

Departamento de Engenharia Elétrica e de Computação

SEL5752 - Dispositivos Reconfiguráveis e Linguagem de Descrição de Hardware

Prof. Dr. Maximilian Luppe

PROJETO FINAL

Processador RISC-V

Projeto

Estudar, modificar e testar uma implementação em VHDL da arquitetura RISC-V, versão RV32I, de ciclo único, baseado na implementação descrita por Harris e Harris [1]

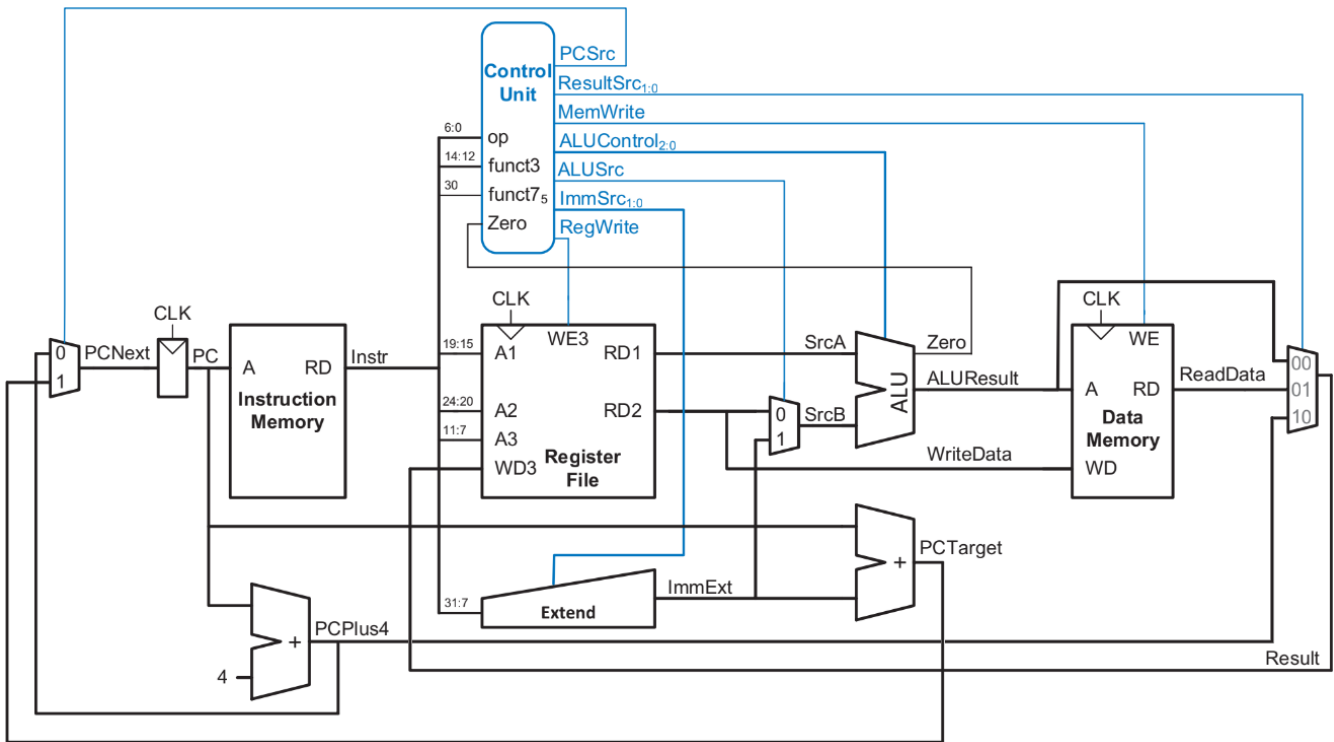
Introdução

Valendo-se da natureza prática da disciplina SEL5752 e visando proporcionar aos alunos uma experiência mais ativa e dinâmica nas atividades de projeto de sistemas digitais baseados em linguagens de descrição de hardware, será adotada a metodologia de aprendizado ativo baseado em implementação de projetos (PBL - *Problem Based Learning*) na atividade final da disciplina.

