



THE IMAGINATION UNIVERSITY PROGRAMME

# **Referência SweRV EH1**

## ***Hierarquia, Módulos, Sinais e Tipos***

Este documento fornece instruções extra sobre os seguintes tópicos:

- Secção 1: **Sigasi Studio**
- Secção 2: **Configuração do processador SweRV EH1**
- Secção 3: **Hierarquia do Sistema RVfpga de módulos e seus sinais mais relevantes**
- Secção 4: **Principais estruturas/tipos para agrupamento de bits de controlo**
- Secção 5: **Instruções comprimidas RISC-V**
- Secção 6: **Avaliações De Desempenho (Benchmarks) reais**

## 1. SIGASI STUDIO

Sigasi Studio melhora a produtividade dos projetistas ao ajudar a escrever, inspecionar e modificar os projetos de circuitos digitais da forma mais intuitiva. Esta ferramenta compreende o contexto do projeto. Características avançadas tais como autocompletamentos inteligentes e refatorização de código tornam o projeto VHDL, Verilog e SystemVerilog mais fácil e mais eficiente..

Sigasi O estúdio necessita de uma comissão para a obtenção de uma licença e para poder utilizá-la profissionalmente. Felizmente, existe uma licença gratuita para fins educacionais que pode facilmente obter em: <https://www.sigasi.com/try-form-edu/>. Assim que preencher os seus dados e a sua licença for aprovada, receberá um e-mail com as instruções e um endereço para descarregar (<https://www.sigasi.com/download/>, ver Figura 1), instalação e utilização do Sigasi Studio. O software está disponível para Windows, Linux e MacOS.

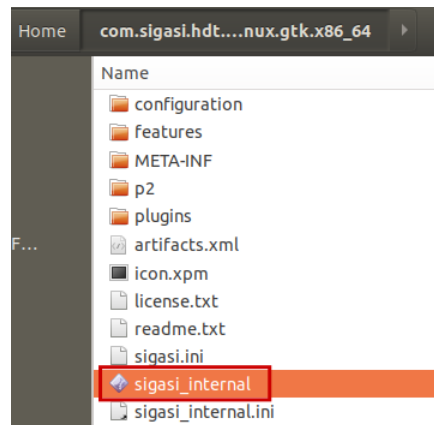


**Figura 1. Ligação para descarregar, instalar e utilizar o Sigasi Studio**

Uma vez instalado o Sigasi Studio no seu sistema, pode começar a utilizá-lo para inspeccionar o RVfpga. Na seguinte ligação, há dois anos, Hendrik Eeckhaut publicou instruções para criar e configurar um projeto para o SweRV EH1: [https://insights.sigasi.com/tech/swerv\\_riscv/](https://insights.sigasi.com/tech/swerv_riscv/). Usando essa informação como ponto de

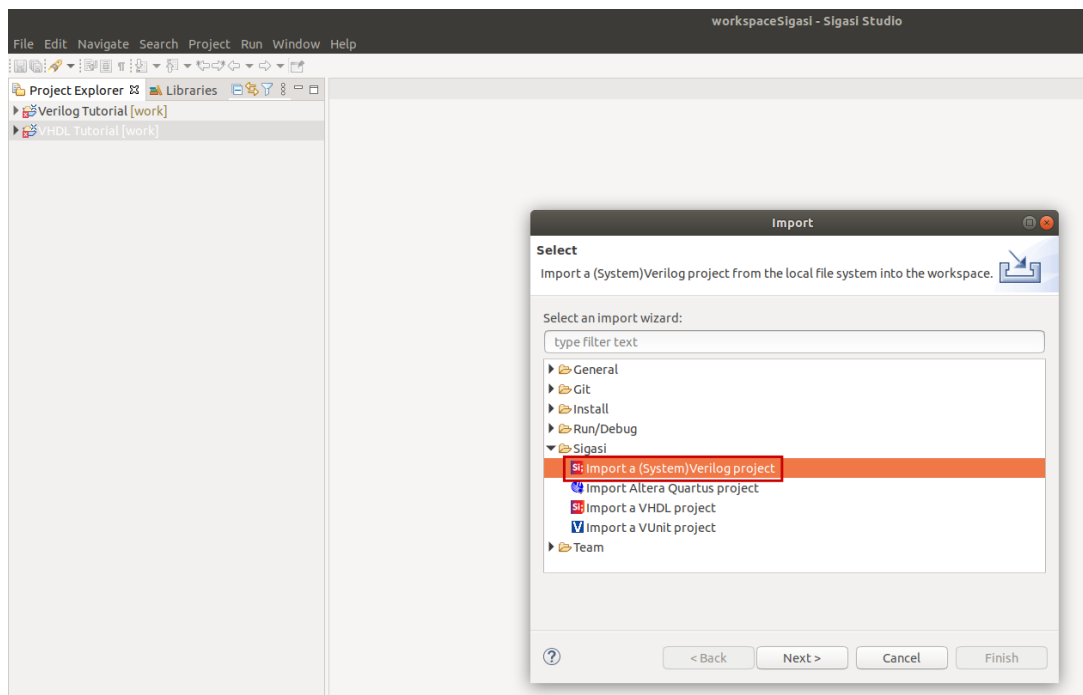
partida, fornecemos em seguida instruções completas para a criação e configuração de um projeto para a RVfpga.

1. Criar uma cópia do directório `[RVfpgaPath]/RVfpga/src` e nomei-o `[RVfpgaPath]/RVfpga/src_SigasiStudio`
2. Abra o Estúdio Sigasi entrando no diretório descarregado e clicando duas vezes no ficheiro `sigasi_internal` (ver Figura 2).



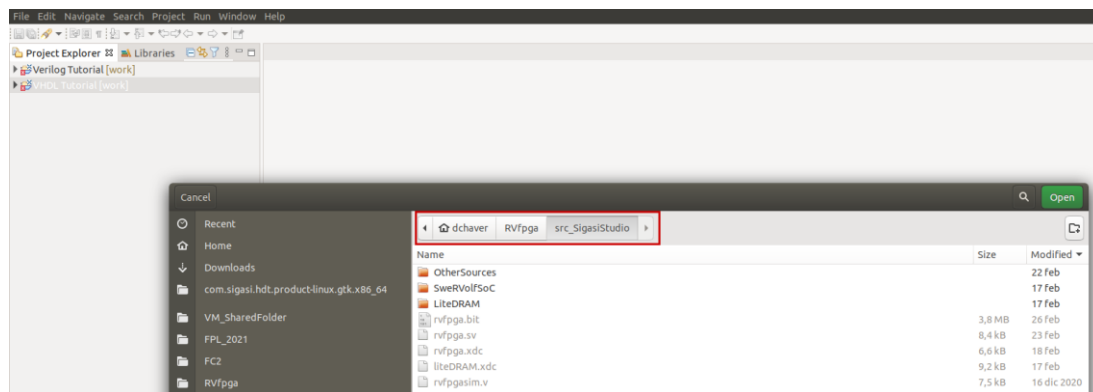
**Figura 2. Abrir o Sigasi Studio**

3. Na janela do Estúdio Sigasi clique em **File** → **Import...** Abrir-se-á uma nova janela que lhe pede para seleccionar o tipo de projeto que pretende adicionar ao seu sistema. Escolha "Import a (System) Verilog project" e clique next (ver Figura 3).



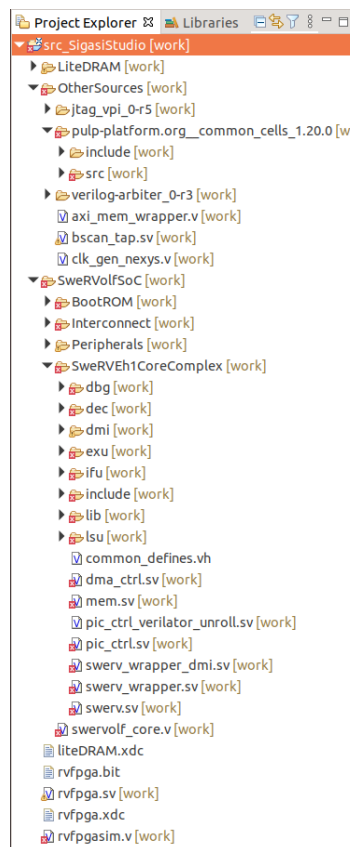
**Figura 3. Importar o projeto RVfpga**

4. Agora clique em "Browse..." e vá para o directório `src_SigasiStudio` e clique **Open** (ver Figura 4) e, em seguida, clicar em **Finish**.



