Trabalho 2

Simulador RISC-V

Gabriel da Silva Corvino Nogueira
18/0113330

Dep. Ciência da Computação – Universidade de Brasília (UnB)
CIC0099 - Organização e Arquitetura de Computadores
Brasília - Distrito Federal
Fevereiro
2022

1. Apresentação do Problema

Uma **arquitetura do conjunto de instruções** (ISA) é as parte do modelo abstrato de um computador que define como o seu processador é controlado pelo *software*. Uma ISA age como interface entre *hardware* e *software*, especificando o que o processador é capaz de fazer, assim como a maneira como as instruções devem ser executadas[ARM 2022].

O **RISC-V** é uma ISA recente (desenvolvida na ultima década) que nasceu com o propósito de se tornar uma arquitetura de conjunto de instruções universal, aberta e livre dos caprichos de qualquer empresa [Patterson and Waterman 2019].

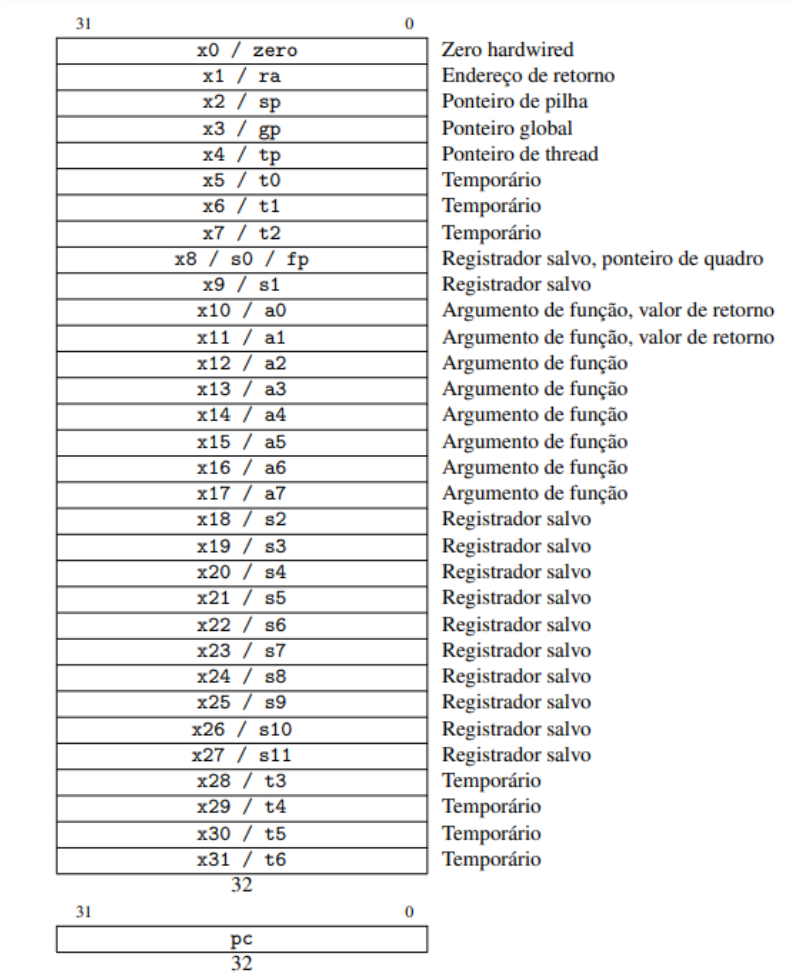
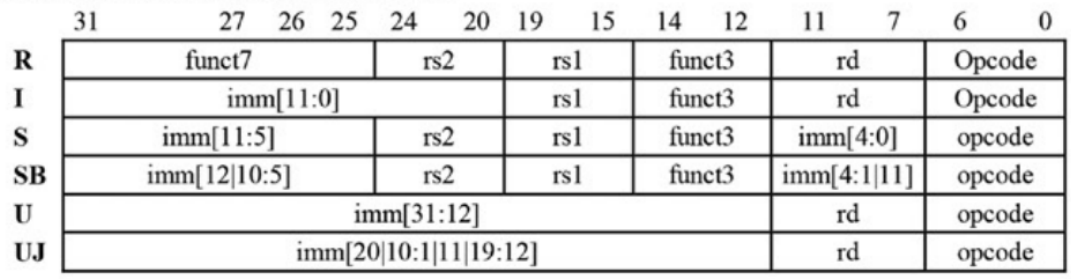


Figura 1. Registradores presentes na arquitetura RISC-V juntamente com seus nomes convencionais e descrição de uso.

