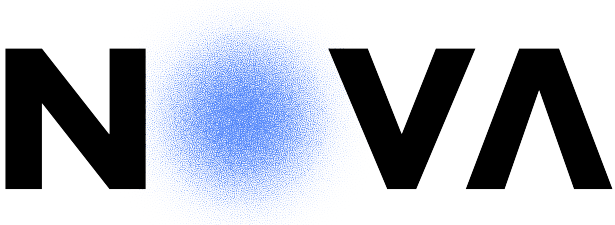


NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Departamento de Engenharia Eletrotécnica e de Computadores

Mestrado em Engenharia Eletrotécnica e de Computadores

Laboratório 1

Co-design e Sistemas Reconfiguráveis

Alunos:
João Pedro Antunes - **70380**
Júlio Lopes- **70512**
Marcos Romão - **71348**

Docentes:
Aniko Costa
Filipe Moutinho

1º Semestre 2025/2026

Conteúdo

1 Introdução

1.1 Contextualização

Este trabalho insere-se no âmbito da unidade curricular de Co-design e Sistemas Reconfiguráveis, focando-se no desenvolvimento e implementação de controladores para sistemas de automação industrial. O caso de estudo considerado é um sistema de produção composto por 3 células em cascata, onde cada célula integra um braço de robô e um tapete rolante.

A complexidade destes sistemas justifica a utilização de metodologias formais de modelação e verificação, permitindo validar o comportamento do controlador antes da sua implementação física. Adicionalmente, a possibilidade de implementar o controlador de forma centralizada ou distribuída oferece diferentes trade-offs entre complexidade de implementação, desempenho e escalabilidade do sistema.

1.2 Objetivos

O presente trabalho tem como objetivo principal o desenvolvimento completo de um controlador para um sistema de automação com três células em cascata (N=3), desde a fase de modelação até à implementação física em hardware reconfigurável.

Os objetivos específicos incluem:

- Modelar o comportamento do sistema utilizando redes de Petri Input-Output Place-Transition (IOPT), explorando as capacidades de composição modular através de operações de adição e fusão de redes;
- Validar e analisar o modelo através de simulação e análise do espaço de estados, verificando propriedades comportamentais críticas do sistema;
- Implementar o controlador em plataformas de hardware distintas (FPGA e Arduino), avaliando a abordagem centralizada de controlo;
- Desenvolver uma arquitetura de controlo distribuído através da decomposição do modelo global, explorando paradigmas de execução síncrona e GALS (Globally Asynchronous Locally Synchronous);
- Analisar o impacto de comunicações não-instantâneas entre controladores distribuídos, introduzindo atrasos pseudo-aleatórios;
- Estender o modelo para suportar buffers de capacidade finita superior a uma unidade, aumentando a flexibilidade do sistema;
- Comparar as diferentes abordagens de implementação, avaliando vantagens, desvantagens e adequação a diferentes contextos aplicacionais.

2 Análise Teórica

2.1 Redes de Petri

As redes de Petri constituem uma ferramenta formal de modelação adequada para sistemas concorrentes e distribuídos. Uma rede de Petri é definida por uma quádrupla $PN = (P, T, F, M_0)$, onde P representa lugares, T transições, F arcos direcionados, e M_0 a marcação inicial. A dinâmica baseia-se no disparo de transições, que removem tokens dos lugares de entrada e adicionam aos de saída.

Redes IOPT (Input-Output Place-Transition) estendem as redes clássicas com capacidade de interagir com o ambiente através de:

- Eventos de entrada associados a sinais externos (sensores);
- Sinais de saída para controlo de atuadores;
- Guardas e prioridades para resolução de conflitos;
- Semântica temporal para modelar delays.

Redes de Petri Coloridas permitem representar de forma compacta sistemas com estrutura repetitiva. Os tokens possuem "cores"(tipos de dados), permitindo que um único modelo represente múltiplas instâncias de subsistemas idênticos. No sistema estudado, tokens $\langle 1 \rangle$, $\langle 2 \rangle$ e $\langle 3 \rangle$ identificam cada célula.

Composição Modular: A construção de sistemas complexos utiliza operações de:

- *Merge Nets*: união de múltiplas redes mantendo nós distintos;
- *Fusion Sets*: identificação de nós a fundir criando sincronização;
- *Net Addition*: combinação de merge e fusão num modelo integrado.

2.2 Execução síncrona vs. GALS

Em contexto síncrono, todos os componentes operam com um relógio global comum. Oferece simplicidade de design e determinismo, mas apresenta limitações de escalabilidade, consumo energético elevado (relógio sempre ativo) e dificuldades na distribuição do sinal de relógio (clock skew) em sistemas grandes.

Já num paradigma GALS, este divide o sistema em domínios síncronos locais que comunicam assincronamente. Cada domínio opera com relógio independente, comunicando através de protocolos de handshaking ou FIFOs assíncronas.

Vantagens do GALS incluem escalabilidade, modularidade, eficiência energética e tolerância a falhas. As desvantagens são a maior complexidade de interface, latência variável na comunicação e necessidade de tratamento de metastabilidade nas fronteiras entre domínios.

2.3 Implementação em FPGA e Arduino

Utilizar-se-á uma **FPGA (Field-Programmable Gate Array)**. A descrição do sistema é feita em VHDL, uma linguagem de descrição de hardware que permite especificar comportamento concorrente. Recorrer-se-á também a um microcontrolador **Arduino**.

