

# Tutorial 3 - FPGA - IP

Nesse tutorial deseja-se desenvolver um periférico customizado para o processador NIOS, esse periférico será dedicado ao controle dos LEDs da placa, o periférico terá um banco de registradores interno para seu controle, e interface de "I/O mapeado em memória".

## Pré-requisitos

Para seguir esse tutorial, é necessário:

- **Hardware:** DE10-Standard e acessórios
- **Softwares:** Quartus 18.01
  - Modelsim Simulator

Entrega no git:

- **Pasta:** `Tutorial-FPGA-NIOS-IP`

## IP Cores

Intellectual Proprety Core (IP Core) são componentes descritos em HDL que possibilitam ser utilizados em múltiplos projetos de Hardware. O Platform Designer (PD) fornece além da interface visual de conexão um padrão de comunicação entre os componentes, facilitando assim o uso desses IPs.

Além da centenas de projetos espalhados pela internet (github), existe um repositório muito completo de IP cores opensource que concentra grande variedade de projeto:

- [opencores](#)

As empresas também disponibilizando IPs, pagos e gratuitos:

- [Altera IP cores](#)

# Platform Designer

O PD é uma ferramenta integradora de IPs, com ela é muito simples inserirmos e criarmos componentes que serão utilizados para formar um sistema mais completo. Como no caso do tutorial passado onde usamos uma série de componentes para criar nosso projeto. Esses componentes são de certa forma IPs (simples como o PIO e complexo como o NIOS).

A integração dos IPs no PD se dá devido a padronização da comunicação entre esses componentes, que é dada via o barramento.

## Barramentos

A Altera define dois tipos de barramento de dados para o PD: **Avalon** e **AXI**. O barramento Avalon é a principal maneira de conectar um periférico ao NIOS (processador), já o AXI é o padrão de barramento do ARM, que também é utilizado no platform designer.

### Avalon

Documentação completa dos tipos do barramento AVALON :

- [https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl\\_avalon\\_spec.pdf](https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf)

O barramento Avalon define basicamente dois tipos de comunicação : **Memory Mapped (MM)** e **Avalon Streaming Interface (ST)**, conforme descrição a seguir extraído da documentação :

- **Avalon Streaming Interface (Avalon-ST)** — an interface that supports the unidirectional flow of data, including multiplexed streams, packets, and DSP data.
- **Avalon Memory Mapped Interface (Avalon-MM)** — an address-based read/write interface typical of master-slave connections.
- **Avalon Conduit Interface** — an interface type that accommodates individual signals or groups of signals that do not fit into any of the other Avalon types.

You can connect conduit interfaces inside a Platform Designer system. Or, you can export them to make connections to other modules in the design or to FPGA pins.

- Avalon Tri-State Conduit Interface (an interface to support connections to off-chip peripherals. Multiple peripherals can share pins through signal multiplexing, reducing the pin count of the FPGA and the number of traces on the PCB.
- Avalon Interrupt Interface – an interface that allows components to signal events to other components.
- Avalon Clock Interface – an interface that drives or receives clocks.
- Avalon Reset Interface – an interface that provides reset connectivity.

## Projeto

### Note

Vamos melhorar o projeto passado, faça uma cópia da pasta do projeto : `Tutorial-FPGA-NIOS` e renomeei para: `Tutorial-FPGA-NIOS-IP` . Iremos agora trabalhar nessa nova pasta.

## Criando um periférico

Vamos criar um novo componente que será capaz de controlar os LEDs com maior autonomia.

Roteiro a ser seguido:

1. Especificação
2. Gerar HDL que representa o periférico com interface Avalon
3. Criar o componente no Platform Designer
  - Associar arquivos ao componente
  - Definições gerais
  - Associar as portas do componente com os sinais do barramento
4. Usar componente no projeto

5. Criar driver (.c e .h)
6. Simular
7. Implementar/ Testar
8. Rever especificação (1.)

Primeiramente precisamos definir o papel principal desse periférico e seu fluxo de dados. Com isso será possível definir se o periférico é do tipo: **Master** ou **Slave** e se sua interface é do tipo **Memory Mapped** ou **Streaming**.

Um periférico pode possuir mais de uma interface, por exemplo: Um periférico que irá processar um áudio em tempo real pode ter até três interfaces: O mesmo irá receber o áudio via a interface **streaming** e retornar o dado por outra interface de **streaming**, porém será necessário uma terceira interface para controle desse periférico, muito provavelmente do tipo **Memory Mapped**.

É possível transmitir pacotes de comando pela interface streaming, mas isso torna o projeto mais complexo.

O nosso simples periférico irá simplesmente receber configurações para acionar o LED, sem nenhum fluxo contínuo ou intenso de dados, sendo a interface mais apropriada a do **periférico mapeado em memória**. Além disso, nosso periférico exclusivo para controle do LED é claramente um **slave** do sistema, já que ele deve ser controlado por outra parte do sistema (no nosso caso o uC) para agir conforme necessário.

### **Avalon Slave Memory Mapped**

Para nosso periférico se comunicar com o processador precisamos implementar o padrão de comunicação utilizado pelo NIOS. Podemos optar por implementar o padrão completo ou apenas uma parte de sua especificação. Por exemplo, se nosso periférico não faz uso do `waitrequest` ou `byteenable` podemos optar por não implementar esses sinais.