No âmbito do RISC-V, existe a arquitetura RV32I, uma variante dedicada a lidar apenas com números inteiros de 32 *bits*. Como apresentado na Figura 1, o RV32I conta com 33 registradores de 32 *bits*, sendo os registradores `x1-x31` destinados ao uso geral, enquanto o registrador `x0` está diretamente ligado à constante 0. Por fim, há também o registrador `pc`, responsável por armazenar o endereço da instrução que está sendo executada.



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

Figura 2. Formatos base para as instruções do RISC-V.

Assim como os registradores, uma instrução da RV32I é também constituída de 32 *bits*. Cada instrução suportada pode ser representada por um dos formatos presentes na Figura 2. Dessa forma, Cada formato é projetado para lidar com um tipo de instrução, como listado a seguir:

- **Tipo R:** operações lógico-aritméticas;
- **Tipo I:** operações com dados imediatos;
- **Tipo S:** operações de armazenamento;
- **Tipo SB:** operações de salto condicional;
- **Tipo U:** operações com dados imediatos grandes;
- **Tipo UJ:** operações de salto incondicional

Neste trabalho, o objetivo é implementar um simulador da arquitetura RV32I em linguagem de alto nível (C++). Os programas binários executados pelo simulador foram gerados pelo montador RARS, juntamente com os respectivos dados. Dessa forma, o simulador deve ler arquivos binários contendo o segmento de código o segmento de dados para sua memória e executa-los.

2. Instruções Implementadas

Nesta seção será feita uma descrição das instruções implementadas no trabalho. Para cada instrução será apresentado seu formato em linguagem de montagem, seu tipo e sua descrição.

add
- **Tipo:** R
- **Assembly:** `add rd, rs1, rs2`
- **Descrição:** soma os valores contidos nos registradores `rs1` e `rs2` e armazena o resultado no registrador `rd`.

addi
- **Tipo:** I
- **Assembly:** `addi rd, rs1, Imm12`
- **Descrição:** Soma o valor contido em `rs1` com um imediato de 12 bits com sinal e armazena o resultado em `rd`.

and
- **Tipo:** R
- **Assembly:** `and rd, rs1, rs2`
- **Descrição:** Realiza a operação lógica AND bit a bit entre os valores contidos em `rs1` e `rs2` e armazena o resultado em `rd`.

andi

- **Tipo:** I
- **Assembly:** `andi rd, rs1, Imm12`
- **Descrição:** Realiza a operação lógica AND bit a bit entre o valor contido em `rs1` e um imediato de 12 bits com o sinal estendido. Armazena o resultado em `rd`.

auipc

- **Tipo:** U
- **Assembly:** `auipc rd, Imm20`
- **Descrição:** Instrução utilizada para gerar um endereço relativo ao endereço armazenado no registrador `pc`. Para esse fim, o imediato de 20 bits representa os 20 bits mais significativos de um *offset* de 32 bits cujos 12 bits menos significativos são preenchidos com o valor 0. Dessa forma o *offset* é somado ao valor contido no registrador `pc` e o resultado é armazenado em `rd`.

beq

- **Tipo:** SB
- **Assembly:** `beq rs1, rs2, Imm13`
- **Descrição:** Compara os conteúdos de `rs1` e `rs2`, caso eles sejam iguais, `pc` será incrementado pelo valor do imediato de 13 bits com sinal.

