

Porto Alegre

2018

**PROJETOS DE SISTEMAS INTEGRADOS 1 – TRABALHO FINAL**

HENRIQUE KRAUSBURG CORRÊA; GIUSEPPE GENEROSO

ESCOLA POLITÉCNICA

CURSO DE ENGENHARIA DA COMPUTAÇÃO

# INTRODUÇÃO

Este relatório apresenta o trabalho final da disciplina Projeto de Sistemas Integrados 1. A proposta é realizar o desenvolvimento e verificação de uma entidade do jogo **BLACKJACK**.

Inicialmente, será apresentada a arquitetura da entidade **BLACKJACK**, destacando as lógicas combinacionas e sequencias utilizadas. Depois, é apresentado o testbench com os casos de testes elaborados e as métricas utilizadas. Por fim, apresenta-se as conclusões finais do trabalho.

# ARQUITETURA

A arquitetura do programa blackjack, conforme demonstrado na Figura 1, é dividida em seis partes lógicas distintas. Todas exceto as lógicas combinacionais recebem os sinais de entrada **CLK** e **RST**.

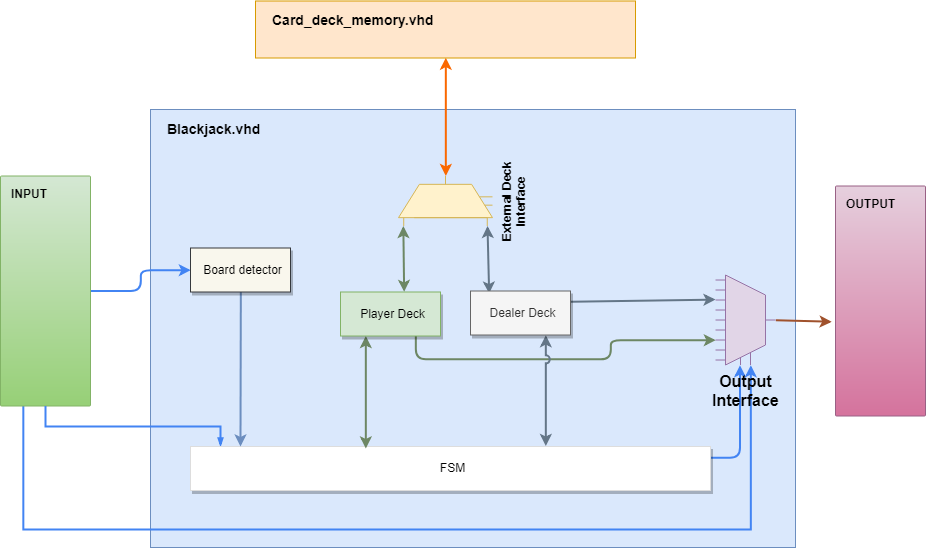


Figura 1 - Diagrama em blocos arquitetura blackjack

A Tabela 1 apresenta a descrição utilizada para a entidade **BLACKJACK:**

|  |
| --- |
| **entity** blackjack **is**  **generic(**  CLK\_EDGE **:** std\_logic **:=** '1'**);**  **port(**  clk **:** **in** std\_logic**;**  reset **:** **in** std\_logic**;**  stay **:** **in** std\_logic**;**  hit **:** **in** std\_logic**;**  debug **:** **in** std\_logic**;**  show **:** **in** std\_logic**;**  card **:** **in** std\_logic\_vector**(**3 **downto** 0**);**  request **:** **out** std\_logic**;**  win **:** **out** std\_logic**;**  lose **:** **out** std\_logic**;**  tie **:** **out** std\_logic**;**  total **:** **out** std\_logic\_vector**(**4 **downto** 0**));**  **end** blackjack**;** |

Tabela 1 - Descrição entidade blackjack

## Board Detector

Realiza a detecção de borda de subida tanto dos sinais de entrada **HIT** e **STAY**. Desta forma, é possível prevenir-se de uma condição inválida de pressionar os dois botões simultaneamente.