O PBL consiste em apresentar ao aluno um projeto a ser desenvolvido, de modo que este busque as informações necessárias para a sua implementação. O aluno poderá utilizar todo o conhecimento adquirido ao longo da disciplina, colocando-os em prática. O problema central será a construção de um processador baseado na implementação do RISC-V, na versão de ciclo único.

A arquitetura RISC-V é baseada em princípios RISC (*Reduced Instruction Set Computer*), desenvolvido desde 2010, na Universidade da Califórnia, em Berkeley. Um esquemático simples do caminho de dados para a versão de ciclo único pode ser visto na figura 1.

Figura 1 - Processador RISC-V ciclo único



Fonte: Digital Design and Computer Architecture - RISC-V Edition <https://doi.org/10.1016/C2019-0-00213-0>

A arquitetura RISC-V define seis formatos de instrução de base (tabela 1): Tipo-R para operações de registradores; Tipo-I para valores imediatos *short* e *loads*; Tipo-S para *stores*; Tipo-B para desvios condicionais; Tipo-U para valores imediatos longos; e tipo-J para saltos incondicionais. As instruções do Tipo-R operam sobre três registradores, como `add rd, rs1, rs2`, que realiza a operação $[rd] = [rs1] + [rs2]$, sendo `rd`, `rs1` e `rs2` os registradores do banco de registradores. Instruções do Tipo-I realizam operações que envolvem o uso de valores imediatos (incluídos nas instruções), como `addi rd, rs1, 42`, que realiza a operação $[rd] = [rs1] + 42$. Instruções do Tipo-S e do Tipo-B, por sua similaridade no formato (operam sobre dois registradores e um valor imediato de 12 ou 13 bits), podem ser consideradas só um grupo, e realizam operações de armazenamento (Tipo-S) e de desvio de fluxo (Tipo-B), como `sw a0, 4(sp)`, que armazena em $M([sp] + 4)$ o valor de `a0`, ou `beq a0, a1, L1`, que desvia o fluxo para o endereço `L1` se $[a0] = [a1]$. Da mesma forma, as instruções do Tipo-U e do Tipo-J também podem ser agrupadas num só grupo (operam sobre um registrador e um valor imediato de 20 ou 21 bits), como `jal ra, factorial`, que desvia o fluxo para o endereço `factorial` e armazena o endereço de retorno em $[ra]$.

