

**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.

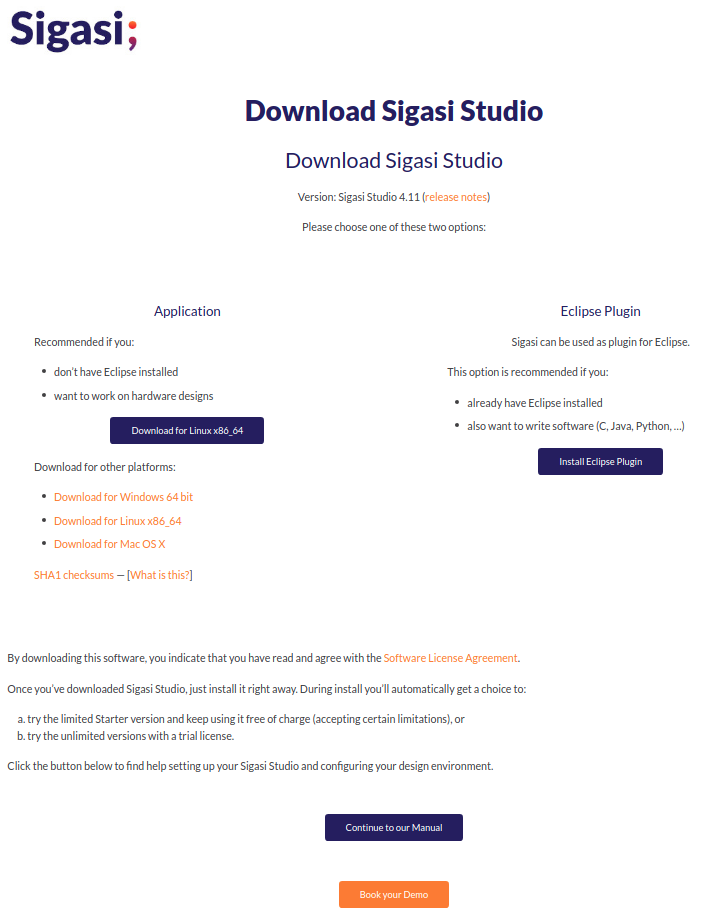


Figura 1. Ligação para descarregar, instalar e utilizar o Sigasi Studio

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/>. 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).

