UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

DESCRIÇÃO VHDL E PROTOTIPAÇÃO EM FPGA DO PROCESSADOR MOS 6502

TRABALHO DE CONCLUSÃO DE CURSO

Bernardo Favero Andreeti

Santa Maria, RS, Brasil

2015

DESCRIÇÃO VHDL E PROTOTIPAÇÃO EM FPGA DO PROCESSADOR MOS 6502

Bernardo Favero Andreeti

Trabalho de conclusão de curso apresentado como requisito parcial para obtenção do grau de **Engenheiro de Computação**.

Orientador: Prof. Dr. Everton Alceu Carara

Santa Maria, RS, Brasil

2015

Universidade Federal de Santa Maria
Centro de Tecnologia
Graduação em Engenharia de Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Conclusão de Curso

DESCRIÇÃO EM VHDL E PROTOTIPAÇÃO EM FPGA DO PROCESSADOR MOS 6502

elaborado por
Bernardo Favero Andreeti

COMISSÃO EXAMINADORA:

_____
Prof. Dr. Everton Carara (UFSM)
(Presidente/orientador)

_____
Prof. Dr. José Eduardo Baggio (UFSM)

_____
Prof. Dr. Giovani Baratto (UFSM)

Santa Maria, 11 de dezembro de 2015.

RESUMO

Trabalho de Conclusão de Curso
Graduação em Engenharia de Computação
Universidade Federal de Santa Maria

IMPLEMENTAÇÃO E PROTOTIPAÇÃO EM FPGA DO PROCESSADOR 6502

AUTOR: BERNARDO FAVERO ANDREETI
ORIENTADOR: EVERTON CARARA

Data e Local da Defesa: Santa Maria, 11 de dezembro de 2015.

O projeto descrito por este trabalho consistiu a revisitação do microprocessador 6502, um circuito integrado amplamente utilizado nas décadas de 1970 e 1980. Os dois primeiros computadores Apple, consoles e computadores da Atari e o primeiro console da empresa Nintendo são alguns exemplos de aplicações famosas desse microprocessador. O projeto teve início com o estudo da documentação já existente acerca do processador de forma a proporcionar o entendimento da arquitetura e organização originais. A etapa seguinte consistiu na definição da estrutura do bloco de dados e bloco de controle a serem utilizados como base da descrição, bem como a definição das micro operações realizadas em cada ciclo de processamento. Posteriormente, iniciamos a implementação na linguagem VHDL seguindo diagramas definidos na etapa anterior, sendo que buscamos obter um circuito completamente sintetizável tanto para o fluxo de projeto de ASIC como para FPGA. Para que fosse possível a verificação da implementação realizada, utilizamos como hardware alvo a FPGA Spartan 6, contida no kit de desenvolvimento Nexys 3. Para verificar o funcionamento do processador em uma aplicação real foi utilizada uma descrição HDL do console *Nintendo Entertainment System* (NES). Dessa forma, pudemos avaliar o comportamento do processador descrito em VHDL em um sistema mais complexo, no qual a carga computacional é grande e há interação com outros circuitos. Ao final do curso do projeto obtivemos um processador funcional e com suporte ao conjunto de instruções original.

Palavras-chave: 6502, microprocessadores, síntese.

ABSTRACT

Course Completion Assignment
Graduate Degree in Computer Engineering
Federal University of Santa Maria

FPGA IMPLEMENTATION AND PROTOTYPING OF THE MOS 6502 PROCESSOR

AUTHOR: BERNARDO FAVERO ANDREETI
COUNSELLOR: EVERTON CARARA
Date and Location of Defence: Santa Maria, 2015.

The project described by this assignment consisted in revisiting the 6502 microprocessor, an integrated circuit widely used in 1970s and 1980s. The first two Apple computers, consoles and computers from Atari and the first Nintendo console are some examples of famous applications of this microprocessor. The project started with a study of the existing documentation about the processor in order to understand its original architecture and organization. The following stage consisted in defining the structure of the data path and control path, which were used as basis for the implementation. Posteriorly, we started the implementation in VHDL language following the diagrams built on the previous stage, whereas we seek to obtain a completely synthesizable circuit for both ASIC and FPGA design flows. In order to verify the implementation developed, we used the Spartan 6 FPGA as target hardware, which is available in Nexys 3 development kit. As validation in a real application, we used a HDL description of the Nintendo Entertainment System (NES). Thus, we could evaluate the behavior of the developed processor in a complex system, in which the computational load is high and there is interaction with other circuits. At the end of the project, we obtained a functional processor with support to the original instruction set.

Key-words: 6502, microprocessors, synthesis.

LISTA DE SIGLAS

APU – *Audio Processing Unit* / Unidade de Processamento de Áudio

CPU – *Central Processing Unit* / Unidade Central de Processamento

DMA – *Direct Memory Access* / Acesso Direto à Memória

FPGA – *Field Programmable Gate Array* / Arranjo de Portas Programável em Campo

NES – *Nintendo Entertainment System*

PPU – *Picture Processing Unit* / Unidade de Processamento de Imagem

RAM – *Random Access Memory* / Memória de Acesso Randômico

ROM – *Read Only Memory* / Memória Somente Leitura

RTL – *Register Transfer Level* / Nível de Transferência de Registradores

SID – *Sound Interface Device* / Dispositivo de Interface de Som

SOC – *System-On-Chip* / Sistema em Único Chip

UART – *Universal Asynchronous Receiver/Transmiter* / Transmissor/Receptor Universal Assíncrono

USB – *Universal Serial Bus* / Barramento Serial Universal

VGA – *Video Graphics Array* / Padrão de Disposição Gráfica

VHDL – *VHSIC Hardware Description Language* / Linguagem de Descrição de Hardware

VHSIC – *Very High Speed Integrated Circuits* / Circuitos Integrados de Alta Velocidade

VLSI – *Very Large Scale Integration* / Integração em Grande Escala

ÍNDICE DE FIGURAS

Índice de Tabelas

ÍNDICE

1 INTRODUÇÃO

O processador 6502 foi desenvolvido pela empresa MOS Technology no ano de 1975 e representou um marco na história da computação por contar com um conjunto de instruções simples, porém versátil, atrelado a um custo acessível se comparado aos concorrentes da época. Dessa forma, esse microprocessador de 8 bits chamou atenção dos desenvolvedores, sendo aplicado em dispositivos famosos como o Apple I e II, computadores Commodore, consoles e computadores Atari, além do *Nintendo Entertainment System*, tornando-se um dos processadores de maior sucesso comercial da história.

Tais projetos representaram uma revolução nos dispositivos computacionais na década de 1980 e contribuíram largamente para o crescimento dos jogos virtuais e primeiros sistemas operacionais. Atualmente, mais de três décadas depois de seu lançamento, ainda podem ser encontrados circuitos integrados baseados na linha 6500 aplicados em periféricos e sistemas embarcados, como microcontroladores da empresa Micronas GmbH e Sunplus Technology. Entretanto, seu maior legado consiste na fundação de conceitos utilizados em processadores modernos. A técnica de *pre-fetch*, que consiste na busca da próxima instrução no mesmo ciclo em que a instrução corrente está sendo concluída, contribuiu posteriormente para o desenvolvimento de algoritmos complexos de previsão de desvio assim como a técnica de *pipeline.*

A origem deste processador assim como o crescimento da empresa na qual foi desenvolvido reside no contexto tecnológico da época. O início da década de 1970 presenciou o surgimento das primeiras CPUs, circuitos complexos e com ampla funcionalidade, mas que possuíam um custo elevado, o que dificultava sua inserção em equipamentos de produção em massa. Diante dessa situação, o projeto da família 6500 teve início com foco voltado ao custo final do produto, o que definiu o limite de suas funcionalidades dentro das necessidades dos consumidores de forma a atingir o valor de 25 dólares por chip. Dessa forma surgiu o 6502, um processador com relação custo/benefício imbatível na época, superando inclusive o Motorola 6800 em capacidade computacional, o que resultou em sua utilização em dispositivos que marcaram época, sendo os principais destacados nas próximas subseções.

1.1 Atari 2600

A empresa Atari pode ser apontada como responsável pela popularização do uso de cartuchos ROM (*Read Only Memory*) para o armazenamento de jogos em consoles de vídeo game a partir do lançamento do Atari 2600, que pode ser visualizado na Figura 1. O uso de cartuchos possibilitava ao sistema a capacidade de rodar uma gama maior de jogos, ao contrário da primeira geração na qual o

código de programa era fixo e incluído no dispositivo. Tais características proporcionaram grande sucesso tanto para este dispositivo como para a segunda geração de sistemas de vídeo game na qual ele se insere.

Figura 1 – Console Atari 2600.