A seguir um exemplo dos sinais de um periférico mapeado em memória que possui como interface com o `Avalon-MM-Slave`.

```

entity peripheral_MM is
    port (
        -- Gloabals
        clk          : in  std_logic          :=
'0';
        reset        : in  std_logic          :=
'0';

        -- Avalon Memmory Mapped Slave
        avs_address   : in  std_logic_vector(3 downto 0) :=
(others => '0');
        avs_read      : in  std_logic          := '0';
        avs_readdata  : out std_logic_vector(31 downto 0) :=
(others => '0');
        avs_write     : in  std_logic          := '0';
        avs_writedata : in  std_logic_vector(31 downto 0) :=
(others => '0')
    );
end entity peripheral_MM;

```

Note que a primeira parte do componente define um sinal de clock ( `clk` ) e um sinal de reset ( `reset` ), lembre que projeto digitais em FPGA devem ser na maioria das vezes síncronos. A segunda parte é a definição dos sinais que irão ser conectados no barramento para acesso de outros periféricos.

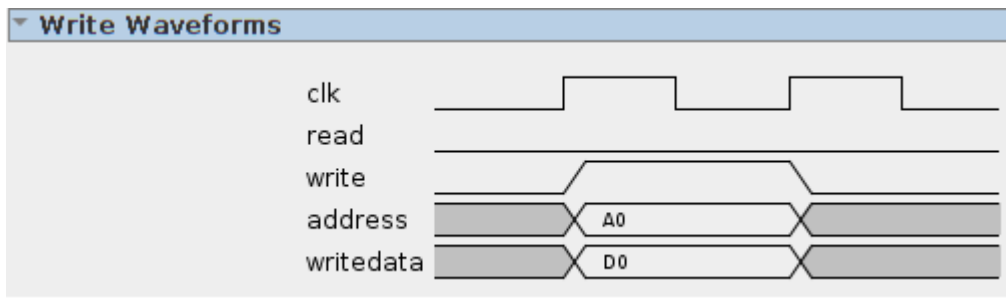
Lembrem que estamos criando um componente mapeado em memória, logo o mesmo deve ter comportamento e interface similar ao de uma memória.

- `avs_address` : Endereço de acesso ao componente, no caso, 4 bits.
- `avs_read` : Indica que é um acesso de leitura
- `avs_readdata` : Dado que será retornado ao Master dado um acesso de leitura.
- `avs_write` : Indica que é um acesso de escrita
- `avs_writedata` : Dado que é transmitido ao componente dado um acesso de escrita.

O tamanho da palavra do `avs_readdata` e do `avs_writedata` é definido pelo componente e não é fixado em 32 bits como no exemplo, pode assumir outros valores.

Uma escrita ao periférico é dada da seguinte forma:

1. Master endereça periférico
2. Endereço absoluto é traduzido em relativo
  - O endereço que o master escreve no periférico é composto por: **addr + offset** porém o slave só possui acesso ao **offset**.
3. Periférico recebe: `avs_address`, `avs_write = '1'` e `avs_writedata`.



Uma leitura ao periférico é dada da seguinte forma:

1. Master endereça periférico
2. Endereço absoluto é traduzido em relativo
3. Periférico recebe: `avs_address` e `avs_read = '1'`
4. Periférico atualiza: `avs_readdata`

### ? Perguntas

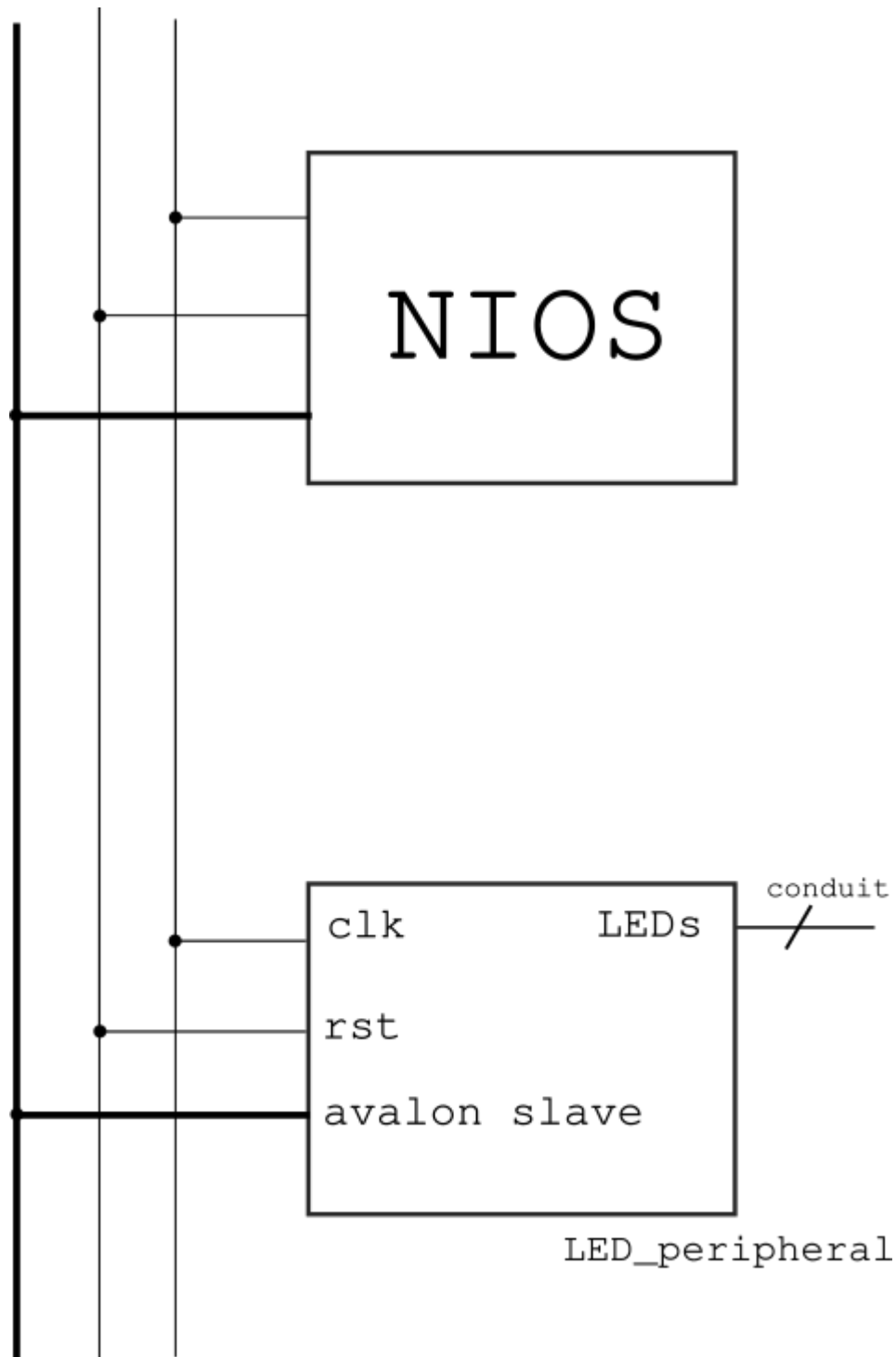
O barramento `AVALON` define outros sinais, responda a seguir sobre alguns desses sinais:

- `waitrequest`
- Qual o papel do `waitrequest`?
- Quem aciona o `waitrequest` (Slave ou Master)?
- `byteenable`
- Qual o papel do `byteenable`?
- Quem aciona o `byteenable` (Slave ou Master)?

## Especificação

Nosso periférico será no começo bem simples, apenas para entendermos todo o processo de desenvolvimento de um periférico e o seu uso. O periférico que iremos desenvolver será um substituto ao periférico PIO fornecido pela Altera, utilizado no projeto do pisca LED com o NIOS.

Nosso periférico será mapeado em memória e possuirá um conduit (saída) onde será realizada o acionamento dos LEDs:



O acesso ao nosso periférico será por uma palavra de 32 bits (para manter um padrão com o NIOS) e terá dois registradores `REG_CONFIG` e `REG_DATA` :

- `REG_CONFIG` : Registrador que controla o periférico, no nosso caso, irá ter somente um bit de: Enable / Disable ( bit0 → Enable/Disable )
- `REG_DATA` : Registrador que possui o valor de cada LED ( bit0 → LED0 ; bit1 → LED1 ....).



## Gerar HDL que representa o periférico com interface Avalon

Partindo da entidade fornecida (peripheral\_MM), podemos criar um componente que implementa parcialmente a especificação anterior, nessa implementação não temos os dois registradores ( REG\_CONFIG e REG\_DATA ), temos apenas a funcionalidade do REG\_DATA . Note que a implementação faz uso de um generic para definir a quantidade de LEDs que esse periférico controla. Esse generic poderá ser configurado pela interface gráfica do Plataform Designer, tornando um componente customizado.

### Info

Crie um arquivo chamado: `peripheral_LED.vhd` e salve na pasta do projeto : `Tutorial-FPGA-NIOS-IP/IP/`

Será necessário criar a pasta IP

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use work.all;

entity peripheral_LED is
    generic (
        LEN : natural := 4
    );
    port (
        -- Gloabals
        clk          : in  std_logic          :=
'0';
        reset        : in  std_logic          :=
'0';

        -- I/Os
        LEDs         : out std_logic_vector(LEN - 1 downto
0) := (others => '0');

        -- Avalion Memmory Mapped Slave
        avs_address  : in  std_logic_vector(3 downto 0) :=
(others => '0');
        avs_read     : in  std_logic          := '0';
```

```

        avs_readdata      : out std_logic_vector(31 downto 0) :=
(others => '0');
        avs_write         : in  std_logic                      := '0';
        avs_writedata     : in  std_logic_vector(31 downto 0) :=
(others => '0')
    );
end entity peripheral_LED;

architecture rtl of peripheral_LED is
begin

    process(clk)
    begin
        if (reset = '1') then
            LEDs <= (others => '0');
        elsif(rising_edge(clk)) then
            if(avs_address = "0001") then -- REG_DATA
                if(avs_write = '1') then
                    LEDs <= avs_writedata(LEN - 1 downto 0);
                end if;
            end if;
        end if;
    end process;