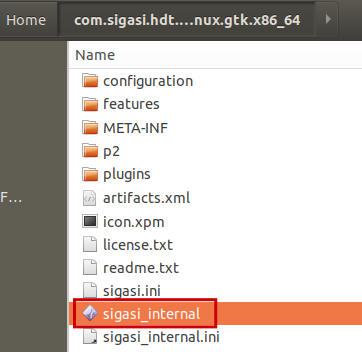


Figura 2. Abrir o Sigasi Studio

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

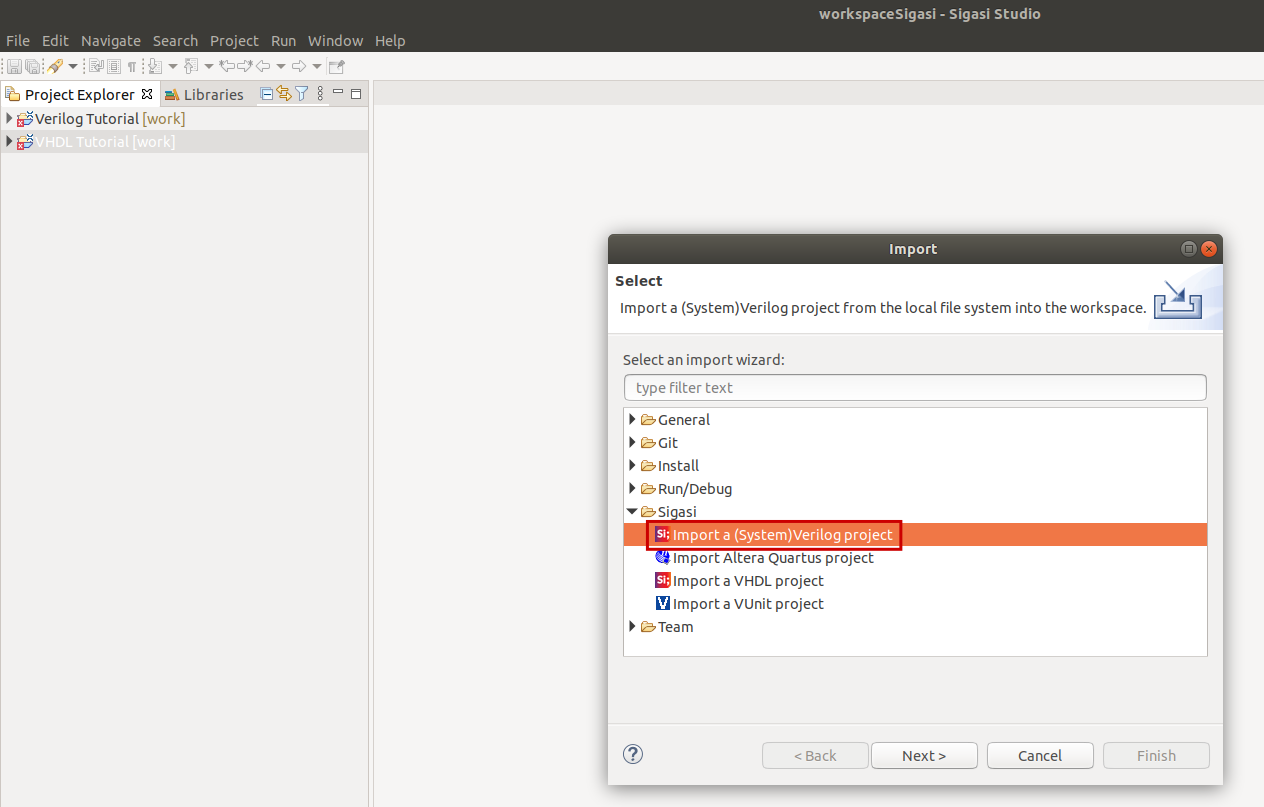


Figura 3. Importar o projeto RVfpga

1. Agora clique em “Browse…” e vá para o directório *src\_SigasiStudio* e clique Open (ver Figura 4) e, em seguida, clicar em Finish.