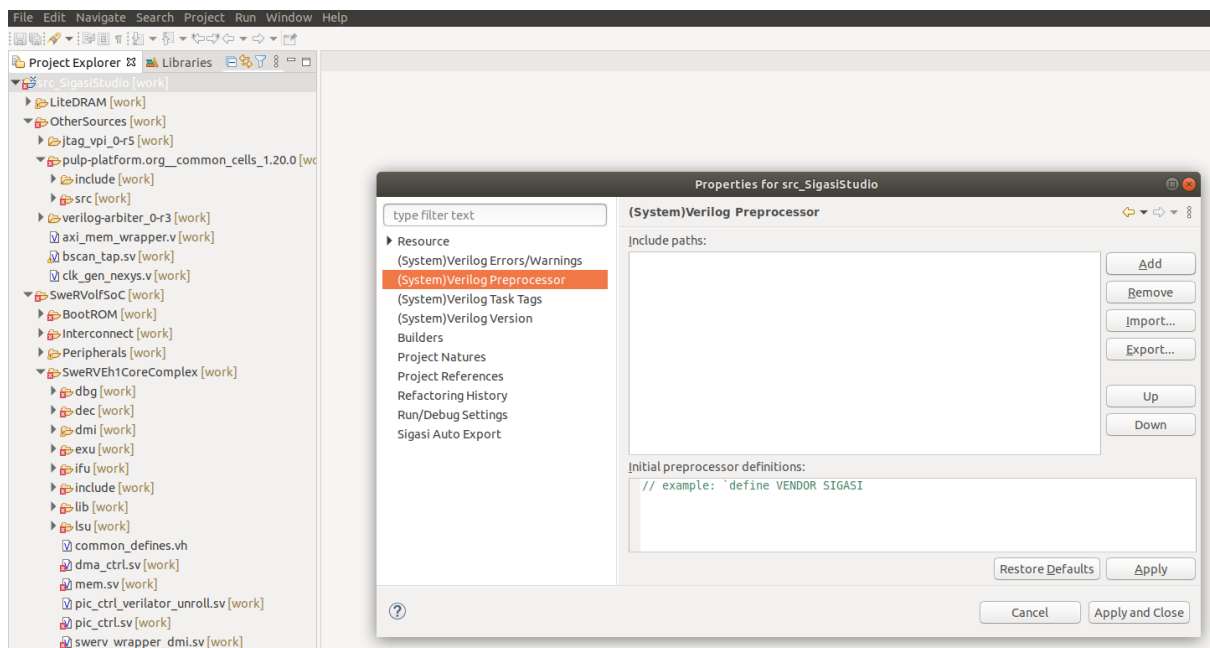
**Figura 4. Abra o diretório de fonte de RVfpga**

- O projeto abrirá com muitos erros (ver Figura 5), a maioria deles devido à falta de muitos incluir ficheiros na configuração do projeto.



**Figura 5. Erros iniciais no projeto RVfpga no Sigasi Studio.**

- No Project Explorer, clique com o botão direito em *src\_SigasiStudio* e abra a janela Properties (ver Figura 6).



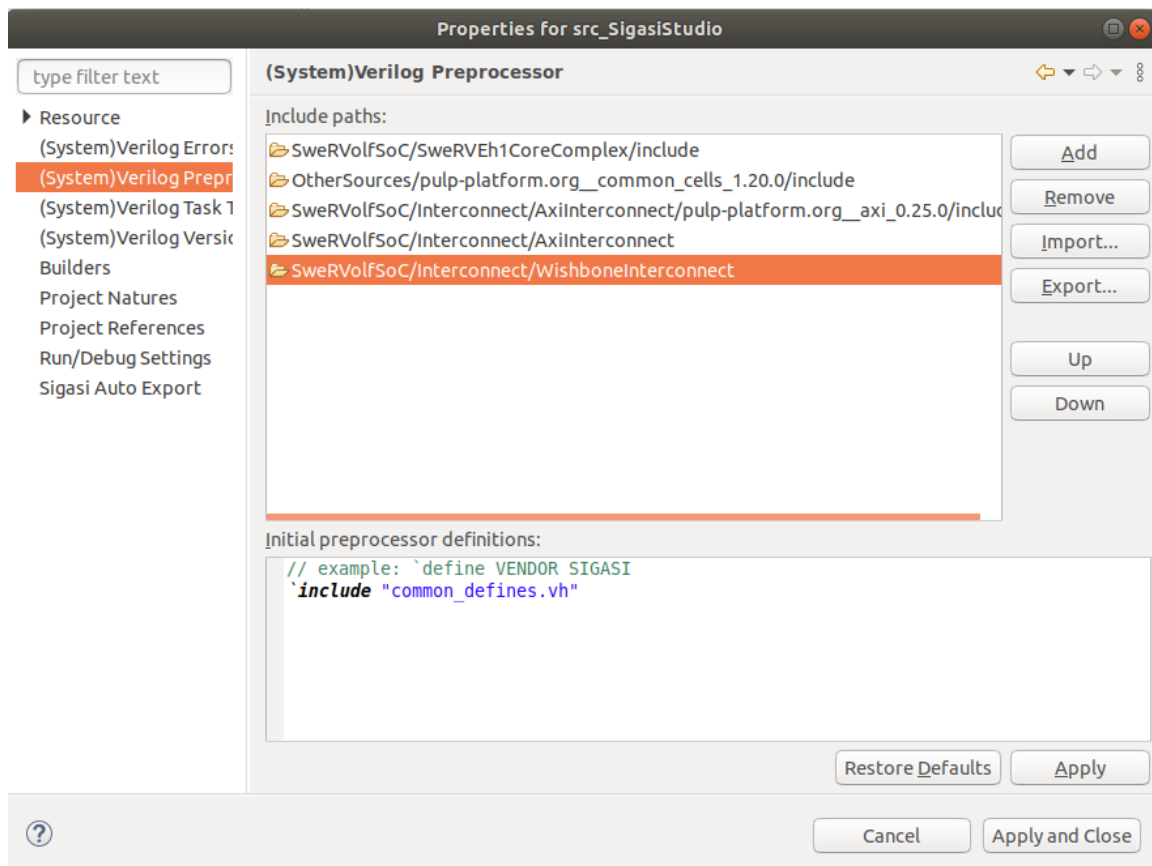
**Figura 6. Propriedades do projeto.**

7. Na janela Properties (Figura 6) selecione o “(System)Verilog Preprocessor” e adicione os seguintes caminhos (clicando no botão *Add* (Adicionar), à direita):
- *[RVfpgaPath]/RVfpga/src\_SigasiStudio/SweRVolfSoC/SweRVEh1CoreComplex/include*
  - *[RVfpgaPath]/RVfpga/src\_SigasiStudio/OtherSources/pulp-platform.org\_\_common\_cells\_1.20.0/include*
  - *[RVfpgaPath]/RVfpga/src\_SigasiStudio/SweRVolfSoC/Interconnect/AxiInterconnect/pulp-platform.org\_\_axi\_0.25.0/include*
  - *[RVfpgaPath]/RVfpga/src\_SigasiStudio/SweRVolfSoC/Interconnect/AxiInterconnect*
  - *[RVfpgaPath]/RVfpga/src\_SigasiStudio/SweRVolfSoC/Interconnect/WishboneInterconnect*

Uma vez adicionados os cinco diretórios, clicar no botão *Apply* (Aplicar).

Depois, na mesma janela, na caixa inferior (Definições iniciais do pré-processador), introduza a seguinte linha: ``include "common_defines.vh"`. Clique no botão *Apply and Close* (Aplicar e Fechar).

A Figura 7 mostra o estado final.



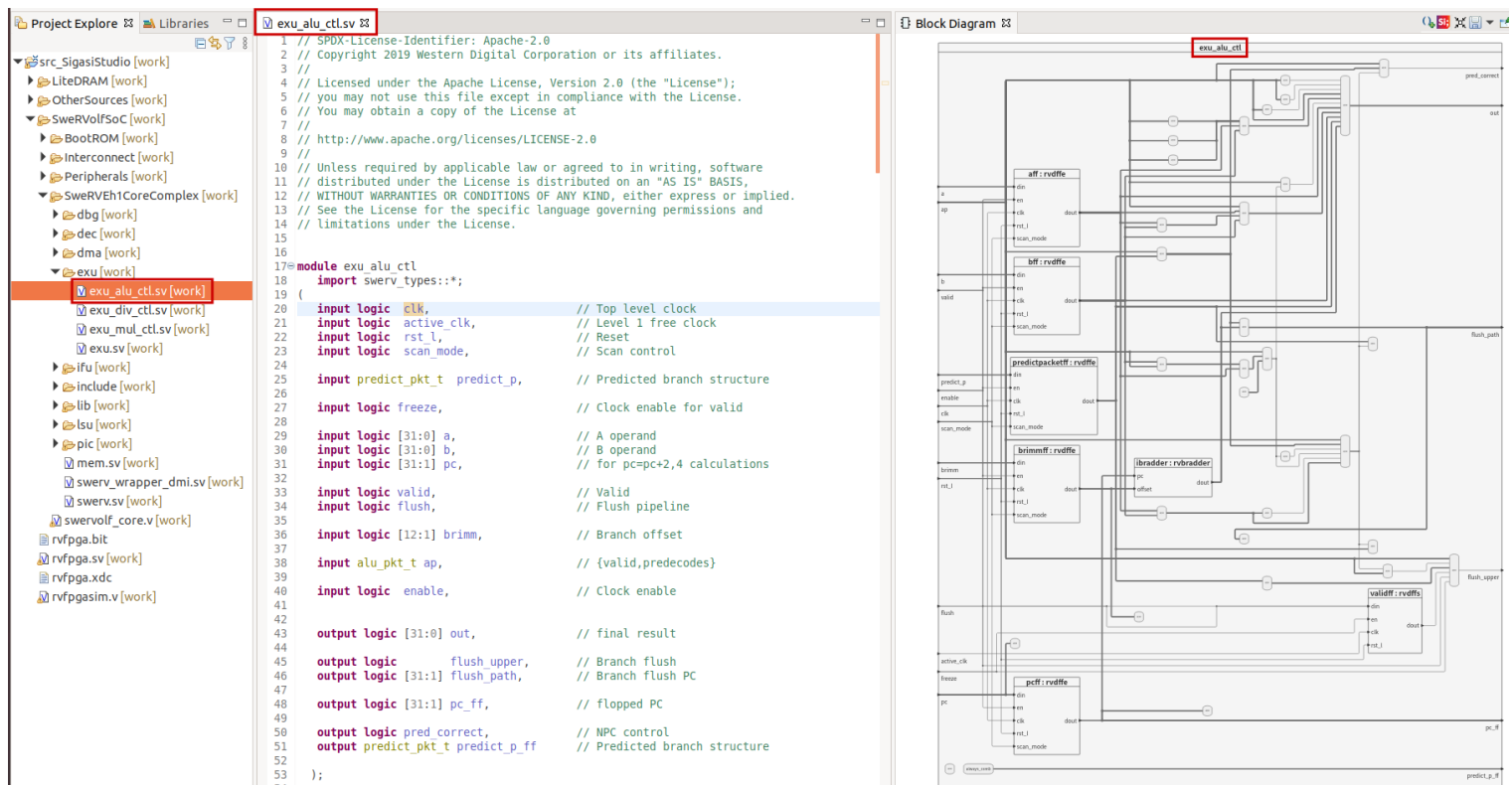
**Figura 7. Incluir directórios e ficheiros**