Outra característica relevante deste sistema é a ausência de um circuito dedicado para renderização da imagem a ser exibida pelo aparelho de televisão. Essa tarefa era então executada pelo processador, técnica conhecida como *CPU-Driven Graphics*, o que ocasionava uma carga excessiva neste circuito, já que deveria distribuir o tempo de execução entre a computação do programa e renderização da imagem (PURCARU, 2014). A Figura 2 ilustra a partir de esquemáticos simples a diferença entre essa técnica e o uso de um chip dedicado ao processamento gráfico.

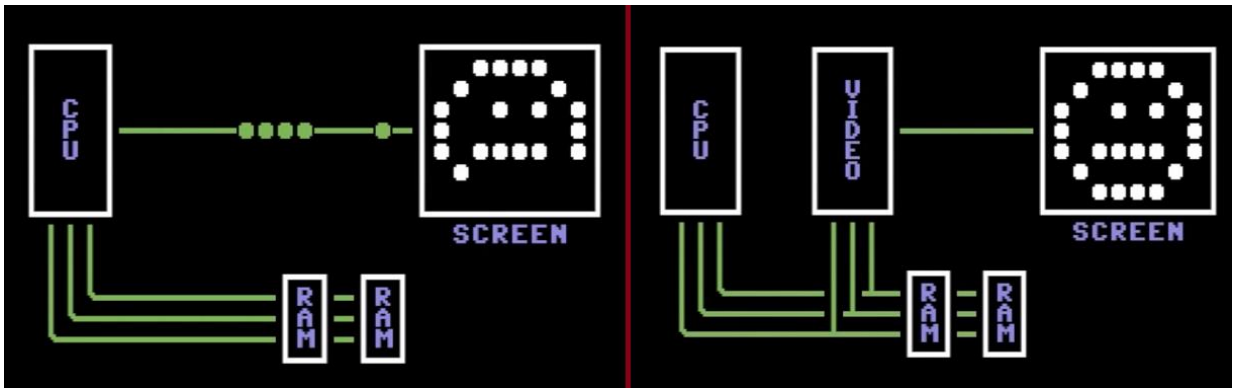

Figura 2 – *CPU-Driven Graphics* x Sistema com Processador Gráfico Dedicado.

1.2 Apple I e Apple II

Os computadores Apple tiveram origem no ano de 1976 com o desenvolvimento do Apple Computer 1 por Steve Wozniak. Esse computador contava com um processador 6502 operando a uma frequência de 1 MHz, 4 KB de RAM expansíveis a 8 KB além de suportar linguagem BASIC de forma a possibilitar a programação e execução de jogos. Inicialmente, tratava-se de um projeto

voltado para entusiastas visto que não havia teclado ou tela pré-instalados, ficando a cargo do usuário adicioná-los.

Diante da aceitação positiva do primeiro dispositivo desenvolvido pela empresa os esforços foram voltados a um sistema baseado no anterior, mas que fosse expansível e amigável ao usuário. Surgiu então o Apple II, ilustrado pela Figura 3, o primeiro microcomputador produzido em massa pela Apple Computer.

Figura 3 – Computador doméstico Apple II.

O dispositivo contava com o mesmo processador de seu predecessor, RAM expansível até 48 KB, interface de fitas cassete para carregamento de programas, linguagem Integer BASIC inserida na ROM, além de 8 slots de expansão, onde poderiam ser ligadas unidades de disquete e modems externos, entre outros periféricos (COMPUTER HISTORY MUSEUM, 1996). A versatilidade, expansibilidade e fácil utilização deste dispositivo garantiram um grande número de vendas, alavancando o início da empresa além de influenciar microcomputadores desenvolvidos posteriormente.

1.3 Commodore PET, VIC-20 e 64

Os computadores domésticos da empresa Commodore foram lançados no ano de 1977 com o modelo PET, um computador pessoal completo de grande sucesso nos mercados educacionais e que formou a base da linha de computadores de 8 bits desta empresa. Além do processador 6502 o dispositivo contava com memória expansível até 96 KB, sistema operacional Commodore BASIC, suporte para fitas cassete, unidades de disquete e tela monocromática.

O computador VIC-20 foi concebido no ano de 1980 como sucessor do PET tendo em vista um dispositivo de custo mais acessível, de forma a competir com o Apple II que vinha crescendo em número de vendas. O dispositivo possuía o mesmo processador 6502, 5 KB de RAM estática, chip gráfico colorido de propósito geral além de contar com um vasto suporte de periféricos que

expandiam as capacidades da máquina, como os *Datasettes*, que providenciavam armazenamento externo de baixo custo. Dessa forma, o VIC-20 foi o primeiro modelo de computador doméstico a ultrapassar a quantidade de 1 milhão de vendas.

Figura 4 – Commodore PET à esquerda e VIC-20 à direita.

O Commodore 64 surgiu como uma versão aprimorada do VIC-20, aproveitando inclusive o mesmo gabinete externo da versão anterior. Dentre as melhorias pode-se citar a presença de 64 KB de RAM, sistema operacional Commodore BASIC 2.0 além de som e gráficos superiores em função dos controladores SID (*Sound Interface Device*) e VIC-II (*Video Integrated Circuit*), ambos desenvolvidos pela MOS Technology e apontados como principais responsáveis pelo grande sucesso comercial do sistema (MATTHEWS, 2003).

1.4 Nintendo Entertainment System

Lançado em 1983 no Japão como *Family Computer*, também conhecido como Famicom, este console de vídeo game foi trazido para a américa no ano de 1985 em meio ao período conhecido como *video game crash*, momento de grande recessão da indústria de entretenimento eletrônico. Entre os fatores desta crise encontra-se o declínio da Atari devido à grande quantidade de jogos disponibilizados para o console 2600, mas que não possuíam o devido controle de qualidade, inundando o mercado com software que não agradou os consumidores.

Em uma atitude ousada, a empresa Nintendo, até então desconhecida no mercado norte americano e europeu, decidiu levar seu novo dispositivo para outros continentes diante de sua aceitação positiva no mercado japonês. A partir de uma remodelação estética, que pode ser observada na Figura 5, e alteração do nome de forma a adaptar o dispositivo ao novo mercado, em poucos anos tornou-se líder em vendas de sua geração ultrapassando a marca de 60 milhões de unidades, além de acabar com a crise que o precedeu.

Figura 5 – Famicom à esquerda e NES à direita.

Aponta-se como principal causa de seu sucesso a política da empresa relacionada à qualidade do software disponibilizado, seja ele desenvolvido pela própria Nintendo ou por desenvolvedores licenciados. Por conseguinte, jogos como Metroid, Legend of Zelda e Super Mario Bros e Megaman obtiveram grande sucesso de público, impulsionando as vendas do console.

Em termos de hardware, o dispositivo tinha como CPU o chip Ricoh 2A03, baseado no core do 6502 porém com o modo decimal desabilitado e que contava com o circuito de áudio integrado. O sistema possuía 2 KB de RAM e os cartuchos de jogos variavam em capacidades partindo de 8 KB até 1 MB. A empresa Ricoh além de desenvolver a CPU, foi requisitada para o desenvolvimento de um chip gráfico customizado (*Picture Processing Unit*) que possuía 2 KB de RAM dedicada ao vídeo e foi responsável pelo avanço gráfico se comparado aos concorrentes. Demais detalhes da arquitetura deste dispositivo serão abordados na seção 4.2.

1.5 Objetivos

Este projeto consiste na revisitação do processador 6502 através da linguagem de descrição de hardware VHDL. O objetivo é alcançar um circuito completamente sintetizável seguindo o fluxo de projeto FPGA ou ASIC. O circuito obtido será posteriormente agregado ao sistema já desenvolvido em linguagem Verilog (BENNET, 2012), o qual descreve o console NES, contendo a PPU, APU e demais circuitos do sistema, visando a prototipação em FPGA. A integração entre tais circuitos representará a validação final do microprocessador projetado visto que possibilitará a execução de jogos do console NES na plataforma de desenvolvimento Nexys 3.

Ao final do curso do projeto proposto, será obtido um sistema completo em único chip (SoC – *System-on-Chip*) com a mesma funcionalidade do console da empresa Nintendo reconstituído na arquitetura moderna do circuito integrado Spartan 6. A escolha de uma FPGA para o projeto justifica-se pela complexidade do hardware a ser prototipado, não sendo possível a utilização de micro controladores ou outras soluções de menor poder computacional.

O estudo aprofundado do funcionamento do processador será imprescindível, envolvendo seu conjunto de instruções, modos de endereçamento, comunicação interna e externa, componentes e

barramentos. Da mesma forma, o conhecimento acerca da plataforma escolhida é de extrema importância, visto que suas limitações e funcionalidades devem nortear a descrição VHDL do processador, para que sejam evitados problemas na etapa de prototipação.

