

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	9
Figura 2 – Endereçamento imediato	12
Figura 3 – Endereçamento absoluto. Note que o primeiro byte na memória é o menos significativo	13
Figura 4 – Endereçamento absoluto - Deslocado em X (ou Y)	14
Figura 5 – Endereçamento <i>Zero-Page</i>	15
Figura 6 – Endereçamento relativo	16
Figura 7 – Endereçamento indireto	17
Figura 8 – Datapath do 6502 implementado	22
Figura 9 – Teste da ULA	43
Figura 10 – Teste do Contador de Programa	43
Figura 11 – Teste do Registrador	44
Figura 12 – Teste da Unidade de Processamento	44

Lista de tabelas

Tabela 1 – Tamanho da instrução por modo de endereçamento	15
Tabela 2 – Conjunto de instruções	19

Lista de Códigos Fonte

1	Enums para barramentos	23
2	Enums para sinais de controle	24
3	Enums para conjunto de instrução	26
4	Módulo ULA	28
5	Módulo Contador de Programa	29
6	Módulo Registrador	30
7	Módulo Processador	31
8	Módulo Registrador de Status	37
9	Testbench da ULA	39
10	Testbench do Contador de Programa	40
11	Testbench do Módulo Registrador	41

Sumário

1	INTRODUÇÃO	7
1.1	Metodologia	7
1.2	Objetivos	7
2	FUNDAMENTAÇÃO TEÓRICA	9
2.1	Visão geral de um sistema computacional	9
2.1.1	Arquitetura de von Neumann	9
2.2	Microprocessador 6502	10
2.2.1	Arquitetura Original	10
2.2.2	Registrador de Status (SR)	10
2.2.3	Contador de Programa (PC)	11
2.2.4	<i>Stack Pointer</i> (SP)	11
2.2.5	Modos de endereçamento	11
2.2.5.1	Endereçamento imediato	11
2.2.5.2	Endereçamento absoluto	12
2.2.5.3	Endereçamento absoluto - Deslocado em X (ou Y)	12
2.2.5.4	Endereçamento <i>Zero-Page</i>	12
2.2.5.5	Endereçamento relativo	12
2.2.5.6	Endereçamento indireto	13
2.2.6	Endereçamento <i>Zero-Page</i> , deslocado em X (ou Y)	13

2.3	RTL - <i>Register-Transfer Logic</i>	13
2.4	Metodologia de Testes	15
3	DESENVOLVIMENTO	19
3.1	Conjunto de instruções	19
3.2	O <i>datapath</i> da implementação	20
3.3	Unidades funcionais	21
3.3.1	Registradores de propósito geral (AC, X, Y)	21
3.3.2	Contador de Programa (PCH e PCL)	21
3.3.3	Registrador de Dados (DR)	21
3.3.4	Registrador de Status (P)	21
3.3.5	Unidade de controle	22
3.3.6	Geração de <i>Clock</i> (CLK)	23
3.3.7	Registrador do barramento de endereço (ABL e ABH)	23
3.3.8	Unidade Lógica aritmética (ALU)	23
3.4	Código Fonte do Processador	23
3.5	Testbenches	38
4	RESULTADOS OBTIDOS E DISCUSSÕES	43
4.1	Formas de onda	43
5	CONSIDERAÇÕES FINAIS	45
	REFERÊNCIAS	47
	APÊNDICES	49
	ANEXO A – TEMPLATE PARA TESTBENCHES	51

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

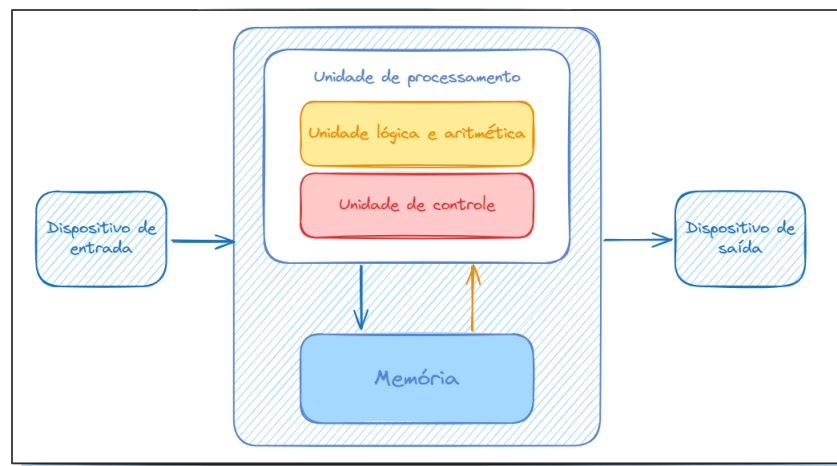
2.1 Visão geral de um sistema computacional

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 possui 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.

