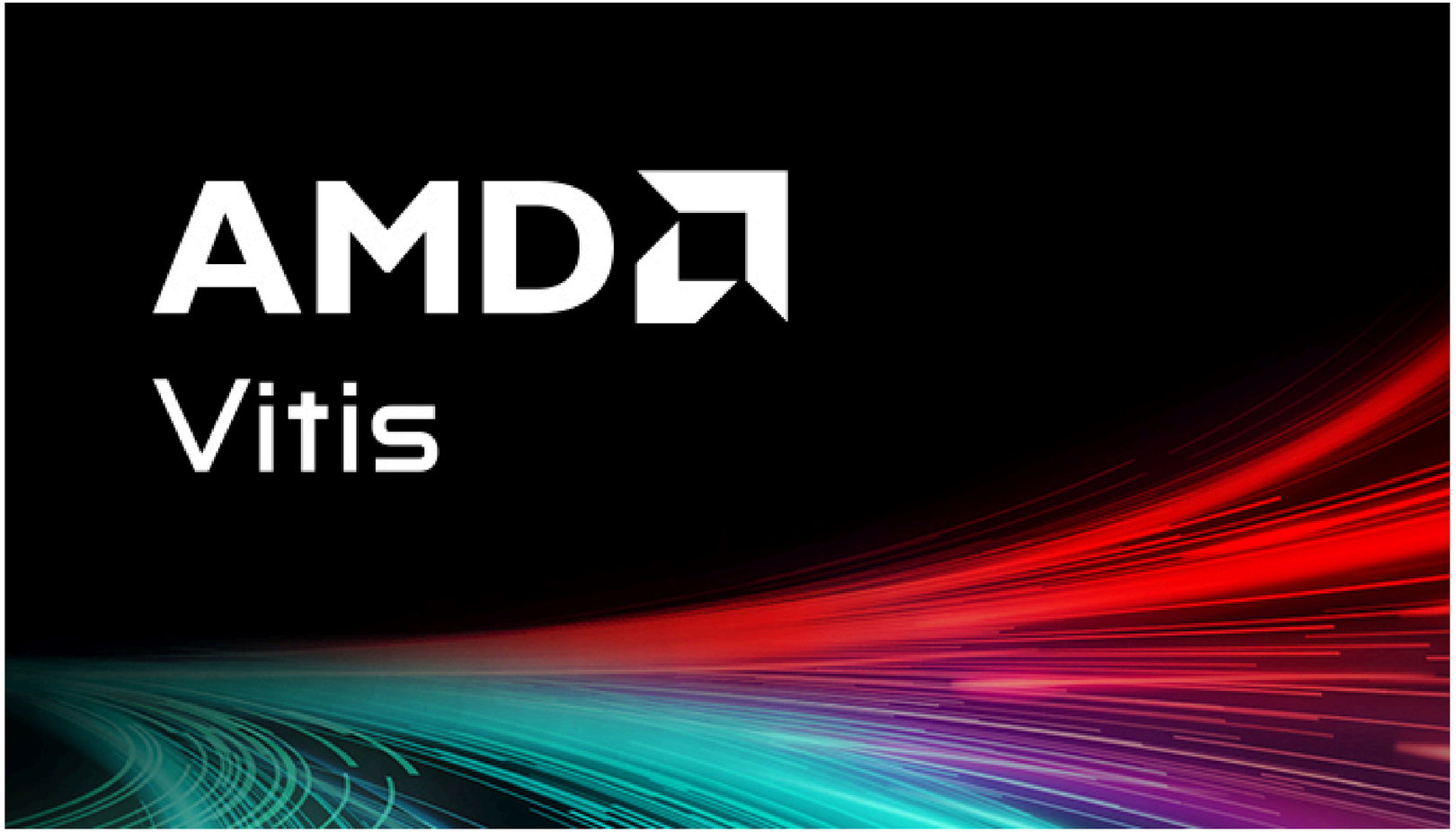
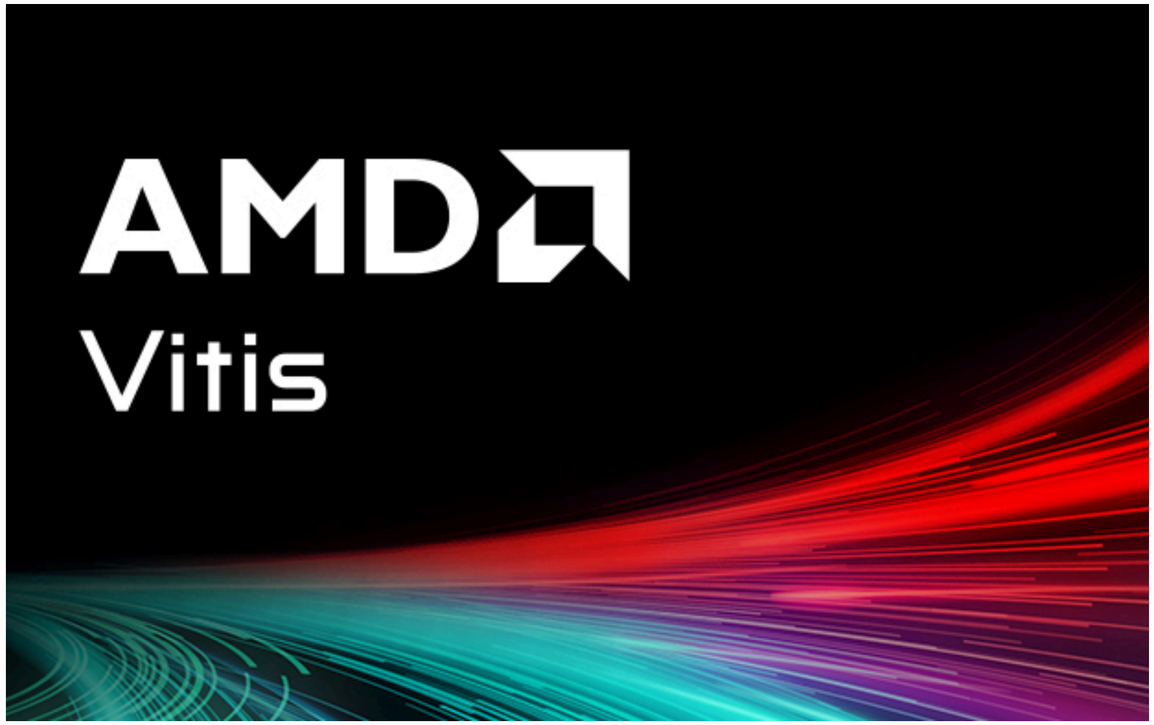


