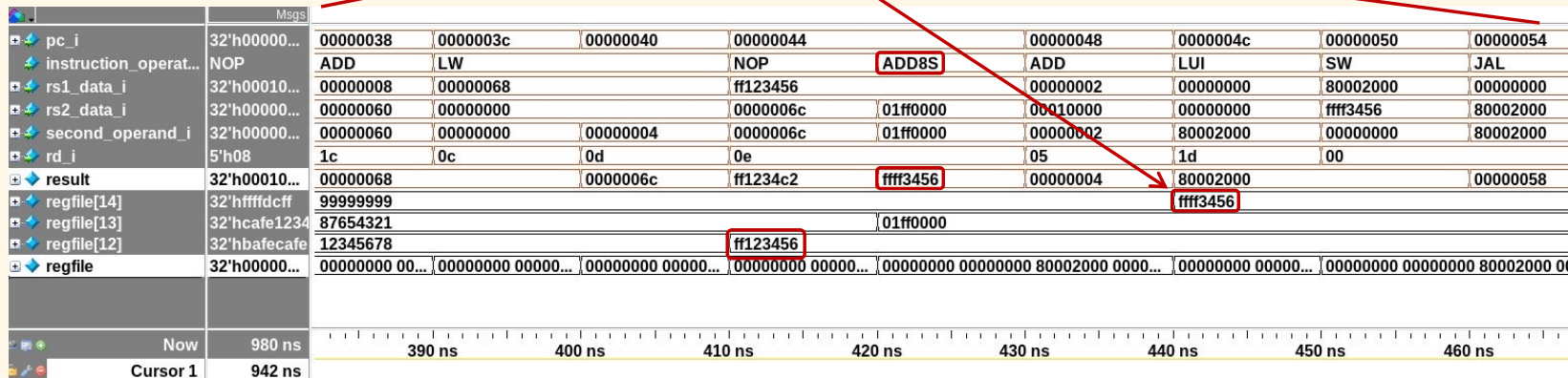
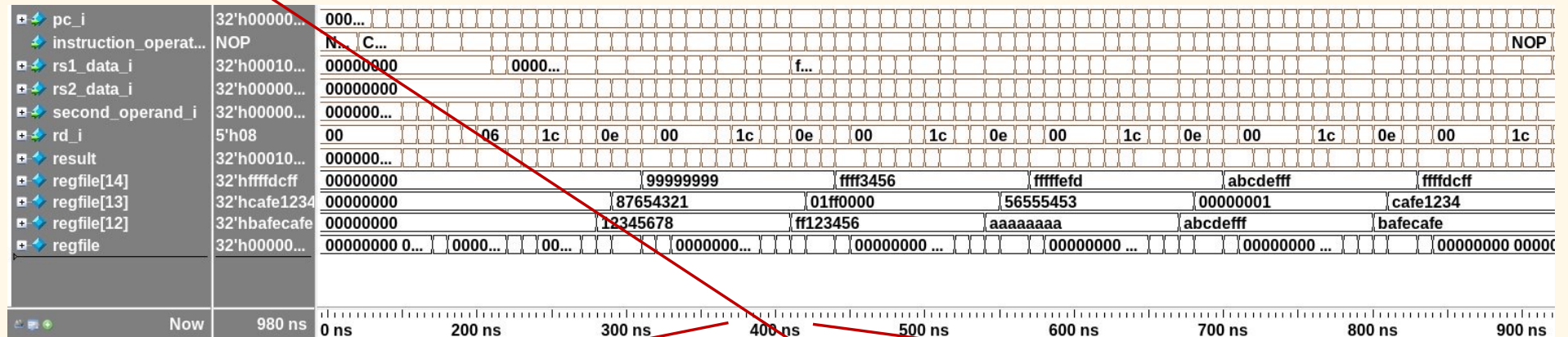


Não é por saturação

0x12 34 56 78 0xFF 12 34 56 0xAA AA AA AA 0xAB CD EF FF 0xBA FE CA FE
 0x87 65 43 21 0x01 FF 00 00 0x56 55 54 53 0x00 00 00 01 0xCA FE 12 34