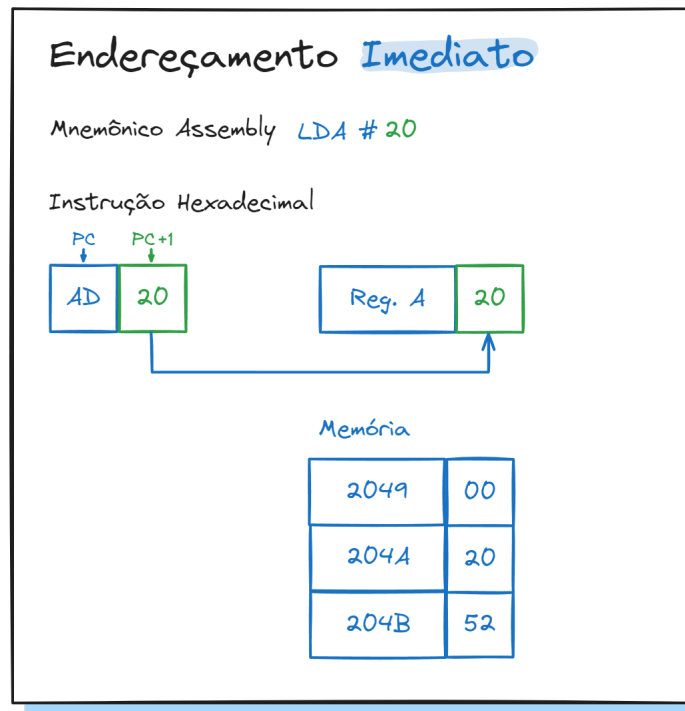
2.2.5 Modos de endereçamento

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).

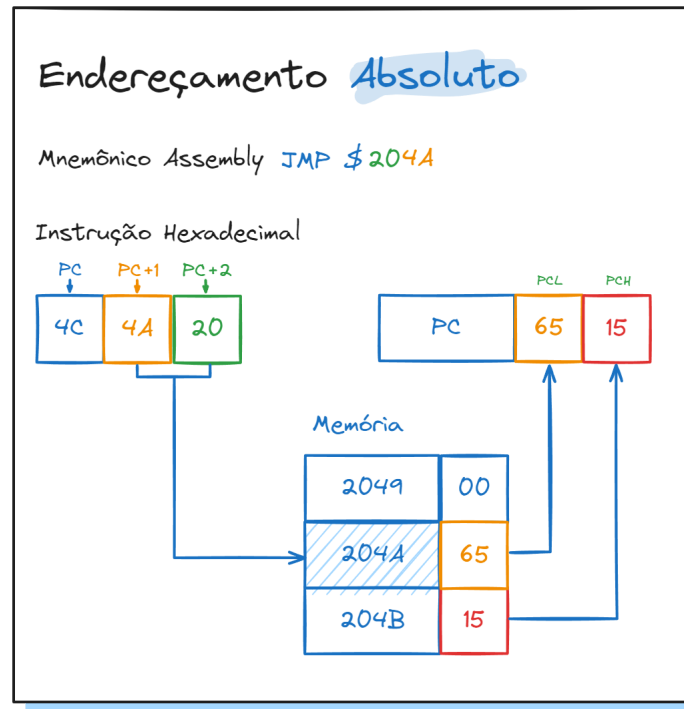
2.2.5.4 Endereçamento Zero-Page

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).

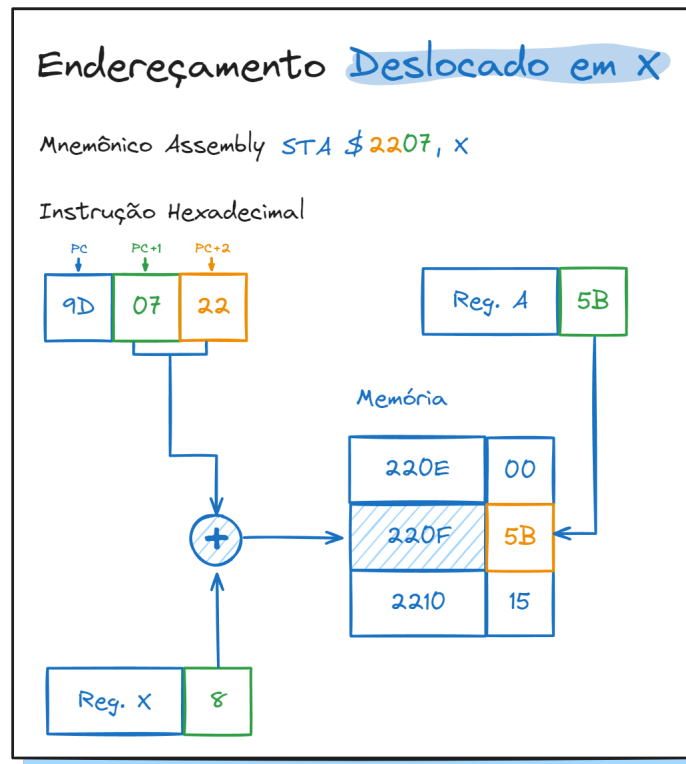
2.2.6 Endereçamento Zero-Page, deslocado em X (ou Y)

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 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.