8. Finalmente, apague o ficheiro  
`[RVfpgaPath]/RVfpga/src_SigasiStudio/SweRVolfSoC/BootROM/sw/boot_main.vh`,  
do qual não precisamos para o nosso projeto e dá alguns erros. Pode apagá-lo no  
seu Explorador de Ficheiros ou no interior do Sigasi Studio.

Todos os erros devem ter desaparecido após estas etapas e apenas alguns avisos devem permanecer, que pode ignorar.

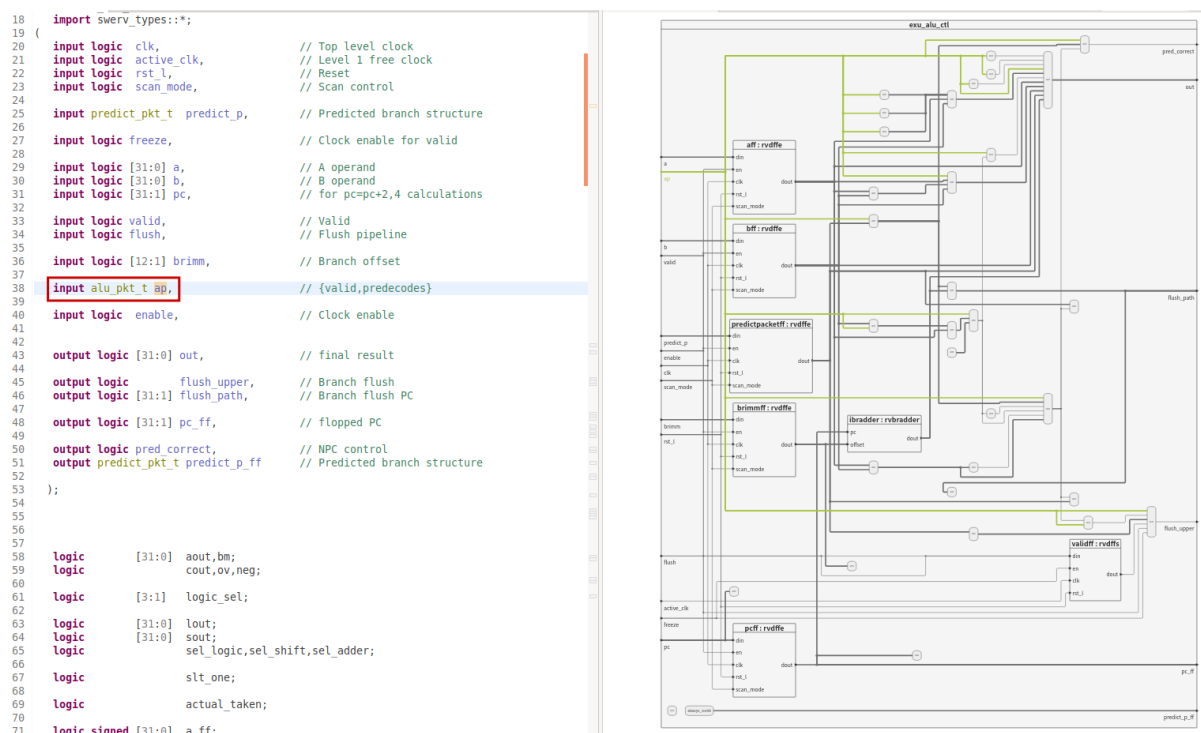
Pode começar a utilizar o Sigasi Studio para inspecionar o RVfpga SoC. Como teste, mostramos a seguir algumas funcionalidades da ferramenta:

1. No menu superior, abrir Window → Show View → Block Diagram, que abre uma nova janela na parte direita da ferramenta que lhe permite navegar graficamente através do módulo.
2. Neste laboratório, analisamos instruções aritméticas e lógicas. Estas instruções são executadas na ALU, que é implementada dentro do módulo **exu\_alu\_ctl**. Abrir esse módulo fazendo duplo clique sobre ele na janela Project Explore. Deverá ver o que mostramos na Figura 8.



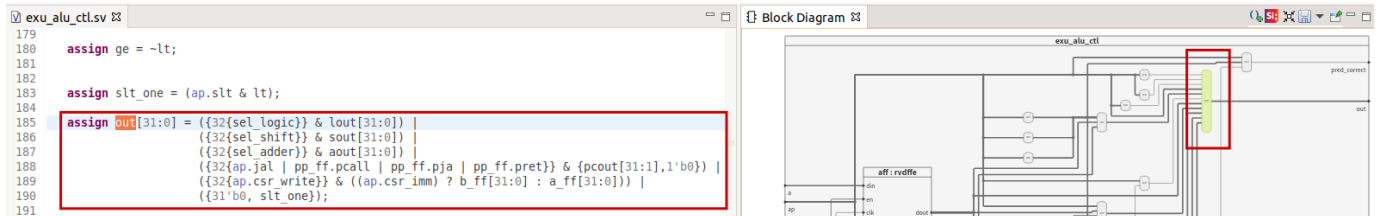
**Figura 8. Ficheiro `exu_alu_ctl.v`: Código Verilog e Diagrama de Blocos**

3. Pode destacar um sinal no diagrama, clicando com o botão direito do rato sobre ele no código Verilog e selecionando Show In → Block Diagram. Os fios associados ao sinal destacar-se-ão na janela Block Diagram, como ilustrado na Figura 9, onde o sinal `ap` está realçado.