VITIS HLS

ÍNDICE

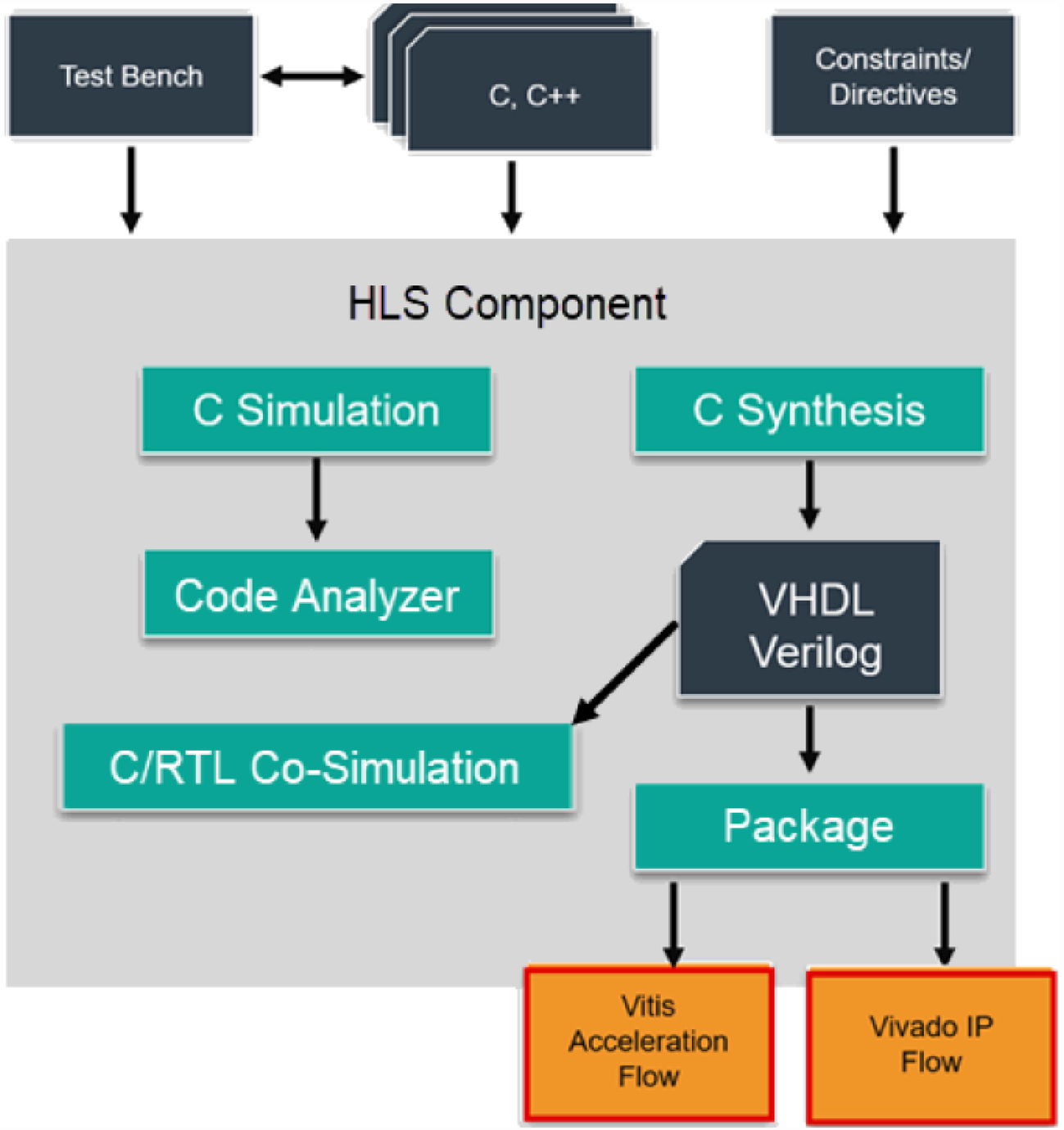
VITIS HLS



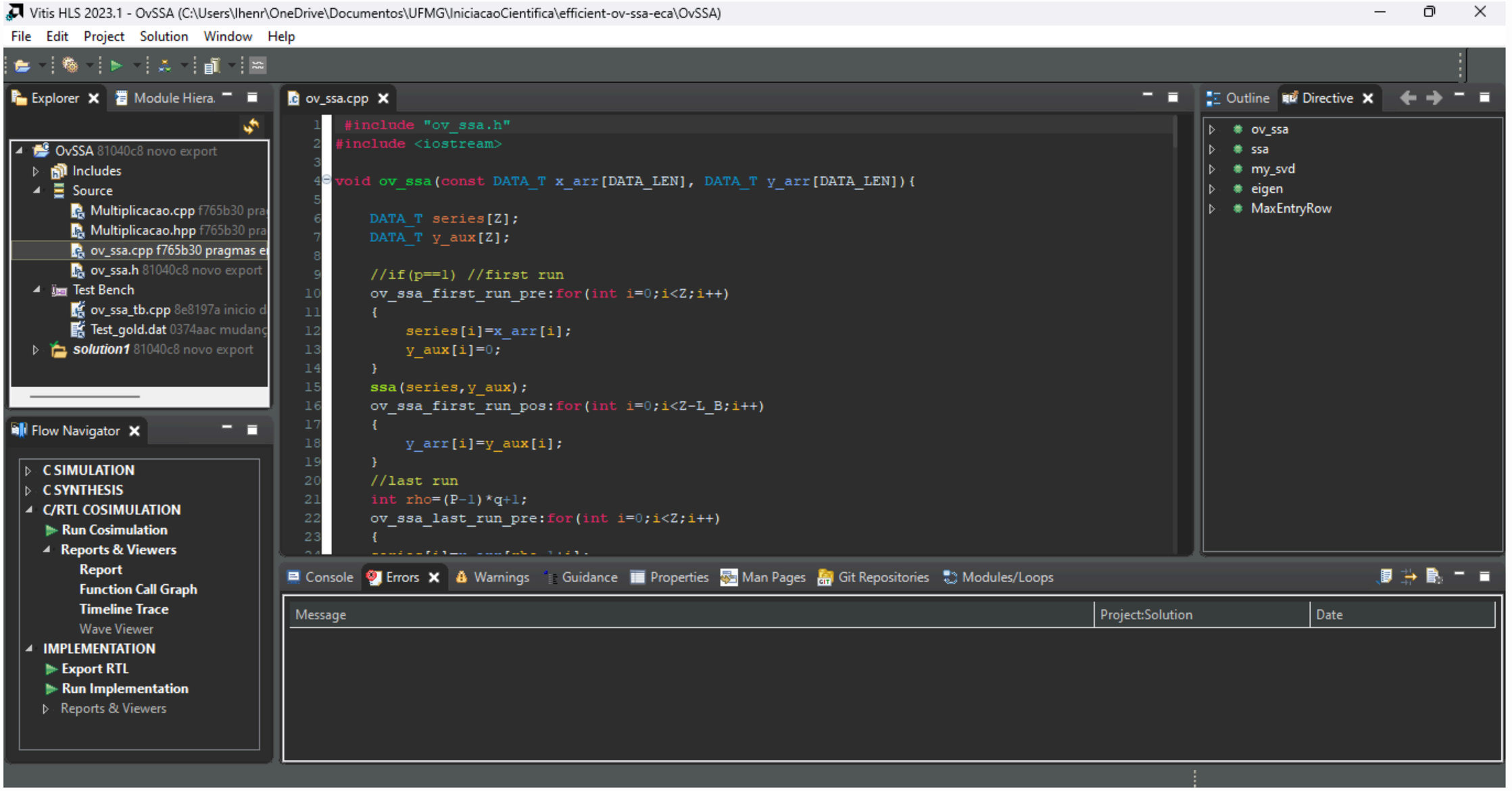
- Algoritmos complexos para FPGA;
- Linguagens de alto nível;
- Bibliotecas próprias;
- Integração com Vivado.

FLUXO DE DESENVOLVIMENTO DE COMPONENTES

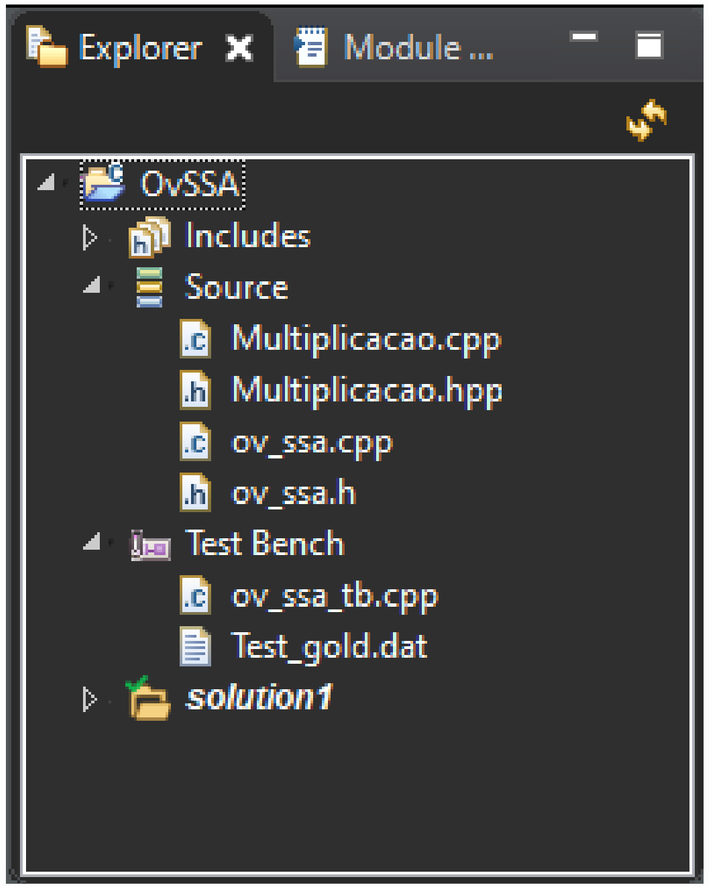
- Criação do algoritmo de acordo com princípios de design
- C-Simulation: verificação da funcionalidade do código em C/C++ com o testbench em C/C++
- Code Analyzer: análise da perfomance, da legalidade e do paralelismo do código em C/C++
- C-Synthesis: obtenção do RTL usando o compilador v++
- C/RTL Co-Simulation: verificação do código RTL gerado usando o testbench em C/C++
- Package: revisão da síntese e dos relatórios gerados