A framework IOPT-Tools permite geração automática de código VHDL para FPGA e código C para Arduino, facilitando a comparação entre as duas abordagens de implementação.

Critério	**FPGA**	**Arduino**
Paralelismo	Hardware real	Software (sequencial)
Tempo de resposta	ns	ms

Tabela 1: Comparação entre plataformas FPGA e Arduino

3 Redes de Petri

3.1 IOPT-Tools

3.1.1 Rede de Petri — Tapete Singular

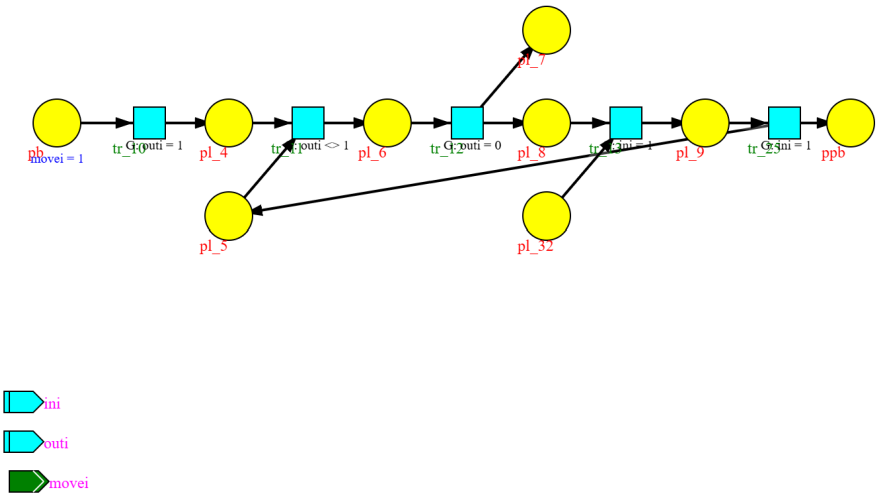


Figura 1: Rede de Petri — Tapete Singular

3.1.2 Rede de Petri — Modelo Completo

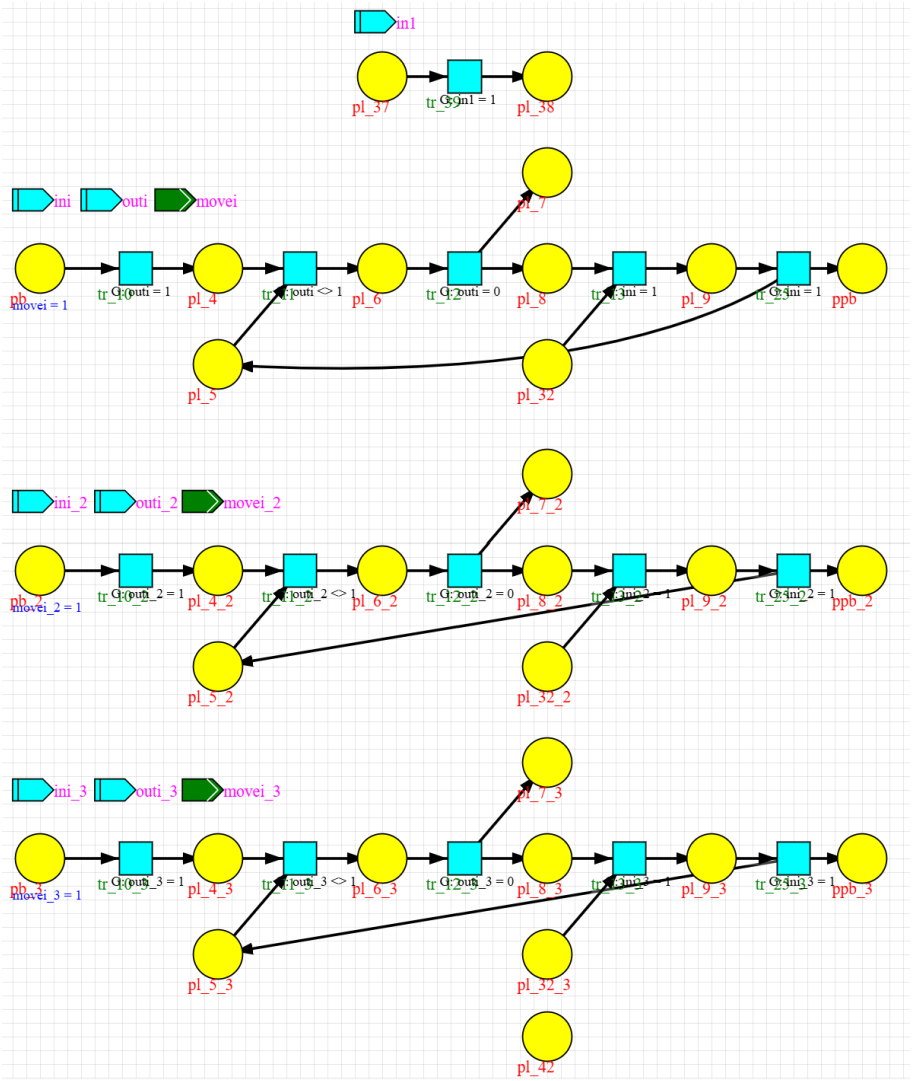


