

PROJETO RISC-V

INFRAESTRUTURA DE HARDWARE

- 2 Módulos do projeto
- 14 Por onde começar?
- 15 Entendendo a instrução (.mif)
- 18 Tipos de instrução (R, I, S, SB, U, UJ)
- 20 Instruções a serem implementadas

Links úteis:

repositório ↗

https://github.com/nathaliafab/Projeto_IH_RISC-V

<https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf> documentação oficial ↗

<https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html#> ↗ guia rápido das instruções ↗

<https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8>

<https://www.intel.com/content/www/us/en/software-kit/750666/modelsim-intel-fpgas-standard-edition-software-version-20-1-1.html> USEM ESSA VERSÃO!!!
(e evitem dor de cabeça)

Módulos do projeto

① Testbench (tb-top)

↳ gerador de estímulos

↳ instancia o módulo top level

② Top Level (riscv)

↳ junta todos os módulos do projeto

↳ instancia controladores e datapath

Controller

→ output:

Sinais que indicam ações

a serem tomadas.

• ALUSrc:

0 - Operando da ALU vem do rs2

1 - Operando da ALU vem dos 16 bits da parte baixa da instrução (em resumo, o segundo operando será imediato/offset)

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

- MemtoReg:

0 - O valor a ser escrito no registrador vem da ALU

1 - O valor a ser escrito no registrador vem da memória

- RegWrite: indica se deve haver escrita do dado no registrador

- MemRead: indica se deve haver leitura da memória

- MemWrite: indica se deve haver escrita na memória

- ALUOp: especifica qual tipo de instrução vamos lidar na ALU (essa saída será entrada no controlador da ALU)

- Branch: indica se o desvio ocorreu ou não (essa saída será entrada do Branch Unit)

ALU Controller

Input: ALUOp,
funct7,
funct3

Output: Operation

Aqui, o controlador da ALU vai receber os dados da instrução (natureza e funcionalidades) e vai "descobrir" qual instrução é para mandar para a ALU.

Exemplo:

↳ ALUOp = 10

↳ Funct7 = 0000000

↳ Funct3 = 111

} AND

Operation = 0000

a ALU vai receber
ISSO e entender
que é a instrução
AND.

para a instrução de AND,
foi atribuído, arbitrariamente,
o código "0000". Cada instrução
vai ter um código escolhido por
vcs.

Datapath

→ Input: Sinais de controle, clock, reset

→ Output: opcode, funct7, funct3, WB-Data,] } dados que será escrito
Sinais para o testbench (depuração)

No datapath está toda a lógica do pipeline, ou seja, os diferentes estágios e a passagem de sinais entre eles por meio de registradores especiais do pipeline (A, B, C e D).
(buffers)

Também é aqui onde a maioria dos módulos está instanciada:

- instruction memory: recebe o PC e retorna a instrução lida no instruction.mif

- Hazard Detection: unidade de detecção de conflitos
- RegFile: banco de registradores
- imm-gen: devolve o imediato (sign extended) a partir da instrução
- Forwarding Unit: unidade de adionamento
- alu: onde são resolvidas instruções lógico-aritméticas
- Branch Unit: onde são resolvidos os desvios
- data memory: onde são feitas as leituras e escritas na memória de dados

Input: instrução

imm-Gen

Output: imediato com sinal extendido

- Onde a saída é usada:

- ① mux para decidir entrada da ALU
- ② unidade de desvio (Branch Unit)

Nesse módulo, são retirados os primeiros 7 bits da instrução (opcode) para saber qual o tipo da instrução. Dependendo do tipo, pode ou não haver imediato, e esse imediato pode estar distribuído de diferentes formas:

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB

O objetivo é "reconstruir" o imediato e transformar em um sinal de 32 bits com extensão do sinal.

No caso do SB, a saída seria:

31 12 11 5 4 0

[000...00] [31:25] [11:7]

ou

[111...11]