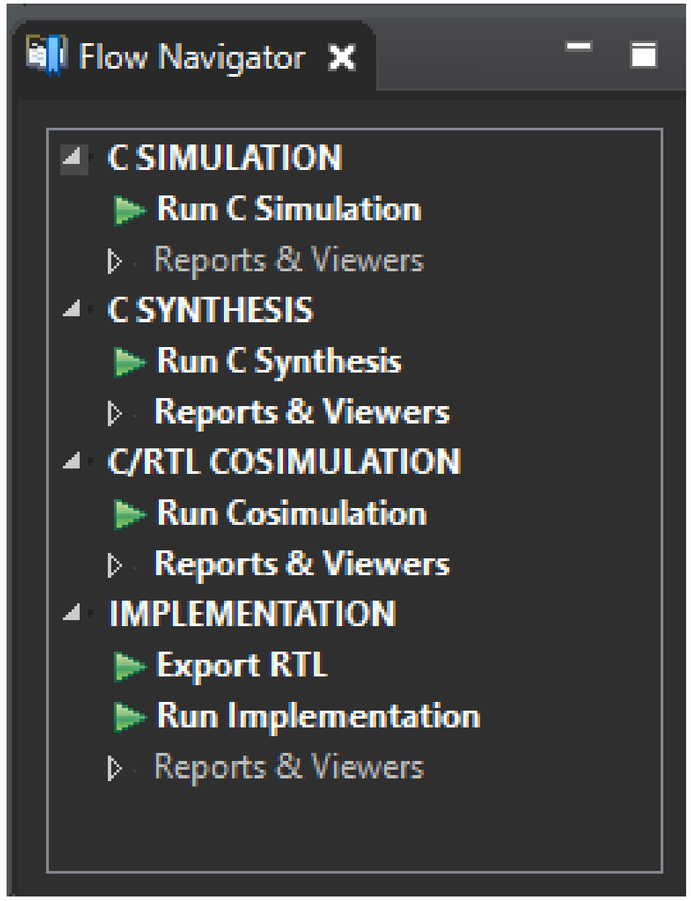

Test Bench
C, C++
Constraints/
Directives
HLS Component
C Simulation
C Synthesis
Code Analyzer
VHDL
Verilog
C/RTL Co-Simulation
Package
Vitis
Acceleration
Flow
Vivado IP
Flow