Figura 2: Rede de Petri — Modelo Completo

3.1.3 Rede de Petri — Fusion set

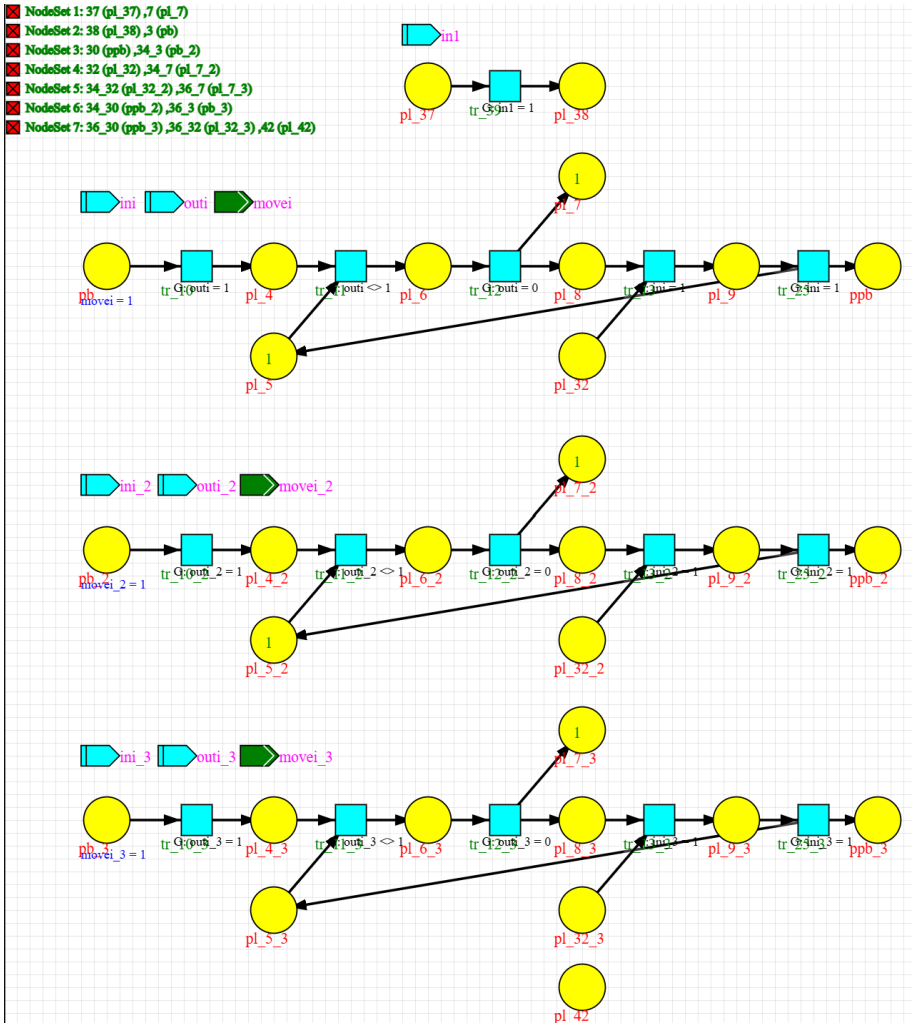


Figura 3: Rede de Petri — Fusion set

3.1.4 Rede de Petri — Node set

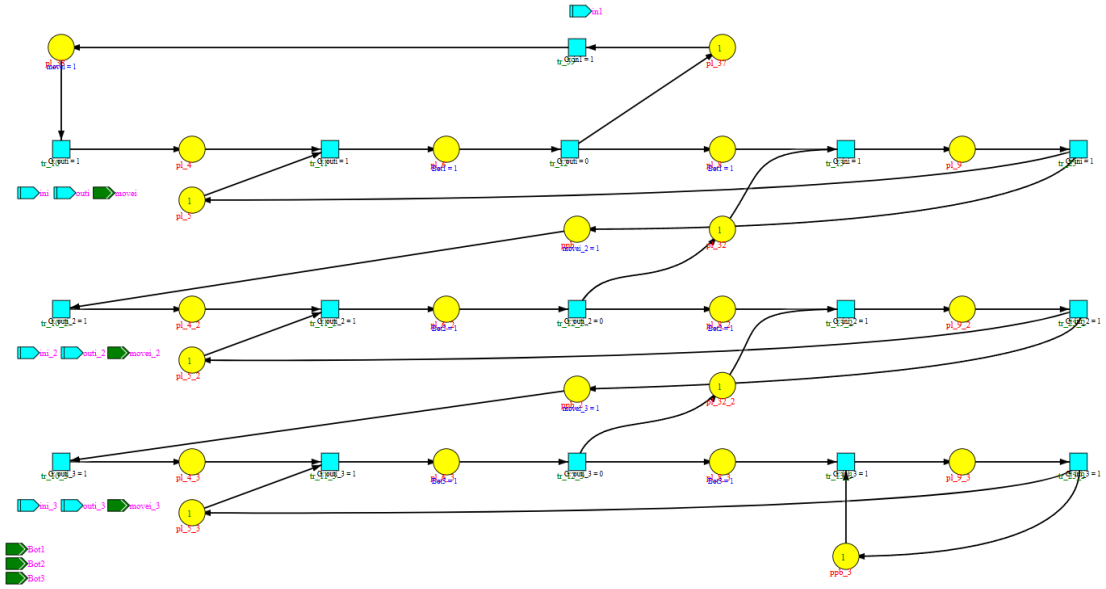


Figura 4: Rede de Petri - Node set

3.2 Simulação e Análise

3.2.1 Simulação com token-player

Neste link tem acesso ao vídeo da simulação com token-player:

https://unlpt-my.sharepoint.com/:v:/g/personal/jafe_lopes_fct_unl_pt/EUvkbe_KlbRFrrdI0gsSh98Bu2URieTdJFFzwphaj5HS6A?e=i5XMNg&nav=eyJyZWZlcnJhbEluZm8iOns3D