end rtl;

```



#### Limitações dessa implementação

- Não possui um registrador de configuração: REG\_CONFIG
- Não é possível ler: REG\_DATA via barramento Avalon
- impede a aplicação de máscaras!

Poderíamos já nessa etapa testar o componente, criando um `testbench` para excitar o módulo e verificar seu comportamento. Grande parte do desenvolvimento de um projeto de hardware é gasto nos testes, que podem ser tão complexos quanto o próprio módulo. Vamos pular essa etapa aqui, iremos simular em um nível mais alto.

## Configurando path

Agora iremos adicionar o nosso periférico no **Platform Designer**, esse novo componente que será criado será incorporado na ferramenta, para isso:

Precisamos indicar para o PD o local que ele deve buscar para encontrar por códigos fontes que não fazem parte do catálogo padrão, para isso:

1. Tools → Options → IP Search Path
2. Adicione a pasta `IP` recém criada.

E agora remova o componente PIO:

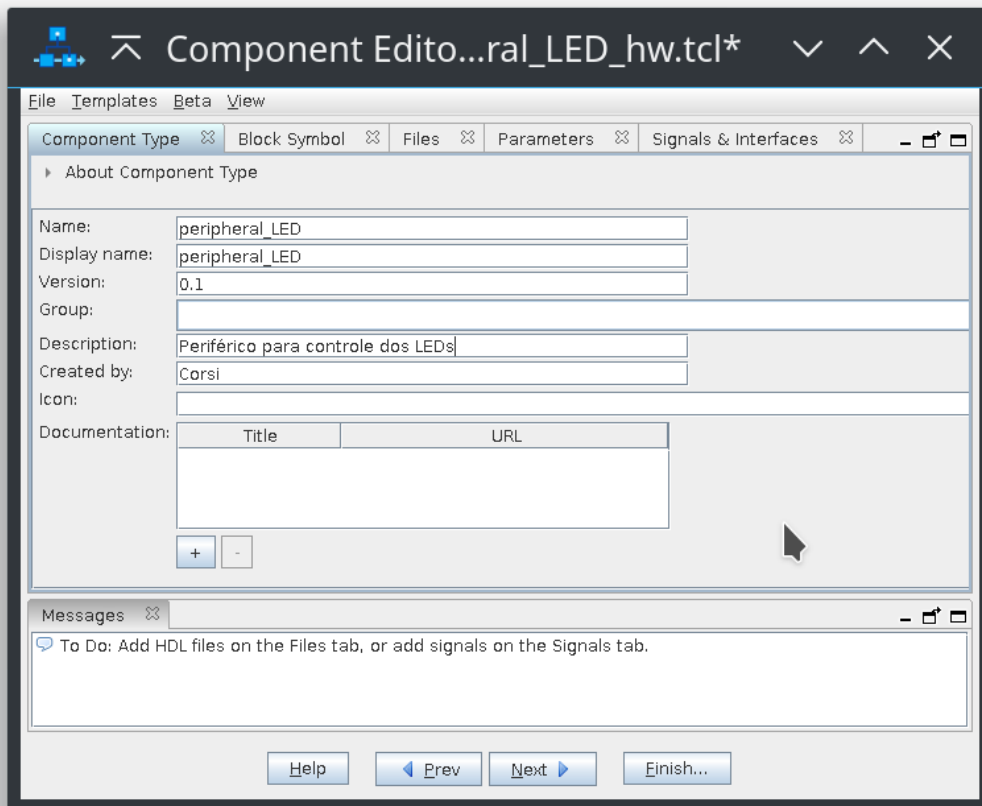
1. Remova o PIO que controlava os LEDs (agora iremos fazer o controle pelo nosso componente)

## Criando componente

Só adicionar o arquivo HDL ( `.vhd` ou `.v` verilog) não é suficiente para o PD reconhecer o componente, precisamos criar um segundo arquivo ( `*_hw.tcl` ) que é lido pelo PD, esse arquivo possuirá todas as configurações e descrições do novo componente. Para isso :

- File → New Component OK

E uma interface gráfica de configuração do componente será exibida. A primeira parte é referente a descrição do próprio componente. De o nome desse componente de : `peripheral_LED` e preencha sua descrição.



Já na aba **Files** temos as informações de quais arquivos pertencem ao componente.

## Files

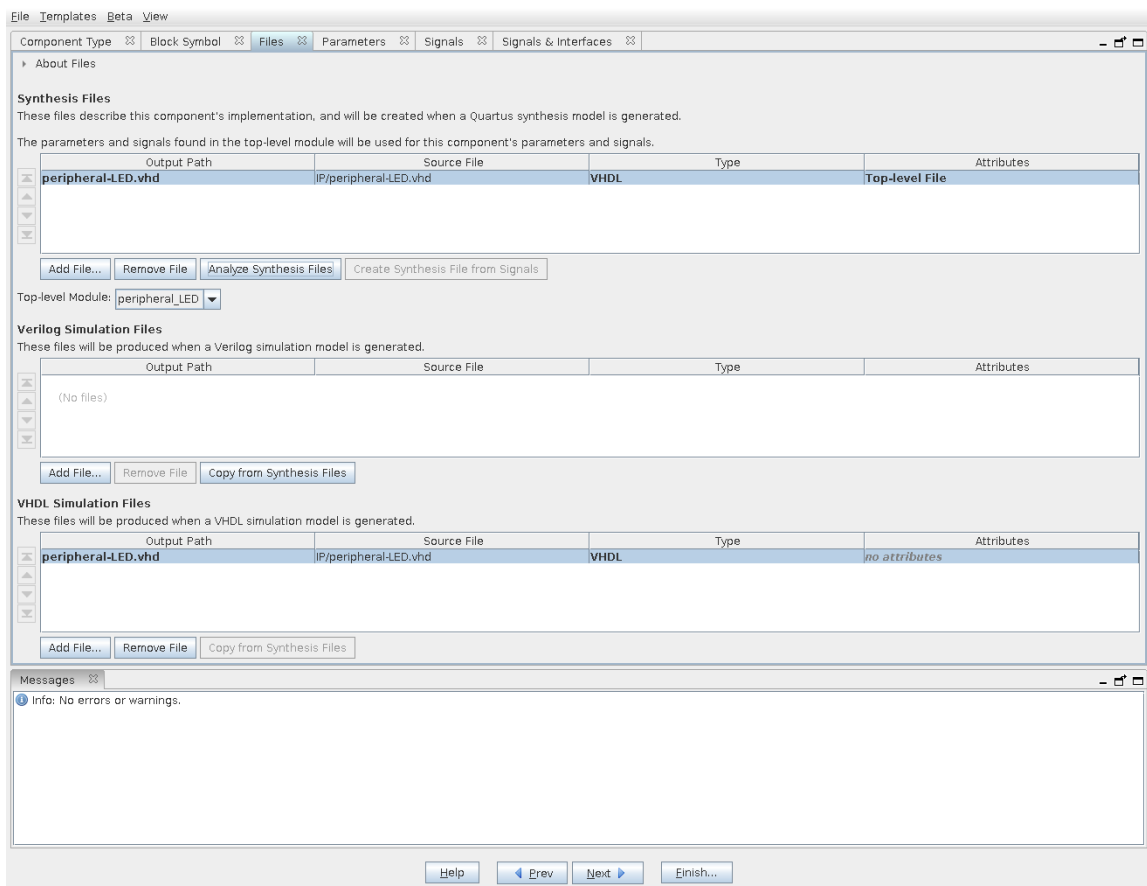
Na aba Files adicione o arquivo `peripheral-LED.vhd` :

1. **Files** → **Syntesis Files** → **add file** → **peripheral-LED.vhd**
2. Clique em **Analyze Synthesis Files** : isso fará com que a ferramenta faça uma breve análise dos arquivos HDL e detecte as interfaces do componente.

Note o atributo do arquivo: **Top-level File** , isso indica que o `peripheral-LED.vhd` é o arquivo principal desse componente, se tivéssemos um desenvolvimento hierárquico do componente, nessa etapa adicionaríamos vários arquivos e deveríamos configurar qual deles é o toplevel.

- Na secção **VHDL Simulation Files** → **Copy from Synthesis Files** **OK**

Note que se não adicionarmos esse arquivo nessa secção, na hora de simular o projeto o componente estaria vazio. Porquê o padrão não é o de automaticamente copiar os arquivos da síntese para a simulação? Pois nem sempre conseguimos simular o que será sintetizado. Pense no caso desse componente ser um controlador de memória, se formos simular não teremos a memória física para o controlador acessar e a simulação não funcionará. Uma solução seria de ter dois componentes, um para simulação (que imita a memória) e outro para síntese.



## Signals & Interfaces

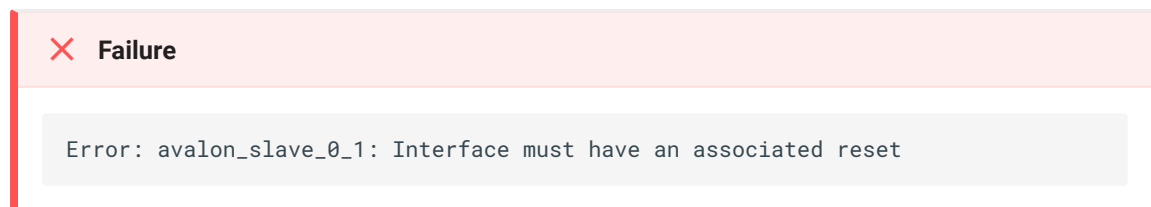
Nessa secção iremos configurar as interfaces do nosso componente, e como o PD irá interpretá-las quando formos conectar ao resto do sistema. Note que algumas interfaces já foram detectadas pelo PD, porém temos um erro que será corrigido.

Nas interfaces padrões note que o `Component Editor` já detectou uma interface:

- **avalon\_slave\_0**
- **clock**
- **reset**

Isso aconteceu pelos nomes da entidade do `peripheral_led`.

Vamos primeiramente editar o `avalon_slave_0`. Clique na interface e note que a ferramenta indica um erro :

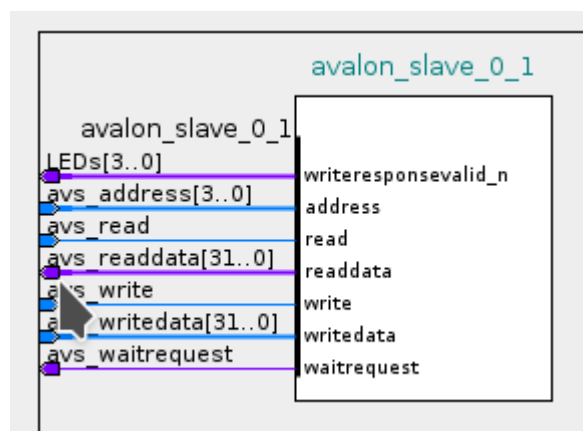


Vamos associar ter que associar um sinal der reset a interface (parte sequência do IP), para isso :

- `avalon_slave_0` ➡ Associated Reset ➡ `reset`

Podemos notar ainda pelo diagrama (e pela mensagem de erro) que a ferramenta interpretou de forma errada o nosso sinal `LEDs`, pertencente a entidade do componente:

```
-- I/Os
LEDs : out std_logic_vector(LEN - 1 downto 0) := (others => '0');
```

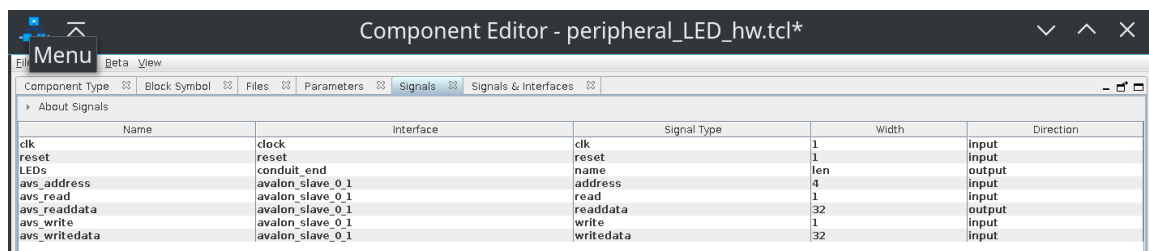


Note pelo diagrama de blocos que o PD atribui essa saída como sendo parte do barramento Avalon: **writerequestvalid\_n**, o que não é verdade. Para corrigir isso, precisamos de uma nova aba que não é padrão de exibição, no **component builder** clique em:

- **Component builder** ➔ **View** ➔ **Signals** **OK**

Essa nova aba permite verificarmos (e associarmos) as entradas e saídas da entidade (toplevel) com sinais e tipos de sinais definido pelo PD.

Iremos indicar agora para a ferramenta que o sinal **LEDs** deve ser interpretado como um **conduite**, edite os sinais como na figura a seguir :

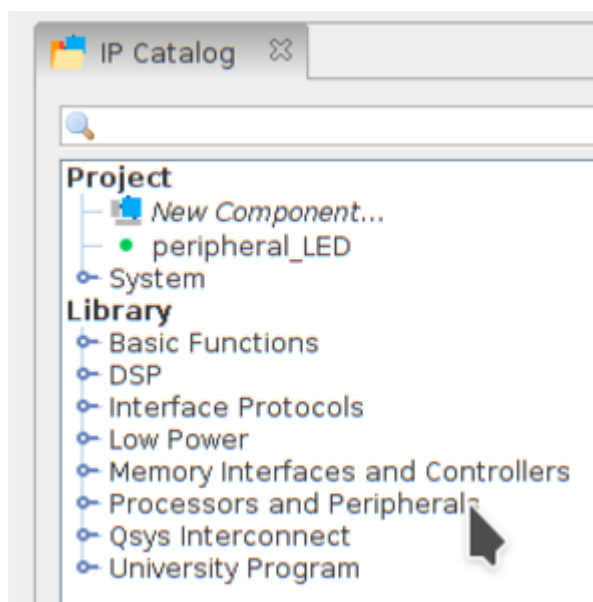


Name	Interface	Signal Type	Width	Direction
clk	clock	clk	1	input
reset	reset	reset	1	input
LEDs	conduit_end	name	len	output
avs_address	avalon_slave_0_1	address	4	input
avs_read	avalon_slave_0_1	read	1	input
avs_readdata	avalon_slave_0_1	readdata	32	output
avs_write	avalon_slave_0_1	write	1	input
avs_writedata	avalon_slave_0_1	writedata	32	input

## Finalizando

Verifique os sinais e o diagrama de bloco antes de continuar e clique em **Finish**.

Quando o componente for gerado, ele automaticamente irá aparecer no catálogo de componentes que podem ser inseridos no SoC :



Porém o arquivo de configuração desse componente (.tcl) foi salvo na pasta raiz do projeto do Quartus :

- tutorial-SoftProcessor-IP/peripheral\_LED\_hw.tcl

Esse arquivo .tcl descreve todas as configurações realizadas anteriormente no componente. O mais natural é que esse arquivo esteja na mesma localidade (pasta IP) que os códigos HDL. Mova esse arquivo para:

- tutorial-SoftProcessor-IP/IP/peripheral\_LED\_hw.tcl

Agora precisamos editar o arquivo .tcl para atualizarmos o local do arquivo peripheral-LED.vhd , procure pela seção **files set**:

- Antes