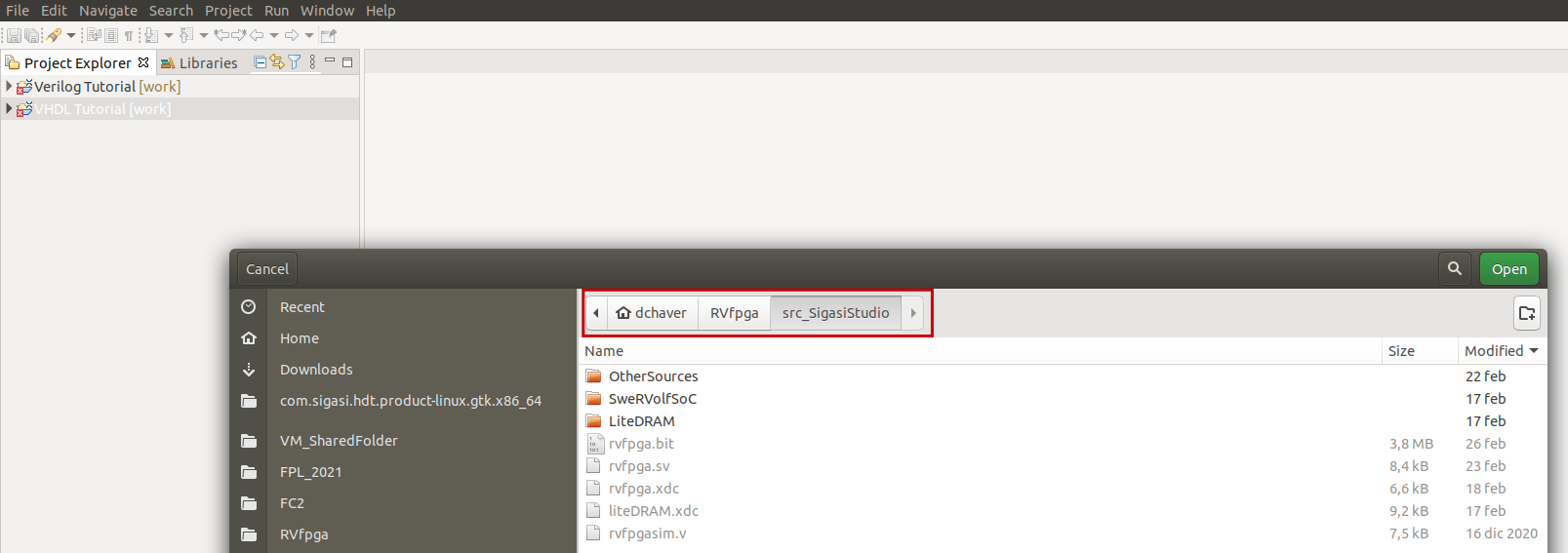


Figura 4. Abra o diretório de fonte de RVfpga

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

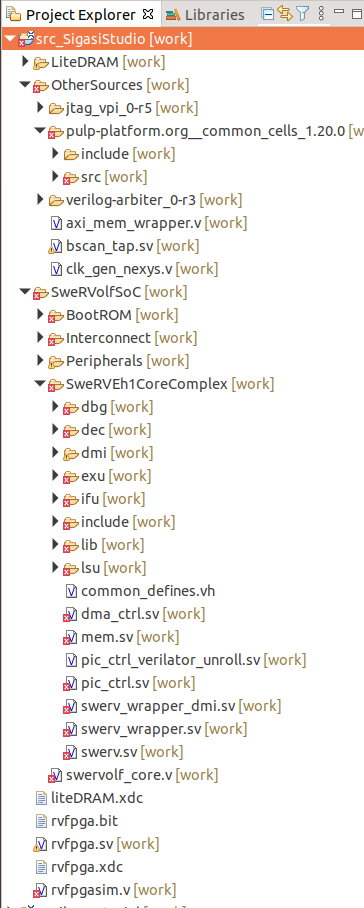


Figura 5. Erros iniciais no projeto RVfpga no Sigasi Studio.

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

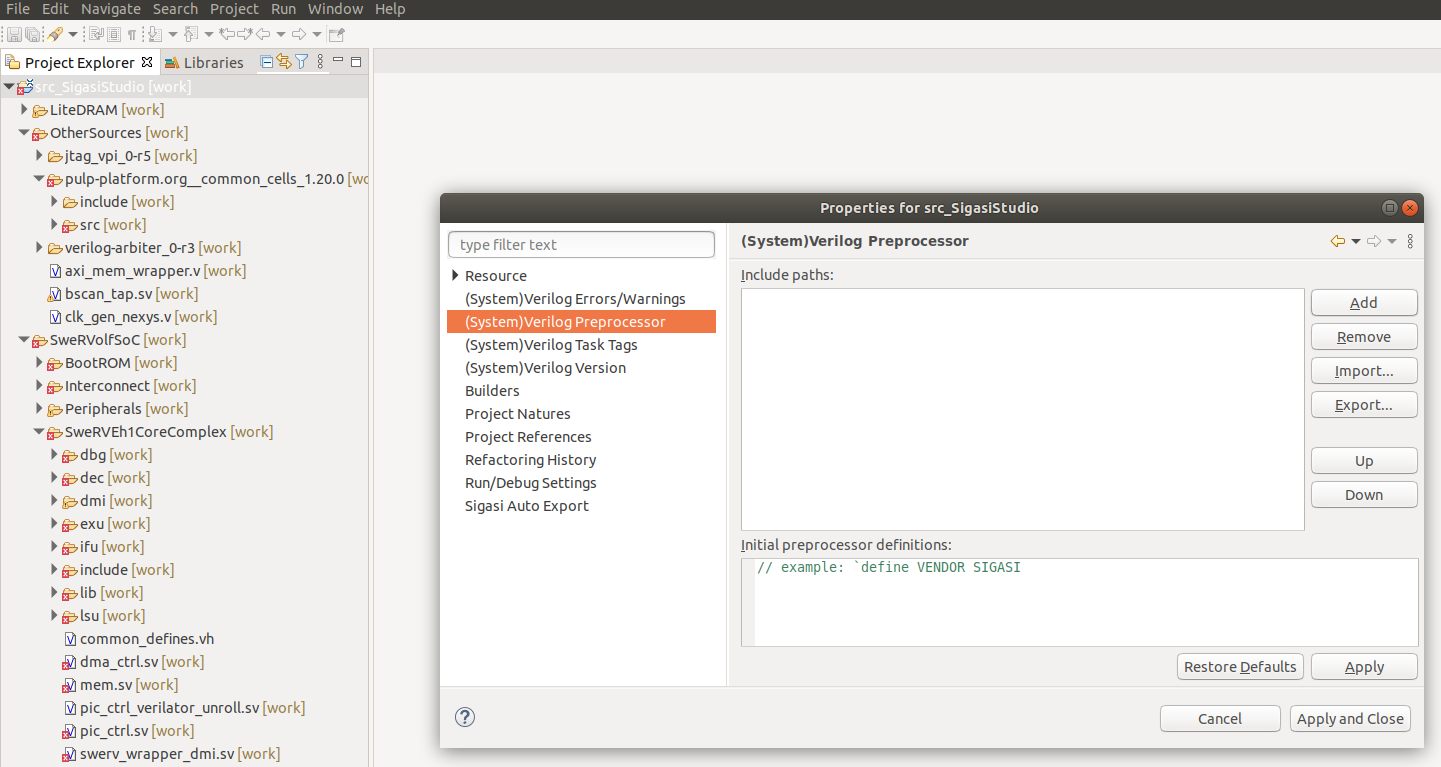


Figura 6. Propriedades do projeto.