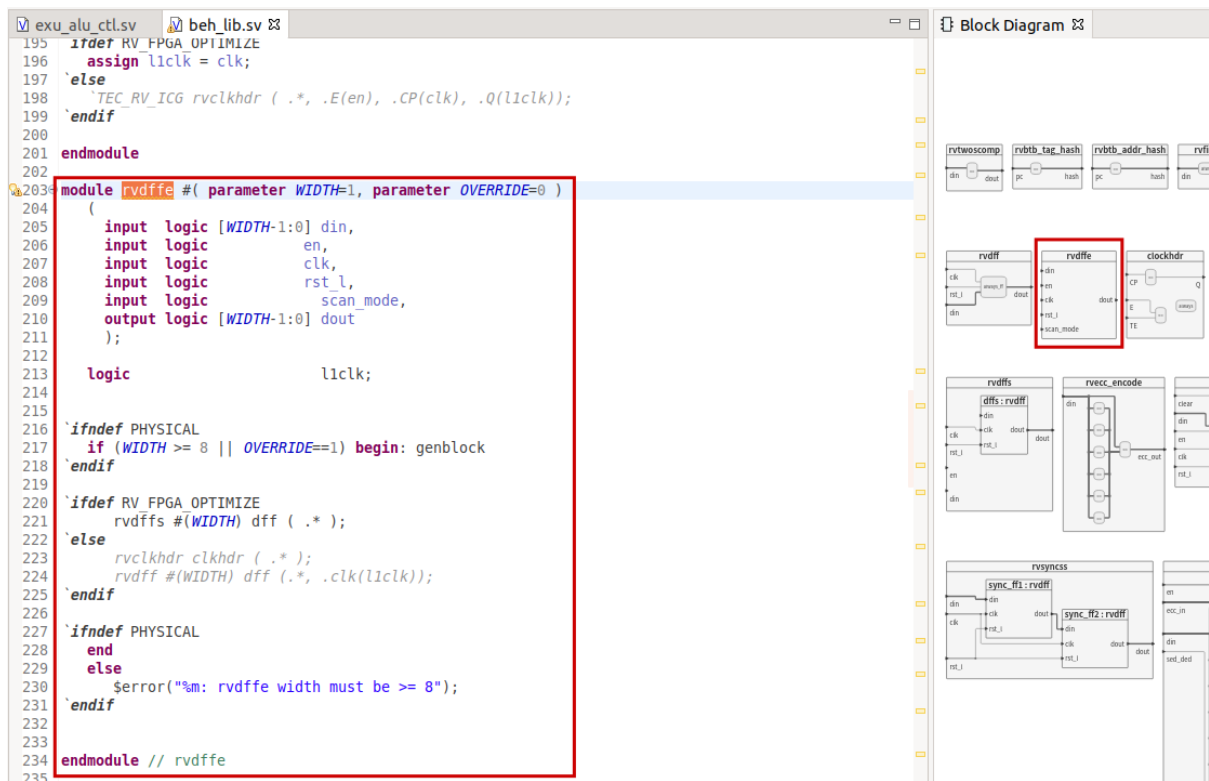
**Figura 9. Destaque do sinal `ap`**

4. Pode também procurar a implementação de um módulo combinatório no código Verilog, fazendo duplo clique no módulo no Diagrama de Blocos. Por exemplo, na Figura 10, o módulo que gera o sinal `out` é mostrado.



**Figura 10. Realce do código Verilog para o módulo combinatório que gera o sinal `out`**

5. Finalmente, abrimos uma declaração do módulo no Diagrama de Bloco, clicando com o botão direito do rato sobre a instanciação do módulo no código Verilog e selecionando Open Declaration. A Figura 11 mostra o módulo `rvdffe`, implementado no ficheiro `beh_lib.sv`.



**Figura 11. Módulo `rvdffe`**

## 2. CONFIGURAÇÃO DO PROCESSADOR SWERV EH1

### A. A. Configurar as Estruturas do Núcleo (Core)

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common\_defines.vh permite ao utilizador configurar muitas estruturas do núcleo, tais como a Cache de Instruções, o ICCM/DCCM, o Preditor de Saltos, etc. Uma configuração predefinida é fornecida no Sistema RVfpga, que pode ser alterada de duas formas diferentes:

- Pode editar manualmente os parâmetros no ficheiro *common\_defines.vh*.
- Pode usar o script *swerv.config* fornecido pela Western Digital com o pacote SweRV EH1. O uso deste script é descrito em <https://github.com/chipsalliance/Cores-SweRV/tree/branch1.8>  
No RVfpga pode encontrar o script *swerv.config* em:  
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/

Depois de ter gerado os novos ficheiros de configuração, pode resintetizar o SoC no Vivado como explicado no Laboratório 1 e obter o novo bitstream do Sistema RVfpga.

### B. Desativar o uso de Instruções Comprimidas

Nalguns casos, podemos estar interessados em desativar a utilização de instruções comprimidas. Para o efeito, temos de fazer duas alterações ao nosso projeto PlatformIO:

- Incluir as seguintes novas linhas no ficheiro *platformio.ini*:  

```
build_unflags = -Wa,-march=rv32imac -march=rv32imac
build_flags = -Wa,-march=rv32ima -march=rv32ima
extra_scripts = extra_script.py
```
- Adicionar o ficheiro *extra\_script.py* aos ficheiros fonte do projeto. Este ficheiro contém as seguintes linhas:  

```
Import("env")
env.Append(
    LINKFLAGS=[
        "-Wa,-march=rv32ima",
        "-march=rv32ima"
    ]
)
```

Na maioria dos exemplos utilizados nos laboratórios 11-20 desativaremos o uso de instruções comprimidas por uma questão de simplicidade.

### C. Habilitar/Desabilitar Características do Núcleo

A Tabela 10-1 do Manual de Referência do Programador SweRV EH1 ([https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V\\_SweRV\\_EH1\\_PRM.pdf](https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf)) mostra os bits do registo *mfdc* (em CSR 0x7F9). Este registo contém os bits de controlo de baixo nível do núcleo para desativar características específicas, tais como a execução em *pipeline* ou *dual-issue*, o Preditor de saltos, etc.. A Tabela 1 mostra as nove características principais que podem ser controladas por este registo. A definição das partes adequadas do registo em 0 ou 1, ativa ou desativa cada uma das características principais. Por exemplo, pode incluir as duas seguintes instruções

Assembly no seu programa Assembly para desativar a execução *dual-issue*, a ALU secundária e a execução em *pipeline*:

```
li t2, 0x481
csrrs t1, 0x7F9, t2
```

**Tabela 1. Registo de Controlo de Desativação de Funcionalidades (*mfdc*: CSR 0x7F9)**

31-11	Reservado	7	0: ativar a ALU secundária 1: desativar a ALU secundária	3	0: ativar predição de saltos e pilha de endereços de retorno 1: desativar predição de saltos e pilha de endereços de retorno
10	0: execução em <i>dual issue</i> 1: execução em <i>single issue</i>	6	0: escritas com efeitos secundários são pipelined 1: As escritas com efeitos secundários bloqueiam todas as transações subsequentes do barramento até resposta da escrita com valor padrão recebido	2	0: ativar a coalescência do Write Buffer 1: desativar a coalescência de Write Buffer
9	Reservado	5	0: ativar carregamentos/divisões não bloqueantes 1: desativar carregamentos/divisões não bloqueantes	1	Reservado
8	0: verificação ICCM/DCCM ECC ativada 1: verificação ICCM/DCCM ECC desativada	4	0: ativar divisão rápida 1: desativar divisão rápida	0	0: execução em <i>pipeline</i> 1: execução de instrução única

Utilizaremos configurações diferentes nos laboratórios 11-20 para comparar o desempenho, acertos/falhas da I\$, acertos/falhas do Branch Predictor, etc., do SweRV EH1 quando as diferentes características principais estiverem ativadas/desativadas.

### 3. PRINCIPAIS MÓDULOS E SINAIS DO NÚCLEO SweRV EH1

O sistema RVfpga funciona na FPGA Artix-7 inserida na placa Nexys A7, como mostra a Figura 12. A figura detalha a hierarquia do sistema, incluindo os nomes dos módulos e submódulos do Verilog. O Sistema RVfpga consiste no núcleo SweRVolf (**swervolf\_core**), o controlador DRAM (**litedram\_top**), o módulo de geração do relógio (**clk\_gen\_nexys**), e alguns módulos de interface. O núcleo do SweRVolf, por sua vez, consiste no processador SweRV EH1 (**swerv\_wrapper\_dmi**) e módulos de interface adicionais (**wb\_intercon**, **axi\_intercon**, **uart\_top**, etc.). O módulo superior para o processador SweRV EH1, **swerv\_wrapper\_dmi**, instância os dois módulos principais do núcleo: **mem** e **swerv**. No resto deste documento, listamos os submódulos e os sinais principais destes dois módulos. Note-se que pode encontrar os restantes sinais de cada módulo na interface do módulo. Nos labs 11-20 estudamos estes sinais quando analisamos o funcionamento das diferentes peças do processador.