3.2.2 Geração e análise do espaço de estados

```
586884 Started state-space generation.

Cycle 1: 1 states + 0 links
Cycle 2: 2 states + 0 links
Cycle 3: 3 states + 0 links
Cycle 4: 4 states + 0 links
Cycle 5: 5 states + 0 links
Cycle 6: 8 states + 0 links
Cycle 7: 13 states + 4 links
Cycle 8: 18 states + 10 links
Cycle 9: 25 states + 18 links
Cycle 10: 34 states + 30 links
Cycle 11: 51 states + 48 links
Cycle 12: 70 states + 98 links
Cycle 13: 103 states + 174 links
Cycle 14: 146 states + 306 links
Cycle 15: 211 states + 510 links
Cycle 16: 304 states + 808 links
Cycle 17: 351 states + 1396 links
Cycle 18: 432 states + 1852 links
Cycle 19: 474 states + 2393 links
Cycle 20: 524 states + 2769 links
Cycle 21: 566 states + 3145 links
Cycle 22: 578 states + 3373 links
Cycle 23: 598 states + 3485 links
Cycle 24: 606 states + 3601 links
Cycle 25: 614 states + 3657 links
Cycle 26: 622 states + 3713 links

MIN Bounds: pl_32=0 pl_32_2=0 pl_37=0 pl_38=0 pl_4=0 pl_4_2=0 pl_4_3=0 pl_5=0 pl_5_2=0 pl_5_3=0 pl_6=0 pl_6_2=0 pl_6_3=0 pl_8=0 pl_8_2=0 pl_8_3=0 pl_9=0 pl_9_2=0 pl_9_3=0 ppb=0 ppb_2=0 ppb_3=0

MAX Bounds: pl_32=1 pl_32_2=1 pl_37=1 pl_38=1 pl_4=1 pl_4_2=1 pl_4_3=1 pl_5=1 pl_5_2=1 pl_5_3=1 pl_6=1 pl_6_2=1 pl_6_3=1 pl_8=1 pl_8_2=1 pl_8_3=1 pl_9=1 pl_9_2=1 pl_9_3=1 ppb=1 ppb_2=1 ppb_3=1

##################
Total States:   622
Total Links:    3741
##################

Executing queries...
Done: found 0 query matching states.

Generation time (sec): 0.00 (when 0.00 it is smaller than 0.01sec)

Generating output file.
Done.

##################
Total States:   622
Total Links:    3741
Deadlock count: 0
Conflict count: 0
Invalid count:  0
##################
```

Figura 5: Rede de Petri — Espaço de Estados

4 Implementação em FPGA

Nesta secção, detalha-se o processo de implementação do sistema em uma FPGA (Field-Programmable Gate Array).

4.1 Geração do código VHDL

O código VHDL encontra-se descrito na figura abaixo.

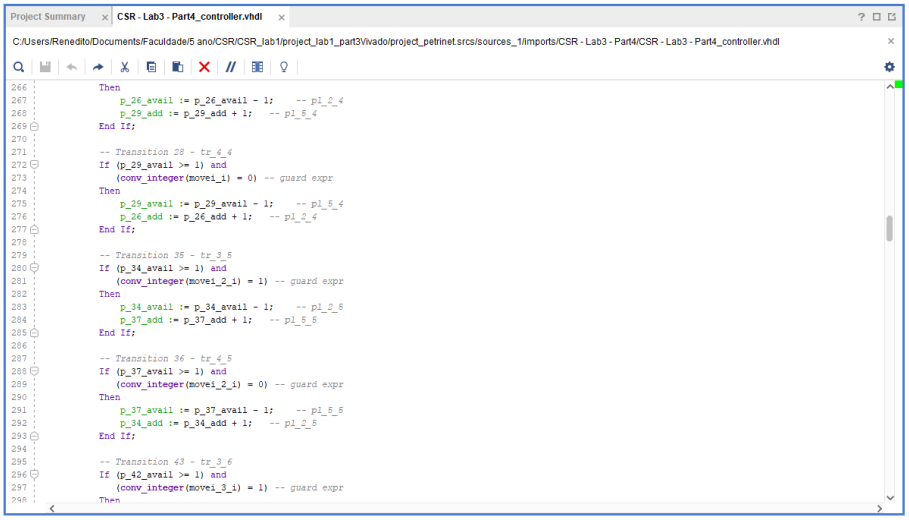


Figura 6: Código VHDL gerado para implementação em FPGA

Este código foi gerado automaticamente a partir do modelo de Rede de Petri utilizando a framework IOPT-Tools, que traduz as especificações do modelo para uma descrição em VHDL adequada para síntese em FPGA.

