Wilson Cazarré Sousa

Desenvolvimento e implementação de um processador compatível com a Arquitetura 6502 em FPGA

São José dos Campos - Brasil

Abril de 2024

Wilson Cazarré Sousa

Desenvolvimento e implementação de um processador compatível com a Arquitetura 6502 em FPGA

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores.

Docente: Prof. Dr. Tiago de Oliveira

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Abril de 2024

Resumo

Esse trabalho irá apresentar o desenvolvimento de um microprocessador capaz de executar um subconjunto das 6502 desenvolvido pela MOS Technology. O 6502 foi um microprocessador lançado em 1975 e redefiniu o que um computador pessoal podia fazer, sendo usado em muitos dispositivos populares da época como o NES e o Apple I. O desenvolvimento do mesmo será realizado em VHDL e implementado em um FPGA. O trabalho também apresenta os detalhes da arquitetura implementada, bem como seu *datapath*, modos de endereçamento e ciclos de execução.

Palavras-chaves: 6502. NES. FPGA. Verilog. SystemVerilog

Lista de ilustrações

Figura 1 – Arquitetura de von Neumann 11
Figura 2 – Endereçamento imediato 14
Figura 3 – Endereçamento absoluto. Note que o primeiro byte na memória é o menos significativo 15
Figura 4 – Endereçamento absoluto - Deslocado em X (ou Y) 16
Figura 5 – Endereçamento *Zero-Page* 17
Figura 6 – Endereçamento relativo 18
Figura 7 – Endereçamento indireto 19
Figura 8 – Datapath do 6502 implementado 23
Figura 9 – Macro arquitetura do microcomputador 26
Figura 10 – Teste da ULA 27
Figura 11 – Teste do Contador de Programa 27
Figura 12 – Teste do Registrador 28
Figura 13 – Teste da Unidade de Processamento 28
Figura 14 – Sequência de Fibonacci 31

Lista de tabelas

Tabela 1 – Tamanho da instrução por modo de endereçamento 18
Tabela 2 – Matriz de Opcodes por modo de endereçamento 22

Lista de Códigos Fonte

1 Assembly para a sequência de Fibonacci 29

Sumário

1 INTRODUÇÃO 9
1.1 Metodologia 9
1.2 Objetivos 9

2 FUNDAMENTAÇÃO TEÓRICA 11
2.1 Visão geral de um sistema computacional 11
2.1.1 Arquitetura de von Neumann 11
2.2 Microprocessador 6502 12
2.2.1 Arquitetura Original 12
2.2.2 Registrador de Status (SR) 12
2.2.3 Contador de Programa (PC) 13
2.2.4 *Stack Pointer* (SP) 13
2.2.5 Modos de endereçamento 13
2.2.5.1 Endereçamento imediato 13
2.2.5.2 Endereçamento absoluto 14
2.2.5.3 Endereçamento absoluto - Deslocado em X (ou Y) 14
2.2.5.4 Endereçamento *Zero-Page* 14
2.2.5.5 Endereçamento relativo 14
2.2.5.6 Endereçamento indireto 15
2.2.6 Endereçamento *Zero-Page*, deslocado em X (ou Y) 15
2.3 6522 - *Versatile Interface Adapter* 15
2.4 RTL - *Register-Transfer Logic* 16
2.5 Metodologia de Testes 17

3 DESENVOLVIMENTO 21
3.1 Conjunto de instruções 21
3.2 O *datapath* da implementação 21
3.3 Unidades funcionais 21
3.3.1 Registradores de propósito geral (AC, X, Y) 23
3.3.2 Contador de Programa (PCH e PCL) 24

3.3.3 Registrador de Dados (DR) . . . 24
3.3.4 Registrador de Status (P) . . . 24
3.3.5 Unidade de controle . . . 24
3.3.6 Geração de *Clock* (CLK) . . . 24
3.3.7 Registrador do barramento de endereço (ABL e ABH) . . . 25
3.3.8 Unidade Lógica aritmética (ALU) . . . 25
3.4 Código Fonte do Processador . . . 25
3.4.1 Unidade de entrada e saída . . . 25
3.4.2 Arquitetura final do microcomputador . . . 26

4 RESULTADOS OBTIDOS E DISCUSSÕES . . . 27
4.1 Formas de onda . . . 27
4.2 Teste final do processador . . . 29

5 CONSIDERAÇÕES FINAIS . . . 33

REFERÊNCIAS . . . 35

APÊNDICES 37

APÊNDICE A – ALU.SV . . . 39

APÊNDICE B – ASYNC_RAM.SV . . . 41

APÊNDICE C – CLOCK.SV . . . 43

APÊNDICE D – CONTROL_UNIT.SV . . . 45

APÊNDICE E – CPU6502.SV . . . 69

APÊNDICE F – DEV.SV . . . 77

APÊNDICE G – INTERFACE_ADAPTER.SV . . . 81

APÊNDICE H – PROGRAM_COUNTER.SV . . . 85

APÊNDICE I – REGISTER.SV . . . 87

APÊNDICE J – ROM.SV . . . 89

APÊNDICE K – STACK_POINTER.SV . . . 91

APÊNDICE L – STATUS_REGISTER.SV . . . 93

APÊNDICE M – STATUS_REGISTER.SV 95

APÊNDICE N – STATUS_REGISTER.SV 97

APÊNDICE O – STATUS_REGISTER.SV 99

ANEXO A – TEMPLATE PARA TESTBENCHES 103

1 Introdução

Durante a disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores, ofertada no Instituto de Ciência e Tecnologia da UNIFESP, é proposto que os discentes escolham uma arquitetura de processador para realizar sua implementação em um dispositivo FPGA. Esse relatório irá apresentar a fundamentação, bem como todo o processo de desenvolvimento de um sistema computacional baseado no microprocessador 6502.

1.1 Metodologia

O trabalho apresentado será desenvolvido no *software* Quartus©Prime da Intel e implementado na linguagem de descrição de *hardware* SystemVerilog. O circuito será implementado usando a abstração de *Register-Transfer Level* onde o fluxo de dados no circuito é representado como registradores e as unidades de lógica combinacional que determinam seus estados. O projeto então será testado em bancada onde deverá ser capaz de executar qualquer tipo de lógica definida como "computável" (ou seja, ter a mesma funcionalidade de uma Máquina de Turing).

1.2 Objetivos

Geral

Desenvolver uma CPU capaz de executar um subconjunto das instruções da família MCS650X. A implementação deverá ser feita em VHDL e sintetizada pelo *software* Quartus©Prime da Intel.

Específico

- Definir o subconjunto de instruções;
- Desenvolver uma unidade lógica e aritmética;
- Desenvolver os registradores do processador;
- Desenvolver a unidade de controle;
- Integrar os registradores, a unidade de controle e a unidade lógica e aritmética;
- Desenvolver casos testes para o processador;

- Testar o processador.

2 Fundamentação Teórica

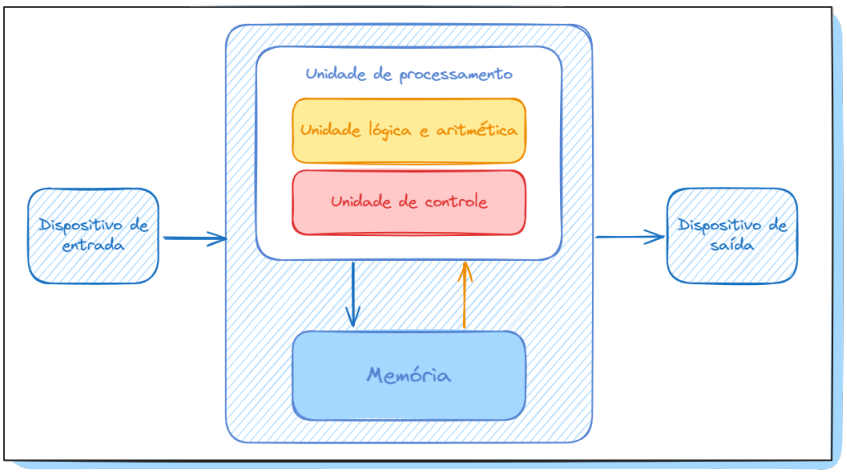
2.1 Visão geral de um sistema computacional

2.1.1 Arquitetura de von Neumann

A Arquitetura de von Neumann foi proposta por um grupo de engenheiros liderados por **Jonh von Neumann** em 1945 (1). O design descrito pelo documento tem como objetivo de ser um caso generalizado para um computador digital e é composto dos seguintes componentes (Figura 1):

1. Uma unidade capaz de executar operações aritméticas ;
2. Uma unidade lógica de controle;
3. Uma memória de "tamanho considerável". Essa memória guarda as instruções e dados do programa.
4. Dispositivos de entrada e saída.

Figura 1 – Arquitetura de von Neumann



Fonte: Autoria própria

Nesse tipo de arquitetura, o processador pode ler apenas uma instrução OU dado por vez. Isso porque ambas as leituras ocorrem por meio do mesmo barramento.

A arquitetura apresentada na seção 2.2 segue exatamente os mesmos princípios apresentados aqui: um único barramento de endereços no qual o processador pode comunicar qual o endereço da informação que está tentando acessar, e um barramento de dados por onde a informação se propaga.

É importante também destacar que ainda que seja possível fisicamente separar as memórias de dados e de programa (o que de fato é algo que será feito) durante esse trabalho, do ponto de vista do processador essa não é uma diferença efetiva. O processador apenas consegue "enxergar" um único barramento de dados e de endereço, não importa quais dispositivos estejam conectados diretamente.

2.2 Microprocessador 6502

O microprocessador 6502 é o segundo membro da família MCS650X. Essa família de microprocessadores de 8 bits foi lançada em 1975 pela *MOS Technology*. Os processadores dessa família apresentam o mesmo conjunto de instruções e modos de endereçamento, com pequenas diferenças em recursos e sua utilização (2). Por conta de sua eficácia e baixo custo, o microprocessador se popularizou rapidamente ao ser usado em diversos sistemas da época como *O Nintendo Entertainment System (NES)*, *Apple II*, *Commodore 64* e muitos outros.

2.2.1 Arquitetura Original

O microprocessador conta com um Barramento de Dados de 8 bits e um Barramento de endereços 16-bits. Qualquer operação que o processador precisa executar normalmente é iniciada colocando o endereço de acesso no Barramento de endereços e posteriormente lendo (ou escrevendo) um valor de 8-bits no Barramento de dados.

Internamente, 3 registradores de propósito geral podem ser usados.

- **Acumulador (A)**: Usado também para armazenar o resultado das operações lógicas e aritméticas;
- ***Index* X e Y**: Ambos os registradores podem ser usados para operações com modos de endereçamento especiais, que serão abordados mais a frente no relatório.

Além dos 3 registradores que podem ser acessados diretamente, o 6502 também possuí alguns registradores usados por funções específicas do processador.

2.2.2 Registrador de Status (SR)

O **registrador de status** é responsável por armazenar *flags* usadas para o controle do fluxo de programa do processador. Elas normalmente são atualizadas durante operações lógicas, aritméticas e de transferência de dados.

- ***Carry (C)***: Indica se a operação gerou um *carry*;

- **Negativo (N)**: Indica se a operação gerou um valor com o bit mais significativo ativo;
- ***Overflow* (V)**: Indica se a operação gerou um...;
- **Zero (Z)**: Indica se a operação gerou o valor zero;
- **Decimal (D)**: Indica se o processador está em modo aritmético decimal BCD;
- **Bloqueio de interrupções (I)**: Indica se o processador está ignorando as requisição de interrupções;
- ***Break* (B)**: Indica se a interrupção atual foi disparada via *software* pela instrução BRK, ao invés de uma interrupção via *hardware*.

2.2.3 Contador de Programa (PC)

O único registrador de 16-bits definido pela arquitetura. Esse registrador é responsável por manter o endereço de memória atualmente acessado pelo processador.

2.2.4 *Stack Pointer* (SP)

O *stack* é uma região de memória destinada para rápido acesso e escrita. A eficácia nessas operações vem do fato de que o processador utiliza o endereço no SP para saber exatamente onde a próxima leitura e escrita vai ocorrer. O registrador é incrementado ou decrementado de acordo após cada operação. O 6502 também utiliza o *stack* para armazenar os endereços de retorno quando subrotinas ou interrupções são executadas.

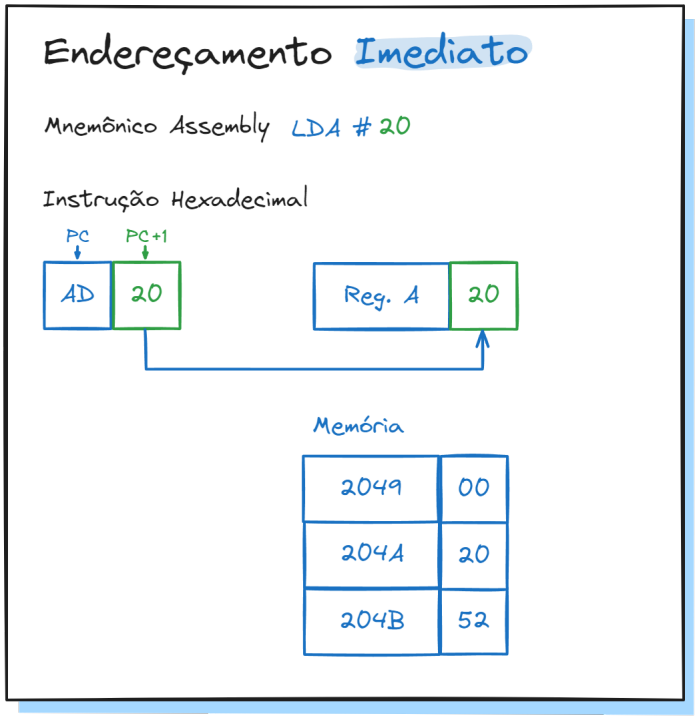
2.2.5 Modos de endereçamento

O 6502 é capaz de endereçar 65.536 bytes de memória. Qualquer operação ou estrutura de dados dentro do processador compartilham esse mesmo espaço de memória. O processador também providencia 13 diferentes métodos de calcular o endereço efetivo de memória na qual a operação vai ser executada (3). Na computação, chamamos esses métodos de **modos de endereçamento** e aqueles disponíveis no 6502 serão descritos aqui.

2.2.5.1 Endereçamento imediato

Nesse tipo de instrução o operando é usado imediatamente após a instrução ter sido lida. Nenhum acesso a memória ou cálculo é realizado (Figura 2). Essa tipo de instrução utiliza 2 bytes de memória.

Figura 2 – Endereçamento imediato



Fonte: Autoria própria

2.2.5.2 Endereçamento absoluto

Nesse tipo de instrução dois bytes são passados além do opcode. O processador usa esses bytes como um endereço de acesso a memória (Figura 3).

2.2.5.3 Endereçamento absoluto - Deslocado em X (ou Y)

Esse modo é uma variação do endereçamento absoluto: dois bytes são buscados da memória e usados como endereço de acesso. A diferença está no fato de que o valor do registrador (X ou Y) é somado ao endereço de acesso. (Figura 4).