bne

- **Tipo:** SB
- **Assembly:** `bne rs1, rs2, Imm13`
- **Descrição:** Compara os conteúdos de `rs1` e `rs2`, caso eles sejam diferentes, `pc` será incrementado pelo valor do imediato de 13 bits com sinal.

bge

- **Tipo:** SB
- **Assembly:** `bge rs1, rs2, Imm13`
- **Descrição:** Compara os conteúdos de `rs1` e `rs2`, caso o conteúdo de `rs1` seja maior ou igual ao conteúdo de `rs2`, `pc` será incrementado pelo valor do imediato de 13 bits com sinal.

bgeu

- **Tipo:** SB
- **Assembly:** `bgeu rs1, rs2, Imm13`
- **Descrição:** Compara os conteúdos de `rs1` e `rs2` interpretando-os como números sem sinal, caso o conteúdo de `rs1` seja maior ou igual ao conteúdo de `rs2`, `pc` será incrementado pelo valor do imediato de 13 bits com sinal.

blt

- **Tipo:** SB
- **Assembly:** `blt rs1, rs2, Imm13`
- **Descrição:** Compara os conteúdos de `rs1` e `rs2`, caso o conteúdo de `rs1` seja menos que o conteúdo de `rs2`, `pc` será incrementado pelo valor do imediato de 13 bits com sinal.

bltu

- **Tipo:** SB
- **Assembly:** `bltu rs1, rs2, Imm13`
- **Descrição:** Compara os conteúdos de `rs1` e `rs2` interpretando-os como números sim sinal, caso o conteúdo de `rs1` seja menos que o conteúdo de `rs2`, `pc` será incrementado pelo valor do imediato de 13 bits com sinal.

jal

- **Tipo:** UJ
- **Assembly:** `jal rd, Imm21`
- **Descrição:** Atribui o valor de `pc` ao registrador `rd`. Incrementa `pc` com o valor do imediato de 21 bits com sinal.

jalr

- **Tipo:** I
- **Assembly:** `jalr rd, rs1, Imm12`
- **Descrição:** Atribui o valor de `pc` ao registrador `rd`. Soma o valor de contido em `rs1` com um imediato de 12 bits com sinal e armazena o resultado em `pc`.

lb

- **Tipo:** I
- **Assembly:** `lb rd, Imm12(rs1)`
- **Descrição:** Atribui a `rd` o valor do byte com extensão de sinal localizado no endereço de memória contido em `rs1` deslocado por um imediato de 12 bits com sinal.

lbu

- **Tipo:** I
- **Assembly:** `lbu rd, Imm12(rs1)`
- **Descrição:** Atribui a `rd` o valor do byte estendido com 0 localizado no endereço de memória contido em `rs1` deslocado por um imediato de 12 bits com sinal.

lui

- **Tipo:** U
- **Assembly:** `lui rd, Imm20`
- **Descrição:** Atribui a `rd` o valor do imediato de 20 bits deslocado 12 bits para a direita.

lw

- **Tipo:** I
- **Assembly:** `lw rd, Imm12(rs1)`
- **Descrição:** Atribui a `rd` o valor de 32 bits localizado no endereço de memória contido em `rs1` deslocado por um imediato de 12 bits com sinal.

nop

- **Tipo:** I
- **Assembly:** `nop`
- **Descrição:** não muda nenhum estado visível pelo usuário, exceto por avançar o `pc`. Esta instrução é codificada como a instrução `addi x0, x0, 0`.

or

- **Tipo:** R
- **Assembly:** `or rd, rs1, rs2`
- **Descrição:** Realiza a operação OR bit a bit entre os valores de `rs1` e `rs2` e armazena o resultado em `rd`.

ori

- **Tipo:** I
- **Assembly:** `ori rd, rs1, Imm12`
- **Descrição:** Realiza a operação OR bit a bit entre os valor de `rs1` e o imediato de 12 bits com sinal estendido. Armazena o resultado em `rd`.

sb

- **Tipo:** S
- **Assembly:** `sb rs2, Imm12(rs1)`
- **Descrição:** Armazena os 8 bits menos significativos de `rs2` no endereço de memória representado por `rs1` deslocado por um imediato de 12 bits.