4.2 Configuração de entradas/saídas

Para este projeto, foram configuradas as seguintes entradas e saídas na FPGA:

```
Port(
    Clk : IN STD_LOGIC;
    ini : IN STD_LOGIC;
    outi : IN STD_LOGIC;
    ini_2 : IN STD_LOGIC;
    outi_2 : IN STD_LOGIC;
    ini_3 : IN STD_LOGIC;
    outi_3 : IN STD_LOGIC;
    inl : IN STD_LOGIC;
    Botl_i : IN STD_LOGIC;
    Bot2_i : IN STD_LOGIC;
    Bot3_i : IN STD_LOGIC;
    movei_i : IN STD_LOGIC;
    movei_2_i : IN STD_LOGIC;
    movei_3_i : IN STD_LOGIC;
    Botl : OUT STD_LOGIC;
    Bot2 : OUT STD_LOGIC;
    Bot3 : OUT STD_LOGIC;
    movei : OUT STD_LOGIC;
    movei_2 : OUT STD_LOGIC;
    movei_3 : OUT STD_LOGIC;
    inl_o : OUT STD_LOGIC;
    ini_o : OUT STD_LOGIC;
    outi_o : OUT STD_LOGIC;
    ini_2_o : OUT STD_LOGIC;
    outi_2_o : OUT STD_LOGIC;
    ini_3_o : OUT STD_LOGIC;
    outi_3_o : OUT STD_LOGIC;
    Enable : IN STD_LOGIC;
    Reset : IN STD_LOGIC
);
End CSR_Lab3_Part4;
```

Figura 7: Configuração de entradas e saídas na FPGA

4.3 Testes experimentais

5 Implementação em Arduino

5.1 Geração do código C

O código C encontra-se descrito na figura abaixo.

net_main.ino | net_exec_step.c | net_functions.c | net_io.c | net_types.h

```c
/* Net CSR___Lab_3___Part1 - IOPT */
/* Automatic code generated by IOPT2C XSLT transformation. */


#include <stdlib.h>
#include "net_types.h"


int trace_control = TRACE_CONT_RUN;

extern void httpServer_init();
extern void httpServer_getRequest();
extern void httpServer_sendResponse();
extern void httpServer_disconnectClient();
extern void httpServer_checkBreakPoints();
extern void httpServer_finish();

static CSR___Lab_3___Part1_NetMarking marking;
static CSR___Lab_3___Part1_InputSignals inputs, prev_inputs;
static CSR___Lab_3___Part1_PlaceOutputSignals place_out;
static CSR___Lab_3___Part1_EventOutputSignals ev_out;

void setup()
{
    createInitial_CSR___Lab_3___Part1_NetMarking( &marking );
    init_CSR___Lab_3___Part1_OutputSignals( &place_out, &ev_out );
    CSR___Lab_3___Part1_InitializeIO();
    CSR___Lab_3___Part1_GetInputSignals( &prev_inputs, NULL );
#ifdef HTTP_SERVER
    httpServer_init();
#endif
}

void loop()
{
#ifdef HTTP_SERVER
    httpServer_getRequest();
#endif

    if( trace_control != TRACE_PAUSE )
        CSR___Lab_3___Part1_ExecutionStep( &marking, &inputs, &prev_inputs, &place_out, &ev_out );
    else CSR___Lab_3___Part1_GetInputSignals( &inputs, NULL );
    if( trace_control > TRACE_PAUSE ) --trace_control;

#ifdef HTTP_SERVER
```

Figura 8: Código C gerado para implementação em Arduino

Este código foi gerado automaticamente a partir do modelo de Rede de Petri utilizando a framework IOPT-Tools, que traduz as especificações do modelo para uma descrição em C adequada para execução em microcontroladores Arduino.

5.2 Configuração de entradas/saídas

Para este projeto, foram configuradas as seguintes entradas e saídas no Arduino:

Figura 9: Configuração de entradas e saídas no Arduino

5.3 Testes

Os testes experimentais foram realizados para validar o funcionamento do sistema implementado no Arduino. Os resultados obtidos foram comparados com os resultados da simulação para garantir a consistência e a precisão do modelo.

Figura 10: Resultados dos testes experimentais no Arduino

5.4 Comparação com simulação e implementação FPGA

6 Controlador Distribuído em regime Síncrono

6.1 Cutting Set

A identificação do conjunto de corte (cutting set) é um passo crucial na decomposição de sistemas distribuídos. Este conjunto consiste em um grupo de canais ou conexões que, quando removidos, dividem o sistema em subsistemas independentes. A escolha adequada do cutting set permite a implementação eficiente de controladores distribuídos, minimizando a complexidade da comunicação entre os módulos.

6.2 Decomposição SPLIT

A decomposição usando SPLIT (Synchronous Parallel Logic Interconnect Technology) permite a criação de uma arquitetura mais flexível e escalável, dividindo o sistema em módulos independentes que se comunicam através de canais síncronos. Essa abordagem facilita a implementação de sistemas complexos, permitindo a reutilização de componentes e a redução do tempo de desenvolvimento. A decomposição SPLIT para esta aplicação pode ser visualizada na figura abaixo.