0x99 99 99 99 0xFF FF 34 56 0xFF FF FE FD 0xAB CD EF FF 0xFF FF DC FF

Simulando com Soma Saturada



Exemplo de implementação Instrução ADD8S

RS5/rtl/execute.sv

```
179  /* "Muxable" operators */
180  assign sum2_result      = first_operand + sum2_opB;
181  assign and_result       = first_operand & second_operand_i;
182  assign or_result        = first_operand | second_operand_i;
183  assign xor_result       = first_operand ^ second_operand_i;
184  assign nor_result       = ~(first_operand | second_operand_i);
185
186  logic [8:0] aux1, aux2, aux3, aux4;
187  assign aux1 = ({1'b0,first_operand[31:24]}) + ({1'b0,second_operand_i[31:24]});
188  assign aux2 = ({1'b0,first_operand[23:16]}) + ({1'b0,second_operand_i[23:16]});
189  assign aux3 = ({1'b0,first_operand[15:8]}) + ({1'b0,second_operand_i[15:8]});
190  assign aux4 = ({1'b0,first_operand[7:0]}) + ({1'b0,second_operand_i[7:0]});
191
192
193  assign adds8_result     =  {(aux1[8] == 1'b1 ? 8'hff : aux1[7:0]),
194                             (aux2[8] == 1'b1 ? 8'hff : aux2[7:0]),
195                             (aux3[8] == 1'b1 ? 8'hff : aux3[7:0]),
196                             (aux4[8] == 1'b1 ? 8'hff : aux4[7:0])};
```

Cada variável **aux**(9 bits),
guarda uma soma de byte.

Variáveis **aux** são concatenadas em **adds8_result**.

Em cada etapa contendo uma verificação (if inline) que observa se ocorreu **overflow**, ou seja, se o bit 9 (mais significativo) é **1**.

Caso seja **1**, este byte receberá **8'hff**, caso não haja **overflow**, será enviado o resultado guardado em **aux**.

<solução não recomendada – só para testes>

Segunda Solução do exercício

1. Copiar a aplicação **fibonacci-c-newlib** para uma nova pasta
2. Usar uma instrução existente, mas que a aplicação não vai usar, exemplo, **sra**

```
#include <stdio.h>
#define N 10

int main() {
    unsigned int a, b, sum=0, i = 0;
    int vet[N] = { 0x12345678, 0x87654321, 0xFF123456, 0x01FF0000, 0xAAAAAAAA,
                  0x56555453, 0xABCDEFFF, 0x1, 0xBAFECAFE, 0xCAFE1234 };

    while (i < N) {
        a = vet[i];
        b = vet[i+1];
        __asm__ (
            "sra %0, %1, %2\n"
            : "+r"(sum), "+r"(a), "+r"(b)
            :
            :
        );
        i += 2;
        printf("a = %8x, b = %8x, Result: %8x\n", a, b, sum);
    }
    return 0;
}
```

Instruções do Tipo R

Inst	Name
add	ADD
sub	SUB
xor	XOR
or	OR
and	AND
sll	Shift Left Logical
srl	Shift Right Logical
sra	Shift Right Arith
slt	Set Less Than
sltu	Set Less Than (U)

Segunda Solução do exercício

3. Altere a decodificação de instruções

```
always_comb begin
  unique case ({funct7, funct3}) inside
    10'b0000000000: decode_op = ADD;
    10'b0001000000: decode_op = ADD8S; // new
  instruction
    10'b0100000000: decode_op = SUB;
    10'b0000000001: decode_op = SLL;
    10'b0000000010: decode_op = SLT;
    10'b0000000011: decode_op = SLTU;
    10'b0000000100: decode_op = XOR;
    10'b0000000101: decode_op = SRL;
    10'b0100000101: decode_op = ADD8S; // trapaça -
    depois voltar SRA;
    10'b0000000110: decode_op = OR;
    ...
  endcase
end
```

Na pasta app: make

Compiling src/teste-sums.c...

Compiling ../common/newlib.c...

Assembling ../common/crt0.S...

Linking teste-sums.elf...

/soft64/util/riscv64-elf/14.2.0/lib/gcc/riscv64-

elf/14.2.0/../../../../riscv64-elf/bin/ld: warning: teste-sums.elf

has a LOAD segment with RWX permissions

Generating teste-sums.bin...

Generating teste-sums.lst...

Na pasta sim (supondo verilator): make

-- RUN -----

a = 12345678, b = 87654321, Result: 99999999

a = ff123456, b = 1ff0000, Result: ffff3456

a = aaaaaaaa, b = 56555453, Result: fffffefd

a = abcdefff, b = 1, Result: abcdefff

a = bafecafe, b = cafe1234, Result: ffffdcff

369910 END OF SIMULATION

- testbench.sv:325: Verilog \$finish

- Simulation Report: Verilator 5.024 2024-04-05

- Verilator: \$finish at 370us; walltime 0.037 s; speed 0.000

Funciona 😊



Fim tutorial 2

RS5

Fernando Gehm Moraes -fernando.moraes@pucrs.br

Willian Analdo Nunes -willian.nunes@edu.pucrs.br

Angelo Dal Zotto -angelo.dalzotto@edu.pucrs.br