slli

- **Tipo:** I
- **Assembly:** `slli rd, rs1, Imm12`
- **Descrição:** Desloca o valor de `rs1` para a esquerda pelo valor de bits especificado pelo imediato de 12 bits sem sinal. Armazena o resultado em `rd`.

slt

- **Tipo:** R
- **Assembly:** `slt rd, rs1, rs2`
- **Descrição:** Compara os valores de `rs1` e `rs2` levando em consideração o sinal. Caso `rs1` seja menor que `rs2`, então será atribuído o valor 1 para `rd`. Caso contrário, será atribuído o valor 0 para `rd`.

sltu

- **Tipo:** R
- **Assembly:** `sltu rd, rs1, rs2`
- **Descrição:** Compara os valores de `rs1` e `rs2` sem levar em consideração o sinal. Caso `rs1` seja menor que `rs2`, então será atribuído o valor 1 para `rd`. Caso contrário, será atribuído o valor 0 para `rd`.

srai

- **Tipo:** I
- **Assembly:** `srai rd, rs1, Imm12`
- **Descrição:** Desloca o valor de `rs1` para a direita pelo valor de bits especificado pelo imediato de 12 bits sem sinal realizando a extensão do sinal. Armazena o resultado em `rd`.

srli

- **Tipo:** I
- **Assembly:** `srli rd, rs1, Imm12`
- **Descrição:** Desloca o valor de `rs1` para a direita pelo valor de bits especificado pelo imediato de 12 bits sem sinal adicionando bits 0 à esquerda. Armazena o resultado em `rd`.

sub

- **Tipo:** R
- **Assembly:** `sub rd, rs1,rs2`
- **Descrição:** subtrai `rs2` de `rs1`. Armazena o resultado em `rd`.

sw

- **Tipo:** S
- **Assembly:** `sw rs2, Imm12(rs1)`
- **Descrição:** Armazena os bits de `rs2` no endereço de memória representado por `rs1` deslocado por um imediato de 12 bits.

xor

- **Tipo:** R
- **Assembly:** `xor rd, rs1, rs2`
- **Descrição:** Realiza a operação lógica XOR bit a bit entre os valores de `rs1` e `rs2`. Armazena o resultado em `rd`.

ecall

- **Tipo:** I
- **Assembly:** `ecall`
- **Descrição:** Executa uma chamada do sistema especificada pelo valor no registrador `a7`. No caso do simulador em questão foram implementadas as seguintes chamadas do sistema:

Nome	a7	Descrição
PrintInt	1	Imprime o inteiro armazenado em `a0`
PrintString	4	Imprime string terminada em `\0` cujo endereço do primeiro caractere está armazenado em `a0`
Exit	10	Termina a execução do programa com código de saída 0.

Tabela 1. Chamadas do sistema suportadas pela instrução ecall.

3. Testes e Resultados

Finalmente, para testar o simulador foram utilizados dois programas. O primeiro deles é apresentado a seguir:

```
.data
primos: .word 1, 3, 5, 7, 11, 13, 17, 19
size: .word 8
msg: .asciz "Os oito primeiros numeros primos sao : "
space: .ascii " "
.text
 la t0, primos # carrega endereço inicial do array
la t1, size # carrega endereço de size
lw t1, 0(t1) # carrega size em t1
li a7, 4 # imprime mensagem inicial
la a0, msg
ecall
loop: beq t1, zero, exit # se processou todo o array, encerra
li a7, 1 # serviço de impressão de inteiros
lw a0, 0(t0) # inteiro a ser exibido
ecall
li a7, 4 # imprime separador
la a0, space
ecall
addi t0, t0, 4 # incrementa indice array
addi t1, t1, -1 # decrementa contador
j loop # novo loop
exit: li a7, 10
ecall
```

O programa acima tem como objetivo mostrar 8 números armazenados em um vetor na memória. Para isso, o programa faz uso de algumas das instruções desenvolvidas para o trabalho. Pode-se perceber a presença de pseudo-instruções no programa. Nesse caso, tais instruções são transformadas em instruções simples pelo montador RARS.