|  |
| --- |
| --board detector process  shift\_reg**:** **process(**clk**)**  **begin**  **if** clk'**event** and clk **=** CLK\_EDGE **then**  **if(**reset **=** '1'**)** **then**  s\_hit\_reg\_a **<=** '0'**;**  s\_hit\_reg\_b **<=** '0'**;**    s\_stay\_reg\_a **<=** '0'**;**  s\_stay\_reg\_b **<=** '0'**;**  **else**  s\_hit\_reg\_b **<=** s\_hit\_reg\_a**;**  s\_hit\_reg\_a **<=** hit**;**    s\_stay\_reg\_b **<=** s\_stay\_reg\_a**;**  s\_stay\_reg\_a **<=** stay**;**  **end** **if;**  **end** **if;**  **end** **process** shift\_reg**;**    --Board detector signal  s\_hit\_rise **<=** '1' **when** **(**s\_hit\_reg\_a **=** '1'**)** and **(**s\_hit\_reg\_b **=** '0'**)** **else**  '0'**;**    s\_stay\_rise **<=** '1' **when** **(**s\_stay\_reg\_a **=** '1'**)** and **(**s\_stay\_reg\_b **=** '0'**)** **else**  '0'**;** |

Tabela 2 - Implementação Board Detector

Os sinais **S\_HIT\_RISE** e **S\_STAY\_RISE** serão lidos posteriormente pela máquina de estados.

## FSM

Máquina de estados mestre do blackjack. É responsável por controlar os estados futuros conforme o valor dos sinais recebidos de outras entidades. O principal objetivo é controlar as entidades **PLAYER** e **DEALER**, indicando quem deve jogar, esperar, pedir uma nova carta e encerrar o jogo. Também limita os sinais de saída conforme estado do jogo. Os estados serão detalhados no capítulo Máquina de estados blackjack.

## Player / Dealer Deck

Entidades que representam o controle do jogador e da banca. Pelo fato de ambos apresentarem um comportamento extremamente semelhante, foi desenvolvida uma única entidade deck controller. Tanto o jogador quanto a banca instanciam esta entidade que controla o respectivo deck. A implementação da entidade será detalhada no capítulo Controlador Player / Dealer.

|  |
| --- |
| --Player deck instantiation  player\_deck**:** **entity** work**.**deck\_controller  **generic** **map**  **(**  CLK\_EDGE **=>** CLK\_EDGE**,**  IS\_FIRST\_TO\_PLAY **=>** '1'  **)**  **port** **map**  **(**  clk\_in **=>** clk**,**  rst\_in **=>** reset**,**  start\_in **=>** s\_player\_start\_in**,**  encoded\_card\_in **=>** card**,**  buy\_card\_out **=>** s\_player\_buy\_card\_out**,**  is\_available\_out **=>** s\_player\_is\_available\_out**,**  score\_out **=>** s\_player\_score\_out  **);**    --Dealer deck instantiation  dealer\_deck**:** **entity** work**.**deck\_controller  **generic** **map**  **(**  CLK\_EDGE **=>** CLK\_EDGE**,**  IS\_FIRST\_TO\_PLAY **=>** '0'  **)**  **port** **map**  **(**  clk\_in **=>** clk**,**  rst\_in **=>** reset**,**  start\_in **=>** s\_dealer\_start\_in**,**  encoded\_card\_in **=>** card**,**  buy\_card\_out **=>** s\_dealer\_buy\_card\_out**,**  is\_available\_out **=>** s\_dealer\_is\_available\_out**,**  score\_out **=>** s\_dealer\_score\_out  **);** |

Tabela 3 - Instanciação Player / Dealer

## External Deck Interface

Lógica combinacional que controla o fluxo de dados entre as entidades **PLAYER** e **DEALER** com o deck externo responsável por fornecer as cartas do jogo.

O sinal de entrada **CARD** é propagado diretamente para ambas as entidades e não necessita de nenhum controle especial.