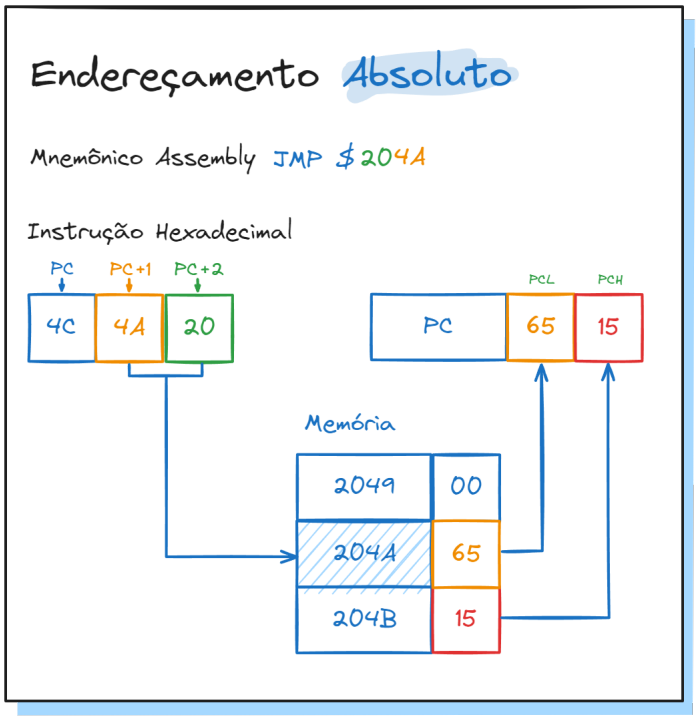
2.2.5.4 Endereçamento *Zero-Page*

Idêntico ao endereçamento absoluto, exceto que apenas um byte é lido da memória (o byte menos significativo). O byte mais significativo é inferido como 0 Figura 5. Logo esse modo de endereçamento sempre retorna um dado localizado na primeira "página" da memória (os primeiros 256 bytes).

2.2.5.5 Endereçamento relativo

O endereçamento relativo é usado especificamente para instruções de *branch*. Nele, o operando contém um valor que será somado ao valor atual do Contador de Programa. Esse valor é então colocado de volta no Contador de programa para que a execução possa

Figura 3 – Endereçamento absoluto. Note que o primeiro byte na memória é o menos significativo



Fonte: Autoria própria

continuar a partir daí. É importante destacar que o byte de deslocamento passado nessa instrução pode possuir sinal positivo ou negativo. Isso significa pular para um endereço posterior ou anterior contando que o valor de deslocamento esteja entre -128 e 127.

2.2.5.6 Endereçamento indireto

Nesse tipo de endereçamento, o processador busca o endereço efetivo no endereço que foi passado pelo operando (Figura 7).

2.2.6 Endereçamento *Zero-Page*, deslocado em X (ou Y)

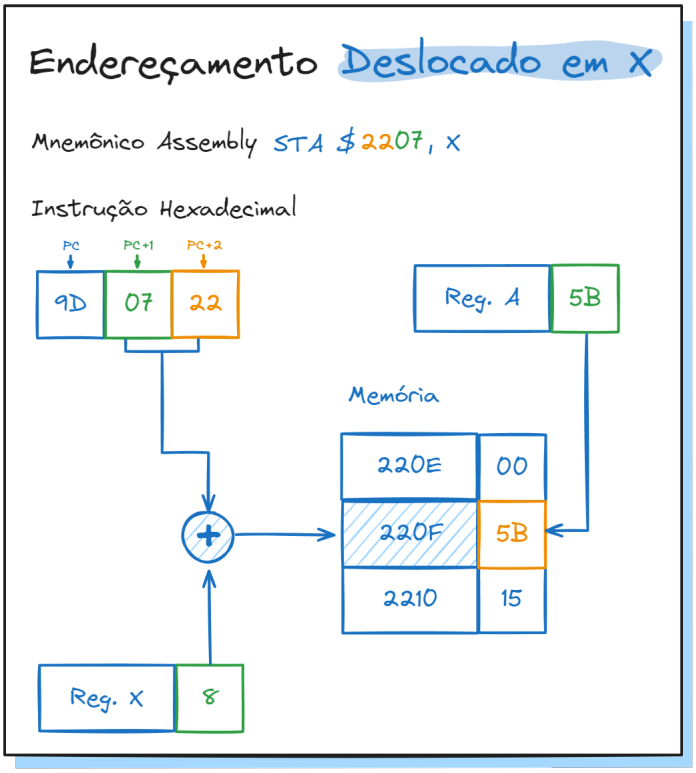
Idêntico ao Endereçamento Absoluto deslocado em X (ou Y), exceto que o endereço de acesso está sempre nos primeiros 256 bytes do espaço de memória (Figura 4).

2.3 6522 - *Versatile Interface Adapter*

O 6522 é uma dispositivo projetado para uso com os microprocessadores da família 65xx (4). Dentre as suas funções estão inclusos:

- 16 Pinos GPIO bidirecionais (disponíveis por meio de dois registradores de 8 bits)

Figura 4 – Endereçamento absoluto - Deslocado em X (ou Y)



Fonte: Autoria própria

- Timer/Contador de programável de 16-bits
- Registrador de deslocamento de 8 bits

Os pinos de GPIO são acessados por meio dos registradores `PORTA` e `PORTB`, e a direção (entrada ou saída) de cada um dos bits nesses registradores pode ser controlada por meio dos registradores de direção de dados `DDRA` e `DDRB`.

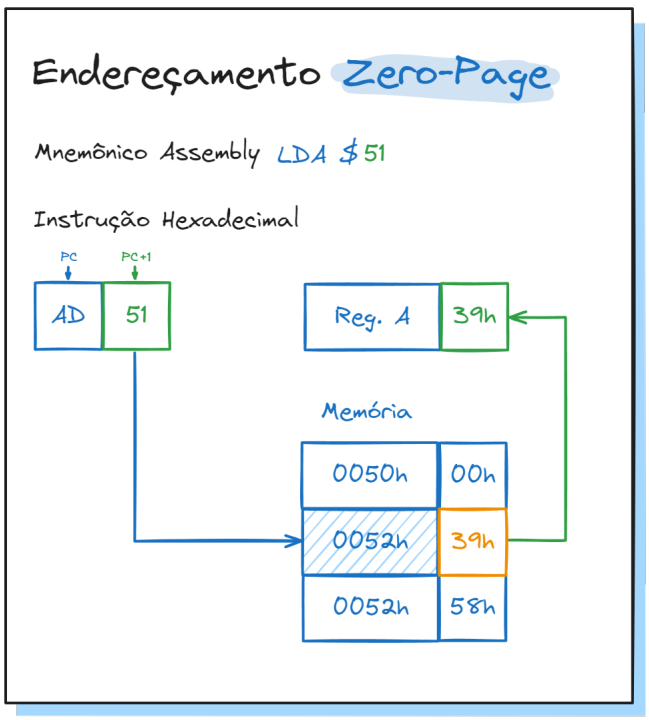
2.4 RTL - *Register-Transfer Logic*

Quando tratamos do design de circuitos digitais complexos, é comum abstrairmos diferentes níveis do design com a intenção de tornar esses problemas mais simples de serem resolvidos.

3 níveis diferentes de abstração são definidos por 5 na construção de circuitos digitais:

1. ***Transistor Level***: Conectar transistores para construir componentes lógicos.
2. ***Logic Level***: Utilizar-se de Portas Lógicas como bloco principal de construção para desenvolver circuitos combinacionais.

Figura 5 – Endereçamento *Zero-Page*



Fonte: Autoria própria

3. ***Register-transfer Level***: Conectar uma rede de registradores e construir blocos que definem a lógica de transferência de estado entre esses registradores.

De maneira geral, no *Register-Transfer Level Design* (ou Design RTL) cada bloco do design deve desempenhar uma (e apenas uma) de duas possíveis funções:

1. **Lógica Combinacional**: São os blocos responsáveis pela computação do próximo estado. De maneira geral, esses blocos devem ser determinísticos e sempre apresentar a mesma saída para uma determinada entrada.

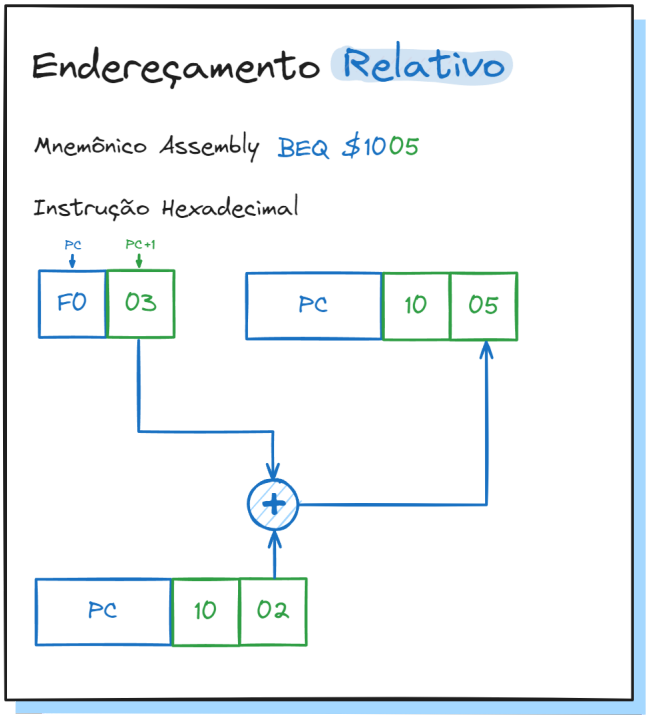
2. **Lógica Sequencial**: São blocos responsáveis por guardar e propagar o estado computado pelos blocos combinacionais de maneira síncrona.

2.5 Metodologia de Testes

Para garantir o funcionamento das partes individuais do processador, a ferramenta de simulação digital ModelSim da Intel foi utilizada.

Um template para os testbenches está disponível no Apêndice A

Figura 6 – Endereçamento relativo



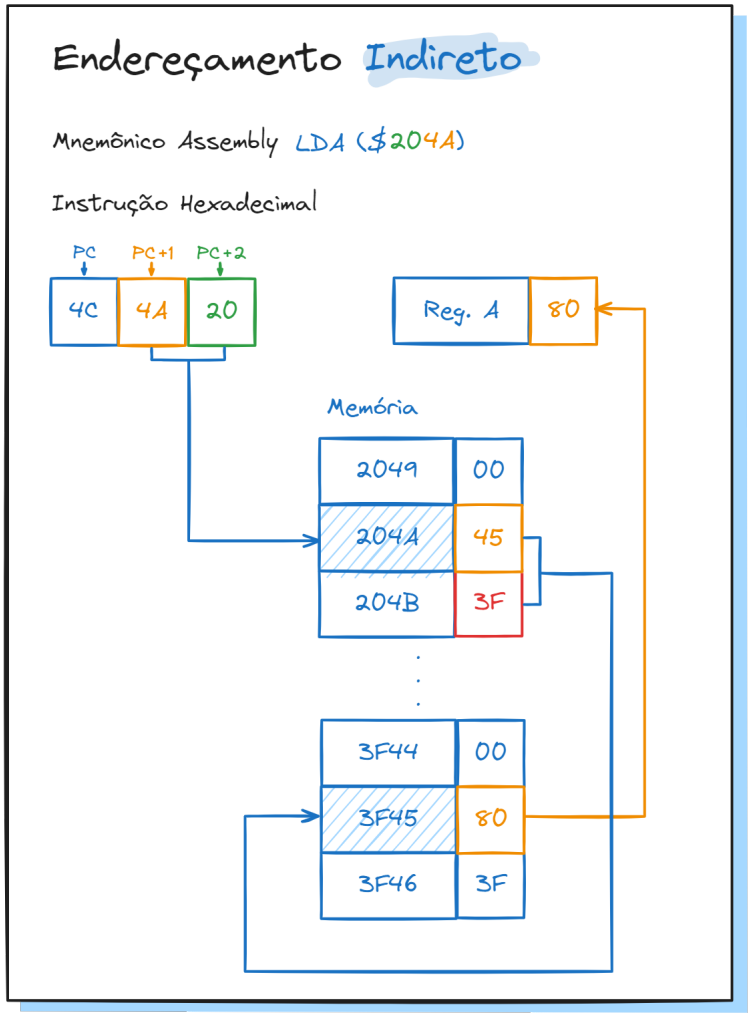
Fonte: Autoria própria

Tabela 1 – Tamanho da instrução por modo de endereçamento

Modo de endereçamento	**Tamanho em bytes**
Acumulador (A)	1
Absoluto (abs)	3
Absoluto, deslocado em X (abs, x)	3
Absoluto, deslocado em Y (abs, y)	3
Imediato (#)	2
Implícito (impl)	1
Indireto (ind)	3
Indireto, deslocado em X (X, ind)	2
Indireto, deslocado em Y (ind, Y)	2
Relativo (rel)	2
Zero-Page (zpg)	2
Zero-Page, deslocado em X (zpg, x)	2
Zero-Page, deslocado em Y (zpg, y)	2

Fonte: Autoria Própria

Figura 7 – Endereçamento indireto



Fonte: Autoria própria

3 Desenvolvimento

O desenvolvimento da CPU se deu em algumas etapas. Primeiramente o conjunto de instrução foi definido como um subconjunto da família MCS650X original. Depois disso foram escolhidos alguns modos de endereçamento e um *datapath* foi definido.

3.1 Conjunto de instruções

O conjunto de instruções apresentado na Tabela 2 é apenas um subconjunto da família MCS650X original. Os mesmos *opcodes* da arquitetura original serão mantidos aqui (6).

3.2 O *datapath* da implementação

A Figura 8 mostra o *datapath* que será implementado durante esse trabalho.

O processador possuí 3 registradores de propósito geral e é capaz de manipular 8-bits por ciclo de clock, por consequência toda instrução no 6502 leva mais de 1 ciclo para ser executada, considerando que o opcode consiste sempre de 8-bits.

A Figura 8 também divide os componentes em dois grupos principais:

- **Registradores externos**: São os registradores que o programador tem consciência que estão lá e pode, por meio do conjunto de instruções, interagir com eles de maneira direta ou indireta.

- **Microarquitetura interna**: São os componentes internos que não são diretamente definidos pela arquitetura MCS650X, são invisíveis do ponto de vista do programador mas são vitais para o funcionamento do processador.

O processador possuí uma Barramento de Endereços de 16-bits, isso significa que ele se comunicar com até 65,536 diferente endereços. Esse diferentes endereços serão mencionados daqui em diante como o Espaço de Memória (EM) do 6502.

3.3 Unidades funcionais

Essa seção apresenta uma breve descrição de cada componente no *datapath*.