1. 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.

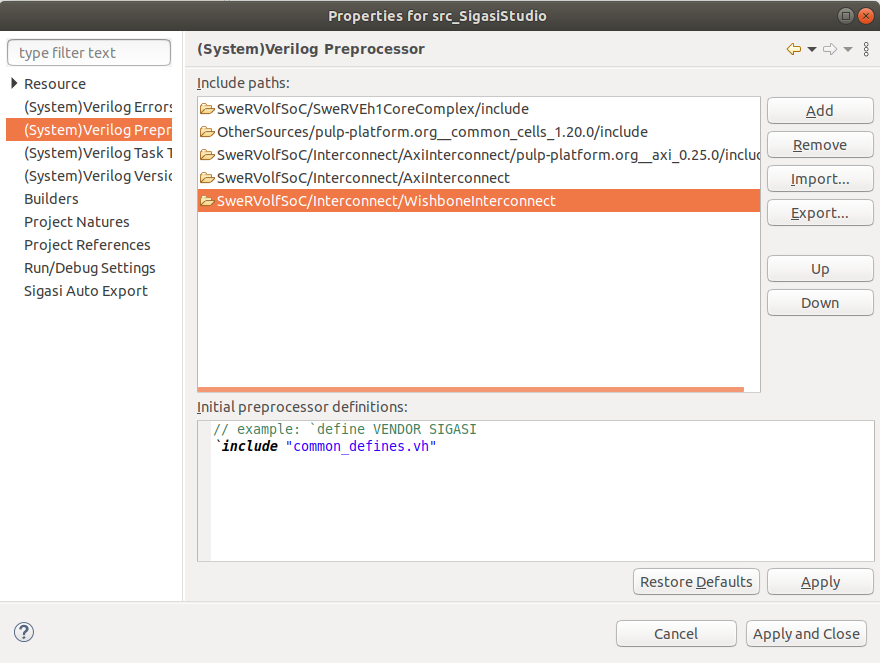


Figura 7. Incluir directórios e ficheiros

1. 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. 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.