No entanto, o sinal de saída **REQUEST** é controlado diretamente pelas entidades **PLAYER / DEALER** e indiretamente controlado pela **FSM**, que coordena qual entidade deve jogar e por sua vez obter o controle sobre o sinal **REQUEST**.

|  |
| --- |
| --Send signal to player deck controller start the card requisition process  s\_player\_start\_in **<=** '1' **when** **(**state **=** PLAYER\_BEGIN\_1**)** or **(**state **=** PLAYER\_BEGIN\_2**)** or **(**state **=** PLAYER\_HIT**)** **else**  '0'**;**    --Send signal to dealer deck controller start the card requisition process  s\_dealer\_start\_in **<=** '1' **when** **(**state **=** DEALER\_BEGIN\_1**)** or **(**state **=** DEALER\_BEGIN\_2**)** or **(**state **=** DEALER\_HIT**)** **else**  '0'**;**  --External deck FIFO card request mux  request **<=** s\_player\_buy\_card\_out **when** s\_player\_is\_available\_out **=** '0' **else**  s\_dealer\_buy\_card\_out **when** s\_dealer\_is\_available\_out **=** '0' **else**  '0'**;** |

Tabela 4 - Implementação External Deck Interface

## Output Interface

Lógica combinacional que controla o fluxo de dados de saída da entidade **BLACKJACK** referente a pontuação do jogo e o resultado final.

Os sinais **WIN / LOSE / TIE** são independentes entre si, avaliando apenas se o jogo já acabou através do estado da **FSM** e a pontuação do **PLAYER / DEALER**.

O sinal **TOTAL** exibe a pontuação do **PLAYER** ou do **DEALER** e é controlado diretamente pelos sinais de entrada **SHOW** e **DEBUG**.

|  |
| --- |
| --Player win signal is displayed when game ends, score is not above 21 and higher than dealer score  win **<=** '1' **when** next\_state **=** GAME\_OVER and s\_player\_score\_out **<=** 21 and **((**s\_dealer\_score\_out **>** 21**)** or **(**s\_player\_score\_out **>** s\_dealer\_score\_out**))** **else**  '0'**;**    --Player lose signal is displayed when game ends, score is above 21 or lower than dealer score  lose **<=** '1' **when** next\_state **=** GAME\_OVER and s\_dealer\_score\_out **<=** 21 and **((**s\_player\_score\_out **>** 21**)** or **(**s\_player\_score\_out **<** s\_dealer\_score\_out**))** **else**  '0'**;**    --Game tie signal is displayed when game ends, player and dealer score are equal  tie **<=** '1' **when** next\_state **=** GAME\_OVER and s\_player\_score\_out **=** s\_dealer\_score\_out **else**  '0'**;**    --Total score output: display player score or dealer score according to the table below:  -- state signal | DEBUG input | SHOW input | TOTAL output  -- x | 1 | 0 | player score  -- x | 1 | 1 | dealer score  -- GAME\_OVER | x | 0 | player score  -- GAME\_OVER | x | 1 | dealer score  -- x | x | 0 | player score  total **<=** s\_dealer\_score\_out **when** show **=** '1' and **((**state **=** GAME\_OVER**)** or **(**debug **=** '1'**))** **else**  s\_player\_score\_out**;** |

Tabela 5 - Implementação Output Interface

## Máquina de estados blackjack

A **FSM** possui 14 estados conforme ilustrado na Figura 2. A transição do estado atual para o estado futuro ocorre apenas durante a borda de subida de **CLK**. O estado futuro é avaliado em um process com uma lista de sensitividade composta pelo estado atual e pelos sinais relevantes para a transição.