Tabela 2 – Matriz de Opcodes por modo de endereçamento

Instrução	**Operação**	**#**	**a**	**a, x**	**a,y**	**zp**	**zp, x**	**s**	**impl**	**rel**
ADC	`A <= A + M + C`	69	6d	7d	79	65				
AND	`A <= A AND M`	29	2d	3d	39	25				
BCC	`Branch If C == 0`									90
BCC	`Branch If C == 1`									b0
BEQ	`Branch If Z == 1`									f0
BMI	`Branch If N == 1`									30
BNQ	`Branch If Z == 0`									d0
BPL	`Branch If N == 0`									10
BVC	`Branch If V == 0`									50
BVS	`Branch If V == 1`									70
CLC	`C <= 0`					18				
CMP	`A - M`		c9	cd	dd		c5	d9		
CPX	`X - M`		e0	ec			e4			
CPY	`Y - M`		c0	e4			c4			
EOR	`A <= A XOR M`	49	4d	5d	59	45	c4			
INX	`Increment X`								e8	
INY	`Increment Y`								c8	
JMP	`Jump M`		4c							
LDA	`A <= M`	a9	ad	bd	b9	a5				
LDX	`X <= M`	a2	ae	be		a6				
LDY	`Y <= M`	a0	ac	bc		a4				
NOP									ea	
ORA	`A <= A OR M`	09	0d	1d	19	05				
PHA	`Push A, S <= S - 1`								48	
PHX	`Push X, S <= S - 1`								da	
PHY	`Push Y, S <= S - 1`								5a	
PLA	`Pull A, S <= S + 1`								68	
PLX	`Pull X, S <= S + 1`								fa	
PLY	`Pull Y, S <= S + 1`								7a	
SBC	`A <= A - M + C`	e9	ed	fd	f9	e5				
SEC	`C <= 1`								38	
STA	`M <= A`		8d	9d	99	85				
STX	`M <= X`		8e			86				
STY	`M <= Y`		8c			84				
TAX	`X <= A`								aa	
TAY	`Y <= A`								a8	
TSX	`X <= S`								ba	
TXA	`A <= X`								8a	
TXS	`S <= X`								9a	
TYA	`A <= Y`								98	

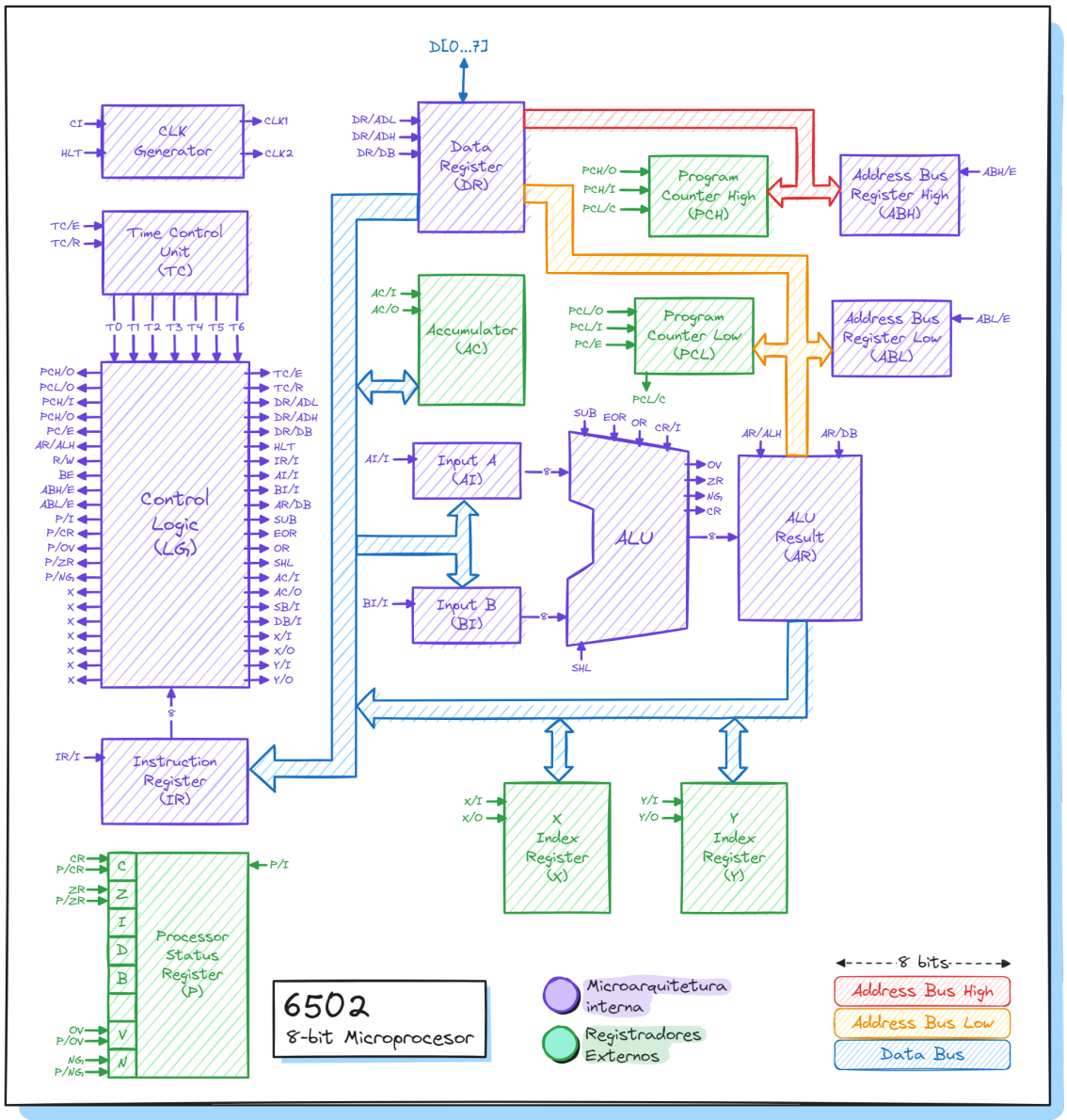
Fonte: Autoria Própria

3.3.1 Registradores de propósito geral (AC, X, Y)

Esses registradores de 8-bits que podem ser acessados diretamente utilizando suas respectivas instruções de *Load* e *Store*. Além disso, eles desempenham funções específicas no processador:

- **Acumulador (AC)**: Toda operação lógica e aritmética tem como base o valor armazenado nesse registrador, além disso o resultado dessas operações também é diretamente armazenado aqui.
- **X e Y**: Esses registradores armazenam o valor de deslocamento das instruções com os modos de endereçamento deslocados.

Figura 8 – Datapath do 6502 implementado



3.3.2 Contador de Programa (PCH e PCL)

O contador de programa é usado para endereçar o espaço de 16-bits de memória disponível para o processador. Por conta do processador conseguir manipular apenas 8-bits por vez, utiliza-se 2 registradores de 8 bits para armazenar o endereço completo. O programador pode manipular seu valor por meio das instruções de *jump* e *branch*.

3.3.3 Registrador de Dados (DR)

O registrador de dados é responsável por controlar a entrada de informações no processador e distribuir para um dos 3 barramentos disponíveis.

3.3.4 Registrador de Status (P)

Essa implementação difere da apresentada em 2.2.2. As *flags* I, D e B não serão implementadas. Os outros valores serão controlados pelo resultado de operações aritméticas, *loads* e *stores*.

3.3.5 Unidade de controle

A unidade de controle é responsável por enviar todos os sinais de controle necessários para a execução de uma instrução em particular. Ela é composta por 3 partes:

1. **Registrador de instrução (IR)**: Armazena o *opcode* da instrução atualmente sendo executada.

2. **Unidade de tempo (TCU)**: No começo de toda instrução, seu valor é definido como T0, na **descida** de cada ciclo de clock seu valor é incrementado (T1, T2, T3, etc).

3. **Lógica de Controle (LG)**: Responsável por definir todos os sinais de controle baseado na instrução que está sendo executado atualmente (armazenada no IR) e qual ciclo o processador se encontra dentro dessa instrução (armazenado no TCU).

3.3.6 Geração de *Clock* (CLK)

Essa unidade é responsável por gerar o sinal de clock do processador. O clock de entrada (CI) é dividido em 2 clocks espelhados. Diferentes componentes podem executar operações em um dos dois ciclos de *clocks*.

3.3.7 Registrador do barramento de endereço (ABL e ABH)

O endereço que está sendo acessado atualmente pelo processador ficará armazenado nesse registrador. Como o endereço é de 16-bits, 2 registradores de 8-bits serão usados.

3.3.8 Unidade Lógica aritmética (ALU)

A ALU é o circuito combinacional responsável por executar todas as operações lógicas e aritméticas do processador. Além disso, registradores auxiliares são acoplados a unidade: AI e BI armazenam os valores de entrada e AR armazena o valor de saída.

3.4 Código Fonte do Processador

O código em sua totalidade está disponível no Github do autor.

Uma breve descrição dos módulos será apresentada aqui.

Com o objetivo de evitar erros redundantes e melhorar consistência ao longo do código, arquivos de *enums* foram feitos para nomear diversas constantes:

- Apêndice O Define os opcodes para o conjunto de instruções, note que cada nome também é acompanhado do seu respectivo modo de endereçamento;
- Apêndice M Define todos os valores que podem ser colocados nos barramentos;
- Apêndice N Define todos os sinais de controle usados no microprocessador.

A unidade de integração apresentada no Apêndice E é o ponto de entrada do processador e descreve como os módulos se comunicam entre si. Note que todas as possíveis entradas para um determinado barramento são agrupadas em uma *array* para que possam ser multiplexadas por um sinal de controle.

A unidade de controle (Apêndice D) é uma máquina de estados que define uma sub tarefa para cada operação e modo de endereçamento. Uma operação e um modo de endereçamento são então combinados para formar um *opcode* completo.

3.4.1 Unidade de entrada e saída

Até o momento tratamos apenas do microprocessador que em seu nenhum momento aborda como a entrada e saída deve ser feita. Para isso, uma versão com recursos limitados do 6522 Apêndice G foi implementada em *SystemVerilog* para que pudesse ser interfaceada com o microprocessador. Essa placa providencia o 6502 com 16 pinos de GPIO que podem ser reconfigurados a qualquer momento do programa. É importante destacar que apenas os registradores `PORTA` e `PORTB` em conjunto com os registradores de direção de dados

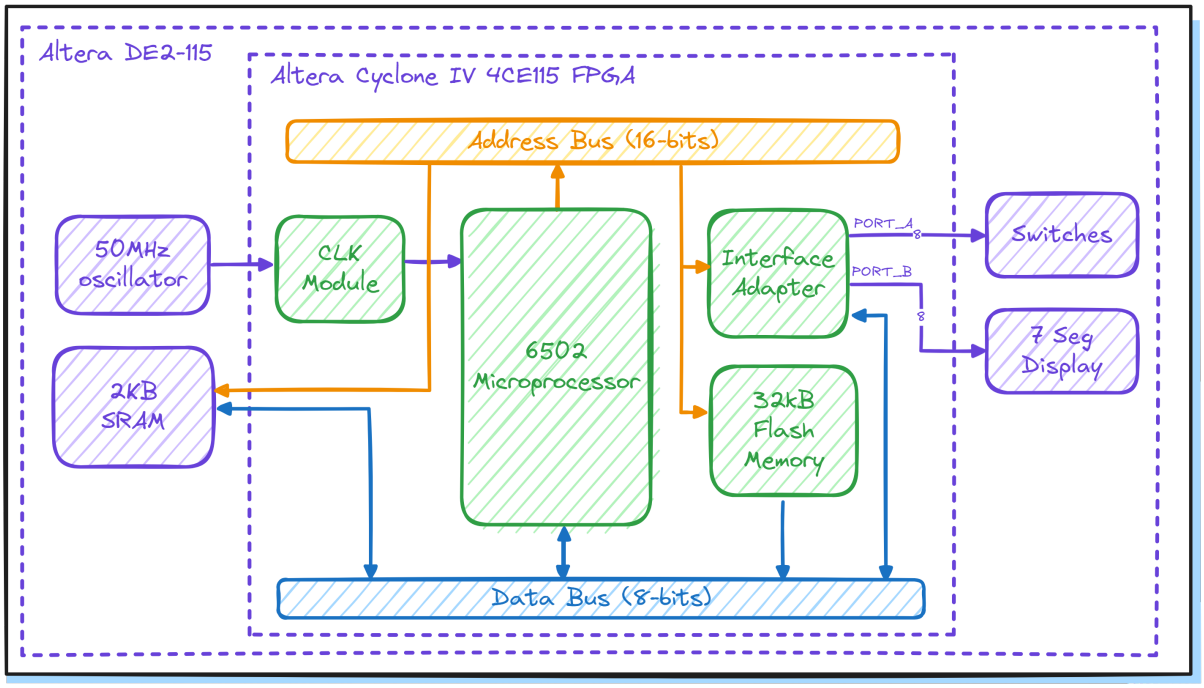
`DDRA` e `DDRB` foram implementados nessa versão. Os contadores e temporizadores foram propositalmente deixados de lado nessa implementação.

3.4.2 Arquitetura final do microcomputador

A Figura 9 apresenta como as conexões entre o microprocessador (6502), adaptador de interface (6522) e demais periféricos foram realizadas. Os módulos em verde foram implementados em *SystemVerilog* sintetizável enquanto os módulos em roxo são dispositivos físicos no kit de desenvolvimento FPGA.

O código para essa implementação pode ser visto no Apêndice F.

Figura 9 – Macro arquitetura do microcomputador

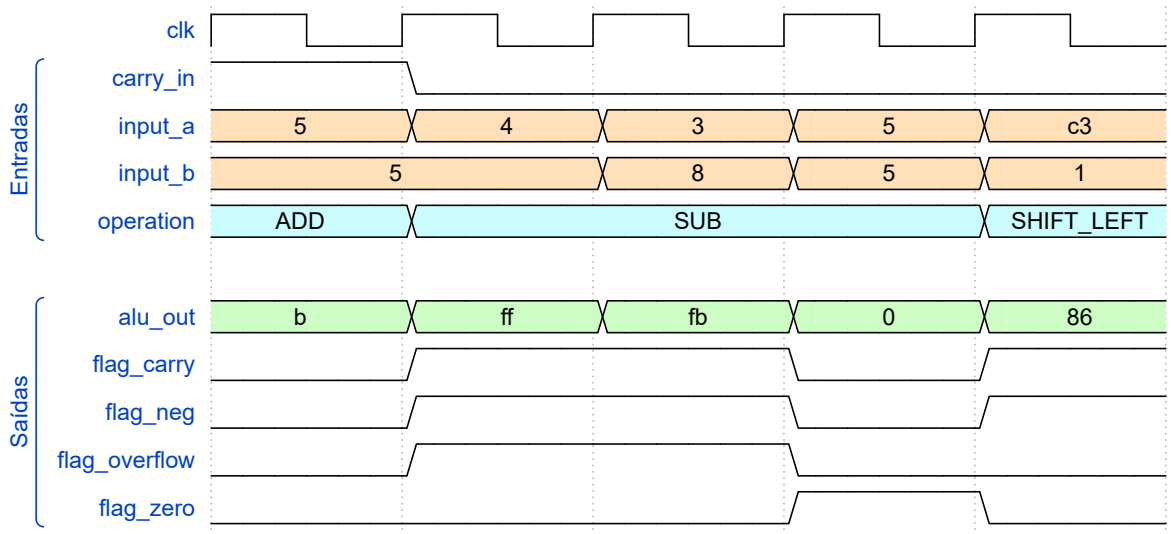


Fonte: Autoria Própria

4 Resultados Obtidos e Discussões

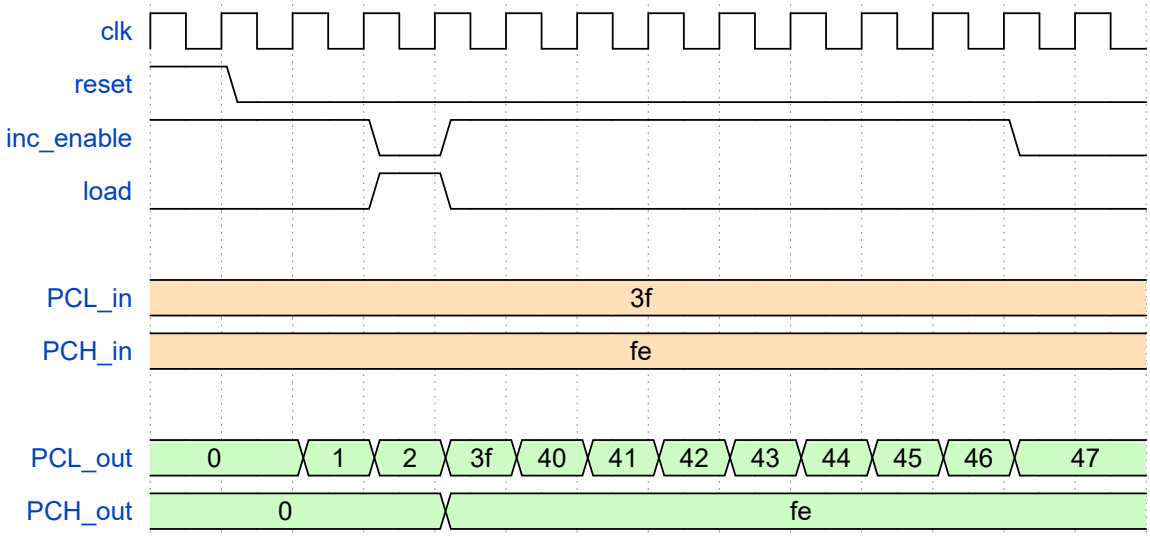
4.1 Formas de onda

Figura 10 – Teste da ULA



Fonte: Autoria Própria

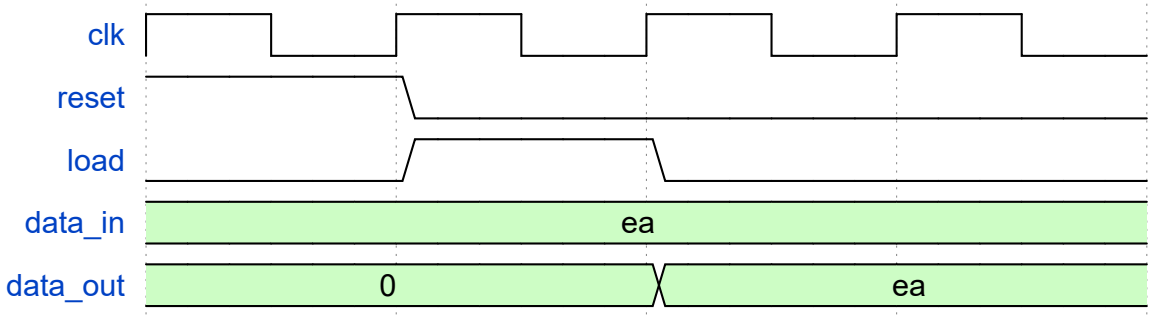
Figura 11 – Teste do Contador de Programa