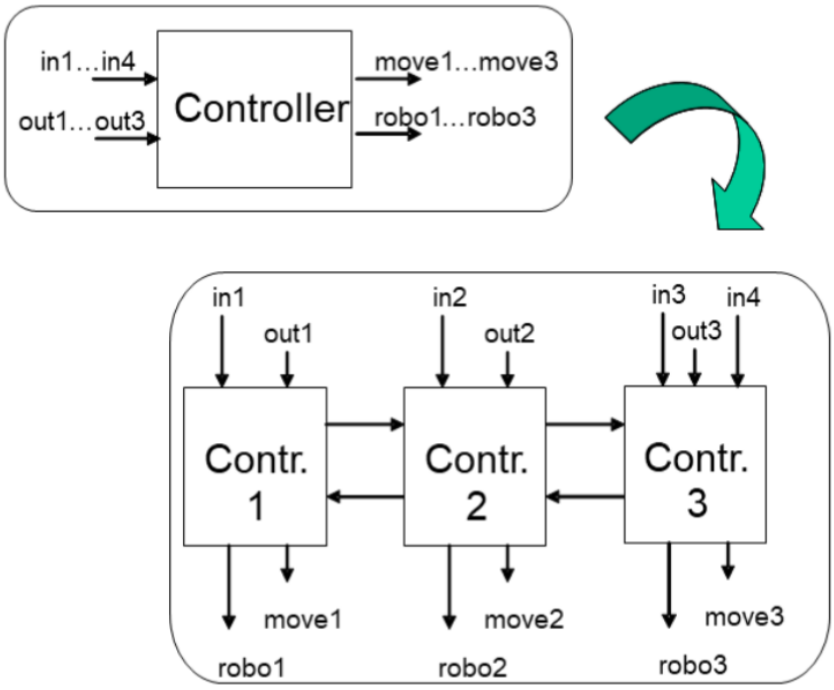


Figura 11: Decomposição SPLIT do sistema em módulos independentes.

6.3 Modelo com canais síncronos

A modelação com canais síncronos envolve a definição de canais de comunicação que operam em sincronia, permitindo a troca de informações entre os módulos de forma coordenada. Essa abordagem é fundamental para garantir a consistência e a previ-

sibilidade no comportamento do sistema. Esta modelação encontra-se representada na figura abaixo.

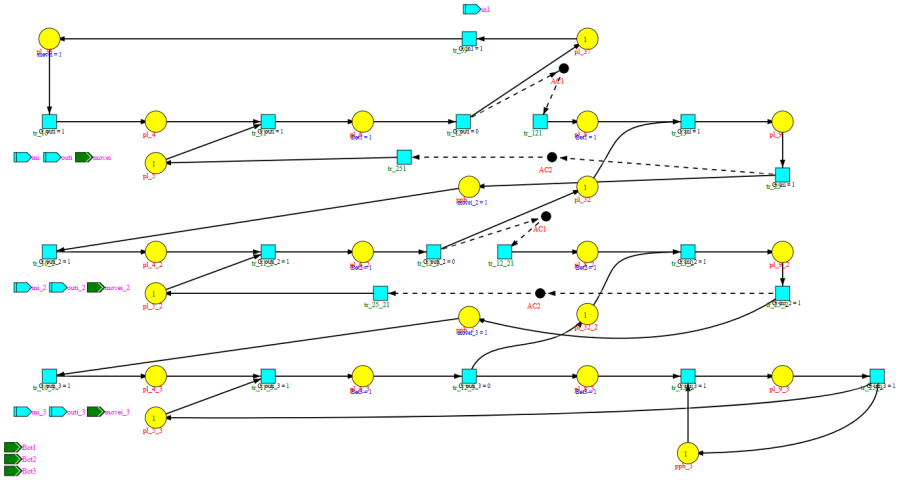


Figura 12: Modelo com canais síncronos entre os módulos.

6.4 Modelo com canais assíncronos

A modelação com canais assíncronos envolve a definição de canais de comunicação que operam de forma independente, permitindo a troca de informações entre os módulos sem a necessidade de sincronização. Essa abordagem é útil em sistemas onde a latência e a largura de banda são variáveis, proporcionando maior flexibilidade na comunicação. Este modelo está descrito na figura abaixo.

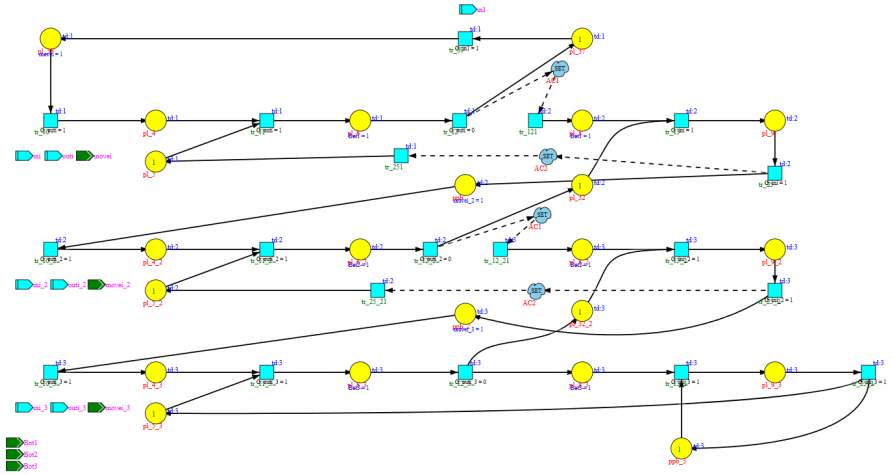
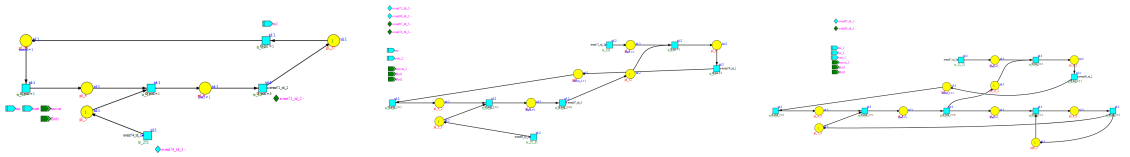


Figura 13: Modelo com canais assíncronos entre os módulos.