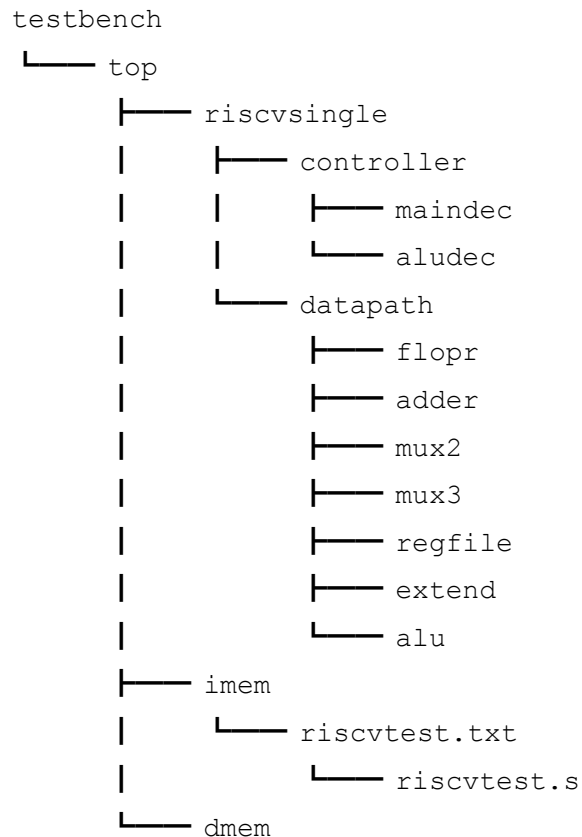
Tabela 1 - Formatos de Instruções RV32I

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
funct7				rs2		rs1	funct3		rd			opcode			Tipo R
imm[11:0]						rs1	funct3		rd			opcode			Tipo I
imm[11:5]				rs2		rs1	funct3		imm[4:0]			opcode			Tipo S
imm[12]	imm[10:5]			rs2		rs1	funct3		imm[4:1]	imm[11]		opcode			Tipo B
imm[31:12]									rd			opcode			Tipo U
imm[20]	imm[10:1]			imm[11]		imm[19:12]			rd			opcode			Tipo J

Fonte: Digital Design and Computer Architecture - RISC-V Edition <https://doi.org/10.1016/C2019-0-00213-0>

O projeto em VHDL apresentado por Harris e Harris [1], para a implementação da arquitetura RISC-V em ciclo único, tem a seguinte estrutura hierárquica, apresentada na listagem 1.

Listagem 1 - Estrutura hierárquica da implementação da arquitetura RISC-V de ciclo único [1]

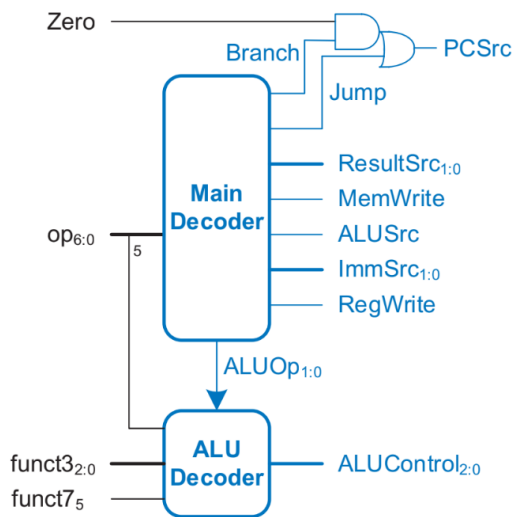


A entidade `testbench` é utilizada para testar a implementação da arquitetura RISC-V, e instancia as entidades `top`, que por sua vez instancia as entidades `riscvsingle` (a implementação da arquitetura RISC-V, propriamente dita), a memória de instruções `imem` (que contém o código em linguagem de máquina em formato texto `riscvtest.txt`, obtido a partir do código em assembly `riscvtest.s`) e a memória de dados `dmem`.

A entidade `riscvsingle`, por sua vez, instancia as entidades `controller` e `datapath`. A entidade `controller` é responsável por decodificar as instruções e gerar os sinais de controle para o `datapath` (`maindec`) e para a Unidade Lógico-Aritmética (ULA) (`aludec`). A entidade `datapath` contém todas as entidades necessárias para a execução das instruções, como `flopr` (para implementar o contador de programas PC), `adder` (para calcular o endereço da nova instrução), `mux2` e `mux3` (para controlar o fluxo de dados), `regfile` (para armazenar os registradores), `extend` (para gerar os valores imediatos para a instruções), e `alu` (para realizar as operações lógicas e aritméticas).

A entidade `controller` é formada pelas entidades `maindec` (Main Decoder) e `aludec` (ALU Decoder), conforme apresentado na figura 2.

Figura 2 - Estrutura interna da entidade `controller`



Fonte: Digital Design and Computer Architecture - RISC-V Edition <https://doi.org/10.1016/C2019-0-00213-0>

A entidade `maindec` é responsável por decodificar as instruções, representada pela entrada `op6:0`, e gerar os sinais que controlarão as entidades que compõem a entidade `datapath`. A entidade `aludec` é responsável por gerar os sinais de controle para a ULA, em função dos campos `op5`, `funct32:0` e `funct75`.

das instruções, e do sinal $ALUOp_{1:0}$, gerado pela entidade *maindec*. Na tabela 2 são apresentados os sinais de controle gerados pela *maindec*, de acordo com o tipo de instrução decodificada.