Fonte: Autoria Própria

O teste da Figura 13 é um teste de integração, onde podemos ver o funcionamento básico do processador. Inicialmente colocamos o opcode da operação `NOP` (`hea`) no barramento de dados (`data_in`). Podemos ver que o processador realiza os ciclos de *Fetch*

Figura 12 – Teste do Registrador



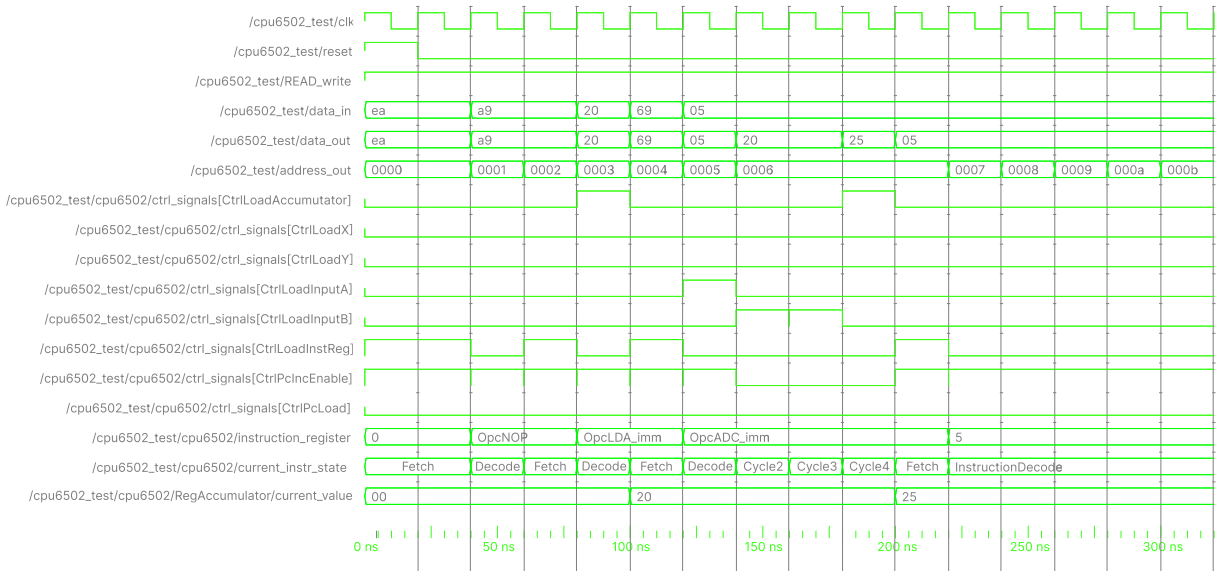
Fonte: Autoria Própria

e *Decode* e depois volta ao ciclo de *Fetch* já que a operação `NOP` não precisa de nenhum ciclo extra de execução.

Em seguida, o código `ha9` é lido do barramento de dados, o que corresponde a instrução `LDA` com imediato. O valor `h20` é lido e observamos que ele é colocado no Registrador A (`RegAccumulator/current_value`).

Posteriormente, o opcode `h69` é lido, que corresponde a instrução `ADC` com imediato. Essa instrução deve ler um valor imediato e somar com o valor atual do Registrador A. Podemos ver o valor final do Registrador A atualizado como `h25` no final da simulação. Esse era o valor esperado já que carregamos o registrador com `h20` e depois somamos mais `h5` ao seu conteúdo.

Figura 13 – Teste da Unidade de Processamento



Fonte: Autoria Própria

4.2 Teste final do processador

Para provar que o processador é capaz de executar qualquer operação computável foi proposto que o mesmo executasse um programa que lesse um valor inserido pelo usuário (por meio das chaves no kit de desenvolvimento) e calculasse a sequência de Fibonacci até aquele valor. Tal programa é descrito em Código 1. Note que o carácter # não indica um comentário e sim um valor imediato.

Como o processador foi construído utilizando os mesmos opcodes do 6502 original, *assemblers* já existentes podem ser utilizados para gerar o código de máquina necessário.

Código 1 – Assembly para a sequência de Fibonacci

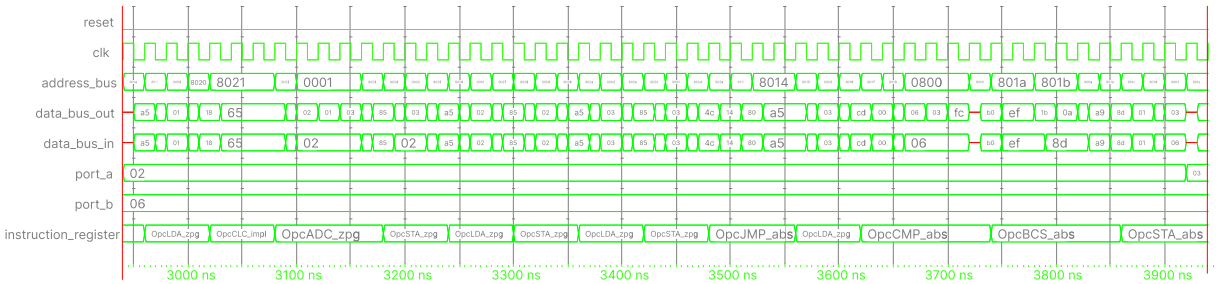
```
; Registradores do adaptador de interface
DDRA  = $0803
DDRB  = $0802
PORTA = $0801
PORTB = $0800

; Endereços para variáveis na RAM
N1 = $00
N2 = $01
R  = $02

  ; Usamos a diretriz .org para indicar para o assembler onde
  ; nosso programa começa no espaço de memória
  .org $8000

  ; Define todos os bits de porta B como Entrada
  lda #$00 ; 0 - Entrada / 1 - Saída
  sta DDRB

  ; Define todos os bits de porta A como saída
  lda #$ff
  sta DDRA


restart:
  ; Inicializa N1 = 0, N2 = 1, R = 1
  lda #$00
  sta N1
  lda #$01
```

```
  sta N2
  sta R
loop:
  ; Esse loop continua executando enquando (R - PORTB) > 0
  ; Do contrário ele pula de volta para restart, para recalcular a
  ↪ sequência
  lda R
  cmp PORTB
  bpl restart
  ; Exibe o valor R na PORTA, efetivamente mostrando a saída para o
  ↪ usuário
  sta PORTA

  ; R = N1 + N2
  lda N1
  clc
  adc N2
  sta R

  ; N1 = N2
  lda N2
  sta N1

  ; N2 = R
  lda R
  sta N2

  jmp loop
```

A figura Figura 14 exibe o processo de cálculo para um valor da sequência. De acordo com o que foi descrito no código *assembly*, `port_b` é usado como valor de entrada e define até qual valor a sequência será calculada. `port_a` é a saída conectada ao display de 7 segmentos e ele mostra o valor atual da sequência. Podemos ver que no começo da sequência o valor mostrado é 2, e na última divisão de tempo apresentada o valor muda para 3. Também é possível notar a natureza multi ciclo do processador se observarmos que o valor no `instruction_register` se mantém constante por alguns ciclos de clock

antes do processador seguir para uma próxima instrução.

Figura 14 – Sequência de Fibonacci



5 Considerações Finais

O microprocessador apresentado nesse trabalho é capaz de executar 86 dos 149 opcodes disponíveis no 6502 original. Isso já é o suficiente para o sistema ser completo em termos de Turing, o que significa que o mesmo é capaz de realizar qualquer operação considerado computável. Notoriamente alguns recursos importantes estão absentes:

- Interrupções de hardware e software
- Sequência de *reset* onde o processador busca o ponto de entrada do programa em um vetor predefinido.
- Subrotinas via instruções `JSR` (*Jump to Subroutine*) e `RTS` (*Return from Subroutine*)
- Instruções lógicas para rotacionar e deslocar bits (`ROL` , `ROR` , `LSR`)

Esses items serão implementados em trabalhos futuros pois fogem do escopo necessário para conclusão dessa disciplina.

Referências

1 NEUMANN, J. von. *Introduction to "The First Draft Report on the EDVAC"*. 1945. Citado na página 11.

2 TECHNOLOGY, I. M. *MCS6500 Microcomputer Family Hardware Manual*. 2nd edition. ed. Norristown, PA: MOS Technology, INC, 1976. Citado na página 12.

3 CENTER, I. T. W. D. *W65C02S 8-bit Microprocessor Datasheet*. Vienna, Austria, 2018. Citado na página 13.

4 CENTER, I. T. W. D. *W65C22 Versatile Interface Adapter (VIA) Datasheet*. Vienna, Austria, 2010. Citado na página 15.

5 VAHID, F. *Digital Design with RTL Design, VHDL, and Verilog*. 2nd edition. ed. Waltham/MA, EUA: John Wiley & Sons, Inc, 2011. Citado na página 16.

6 LANDSTEINER, M. N. *6502 Instruction Set*. Disponível em: <https://www.masswerk.at/6502/6502_instruction_set.html>. Citado na página 21.

Apêndices

APÊNDICE A – alu.sv

```
module alu (
    input wire                          carry_in,
    input wire                  [7:0]   input_a,
    input wire                  [7:0]   input_b,
    input wire                          invert_b,
    input control_signals::alu_op_t     operation,

    output wire [7:0] alu_out,
    output wire       overflow_out,
    output wire       zero_out,
    output wire       negative_out,
    output wire       carry_out
);

  logic [8:0] result;
  logic [7:0] effective_b;
  assign effective_b = invert_b ? ~input_b : input_b;

  assign alu_out = result[7:0];

  assign carry_out = result[8];
  assign negative_out = result[7];
  assign zero_out = ~|alu_out;
  assign overflow_out = (~input_a[7] & ~input_b[7] & result[7]) |
                        (input_a[7] & input_b[7] & ~result[7]) ;

  always_comb begin
    case (operation)
      control_signals::ALU_ADD: begin
        result = input_a + effective_b + carry_in;
      end
      control_signals::ALU_AND: begin
        result = input_a & input_b;
      end
      control_signals::ALU_OR: begin
```

```
        result = input_a | input_b;
      end
      control_signals::ALU_XOR: begin
        result = input_a ^ input_b;
      end
      control_signals::ALU_SHIFT_LEFT: begin
        result = (input_a << 1) + carry_in;
      end
      default: begin
        result = 8'b0;
      end
    endcase
  end

endmodule
```

APÊNDICE B – async_ram.sv

```
module async_ram #(
    depth = 8
) (
    input  logic [11:0] address,
    input  logic [ 7:0] data_in,
    output logic [ 7:0] data_out,

    input wrt_en,
    input out_en,
    input chip_en
);

  logic [7:0] ram[2**depth];

  always @(wrt_en, address, data_in, chip_en, out_en) begin
    if (chip_en) begin
      if (wrt_en) begin
        ram[address] <= data_in;
      end
      if (out_en) begin
        data_out <= ram[address];
      end
    end
  end
endmodule
```

APÊNDICE C – clock.sv

```
module clock (
    input clk_in,
    output reg clk_out
);
  reg [27:0] counter = 28'd0;
  parameter DIVISOR = 28'd20_000_00;
  // parameter DIVISOR = 28'd1;
  // The frequency of the output clk_out
  //  = The frequency of the input clk_in divided by DIVISOR
  // For example: Fclk_in = 50Mhz, if you want to get 1Hz signal to
  ↪  blink LEDs
  // You will modify the DIVISOR parameter value to 28'd50.000.000
  // Then the frequency of the output clk_out = 50Mhz/50.000.000 = 1Hz
  always @(posedge clk_in) begin
    counter <= counter + 28'd1;
    if (counter >= (DIVISOR - 1)) counter <= 28'd0;
    clk_out <= (counter < DIVISOR / 2) ? 1'b1 : 1'b0;
  end
endmodule
```