6.5 Decomposição GALS

A decomposição GALS (Globally Asynchronous, Locally Synchronous) envolve a criação de três controladores separados que operam de forma independente, permitindo uma maior flexibilidade e escalabilidade no sistema. Essa abordagem facilita

a integração de diferentes módulos e a adaptação a variações nas condições de operação. A decomposição GALS para esta aplicação pode ser visualizada nas figuras abaixo.



(a) Tapete 1 com controlo por eventos

(b) Tapete 2 com controlo por eventos

(c) Tapete 3 com controlo por eventos

Figura 14: Decomposição GALS — Tapetes 1–3 com controlo por eventos

6.6 Simulação e Análise

6.6.1 Simulação com token-player

6.6.2 Geração e análise do espaço de estados

7 Implementação em FPGA - Distribuição Síncrona

7.1 Geração do código VHDL

7.2 Implementação do LFSR

O LFSR (Linear Feedback Shift Register) é um componente essencial para a geração de números pseudoaleatórios no sistema. A sua implementação em VHDL envolve a definição dos registos de shift (deslocamento) e das funções de feedback necessárias para garantir a pseudo-aleatoriedade dos valores gerados. Usando como referência o documento fornecido pelos docentes, procedeu-se à implementação do LFSR com as características desejadas.

Esta secção detalha o processo de implementação do LFSR, incluindo a configuração dos seus parâmetros e a integração com os restantes componentes do sistema.

7.2.1 Gestão dos atrasos

7.3 Deployment na FPGA

7.4 Testes e análise de resultados

8 Buffers de Capacidade Finita

8.1 Modificação do modelo

A modificação do modelo para múltiplas peças envolve a adaptação dos componentes do sistema para lidar com a entrada e saída de várias peças simultaneamente. Isso inclui a implementação de buffers de capacidade finita que permitem o armazenamento temporário das peças, garantindo que o sistema possa operar de forma eficiente mesmo quando há variações na taxa de entrada ou saída.

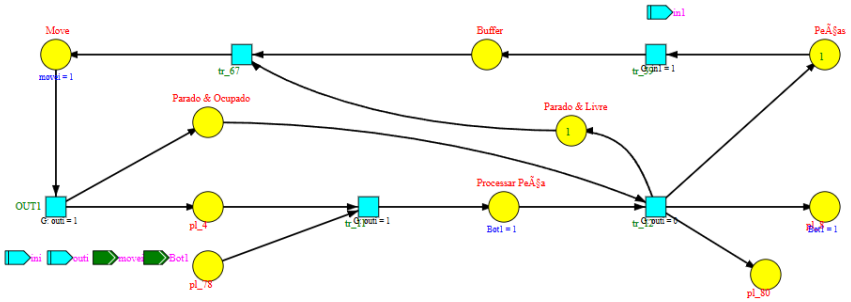


Figura 15: Rede de Petri - modelo modificado para múltiplas peças (tapete único)

8.2 Interconexão dos componentes

A interconexão dos componentes no modelo modificado é realizada através de estados capazes de gerir a presença de múltiplos objetos. Garante-se assim o fluxo das peças de uma forma mais eficiente e que os buffers possam operar corretamente dentro das suas capacidades definidas.