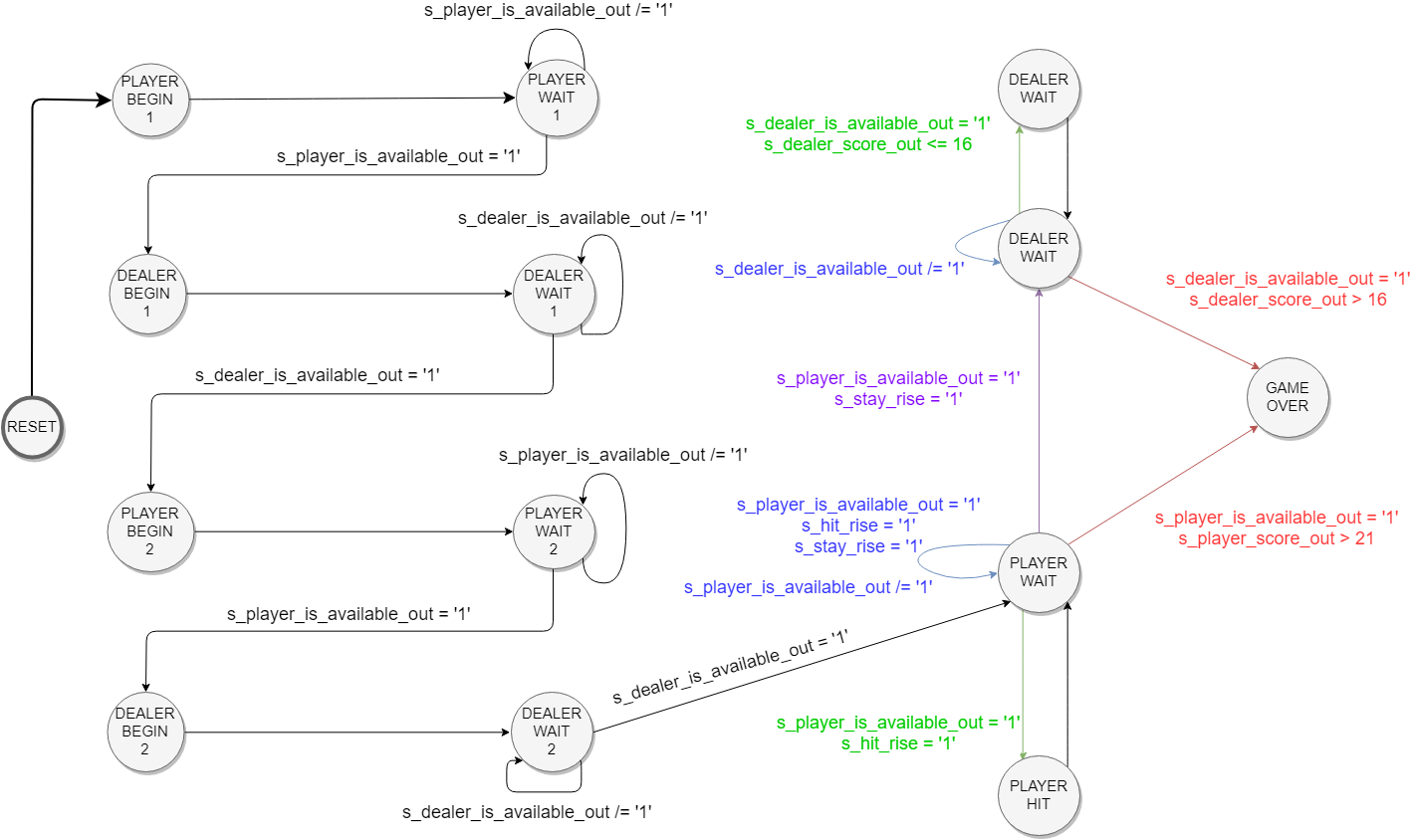


Figura 2 - Máquina de estados blackjack

### RESET

Condição inicial da **FSM**. Independente do estado atual que estiver, salta para este estado quando se detecta que o sinal **RESET** está em ‘1’ durante a borda de subida do sinal **CLK**.

Quando **RESET** não estiver em ‘1’, salta para o estado PLAYER BEGIN 1.

### PLAYER / DEALER BEGIN 1 e 2

Habilita entidade para comprar uma carta e pula para o próximo estado.