Para tanto, será realizado o aprofundamento da documentação relacionada ao funcionamento e construção do processador, disponibilizando ao final todos os códigos desenvolvidos bem como documentos descritivos e esquemáticos em nível RTL. O foco da documentação construída será direcionado às micro operações realizadas pelo circuito, ou seja, dados como a quantidade de ciclos necessários para execução das instruções e as operações realizadas em cada ciclo serão levantados de maneira a ampliar entendimento de seu funcionamento.

O projeto proposto possui um caráter descritivo, visto que se busca o entendimento e revisitação de uma arquitetura marcante para a história da tecnologia e computação. Sua relevância encontra-se no aprofundamento da documentação já existente assim como no processo de adaptação de uma arquitetura antiga para sistemas modernos, como o Spartan 6. Essa adaptação envolverá alterações do projeto original de forma a alcançar um sistema completamente sintetizável na plataforma em questão.

Por envolver o console NES, dispositivo de entretenimento bastante conhecido e de grande contribuição para a evolução dos jogos eletrônicos, esse projeto deve atrair a atenção não só de indivíduos com conhecimento técnico acerca do tema, mas também de outras pessoas que tiveram algum contato com o console.

2 ARQUITETURA

O projeto descrito por este relatório iniciou-se com o estudo aprofundado da arquitetura do processador 6502, tendo como principais referências os documentos *Programming Manual* (MOS TECHNOLOGY, 1976), *Hardware Manual* (MOS TECHNOLOGY, 1976) e *Architecture 6502* (KOWALSKI, 2003). Os dois primeiros documentos tratam-se do manual de programação e manual de hardware que foram disponibilizados pela própria empresa que o desenvolveu com o objetivo de esclarecer as considerações básicas para utilização do processador. O terceiro texto citado apresenta um resumo da arquitetura e é disponibilizado juntamente com o simulador e montador de código desenvolvido por pesquisadores deste processador. Os principais aspectos da arquitetura serão abordados nas próximas seções.

2.1 Registradores

Por possuir um barramento de endereços de 16 bits, o processador 6502 é capaz de endereçar 65536 bytes de memória, sendo que os endereços são representados em caracteres hexadecimais e variam entre 0x0000 e 0xFFFF. Por esse motivo, os registradores de endereço e contador de programa possuem largura de 16 bits e são constituídos por dois registradores de 8 bits, divididos em parte baixa e parte alta, de forma a coincidir com a largura do barramento de memória. O ordenamento dos bytes na memória é *little endian*, no qual o byte menos significativo é armazenado em um endereço de memória menor que os demais bytes. A Tabela 1 ilustra o mapeamento de memória comumente utilizado para esse processador.

Tabela 1 – Mapeamento de memória utilizado nos processadores 6502.

Intervalo de Endereços	Função
0x0000 – 0x00FF	Página zero.
0x0100 – 0x01FF	Armazenamento da pilha.
0x0200 – 0x3FFF	Memória de acesso randômico (RAM).
0x4000 – 0x7FFF	Operações de entrada e saída de dados.
0x8000 – 0xFFF9	Armazenamento da memória de programa, normalmente somente leitura (ROM).
0xFFFA – 0xFFFB	Vetor da interrupção NMI.
0xFFFC – 0xFFFD	Vetor da interrupção RESET.
0xFFFE – 0xFFFF	Vetor das interrupções IRQ e BREAK.

A família 6500 é constituída de processadores de 8 bits, ou seja, somente 8 bits de dados são transferidos ou manipulados durante um ciclo de relógio. Portanto, os 5 registradores destinados ao armazenamento temporário de dados possuem essa capacidade, sendo eles:

- Registrador acumulador **A**: utilizado principalmente para armazenamento do resultado de operações lógicas e aritméticas. Também pode ser utilizado como armazenamento temporário durante a movimentação de valores de um local para o outro na memória;
- Registradores de índice **X** e **Y**: Como o próprio nome indica, normalmente utilizados para armazenamento de contadores ou deslocamentos para acesso à memória;
- Ponteiro de pilha **S**: Utilizado para armazenamento do byte menos significativo da próxima posição livre da pilha, que está contida nos 256 bytes localizados entre os endereços 0x0100 e 0x01FF. A operação de inserção de dado na pilha faz com que o registrador S seja decrementado, assim como a remoção de dado resulta no incremento do mesmo registrador, fazendo com que a pilha aumente para baixo e inicie no endereço 0x01FF;
- Registrador de status **P**: Trata-se do registrador que armazena as *flags* de estado do processador, sendo composto pelos bits de *Carry* (C), *Zero* (Z), *Interrupt Disable* (I), *Decimal Mode* (D), *Break Command* (B), *Overflow Flag* (V) e *Negative Flag* (N). A utilização e alteração destas *flags* será abordada de forma mais detalhada nas Seções 2.2 e 2.3;

2.2 Modos de Endereçamento

A partir da apresentação dos registradores e introdução do mapeamento de memória, pode-se realizar a abordagem dos modos de endereçamento disponíveis no microprocessador em questão, os quais representam as mais variadas formas de acesso à memória. A instrução LDA, que tem como função carregar dados no registrador acumulador, será utilizada de forma a exemplificar a utilização de cada modo.

2.2.1 Absoluto

As instruções em modo absoluto são constituídas por 3 bytes, sendo que o primeiro se refere ao código que identifica a instrução e os demais representam o endereço de acesso à memória. Por conseguinte, qualquer posição de memória pode ser acessada a partir do modo absoluto.

Exemplo de utilização em código *assembly*:

```
LDA $C000   ; carrega acumulador com o valor contido no endereço C000.
```

2.2.2 Página Zero

A utilização do modo página zero implica instruções de tamanho 2 bytes. O primeiro byte identifica a instrução a ser executada e o segundo é a parte baixa do endereço de acesso à memória,

sendo que a parte alta é assumida como zero. Dessa forma, somente endereços contidos na página zero podem ser acessados.

O objetivo deste modo é proporcionar ao programador a capacidade de economizar memória e agilizar o tempo de execução, visto que a instrução é 1 byte menor que sua equivalente em modo absoluto.

Exemplo de utilização em código *assembly*:

```
LDA $C0      ; carrega acumulador com o valor contido no endereço 0x00C0.
```

2.2.3 Imediato

O modo imediato difere dos descritos previamente pelo fato de que o operando não é um endereço, mas sim um dado, configurando-se como a maneira mais simples de manipulação de constantes. A instrução em modo imediato tem tamanho total de 2 bytes, sendo eles o código identificador e o valor a ser armazenado, além de possuir um tempo de execução curto, visto que não há acesso à memória.

Exemplo de utilização em código *assembly*:

```
LDA #$55     ; carrega o acumulador com o valor 0x55.
```

2.2.4 Implícito

As instruções em modo implícito possuem um código identificador que define operações internas no processador, não necessitando de um operando. Dessa forma, tais instruções possuem tamanho de 1 byte. Para exemplificar o modo, será utilizada a instrução CLC, que tem como função limpar a *flag* do registrador de estado do processador que indica *carry*.

Exemplo de utilização em código *assembly*:

```
CLC    ; define o valor da carry flag como zero.
```

2.2.5 Acumulador

Possibilita a manipulação direta do valor armazenado no registrador acumulador, sendo utilizada apenas pelo grupo de instruções destinado ao deslocamento e rotação de bits, a ser detalhado na Seção 2.3. A instrução nesse modo de endereçamento também possui tamanho de 1 byte. Para exemplificar o modo, será utilizada a instrução ASL, que tem como função deslocar o valor manipulado em um bit para a esquerda.

Exemplo de utilização em código *assembly*:

```
ASL    ; desloca conteúdo do acumulador em um bit para a esquerda.
```

2.2.6 Absoluto Indexado por X ou por Y

Assim como no modo absoluto, a instrução tem tamanho de 3 bytes e o operando é um valor de 2 bytes que corresponde a um endereço de memória. Entretanto, o endereço de acesso à memória é obtido a partir do operando informado na instrução adicionado ao valor armazenado no registrador de índice X ou Y.

Exemplo de utilização em código *assembly*:

LDA $C000, X ; carrega o acumulador com o valor contido no endereço 0xC000 + X.

2.2.7 Página Zero Indexado por X ou por Y

Assim como no modo página zero, o operando neste caso é um endereço base de 1 byte e a instrução totaliza 2 bytes. O endereço base é somado ao valor contido em X ou Y para obtenção do endereço de acesso à memória, da mesma forma que no modo absoluto indexado. Porém, neste modo não há tratamento de cruzamento de página, ou seja, fica a cargo do programador garantir que o endereço de acesso à memória resultante não exceda os limites da página zero.