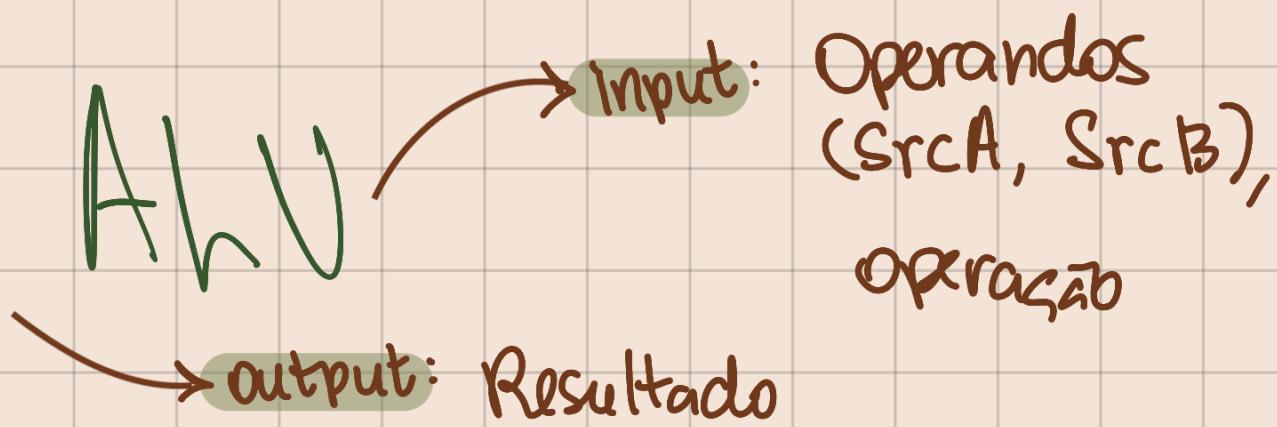


0: se for positivo
1: se for negativo

↓
os bits 0 a 4
do imediato são os
bits 7 a 11 da instru-
ção SB.

Se não for identificada a instrução, a saída

0.



Ambos os operandos são decididos por muxes no datapath.

SrcA: ou vem do rs1 ou do valor que
será escrito no registrador.
(hazard)

SrcB: ou vem do rs2 ou do valor que
será escrito no registrador ou é um
imediato.
(hazard)

A operação vem do controlador da ALU,
que já fez a identificação da instrução.

Branch Unit

Input:

PC atual,
imediato,
sinal de branch,
resultado da ALU

Output:

Novos valores
de PC,
PCSel

A unidade de branch tem como função mudar o rumo do código. Aqui, vários possíveis novos PC são enviados na saída junto de um seletor, que vai selecionar qual desses valores será usado.

- PC-imm: PC atual + imediato ou seja, houve desvio e o código vai avançar ou retroceder de acordo com o imediato.
- PC-Four: PC atual + 4 ou seja, não houve desvio e o código vai avançar normalmente para próxima instrução
- PCSel: 1 se tiver branch, 0 se não

datamemory

→ **input:**

dado (rd)

→ **Input:**

Sinais de controle,
endereço (a),
dado (wd),
funct 3

Aqui é instanciado a Memória 32 Data, que faz a leitura e escrita no módulo de memória da Altera (romOnChipData). Esse módulo tem como entradas:

- **raddress** (endereço de leitura)
- **Waddress** (endereço de escrita)
- **datain** (dado a ser escrito)
- **Wr** (write enable — identifica se é escrita ou leitura)

Assim como a Memória de instruções, a memória de dados é inicializada através de um arquivo .mif

* O .mif não guarda o estado atual da memória, apenas a inicializa.

No data memory, vamos adaptar as entradas para enviar ao memória 32Data. Dependendo do funct3, o endereço da leitura/escrita pode ser diferente, assim como o Wr e o datain.

- LB: Lê apenas um byte e "extende" o sinal
- LBV: Lê apenas um byte e completa com 0's
- LH: Lê apenas meia palavra (2 bytes) e "extende" o sinal

Como não há escrita, o Wr = 0000

- SW: escreve na palavra (4 bytes)
 - ↪ Wr = 1111

A memória é endereçada por byte e alinhada. Sendo assim, numa instrução LW ou SW, endereços "quebrados" são interpretados como pertencentes a mesma palavra:

Palavra 0

$lw \times b, 0(x0)$

Palavra 0

$lw \times b, z(x0)$

Palavra 1

$\neq lw \times b, 4(x0)$

Da mesma forma, LH e SH:

half 0

$lh \times b, 0(x0) \equiv lh \times b, 1(x0)$

half 0

Palavra 0

half 1

$lh \times b, 2(x0) \equiv lh \times b, 3(x0)$

half 1

E o LB e SB:

byte 0

$lb \times b, 0(x0) \neq lb \times b, 1(x0)$

byte 1

Palavra 0

byte 2

$lb \times b, 2(x0) \neq lb \times b, 3(x0)$

byte 3

\neq resultados diferentes

\equiv resultados iguais

"Por onde começar?"

- ① Rode o projeto como está no ModelSim.
- ② Observe as instruções já implementadas.

BEQ

LW

SW

ADD

AND

} "quais eu preciso implementar que são parecidas com as que eu já tenho?"