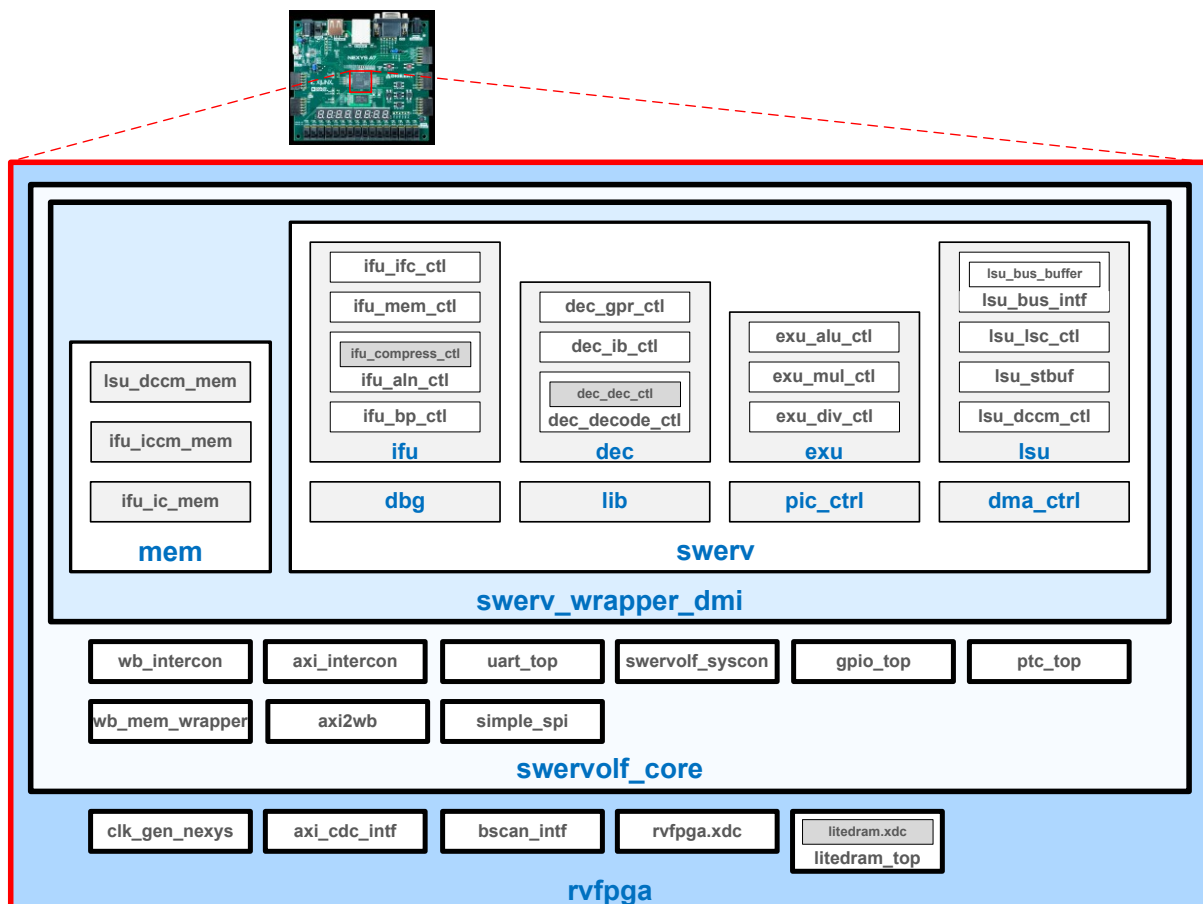
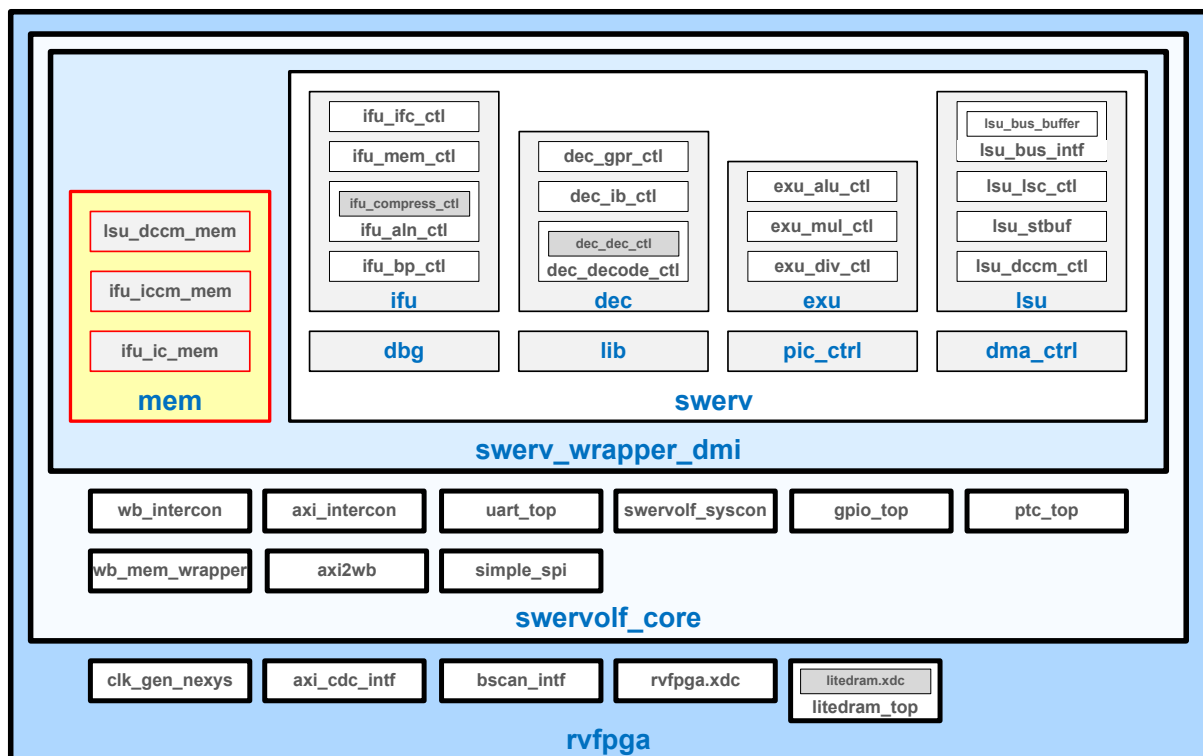


Figura 12. Hierarquia do Sistema RVfpga

## MODULO: *mem*

**FUNÇÃO:** Este módulo instância as três memórias internas disponíveis no SweRV: ICCM, DCCM, e I\$. A Tabela 2 lista os submódulos do *mem* e os seus sinais de interface.



**Figura 13. Módulo *mem* e os seus submódulos**

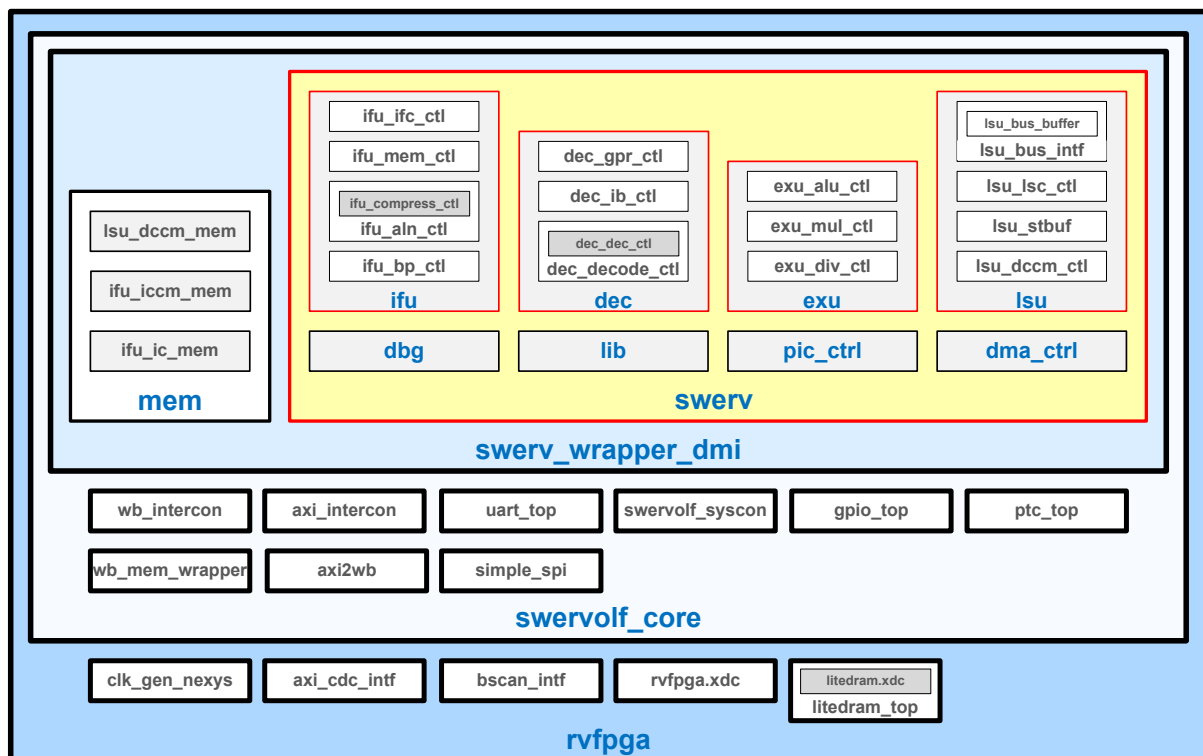
**Tabela 2. submódulos e E/S do *mem***

Unidade	E/S	Nome	Descrição
ICCM: <i>ifu_iccm_mem</i> (Contém o pacote do módulo ICCM)	Entrada	<i>iccm_wren</i>	Permitir a escrita
		<i>iccm_rden</i>	Permitir a leitura
		[`RV_ICCM_BITS-1:2] <i>iccm_rw_addr</i>	Endereço de Leitura/Escrita
		[77:0] <i>iccm_wr_data</i>	Dados de escrita
	Saída	[155:0] <i>iccm_rd_data</i>	Dados de leitura
I\$: <i>ifu_ic_mem</i> (Contém o pacote do módulo Instruction Cache Data & Tag)	Entrada	[3:0] <i>ic_wr_en</i>	Permitir a escrita
		<i>ic_rd_en</i>	Permitir a leitura
		[31:2] <i>ic_rw_addr</i>	Endereço de Leitura/Escrita
		[67:0] <i>ic_wr_data</i>	Dados para escrever na <i>lcache</i> . Com paridade.
	Saída	[135:0] <i>ic_rd_data</i>	Dados lidos da <i>lcache</i> . Etapa F2. Com paridade.
		[3:0] <i>ic_rd_hit</i>	Acerto/Falha em cada via
DCCM: <i>lsu_dccm_mem</i> (Contém o pacote do módulo)	Entrada	<i>dccm_wren</i>	Permitir a escrita
		<i>dccm_rden</i>	Permitir a leitura
		[`RV_DCCM_BITS-1:0] <i>dccm_wr_addr</i>	Endereço de escrita
		[`RV_DCCM_BITS-1:0] <i>dccm_rd_addr_lo</i>	Endereço de leitura