Exemplo de utilização em código *assembly*:

LDA $C0, X ; carrega o acumulador com o valor contido no endereço 0x00C0 + X.

2.2.8 Absoluto Indireto

Assim como no modo absoluto, o operando é um endereço de memória de 2 bytes. Entretanto, este endereço contém o byte menos significativo do endereço final de destino, já o byte mais significativo encontra-se no endereço seguinte. Este modo é utilizado apenas pela instrução JMP e sua execução é ilustrada na Figura 6.

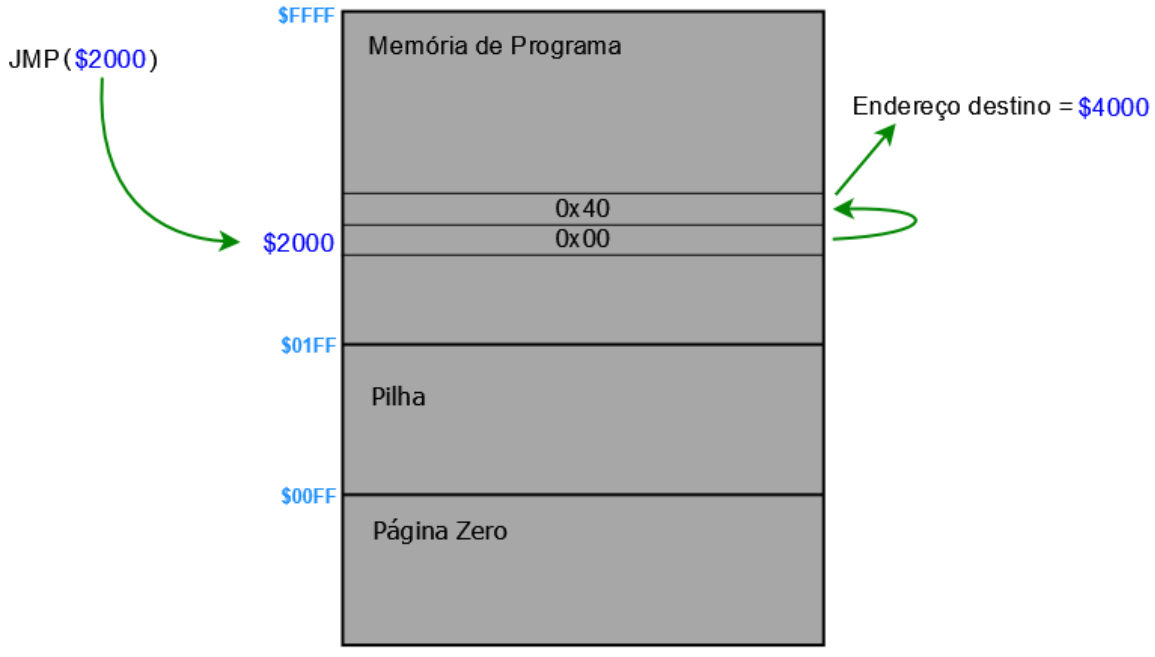

Figura 6 – Endereçamento absoluto indireto.

2.2.9 Indexado Indireto

Neste modo, o operando é 1 byte de endereço da página zero totalizando 2 bytes de tamanho de instrução. O operando é então adicionado ao valor contido no registrador X, resultando em um local de memória que deve conter o byte menos significativo de um endereço de 2 bytes. O byte mais significativo do endereço de acesso à memória é obtido no endereço seguinte. A Figura 7 ilustra esse processo de forma mais clara.

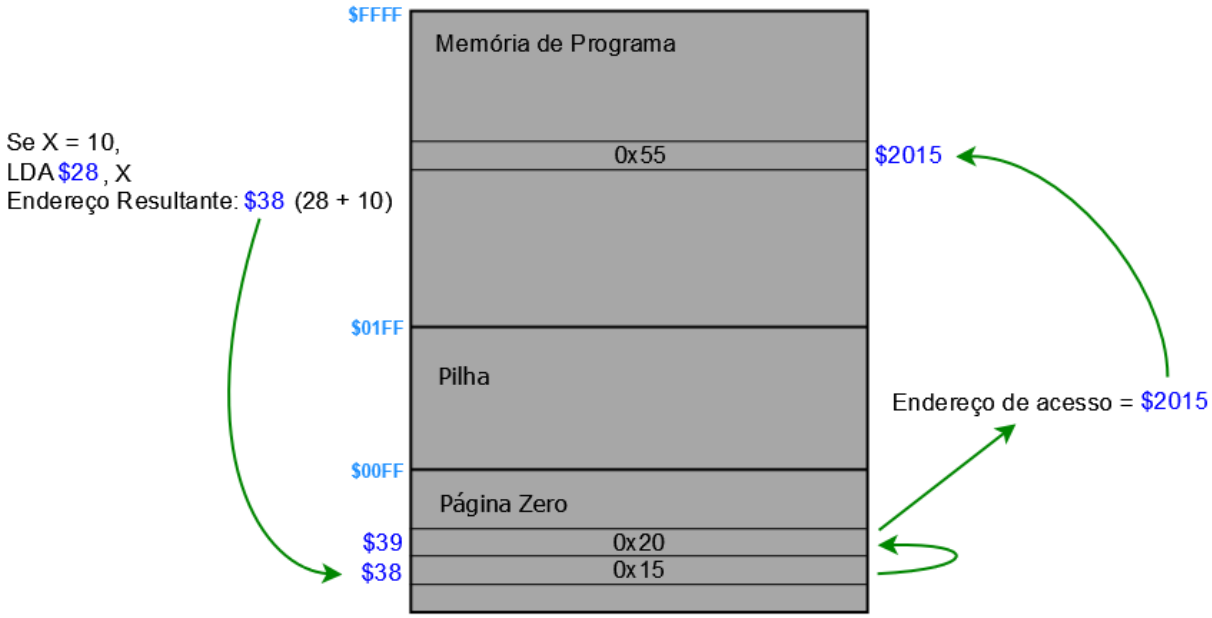

Figura 7 – Endereçamento indexado indireto.

Ao final da execução da instrução LDA utilizada como exemplo, o valor 0x55 será armazenado no registrador acumulador.

2.2.10 Indireto Indexado

Semelhante ao modo indexado indireto, porém neste caso o registrador de índice utilizado é o Y. Outra diferença está na ordem das operações, visto que neste modo primeiramente se obtém o endereço de 2 bytes para depois ser somado ao conteúdo de Y, gerando o endereço de acesso à memória. A Figura 8 ilustra o processo de maneira mais clara, sendo que ao final da execução da instrução, o valor 0xFF será armazenado no registrador acumulador.

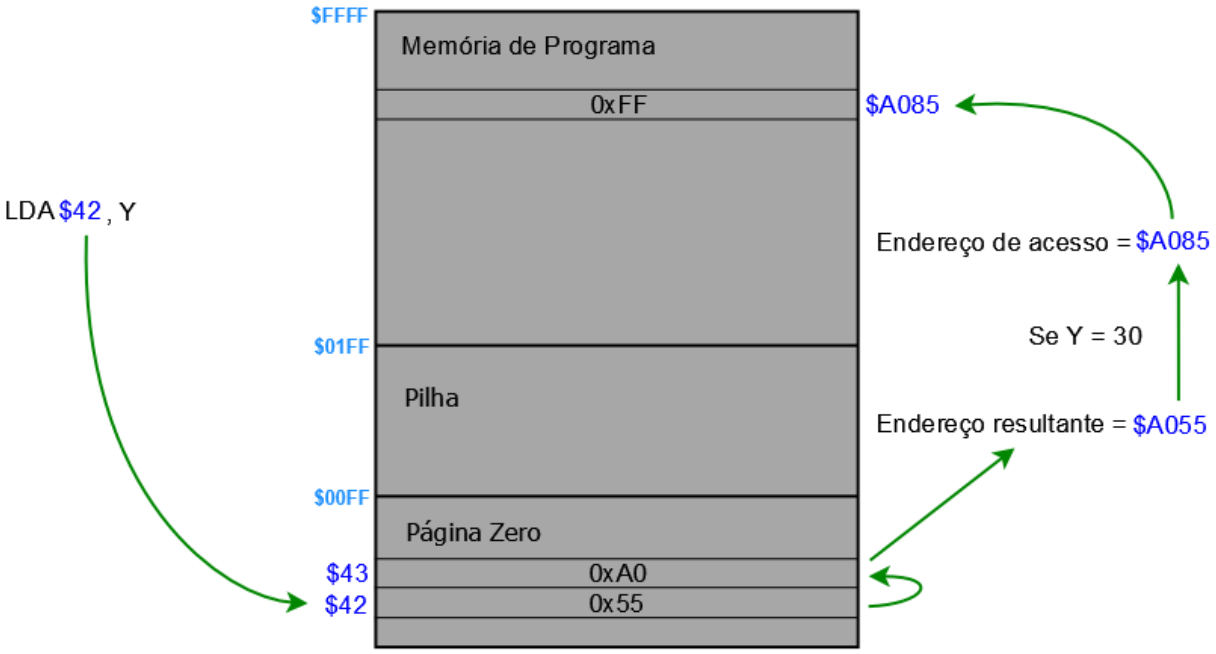

Figura 8 – Endereçamento indireto indexado.