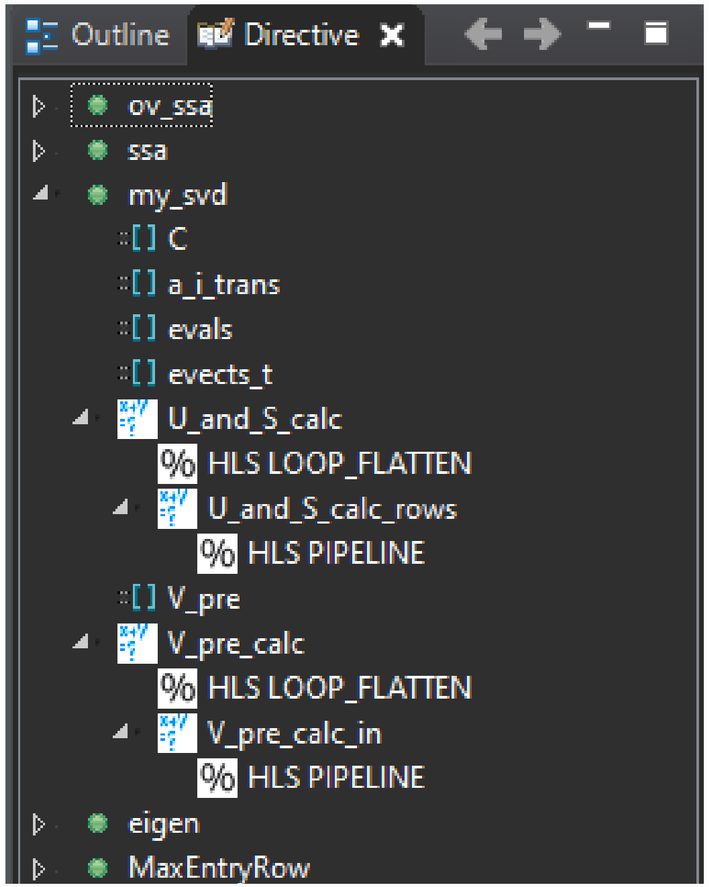
VITIS HLS - PLATAFORMA

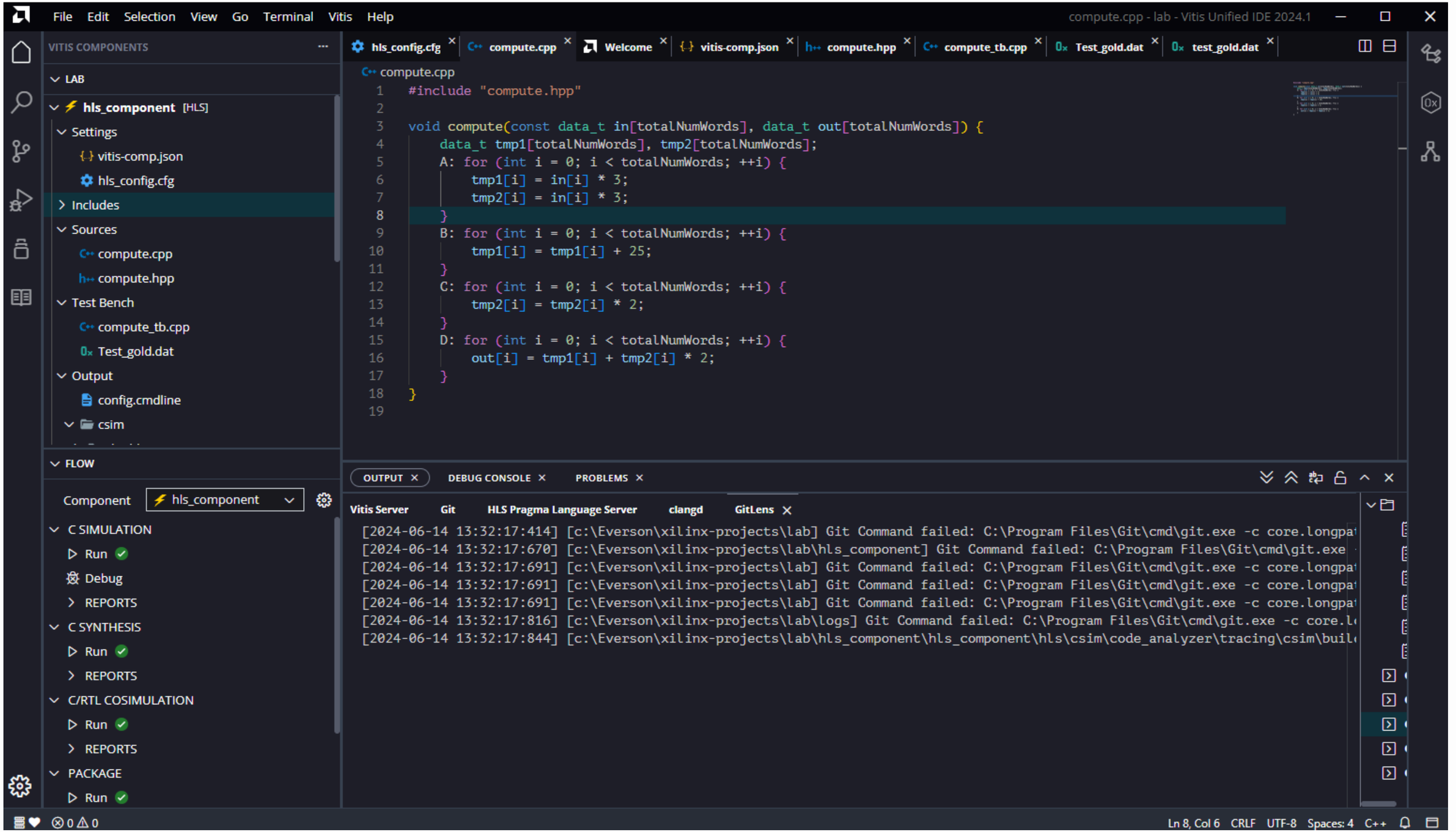


VITIS HLS - PLATAFORMA

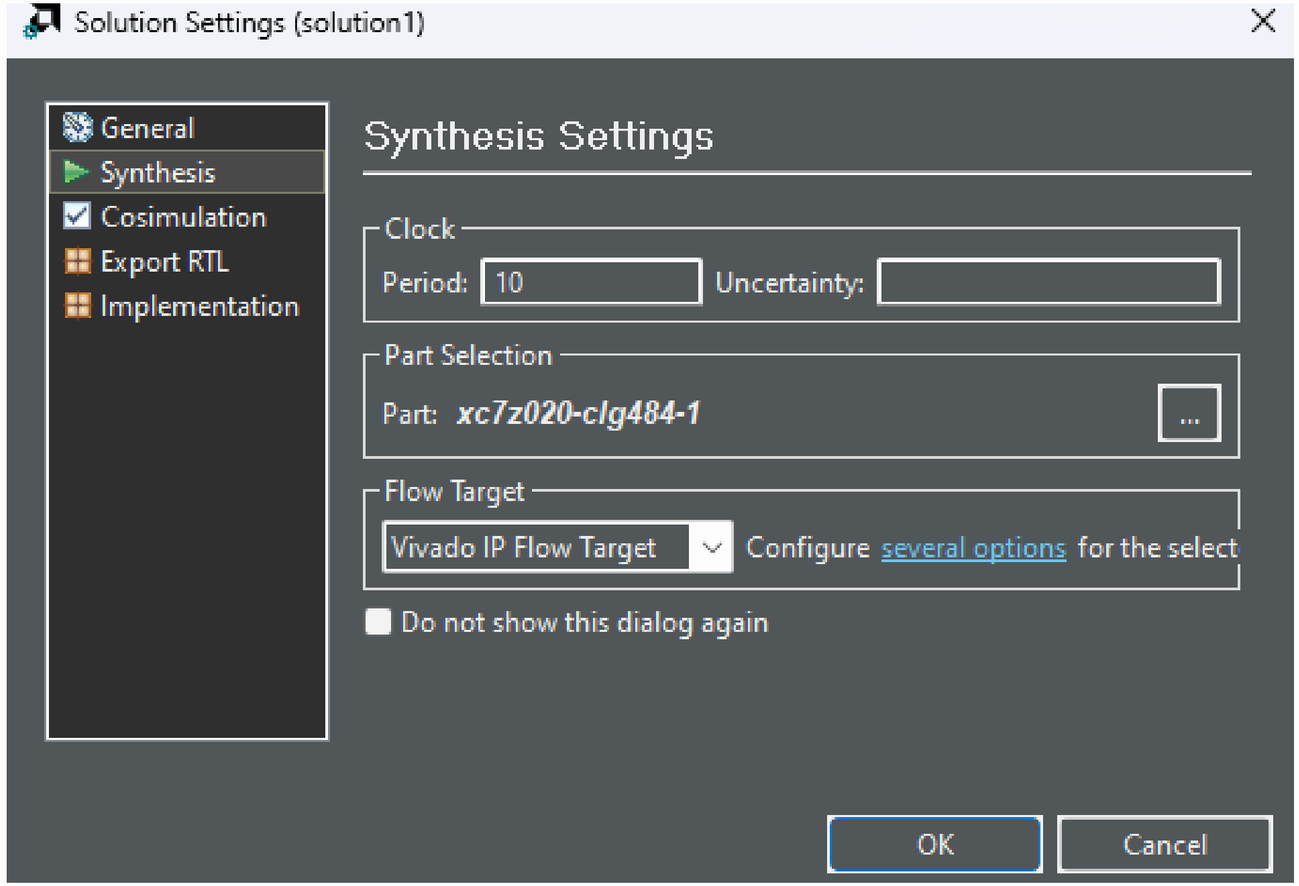




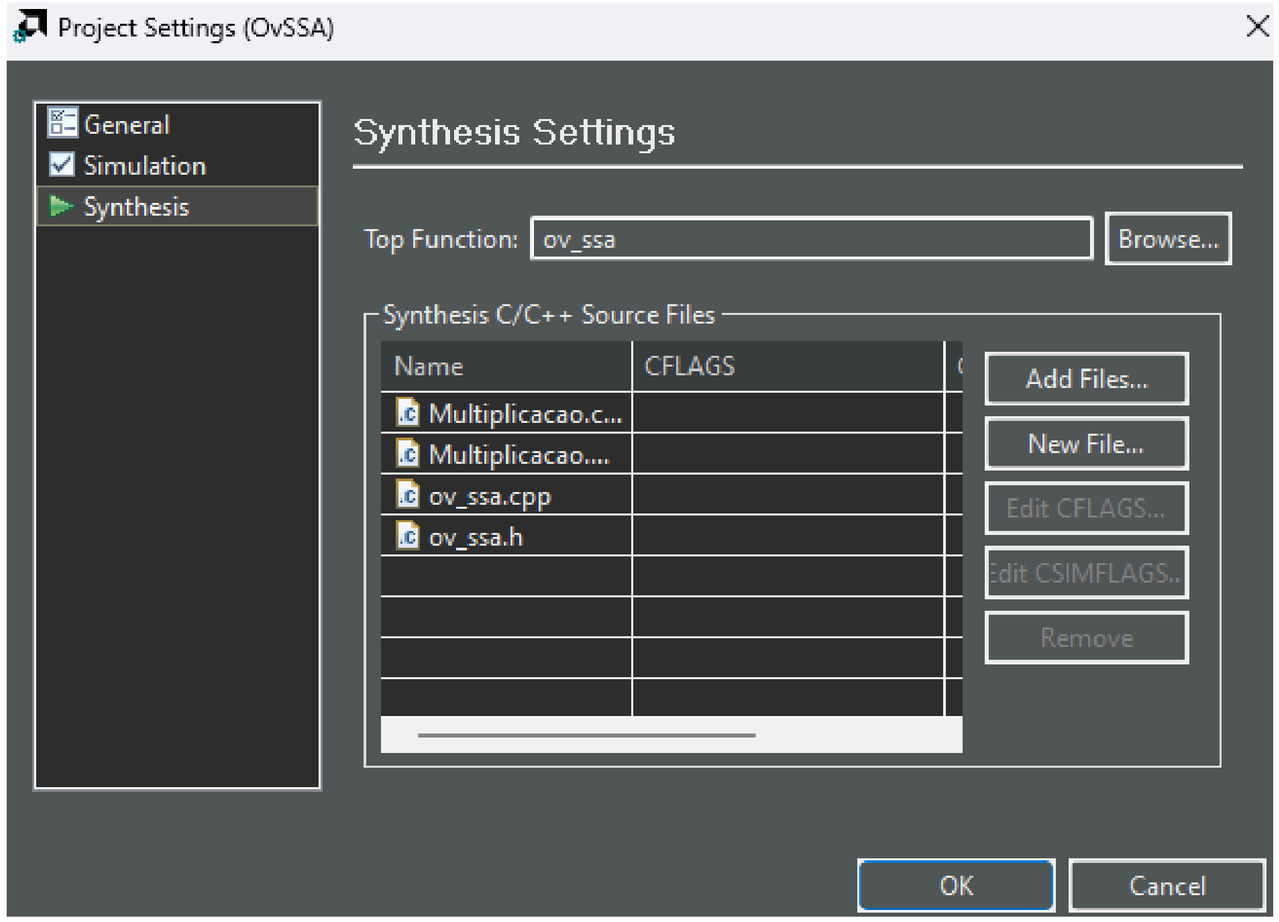



File Edit Selection View Go Terminal Vitis Help
compute.cpp - lab - Vitis Unified IDE 2024.1
VITIS COMPONENTS
LAB
hls_component [HLS]
Settings
vitis-comp.json
hls_config.cfg
Includes
Sources
compute.cpp
compute.hpp
Test Bench
compute_tb.cpp
Test_gold.dat
Output
config.cmdline
csim
FLOW
Component
hls_component
C SIMULATION
Run
Debug
REPORTS
C SYNTHESIS
Run
REPORTS
C/RTL COSIMULATION
Run
REPORTS
PACKAGE
Run
hls_config.cfg
compute.cpp
Welcome
vitis-comp.json
compute.hpp
compute_tb.cpp
Test_gold.dat
test_gold.dat
compute.cpp
#include "compute.hpp"
void compute(const data_t in[totalNumWords], data_t out[totalNumWords]) {
data_t tmp1[totalNumWords], tmp2[totalNumWords];
A: for (int i = 0; i < totalNumWords; ++i) {
tmp1[i] = in[i] * 3;
tmp2[i] = in[i] * 3;
}
B: for (int i = 0; i < totalNumWords; ++i) {
tmp1[i] = tmp1[i] + 25;
}
C: for (int i = 0; i < totalNumWords; ++i) {
tmp2[i] = tmp2[i] * 2;
}
D: for (int i = 0; i < totalNumWords; ++i) {
out[i] = tmp1[i] + tmp2[i] * 2;
}
}
OUTPUT
DEBUG CONSOLE
PROBLEMS
Vitis Server
Git
HLS Pragma Language Server
clangd
GitLens
[2024-06-14 13:32:17:414] [c:\Everson\xilinx-projects\lab] Git Command failed: C:\Program Files\Git\cmd\git.exe -c core.longpat