APÊNDICE D – control_unit.sv

```
module control_unit (
    input logic status_flags[8],
    input logic [7:0] data_in_latch,
    input instruction_set::opcode_t current_opcode,
    input clk,
    input reset,
    input alu_carry,

    output logic ctrl_signals[control_signals::CtrlSignalEndMarker],
    output control_signals::alu_op_t alu_op,
    output bus_sources::data_bus_source_t current_data_bus_input,
    output bus_sources::address_low_bus_source_t
    ↪  current_address_low_bus_input,
    output bus_sources::address_high_bus_source_t
    ↪  current_address_high_bus_input
);
  typedef enum logic [31:0] {
    InstructionFetch,
    InstructionDecode,
    InstructionMem1,
    InstructionMem2,
    InstructionMem3,
    InstructionMem4,
    InstructionExec1,
    InstructionExec2,
    InstructionExec3,
    InstructionExec4,
    InstructionExec5,
    InstructionInvalid,
    InstructionStateEndMarker
  } instruction_state_t;

  instruction_state_t current_instr_state, next_instr_state;
  instruction_set::address_mode_t current_addr_mode, next_addr_mode;
  logic negative_data_in;
```

```
always_ff @(posedge clk) begin
  if (reset) begin
    current_instr_state <= InstructionFetch;
  end else begin
    negative_data_in <= data_in_latch[7];
    current_instr_state <= next_instr_state;
    current_addr_mode <= next_addr_mode;
  end
end

always_comb begin
  ctrl_signals = '{default: '0};

  current_data_bus_input = bus_sources::DataBusSrcDataIn;
  current_address_low_bus_input = bus_sources::AddressLowSrcPcLow;
  current_address_high_bus_input = bus_sources::AddressHighSrcPcHigh;

  next_instr_state = InstructionInvalid;
  next_addr_mode = instruction_set::AddrModeImpl;
  alu_op = control_signals::ALU_ADD;

  case (current_instr_state)
    InstructionFetch: begin
      current_data_bus_input = bus_sources::DataBusSrcDataIn;
      current_address_low_bus_input = bus_sources::AddressLowSrcPcLow;
      current_address_high_bus_input =
      ↪  bus_sources::AddressHighSrcPcHigh;

      next_instr_state = InstructionDecode;
      ctrl_signals[control_signals::CtrlLoadInstReg] = 1;
      ctrl_signals[control_signals::CtrlIncEnablePc] = 1;
    end
    InstructionDecode: begin
      case (current_opcode)
        instruction_set::OpcADC_imm:  imm_addr_mode();
        instruction_set::OpcADC_abs:  abs_addr_mode();
        instruction_set::OpcADC_absx:
        ↪  absx_addr_mode(bus_sources::DataBusSrcRegX);
```

```
instruction_set::OpcADC_absy:
↪  absx_addr_mode(bus_sources::DataBusSrcRegY);
instruction_set::OpcADC_zpg:  zpg_addr_mode();

instruction_set::OpcAND_imm:  imm_addr_mode();
instruction_set::OpcAND_abs:  abs_addr_mode();
instruction_set::OpcAND_absx:
↪  absx_addr_mode(bus_sources::DataBusSrcRegX);
instruction_set::OpcAND_absy:
↪  absx_addr_mode(bus_sources::DataBusSrcRegY);
instruction_set::OpcAND_zpg:  zpg_addr_mode();

instruction_set::OpcBCC_abs: imm_addr_mode();
instruction_set::OpcBCS_abs: imm_addr_mode();
instruction_set::OpcBEQ_abs: imm_addr_mode();
instruction_set::OpcBMI_abs: imm_addr_mode();
instruction_set::OpcBNE_abs: imm_addr_mode();
instruction_set::OpcBPL_abs: imm_addr_mode();
instruction_set::OpcBVC_abs: imm_addr_mode();
instruction_set::OpcBVS_abs: imm_addr_mode();

instruction_set::OpcCLC_impl: impl_addr_mode();

instruction_set::OpcCMP_imm:  imm_addr_mode();
instruction_set::OpcCMP_abs:  abs_addr_mode();
instruction_set::OpcCMP_absx:
↪  absx_addr_mode(bus_sources::DataBusSrcRegX);
instruction_set::OpcCMP_absy:
↪  absx_addr_mode(bus_sources::DataBusSrcRegY);
instruction_set::OpcCMP_zpg:  zpg_addr_mode();

instruction_set::OpcCPX_imm: imm_addr_mode();
instruction_set::OpcCPX_abs: abs_addr_mode();
instruction_set::OpcCPX_zpg: zpg_addr_mode();

instruction_set::OpcCPY_imm: imm_addr_mode();
instruction_set::OpcCPY_abs: abs_addr_mode();
instruction_set::OpcCPY_zpg: zpg_addr_mode();
```

```
instruction_set::OpcEOR_imm:  imm_addr_mode();
instruction_set::OpcEOR_abs:  abs_addr_mode();
instruction_set::OpcEOR_absx:
↪  absx_addr_mode(bus_sources::DataBusSrcRegX);
instruction_set::OpcEOR_absy:
↪  absx_addr_mode(bus_sources::DataBusSrcRegY);
instruction_set::OpcEOR_zpg:  zpg_addr_mode();

instruction_set::OpcINX_impl: impl_addr_mode();
instruction_set::OpcINY_impl: impl_addr_mode();

instruction_set::OpcJMP_abs: abs_addr_mode();

instruction_set::OpcLDA_imm:  imm_addr_mode();
instruction_set::OpcLDA_abs:  abs_addr_mode();
instruction_set::OpcLDA_absx:
↪  absx_addr_mode(bus_sources::DataBusSrcRegX);
instruction_set::OpcLDA_absy:
↪  absx_addr_mode(bus_sources::DataBusSrcRegY);
instruction_set::OpcLDA_zpg:  zpg_addr_mode();

instruction_set::OpcLDX_imm:  imm_addr_mode();
instruction_set::OpcLDX_abs:  abs_addr_mode();
instruction_set::OpcLDX_absy:
↪  absx_addr_mode(bus_sources::DataBusSrcRegY);
instruction_set::OpcLDX_zpg:  zpg_addr_mode();

instruction_set::OpcLDY_imm:  imm_addr_mode();
instruction_set::OpcLDY_abs:  abs_addr_mode();
instruction_set::OpcLDY_absx:
↪  absx_addr_mode(bus_sources::DataBusSrcRegX);
instruction_set::OpcLDY_zpg:  zpg_addr_mode();

instruction_set::OpcNOP_impl: next_instr_state =
↪  InstructionFetch;

instruction_set::OpcORA_imm:  imm_addr_mode();
instruction_set::OpcORA_abs:  abs_addr_mode();
```

```
instruction_set::OpcORA_absx:
↪  absx_addr_mode(bus_sources::DataBusSrcRegX);
instruction_set::OpcORA_absy:
↪  absx_addr_mode(bus_sources::DataBusSrcRegY);
instruction_set::OpcORA_zpg:  zpg_addr_mode();

instruction_set::OpcPHA_impl: impl_addr_mode();
instruction_set::OpcPHX_impl: impl_addr_mode();
instruction_set::OpcPHY_impl: impl_addr_mode();

instruction_set::OpcPLA_impl: impl_addr_mode();
instruction_set::OpcPLX_impl: impl_addr_mode();
instruction_set::OpcPLY_impl: impl_addr_mode();

instruction_set::OpcSBC_imm:  imm_addr_mode();
instruction_set::OpcSBC_abs:  abs_addr_mode();
instruction_set::OpcSBC_absx:
↪  absx_addr_mode(bus_sources::DataBusSrcRegX);
instruction_set::OpcSBC_absy:
↪  absx_addr_mode(bus_sources::DataBusSrcRegY);
instruction_set::OpcSBC_zpg:  zpg_addr_mode();

instruction_set::OpcSEC_impl: impl_addr_mode();

instruction_set::OpcSTA_abs:  abs_addr_mode();
instruction_set::OpcSTA_absx:
↪  absx_addr_mode(bus_sources::DataBusSrcRegX);
instruction_set::OpcSTA_absy:
↪  absx_addr_mode(bus_sources::DataBusSrcRegY);
instruction_set::OpcSTA_zpg:  zpg_addr_mode();

instruction_set::OpcSTX_abs: abs_addr_mode();
instruction_set::OpcSTX_zpg: zpg_addr_mode();

instruction_set::OpcSTY_abs: abs_addr_mode();
instruction_set::OpcSTY_zpg: zpg_addr_mode();

instruction_set::OpcTAX_impl: impl_addr_mode();
instruction_set::OpcTAY_impl: impl_addr_mode();
```

```
        instruction_set::OpcTSX_impl: impl_addr_mode();
        instruction_set::OpcTXA_impl: impl_addr_mode();
        instruction_set::OpcTXS_impl: impl_addr_mode();
        instruction_set::OpcTYA_impl: impl_addr_mode();

        default: next_addr_mode = instruction_set::AddrModeImpl;
      endcase
    end
    default: begin
      case (current_addr_mode)
        instruction_set::AddrModeImm: imm_addr_mode();
        instruction_set::AddrModeAbs: abs_addr_mode();
        instruction_set::AddrModeAbsX:
        ↪  absx_addr_mode(bus_sources::DataBusSrcRegX);
        instruction_set::AddrModeAbsY:
        ↪  absx_addr_mode(bus_sources::DataBusSrcRegY);
        instruction_set::AddrModeImpl: impl_addr_mode();
        instruction_set::AddrModeZpg: zpg_addr_mode();
        default: invalid_state();
      endcase
    end
  endcase
end

// ----------------------------------------------------
// ---------- Address Modes System Tasks --------------
// ----------------------------------------------------
task abs_addr_mode();
  next_addr_mode = instruction_set::AddrModeAbs;
  case (current_instr_state)
    InstructionDecode: begin
      next_instr_state = InstructionMem1;
      ctrl_signals[control_signals::CtrlIncEnablePc] = 1;
      ctrl_signals[control_signals::CtrlLoadAddrLow] = 1;
    end
    InstructionMem1: begin
      next_instr_state = InstructionExec1;
      ctrl_signals[control_signals::CtrlIncEnablePc] = 1;
      ctrl_signals[control_signals::CtrlLoadAddrHigh] = 1;
```

```
    end
    default: begin
      current_address_low_bus_input  =
      ↪  bus_sources::AddressLowSrcAddrLowReg;
      current_address_high_bus_input =
      ↪  bus_sources::AddressHighSrcAddrHighReg;
      opcode_exec();
    end
  endcase
endtask

task imm_addr_mode();
  next_addr_mode = instruction_set::AddrModeImm;
  case (current_instr_state)
    InstructionDecode: begin
      next_instr_state = InstructionExec1;
    end
    InstructionExec1: begin
      ctrl_signals[control_signals::CtrlIncEnablePc] = 1;
      opcode_exec();
    end
    default: opcode_exec();
  endcase
endtask

task impl_addr_mode();
  next_addr_mode = instruction_set::AddrModeImpl;
  case (current_instr_state)
    InstructionDecode: begin
      next_instr_state = InstructionExec1;
    end
    default: opcode_exec();
  endcase
endtask

task absx_addr_mode(bus_sources::data_bus_source_t idx_reg);
  next_addr_mode = instruction_set::AddrModeAbsX;
  case (current_instr_state)
    InstructionDecode: begin
```

```
    next_instr_state = InstructionMem1;
    current_data_bus_input = idx_reg;
    ctrl_signals[control_signals::CtrlLoadInputA] = 1;
    ctrl_signals[control_signals::CtrlClearFlagCarry] = 1;
  end
  InstructionMem1: begin
    next_instr_state = InstructionMem2;

    current_data_bus_input = bus_sources::DataBusSrcDataIn;

    ctrl_signals[control_signals::CtrlLoadInputB] = 1;
    ctrl_signals[control_signals::CtrlIncEnablePc] = 1;

    alu_op = control_signals::ALU_ADD;
  end
  InstructionMem2: begin
    next_instr_state = InstructionMem3;

    current_data_bus_input = bus_sources::DataBusSrcRegAluResult;

    ctrl_signals[control_signals::CtrlUpdateFlagCarry] = 1;

    ctrl_signals[control_signals::CtrlLoadAddrLow] = 1;
  end
  InstructionMem3: begin
    if (status_flags[control_signals::StatusFlagCarry]) begin
      next_instr_state = InstructionMem4;
    end else begin
      next_instr_state = InstructionExec1;
    end
    current_data_bus_input = bus_sources::DataBusSrcDataIn;
    ctrl_signals[control_signals::CtrlIncEnablePc] = 1;
    ctrl_signals[control_signals::CtrlLoadAddrHigh] = 1;
  end
  InstructionMem4: begin
    next_instr_state = InstructionExec1;
    ctrl_signals[control_signals::CtrlIncAddressHighReg] = 1;
  end
  default: begin
```

```
      current_address_low_bus_input  =
      ↪  bus_sources::AddressLowSrcAddrLowReg;
      current_address_high_bus_input =
      ↪  bus_sources::AddressHighSrcAddrHighReg;
      opcode_exec();
    end
  endcase
endtask

task zpg_addr_mode();
  next_addr_mode = instruction_set::AddrModeZpg;
  case (current_instr_state)
    InstructionDecode: begin
      next_instr_state = InstructionExec1;
      ctrl_signals[control_signals::CtrlIncEnablePc] = 1;
      ctrl_signals[control_signals::CtrlLoadAddrLow] = 1;
    end
    default: begin
      current_address_low_bus_input  =
      ↪  bus_sources::AddressLowSrcAddrLowReg;
      current_address_high_bus_input = bus_sources::AddressHighSrcZero;
      opcode_exec();
    end
  endcase
endtask

task zpgx_addr_mode(bus_sources::data_bus_source_t idx_reg);
  next_addr_mode = instruction_set::AddrModeZpgX;
  case (current_instr_state)
    InstructionDecode: begin
      next_instr_state = InstructionMem1;
      current_data_bus_input = idx_reg;
      ctrl_signals[control_signals::CtrlLoadInputA] = 1;
      ctrl_signals[control_signals::CtrlClearFlagCarry] = 1;
    end
    InstructionMem1: begin
      next_instr_state = InstructionMem2;

      current_data_bus_input = bus_sources::DataBusSrcZero;
```

```
      ctrl_signals[control_signals::CtrlLoadInputB] = 1;
      ctrl_signals[control_signals::CtrlIncEnablePc] = 1;

      alu_op = control_signals::ALU_ADD;
    end
    InstructionMem2: begin
      next_instr_state = InstructionMem3;

      current_data_bus_input = bus_sources::DataBusSrcRegAluResult;

      ctrl_signals[control_signals::CtrlUpdateFlagCarry] = 1;

      ctrl_signals[control_signals::CtrlLoadAddrLow] = 1;
    end
    InstructionMem3: begin
      if (status_flags[control_signals::StatusFlagCarry]) begin
        next_instr_state = InstructionMem4;
      end else begin
        next_instr_state = InstructionExec1;
      end
      current_data_bus_input = bus_sources::DataBusSrcDataIn;
      ctrl_signals[control_signals::CtrlIncEnablePc] = 1;
      ctrl_signals[control_signals::CtrlLoadAddrHigh] = 1;
    end
    InstructionMem4: begin
      next_instr_state = InstructionExec1;
      ctrl_signals[control_signals::CtrlIncAddressHighReg] = 1;
    end
    default: begin
      current_address_low_bus_input  =
      ↪  bus_sources::AddressLowSrcAddrLowReg;
      current_address_high_bus_input = bus_sources::AddressHighSrcZero;
      opcode_exec();
    end
  endcase
endtask

task invalid_state();
```

```
    $display("Invalid state");
    next_instr_state = InstructionInvalid;
    next_addr_mode   = instruction_set::AddrModeImpl;
  endtask

  // ------------------------------------------------------
  // ---------- Opcode execution system tasks -------------
  // ------------------------------------------------------
  task opcode_exec();
    case (current_opcode)
      instruction_set::OpcADC_imm:
      ↪  exec_arithmetic_op(control_signals::ALU_ADD);
      instruction_set::OpcADC_abs:
      ↪  exec_arithmetic_op(control_signals::ALU_ADD);
      instruction_set::OpcADC_absx:
      ↪  exec_arithmetic_op(control_signals::ALU_ADD);
      instruction_set::OpcADC_absy:
      ↪  exec_arithmetic_op(control_signals::ALU_ADD);
      instruction_set::OpcADC_zpg:
      ↪  exec_arithmetic_op(control_signals::ALU_ADD);

      instruction_set::OpcAND_imm:
      ↪  exec_logic_op(control_signals::ALU_AND);
      instruction_set::OpcAND_abs:
      ↪  exec_logic_op(control_signals::ALU_AND);
      instruction_set::OpcAND_absx:
      ↪  exec_logic_op(control_signals::ALU_AND);
      instruction_set::OpcAND_absy:
      ↪  exec_logic_op(control_signals::ALU_AND);
      instruction_set::OpcAND_zpg:
      ↪  exec_logic_op(control_signals::ALU_AND);

      instruction_set::OpcBCC_abs: exec_branch();
      instruction_set::OpcBCS_abs: exec_branch();
      instruction_set::OpcBEQ_abs: exec_branch();
      instruction_set::OpcBMI_abs: exec_branch();
      instruction_set::OpcBNE_abs: exec_branch();
      instruction_set::OpcBPL_abs: exec_branch();
      instruction_set::OpcBVC_abs: exec_branch();
```

```
instruction_set::OpcBVS_abs: exec_branch();

instruction_set::OpcCLC_impl: exec_clc();

instruction_set::OpcCMP_imm:
↪  exec_cmp(bus_sources::DataBusSrcRegAccumulator);
instruction_set::OpcCMP_abs:
↪  exec_cmp(bus_sources::DataBusSrcRegAccumulator);
instruction_set::OpcCMP_absx:
↪  exec_cmp(bus_sources::DataBusSrcRegAccumulator);
instruction_set::OpcCMP_absy:
↪  exec_cmp(bus_sources::DataBusSrcRegAccumulator);
instruction_set::OpcCMP_zpg:
↪  exec_cmp(bus_sources::DataBusSrcRegAccumulator);

instruction_set::OpcCPX_imm: exec_cmp(bus_sources::DataBusSrcRegX);
instruction_set::OpcCPX_abs: exec_cmp(bus_sources::DataBusSrcRegX);
instruction_set::OpcCPX_zpg: exec_cmp(bus_sources::DataBusSrcRegX);

instruction_set::OpcCPY_imm: exec_cmp(bus_sources::DataBusSrcRegY);
instruction_set::OpcCPY_abs: exec_cmp(bus_sources::DataBusSrcRegY);
instruction_set::OpcCPY_zpg: exec_cmp(bus_sources::DataBusSrcRegY);

instruction_set::OpcEOR_imm:
↪  exec_logic_op(control_signals::ALU_XOR);
instruction_set::OpcEOR_abs:
↪  exec_logic_op(control_signals::ALU_XOR);
instruction_set::OpcEOR_absx:
↪  exec_logic_op(control_signals::ALU_XOR);
instruction_set::OpcEOR_absy:
↪  exec_logic_op(control_signals::ALU_XOR);
instruction_set::OpcEOR_zpg:
↪  exec_logic_op(control_signals::ALU_XOR);

instruction_set::OpcINX_impl: exec_inx();

instruction_set::OpcINY_impl: exec_iny();

instruction_set::OpcJMP_abs: exec_jmp();
```

```
instruction_set::OpcLDA_imm:  exec_lda();
instruction_set::OpcLDA_abs:  exec_lda();
instruction_set::OpcLDA_absx: exec_lda();
instruction_set::OpcLDA_absy: exec_lda();
instruction_set::OpcLDA_zpg:  exec_lda();

instruction_set::OpcLDX_imm:  exec_ldx();
instruction_set::OpcLDX_abs:  exec_ldx();
instruction_set::OpcLDX_absy: exec_ldx();
instruction_set::OpcLDX_zpg:  exec_ldx();

instruction_set::OpcLDY_imm:  exec_ldy();
instruction_set::OpcLDY_abs:  exec_ldy();
instruction_set::OpcLDY_absx: exec_ldy();
instruction_set::OpcLDY_zpg:  exec_ldy();

instruction_set::OpcORA_imm:
↪  exec_logic_op(control_signals::ALU_OR);
instruction_set::OpcORA_abs:
↪  exec_logic_op(control_signals::ALU_OR);
instruction_set::OpcORA_absx:
↪  exec_logic_op(control_signals::ALU_OR);
instruction_set::OpcORA_absy:
↪  exec_logic_op(control_signals::ALU_OR);
instruction_set::OpcORA_zpg:
↪  exec_logic_op(control_signals::ALU_OR);

instruction_set::OpcPHA_impl:
↪  exec_pha(bus_sources::DataBusSrcRegAccumulator);
instruction_set::OpcPHX_impl:
↪  exec_pha(bus_sources::DataBusSrcRegX);
instruction_set::OpcPHY_impl:
↪  exec_pha(bus_sources::DataBusSrcRegY);

instruction_set::OpcPLA_impl:
↪  exec_pla(control_signals::CtrlLoadAccumutator);
instruction_set::OpcPLA_impl: exec_pla(control_signals::CtrlLoadX);
instruction_set::OpcPLA_impl: exec_pla(control_signals::CtrlLoadY);
```

```
instruction_set::OpcSBC_imm:
↪  exec_arithmetic_op(control_signals::ALU_ADD, 1);
instruction_set::OpcSBC_abs:
↪  exec_arithmetic_op(control_signals::ALU_ADD, 1);
instruction_set::OpcSBC_absx:
↪  exec_arithmetic_op(control_signals::ALU_ADD, 1);
instruction_set::OpcSBC_absy:
↪  exec_arithmetic_op(control_signals::ALU_ADD, 1);
instruction_set::OpcSBC_zpg:
↪  exec_arithmetic_op(control_signals::ALU_ADD, 1);

instruction_set::OpcSEC_impl: exec_sec();

instruction_set::OpcSTA_abs:  exec_sta();
instruction_set::OpcSTA_absx: exec_sta();
instruction_set::OpcSTA_absy: exec_sta();
instruction_set::OpcSTA_zpg:  exec_sta();

instruction_set::OpcSTX_abs: exec_stx();
instruction_set::OpcSTX_zpg: exec_stx();

instruction_set::OpcSTY_abs: exec_sty();
instruction_set::OpcSTY_zpg: exec_sty();

instruction_set::OpcTAX_impl:
exec_transfer(bus_sources::DataBusSrcRegAccumulator,
↪  control_signals::CtrlLoadX);

instruction_set::OpcTAY_impl:
exec_transfer(bus_sources::DataBusSrcRegAccumulator,
↪  control_signals::CtrlLoadY);

instruction_set::OpcTSX_impl: exec_tsx();

instruction_set::OpcTXA_impl:
exec_transfer(bus_sources::DataBusSrcRegX,
↪  control_signals::CtrlLoadAccumutator);
```

```
    instruction_set::OpcTXS_impl: exec_txs();

    instruction_set::OpcTYA_impl:
    exec_transfer(bus_sources::DataBusSrcRegY,
    ↪  control_signals::CtrlLoadAccumutator);

    default: invalid_state();
  endcase
endtask

task exec_arithmetic_op(control_signals::alu_op_t alu_op_arg, logic
↪  invert_b = 0);
  alu_op = alu_op_arg;
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionExec2;
      current_data_bus_input = bus_sources::DataBusSrcDataIn;
      ctrl_signals[control_signals::CtrlLoadInputB] = 1;
    end
    InstructionExec2: begin
      next_instr_state = InstructionExec3;
      current_data_bus_input = bus_sources::DataBusSrcRegAccumulator;
      ctrl_signals[control_signals::CtrlLoadInputA] = 1;
    end
    InstructionExec3: begin
      next_instr_state = InstructionFetch;
      ctrl_signals[control_signals::CtrlAluCarryIn] =
      ↪  status_flags[control_signals::StatusFlagCarry];
      if (invert_b) begin
        ctrl_signals[control_signals::CtrlAluInvertB] = 1;
      end
      current_data_bus_input = bus_sources::DataBusSrcRegAluResult;
      ctrl_signals[control_signals::CtrlLoadAccumutator] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagNegative] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagZero] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagCarry] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagOverflow] = 1;
    end
    default: invalid_state();
```

```
    endcase
  endtask

  task exec_logic_op(control_signals::alu_op_t alu_op_arg);
    alu_op = alu_op_arg;
    case (current_instr_state)
      InstructionExec1: begin
        next_instr_state = InstructionExec2;
        current_data_bus_input = bus_sources::DataBusSrcDataIn;
        ctrl_signals[control_signals::CtrlLoadInputB] = 1;
      end
      InstructionExec2: begin
        next_instr_state = InstructionExec3;
        current_data_bus_input = bus_sources::DataBusSrcRegAccumulator;
        ctrl_signals[control_signals::CtrlLoadInputA] = 1;
      end
      InstructionExec3: begin
        next_instr_state = InstructionFetch;
        current_data_bus_input = bus_sources::DataBusSrcRegAluResult;
        ctrl_signals[control_signals::CtrlLoadAccumutator] = 1;
        ctrl_signals[control_signals::CtrlUpdateFlagNegative] = 1;
        ctrl_signals[control_signals::CtrlUpdateFlagZero] = 1;
      end
      default: invalid_state();
    endcase
  endtask

  task exec_branch();
    case (current_instr_state)
      InstructionExec1: begin
        next_instr_state = InstructionExec2;
        ctrl_signals[control_signals::CtrlIncEnablePc] = 0;
        current_data_bus_input = bus_sources::DataBusSrcDataIn;
        ctrl_signals[control_signals::CtrlLoadInputA] = 1;
        ctrl_signals[control_signals::CtrlIncEnablePc] = 1;
      end
      InstructionExec2: begin
        next_instr_state = InstructionExec3;
        current_address_low_bus_input = bus_sources::AddressLowSrcPcLow;
```

```
    current_data_bus_input = bus_sources::DataBusSrcAddrLowBus;
    ctrl_signals[control_signals::CtrlLoadInputB] = 1;
    ctrl_signals[control_signals::CtrlIncEnablePc] = 0;
  end
  InstructionExec3: begin
    next_instr_state = InstructionExec4;
    current_data_bus_input = bus_sources::DataBusSrcRegAluResult;
    ctrl_signals[control_signals::CtrlAluCarryIn] = 0;
    ctrl_signals[control_signals::CtrlLoadAddrLow] = 1;
    ctrl_signals[control_signals::CtrlIncEnablePc] = 0;
  end
  InstructionExec4: begin
    if (negative_data_in == 0 && !alu_carry || negative_data_in == 1
    ↪  && alu_carry) begin
      next_instr_state = InstructionFetch;
      ctrl_signals[control_signals::CtrlIncEnablePc] = 0;
      current_address_high_bus_input =
      ↪  bus_sources::AddressHighSrcPcHigh;
      current_address_low_bus_input =
      ↪  bus_sources::AddressLowSrcAddrLowReg;
      ctrl_signals[control_signals::CtrlLoadPc] =
      ↪  status_flags[current_opcode[7:6]] ~^ current_opcode[5];
    end else if (negative_data_in) begin
      next_instr_state = InstructionExec5;
      ctrl_signals[control_signals::CtrlIncAddressHighReg] = 1;
      ctrl_signals[control_signals::CtrlIncEnablePc] = 0;
    end else begin
      next_instr_state = InstructionExec5;
      ctrl_signals[control_signals::CtrlDecAddressHighReg] = 1;
      ctrl_signals[control_signals::CtrlIncEnablePc] = 0;
    end
  end
  InstructionExec5: begin
    next_instr_state = InstructionFetch;
    ctrl_signals[control_signals::CtrlIncEnablePc] = 0;
    current_address_high_bus_input =
    ↪  bus_sources::AddressHighSrcPcHigh;
    current_address_low_bus_input =
    ↪  bus_sources::AddressLowSrcAddrLowReg;
```

```
      ctrl_signals[control_signals::CtrlLoadPc] =
      ↪  status_flags[current_opcode[7:6]] ~^ current_opcode[5];
    end
    default: invalid_state();
  endcase
endtask

task exec_clc();
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionFetch;
      ctrl_signals[control_signals::CtrlClearFlagCarry] = 1;
    end
    default: invalid_state();
  endcase
endtask

task exec_cmp(bus_sources::data_bus_source_t cmp_src);
  alu_op = control_signals::ALU_ADD;
  ctrl_signals[control_signals::CtrlAluInvertB] = 1;
  ctrl_signals[control_signals::CtrlAluCarryIn] = 0;
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionExec2;
      current_data_bus_input = bus_sources::DataBusSrcDataIn;
      ctrl_signals[control_signals::CtrlLoadInputB] = 1;
    end
    InstructionExec2: begin
      next_instr_state = InstructionExec3;
      current_data_bus_input = cmp_src;
      ctrl_signals[control_signals::CtrlLoadInputA] = 1;
    end
    InstructionExec3: begin
      next_instr_state = InstructionFetch;
      current_data_bus_input = bus_sources::DataBusSrcRegAluResult;
      ctrl_signals[control_signals::CtrlUpdateFlagNegative] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagZero] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagCarry] = 1;
    end
```

```
    default: invalid_state();
  endcase
endtask

task exec_inc();
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionExec2;
      current_data_bus_input = bus_sources::DataBusSrcRegAccumulator;
      ctrl_signals[control_signals::CtrlLoadInputB] = 1;
      ctrl_signals[control_signals::CtrlResetInputA] = 1;
    end
    InstructionExec2: begin
      next_instr_state = InstructionFetch;
      current_data_bus_input = bus_sources::DataBusSrcRegAluResult;
      ctrl_signals[control_signals::CtrlAluCarryIn] = 1;
      ctrl_signals[control_signals::CtrlLoadAccumutator] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagNegative] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagZero] = 1;
    end
    default: invalid_state();
  endcase
endtask

task exec_inx();
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionExec2;
      current_data_bus_input = bus_sources::DataBusSrcRegX;
      ctrl_signals[control_signals::CtrlLoadInputB] = 1;
      ctrl_signals[control_signals::CtrlResetInputA] = 1;
    end
    InstructionExec2: begin
      next_instr_state = InstructionFetch;
      current_data_bus_input = bus_sources::DataBusSrcRegAluResult;
      ctrl_signals[control_signals::CtrlAluCarryIn] = 1;
      ctrl_signals[control_signals::CtrlLoadX] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagNegative] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagZero] = 1;
```

```
    end
    default: invalid_state();
  endcase
endtask

task exec_iny();
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionExec2;
      current_data_bus_input = bus_sources::DataBusSrcRegY;
      ctrl_signals[control_signals::CtrlLoadInputB] = 1;
      ctrl_signals[control_signals::CtrlResetInputA] = 1;
    end
    InstructionExec2: begin
      next_instr_state = InstructionFetch;
      current_data_bus_input = bus_sources::DataBusSrcRegAluResult;
      ctrl_signals[control_signals::CtrlAluCarryIn] = 1;
      ctrl_signals[control_signals::CtrlLoadX] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagNegative] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagZero] = 1;
    end
    default: invalid_state();
  endcase
endtask

task exec_jmp();
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionFetch;
      ctrl_signals[control_signals::CtrlIncEnablePc] = 0;
      ctrl_signals[control_signals::CtrlLoadPc] = 1;
    end
    default: invalid_state();
  endcase
endtask

task exec_lda();
  case (current_instr_state)
    InstructionExec1: begin
```

```
      next_instr_state = InstructionFetch;
      current_data_bus_input = bus_sources::DataBusSrcDataIn;
      ctrl_signals[control_signals::CtrlLoadAccumutator] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagNegative] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagZero] = 1;
    end
    default: invalid_state();
  endcase
endtask

task exec_ldx();
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionFetch;
      current_data_bus_input = bus_sources::DataBusSrcDataIn;
      ctrl_signals[control_signals::CtrlLoadX] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagNegative] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagZero] = 1;
    end
    default: invalid_state();
  endcase
endtask

task exec_ldy();
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionFetch;
      current_data_bus_input = bus_sources::DataBusSrcDataIn;
      ctrl_signals[control_signals::CtrlLoadY] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagNegative] = 1;
      ctrl_signals[control_signals::CtrlUpdateFlagZero] = 1;
    end
    default: invalid_state();
  endcase
endtask

task exec_pha(bus_sources::data_bus_source_t push_src);
  case (current_instr_state)
    InstructionExec1: begin
```

```
      next_instr_state = InstructionFetch;
      current_data_bus_input = push_src;
      current_address_low_bus_input =
      ↪  bus_sources::AddressLowSrcStackPointer;
      current_address_high_bus_input =
      ↪  bus_sources::AddressHighSrcStackPointer;
      ctrl_signals[control_signals::CtrlRead0Write1] = 1;
      ctrl_signals[control_signals::CtrlDecStackPointer] = 1;
    end
    default: invalid_state();
  endcase
endtask

task exec_pla(control_signals::ctrl_signals_t pull_src);
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionExec2;
      ctrl_signals[control_signals::CtrlRead0Write1] = 1;
      ctrl_signals[control_signals::CtrlIncStackPointer] = 1;
    end
    InstructionExec2: begin
      next_instr_state = InstructionFetch;
      current_data_bus_input = bus_sources::DataBusSrcDataIn;
      current_address_low_bus_input =
      ↪  bus_sources::AddressLowSrcStackPointer;
      current_address_high_bus_input =
      ↪  bus_sources::AddressHighSrcStackPointer;
      ctrl_signals[control_signals::CtrlDecStackPointer] = 1;
      ctrl_signals[pull_src] = 1;
    end
    default: invalid_state();
  endcase
endtask

task exec_sec();
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionFetch;
      ctrl_signals[control_signals::CtrlSetFlagCarry] = 1;
```

```
    end
    default: invalid_state();
  endcase
endtask

task exec_sta();
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionFetch;
      current_data_bus_input = bus_sources::DataBusSrcRegAccumulator;
      ctrl_signals[control_signals::CtrlRead0Write1] = 1;
    end
    default: invalid_state();
  endcase
endtask

task exec_stx();
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionFetch;
      current_data_bus_input = bus_sources::DataBusSrcRegX;
      ctrl_signals[control_signals::CtrlRead0Write1] = 1;
    end
    default: invalid_state();
  endcase
endtask

task exec_sty();
  case (current_instr_state)
    InstructionExec1: begin
      next_instr_state = InstructionFetch;
      current_data_bus_input = bus_sources::DataBusSrcRegY;
      ctrl_signals[control_signals::CtrlRead0Write1] = 1;
    end
    default: invalid_state();
  endcase
endtask

task exec_transfer(bus_sources::data_bus_source_t src,
```

```
                    control_signals::ctrl_signals_t load_target);
    case (current_instr_state)
      InstructionExec1: begin
        next_instr_state = InstructionFetch;
        current_data_bus_input = src;
        ctrl_signals[load_target] = 1;
      end
    endcase
  endtask

  task exec_txs();
    case (current_instr_state)
      InstructionExec1: begin
        next_instr_state = InstructionFetch;
        current_address_low_bus_input =
        ↪  bus_sources::AddressLowSrcStackPointer;
        current_data_bus_input = bus_sources::DataBusSrcAddrLowBus;
        ctrl_signals[control_signals::CtrlLoadX] = 1;
      end
    endcase
  endtask

  task exec_tsx();
    case (current_instr_state)
      InstructionExec1: begin
        next_instr_state = InstructionFetch;
        current_data_bus_input = bus_sources::DataBusSrcRegX;
        current_address_low_bus_input =
        ↪  bus_sources::AddressLowSrcDataBus;
        ctrl_signals[control_signals::CtrlLoadStackPointer] = 1;
      end
    endcase
  endtask



endmodule
```