### PLAYER / DEALER WAIT 1 e 2

Aguarda entidade concluir sua jogada. Quando concluído, salta para o estado.

### PLAYER WAIT

Aguarda interação do jogador realizando a leitura dos comandos **HIT** e **STAY**. Quando a entidade **PLAYER** está disponível (s\_player\_is\_available\_out=’1’), verifica as seguintes condições:

* Se a pontuação do **PLAYER** ultrapassou os 21 pontos, o jogo é encerrado pulando para o estado GAME OVER.
* Se os ‘botões’ **HIT** e **STAY** foram pressionados simultaneamente (‘1’), ignora instrução e permanece no mesmo estado.
* Se o ‘botão’ **STAY** foi pressionado (‘1’), encerra jogadas do **PLAYER** saltando para o estado DEALER WAIT.
* Se o ‘botão’ **HIT** foi pressionado (‘1’), salta para o estado PLAYER HIT.

Nenhum procedimento é realizado se a entidade não estiver disponível ou se não ocorreu nenhuma das condições apresentadas.

### PLAYER HIT

Semelhante ao estado PLAYER BEGIN, Habilita **PLAYER** para comprar uma carta e retorna para o estado para o estado PLAYER WAIT.

### DEALER WAIT

Quando a entidade **DEALER** está disponível (s\_dealer\_is\_available\_out=’1’), verifica se sua pontuação atual ultrapassou 16 pontos. Enquanto não ultrapassar, salta para o estado DEALER HIT para permanecer comprando cartas até que ultrapasse para então pular para o estado GAME OVER.

Nenhum procedimento é realizado se a entidade não estiver disponível ou se não ocorreu nenhuma das condições apresentadas.

### DEALER HIT

Semelhante ao estado DEALER BEGIN, Habilita **DEALER** para comprar uma carta e retorna para o estado para o estado DEALER WAIT.

### GAME OVER

Habilita os resultados finais do jogo. Permanece nesse estado até que o jogo seja reiniciado.

## Controlador Player / Dealer

A implementação do controlador é definida no arquivo deck\_controller.vhd. A principal diferença entre as instanciações é que a entidade **PLAYER** atribui ‘1’ para a porta genérica **IS\_FIRST\_TO\_PLAY**, enquanto **DEALER** atribui ‘0’.

Seu propósito é solicitar cartas ao deck externo, decodifica-las, computar a pontuação e avaliar o caso especial da carta às, caso tenha recebido uma.

|  |
| --- |
| **entity** deck\_controller **is**  **generic(**  CLK\_EDGE **:** std\_logic **:=** '1'**;**  IS\_FIRST\_TO\_PLAY **:** std\_logic **:=** '0'**);**    **port(**  clk\_in **:** **in** std\_logic**;**  rst\_in **:** **in** std\_logic**;**  start\_in **:** **in** std\_logic**;**  encoded\_card\_in **:** **in** std\_logic\_vector**(**3 **downto** 0**);**  buy\_card\_out **:** **out** std\_logic**;**  is\_available\_out **:** **out** std\_logic**;**  score\_out **:** **out** std\_logic\_vector**(**4 **downto** 0**));**  **end** deck\_controller**;** |

Tabela 6 - Entidade deck\_controller

O controlador é composto por uma lógica combinacional para controlar quatro sinais e é apresentada na Tabela 7 e uma máquina de estados ilustrada na Figura 3.

|  |
| --- |
| --Controls busy-wait flag  s\_is\_available **<=** '1' **when** state **=** RESET or next\_state **=** IDLE **else**  '0'**;**    is\_available\_out **<=** s\_is\_available**;**    --Asks for a new card according to FSM state  buy\_card\_out **<=** '1' **when** state **=** BUY\_CARD **else**  '0'**;**    --Decode requested card  s\_decoded\_card **<=** encoded\_card\_in **when** s\_is\_available **=** '0' and encoded\_card\_in **<** 10 **else** --Card is invalid (0) or an A, 2, 3, 4, 5, 6, 7, 8 or 9  x"A" **when** s\_is\_available **=** '0' and encoded\_card\_in **<** 14 **else** -- Card is 10, J, Q or K  "0000"**;** --Invalid decoded value    --Displays score. While working, displays the previus calculated score  score\_out **<=** s\_score **when** s\_is\_available **=** '1'**;** |