[2024-06-14 13:32:17:670] [c:\Everson\xilinx-projects\lab\hls_component] Git Command failed: C:\Program Files\Git\cmd\git.exe
[2024-06-14 13:32:17:691] [c:\Everson\xilinx-projects\lab] Git Command failed: C:\Program Files\Git\cmd\git.exe -c core.longpat
[2024-06-14 13:32:17:691] [c:\Everson\xilinx-projects\lab] Git Command failed: C:\Program Files\Git\cmd\git.exe -c core.longpat
[2024-06-14 13:32:17:691] [c:\Everson\xilinx-projects\lab] Git Command failed: C:\Program Files\Git\cmd\git.exe -c core.longpat
[2024-06-14 13:32:17:816] [c:\Everson\xilinx-projects\lab\logs] Git Command failed: C:\Program Files\Git\cmd\git.exe -c core.l
[2024-06-14 13:32:17:844] [c:\Everson\xilinx-projects\lab\hls_component\hls_component\hls\csim\code_analyzer\tracing\csim\buil
0 0
Ln 8, Col 6 CRLF UTF-8 Spaces: 4 C++

VITIS HLS - PLATAFORMA



VITIS HLS - PLATAFORMA



VITIS HLS - PLATAFORMA

Synthesis Summary Report of 'ov_ssa'

General Information

Timing Estimate

Performance & Resource Estimates

Modules Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
ov_ssa				-1,16	-	-	-	-	-	no	0	55	13835	21671	0
ov_ssa_Pipeline_ov_ssa_first_run_pre				-	10	100.000	-	10	-	no	0	0	11	64	0
ssa	II Violation			-1,16	-	-	-	-	-	no	0	55	13648	20839	0
ov_ssa_Pipeline_ov_ssa_last_run_pre				-	10	100.000	-	10	-	no	0	0	11	64	0
ov_ssa_first_run_pos				-	5	50.000	2	1	5	yes	-	-	-	-	-
ov_ssa_last_run_pos				-	17	170.000	2	1	17	yes	-	-	-	-	-
ssa_iterations				-	-	-	-	-	3	no	-	-	-	-	-