2.2.11 Relativo

O modo relativo é utilizado apenas pelas instruções de desvio (*branches*) e possui tamanho de 2 bytes. O operando deste modo é um valor de 1 byte em complemento de 2, o qual representa um deslocamento entre -128 e +127 a ser somado ao endereço da próxima instrução. Para exemplificação do modo relativo será utilizada a instrução BEQ, a qual causa um salto no programa caso a *flag* Z esteja definida com o valor 1.

Exemplo de utilização em código *assembly*:

```
BEQ $F5        ; F5 equivale a -11 em decimal.
```

No exemplo anterior, o contador de programa terá seu valor atual subtraído em 11 unidades e o programa continuará a partir endereço resultante dessa operação, caso a condição seja satisfeita. De forma a facilitar seu uso, a utilização de *labels* para identificar trechos de código é suportada pelos montadores de código.

A análise dos modos de endereçamentos disponíveis na arquitetura do microprocessador 6502 revela a preocupação dos desenvolvedores no que diz respeito à eficiência, já que modos como o de página zero proporcionam economia de memória aliada ao ganho de desempenho. Ao mesmo tempo foram disponibilizados vários modos distintos de forma a flexibilizar as opções do programador durante a construção do software.

2.3 Conjunto de Instruções

O conjunto de instruções do processador é dividido inicialmente em três grandes grupos relacionados à sua funcionalidade básica. O primeiro deles é constituído das instruções de propósito geral e que apresentam grande flexibilidade quanto aos modos de endereçamento disponíveis, como por exemplo as instruções de leitura, escrita e soma de dados. O segundo grande grupo consiste nas que englobam as operações de leitura, modificação e escrita de dados em uma só instrução, como nos deslocamentos, rotações e incrementos de valores. O terceiro e último grupo refere-se às demais instruções, como as de operações com a pilha e comparações com registradores X e Y, as quais não podem ser atribuídas aos dois primeiros grupos.

Essa divisão apresentada pelo manual de programação do processador é bastante genérica, podendo ser aprofundada a partir da criação de subgrupos de instruções que desempenham funções semelhantes, conforme pode ser observado nas próximas seções.

2.3.1 Leitura e Escrita

O primeiro subgrupo a ser analisado é o que tem como função ler e escrever dados na memória acessível pelo processador, caracterizando-se como instruções do primeiro grande grupo. A Tabela 2 apresenta as instruções disponíveis assim como a descrição da operação realizada por cada uma delas e as *flags* afetadas.

Tabela 2 – Instruções de leitura e escrita.

Mnemônico	Operação	*Flags*
LDA	Carrega acumulador	N, Z
LDX	Carrega registrador X	N, Z
LDY	Carrega registrador Y	N, Z
STA	Armazena acumulador na memória	-
STX	Armazena registrador X na memória	-
STY	Armazena registrador Y na memória	-

Os modos de endereçamento disponíveis para as instruções LDA e STA são absoluto, página zero, imediato, absoluto indexado, página zero indexado por X, indexado indireto e indireto indexado. Para as instruções que manipulam valores de X e Y os modos indexado indireto e indireto indexado não são suportados e para as que realizam escrita de dados não há suporte para o modo absoluto indexado.

2.3.2 Aritméticas

As instruções aritméticas também estão inseridas no primeiro grande grupo e, como o próprio nome indica, têm como objetivo a realização de operações aritméticas entre valores armazenados na

memória, o valor atual do acumulador e o valor atual da *flag carry*. Deve-se destacar que o resultado de tais operações sobrescreve o valor do acumulador, não sendo armazenado na memória. O grupo conta com suporte para os modos absoluto, página zero, imediato, absoluto indexado, página zero indexado, indireto indexado e indexado indireto.

Tabela 3 – Instruções aritméticas.

Mnemônico	Operação	*Flags*
ADC	Acumulador + valor em memória + *carry*	N, V, Z, C
SBC	Acumulador – valor em memória – *carry*	N, V, Z, C

2.3.3 Incremento e Decremento

Este grupo apresenta instruções capazes de realizar o incremento ou decremento em uma unidade dos valores contidos nos registradores X e Y ou de dados armazenados na memória, conforme ilustrado na Tabela 4. As instruções INC e DEC apresentam suporte para os modos absoluto, página zero, absoluto indexado e página zero indexado além de caracterizarem-se como pertencentes ao segundo grande grupo de instruções. Já as instruções INX, INY, DEX e DEY utilizam somente o modo implícito e inserem-se no primeiro grande grupo.

Tabela 4 – Instruções de incremento e decremento.

Mnemônico	Operação	*Flags*
INC	Incrementa dado na memória	N, Z
INX	Incrementa registrador X	N, Z
INY	Incrementa registrador Y	N, Z
DEC	Decrementa dado na memória	N, Z
DEX	Decrementa registrador X	N, Z
DEY	Decrementa registrador Y	N, Z

2.3.4 Transferência de Registradores

Grupo destinado a dar suporte à transferência de valores entre os registradores internos do processador, inserindo-se no terceiro grande grupo de instruções. Por definirem operações internas, o único modo de endereçamento disponível é o implícito.

Tabela 5 – Instruções de transferência de registradores.

Mnemônico	Operação	*Flags*
TAX	Transfere A para X	N, Z
TAY	Transfere A para Y	N, Z
TXA	Transfere X para A	N, Z
TYA	Transfere Y para A	N, Z

2.3.5 Lógicas

Grupo de instruções semelhante às aritméticas, porém com propósito de dar ao programador o suporte para realização das operações lógicas básicas entre um dado armazenado na memória e o acumulador. Novamente, trata-se de instruções do primeiro grande grupo e os modos de endereçamento disponíveis são absoluto, página zero, imediato, absoluto indexado, página zero indexado, indexado indireto e indireto indexado.

Tabela 6 – Instruções lógicas.

Mnemônico	Operação	*Flags*
AND	Acumulador *AND* valor em memória	N, Z
EOR	Acumulador *OR* valor em memória	N, Z
ORA	Acumulador *OR* valor em memória	N, Z

2.3.6 Comparação e Teste de Bit

Grupo destinado à comparação do conteúdo da memória com o valor contido nos registradores internos do processador. Deve-se destacar que o resultado da comparação tem reflexo somente na alteração do valor das *flags* de estado, não sendo armazenado em registradores ou escrito na memória. Tais instruções são normalmente utilizadas antes de desvios condicionais para o teste de valores.

Tabela 7 – Instruções de comparação e teste de bit.

Mnemônico	Operação	*Flags*
CMP	Compara acumulador com valor em memória	N, Z, C
CPX	Compara registrador X com valor em memória	N, Z, C
CPY	Compara registrador Y com valor em memória	N, Z, C
BIT	Testa valor de bit	N, V, Z

A instrução CMP é a única que conta com suporte para todos os principais modos de endereçamento, já CPX e CPY suportam somente os modos absoluto, página zero e imediato. A instrução BIT suporta apenas os modos absoluto e página zero.

A notação simbólica para as instruções de comparação é A, X ou Y – Memória, ou seja, por meio de uma subtração é realizada a comparação dos valores. Caso sejam iguais, a *flag* Z recebe o valor 1, caso contrário recebe 0. A *flag* N é alterada conforme o bit 7 do resultado da operação de subtração, já a *flag* C recebe 1 quando o valor em memória é menor ou igual ao do registrador ao qual está sendo comparado. Se o valor em memória for maior, a *flag* C recebe 0.

A instrução BIT tem como função proporcionar a capacidade de determinar a condição individual de um determinado bit. Dessa forma, a operação realizada é a função lógica AND entre um valor armazenado em memória e o valor atual do registrador A. Essa operação poderia ser realizada com a própria instrução AND, entretanto essa abordagem alteraria o valor armazenado no acumulador. Utilizando a instrução BIT, o teste pode ter seu resultado avaliado a partir de instruções que testam os valores das *flags*, sem que o valor de A seja alterado.