Tabela 7 - Lógica combinacional deck\_controller

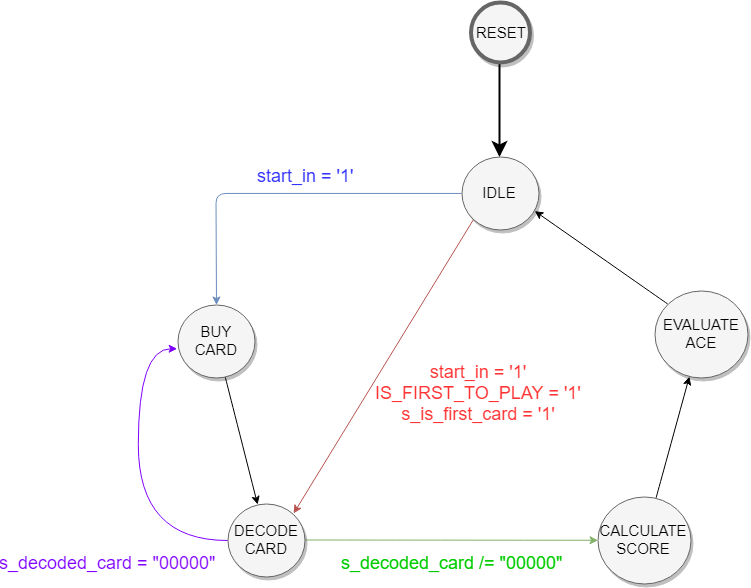


Figura 3 - Máquina de estados deck\_controller

### RESET

Condição inicial da **FSM**. Independente do estado atual que estiver, salta para este estado quando se detecta que o sinal **RESET** está em ‘1’ durante a borda de subida do sinal **CLK**.

Quando **RESET** não estiver em ‘1’, salta para o estado IDLE e atribui ‘1’ ao sinal interno s\_is\_first\_card.

### IDLE

Permanece esperando até que habilitado pela entidade **BLACKJACK** para comprar uma carta (start\_in = ‘1’).

Quando recebe esta instrução, antes é avaliado se esta entidade é o **PLAYER** (IS\_FIRST\_TO\_PLAY = ‘1’) e se é a primeira carta do jogo (s\_is\_first\_card = ‘1’). Caso afirmativo, pula diretamente para o estado DECODE CARD. Caso negativo, segue o procedimento normal saltando para o estado BUY CARD.

### BUY CARD

Envia solicitação ao deck externo para comprar uma carta. Permanece por um ciclo neste estado e pula para DECODE CARD.

### DECODE CARD

Verifica se a carta foi convertida corretamente para a pontuação correspondente avaliando o sinal s\_decoded\_card. Caso o valor seja 0, significa que o código da carta é inválida e deve retornar ao estado BUY CARD. Caso contrário, prossegue saltando para CALCULATE SCORE.

Ao sair desse estado, o sinal s\_is\_first\_card recebe o valor ‘0’.

### CALCULATE SCORE

Quando neste estado, habilita o processo calculate\_score\_proc para somar a pontuação atual com a pontuação da nova carta. Ao final, salta para o estado EVALUATE ACE.

### EVALUATE ACE

Quando neste estado, habilita o processo calculate\_score\_proc para avaliar o caso especial da carta às.

Primeiro, verifica se a carta decodificada é um às e se a nova pontuação é inferior a 11 pontos. Caso afirmativo, soma-se mais 10 pontos à pontuação atual (considera-se que o às tem o valor de 11 pontos) e atribui TRUE para a variável interna do processo v\_ace\_in\_the\_hole (tenho um “às na manga”).