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



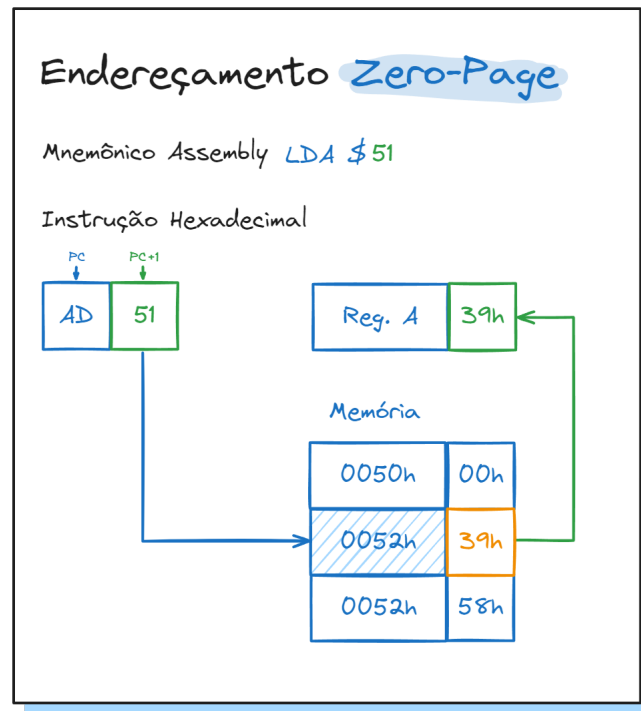
Fonte: Autoria própria

3 níveis diferentes de abstração são definidos por 4 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.
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.

Figura 5 – Endereçamento *Zero-Page*

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

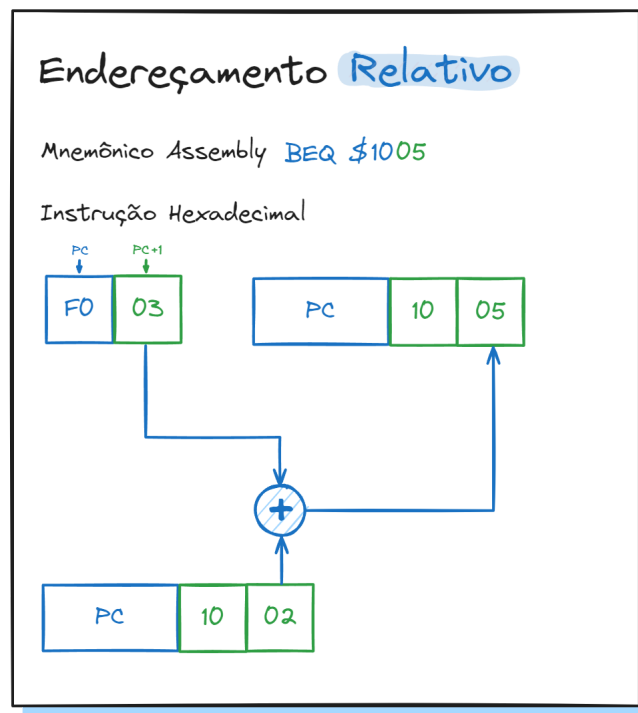
Fonte: Autoria Própria

2.4 Metodologia de Testes

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

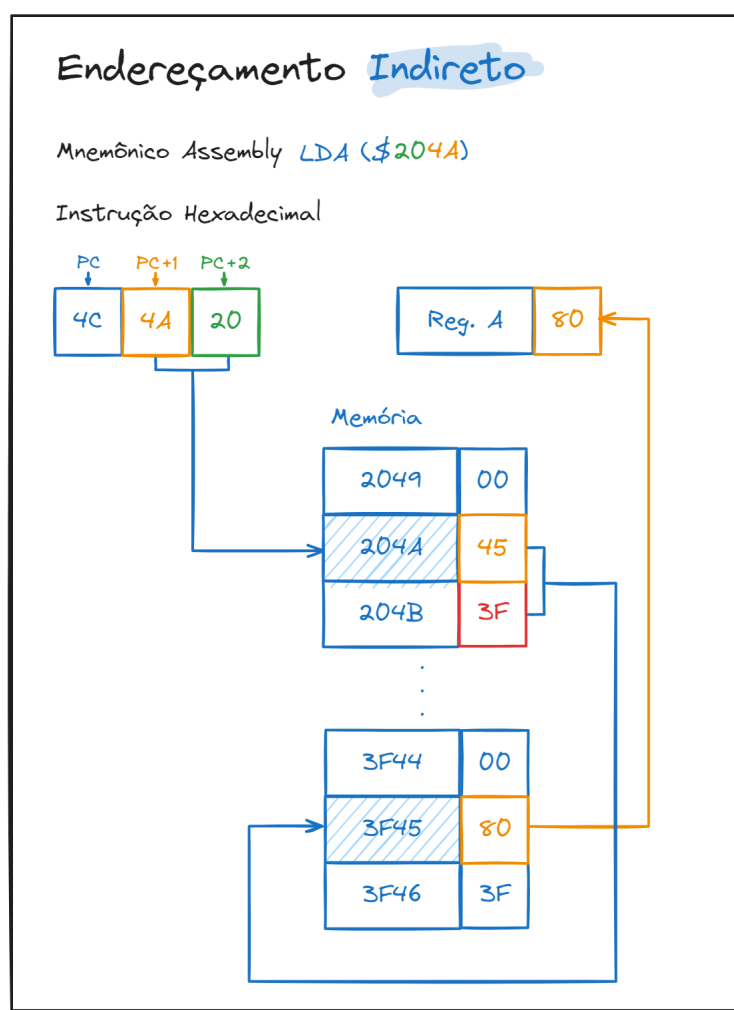
Um template para os testbenches está disponível no [Apêndice A](#)

Figura 6 – Endereçamento relativo



Fonte: Autoria própria

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 [\(5\)](#).