2.3.7 Deslocamento e Rotação

Este grupo propicia suporte às instruções de deslocamento e rotação de bits, algo que contribui para operações seriais onde um bit é processado a cada ciclo de relógio, assim como facilita rotinas de multiplicação e divisão de valores. Os modos de endereçamento disponíveis são absoluto, página zero, absoluto indexado, página zero indexado e acumulador, sendo que o último é exclusivo para o grupo em questão.

Tabela 8 – Instruções de deslocamento e rotação.

Mnemônico	Operação	*Flags*
ASL	Deslocamento esquerda	N, Z, C
LSR	Deslocamento para direita	N, Z, C
ROL	Rotação para esquerda	N, Z, C
ROR	Rotação para direita	N, Z, C

A instrução ASL realiza o deslocamento em 1 bit para a esquerda do valor armazenado no acumulador ou em memória. O bit 0 do valor resultante é sempre definido como 0 e o bit 7 do valor de entrada é enviado para a *flag* C, sendo que a *flag* N é afetada conforme o valor do bit 6 da entrada. No caso de LSR, por se tratar de um deslocamento em 1 bit para a direita, o bit 7 do resultado é sempre definido como 0 e o bit 0 da entrada é enviado para a *flag* C. Dessa forma, a *flag* N é sempre definida como 0.

A instrução ROL causa a rotação da entrada para a esquerda em 1 bit, sendo que o *carry* de entrada é armazenado no bit 0 do resultado e o bit 7 afeta *carry out* da operação e a *flag* N é modificada conforme o valor do bit 6 da entrada. Por último, a instrução ROR resulta na rotação da entrada para a direita em 1 bit. Dessa maneira, o *carry* de entrada é armazenado no bit 7 do resultado e o bit 0 de entrada define o *carry out* da operação.

2.3.8 Salto e Desvio Condicional

As instruções contidas neste grupo objetivam alterar a sequência de execução do programa a partir da modificação do valor do contador de programa, sendo este o único registrador alterado durante sua execução. A Tabela 9 ilustra as operações básicas e condições de cada salto.

Tabela 9 – Instruções de salto e desvio condicional.

Mnemônico	Operação
JMP	Salto incondicional
BCC	Salta se *carry* = 0
BCS	Salta se *carry* = 1
BEQ	Salta se *zero* = 1
BMI	Salta se *negative* = 1
BNE	Salta se *zero* = 0
BPL	Salta se *negative* = 0
BVC	Salta se *overflow* = 0
BVS	Salta se *overflow* = 1

A instrução JMP é a única do grupo em que é possível utilizar dois modos distintos de endereçamento, sendo eles o absoluto e o absoluto indireto, além de tratar-se de um salto incondicional. As demais instruções utilizam modo relativo para calcular o endereço de destino e configuram-se como saltos condicionais, onde os valores das *flags* definem se o programa terá sua sequência alterada.

2.3.9 Pilha

Destinadas à modificação do conteúdo do ponteiro de pilha e para escrita ou leitura de dados na região de memória por ele apontada. São caracterizadas como instruções do grupo três e possuem somente suporte ao modo implícito de endereçamento. Destaca-se o fato de que o ponteiro de pilha sempre aponta para um endereço vazio, onde o próximo dado escrito será salvo. Por esse motivo, as instruções de leitura de dados efetuam o incremento do ponteiro um ciclo antes de realizar a leitura do dado.

Tabela 10 – Instruções de manipulação de pilha.

Mnemônico	Operação	*Flags*
TSX	Transfere S para X	N, Z
TXS	Transfere X para S	-
PHA	Armazena A na pilha	-
PHP	Armazena P na pilha	-
PLA	Lê A da pilha	N, Z
PLP	Lê P da pilha	Todas

2.3.10 Modificação de *Flags* de Estado

Trata-se de instruções executadas somente em modo implícito, visto que objetivam apenas a alteração dos bits contidos no registrador de estado. A Tabela 11 apresenta a função de cada instrução do grupo, sendo que a *flag* decimal definida com valor 1 faz com que o processador realize as

operações na unidade lógico aritmética considerando os valores em modo BCD (*Binary Coded Decimal*).

Tabela 11 – Instruções de modificação de *flags.*

Mnemônico	Operação	*Flags*
CLC	*carry* <= 0	C
CLD	*decimal* <= 0	D
CLI	*interrupt disable* <= 0	I
CLV	*overflow* <= 0	V
SEC	*carry* <= 1	C
SED	*decimal* <= 1	D
SEI	*interrupt disable* <= 1	I

2.3.11 Sub-rotina e Interrupção

O grupo de sub-rotina e interrupção destina-se ao tratamento da sequência necessária para a realização destas operações. A chamada de sub-rotina assemelha-se ao salto incondicional, entretanto o valor do contador de programa é escrito na pilha antes do salto para que seja possível o retorno à sequência normal do programa após sua conclusão. O tratamento de interrupções possui o mesmo comportamento com adição da escrita do valor atual do registrador de estado na pilha.

Tabela 12 – Instruções de sub-rotina e interrupção.

Mnemônico	Operação	*Flags*
JSR	Salta para sub-rotina	-
RTS	Retorna de sub-rotina	-
BRK	Força interrupção	B
RTI	Retorna de interrupção	Todas
NOP	Sem operação	-

O modo de endereçamento implícito é utilizado por todas instruções, com exceção da JSR que opera em modo absoluto. Conforme pode-se observar na Tabela 12, as instruções RTS e RTI proporcionam o retorno do tratamento de sub-rotinas e interrupções, respectivamente.

A instrução BRK é normalmente utilizada pelos programadores como ferramenta de depuração, visto que se trata de uma interrupção forçada em software. Tal instrução realiza o salvamento de contexto no momento em que é executada, armazenando os valores atuais do contador de programa e registrador de estado na pilha, sendo que o armazenamento dos demais registradores deve ser realizado pelo programador.

O processador 6502 possui 3 linhas de interrupção de hardware denominadas IRQ, RESET e NMI, as quais são ativadas em nível lógico baixo e fazem com que o contador de programa receba o valor contido no vetor de interrupção correspondente, conforme a Tabela 1. O tratamento de tais interrupções segue uma sequência de operações semelhantes às da instrução BRK, tendo como única

diferença o fato de não alterarem o valor de nenhuma *flag*. A interrupção gerada pela linha IRQ é a única que pode ser mascarada pelo programador através da instrução SEI, a qual define a *flag interrupt disable* em 1, conforme citado na Seção 2.3.10.

3 ORGANIZAÇÃO

A documentação oficial disponível contribuiu para o princípio dos estudos acerca do processador 6502, entretanto o grande avanço em seu entendimento ocorreu quando alguns pesquisadores utilizaram a engenharia reversa com o objetivo de esclarecer certos detalhes sobre sua arquitetura e organização, surgindo então os documentos não oficiais. Um dos trabalhos mais importantes foi construído a partir de fotos tiradas com microscópios eletrônicos do próprio *die* de um 6502, o que possibilitou a construção de um diagrama completo em nível de transistores (BEREGNYEI, 2001). Esse esquemático alavancou as pesquisas relacionadas e incentivou mais pesquisadores a voltarem seus esforços para contribuir com a exploração.

A partir da técnica de vetorização aplicada sobre as fotos obtidas por microscópios, o projeto *The Visual 6502* (JAMES; SILVERMAN; SPITTLES, 2009) foi desenvolvido, onde é possível visualizar o *layout* do processador durante a execução de programas e as respectivas porções do circuito ativadas em cada etapa. A Figura 9 ilustra o resultado obtido pelo projeto.

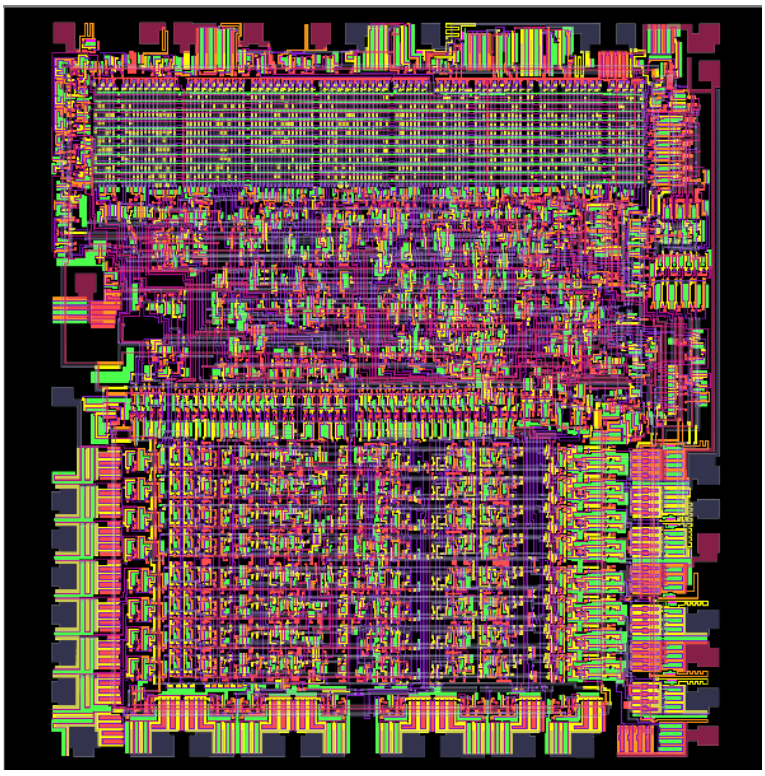

Figura 9 – *Layout* obtido a partir de imagens de microscópios eletrônicos.