Performance Pragma

HW Interfaces

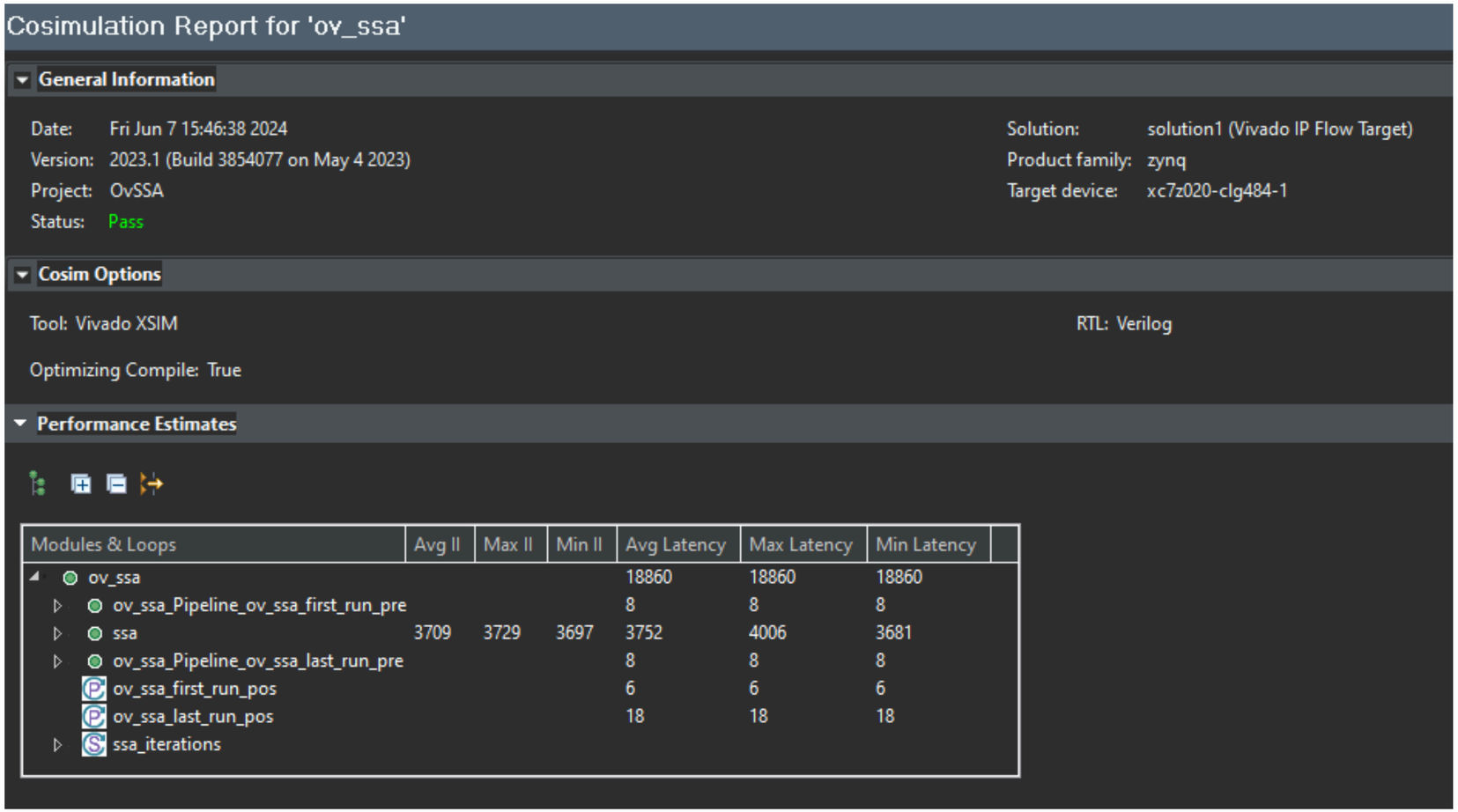
SW I/O Information

Pragma Report

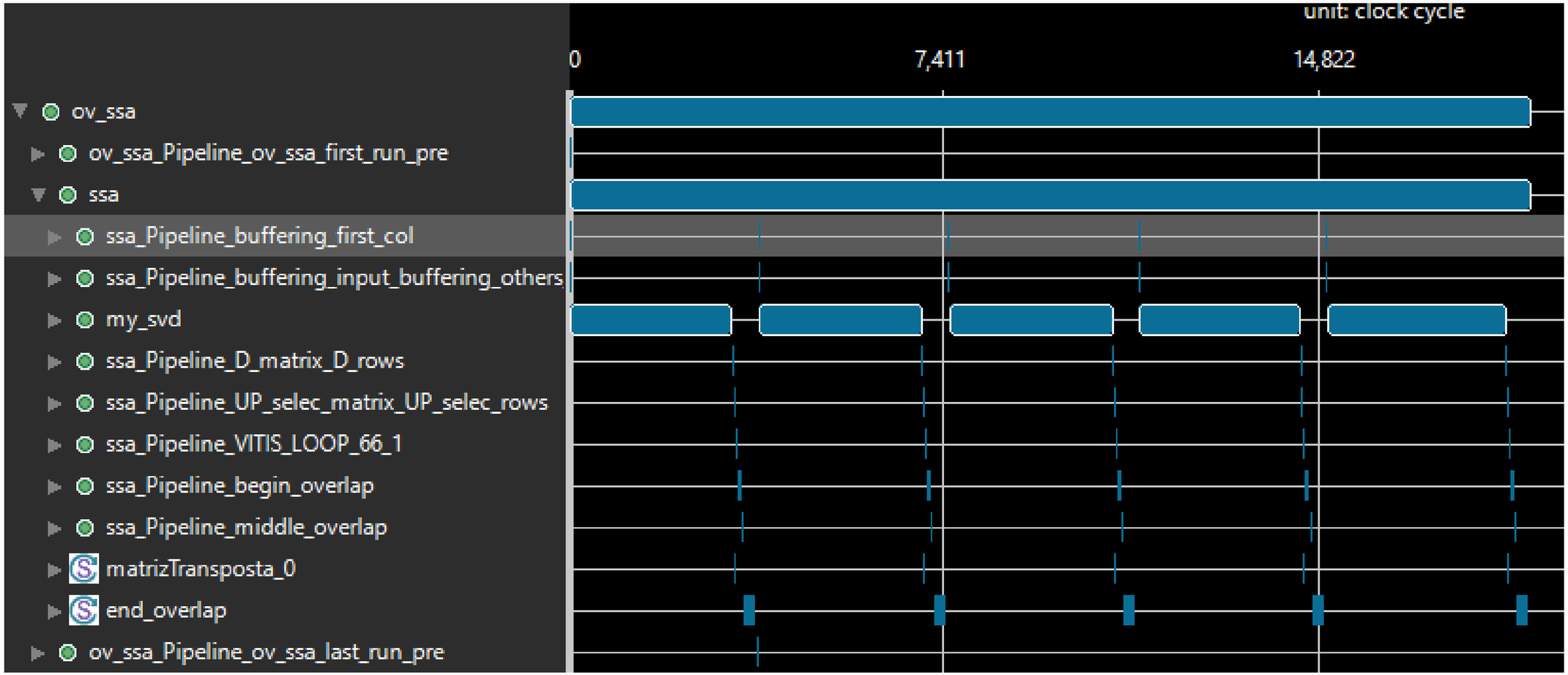
Bind Op Report

Bind Storage Report

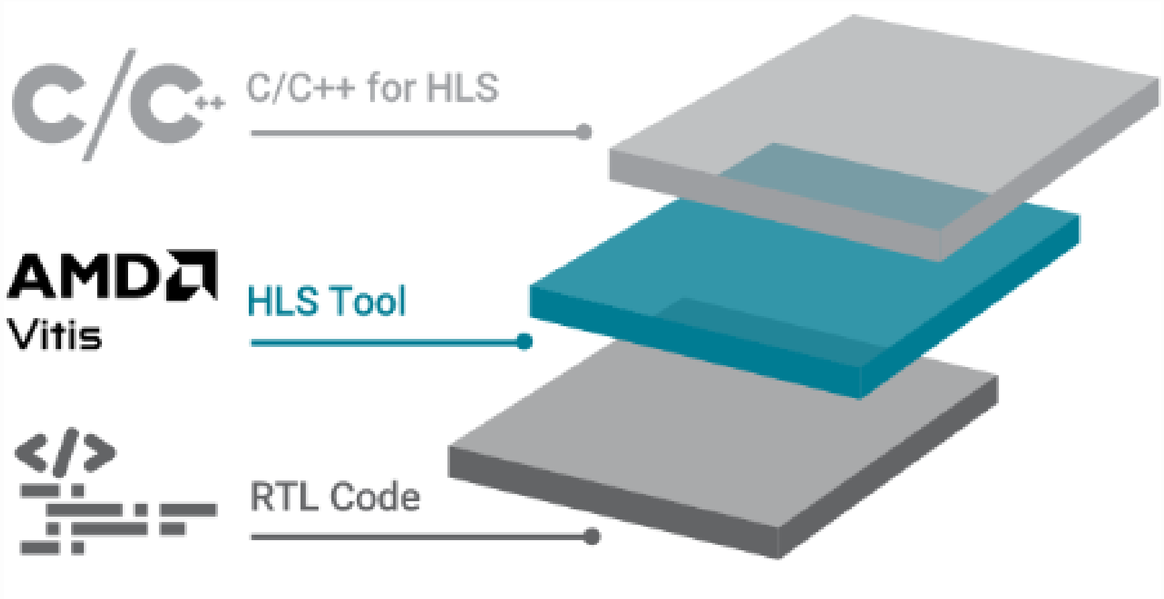
VITIS HLS - PLATAFORMA



VITIS HLS - PLATAFORMA



PRAGMAS



- Diretivas HLS;
- Síntese RTL;
- Impacto na latência e no uso de recursos.