Tabela 2 – Conjunto de instruções

Instruções de transferência				
Instrução	Opcode	Mod. End	Assembly	Operação
LDA	a9	imm	LDA imm	RegAC \leftarrow imm
	ad	abs	LDA addr	RegAC \leftarrow Mem[addr]
	bd	(abs, x)	LDA addr, x	RegAC \leftarrow Mem[addr + x]
LDX	a2	imm	LDX imm	RegX \leftarrow imm
	ae	abs	LDX addr	RegX \leftarrow Mem[addr]
	be	(abs, y)	LDY addr, y	RegY \leftarrow Mem[addr + x]
LDY	a0	imm	LDY imm	RegY \leftarrow imm
	ac	abs	LDY addr	RegY \leftarrow Mem[addr]
	bc	(abs, x)	LDY addr, x	RegY \leftarrow Mem[addr + x]
STA	8d	abs	STA addr	Mem[addr] \leftarrow RegAC
	9d	(abs, x)	STA addr, x	Mem[addr + x] \leftarrow RegAC
STX	8e	abs	STX addr	Mem[addr] \leftarrow RegX
STY	8c	abs	STY addr	Mem[addr] \leftarrow RegY
Instruções lógicas e aritméticas				
Instrução	Opcode	Mod. End	Assembly	Operação
ADC	69	imm	ADC imm	RegAC \leftarrow RegAC + imm + C
	6d	abs	ADC addr	RegAC \leftarrow RegAC + Mem[addr] + C
	7d	(abs, x)	ADC addr, x	RegAC \leftarrow RegAC + Mem[addr + x] + C
SBC	e9	imm	SBC imm	RegAC \leftarrow RegAC - imm - C
	ed	abs	SBC addr	RegAC \leftarrow RegAC - Mem[addr] - C
	fd	(abs, x)	SBC addr, x	RegAC \leftarrow RegAC - Mem[addr + x] - C
AND	29	imm	AND imm	RegAC \leftarrow RegAC AND imm
	2d	abs	AND addr	RegAC \leftarrow RegAC AND Mem[addr]
	3d	(abs, x)	AND addr, x	RegAC \leftarrow RegAC AND Mem[addr + x]
EOR	49	imm	EOR imm	RegAC \leftarrow RegAC XOR imm
	4d	abs	EOR addr	RegAC \leftarrow RegAC XOR Mem[addr]
	5d	(abs, x)	EOR addr, x	RegAC \leftarrow RegAC XOR Mem[addr + x]
ORA	09	imm	ORA imm	RegAC \leftarrow RegAC OR imm
	0d	abs	ORA addr	RegAC \leftarrow RegAC OR Mem[addr]

ORA	1d	(abs, x)	ORA addr, x	RegAC <= RegAC OR Mem[addr + x]
Comparação				
Instrução	Opcode	Mod. End	Assembly	Operação
CMP	c9	imm	CMP imm	C, N, V, Z <= ACC - imm
	cd	abs	CMP addr	C, N, V, Z <= ACC - Mem[addr]
	dd	(abs, x)	CMP addr, x	C, N, V, Z <= ACC - Mem[addr - x]
Flags				
Instrução	Opcode	Mod. End	Assembly	Operação
SEC	38	impl	SEC	C <= 1
CLC	18	impl	CLC	C <= 0
Instruções de branch				
Instrução	Opcode	Mod. End	Assembly	Operação
BCC	90	rel	BCC	branch on C = 0
BCS	b0	rel	BCS	branch on C = 1
BEQ	f0	rel	BEQ	branch on Z = 1
BMI	30	rel	BMI	branch on N = 1
BNE	d0	rel	BNE	branch on Z = 0
BPL	10	rel	BPL	branch on N = 0
BVC	50	rel	BVC	branch on V = 0
BVS	70	rel	BVS	branch on V = 1
Instruções de controle				
JMP	6c	abs	JMP HHLL	PCL <= LL PCH <= HH
NOP	ea	impl	NOP	
HLT	db	impl	HLT	

Fonte: Autoria Própria

3.2 O *datapath* da implementação

A [Figura 8](#) mostra o *datapath* que será implementado durante esse trabalho.

O processador possui 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 possui um Barramento de Endereços de 16-bits, isso significa que ele se comunica com até 65,536 diferentes endereços. Esses diferentes endereços serão mencionados daqui em diante como o Espaço de Memória (EM) de 6502.

3.3 Unidades funcionais

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

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.

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*.