DCCM)		[`RV_DCCM_BITS-1:0] dccm_rd_addr_hi	Ler o endereço do banco superior (alto) quando o acesso está desalinhado
		[`RV_DCCM_FDATA_WIDTH-1:0] dccm_wr_data	Dados de escrita
	Saída	[`RV_DCCM_FDATA_WIDTH-1:0] dccm_rd_data_lo	Banco baixo de dados de leitura
		[`RV_DCCM_FDATA_WIDTH-1:0] dccm_rd_data_hi	Banco alto de dados de leitura

## MÓDULO: *swerv*

**FUNÇÃO:** Como se pode ver na Figura 14, *swerv* é o módulo de nível superior para o núcleo SweRV EH1. Instancia os principais módulos do núcleo, sobretudo: *ifu*, *dec*, *exu* e *lsu*. Tabela 3 – A Tabela 6 listar cada um dos submódulos e sinais de interface destas unidades. O módulo *swerv* comunica com o módulo *mem* através do pacote SweRV (*swerv\_wrapper\_dmi*).



**Figura 14. *swerv* e os seus submódulos**

**Tabela 3. E/S e os submódulos do *ifu* (Instruction Fetch Unit)**

Unidade	E/S	Nome	Descrição
Instruction Fetch Unit: <i>ifu</i>	Entrada/Saída	Vários sinais	Portos do ICCM de/para modulo mem
		Vários sinais	Portos do I\$ de/para modulo mem

(Este é o módulo de nível superior para o carregamento das instruções, a previsão do preditor de salto e o alinhamento)		Vários sinais	Portos IFU AXI
	Entrada	exu_flush_final	Limpar o <i>pipeline</i>
		[31:1] exu_flush_path_final	Limpar o endereço de fetch
	Saída	[31:0] ifu_i0_instr	Instrução 0. De Align para Decode
		[31:0] ifu_i1_instr	Instrução 2. De Align para Decode
		[31:1] ifu_i0_pc	Instrução 0 PC (program counter). de Align para Decode
		[31:1] ifu_i1_pc	Instrução 1 PC. deAlign para Decode
Fetch Control: <b>ifu_ifc_ctl</b> (Este módulo implementa o <i>Fetch Pipe Control</i> . Gera o próximo endereço a ir buscar à Memória de Instruções.)	Entrada	exu_flush_final	Limpar o <i>pipeline</i>
		[31:1] ifu_bp_btb_target_f2	PC alvo previsto
		[31:1] exu_flush_path_final	Caminho de descarga
	Saída	output logic [31:1] ifc_fetch_addr_f1	Endereço de carregamento em FC1
	Interno	logic [31:1] fetch_addr_next	Endereço sequencial
Controlo da Memória de Instruções (I\$ e ICCM): <b>ifu_mem_ctl</b> (Instruction Memory Control – lcache e ICCM –)	Entrada	[31:1] fetch_addr_f1	Endereço de carregamento em FC1 (ifc_fetch_addr_f1 renomeado)
	Saída	[127:0] ic_data_f2	Dados lidos em FC2 de I\$ ou ICCM para Etapa de Alinhamento
Align Control: <b>ifu_aln_ctl</b> (Alinhador de instruções)	Entrada	[127:0] ifu_fetch_data	128-bit de dados carregados na etapa de carregamento (Fetch)
	Interno	logic [127:0] q2,q1,q0	3 <i>Buffers</i>
	Saída	[31:0] ifu_i0_instr	Instrução Via 0
		[31:0] ifu_i1_instr	Instrução Via 1
		[31:1] ifu_i0_pc	Instrução Via 0 PC
		[31:1] ifu_i1_pc	Instrução Via 1 PC
Preditor de Salto: <b>ifu_bp_ctl</b>	Entrada	[31:1] ifc_fetch_addr_f1	Endereço de carregamento em FC1
	Saída	[31:1] ifu_bp_btb_target_f2	PC alvo previsto
		ifu_bp_kill_next_f2	Salto tomado/não-tomado

**Tabela 4. E/S e submódulos da dec (Decode Unit)**

Unidade	E/S	Nome	Descrição
Decode Unit: <b>dec</b> (Este é o módulo de nível superior para a Descodificação das Instruções, o Painel de Avaliação de Dependência e o acesso ao Ficheiro de Registo)	Entrada	exu_flush_final	Limpar o pipeline quando 1
		[31:0] ifu_i0_instr, [31:1] ifu_i1_instr	Instruções vindas do Align
		[31:1] ifu_i0_pc [31:1] ifu_i1_pc	PCs do Align
	Saída	alu_pkt_t i0_ap alu_pkt_t i1_ap	Sinais de controlo da ALU
		lsu_pkt_t lsu_p	Sinais de controlo da LSU
		mul_pkt_t mul_p	Sinais de controlo do MUL
		div_pkt_t div_p	Sinais de controlo do DIV
		predict_pkt_t i0_predict_p_d i1_predict_p_d	sinais de previsão para ALUs
		[31:1] dec_i0_pc_d [31:1] dec_i1_pc_d	Endereço das instruções no andar de Decode
		[31:0] gpr_i0_rs1_d [31:0] gpr_i0_rs2_d [31:0] gpr_i1_rs1_d [31:0] gpr_i1_rs2_d	dados do ficheiro de registo I0/I1 rs1/rs2
		[31:0] dec_i0_immed_d [31:0] dec_i1_immed_d	Valor imediato
		[12:1] dec_i0_br_immed_d [12:1] dec_i1_br_immed_d	Deslocamento de salto
		[31:0] i0_rs1_bypass_data_d [31:0] i0_rs2_bypass_data_d [31:0] i0_rs1_bypass_data_e2 [31:0] i0_rs2_bypass_data_e2 [31:0] i0_rs1_bypass_data_e3 [31:0] i0_rs2_bypass_data_e3	Dados de deslocamento I0 rs1/rs2
		[31:0] i1_rs1_bypass_data_d [31:0] i1_rs2_bypass_data_d [31:0] i1_rs1_bypass_data_e2	Dados de deslocamento I1 rs1/rs2

		[31:0] i1_rs2_bypass_data_e2 [31:0] i1_rs1_bypass_data_e3 [31:0] i1_rs2_bypass_data_e3	
	Interno	[31:0] dec_i0_instr_d [31:0] dec_i1_instr_d	Instruções no andar de Decode
		[31:0] dec_i0_rs1_d [31:0] dec_i0_rs2_d [31:0] dec_i1_rs1_d [31:0] dec_i1_rs2_d	Dados rs1/rs2
Instruções/PC enviado do Align para Decode: <b>dec_ib_ctl</b> (Buffers para propagar as instruções e PCs do Aligner para o Decoder)	Entrada	[31:0] ifu_i0_instr [31:0] ifu_i1_instr	instrução em Align I0/I1
		[31:1] ifu_i0_pc [31:1] ifu_i1_pc	PC de Align I0/I1
	Saída	[31:0] dec_i0_instr_d [31:0] dec_i1_instr_d	instrução em Decode I0/I1
		[31:1] dec_i0_pc_d [31:1] dec_i1_pc_d	PC em Decode I0/I1
Descodifica a instrução e calcula os valores de bypass: <b>dec_decode_ctl</b> (Descodifica as 2 instruções e calcula os valores de bypass)	Entrada	[31:1] dec_i0_pc_d [31:1] dec_i1_pc_d [31:0] exu_i0_result_e1	PC I0/I1
		[31:0] dec_i0_instr_d, [31:0] dec_i1_instr_d	Instrução no andar de Decode
	Saída	alu_pkt_t i0_a alu_pkt_t i1_ap	Sinais de controlo da ALU
		lsu_pkt_t lsu_p	Sinais de controlo da LSU
		mul_pkt_t mul_p	Sinais de controlo do MUL
		div_pkt_t div_p	Sinais de controlo do DIV
		predict_pkt_t i0_predict_p_d i1_predict_p_d	sinais de predição para a ALU
		[4:0] dec_i0_rs1_d [4:0] dec_i0_rs2_d [4:0] dec_i1_rs1_d [4:0] dec_i1_rs2_d	Índice I0/I1 rs1/rs2
		[31:0] dec_i0_immed_d [31:0] dec_i1_immed_d	Valor imediato/constante
		[12:1] dec_i0_br_immed_d [12:1] dec_i1_br_immed_d	Deslocação de salto
		[31:0] i0_rs1_bypass_data_d [31:0] i0_rs2_bypass_data_d [31:0] i0_rs1_bypass_data_e2	Dados bypass I0 rs1/rs2