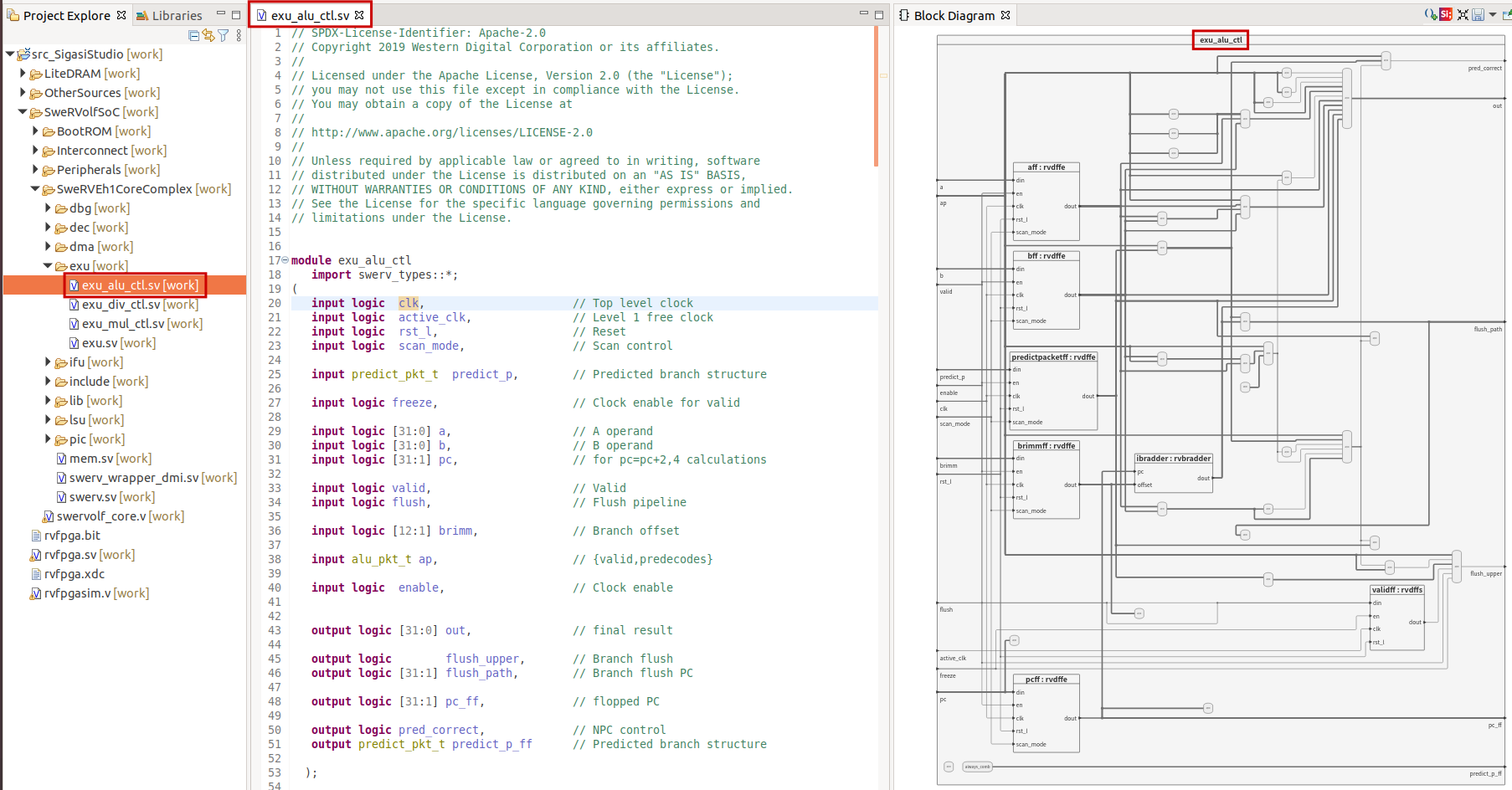


Figura 8. Ficheiro *exu\_alu\_ctl.sv*: Código Verilog e Diagrama de Blocos

1. 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.