Tabela 2 - Conjunto de sinais de controle gerados pela entidade *maindec*

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
lw	0000011	1	00	1	0	01	0	00	0
sw	0100011	0	01	1	1	xx	0	00	0
R-type	0110011	1	xx	0	0	00	0	10	0
beq	1100011	0	10	0	0	xx	1	01	0
I-type ALU	0010011	1	00	1	0	00	0	10	0
jal	1101111	1	11	x	0	10	0	xx	1

Fonte: Digital Design and Computer Architecture - RISC-V Edition <https://doi.org/10.1016/C2019-0-00213-0>

A entidade *datapath* é responsável pela execução das instruções, de acordo com os sinais de controle gerados pela entidade *controller*. Ela é composta de diversas entidades, como a *alu*, que implementa a ULA. A ULA é um circuito que realiza operações aritméticas (como Soma e Subtração) e operações lógicas (como AND e OR) entre dois valores de N bits, resultando em outro valor também de N bits, além de alguns sinais adicionais, como *flags* de Zero, de Negativo, de *Carry*, de *Overflow* etc. A largura do barramento de dados da ULA é, em algumas classificações, utilizado para classificar o tipo de arquitetura: 8 bits, 16 bits, 32 bits, etc. Na tabela 3 temos as operações realizadas pela ULA da arquitetura RISC-V implementada, de acordo com o valor do sinal de controle $ALUControl_{2:0}$.

Tabela 3 - Conjunto de operações da ULA

$ALUControl_{2:0}$	Function
000	Add
001	Subtract
010	AND
011	OR
101	SLT

Fonte: Digital Design and Computer Architecture - RISC-V Edition <https://doi.org/10.1016/C2019-0-00213-0>

A operação SLT significa *Set if Less Than*, ou seja, verifica se A é menor do que B, de acordo com a Tabela 4, sendo A e B dois valores de N bits que alimentam a ULA.

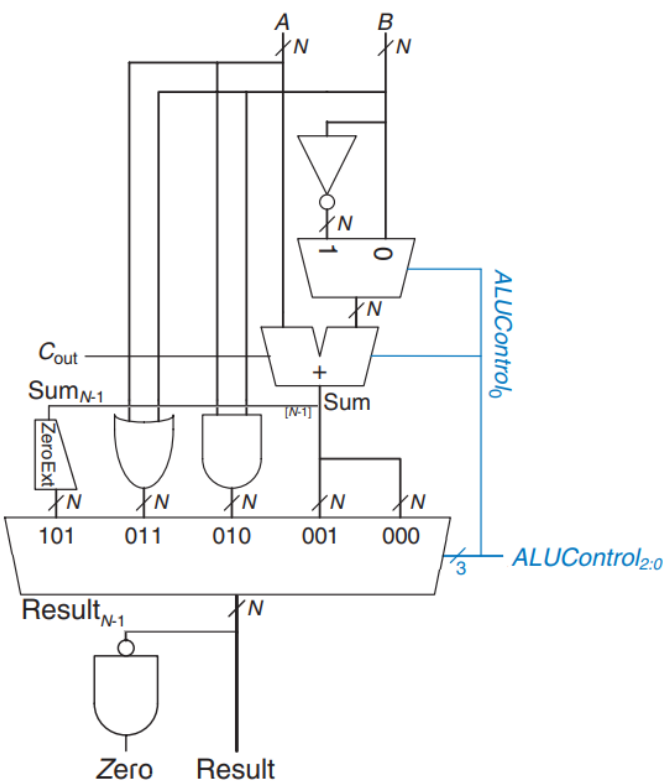
Tabela 4 - Resultado da operação SLT para N=8

SLT	
A >= B	A < B
00000000	00000001

Fonte: o autor

Na figura 3 é apresentado o esquemático em alto nível de uma possível implementação de uma ULA de N bits. Além do resultado descrito na Tabela 3, a ULA deverá ter como saída uma *flag* de Zero.