APÊNDICE E – cpu6502.sv

```
module cpu6502 (
    input logic reset,
    input logic clk_in,
    output logic READ_write,
    input logic [7:0] data_in,
    output logic [7:0] data_out,
    output logic [15:0] address_out
);

  // ---------------------------------------------------------
  // ----------------- CONTROL SIGNALS -----------------------
  // ---------------------------------------------------------
  logic ctrl_signals[control_signals::CtrlSignalEndMarker];

  control_signals::alu_op_t alu_op;

  logic [7:0] data_in_latch;

  assign READ_write = ctrl_signals[control_signals::CtrlRead0Write1];

  // ---------------------------------------------------------------
  // ----------------- Data and Address Buses ---------------------
  // ---------------------------------------------------------------

  bus_sources::data_bus_source_t current_data_bus_input;
  bus_sources::address_low_bus_source_t current_address_low_bus_input;
  bus_sources::address_high_bus_source_t current_address_high_bus_input;

  logic [7:0] data_bus, address_low_bus, address_high_bus;
  logic [7:0] data_bus_inputs[bus_sources::DataBusSrcEndMarker];
  logic [7:0]
  ↪  address_low_bus_inputs[bus_sources::AddressLowSrcEndMarker];
  logic [7:0]
  ↪  address_high_bus_inputs[bus_sources::AddressHighSrcEndMarker];
```

```
assign data_bus = data_bus_inputs[current_data_bus_input];
assign data_out = data_bus;

assign data_bus_inputs[bus_sources::DataBusSrcDataIn] = data_in;
assign data_bus_inputs[bus_sources::DataBusSrcDataInLatch] =
↪  data_in_latch;
assign data_bus_inputs[bus_sources::DataBusSrcFF] = 8'hff;
assign data_bus_inputs[bus_sources::DataBusSrcZero] = 8'h00;

assign address_low_bus =
↪  address_low_bus_inputs[current_address_low_bus_input];
assign address_high_bus =
↪  address_high_bus_inputs[current_address_high_bus_input];
assign address_out = {
  address_high_bus_inputs[current_address_high_bus_input],
  address_low_bus_inputs[current_address_low_bus_input]
};
assign address_high_bus_inputs[bus_sources::AddressHighSrcStackPointer]
↪  = 8'h01;
assign address_low_bus_inputs[bus_sources::AddressLowSrcZero] = 8'h00;
assign address_high_bus_inputs[bus_sources::AddressHighSrcZero] =
↪  8'h00;

assign data_bus_inputs[bus_sources::DataBusSrcAddrLowBus] =
↪  address_low_bus;
assign data_bus_inputs[bus_sources::DataBusSrcAddrHighBus] =
↪  address_high_bus;

assign address_low_bus_inputs[bus_sources::AddressLowSrcDataBus] =
↪  data_bus;
assign address_high_bus_inputs[bus_sources::AddressHighSrcDataBus] =
↪  data_bus;

// ------------------------------------------------------------
// ----------------- Datapath Components ----------------------
// ------------------------------------------------------------

// GPR registers
register RegAccumulator (
```

```
    .data_in(data_bus),
    .data_out(data_bus_inputs[bus_sources::DataBusSrcRegAccumulator]),
    .clk(clk_in),
    .load(ctrl_signals[control_signals::CtrlLoadAccumutator]),
    .reset(reset),
    .inc(1'b0),
    .dec(1'b0)
);

register RegX (
    .data_in(data_bus),
    .data_out(data_bus_inputs[bus_sources::DataBusSrcRegX]),
    .clk(clk_in),
    .load(ctrl_signals[control_signals::CtrlLoadX]),
    .reset(reset),
    .inc(1'b0),
    .dec(1'b0)
);
register RegY (
    .data_in(data_bus),
    .data_out(data_bus_inputs[bus_sources::DataBusSrcRegY]),
    .clk(clk_in),
    .load(ctrl_signals[control_signals::CtrlLoadY]),
    .reset(reset),
    .inc(1'b0),
    .dec(1'b0)
);

register AddressLowReg (
    .data_in(data_bus),

    ↪  .data_out(address_low_bus_inputs[bus_sources::AddressLowSrcAddrLowReg]),
    .clk(clk_in),
    .load(ctrl_signals[control_signals::CtrlLoadAddrLow]),
    .reset(reset),
    .inc(1'b0),
    .dec(1'b0)
);
register AddressHighReg (
```

```
    .data_in(data_bus),

    ↪  .data_out(address_high_bus_inputs[bus_sources::AddressHighSrcAddrHighReg]),
    .clk(clk_in),
    .load(ctrl_signals[control_signals::CtrlLoadAddrHigh]),
    .inc(ctrl_signals[control_signals::CtrlIncAddressHighReg]),
    .dec(ctrl_signals[control_signals::CtrlDecAddressHighReg]),
    .reset(reset)
);

stack_pointer StackPointer (
    .data_in(data_bus),

    ↪  .data_out(address_low_bus_inputs[bus_sources::AddressLowSrcStackPointer]),
    .clk(clk_in),
    .load(ctrl_signals[control_signals::CtrlLoadStackPointer]),
    .dec(ctrl_signals[control_signals::CtrlDecStackPointer]),
    .inc(ctrl_signals[control_signals::CtrlIncStackPointer]),
    .reset(reset)
);

// ALU + registers
logic [7:0] alu_input_a, alu_input_b;
logic alu_overflow, alu_zero, alu_negative, alu_carry;
register InputA (
    .data_in(data_bus),
    .data_out(alu_input_a),
    .clk(clk_in),
    .load(ctrl_signals[control_signals::CtrlLoadInputA]),
    .reset(reset | ctrl_signals[control_signals::CtrlResetInputA]),
    .inc(1'b0),
    .dec(1'b0)
);
register InputB (
    .data_in(data_bus),
    .data_out(alu_input_b),
    .clk(clk_in),
    .load(ctrl_signals[control_signals::CtrlLoadInputB]),
    .reset(reset),
```

```
    .inc(1'b0),
    .dec(1'b0)
);

logic status_flags[8];
logic flag_carry, flag_zero, flag_negative, flag_overflow;

alu alu (
    .carry_in(ctrl_signals[control_signals::CtrlAluCarryIn]),
    .input_a(alu_input_a),
    .input_b(alu_input_b),
    .invert_b(ctrl_signals[control_signals::CtrlAluInvertB]),
    .operation(alu_op),
    .alu_out(data_bus_inputs[bus_sources::DataBusSrcRegAluResult]),
    .overflow_out(alu_overflow),
    .zero_out(alu_zero),
    .negative_out(alu_negative),
    .carry_out(alu_carry)
);

// Program Counter
logic [15:0] program_counter;
assign program_counter = {
  address_high_bus_inputs[bus_sources::AddressHighSrcPcHigh],
  address_low_bus_inputs[bus_sources::AddressLowSrcPcLow]
};
program_counter ProgramCounter (
    .PCL_in(address_low_bus),
    .PCH_in(address_high_bus),
    .clk(clk_in),
    .inc_enable(ctrl_signals[control_signals::CtrlIncEnablePc]),
    .load(ctrl_signals[control_signals::CtrlLoadPc]),
    .reset(reset),
    .PCL_out(address_low_bus_inputs[bus_sources::AddressLowSrcPcLow]),

    ↪  .PCH_out(address_high_bus_inputs[bus_sources::AddressHighSrcPcHigh])
);

// Status Register
```

```
assign status_flags[control_signals::StatusFlagCarry] = flag_carry;
assign status_flags[control_signals::StatusFlagZero] = flag_zero;
assign status_flags[control_signals::StatusFlagNegative] =
↪  flag_negative;
assign status_flags[control_signals::StatusFlagOverflow] =
↪  flag_overflow;
status_register status_register (
    .data_in        (data_bus),
    .update_zero
    ↪  (ctrl_signals[control_signals::CtrlUpdateFlagZero]),

    ↪  .update_negative(ctrl_signals[control_signals::CtrlUpdateFlagNegative]),
    .update_carry
    ↪  (ctrl_signals[control_signals::CtrlUpdateFlagCarry]),

    ↪  .update_overflow(ctrl_signals[control_signals::CtrlUpdateFlagOverflow]),
    .set_carry      (ctrl_signals[control_signals::CtrlSetFlagCarry]),
    .set_overflow
    ↪  (ctrl_signals[control_signals::CtrlSetFlagOverflow]),
    .clear_carry
    ↪  (ctrl_signals[control_signals::CtrlClearFlagCarry]),
    .clear_overflow
    ↪  (ctrl_signals[control_signals::CtrlClearFlagOverflow]),
    .clk            (clk_in),
    .reset          (reset),
    .carry_in       (alu_carry),
    .overflow_in    (alu_overflow),
    .flag_carry     (flag_carry),
    .flag_zero      (flag_zero),
    .flag_negative  (flag_negative),
    .flag_overflow  (flag_overflow)
);

// Instruction Register
instruction_set::opcode_t instruction_register;
register InstructionRegister (
    .data_in(data_bus),
```

```
        // Expliciting telling to pass every bit to cast the enum reg into
        ↪  a reg
        .data_out(instruction_register[7:0]),
        .clk(clk_in),
        .load(ctrl_signals[control_signals::CtrlLoadInstReg]),
        .reset(reset),
        .inc(1'b0),
        .dec(1'b0)
    );

    control_unit control_unit (
        .status_flags                   (status_flags),
        .alu_carry                      (alu_carry),
        .data_in_latch                  (data_in_latch),
        .current_opcode                 (instruction_register),
        .ctrl_signals                   (ctrl_signals),
        .alu_op                         (alu_op),
        .current_data_bus_input         (current_data_bus_input),
        .current_address_low_bus_input  (current_address_low_bus_input),
        .current_address_high_bus_input(current_address_high_bus_input),
        .clk                            (clk_in),
        .reset                          (reset)
    );


    // ------------------------------------------------------
    // ------------------ CONTROL LOGIC ---------------------
    // ------------------------------------------------------

    always_ff @(posedge clk_in) begin
      data_in_latch <= data_in;
    end
endmodule
```