Juntamente com conhecimentos adquiridos nos projetos anteriormente citados, o diagrama de blocos apresentado pela Figura 10 foi divulgado em 1995 (HANSON, 1995). Este diagrama representa uma visão geral do processador apresentada de maneira mais detalhada se comparada aos diagramas presentes nos documentos originais. Observa-se a separação em duas grandes áreas, sendo

que a da esquerda representa os componentes do bloco de controle, já a da direita contém o bloco de dados. A estrutura apresentada por esse diagrama foi utilizada como ponto de partida para implementação proposta por este projeto.

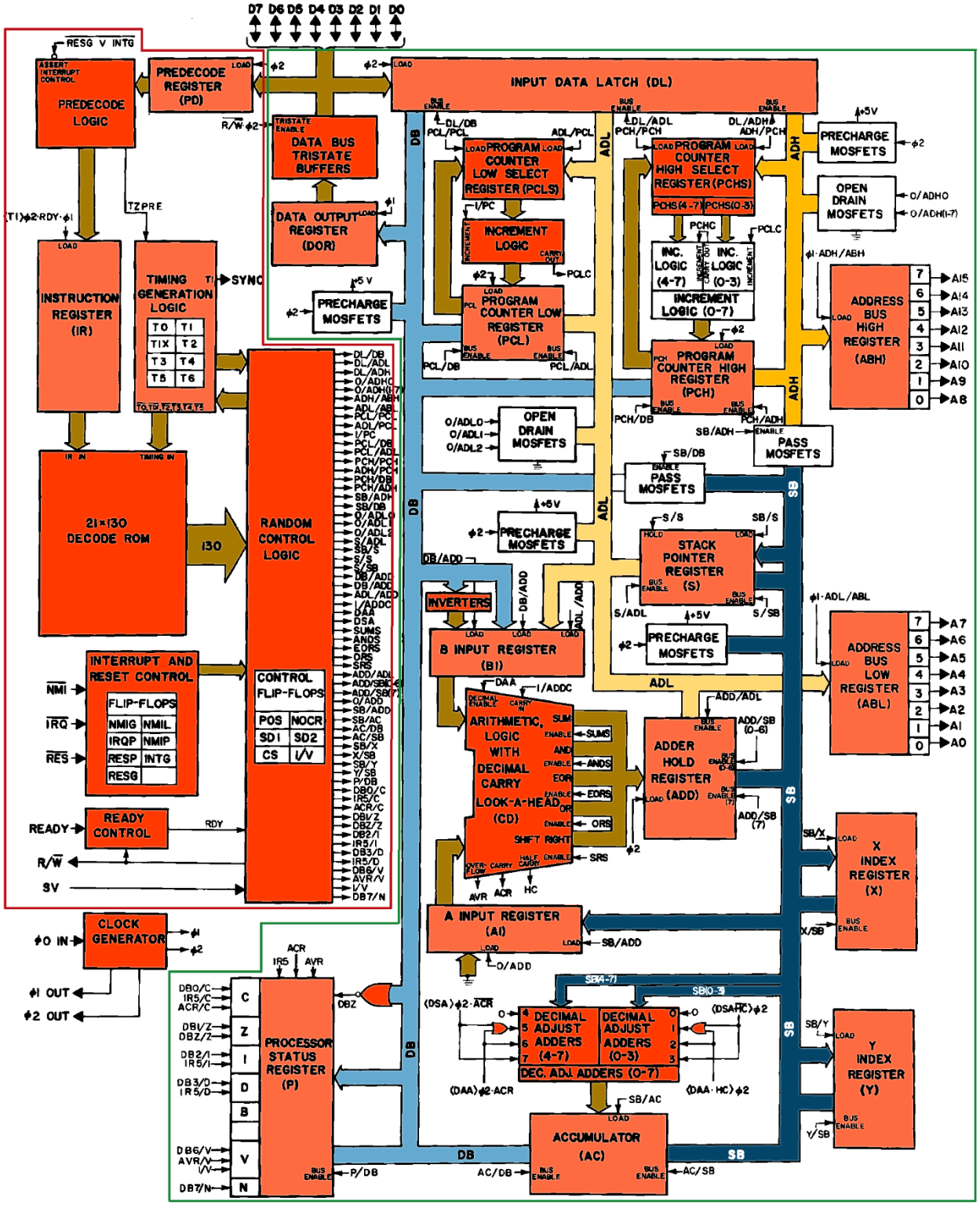

Figura 10 – Diagrama de blocos do processador 6502.

3.1 Bloco de Dados

A partir do estudo da documentação e familiarização com a arquitetura do processador, foi possível a elaboração da versão do bloco de dados a ser utilizada no projeto com auxílio do software Logisim (BURCH, 2005). Iniciou-se a implementação por esta etapa visto que, diante da estrutura do bloco de dados, puderam ser definidos os sinais de controle necessários bem como o caminho de dados para cada etapa de execução das instruções, facilitando o desenvolvimento do bloco de controle e a definição da primeira versão das micro operações realizadas em cada ciclo.

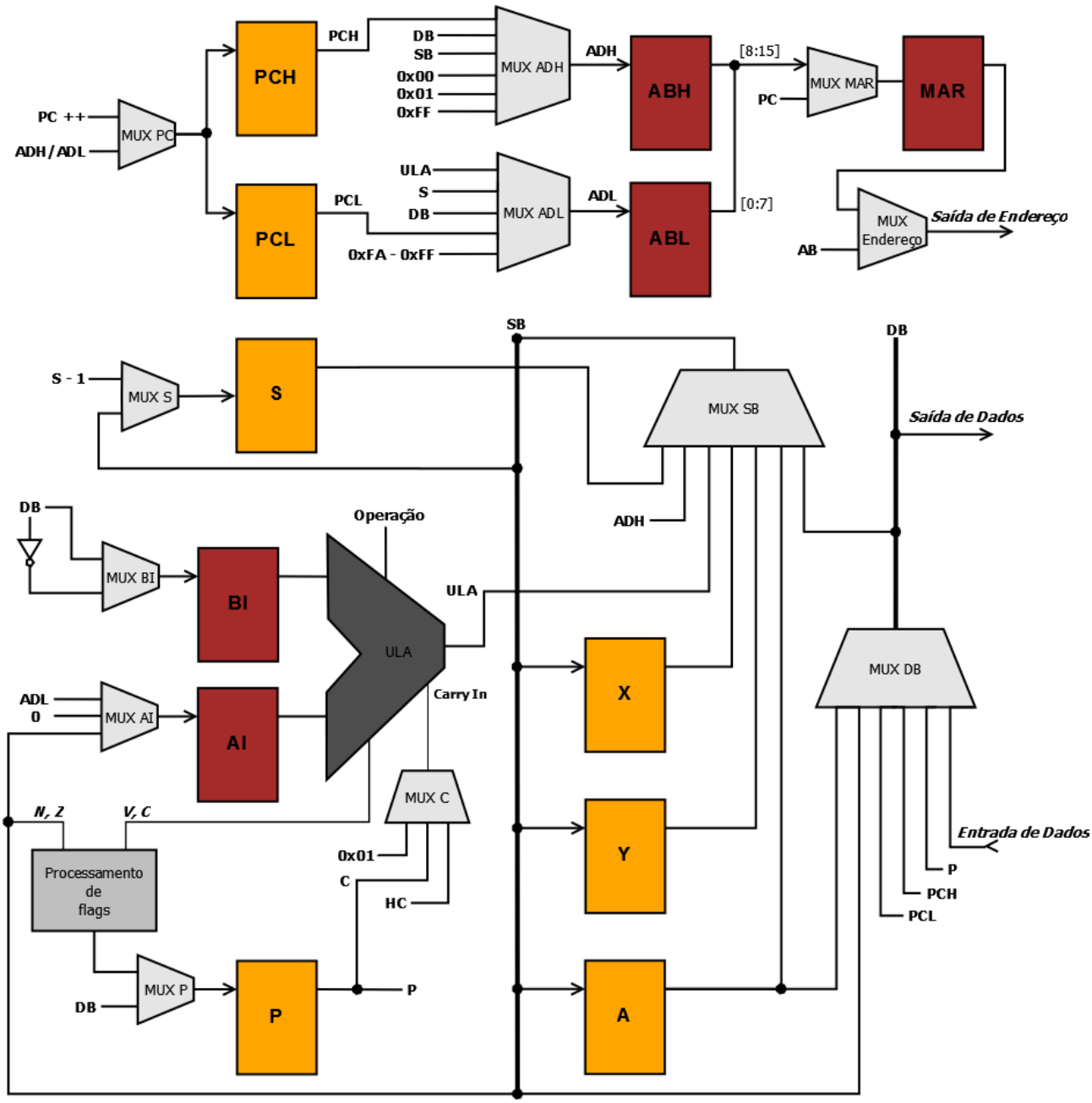

Figura 11 – Diagrama do bloco de dados utilizado no projeto.