Caso contrário, verifica se possui um “às na manga” (v\_ace\_in\_the\_hole = TRUE) e se a pontuação atual é maior que 21. Se este é o caso, desconsidera que o às que foi sacado em outra rodada tenha o valor 11, mas sim 1 ponto. Para representar este comportamento, subtrai 10 pontos da pontuação atual e atribui FALSE para a variável interna do processo v\_ace\_in\_the\_hole.

No final da avaliação, salta para o estado IDLE e conclui a rodada.

# TESTBENCH

O testbench foi desenvolvido com base em procedimentos definidos em um package **pkg\_tb\_blackjack**. O objetivo foi desenvolver os testes de forma que um teste não interfira em outro e, reaproveitar o máximo possível trechos de código utilizados nos testes.

O pacote apresenta 8 procedimentos de testes:

* Test\_1\_success\_win: Jogo em que o **PLAYER** vence por ter uma pontuação maior que o **DEALER**.
* Test\_2\_player\_lose: Jogo em que o **DEALER** vence por ter uma pontuação maior que o **PLAYER**.
* Test\_3\_tie: Jogo em que ocorre um empate, pois **PLAYER** e **DEALER** terminam o jogo com a mesma pontuação.
* Test\_4\_ace\_only: Jogo em que o deck externo oferece apenas cartas às, realizando um teste de robustez para testar se a entidade **BLACKJACK** não se perde com mais de uma carta às no jogo.
* Test\_5\_player\_passing\_21: Jogo em que o jogador ‘pressiona’ **HIT** até perder o jogo por ultrapassar 21 pontos.
* Test\_6\_dealer\_passing\_21: Jogo em que o **PLAYER** possui uma pontuação inferior ao **DEALER** e ainda assim deve ser declarado vencedor, pois o jogo deve encerrar com **DEALER** tendo uma pontuação que ultrapassa 21 pontos.
* Test\_7\_decode\_all\_cards: Jogo em que a entidade **BLACKJACK** deve receber e decodificar todas os possíveis códigos de cartas, sejam elas válidas ou inválidas.
* Test\_8\_hit\_stay\_press\_test: Jogo em que **PLAYER** ‘pressiona’ simultaneamente **HIT** e **STAY**. Avalia se este caso é ignorado.

Os casos de testes utilizam procedimentos auxiliares que realizam tarefas pequenas, como por exemplo, reiniciar o jogo, avaliar as saídas, simular a compra de uma carta, etc.

Para cada caso de teste existe seu próprio deck externo, garantindo que, caso um teste falhe, o resultado dele não interfira no próximo teste.

Todas as cartas dos casos de teste são definidos dentro da entidade card\_deck\_memory. Os testes, por sua vez, conhecem todas as cartas do jogo, permitindo antecipar qual deve ser a próxima carta e qual a pontuação esperada para cada etapa do teste.

# CONCLUSÃO

Este relatório apresentou a arquitetura e o testbench do trabalho final da disciplina Projeto de Sistemas Integrados 1.

Na etapa inicial de desenvolvimento, o principal ganho da entidade **BLACKJACK** foi criar entidade **DECK\_CONTROLLER** para ser utilizada pelo **PLAYER** e **DEALER**. Além das vantagens em simplificar a arquitetura, quando foram detectados erros nas etapas de verificação, foi extremamente rápido e fácil aplicar as devidas correções.

Quanto ao testbench, foi aplicado um esforço considerável e tempo gasto no desenvolvimento dos procedimentos auxiliares. Depois que o primeiro teste foi concluído, foi muito rápido e simples criar os novos casos de teste. Constatamos que esta metodologia é eficiente para testar entidades que sejam muito complexas ou que tenham muitos requisitos e funcionalidades para serem testadas. Uma das vantagens obtidas foi a de fazer a manutenção dos testes, pois estes utilizam os mesmos procedimentos auxiliares e todos os testes eram corrigidos de uma única vez.