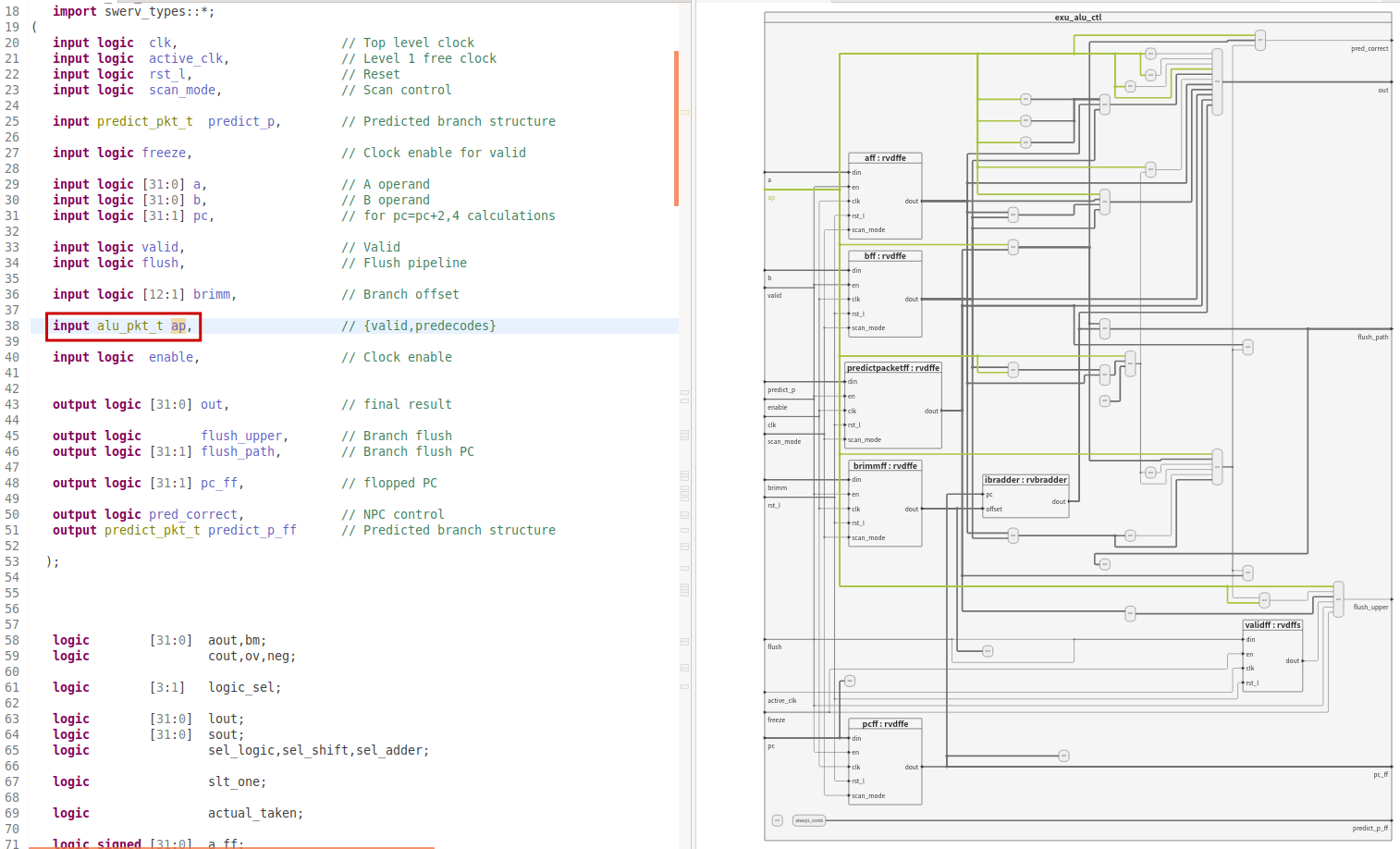


Figura 9. Destaque do sinal ap

1. Pode também procurar a implementação de um módulo combinado 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.

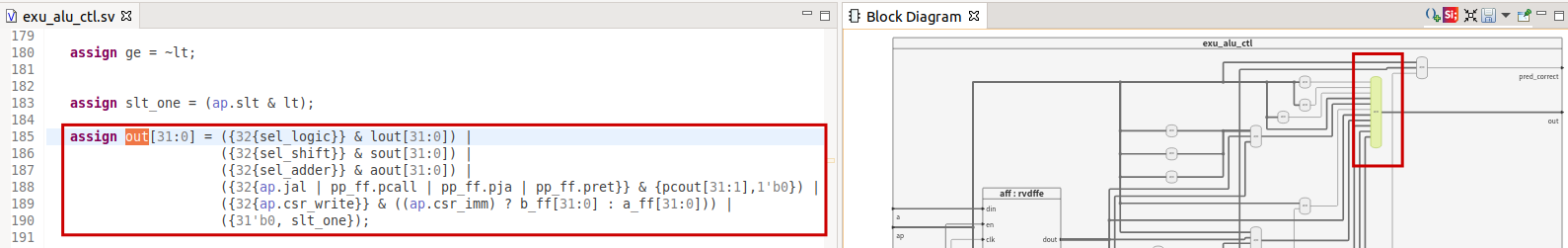


Figura 10. Realce do código Verilog para o módulo combinatório que gera o sinal out

1. 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*.