LOOPS

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        printf("%d ", i);
    }

    return 0;
}
```

- Amplamente utilizados em códigos em alto nível;
- Se tornam gargalos em arquiteturas de hardware;
- Foco para otimizações.

LOOPS ANINHADOS

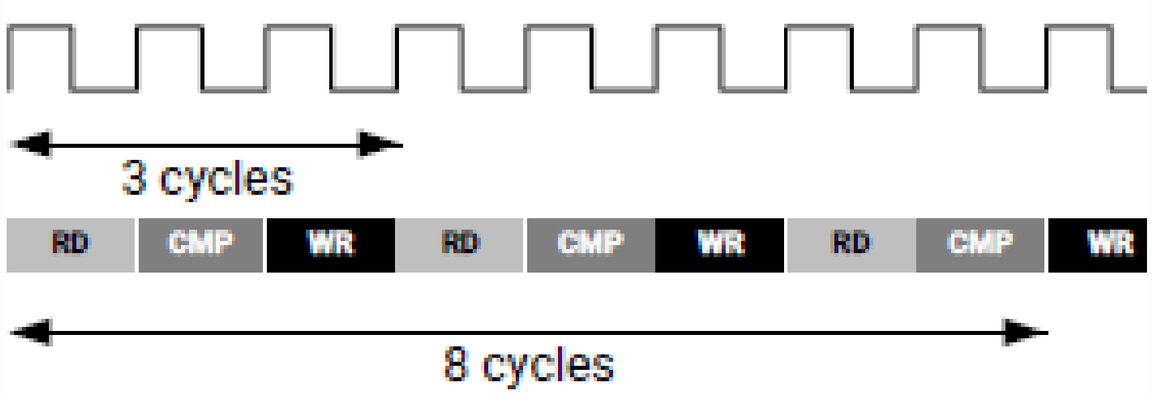
```
#include "loop_pipeline.h"

dout_t loop_pipeline(din_t A[N]) {
    int i,j;
    static dout_t acc;

    LOOP_I:for(i=0; i < 20; i++){
        LOOP_J: for(j=0; j < 20; j++){
            acc += A[j] * i;
        }
    }
    return acc;
}
```

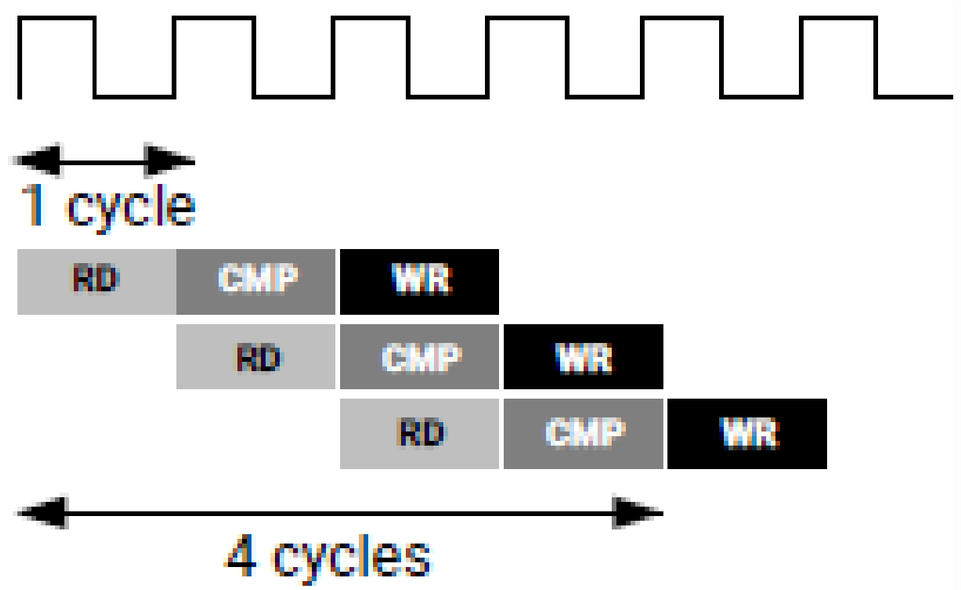
- Perfeitos: apenas o loop mais interno possui lógica e os parâmetros são constantes;
- Semi-perfeitos: apenas o loop mais interno possui lógica, mas os parâmetros do loop mais externo são variáveis;
- Imperfeitos: há lógica fora do loop mais interno ou os parâmetros são variáveis.

PIPELINE



- default: II = 1;
- loop mais interno --> loop mais externo;
- dependências de memória.

pragma HLS PIPELINE



UNROLL

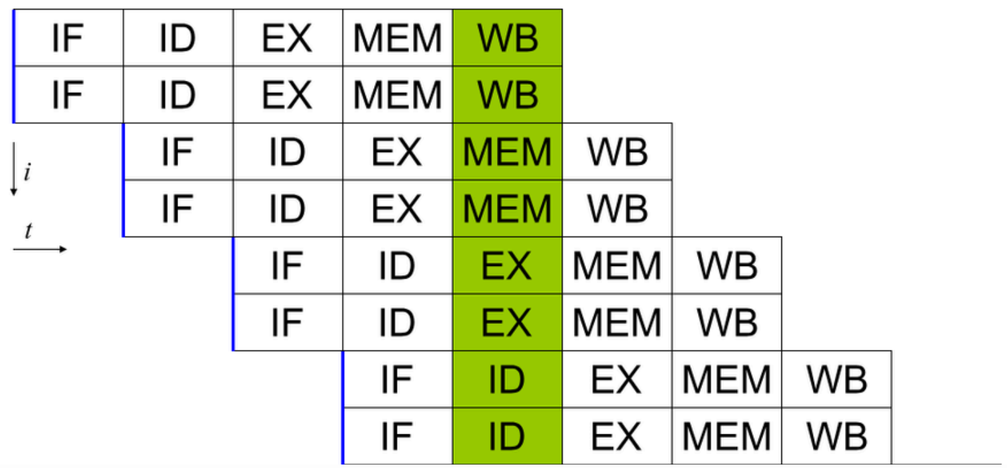
```
#include "test.h"

dout_t test(din_t A[N]) {
  dout_t out_accum=0;
  dsel_t x;

  LOOP_1:for (x=0; x<N; x++) {
      out_accum += A[x];
  }
  return out_accum;
}
```

pragma HLS UNROLL

- réplica do bloco;
- diminuição da latência X aumento de recursos;
- dependências de memória.



LOOP MERGE

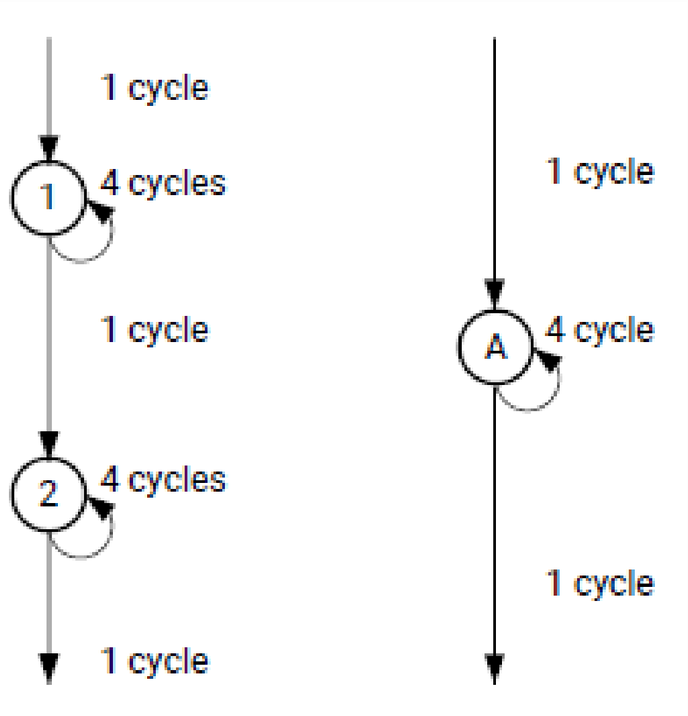
```
void top (a[4],b[4],c[4],d[4]...) {

  ...   Add: for (i=3;i>=0;i--)  {
    if  (d[i])
      a[i] = b[i] + c[i];
  }

  Sub: for (i=3;i>=0;i--)  {
    if (!d[i])
     a[i] = b[i] - c[i];
  }
```

#pragma HLS MERGE

- ciclos de clock desnecessários;
- lógicas otimizadas juntas.



LOOP FLATTEN

```
#include "loop_pipeline.h"

dout_t loop_pipeline(din_t A[N]) {
    int i,j;
    static dout_t acc;

    LOOP_I:for(i=0; i < 20; i++){
        LOOP_J: for(j=0; j < 20; j++){
            acc += A[j] * i;
        }
    }
    return acc;
}
```

- loops perfeitos ou semi-perfeitos;
- pipeline o loop mais interno;
- único loop com 400 repetições e apenas um multiplicador utilizado.

PERFORMANCE