		[31:0] i0_rs2_bypass_data_e2 [31:0] i0_rs1_bypass_data_e3 [31:0] i0_rs2_bypass_data_e3	
		[31:0] i1_rs1_bypass_data_d [31:0] i1_rs2_bypass_data_d [31:0] i1_rs1_bypass_data_e2 [31:0] i1_rs2_bypass_data_e2 [31:0] i1_rs1_bypass_data_e3 [31:0] i1_rs2_bypass_data_e3	Dados bypass I1 rs1/rs2
Ficheiro de registos: <b>dec_gpr_ctl</b> (Ficheiro de registos)	Entrada	[4:0] raddr0, raddr1 [4:0] raddr2, raddr3	Endereço de leitura
		[4:0] waddr0, waddr1 [4:0] waddr2	Endereço de escrita
		[31:0] wd0, wd1, wd2	Dados de escrita
		rden0, rden1, rden2, rden3	Leitura ativa
		wen0, wen1, wen2	Escritra ativa
	Saída	[31:0] rd0, rd1, rd2, rd3	Dados de leitura1

**Tabela 5. E/S e submódulos da exu (Execute Unit)**

Unidade	E/S	Nome	Descrição
Execute: <b>exu</b> (Este é o módulo de nível superior para a Execução de instruções A-L)	Entrada	alu_pkt_t i0_ap, alu_pkt_t i1_ap	Controlo ALU
		mul_pkt_t mul_p	Controlo MUL
		div_pkt_t div_p	Controlo DIV
		[31:1] dec_i0_pc_d, dec_i1_pc_d	PCs para Decode
		[31:0] gpr_i0_rs1_d [31:0] gpr_i0_rs2_d [31:0] gpr_i1_rs1_d [31:0] gpr_i1_rs2_d	I0/I1 rs1/rs2
		[31:0] dec_i0_immed_d [31:0] dec_i1_immed_d	Valores imediatos
		[12:1] dec_i0_br_immed_d [12:1] dec_i1_br_immed_d	Deslocamentos de salto (Branch offsets)
		[31:0] i0_rs1_bypass_data_d [31:0] i0_rs2_bypass_data_d [31:0] i0_rs1_bypass_data_e2 [31:0] i0_rs2_bypass_data_e2 [31:0] i0_rs1_bypass_data_e3 [31:0] i0_rs2_bypass_data_e3	Dados de bypass I0 rs1/rs2
		[31:0] i1_rs1_bypass_data_d [31:0] i1_rs2_bypass_data_d [31:0] i1_rs1_bypass_data_e2	Dados de bypass I1 rs1/rs2

		[31:0] i1_rs2_bypass_data_e2 [31:0] i1_rs1_bypass_data_e3 [31:0] i1_rs2_bypass_data_e3	
	Saída	exu_flush_final	Descarrega o pipeline quando a 1
		[31:0] exu_i0_result_e1 [31:0] exu_i1_result_e1	Resultado da ALU primária
		[31:0] exu_i0_result_e4 [31:0] exu_i1_result_e4	Resultado da ALU secundária
		[31:0] exu_mul_result_e3	Resultado MUL
		[31:0] exu_div_result	Resultado DIV
		[31:0] exu_lsu_rs1_d	Endereço de Leitura/Escreita
		[31:0] exu_lsu_rs2_d	Dados de escrita
ALU: <b>exu_alu_ctl</b> (Unidade de Aritmética e Lógica - <i>Arithmetic Logic Unit</i> )	Entrada	[31:0] a	Operando A
		[31:0] b	Operando B
		[31:1] pc	Cálculo do próximo PC (i.e., pc+2 or pc+4)
		[12:1] brimm	Deslocamento de salto
		alu_pkt_t ap	Controlo da ALU
	Saída	[31:0] out	Resultado da ALU
		flush_upper	Descarte do salto
		[31:1] flush_path	PC de destino
		[31:1] pc_ff	
Multiplicador: <b>exu_mul_ctl</b>	Entrada	[31:0] a	Operando A
		[31:0] b	Operando B
		mul_pkt_t mp	Controlo MUL
	Saída	[31:0] out	Resultado MUL
Divisor: <b>exu_div_ctl</b>	Entrada	[31:0] dividend	Numerador
		[31:0] divisor	Denominador
		div_pkt_t dp	Controlo DIV
	Saída	[31:0] out	Resultado DIV

**Tabela 6. Isu (Load/Store Unit) E/S e submódulos (incluindo as suas E/S)**

Unidade	E/S	Nome	Descrição
Unidade de leitura e escrita <i>Load/Store Unit</i> : <b>Isu</b> (Este é o módulo de nível superior para a unidade)	Entrada/Saída	Vários sinais	Portos DCCM de/para módulo de memória
		Vários sinais	Escravo <i>slave</i> DMA
		Vários sinais	Portos LSU AXI
	Entrada	[31:0] exu_lsu_rs1_d	Endereço de leitura/escrita
		[31:0] exu_lsu_rs2_d	Dados de escrita
		[11:0] dec_lsu_offset_d	Deslocamento de endereço

de leitura/escrita das instruções)		lsu_pkt_t lsu_p	Controlo LSU
	Saída	[31:0] lsu_result_dc3	Dados lidos da LSU
Cálculo do endereço: <b>lsu_lsc_ctl</b> (Controlo da LSU e cálculo dos endereços de Leitura/Escrita)	Entrada	[31:0] exu_lsu_rs1_d	Endereço de leitura/escrita
		[31:0] exu_lsu_rs2_d	Endereço de escrita
		[11:0] dec_lsu_offset_d	Deslocamento de endereço
		lsu_pkt_t lsu_p	Controlo LSU
	Saída	[31:0] lsu_addr_dc1 [31:0] end_addr_dc1	Endereço inicial/final
Controlo da DCCM: <b>lsu_dccm_ctl</b> (DCCM Control)	Entrada	[`RV_DCCM_FDATA_WIDTH-1:0] dccm_rd_data_lo	Dados lidos (banco Lo)
		[`RV_DCCM_FDATA_WIDTH-1:0] dccm_rd_data_hi	Dados lidos (banco Hi)
	Saída	dccm_wren	Escrita ativa
		dccm_rden	Leitura ativa
		[`RV_DCCM_BITS-1:0] dccm_wr_addr	Endereço de escrita
		[`RV_DCCM_BITS-1:0] dccm_rd_addr_lo	Endereço de leitura (lo)
		[`RV_DCCM_BITS-1:0] dccm_rd_addr_hi	Endereço de leitura (hi): necessário para leituras desalinhadas
		[`RV_DCCM_FDATA_WIDTH-1:0] dccm_wr_data	Dados escritos
Buffer de escrita: <b>lsu_stbuf</b> (Store Buffer)	Entrada	lsu_addr_dc3	Endereço
		[`RV_DCCM_DATA_WIDTH-1:0] store_ecc_datafn_hi_dc3	Dados a escrever (hi)
		[`RV_DCCM_DATA_WIDTH-1:0] store_ecc_datafn_lo_dc3	Dados a escrever (lo)
	Saída	[`RV_LSU_SB_BITS-1:0] stbuf_addr_any	Endereço do buffer de escrita
		[`RV_DCCM_DATA_WIDTH-1:0] stbuf_data_any	Dados do buffer de escrita

## 4. ESTRUTURAS E TIPOS PARA AGRUPAMENTO DE BITS DE CONTROLO

Abaixo está um resumo dos principais tipos de estrutura definidos no ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv` e utilizado no processador SweRV EH1 para agrupar os sinais de controlo.

- **dec\_pkt\_t**: Este é o tipo de estrutura de controlo principal e contém os sinais de controlo principais do processador, tais como `alu` (1 se uma instrução aritmética-lógica for executada, 0 caso contrário), `load` (1 se uma instrução `load` for executada, 0 caso contrário), `legal` (1 se a instrução for legal, 0 se não for), `rs1` (1 se a instrução obtém o primeiro operando de entrada do ficheiro de registos, 0 caso contrário), `imm12` (1 se a instrução utiliza um imediato de 12-bit como um operando de entrada, 0 caso contrário), etc.

Este tipo de estrutura é utilizado no interior do módulo **dec\_decode\_ctl** para gerar outros sinais de controlo. Quatro sinais deste tipo são declarados (Via-0: `i0_dp_raw`, `i0_dp`. Via-1: `i1_dp_raw`, `i1_dp`) e são utilizados para gerar os bits de controlo de outras estruturas definidas no ficheiro `swerv_types.sv`.

Estes bits são atribuídos dentro do módulo **dec\_dec\_ctl**, um módulo que é gerado automaticamente usando ferramentas de código-aberto (*coredecode* e *espresso*) e que pode ser encontrado no fim do ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv`.

- **alu\_pkt\_t**: Este tipo de estrutura contém os sinais de controlo relacionados com a operação da ALU, tais como `valid` (1 se uma instrução aritmética-lógica for executada, 0 caso contrário), `add` (1 se uma instrução `add` for executada, 0 caso contrário), `beq` (1 se uma instrução `beq` for executada, 0 caso contrário), etc. Dois sinais deste tipo, chamados `i0_ap` e `i1_ap`, estão definidos dentro do módulo **dec\_decode\_ctl**.