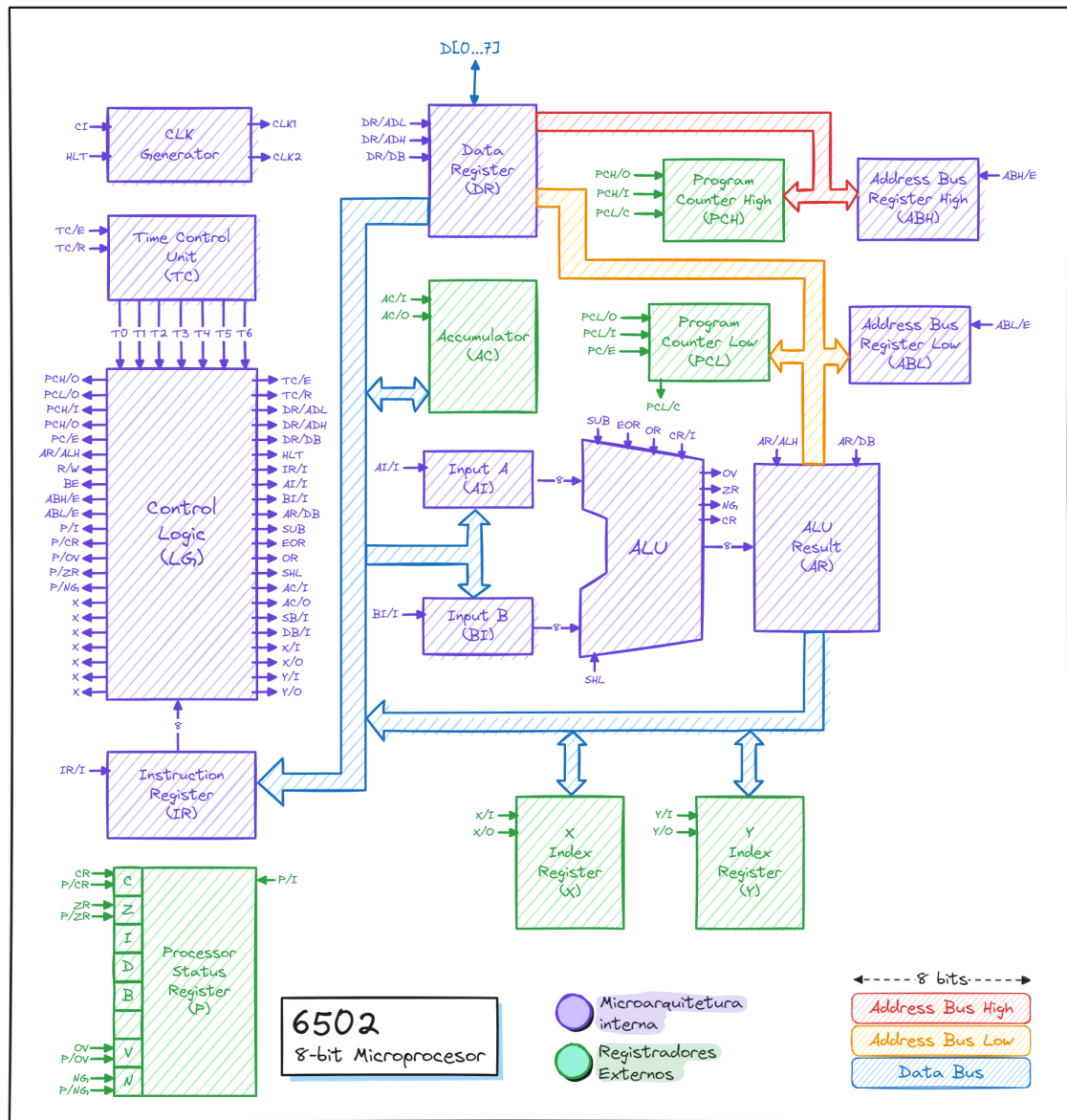
3.3.3 Registrador de Dados (DR)

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*.

Figura 8 – Datapath do 6502 implementado



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 do processador é apresentado nessa seção. Os componentes de *datapath* são descritos em [Módulo ULA](#), [Módulo Contador de Programa](#), [Módulo Registrador](#) e [Módulo Registrador de Status](#). O [Módulo Registrador](#) é usado para implementar os registradores de propósito geral Acumulador, X e Y.

[Enums para barramentos](#), [Enums para sinais de controle](#) e [Enums para conjunto de instrução](#) são usados para providenciar constantes que são usadas pelos diversos módulos do projeto.

Código 1 – bus_sources.sv

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

    DataBusSrcFF,
    DataBusSrcZero,
```

```

    DataBusSrcDataIn,
    DataBusSrcDataInLatch,

    DataBusSrcEndMarker
} data_bus_source_t;

typedef enum logic [15:0] {
    AddressLowSrcPcLow,

    AddressLowSrcDataIn,
    AddressLowSrcDataInLatch,

    AddressLowSrcAddrLowReg,

    AddressLowSrcStackPointer,

    AddressLowSrcEndMarker
} address_low_bus_source_t;

typedef enum logic [15:0] {
    AddressHighSrcPcHigh,

    AddressHighSrcDataIn,
    AddressHighSrcDataInLatch,

    AddressHighSrcAddrHighReg,

    AddressHighSrcStackPointer,

    AddressHighSrcEndMarker
} address_high_bus_source_t;

endpackage

```

Código 2 – control_signals.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,
    CtrlReadWrite1 = 9,
    CtrlLoadAddrLow = 10,
    CtrlLoadAddrHigh = 11,
    CtrlUpdateFlagCarry = 12,
    CtrlUpdateFlagOverflow = 13,
    CtrlUpdateFlagNegative = 14,
    CtrlUpdateFlagZero = 15,
    CtrlSetFlagCarry = 16,
    CtrlSetFlagOverflow = 17,
    CtrlClearFlagCarry = 18,
    CtrlClearFlagOverflow = 19,
    CtrlIncAddressHighReg = 20,
    CtrlAluCarryIn = 21,
    CtrlResetInputA = 22,
    CtrlLoadStackPointer = 23,
    CtrlIncStackPointer = 24,
    CtrlDecStackPointer = 25,
    CtrlAluInvertB = 26,

    CtrlSignalEndMarker
} ctrl_signals_t;
```