- ③ Observe o código e o diagrama e tente imaginar onde cada instrução deve ir.

SUB, OR, XOR, outras!

★ ALU

★ ALU Controller

(etc)

LB, LH, SB, outras!

★ data memory

★ Memoria 32 Data

(etc)

JAL, JALR, BNE, outras!

* Branch Unit (etc)

* Imm Gen

Se necessário, mexte e adapte outros módulos.
Também é liberado criar módulos novos p/
auxiliar o processo :)

Entendendo a instrução

O RISC-V é little-endian.

Little-endian significa **armazenar bytes na ordem do menor para o mais significativo** (onde o byte menos significativo ocupa o primeiro, ou menor, endereço), comparável a maneira comum de escrever datas na Europa (por exemplo, 31 Dezembro de 2050).

Ou seja, os bits mais significativos estão em endereços mais altos. Vamos ver como uma instrução de ADD fica na memória:

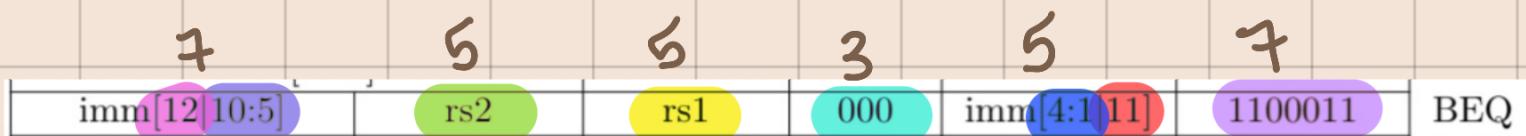


→ Começa aqui (bit 0)

```
020: 00110011;          -- add x6, x4, x2  
021: 00000011, → bit 8  
022: 00100010; → bit 16  
023: 00000000; → bit 24
```

→ Termina aqui (bit 31)

Agora uma mais complexa, mas a lógica é a mesma:



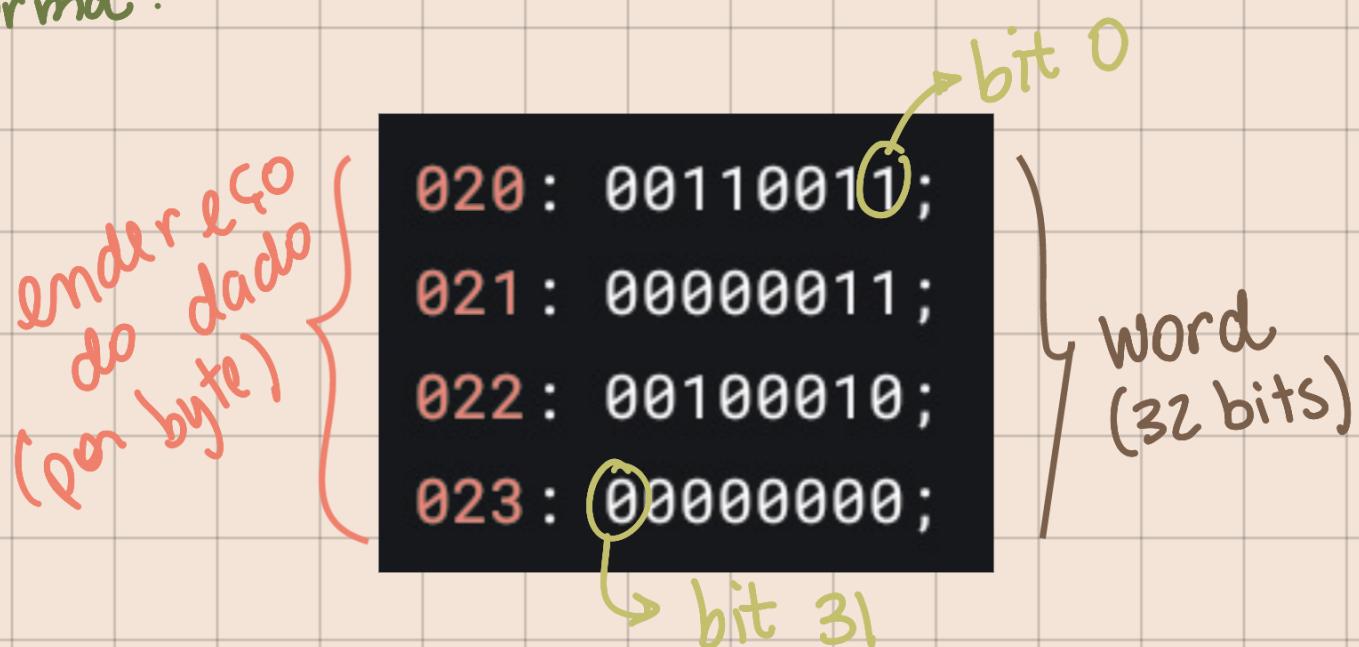
Lembrando: $-8 = 1111\dots11000$

```
028: 11100011;          -- beq x8, x7, -8  
029: 00001100;  
030: 01110100;  
031: 11111110;
```