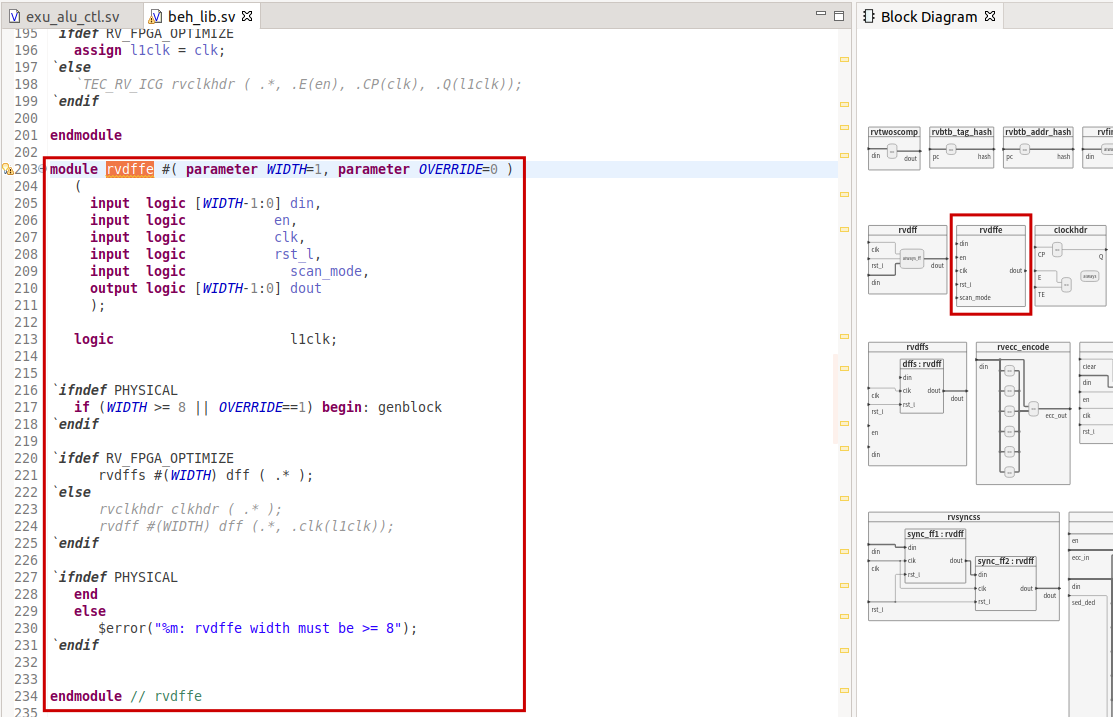


Figura 11. Modulo rvdffe

# 2. CONFIGURAÇÃO DO PROCESSADOR SWERV EH1

1. **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.

1. **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.

1. **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>) 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 Predictor 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, 0x481csrrs 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 *dual issue*  1: execução em *single issue* | 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 *pipeline*  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: ***ifu\_iccm\_mem***  (Contém o pacote do módulo ICCM) | Entrada | iccm\_wren | Permitir a escrita |
| iccm\_rden | Permitir a leitura |
| [`RV\_ICCM\_BITS-1:2] iccm\_rw\_addr | Endereço de Leitura/Escrita |
| [77:0] iccm\_wr\_data | Dados de escrita |
| Saída | [155:0] iccm\_rd\_data | Dados de leitura |
| I$: ***ifu\_ic\_mem***  (Contém o pacote do módulo Instruction Cache Data & Tag) | Entrada | [3:0] ic\_wr\_en | Permitir a escrita |
| ic\_rd\_en | Permitir a leitura |
| [31:2] ic\_rw\_addr | Endereço de Leitura/Escrita |
| [67:0] ic\_wr\_data | Dados para escrever na *Icache*. Com paridade. |
| Saída | [135:0] ic\_rd\_data | Dados lidos da Icache. Etapa F2. Com paridade. |
| [3:0] ic\_rd\_hit | Acerto/Falha em cada via |
| DCCM:  ***lsu\_dccm\_mem***  (Contém o pacote do módulo DCCM) | Entrada | dccm\_wren | Permitir a escrita |
| dccm\_rden | Permitir a leitura |
| [`RV\_DCCM\_BITS-1:0] dccm\_wr\_addr | Endereço de escrita |
| [`RV\_DCCM\_BITS-1:0] dccm\_rd\_addr\_lo | Endereço de leitura |
| [`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:  *ifu*  (Este é o módulo de nível superior para o carregamento das instruções, a previsão do preditor de salto e o alinhamento) | Entrada/Saída | Vários sinais | Portos do ICCM de/para modulo mem |
| Vários sinais | Portos do I$ de/para modulo mem |
| Vários sinais | Portos IFU AXI |
| Entrada | exu\_flush\_final | Limpar o *pipeline* |
| [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:  ***ifu\_ifc\_ctl***  (Este módulo implementa o *Fetch Pipe Control*. Gera o próximo endereço a ir buscar à Memória de Instruções.) | Entrada | exu\_flush\_final | Limpar o *pipeline* |
| [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):  ***ifu\_mem\_ctl***  (Instruction Memory Control – Icache 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:  ***ifu\_aln\_ctl***  (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 *Buffers* |
| 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:  ***ifu\_bp\_ctl*** | 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:  *dec*  (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  [31:0] i1\_rs2\_bypass\_data\_e2  [31:0] i1\_rs1\_bypass\_data\_e3  [31:0] i1\_rs2\_bypass\_data\_e3 | Dados de deslocamento I1 rs1/rs2 |
| 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:  *dec\_ib\_ctl*  (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:  ***dec\_decode\_ctl***  (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  [31:0] i0\_rs2\_bypass\_data\_e2  [31:0] i0\_rs1\_bypass\_data\_e3  [31:0] i0\_rs2\_bypass\_data\_e3 | Dados 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  [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: *dec\_gpr\_ctl*  (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: *exu*  (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  [31:0] i1\_rs2\_bypass\_data\_e2  [31:0] i1\_rs1\_bypass\_data\_e3  [31:0] i1\_rs2\_bypass\_data\_e3 | Dados de bypass I1 rs1/rs2 |
| 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/Escrita |
| [31:0] exu\_lsu\_rs2\_d | Dados de escrita |
| ALU:  *exu\_alu\_ctl*  (Unidade de Aritmética e Lógica - *Arithmetic Logic Unit*) | 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:  *exu\_mul\_ctl* | Entrada | [31:0] a | Operando A |
| [31:0] b | Operando B |
| mul\_pkt\_t mp | Controlo MUL |
| Saída | [31:0] out | Resultado MUL |
| Divisor:  *exu\_div\_ctl* | Entrada | [31:0] dividend | Numerador |
| [31:0] divisor | Denominador |
| div\_pkt\_t dp | Controlo DIV |
| Saída | [31:0] out | Resultado DIV |