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

    StatusFlagEndMarker
} status_flags_t;
```

```
endpackage
```

Código 3 – instruction_set.sv

```
package instruction_set;

typedef enum logic [7:0] {
    OpcADC_imm   = 8'h69,
    OpcADC_abs   = 8'h6d,
    OpcADC_absx  = 8'h7d,

    OpcAND_imm   = 8'h29,
    OpcAND_abs   = 8'h2d,
    OpcAND_absx  = 8'h3d,

    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,

    OpcCPX_imm   = 8'he0,
```

```
OpcCPX_abs = 8'hec,
```

```
OpcCPY_imm = 8'hc0,
```

```
OpcCPY_abs = 8'hcc,
```

```
OpcEOR_imm = 8'h49,
```

```
OpcEOR_abs = 8'h4d,
```

```
OpcEOR_absx = 8'h5d,
```

```
OpcINX_impl = 8'he8,
```

```
OpcJMP_abs = 8'h4c,
```

```
OpcLDA_imm = 8'ha9,
```

```
OpcLDA_abs = 8'had,
```

```
OpcLDA_absx = 8'hbd,
```

```
OpcLDX_imm = 8'ha2,
```

```
OpcLDX_abs = 8'hae,
```

```
OpcLDY_imm = 8'ha0,
```

```
OpcLDY_abs = 8'hac,
```

```
OpcLDY_absx = 8'hbc,
```

```
OpcNOP_impl = 8'hea,
```

```
OpcORA_imm = 8'h09,
```

```
OpcORA_abs = 8'h0d,
```

```
OpcORA_absx = 8'h1d,
```

```
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,

    OpcSEC_impl = 8'h38,

    OpcSTA_abs  = 8'h8d,
    OpcSTA_absx = 8'h9d,

    OpcSTX_abs = 8'h8e,

    OpcSTY_abs = 8'h8c
} opcode_t;

typedef enum logic [7:0] {
    AddrModeImm,
    AddrModeAbs,
    AddrModeAbsX,
    AddrModeStack,
    AddrModeImpl,
    AddrModeRel
} address_mode_t;

endpackage

```

Código 4 – 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
```

```
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 <= 0;  
    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
```

Código 6 – 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
);
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 begin
        current_value <= current_value;
    end
end

assign data_out = current_value;

endmodule

```

Código 7 – 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;

// -----
// ----- 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)
);
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)
);
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)
);

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)
);
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]),
        .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)
    );
    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)
    );

    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)
);

control_unit control_unit (
    .status_flags      (status_flags),
    .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

```

Código 8 – 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
```

3.5 Testbenches

Além dos módulos sintetizáveis apresentados na sessão anterior, também foram desenvolvidos módulos de *testbench*, com o objetivo de testar os módulos em um ambiente digital. Note que para o teste de módulo da ULA, o clock é incluído, apesar de não ser necessário visto que a ULA em si não utiliza esse sinal, ele está lá apenas para controlar a

passagem de tempo durante a simulação.

Código 9 – alu.test.sv

```
`timescale 1ns / 1ps

module alu_test ();
    reg                carry_in;
    reg                [7:0] input_a;
    reg                [7:0] input_b;
    control_signals::alu_op_t operation;
    reg                flag_overflow;
    reg                flag_zero;
    reg                flag_neg;
    reg                flag_carry;
    reg                [7:0] alu_out;

    logic                clk;
    initial clk = 1;
    always #10 clk = ~clk;

    alu alu (
        .carry_in(carry_in),
        .input_a(input_a),
        .input_b(input_b),
        .operation(operation),
        .overflow_out(flag_overflow),
        .zero_out(flag_zero),
        .negative_out(flag_neg),
        .carry_out(flag_carry),
        .alu_out(alu_out)
    );

    initial begin
        operation = control_signals::ALU_ADD;
        carry_in  = 1'b1;
        input_a   = 8'h5;
        input_b   = 8'h5;
        repeat (1) @(posedge clk);
    end
endmodule
```

```

    operation = control_signals::ALU_SUB;
    carry_in  = 1'b0;
    input_a   = 8'h4;
    input_b   = 8'h5;
    repeat (1) @(posedge clk);

    input_a = 8'h3;
    input_b = 8'h8;
    repeat (1) @(posedge clk);

    input_a = 8'h5;
    input_b = 8'h5;
    repeat (1) @(posedge clk);

    operation = control_signals::ALU_SHIFT_LEFT;
    input_a    = 8'b1100_0011;
    input_b    = 8'b1;
    repeat (1) @(posedge clk);

    $stop;
end
endmodule

```

Código 10 – program_counter.test.sv

```

`timescale 1ns / 1ps

module program_counter_test ();

    logic [7:0] PCL_in;
    logic [7:0] PCH_in;
    logic clk;
    logic inc_enable;
    logic load;
    logic reset;

    logic [7:0] PCL_out;
    logic [7:0] PCH_out;

    program_counter pc (

```

```
.PCL_in(PCL_in),  
.PCH_in(PCH_in),  
.clk(clk),  
.inc_enable(inc_enable),  
.load(load),  
.reset(reset),  
.PCL_out(PCL_out),  
.PCH_out(PCH_out)  
);  
  
initial clk = 1;  
  
always #10 clk = ~clk;  
  