Figura 3 - ULA de N bits com suporte para SLT e *flag* de Zero



Fonte: Digital Design and Computer Architecture - RISC-V Edition <https://doi.org/10.1016/C2019-0-00213-0>

Na tabela 5 é apresentado o sinal de controle $ALUControl_{2:0}$ gerado pela entidade `aludec`, de acordo com o tipo de instrução decodificada.

Tabela 5 - Sinal de controle $ALUControl_{2:0}$ gerado pela entidade `aludec`

ALUOp	funct3	{op ₅ , funct ₇ ₅ }	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt
	110	x	011 (or)	or
	111	x	010 (and)	and

Fonte: Digital Design and Computer Architecture - RISC-V Edition <https://doi.org/10.1016/C2019-0-00213-0>

A entidade `extend` é responsável por gerar o valor imediato (ImmExt) a partir de campos específicos das instruções da arquitetura, conforme descrito na tabela 1. O valor de ImmSrc, gerado pela entidade `maindec`, é utilizado pela entidade `extend` para compor corretamente o valor imediato da instrução, agrupando os campos específicos, de acordo com a tabela 6.

Tabela 6 - Composição dos valores imediatos das instruções, de acordo com ImmSrc

ImmSrc	ImmExt	Type	Description
00	{{20{Instr[31]}}, Instr[31:20]}	I	12-bit signed immediate
01	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	S	12-bit signed immediate
10	{{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}	B	13-bit signed immediate
11	{{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0}	J	21-bit signed immediate

Fonte: Digital Design and Computer Architecture - RISC-V Edition <https://doi.org/10.1016/C2019-0-00213-0>

A entidade `floppc` tem como função implementar o contador de programas PC, armazenando o sinal PCNext, que é o endereço da próxima instrução a ser decodificada. A entidade `adder` fica responsável por calcular o endereço da nova instrução, tanto gerado pelo incremento automático do PC (PCPlus4), acrescentando 4 ao valor de PC, como gerado pelas instruções Tipo-J e Tipo-B (PCTarget),

acrescentando ao valor de PC o valor de ImmExt. A entidade `regfile` é responsável por implementar o banco de registradores, e as entidades `mux2` e `mux3` são responsáveis por controlar o fluxo de dados para o PC (`mux2`) e para banco de registradores (`mux3`).

Atividades

Para estudar, modificar e testar uma implementação em VHDL da arquitetura RISC-V, versão RV32I, de ciclo único, baseado na implementação descrita por Harris e Harris [1]. diversas atividades deverão ser realizadas. A estrutura de arquivos será baseada na listagem 2:

Listagem 2 - Estrutura de arquivos para o projeto [1]

```
SEL5752
└── Projeto2023
    ├── docs
    ├── modelsim
    ├── quartus
    └── src
        ├── testbench.vhd
        ├── top.vhd
        ├── riscvsingle.vhd
        ├── imem.vhd
        ├── dmem.vhd
        ├── controller.vhd
        ├── maindec.vhd
        ├── aludec.vhd
        ├── datapath.vhd
        ├── flopr.vhd
        ├── adder.vhd
        ├── mux2.vhd
        ├── mux3.vhd
        ├── regfile.vhd
        ├── extend.vhd
        ├── alu.vhd
        ├── riscvtest.txt
        └── riscvtest.s
```


A pasta `modelsim` será utilizada como base para a simulação da arquitetura, por meio da ferramenta *ModelSim*. A pasta `quartus` será utilizada como base para o projeto desenvolvido por meio da ferramenta Quartus. Todo o código fonte, original, modificado ou criado, será armazenado na pasta `src`. Na pasta `docs` serão armazenados os relatórios de cada atividade, incluindo este arquivo. Neste relatório deverá constar uma capa, com o nome do aluno, a atividade a ser desenvolvida, a(s) solução(ões) implementada(s), o resumo da compilação (report) e o circuito em nível RTL (*Register Transfer Level*) da atividade.