APÊNDICE F – dev.sv

```
module dev (

    ////////////// CLOCK //////////
    input          CLOCK_50,
    // input CLOCK2_50,
    // input CLOCK3_50,
    // ////////////// SEG7 //////////
    output [ 0:6] HEX0,
    output [ 0:6] HEX1,
    output [ 0:6] HEX2,
    output [ 6:0] HEX3,
    output [ 0:6] HEX4,
    output [ 0:6] HEX5,
    output [ 0:6] HEX6,
    output [ 0:6] HEX7,
    ////////////// PUSH BUTTON ////
    input  [17:0] SW,
    ////////////// LEDS //////////
    output [17:0] LEDR,
    output [ 8:0] LEDG,
    ////////////// LCD //////////
    output        LCD_BLON,
    output [ 7:0] LCD_DATA,
    output        LCD_EN,
    output        LCD_ON,
    output        LCD_RS,
    output        LCD_RW,
    ////////////// SRAM //////////
    output [19:0] SRAM_ADDR,
    output        SRAM_CE_N,
    inout  [15:0] SRAM_DQ,
    output        SRAM_LB_N,
    output        SRAM_OE_N,
    output        SRAM_UB_N,
    output        SRAM_WE_N
```

```
);
  logic clk;
  logic reset, read_write;
  logic [7:0] data_in_cpu, data_out_cpu;
  logic [15:0] address_out;

  assign reset   = SW[0];
  assign LEDG[0] = reset;
  assign LEDG[1] = clk;

  hex_display hex_display_a (
      .value       (port_a_out),
      .hex_ones    (HEX4),
      .hex_sixteens(HEX5),
  );

  hex_display hex_display_b (
      .value       (port_b_out),
      .hex_ones    (HEX6),
      .hex_sixteens(HEX7),
  );

  bcd_display bcd_display (
      .value       (port_a_out),
      .hex_ones    (HEX2),
      .hex_tens    (HEX1),
      .hex_hundreds(HEX0)
  );

  logic clk_div;

  clock u_clock (
      .clk_in (CLOCK_50),
      .clk_out(clk_div)
  );

  assign clk = SW[2] ? CLOCK_50 : clk_div;

  cpu6502 cpu6502 (
```

```
    .reset      (reset),
    .clk_in     (clk),
    .READ_write (read_write),
    .data_in    (data_in_cpu),
    .data_out   (data_out_cpu),
    .address_out(address_out)
);

logic [7:0] ram_out;
logic ram_cs;
assign ram_cs = address_out < 16'h800;

assign ram_out = read_write ? 8'bz : SRAM_DQ[7:0];
assign SRAM_DQ = read_write ? {8'b0, data_out_cpu} : 16'bz;
assign SRAM_ADDR = {8'b0, address_out[11:0]};
assign SRAM_CE_N = ~ram_cs;
assign SRAM_LB_N = 0;
assign SRAM_UB_N = 0;
assign SRAM_WE_N = ~read_write;
assign SRAM_OE_N = read_write;

logic [7:0] rom_out;
logic rom_cs;
assign rom_cs = address_out >= 16'h8000;
rom #(
    .init_file("simulation/rom.hex"),
    .depth(15)
) prg_rom (
    .address (address_out[14:0]),
    .data_out(rom_out),
    .clk     (~clk)
);

logic [7:0] port_a_in, port_a_out, port_b_in, port_b_out;
logic [7:0] interface_adapter_out;
logic interface_adapter_cs;
assign port_b_in = SW[17:9];
assign interface_adapter_cs = address_out >= 16'h800 && address_out <
↪  16'h810;
```

```
  interface_adapter interface_adapter (
      .port_a_in      (port_a_in),
      .port_a_out     (port_a_out),
      .port_b_in      (port_b_in),
      .port_b_out     (port_b_out),
      .data_in        (data_out_cpu),
      .data_out       (interface_adapter_out),
      .register_select(address_out[3:0]),
      .chip_en        (interface_adapter_cs),
      .clk            (clk),
      .reset          (reset)
  );

  always_comb begin
    data_in_cpu = 8'bz;
    if (interface_adapter_cs) begin
      data_in_cpu = interface_adapter_out;
    end else if (rom_cs) begin
      data_in_cpu = rom_out;
    end else if (ram_cs) begin
      data_in_cpu = ram_out;
    end
  end

  assign LEDR = ~SW[1] ? {address_out, 2'b0} : {port_a_out, 2'b0,
  ↪  port_b_out};

  assign HEX3 = 7'hff;
  // assign LCD_EN = port_a_out[7];
  // assign LCD_RW = port_a_out[6];
  // assign LCD_RS = port_a_out[5];
  // assign LCD_ON = 1'b1;
  // assign LCD_BLON = SW[2];
  // assign LCD_DATA = port_b_out;


endmodule
```