Estes bits estão atribuídos dentro do módulo **dec\_decode\_ctl** (implementados em: `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv`), com base nos bits da estrutura `dec_pkt_t` (ver linhas 711-770 de **dec\_decode\_ctl**).

- **reg\_pkt\_t**: Este tipo de estrutura contém os identificadores dos dois registos-fonte (campos `rs1` e `rs2`) e o registo de destino (campo `rd`). Dois sinais deste tipo, chamados `i0r` e `i1r`, estão definidos dentro do módulo **dec\_decode\_ctl**. Estes sinais são atribuídos a partir dos campos próprios do Instruction Register dentro do módulo **dec\_decode\_ctl** (ver linhas 1121-1127 deste módulo).
- **dest\_pkt\_t**: Este tipo de estrutura contém bits de controlo utilizados no andar de "Write-Back", que iremos analisar numa próxima secção. Um sinal deste tipo, chamado `dd`, é definido dentro do módulo **dec\_decode\_ctl**.
- **rets\_pkt\_t**, **br\_pkt\_t**, **br\_tlu\_pkt\_t**, e **predict\_pkt\_t**: Estes tipos de estrutura estão relacionados com as instruções de salto e preditor de salto (Branch Predictor).
- **lsu\_pkt\_t**: Este tipo de estrutura contém os sinais de controlo relacionados com a

unidade Load/Store, tal como `half` (1 se meia palavra é lida/escrita, 0 caso contrário), `load` (1 se uma instrução `load` é executada, 0 caso contrário), `valid` (1 se a instrução é válida, 0 caso contrário), etc. Um sinal deste tipo, chamado `lsu_p`, é definido dentro do módulo **`dec_decode_ctl`**.

- **`mul_pkt_t`**: Este tipo de estrutura contém os sinais de controlo relacionados com a Unidade de Multiplicação, tais como `rs1_sign` e `rs2_sign` (que determinam se os operandos de entrada são tratados como com ou sem sinal), `valid` (1 se a instrução for válida, 0 caso contrário), etc. Um sinal deste tipo, chamado `mul_p`, é definido dentro do módulo **`dec_decode_ctl`**.
- **`div_pkt_t`**: Este tipo de estrutura contém os sinais de controlo relacionados com a Unidade de Divisão, tais como `unsign` (1 se a operação não tiver sinal, 0 caso contrário), `valid` (1 se a instrução for válida, 0 caso contrário), etc. Um sinal deste tipo, chamado `div_p`, está definido dentro do módulo **`dec_decode_ctl`**.

**TAREFA:** Abra o ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv` e analise durante as descrições seguintes dos tipos de estrutura que agrupam os bits de controlo.

**TAREFA:** Veja rapidamente os módulos **`dec_decode_ctl`** e **`dec_dec_ctl`** para ver como os campos dos sinais de controlo são atribuídos com base nos 32 bits da instrução. Estes dois módulos são muito extensos e bastante complexos, pelo que a ideia não é analisá-los em pormenor. Além disso, repare que o módulo **`dec_dec_ctl`** é criado automaticamente como explicado em linhas 2482-2495 de `dec_decode_ctl.sv`.



O código ilustrado na parte superior da Figura 16 mostra o programa simples em C do Capítulo 6 - Exemplo de Código 31 - DDCARV. O código ilustrado na parte inferior da Figura 16 mostra o código Assembly gerado quando o programa C é compilado no PlatformIO com a extensão RVC ativada (note que o código Assembly é ligeiramente diferente do mostrado em [DDCARV]). Destacamos a vermelho as instruções que compõem o corpo do ciclo, que são uma combinação de instruções de 16 bits e 32 bits.

```

int scores[200];
int main(void) {
    int i;
    for (i = 0; i < 200; i = i + 1){
        scores[i] = scores[i] + 10;
    }
    return(0);
}

```

```

00000088 <main>:
88: 6789          lui      a5,0x2
8a: 12078793      addi     a5,a5,288 # 2120 <scores>
8e: 32078693      addi     a3,a5,800
92: 4398          lw       a4,0(a5)
94: 0791          addi     a5,a5,4
96: 0729          addi     a4,a4,10
98: fee7ae23      sw       a4,-4(a5)
9c: fed79be3      bne     a5,a3,92 <main+0xa>
a0: 4501          li       a0,0
a2: 8082          ret

```

**Figura 16. Exemplo de instruções comprimidas**

A Figura 17 mostra a simulação do Verilator de toda uma iteração do ciclo na Figura 16. Note que quando a instrução `addi a5,a5,4` está no andar de Align (destacado a vermelho no primeiro ciclo da figura), a instrução é extraída do pacote de 64 bits (`aligndata[63:0]`) e descomprimida a partir de uma instrução de 16 bits (**0x0791**) numa instrução de 32 bits (**0x00478793**). (O código é fornecido em `[RVfpgaPath]/RVfpga/Labs/Lab11/Compressed_C-Example` para que possa executar a sua própria simulação do Verilator.)

- No RISC-V, o *opcode* para a instrução a 16 bits `c.addi` é (ver Apêndice B de [DDCARV]):

000 | imm(1-bit) | rd/rs1 | imm(5-bits) | 01

Para que possa verificar facilmente que **0x0791** (**0000011110010001**) corresponde a: `c.addi a5,4` (lembre-se que `a5=x15`).

- Imm = 000100
- rd = rs1 = 01111 (`x15`)

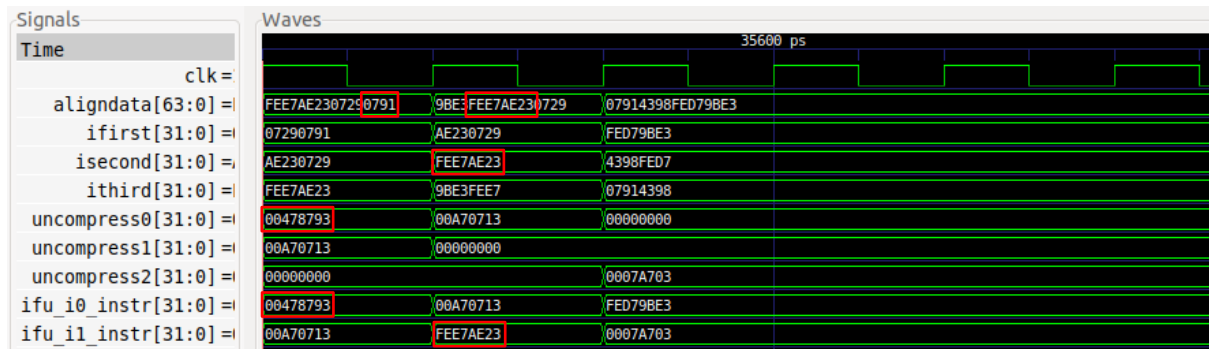
- No RISC-V, o *opcode* para a instrução de 32 bits `addi` é (ver Apêndice B de [DDCARV]):

imm(12-bits) | rs1 | 000 | rd | 0010011

Assim pode verificar facilmente que **0x00478793** (**00000000010001111000011110010011**) corresponde a: `addi a5,a5,4` (lembre-se que `a5 = x15`).

- Imm = 000000000100
- rs1 = 01111 (`x15`)
- rd = 01111 (`x15`)

No segundo ciclo apresentado na Figura 17, a instrução `sw` está alinhada. Dado que esta instrução carece da versão comprimida correspondente na arquitetura RISC-V, não necessita de ser descomprimida e é selecionada e propagada para o andar de Decode diretamente a partir do sinal `aligndata[63:0]`.



**Figura 17. Simulação do código mostrado na Figura 16**

**TAREFA:** Analise as restantes instruções do corpo do ciclo em termos de instruções comprimidas/descomprimidas.

**TAREFA:** Veja o módulo interior `ifu_compress_ctl` e tente ter uma ideia de como funciona.

## 6. AVALIAÇÕES DE DESEMPENHO (BENCHMARKS) REAIS

No diretório *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks* fornecemos três aplicações reais que irá utilizar no Lab 20 para testar as diferentes características do nosso processador SweRV EH1. Nesse laboratório poderá encontrar mais descrição sobre estes três pontos de avaliação e as diferentes versões que fornecemos para cada um deles.

- **CoreMark:** No diretório *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark\_HwCounters* pode encontrar um projeto PlatformIO que contém a avaliação CoreMark para correr no RVfpgaNexys. Utilizamos as fontes fornecidas em <https://github.com/chipsalliance/Cores-SweRV> e adaptaram-se ao nosso sistema RVfpga.
- **Dhrystone:** No diretório *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Dhrystone\_HwCounters* pode encontrar um projeto PlatformIO que contém a avaliação Dhrystone para correr em RVfpgaNexys. Utilizamos as fontes fornecidas em <https://github.com/chipsalliance/Cores-SweRV> e adaptaram-se ao nosso sistema RVfpga.
- **Image Processing:** No diretório *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/ImageProcessing\_HwCounters* pode encontrar um projeto PlatformIO que contém a aplicação que utilizamos no Lab 5 para transformar uma imagem RGB em escala de cinzentos.