A primeira atividade é criar um projeto denominado `riscvsingle`, na ferramenta Quartus (que servirá de base para testar todas as funcionalidades da arquitetura) e incluir o arquivo `mux2.vhd` ao projeto, defini-lo como *toplevel*, e realizar alterações na entidade `mux2`, de forma a implementar um multiplexador de 2 para 1, utilizando a construção `with-select`, e parametrizando o barramento de dados com um parâmetro genérico de nome `Width`.

A segunda atividade é incluir o arquivo `mux3.vhd` ao projeto, defini-lo como *toplevel*, e realizar alterações na entidade `mux3`, de forma a implementar um multiplexador de 3 para 1, utilizando a construção `when-else`, e parametrizá-la em função de `Width` (da mesma forma que foi feito com a entidade `mux2`).

A terceira atividade é incluir o arquivo `flop_r.vhd` ao projeto, defini-lo como *toplevel*, e realizar alterações na entidade `flop_r` de forma a parametrizá-la em função de `Width` (da mesma forma que foi feito com a entidade `mux3`). Este projeto deverá implementar um registrador de `Width-1` bits, com transição positiva de clock e reset assíncrono quando a entrada `reset` for igual a '1'.

A quarta atividade é incluir o arquivo `flop_enr.vhd` ao projeto, defini-lo como *toplevel*, e realizar alterações na entidade `flop_enr` de forma a parametrizá-la em função de `Width` (da mesma forma que foi feito com a entidade `flop_r`). Este projeto deverá implementar um registrador de `Width-1` bits, com transição positiva de clock, habilitador de clock em '1' e reset assíncrono quando a entrada `reset` for igual a '1'.

A quinta atividade é incluir o arquivo `extend.vhd` ao projeto, defini-lo como *toplevel*, e realizar alterações na entidade `extend` de forma a substituir os campos da instrução (*signal*) `instr` por *alias*, baseado na tabela 6.

A sexta atividade é criar um pacote chamado `riscv_pkg` (arquivo `riscv_pkg`), na pasta `src`, e incluir a declaração de componente para as entidades `mux2`, `mux3`, `flopr`, `flopenr` e `extend`, já considerando as implementações com genéricos. Definir no mesmo pacote uma constante global chamada `RISCV_Data_Width`, com valor igual a 32.

A sétima atividade é incluir no pacote `riscv_pkg` (arquivo `riscv_pkg`), na pasta `src`, as seguintes funções:

- i) uma função chamada `to_uinteger`, que converte um objeto do tipo `bit_vector` para `integer` sem sinal.
- ii) criar uma função sobrecarregando o operador "+", que realize a soma de dois objetos do tipo `bit_vector`.
- iii) criar uma função sobrecarregando o operador "-", que realize a subtração de dois objetos do tipo `bit_vector`, fazendo uso do operador sobrecarregado "+".

A oitava atividade é incluir o arquivo `alu.vhd`, que deverá ser definido como *toplevel*. Realize alterações na entidade `alu` de forma a:

- i) simplificar a linha 28;
- ii) parametrizar o barramento de dados com um parâmetro genérico de nome `Width`;
- iii) incluir sua declaração de componente no pacote `riscv_pkg`;
- iv) indicar se é possível propor uma melhor implementação, baseado na figura 3.

A nona atividade é incluir o arquivo `regfile.vhd` ao projeto, defini-lo como *toplevel*, e realizar alterações na entidade `regfile` de forma a parametrizá-la em função de `Width` (da mesma forma que foi feito com a entidade `flopr`), fazendo uso do pacote `riscv_pkg`, e incluir sua declaração de componente no pacote.

A décima atividade é incluir o arquivo `adder.vhd` ao projeto, utilizando a função sobrecarregada do operador “+” (fazendo uso do pacote `riscv_pkg`), defini-lo como *toplevel*, e realizar alterações na entidade `adder` de forma a parametrizá-la em função de `Width`, e incluir sua declaração de componente no pacote.