initial begin  
    inc_enable = 1'b1;  
    load = 1'b0;  
    reset = 1;  
    PCL_in = 8'h3f;  
    PCH_in = 8'hfe;  
    repeat (1) @(posedge clk);  
    reset = 0;  
  
    repeat (2) @(posedge clk);  
    inc_enable = 0;  
    load = 1;  
    repeat (1) @(posedge clk);  
    inc_enable = 1;  
    load = 0;  
  
    repeat (8) @(posedge clk);  
    inc_enable = 0;  
  
    repeat (2) @(posedge clk);  
    $stop;  
end  
endmodule
```

```
module register_test ();
    logic clk;
    initial clk = 1;
    always #10 clk = ~clk;

    logic [7:0] data_in, data_out;
    logic load, reset;

    register register (
        .data_in (data_in),
        .data_out(data_out),
        .clk      (clk),
        .load     (load),
        .reset    (reset)
    );

    initial begin
        reset = 1;
        load = 0;
        data_in = 8'hea;
        repeat (1) @(posedge clk);
        reset = 0;
        load = 1;
        repeat (1) @(posedge clk);
        load = 0;
        repeat (2) @(posedge clk);

        $stop;
    end

endmodule
```

4 Resultados Obtidos e Discussões

4.1 Formas de onda

Todas as formas de onda foram geradas a partir das testbenches apresentadas na seção 3.5.

Figura 9 – Teste da ULA

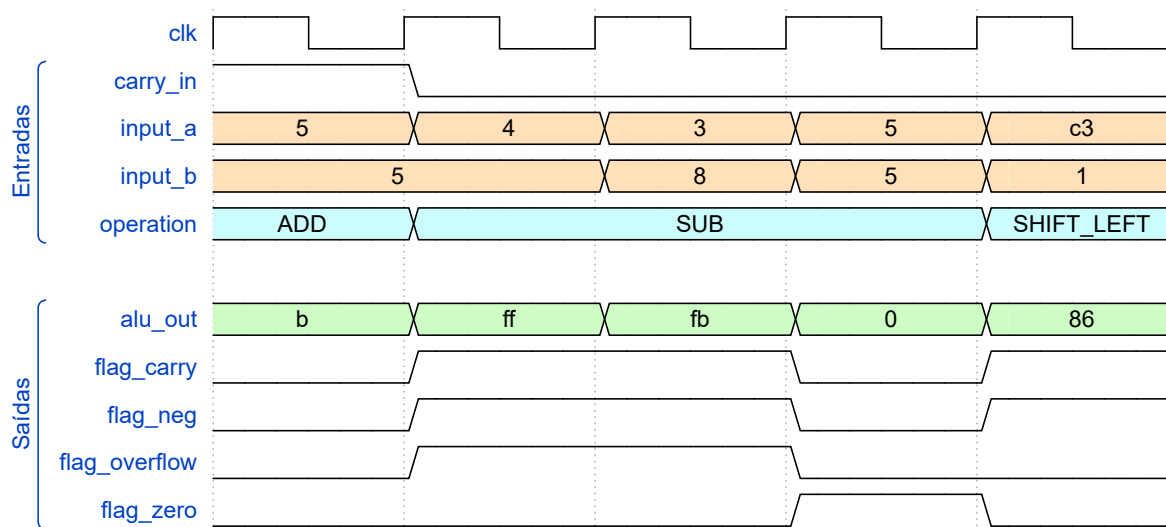
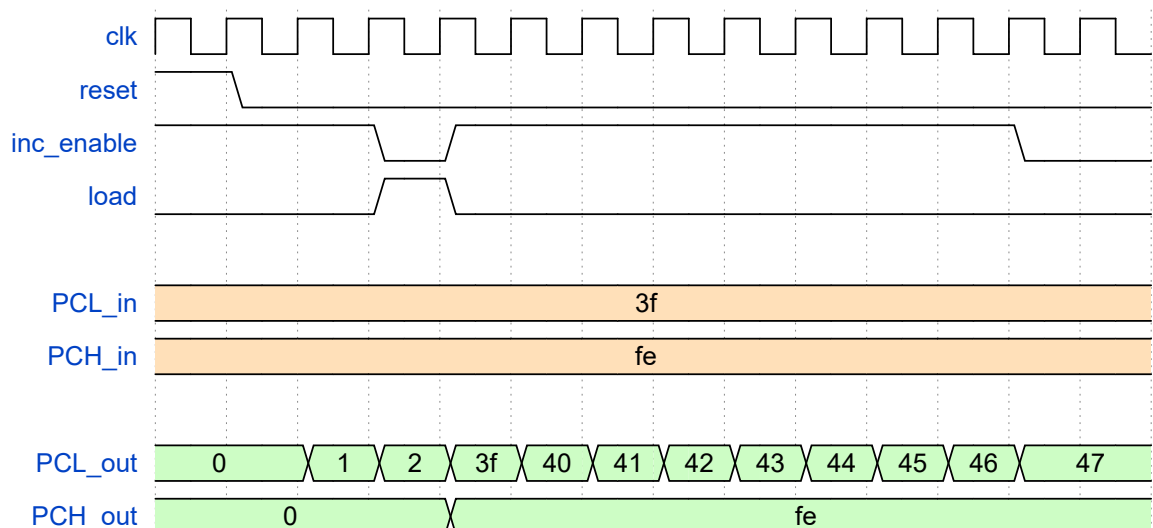
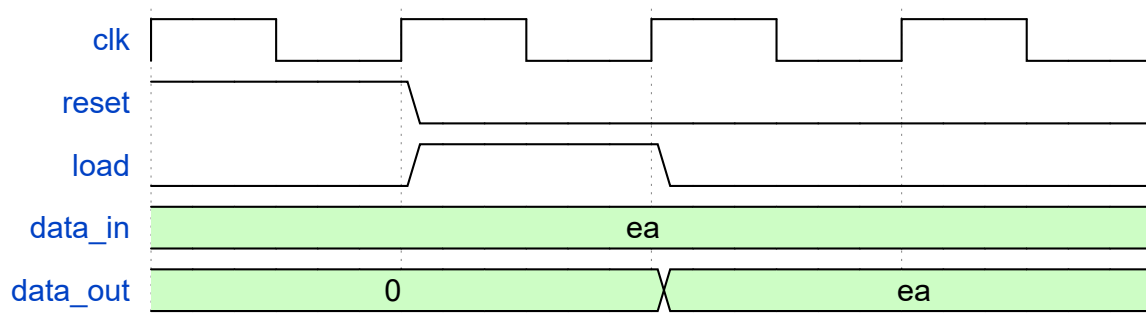


Figura 10 – Teste do Contador de Programa



O teste da Figura 12 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* e *Decode* e depois volta ao ciclo de *Fetch* já que a operação **NOP** não precisa de nenhum ciclo extra de execução.