Sendo assim, o programa e os dados armazenados foram exportados em forma de arquivos binários para o serem utilizados pelo simulador. O resultado obtido foi o seguinte:

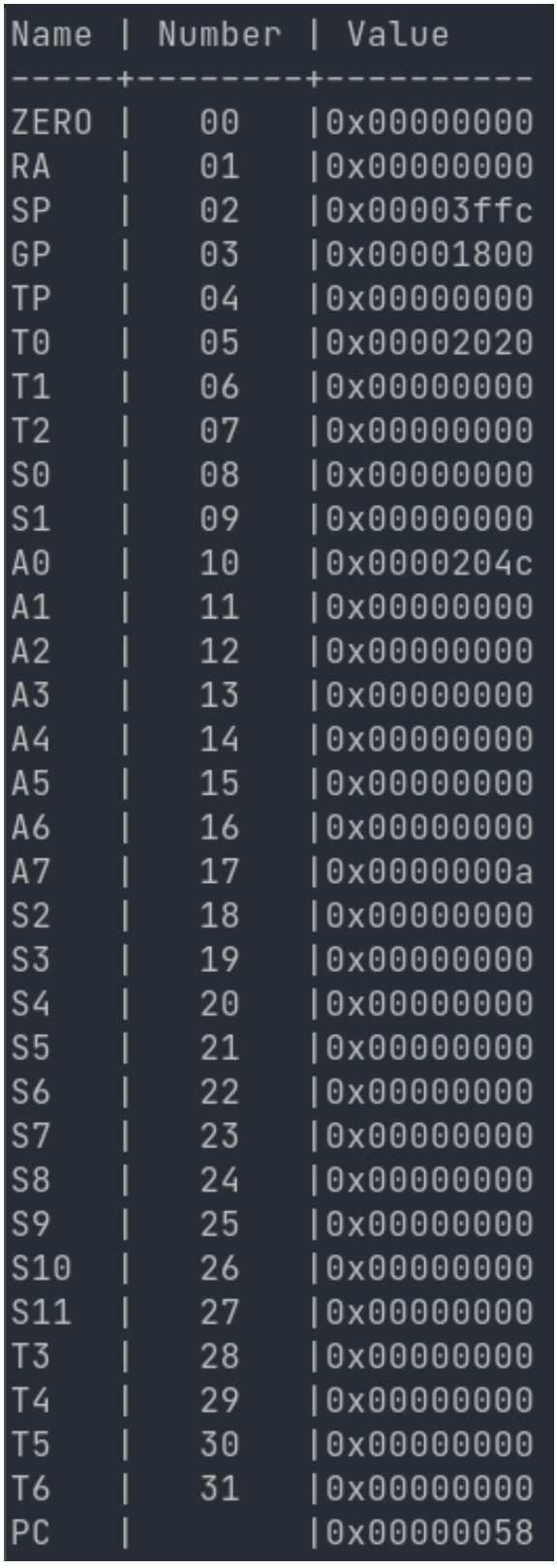
```
OAC/Trabalhos/Trabalho-2 on  main [!?] ❯ ./simulador
Os oito primeiros numeros primos sao : 1 3 5 7 11 13 17 19 %
OAC/Trabalhos/Trabalho-2 on  main [!?] ❯
```

Figura 3. Terminal apos a execução do simulador para o primeiro programa de teste.

Apesar do resultado ser o esperado, foi feita uma verificação do banco de registradores do simulador após a execução do programa para confirmar se ele apresentava os valores contidos no banco de registradores do RARS.

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x00003ffc
gp	3	0x00001800
tp	4	0x00000000
t0	5	0x00002020
t1	6	0x00000000
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x0000204c
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x0000000a
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00000058

Figura 4. Banco de registradores do RARS após a execução do primeiro programa de teste.