APÊNDICE G – interface_adapter.sv

```
typedef enum logic [3:0] {
  ORB_IRB,
  ORA_IRA,
  DDRB,
  DDRA,

  CtrlRegisterEndMarker
} ctrl_register;


module interface_adapter (
    input  logic [7:0] port_a_in,
    output logic [7:0] port_a_out,

    input  logic [7:0] port_b_in,
    output logic [7:0] port_b_out,

    input  logic [7:0] data_in,
    output logic [7:0] data_out,

    input logic [3:0] register_select,
    input logic chip_en,

    input logic clk,
    input logic reset
    // input logic readb_write
);


  typedef enum logic [3:0] {
    IDLE,
    READ_PORT_A,
    READ_PORT_B,
    WRITE_PORT_A,
    WRITE_PORT_B
```

```
} ctrl_state;

logic [7:0] ctrl_registers[CtrlRegisterEndMarker];
logic [7:0] port_a, port_b;

assign port_a_out = port_a;
assign port_b_out = port_b;

ctrl_state current_state, next_state;

always_ff @(posedge clk) begin
  if (reset) begin
    ctrl_registers <= '{default: '0};
    port_a <= 8'h0;
    port_b <= 8'h0;
  end else
  if (!chip_en) begin

  end else begin
    case (register_select)
      DDRA: ctrl_registers[DDRA] <= data_in;
      DDRB: ctrl_registers[DDRB] <= data_in;
      ORA_IRA: begin
        for (int i = 0; i < 8; i++) begin
          if (ctrl_registers[DDRA][i]) begin
            port_a[i] <= data_in[i];
          end else begin
            data_out[i] <= port_a_in[i];
            port_a[i]   <= port_a_in[i];
          end
        end
      end
      ORB_IRB: begin
        for (int i = 0; i < 8; i++) begin
          if (ctrl_registers[DDRB][i]) begin
            port_b[i] <= data_in[i];
          end else begin
            data_out[i] <= port_b_in[i];
            port_b[i]   <= port_b_in[i];
```

```
            end
          end
        end
      endcase
    end
  end

endmodule
```

APÊNDICE H – program_counter.sv

```
module program_counter (
    input wire [7:0] PCL_in,
    input wire [7:0] PCH_in,

    input wire clk,
    input wire inc_enable,
    input wire load,
    input wire reset,

    output wire [7:0] PCL_out,
    output wire [7:0] PCH_out

);

  reg [15:0] current_pc;

  assign PCL_out = current_pc[7:0];
  assign PCH_out = current_pc[15:8];

  always @(posedge clk) begin

    if (reset) begin
      current_pc <= 16'h8000;
    end else if (load) begin
      current_pc <= {PCH_in, PCL_in};
    end else begin
      if (inc_enable) begin
        current_pc <= current_pc + 1'b1;
      end else begin
        current_pc <= current_pc;
      end
    end
  end

endmodule
```

APÊNDICE I – register.sv

```
module register (
    input  logic [7:0] data_in,
    output logic [7:0] data_out,

    input logic clk,
    input logic load,
    input logic reset,
    input logic inc,
    input logic dec
);
  reg [7:0] current_value;

  always @(posedge clk) begin
    if (reset) begin
      current_value <= 8'b0;
    end else if (load) begin
      current_value <= data_in;
    end else if (inc) begin
      current_value <= current_value + 1;
    end else if (dec) begin
      current_value <= current_value - 1;
    end else begin
      current_value <= current_value;
    end
  end

  assign data_out = current_value;

endmodule
```

APÊNDICE J – rom.sv

```
module rom #(
    parameter init_file = "synthesis/rom_init.mif",
    depth = 15
) (
    input clk,
    input logic [14:0] address,
    output logic [7:0] data_out
);
  // (* ram_init_file = init_file *) logic [7:0] ram[2**depth];

  logic [7:0] rom[2**depth];
  initial begin
    $readmemh(init_file, rom);
  end
  // assign data_out = rom[address];
  always_ff @(posedge clk) begin
    data_out <= rom[address];
  end
endmodule
```

APÊNDICE K – stack_pointer.sv

```
module stack_pointer (
    input  logic [7:0] data_in,
    output logic [7:0] data_out,

    input logic clk,
    input logic reset,
    input logic inc,
    input logic dec,
    input logic load
);

  always_ff @(posedge clk) begin
    if (reset) begin
      data_out <= 8'hff;
    end else if (load) begin
      data_out <= data_in;
    end else if (inc) begin
      data_out <= data_out + 1;
    end else if (dec) begin
      data_out <= data_out - 1;
    end else begin
      data_out <= data_out;
    end
  end

endmodule
```

APÊNDICE L – status_register.sv

```
module status_register (
    input logic [7:0] data_in,

    input wire update_carry,
    input wire update_zero,
    input wire update_negative,
    input wire update_overflow,

    input wire set_carry,
    input wire clear_carry,
    input wire set_overflow,
    input wire clear_overflow,

    input wire clk,
    input wire reset,

    input wire carry_in,
    input wire overflow_in,

    output logic flag_carry,
    output logic flag_zero,
    output logic flag_negative,
    output logic flag_overflow
);

  always_ff @(posedge clk) begin
    if (reset) begin
      flag_carry <= 0;
      flag_zero <= 0;
      flag_negative <= 0;
      flag_overflow <= 0;
    end else begin
      if (set_carry) begin
        flag_carry <= 1;
      end else if (clear_carry) begin
```

```
        flag_carry <= 0;
      end else if (update_carry) begin
        flag_carry <= carry_in;
      end else begin
        flag_carry <= flag_carry;
      end

      if (set_overflow) begin
        flag_overflow <= 1;
      end else if (clear_overflow) begin
        flag_overflow <= 0;
      end else if (update_overflow) begin
        flag_overflow <= overflow_in;
      end else begin
        flag_overflow <= flag_overflow;
      end

      flag_zero <= update_zero ? ~|data_in : flag_zero;
      flag_negative <= update_negative ? data_in[7] : flag_negative;
    end
  end

endmodule
```

APÊNDICE M – status_register.sv

```
package bus_sources;
  typedef enum logic [15:0] {
    DataBusSrcRegAccumulator,
    DataBusSrcRegX,
    DataBusSrcRegY,
    DataBusSrcRegAluResult,

    DataBusSrcFF,
    DataBusSrcZero,

    DataBusSrcDataIn,
    DataBusSrcDataInLatch,

    DataBusSrcAddrLowBus,
    DataBusSrcAddrHighBus,

    DataBusSrcEndMarker
  } data_bus_source_t;

  typedef enum logic [15:0] {
    AddressLowSrcPcLow,

    AddressLowSrcDataIn,
    AddressLowSrcDataInLatch,

    AddressLowSrcAddrLowReg,
    AddressLowSrcZero,

    AddressLowSrcStackPointer,
    AddressLowSrcDataBus,

    AddressLowSrcEndMarker
  } address_low_bus_source_t;

  typedef enum logic [15:0] {
```

```
    AddressHighSrcPcHigh,

    AddressHighSrcDataIn,
    AddressHighSrcDataInLatch,

    AddressHighSrcAddrHighReg,

    AddressHighSrcZero,

    AddressHighSrcStackPointer,
    AddressHighSrcDataBus,

    AddressHighSrcEndMarker
  } address_high_bus_source_t;

endpackage
```

APÊNDICE N – status_register.sv

```
package control_signals;

  typedef enum logic [3:0] {
    ALU_ADD,
    ALU_SUB,
    ALU_AND,
    ALU_OR,
    ALU_XOR,
    ALU_SHIFT_LEFT
  } alu_op_t;

  typedef enum logic [31:0] {
    CtrlLoadAccumutator = 0,
    CtrlLoadX = 1,
    CtrlLoadY = 2,
    CtrlLoadInputA = 3,
    CtrlLoadInputB = 4,
    CtrlLoadInstReg = 5,
    CtrlIncEnablePc = 6,
    CtrlLoadPc = 7,
    CtrlLoadStatusReg = 8,
    CtrlRead0Write1 = 9,
    CtrlLoadAddrLow = 10,
    CtrlLoadAddrHigh = 11,
    CtrlUpdateFlagCarry = 12,
    CtrlUpdateFlagOverflow = 13,
    CtrlUpdateFlagNegative = 14,
    CtrlUpdateFlagZero = 15,
    CtrlSetFlagCarry = 16,
    CtrlSetFlagOverflow = 17,
    CtrlClearFlagCarry = 18,
    CtrlClearFlagOverflow = 19,
    CtrlIncAddressHighReg = 20,
    CtrlDecAddressHighReg = 21,
    CtrlAluCarryIn = 22,
```

```
    CtrlResetInputA = 23,
    CtrlLoadStackPointer = 24,
    CtrlIncStackPointer = 25,
    CtrlDecStackPointer = 26,
    CtrlAluInvertB = 27,

    CtrlSignalEndMarker
  } ctrl_signals_t;

  typedef enum logic [31:0] {
    StatusFlagNegative,
    StatusFlagOverflow,
    StatusFlagCarry,
    StatusFlagZero,

    StatusFlagEndMarker
  } status_flags_t;

endpackage
```

APÊNDICE O – status_register.sv

```
package instruction_set;

  typedef enum logic [7:0] {
    // 86/149
    OpcADC_imm  = 8'h69,
    OpcADC_abs  = 8'h6d,
    OpcADC_absx = 8'h7d,
    OpcADC_absy = 8'h79,
    OpcADC_zpg  = 8'h65,

    OpcAND_imm  = 8'h29,
    OpcAND_abs  = 8'h2d,
    OpcAND_absx = 8'h3d,
    OpcAND_absy = 8'h39,
    OpcAND_zpg  = 8'h25,

    OpcBCC_abs = 8'h90,
    OpcBCS_abs = 8'hb0,
    OpcBEQ_abs = 8'hf0,
    OpcBMI_abs = 8'h30,
    OpcBNE_abs = 8'hd0,
    OpcBPL_abs = 8'h10,
    OpcBVC_abs = 8'h50,
    OpcBVS_abs = 8'h70,

    OpcCLC_impl = 8'h18,

    OpcCMP_imm  = 8'hc9,
    OpcCMP_abs  = 8'hcd,
    OpcCMP_absx = 8'hdd,
    OpcCMP_absy = 8'hd9,
    OpcCMP_zpg  = 8'hc5,

    OpcCPX_imm = 8'he0,
    OpcCPX_abs = 8'hec,
```

```
    OpcCPX_zpg = 8'he4,

    OpcCPY_imm = 8'hc0,
    OpcCPY_abs = 8'hcc,
    OpcCPY_zpg = 8'hc4,

    OpcEOR_imm  = 8'h49,
    OpcEOR_abs  = 8'h4d,
    OpcEOR_absx = 8'h5d,
    OpcEOR_absy = 8'h59,
    OpcEOR_zpg  = 8'h45,

    OpcINX_impl = 8'he8,

    OpcINY_impl = 8'hc8,

    OpcJMP_abs = 8'h4c,

    OpcLDA_imm  = 8'ha9,
    OpcLDA_abs  = 8'had,
    OpcLDA_absx = 8'hbd,
    OpcLDA_absy = 8'hb9,
    OpcLDA_zpg  = 8'ha5,

    OpcLDX_imm  = 8'ha2,
    OpcLDX_abs  = 8'hae,
    OpcLDX_absy = 8'hbe,
    OpcLDX_zpg  = 8'ha6,

    OpcLDY_imm  = 8'ha0,
    OpcLDY_abs  = 8'hac,
    OpcLDY_absx = 8'hbc,
    OpcLDY_zpg  = 8'ha4,

    OpcNOP_impl = 8'hea,

    OpcORA_imm  = 8'h09,
    OpcORA_abs  = 8'h0d,
    OpcORA_absx = 8'h1d,
```

```
  OpcORA_absy = 8'h19,
  OpcORA_zpg  = 8'h05,

  OpcPHA_impl = 8'h48,
  OpcPHX_impl = 8'hda,
  OpcPHY_impl = 8'h5a,

  OpcPLA_impl = 8'h68,
  OpcPLX_impl = 8'hfa,
  OpcPLY_impl = 8'h7a,

  OpcSBC_imm  = 8'he9,
  OpcSBC_abs  = 8'hed,
  OpcSBC_absx = 8'hfd,
  OpcSBC_absy = 8'hf9,
  OpcSBC_zpg  = 8'he5,

  OpcSEC_impl = 8'h38,

  OpcSTA_abs  = 8'h8d,
  OpcSTA_absx = 8'h9d,
  OpcSTA_absy = 8'h99,
  OpcSTA_zpg  = 8'h85,

  OpcSTX_abs = 8'h8e,
  OpcSTX_zpg = 8'h86,

  OpcSTY_abs = 8'h8c,
  OpcSTY_zpg = 8'h84,

  OpcTAX_impl = 8'haa,
  OpcTAY_impl = 8'ha8,
  OpcTSX_impl = 8'hba,
  OpcTXA_impl = 8'h8a,
  OpcTXS_impl = 8'h9a,
  OpcTYA_impl = 8'h98
} opcode_t;

typedef enum logic [7:0] {
```

```
    AddrModeImm,
    AddrModeAbs,
    AddrModeAbsX,
    AddrModeAbsY,
    AddrModeStack,
    AddrModeImpl,
    AddrModeZpg,
    AddrModeZpgX,
    AddrModeRel
  } address_mode_t;

endpackage
```

ANEXO A – Template para testbenches

```
`timescale 1ns / 1ps

module processador_test ();
  logic clk = 1;
  always #10 clk = ~clk;

  // signals and modules declaration here

  // unit under test
  uut_module uut()

  initial begin
    // write the simulation stimulus here


    // use this snippet to wait for 1 (or more) clk cycles
    repeat (1) @(posedge clk);

    $stop;
  end
endmodule
```

Fonte: Autoria Própria