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

|  |  |  |  |
| --- | --- | --- | --- |
| Unidade | E/S | Nome | Descrição |
| Unidade de leitura e escrita  *Load/Store Unit*:  *lsu*  (Este é o módulo de nível superior para a unidade de leitura/escrita das instruções) | Entrada/Saída | Vários sinais | Portos DCCM de/para módulo de memória |
| Vários sinais | Escravo *slave* 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 |
| lsu\_pkt\_t lsu\_p | Controlo LSU |
| Saída | [31:0] lsu\_result\_dc3 | Dados lidos da LSU |
| Cálculo do endereço:  *lsu\_lsc\_ctl*  (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:  ***lsu\_dccm\_ctl***  (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:  *lsu\_stbuf*  (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*.

# 5. INSTRUÇÕES COMPRIMIDAS

Embora na maioria das experiências que incluímos nos laboratórios desativemos o uso de instruções comprimidas por uma questão de simplicidade, nesta secção descrevemos e analisamos a extensão das instruções comprimidas do RISC-V (RVC) e a execução de instruções comprimidas no SweRV EH1. Obviamente, é livre de permitir a utilização de instruções comprimidas nas experiências e prolongar a análise por si próprio.

**NOTA:** Antes de iniciar este laboratório, recomendamos a leitura da Secção 6.6.5 do livro por S. Harris e D. Harris, “*Digital Design and Computer Architecture: RISC-V Edition*”, Morgan Kaufmann [DDCARV]. Alguns dos conteúdos desta secção são inspirados por esse livro.

A extensão RVC reduz o tamanho do inteiro comum e das instruções de ponto flutuante para 16 bits, reduzindo os tamanhos dos campos de controlo, imediato, e de registo e tirando partido de registos redundantes ou implícitos. Este tamanho reduzido das instruções diminui o custo, a potência e a memória necessária - tudo isto pode ser crucial para aplicações portáteis e móveis. Os nossos programas Assembly podem utilizar uma mistura de instruções comprimidas e 32 bits, dado que SweRV EH1 inclui o RVC.

No SweRV EH1 existe um módulo especificamente dedicado à descompressão de instruções: **ifu\_compress\_ctl**. Este módulo recebe uma instrução comprimida de 16 bits e produz a correspondente instrução não comprimida de 32 bits. Na Figura 15 mostramos o andar de Alinhamento (Align Stage) com um pouco mais de detalhe que no Lab 11 (deixamos ainda algumas caixas negras que pode analisar por si próprio). Três módulos **ifu\_compress\_ctl** são instanciados dentro do módulo **ifu\_aln\_ctl**, que recebem uma instrução comprimida do sinal aligndata[63:0] e devolve a instrução correspondente não comprimida nos sinais uncompress0[31:0], uncompress1[31:0] e uncompress2[31:0]. Se as instruções já estiverem no seu formato descomprimido, são fornecidas diretamente a partir do sinal aligndata[63:0].



Figura 15. Fase de Alinhamento (Align)

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].

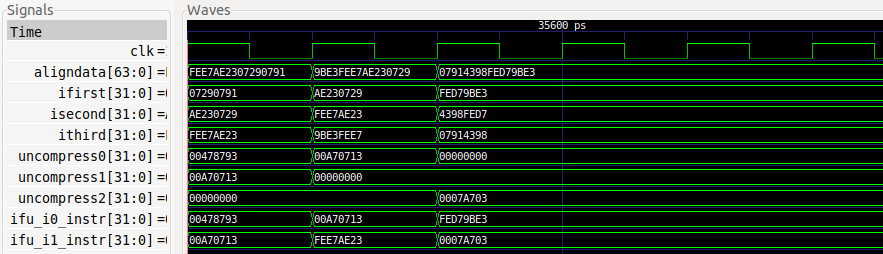


Figura 17. Simulação do código mostrado na Figura 16

**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. Utilizámos 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. Utilizámos 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 utilizámos no Lab 5 para transformar uma imagem RGB em escala de cinzentos.