```
add_fileset_file peripheral-LED.vhd VHDL PATH IP/peripheral-LED.vhd TOP_LEVEL_FILE
...
add_fileset_file peripheral-LED.vhd VHDL PATH IP/peripheral-LED.vhd
```

E edite para:

```
add_fileset_file peripheral-LED.vhd VHDL PATH peripheral-LED.vhd TOP_LEVEL_FILE
...
add_fileset_file peripheral-LED.vhd VHDL PATH peripheral-LED.vhd
```

## Utilizando o componente no PD

Agora adicione o componente no projeto e faça as conexões corretas (como se fosse outro componente), exporte o sinal dos LEDs, o resultado final deve ser algo como :





Gere o componente: Clique em [Generate HDL](#)  [Generate](#).

The screenshot shows the 'Generation' dialog box in Quartus II. The dialog is titled 'Generation' and has a standard Windows window appearance. It contains three main sections: 'Synthesis', 'Simulation', and 'Output Directory'.  
1. **Synthesis**: This section explains that synthesis files are used to compile the system in a Quartus project. It includes a dropdown menu for 'Create HDL design files for synthesis' set to 'Verilog'. There are two unchecked checkboxes: 'Create timing and resource estimates for third-party EDA synthesis tools.' and 'Create block symbol file (.bsf)'.  
2. **Simulation**: This section explains that the simulation model contains generated HDL files for the simulator. It states that simulation scripts will be generated in a vendor-specific sub-directory. It provides guidance on using 'ip-setup-simulation' and 'ip-make-simscript' utilities. It includes a dropdown menu for 'Create simulation model' set to 'VHDL'.  
3. **Output Directory**: This section has a 'Path' label followed by a text box containing the path 'cadosAvancados/EmbarcadosAvancados/EmbarcadosAvancados/03-SoC-IP/niosHello' and a browse button ('...').  
At the bottom right, there are two buttons: 'Generate' and 'Cancel'. The 'Generate' button is highlighted with a blue border.

## Utilizando o componente no topLevel.vhd

Precisamos agora modificar o componente inserido no topLevel, para isso no PD gere novamente o template de utilização :

- No Platform Designer: Generate  Show Instatiation Template   
VHDL

No meu caso o resultado foi:

```
component niosHello is
  port (
    buts_export    : in  std_logic_vector(2 downto 0) :=
(others => 'X'); -- export
    clk_clk        : in  std_logic                    :=
'X';             -- clk
    reset_reset_n  : in  std_logic                    :=
'X';             -- reset_n
    leds_name      : out std_logic_vector(3 downto
0)              -- name
  );
end component niosHello;

u0 : component niosHello
  port map (
    buts_export    => CONNECTED_TO_buts_export, --
buts.export
    clk_clk        => CONNECTED_TO_clk_clk,      --
clk.clk
    reset_reset_n  => CONNECTED_TO_reset_reset_n, --
reset.reset_n
    leds_name      => CONNECTED_TO_leds_name     --
leds.name
  );
```

Devemos inserir agora esse componente com a nova interface (**leds\_name**) no topLevel.vhd .

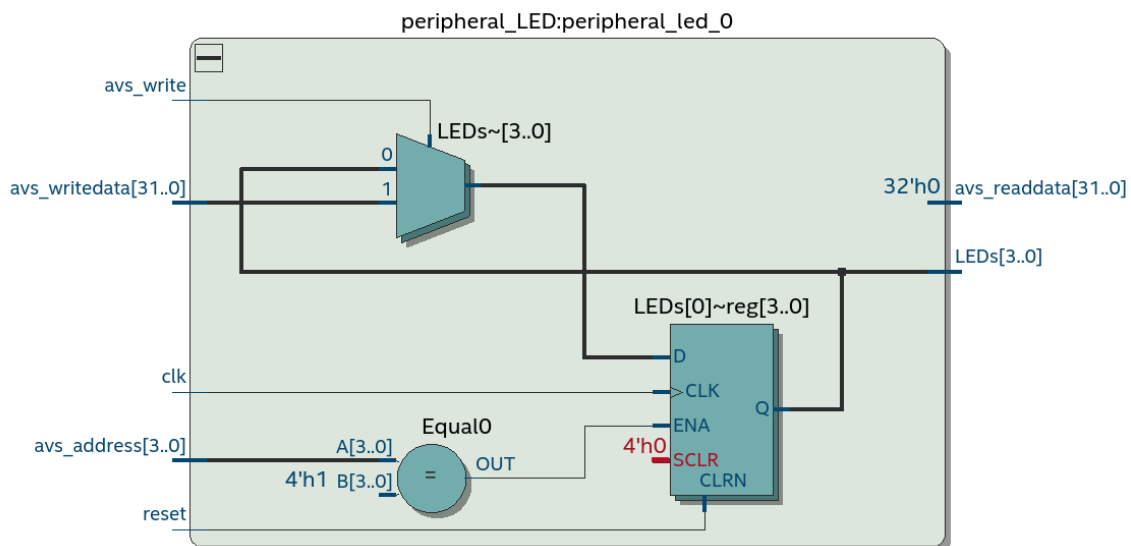
Você deve fazer essa etapa com cuidado. Esses nomes podem alterar entre versões da ferramenta.

Editando o topLevel.vhd :

## ✓ Recompile

Salve, compile o projeto e programe a FPGA

Podemos analisar agora o RTL do projeto e mais especificamente o do componente criado:



Verificamos que a ferramenta inferiu um registrador de 4 bits para armazenar o valor dos LEDs, um Mux para indicar se os registradores serão ou não atualizados com um novo valor e um comparador para verificar se o endereço é equivalente a `0x01`.

## Firmware

Devemos agora escrever um firmware que será executado no NIOS e que acesse e controle nosso periférico. Para isso será necessário criarmos um novo BSP para o projeto. Abra o **NIOS II Software Build ...** e refaça a etapa do tutorial anterior com o novo SoC e adicione o código a seguir:

```
#include <stdio.h>
#include "system.h"
#include <alt_types.h>
#include <io.h> /* Leiutura e escrita no Avalon */

// #define SIM
```

```
// LED Peripheral
#define REG_DATA_OFFSET 1

int main(void){
    unsigned int led = 0;
    unsigned int *p_led = (unsigned int *) PERIPHERAL_LED_0_BASE;