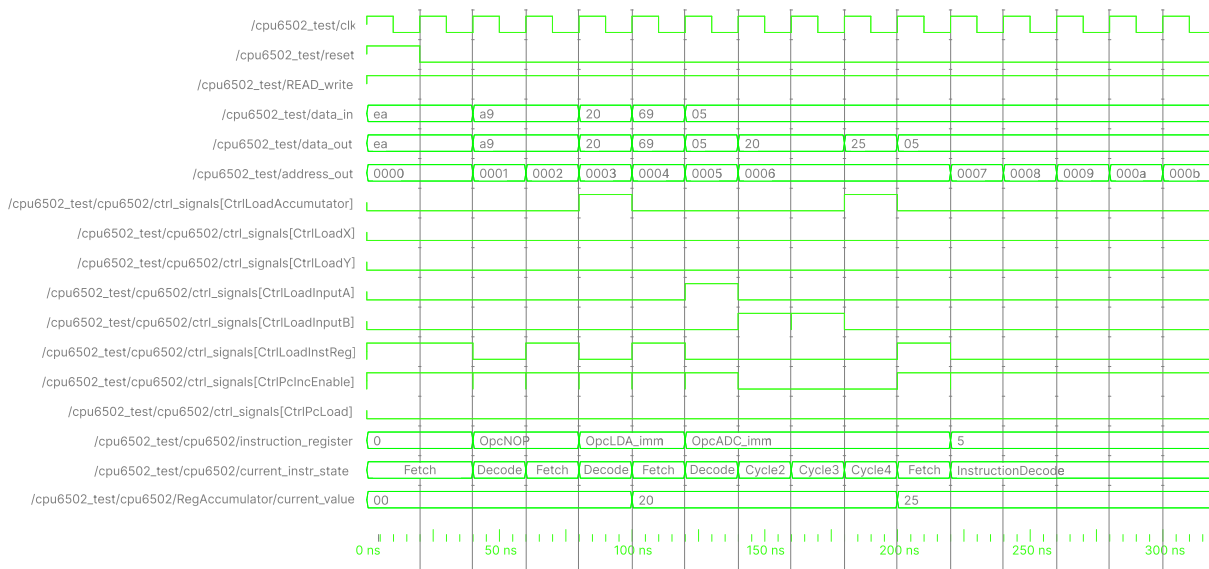
Figura 11 – Teste do Registrador



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 12 – Teste da Unidade de Processamento



5 Considerações Finais

Em seu estado atual o processador pode executar 3 opcodes do conjunto de instrução. Vale também destacar que em nenhum dos testes apresentados o processador realiza acesso a uma memória. Todas as operações foram colocadas no barramento de dados pelo próprio simulador. Isso se deu ao fato de que o processador ainda não é capaz de executar o endereçamento absoluto.

Como desenvolvimento para os próximos pontos de controle, os demais opcodes serão implementados, bem como acesso a memória externa será testado.

Referências

- 1 NEUMANN, J. von. *Introduction to “The First Draft Report on the EDVAC”*. 1945. Citado na página 9.
- 2 TECHNOLOGY, I. M. *MCS6500 Microcomputer Family Hardware Manual*. 2nd edition. ed. Norristown, PA: MOS Technology, INC, 1976. Citado na página 10.
- 3 CENTER, I. T. W. D. *W65C02S 8-bit Microprocessor Datasheet*. Vienna, Austria, 2018. Citado na página 11.
- 4 VAHID, F. *Digital Design with RTL Design, VHDL, and Verilog*. 2nd edition. ed. Waltham/MA, EUA: John Wiley & Sons, Inc, 2011. Citado na página 14.
- 5 LANDSTEINER, M. N. *6502 Instruction Set*. Disponível em: <https://www.masswerk.at/6502/6502_instruction_set.html>. Citado na página 19.

Apêndices

ANEXO A – Template para testbenches

```
`timescale 1ns / 1ps

module template_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
```