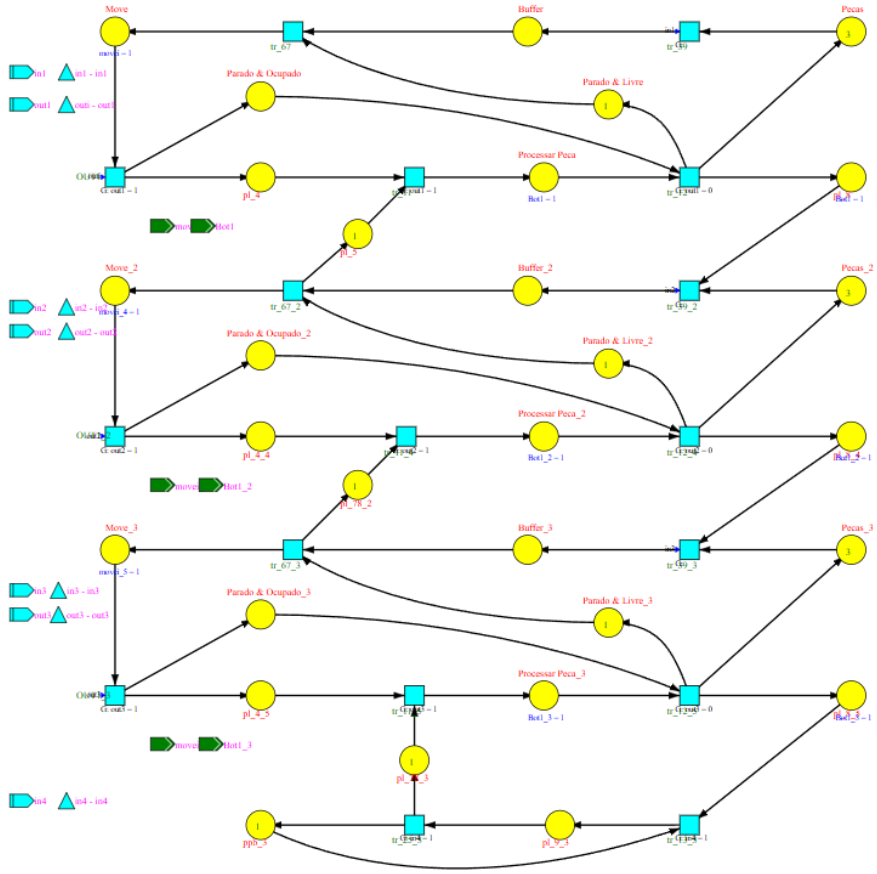


Figura 16: Rede de Petri - interconexão dos diferentes tapetes

8.3 Módulo de visualização do número de objetos

8.4 Simulação e Análise

8.4.1 Simulação com token-player

8.4.2 Geração e análise do espaço de estados

8.5 Implementação em FPGA

8.6 Análise de resultados

9 Distribuído com Buffers (Síncrono)

9.1 Aplicação síncrona ao buffer de capacidade finita

A aplicação síncrona ao buffer de capacidade finita envolve a utilização de canais síncronos para gerir o fluxo de dados entre os diferentes módulos do sistema. Esta abordagem permite que o buffer opere de forma eficiente, garantindo que as peças sejam armazenadas e transferidas de acordo com um clock global, o que facilita a coordenação entre os componentes (minimizando a chance de perda de peças).

9.2 Decomposição e implementação distribuída

A decomposição e implementação distribuída do sistema com buffers de capacidade finita envolve a divisão do sistema em módulos independentes que se comunicam através de canais síncronos. Cada módulo é responsável por uma parte específica do processamento, como a gestão do buffer, a entrada e saída de peças, e o controlo dos atuadores. Esta abordagem permite uma maior flexibilidade e escalabilidade, facilitando a adaptação do sistema a diferentes requisitos operacionais.

9.3 Execução síncrona global

9.4 Testes e análise

10 Distribuído com Buffers (Assíncrono)

10.1 Aplicação assíncrona ao buffer de capacidade finita

A aplicação assíncrona ao buffer de capacidade finita envolve a utilização de canais assíncronos para gerir o fluxo de dados entre os diferentes módulos do sistema. Esta abordagem permite que o buffer opere de forma eficiente, garantindo que as peças sejam armazenadas e transferidas sem a necessidade de um clock global, o que facilita a coordenação entre os componentes.

10.2 Comunicações não-instantâneas com buffers

10.3 Deployment e testes

10.4 Análise comparativa

11 Comparação de abordagens

11.1 Centralizado vs Distribuído

Na abordagem centralizada, todas as operações e decisões foram geridas por um único componente, o que simplificou o design e a implementação inicial. No entanto, esta abordagem levou a limitações de escalabilidade, especialmente em sistemas complexos. Na abordagem distribuída, o sistema foi dividido em múltiplos módulos independentes que comunicam entre si. Esta divisão permitiu uma maior flexibilidade, escalabilidade e resiliência, embora tenha introduzido desafios adicionais relacionados com a comunicação e sincronização entre os módulos.

11.2 Síncrono vs Assíncrono

Em regime síncrono, todos os componentes do sistema operaram em sincronia com um clock global, o que facilitou a coordenação e a previsibilidade do comportamento do sistema. No entanto, esta abordagem levou a ineficiências, especialmente quando há variações nas peças entre os tapetes. Já em regime assíncrono, os componentes operaram de forma independente, permitindo uma maior flexibilidade e adaptabilidade às variações no fluxo de peças. Esta abordagem, no entanto, introduziu complexidades adicionais na gestão da comunicação e na garantia da integridade dos dados transmitidos entre os módulos.

11.3 Impacto dos atrasos de comunicação

Os atrasos de comunicação tiveram um impacto significativo no desempenho do sistema, especialmente nas abordagens distribuídas. Em sistemas síncronos, os atrasos podem levar a falhas na sincronização, resultando em perda de peças ou congestionamento. Em sistemas assíncronos, os atrasos podem causar inconsistências na comunicação entre os módulos, exigindo mecanismos adicionais para garantir a integridade dos dados.

11.4 Vantagens e Desvantagens

Tabela 2: Comparação: Centralizado vs Distribuído segundo modo e presença de atrasos

Abordagem	**Síncrono**	**Assíncrono**	**Sem delay**	**Com delay**
Centralizado	Coordenação simples; comportamento previsível; fácil depuração.	Menos flexível; conflitos centralizados; maior complexidade de gestão.	Baixa latência global; desempenho ideal quando não há comunicação intensa.	Propenso a gargalos; perda de sincronismo; ponto único de falha evidente.
Distribuído	Requer mecanismos de sincronização entre nós; overhead de coordenação.	Boa adaptação a ritmos locais; maior modularidade e resiliência.	Eficiência local elevada; módulos podem operar com baixa latência interna.	Necessita de protocolos, buffers e tolerância a atrasos; maior complexidade de implementação.

11.5 Análise de escalabilidade

12 Conclusões

12.1 Resultados alcançados

12.2 Dificuldades encontradas e soluções adotadas