Perceba que o imediato é completamente fragmentado na instrução (por isso o imm-gen é importante, ele quem arruma :))

Na prática, o script em python faz essa "tradução" do assembly para o formato do .mif (binário), mas é bom entender onde a instrução começa, onde termina e o que cada bit significa. Isso é feito observando a documentação oficial do RV32I. (ver pág 116 do PDF)

A memória de dados é lida da mesma forma:



020: 00110011;

byte
(8 bits)

020: 00110011;

021: 00000011;

half-word
(16 bits)

Sendendo assim, uma instrução de SH (store half) deve escrever **apenas** 16 bits deixando os outros 16 intactos; o mesmo vale para o SB (store byte), que deve sobrescrever 8.

Tipos de instrução (pág 24 do PDF)

31	30	25 24	21	20	19	15 14	12 11	8	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]	imm[10:5]		rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
		imm[31:12]						rd		opcode		U-type
		imm[20]	imm[10:1]	imm[11]	imm[19:12]			rd		opcode		J-type

R: registrador-registrador

↳ add, sub, or...

I: loads e short immediates (12 bits)

↳ addi, lw, srai...

S: stores

↳ sw, sh, sb

(ou SB)

B: branch condicional

↳ beq, bne, bge ...

V: long immediates (20 bits)

↳ lui, auipc

(ou VJ)

J: jump

↳ jal

Como está o imediato em cada tipo:

31	30	20 19	12	11	10	5	4	1	0	
— inst[31] —		inst[30:25] inst[24:21] inst[20]						I-immediate		
— inst[31] —		inst[30:25] inst[11:8] inst[7]						S-immediate		
— inst[31] —		inst[7] inst[30:25] inst[11:8] 0						B-immediate		
inst[31]	inst[30:20]	inst[19:12]	— 0 —						U-immediate	
— inst[31] —	inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0				J-immediate	

(viaja o Imm-Gen)

Instruções a serem implementadas (página no PDF entre parênteses)

#	Instrução	Implementada	Testada	Funcionando
1	BEQ	✓	✓	✓
2	LW	✓	✓	✓
3	SW	✓	✓	✓
4	ADD	✓	✓	✓
5	AND	✓	✓	✓

já no projeto

Jumps | Branches

#	Instrução	Implementada	Testada	Funcionando
1	JAL	✗	✗	✗
2	JALR	✗	✗	✗
3	BNE	✗	✗	✗
4	BLT	✗	✗	✗
5	BGE	✗	✗	✗

(27-28)

(29-30)

- JAL: pula pra PC + offset e guarda o PC + 4 em rd
- JALR: pula pra PC + rs1 + offset e guarda o PC + 4 em rd
- Branches condicionais
 - BNE ≠
 - BLT <
 - BGE ≥

• Loads e Stores

6	LB	X	X	X
7	LH	X	X	X
8	LBU	X	X	X
9	SB	X	X	X
10	SH	X	X	X

(30-31)

• Lógico-aritméticas

11	SLTI	X	X	X
12	ADDI	X	X	X
13	SLLI	X	X	X
14	SRLI	X	X	X
15	SRAI	X	X	X
16	SUB	X	X	X
17	SLT	X	X	X
18	XOR	X	X	X
19	OR	X	X	X
20	LUI	X	X	X

I-TYPE (25-26)

R-TYPE (27)

J-TYPE (26)

OBS: os shifts não devem funcionar para valores de imediato negativos.

• Halt

21	HALT	X	X	X
----	------	---	---	---

halt é uma instrução que não existe no RV32I. Seu objetivo é indicar o término do programa de forma explícita.

- ↳ ① Trava o PC
- ↳ ② Insere zeros (ou outros valor) no pipeline



"Em que partes eu devo ou não mexer?"

Não se preocupe com:

- | | |
|----------------------|--------------------------------------|
| ↳ instruction memory | ↳ módulos da Altera
(cramOn Chip) |
| ↳ Hazard Detection | |
| ↳ Forwarding Unit | ↳ Memória 32 |
| ↳ RegFile | |

O resto precisará de adaptações.