A décima-primeira atividade é incluir o arquivo `datapath.vhd` ao projeto, defini-lo como *toplevel*, e realizar alterações na entidade `datapath` de forma a:

- i) parametrizá-la em função de `Width` (da mesma forma que foi feito com a entidade `flop_r`);
- ii) incluir a solicitação de uso do pacote `riscv_pkg`, tornando visível todas as declarações;
- iii) acrescentar ao pacote `riscv_pkg` a declaração de componente para a entidade `datapath`.

A décima-segunda atividade é incluir o arquivo `maindec.vhd` ao projeto, defini-lo como *toplevel*, e acrescentar ao pacote `riscv_pkg` a declaração de componente para a entidade `maindec`.

A décima-terceira atividade é incluir o arquivo `aludec.vhd` ao projeto, defini-lo como *toplevel*, e acrescentar ao pacote `riscv_pkg` a declaração de componente para a entidade `aludec`.

A décima-quarta atividade é incluir o arquivo `controller.vhd` ao projeto, defini-lo como *toplevel*, e realizar alterações na entidade `controller` de forma a:

- i) incluir a solicitação de uso do pacote `riscv_pkg`, tornando visível todas as declarações.
- ii) acrescentar ao pacote `riscv_pkg` a declaração de componente para a entidade `controller`.

A décima-quinta atividade é incluir o arquivo `riscvsingle.vhd` ao projeto, defini-lo como *toplevel*, e realizar alterações na entidade `riscvsingle` de forma a:

- i) parametrizá-la em função de `Width` (da mesma forma que foi feito com a entidade `datapath`);
- ii) incluir a solicitação de uso do pacote `riscv_pkg`, tornando visível todas as declarações.
- iii) acrescentar ao pacote `riscv_pkg` a declaração de componente para a entidade `riscvsingle`.

A décima-sexta atividade é criar um projeto denominado `riscvsingle`, na ferramenta *ModelSim* (que servirá de base para simular todas as funcionalidades da arquitetura) e incluir todos os arquivos da pasta `src` e realize alterações nas entidades `dmem`, `imem`, `top` e `testbench`, de forma a:

- i) incluir a solicitação de uso do pacote `riscv_pkg`, tornando visível todas as declarações;
- ii) parametrizá-la em função de `RISCV_Data_Width` (da mesma forma que foi feito com a entidade `riscvsingle`);
- iii) acrescentar ao pacote `riscv_pkg` a declaração de componente para as entidades `top`, `imem` e `dmem`.
- iv) apresentar o resultado da simulação.

A décima-sétima atividade é realizar a alteração na arquitetura implementada, adicionando a instrução `MULDIV` para multiplicação de dois valores com sinal de 32 bits. A função `MUL` realiza uma multiplicação de `XLEN` bits por `XLEN` bits e coloca os `XLEN` bits menos significativos no registrador de destino. `MULH`, `MULHU` e `MULHSU` realiza a mesma multiplicação, mas retorna os `XLEN` bits mais significativos do produto completo de $2 \times \text{XLEN}$ bits, para uma multiplicação de signed x signed, unsigned x unsigned e signed x unsigned, respectivamente [2].

Na tabela 7 é apresentado o formato da instrução `MULDIV`, e na tabela 8, os respectivos opcodes.

Tabela 7 - Formato da instrução `MULDIV`,

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
MULDIV	multiplier	multiplicand	MUL/MULH[[S]U]	dest	OP	

Tabela 8 - Opcodes da instrução `MULDIV`

RV32M Standard Extension						
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Referência bibliográfica

- [1] Harris, S. L., Harris, D., “Digital Design and Computer Architecture RISC-V Edition”, Morgan Kaufmann, 2022
- [2] Waterman, A., Asanovic, K., The RISC-V Instruction Set Manual - Volume I: User-Level ISA - Document Version 2.2, <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>, acessado em 17/10/2023