Name	Number	Value
ZERO	00	0x00000000
RA	01	0x00000000
SP	02	0x00003ffc
GP	03	0x00001800
TP	04	0x00000000
T0	05	0x00002020
T1	06	0x00000000
T2	07	0x00000000
S0	08	0x00000000
S1	09	0x00000000
A0	10	0x0000204c
A1	11	0x00000000
A2	12	0x00000000
A3	13	0x00000000
A4	14	0x00000000
A5	15	0x00000000
A6	16	0x00000000
A7	17	0x0000000a
S2	18	0x00000000
S3	19	0x00000000
S4	20	0x00000000
S5	21	0x00000000
S6	22	0x00000000
S7	23	0x00000000
S8	24	0x00000000
S9	25	0x00000000
S10	26	0x00000000
S11	27	0x00000000
T3	28	0x00000000
T4	29	0x00000000
T5	30	0x00000000
T6	31	0x00000000
PC		0x00000058

Figura 5. Banco de registradores do simulador após a execução do primeiro programa de teste.

Como pode ser visto nas Figuras 4 e 5, os valores apresentados nos registradores são os mesmos, o que confirma a execução corretado do programa.

Em sequência, foi utilizado o programa testador fornecido juntamente com a descrição do trabalho. O programa tem a função de testar todas as funções implementadas e informar se as mesmas estão apresentando um comportamento válido.

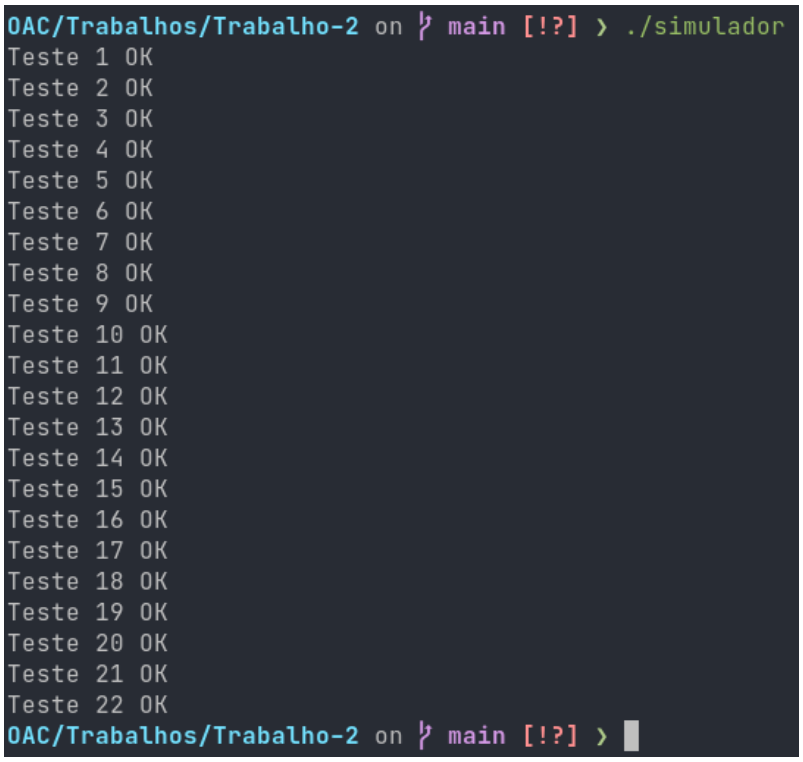


Figura 6. Banco de registradores do simulador após a execução do primeiro programa de teste.

Finalmente, por meio da Figura 6, é possível concluir que o simulador está executando todas as instruções de forma correta.

Referências

[ARM 2022] ARM (2022). Instruction set architecture (isa). `https://www.arm.com/glossary/isa#:~:text=An%20Instruction%20Set%20Architecture%20(ISA,as%20how%20it%20gets%20done.` [Online; accessed 27-Feburary-2022].

[Patterson and Waterman 2019] Patterson, D. and Waterman, A. (2019). *Guia prático RISC-V: Atlas de uma Arquitetura Aberta*. Strawberry Canyon LLC, 1th edition.