```
  for (int i =0; i < 1000; ++i) {
#pragma HLS performance target_ti=1000
    for (int j = 0; j < 8; ++j) {
      int tmp = b_buf[j].read();
      b[i * 8 + j] = tmp + 2;
    }
  }
```

- Permite especificar uma restrição de alto nível
  - target_ti (define o número de ciclos para cada interação do loop ou função)
  - target_tl (latência alvo - nº de clock para conclusão de todas as interações do loop)
- Ferramenta infere os pragmas de UNROLL, PIPELINE, ARRAY_PARTITION e INLINE necessárias para atingir o resultado
- Importante: PERFORMANCE não garante que seja alcançado mas é uma META

```
#pragma HLS performance target_ti=<value> target_tl=<value> unit=[sec|cycle]
```

ALLOCATION

```
void my_func(data_t angle) {
#pragma HLS allocation operation instances=mul limit=1
...
}
```

- Define um limite máximo de recursos para implementar o kenel.
- Usado para limitar o número máximo de instâncias do RTL, funções, loops ou operações específicas.
- Resultado:
  - Redução dos recursos utilizados
  - Maior compartilhamento de recursos
  - Impacta negativamente no desempenho

```
#pragma HLS allocation <type> instances=<list>
limit=<value>
```

FUNCTION_INSTANTIATE


```
char func_sub(char inval, char incr) {
#pragma HLS INLINE OFF
#pragma HLS FUNCTION_INSTANTIATE variable=incr
 return inval + incr;
}
void func(char inval1, char inval2, char inval3,
 char *outval1, char *outval2, char * outval3)
{
 *outval1 = func_sub(inval1, 1);
 *outval2 = func_sub(inval2, 2);
 *outval3 = func_sub(inval3, 3);
}
```

- Técnica de otimização que matém a hirarquia das funções, realizando otimizações locais direcionadas para uma determinada instância.
- Cria uma implementação RTL para cada instância da função
- Resultado:
  - Simplificação da lógica de controle
  - Pode potencialmente melhorar a latência e o throughput.

```
#pragma HLS FUNCTION_INSTANTIATE variable=<variable>
```

LABORATÓRIO

Esse programa é executado sequencialmente em uma FPGA, produzindo resultados corretos sem nenhum ganho de desempenho. Para obter maior desempenho em uma FPGA, o programa deve ser refatorado para permitir o paralelismo no hardware

```
#include "compute.hpp"

void compute(const data_t in[totalNumWords], data_t out[totalNumWords]) {
    data_t tmp1[totalNumWords], tmp2[totalNumWords];
    A: for (int i = 0; i < totalNumWords; ++i) {
        tmp1[i] = in[i] * 3;
        tmp2[i] = in[i] * 3;
    }
    B: for (int i = 0; i < totalNumWords; ++i) {
        tmp1[i] = tmp1[i] + 25;
    }
    C: for (int i = 0; i < totalNumWords; ++i) {
        tmp2[i] = tmp2[i] * 2;
    }
    D: for (int i = 0; i < totalNumWords; ++i) {
        out[i] = tmp1[i] + tmp2[i] * 2;
    }
}
```

LABORATÓRIO

A função de *compute()* pode ser iniciada antes que todos os dados sejam transferidos para ela.

```
1   #include "compute.hpp"
2
3   void compute(const data_t in[totalNumWords], data_t out[totalNumWords]) {
4       data_t tmp1[totalNumWords], tmp2[totalNumWords];
5       A: for (int i = 0; i < totalNumWords; ++i) {
6           tmp1[i] = in[i] * 3;
7           tmp2[i] = in[i] * 3;
8       }
9       B: for (int i = 0; i < totalNumWords; ++i) {
10          tmp1[i] = tmp1[i] + 25;
11      }
12      C: for (int i = 0; i < totalNumWords; ++i) {
13          tmp2[i] = tmp2[i] * 2;
14      }
15      D: for (int i = 0; i < totalNumWords; ++i) {
16          out[i] = tmp1[i] + tmp2[i] * 2;
17      }
18  }
```

LABORATÓRIO

Várias funções *compute()* podem ser executadas de forma sobreposta, por exemplo, um loop *for* pode iniciar a próxima iteração antes que a iteração anterior tenha sido concluída.

```
#include "compute.hpp"

void compute(const data_t in[totalNumWords], data_t out[totalNumWords]) {
    data_t tmp1[totalNumWords], tmp2[totalNumWords];
    A: for (int i = 0; i < totalNumWords; ++i) {
        tmp1[i] = in[i] * 3;
        tmp2[i] = in[i] * 3;
    }
    B: for (int i = 0; i < totalNumWords; ++i) {
        tmp1[i] = tmp1[i] + 25;
    }
    C: for (int i = 0; i < totalNumWords; ++i) {
        tmp2[i] = tmp2[i] * 2;
    }
    D: for (int i = 0; i < totalNumWords; ++i) {
        out[i] = tmp1[i] + tmp2[i] * 2;
    }
}
```

LABORATÓRIO

As operações em um loop *for* podem ser executadas simultaneamente em várias palavras e não precisam ser executadas por palavra.

```cpp
#include "compute.hpp"

void compute(const data_t in[totalNumWords], data_t out[totalNumWords]) {
    data_t tmp1[totalNumWords], tmp2[totalNumWords];
    A: for (int i = 0; i < totalNumWords; ++i) {
        tmp1[i] = in[i] * 3;
        tmp2[i] = in[i] * 3;
    }
    B: for (int i = 0; i < totalNumWords; ++i) {
        tmp1[i] = tmp1[i] + 25;
    }
    C: for (int i = 0; i < totalNumWords; ++i) {
        tmp2[i] = tmp2[i] * 2;
    }
    D: for (int i = 0; i < totalNumWords; ++i) {
        out[i] = tmp1[i] + tmp2[i] * 2;
    }
}
```

LABORATÓRIO

No exemplo anterior, é a função *compute()* que precisa ser rearquitetada para aceleração baseada em FPGA.

A função *compute()* Loop A multiplica um valor de entrada por 3 e cria dois caminhos separados, B e C. Os loops B e C executam operações e alimentam os dados em D.

Essa é uma representação simples de um caso realista em que você tem várias tarefas a serem executadas uma após a outra e essas tarefas estão conectadas entre si como uma rede, como a mostrada abaixo.

LABORATÓRIO

As principais conclusões para rearquitetar o módulo de hardware são:

- O paralelismo no nível da tarefa é implementado no nível da função. Para implementar o paralelismo em nível de tarefa, os loops são colocados em funções separadas. A função original *compute()* é dividida em várias subfunções. Como regra geral, as funções sequenciais podem ser executadas simultaneamente e os *loops* sequenciais podem ser canalizados.

LABORATÓRIO

As principais conclusões para rearquitetar o módulo de hardware são:

- O paralelismo em nível de instrução é implementado pela leitura de 16 palavras de 32 bits da memória (ou 512 bits de dados). Os cálculos podem ser executados em todas essas palavras em paralelo. A classe *hls::vector* é uma classe de modelo C++ para executar operações de vetor em várias amostras simultaneamente.

LABORATÓRIO

As principais conclusões para rearquitetar o módulo de hardware são:

- A função *compute()* precisa ser rearquitetada em subfunções *load-compute-store*, conforme mostrado no exemplo abaixo. As funções *load* e *store* encapsulam os acessos aos dados e isolam os cálculos realizados pelas várias funções de cálculo.

- Além disso, há diretivas do compilador que começam com *#pragma* e que podem transformar o código sequencial em execução paralela.

LABORATÓRIO

```
[hls]
clock=5
flow_target=vitis
syn.file=diamond.cpp
syn.top=diamond
tb.file=diamond_test.cpp
tb.file=result.golden.dat
syn.dataflow.default_channel=fifo
syn.dataflow.fifo_depth=2
package.output.format=xo
package.output.syn=false
```

BIBLIOGRAFIA

- Vitis HLS users guide:
  - https://docs.amd.com/r/en-US/ug1399-vitis-hls