#ifdef SIM
    printf("Embarcados++ \n");
#endif

    while(1){
        if (led < 4){
            *(p_led+REG_DATA_OFFSET) = (0x1 << led++);
#ifdef SIM
            usleep(500000); // remover durante a simulação
#endif
        }
        else{
            led = 0;
        }
    };

    return 0;
}
```

O firmware utiliza o `peripheral-LED` para controlar os LEDs da placa, note que o acesso dessa vez é feito pelo ponteiro `p_led` e não mais pela função da Altera `IOWR_32DIRECT` (deveria funcionar também).

## Executando

Execute o firmware no kit de desenvolvimento e verifique se sua funcionalidade está correta. Lembre que o HW já deve estar programado (via quartus), caso contrário não funcionará.

## Simulando

Uma das grandes vantagens de trabalharmos com SoftProcessor é que temos acesso ao seu código fonte (criptografado ou não) e isso possibilita que possamos simular todo o sistema, verificando suas funcionalidades internas, comunicação da CPU com os periféricos, interface do firmware com o resto do

sistema. Vamos nessa etapa simular a interface do NIOS com o nosso periférico e verificar se está tudo certo.

Note que no código anterior, o `printf` foi comentando, assim como o `delay` de 50000 us, que no lugar foi inserido um de 1us. Isso foi feito para acelerar a simulação e verificarmos mais rapidamente o acesso do NIOS ao periférico, que acontece na linha :

```
*(p_led+REG_DATA_OFFSET) = (0x1 << led++);
```

Nesse momento, o NIOS envia um comando ao barramento no endereço **PERIPHERAL\_LED\_0\_BASE + REG\_DATA\_OFFSET**, o comando carrega a mensagem : **0x01 << led**, gravando no registrador `REG_DATA` qual LED deve ser acionado.

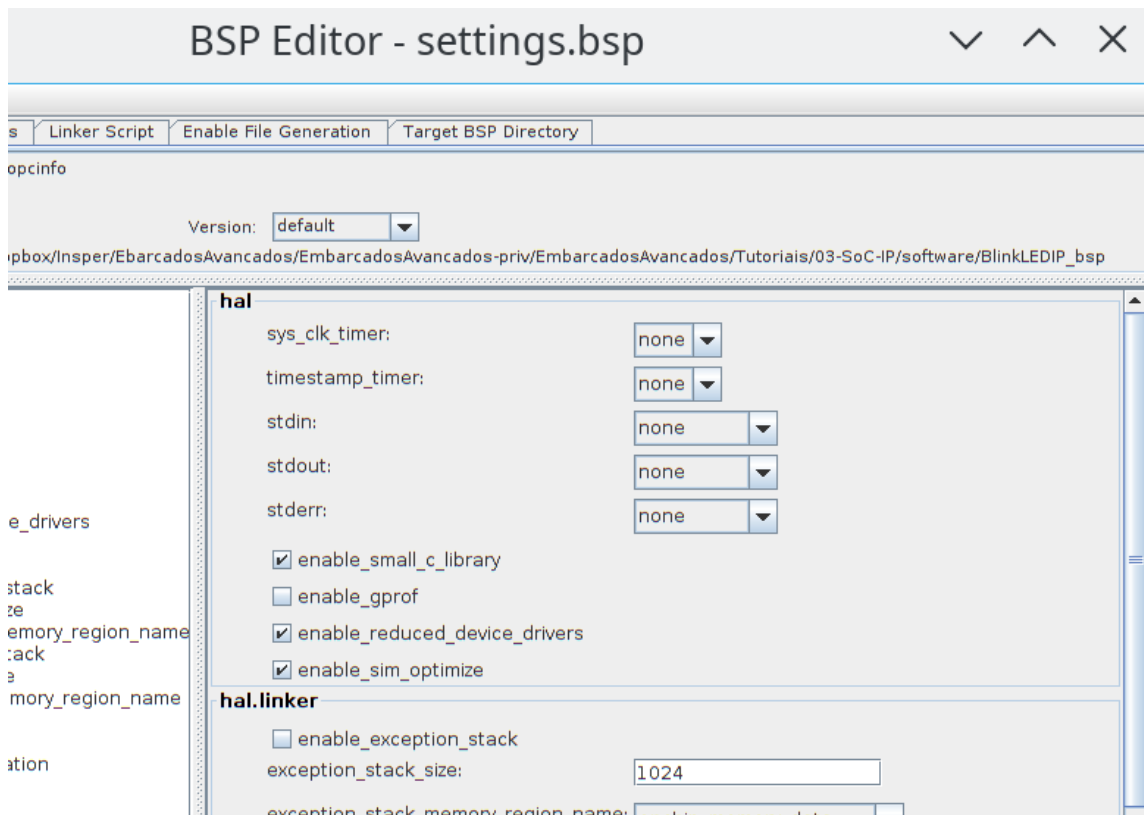
## Configurando o bsp

Para obtermos um resultado mais rápido é possível ativarmos uma opção no bsp chamada de: **enable\_sim\_optimize**. Quando ativada, o binário compilado só poderá ser usado para simulação, **não pode ser embarcado no HW!**. Com essa opção temos um ganho significativo no tempo de execução do modelo no modelsim.

Além de configurarmos a otimização durante a simulação, iremos desativar o **stdin, stdout, stderr** para a simulação ficar ainda mais rápida, caso contrário teremos que esperar por muito tempo até verificarmos o resultado do código. Note que a simulação abrange todo o HW desde o processador até o barramento e periféricos.

### Note

Para simularmos 1 ms de execução do HW será necessário muito mais que 1 ms de esforço computacional! O tempo pode chegar a unidades de hora!!



## ModelSim

No **Eclipse**, após ter compilado o projeto:

- Run ➡ Run configuration ➡ Nios II ModelSim

O simulador a ser utilizado é o modelsim da Mentor, o mais completo do mercado e fornecido com algumas customizações pela Altera. No modelsim, iremos adicionar os sinais que desejamos visualizar, para isso, siga o que indica a figura a seguir:



## Entrega 3

Siga para a terceira entrega:

- [Entega 3](#)