O diagrama da Figura 11 consiste em uma versão simplificada do que foi construído com o software Logisim, contendo apenas os registradores, multiplexadores e barramentos presentes no bloco de dados de forma a facilitar sua análise e entendimento. Observa-se na estrutura do diagrama a presença dos registradores que foram abordados na Seção 2.1, além de mais cinco registradores que terão seu funcionamento esclarecido na presente seção por se tratarem unidades de suporte às operações internas do processador. Destaca-se também que a execução detalhada de cada instrução está disponível na documentação complementar desenvolvida.

Iniciando a análise do bloco de dados pela parte superior da Figura 11 observa-se os registradores PCL e PCH, os quais representam a parte baixa e parte alta do contador de programa. O multiplexador inserido na entrada desses registradores tem como função selecionar se o valor a ser escrito será o incremento de PC, como por exemplo no caso do ciclo de busca da próxima instrução, ou o valor de endereço de destino de um salto ou desvio de programa. O bloco somador que realiza a operação de incremento do valor armazenado em PC foi suprimido do esquemático exposto de forma a torná-lo mais legível, sendo representado pela entrada PC ++.

Os barramentos ADL e ADH correspondem às entradas dos registradores de endereçamento de memória ABL e ABH (AB), respectivamente, os quais armazenam o endereço atual de acesso à memória. Dessa forma, em caso de execução de uma instrução que cause a mudança na sequência normal do programa, o endereço de destino é escrito simultaneamente em PC e nos registradores ABL e ABH, de forma que ao final do salto seja possível o prosseguimento correto da execução do programa. No caso da sequência normal de execução, o valor de PC é transferido para AB através dos multiplexadores ADL e ADH no estado de busca de instrução.

O registrador MAR, posicionado à direita dos registradores ABL e ABH, tem como função auxiliar o armazenamento de endereços de acesso a memória. Um exemplo de utilização desse registrador é a execução do modo de endereçamento Absoluto. Inicialmente, a busca e decodificação da instrução é realizada através de MAR, sendo que durante a decodificação o endereço de acesso à memória efetivo é copiado para AB, enquanto MAR é utilizado para a sequência de programa. Ao final, o dado é obtido da memória a partir do endereçamento realizado por AB.

A primeira entrada de MUX ADL consiste na saída da Unidade Lógico Aritmética (ULA), tornando possível a escrita dos endereços de acesso à memória que são obtidos através de operações aritméticas, como é o caso dos modos de endereçamento indexados. A segunda entrada do multiplexador é referente ao valor armazenado no ponteiro de pilha (S) e é utilizada por instruções que acessam essa região de memória, como PHP e PLA. Na sequência, observa-se o sinal DB, o qual representa o principal barramento do bloco de dados por proporcionar tanto a leitura quanto a escrita na memória. Através dessa entrada pode-se escrever os endereços de acesso à memória que estão

contidos na instrução, como no caso do modo de endereçamento absoluto. O barramento DB também está conectado ao MUX ADH para que a parte alta do endereço também seja escrita nesses casos.

O barramento SB, que está conectado ao MUX ADH, consiste no barramento auxiliar do bloco de dados e está conectado ao registrador ABH de forma a tratar casos em que há cruzamento de página durante a manipulação do endereço. Um exemplo simples para ilustrar esse fenômeno pode ser construído com o modo de endereçamento absoluto indexado por X. Suponha que o endereço indicado pela instrução seja 0x02FF e que o valor contido em X seja 0x01. O endereço de acesso à memória é obtido a partir da soma de 0xFF com 0x01 na unidade lógico aritmética, resultando no valor 0x00 e em *carry out*, ou seja, deve-se somar 1 ao valor de ABH para que o endereço final resulte em 0x0300. Para tal, o valor de ABH é incrementado no próximo ciclo e enviado ao registrador através do barramento SB. Isso se deve ao fato de que a largura do barramento de endereços é 16 bits e a ULA operar apenas sobre valores de 8 bits, sendo necessária a computação separada das duas metades que compõem o endereço de acesso à memória.

A constante 0x00 é utilizada pelo processador no tratamento de instruções em modo de endereçamento página zero, já que a parte alta do endereço é desnecessária. Já o valor 0x01 tem propósito de atuar em conjunto com o registrador S, conectado ao multiplexador ADL, para que o endereço de acesso à pilha esteja sempre contido no intervalo 0x0100 e 0x0FF. As demais constantes são utilizadas para tratamento de interrupções, já que consistem nos vetores de acesso à memória para a busca do endereço da rotina correspondente a cada uma delas. Esses endereços foram representados em forma de uma única entrada de MUX ADL, entretanto na implementação realizada existe uma entrada para cada um dos bytes de forma que o endereço da rotina de interrupção seja escrito em AB.

Prosseguindo a análise, o ponteiro de pilha (S) está localizado logo abaixo dos registradores que compõem PC, possuindo também um multiplexador conectado em sua entrada de dados. A primeira entrada do MUX S serve para dar suporte às instruções de transferência de registradores TXS e TSX através do barramento SB. A segunda entrada consiste no circuito dedicado ao decremento do registrador de pilha, operação realizada nos momentos em que um dado é armazenado nessa região de memória. Novamente, o bloco subtrator que realiza essa operação foi representado pela entrada **S --** para evitar o excesso de informações no diagrama. Deve-se destacar que não há um somador dedicado ao incremento do valor de S no projeto original do processador, sendo que esta tarefa é realizada através da ULA.

A Unidade Lógico Aritmética possui os registradores AI e BI conectados em suas entradas para que sejam armazenados temporariamente os operandos a serem manipulados por ela. A única fonte de dados do registrador BI é o barramento DB para que valores lidos da memória sejam utilizados nas operações. Porém, observa-se que existe a opção de inserir o valor proveniente da memória de forma negada através de um inversor conectado na segunda entrada de MUX BI. Essa estratégia foi

utilizada para que as operações que envolvem subtração sejam realizadas pelo mesmo circuito que realiza as somas, não havendo a necessidade de um circuito dedicado à realização de subtrações. Para exemplificar esse funcionamento, suponha a execução de uma instrução para subtrair 5 unidades do valor 0x20 que está contido no acumulador, a sequência de operações seria:

- Leitura do valor 0x05 da memória, o qual fica disponível no barramento DB;
- Negação do valor 0x05, resultando em 0xFA, através do inversor conectado na entrada de MUX BI;
- Armazenamento de 0xFA em BI e de 0x20 em AI;
- Operação da ULA é definida como soma com *carry*;
- Força-se o valor da entrada c*arry in* da ULA em 1, resultando na operação: 0x20 + 0xFA + 0x01 = 0x1B;

O registrador AI por sua vez possui 3 possíveis entradas que são selecionadas através de MUX AI, sendo elas: ADL, 0x00 e SB. A primeira entrada é utilizada principalmente pelas instruções de desvio condicional, abordadas detalhadamente na Seção 0. Para essas instruções, o valor de deslocamento oriundo da memória é armazenado no registrador BI e o valor correspondente ao endereço atual de PC é armazenado em AI, através do barramento ADL. No ciclo seguinte, é realizada a soma dos dois valores e o endereço de destino é obtido.

A segunda entrada de MUX AI é a constante 0x00, utilizada quando é necessária a realização de incremento de algum valor, como no caso da execução da instrução INC. Nessas situações, o valor de ABH é copiado para o registrador BI e a constante 0x00 para o registrador AI. A operação da ULA é então definida de forma a realizar a soma AI + BI + 1, resultando no incremento necessário do dado armazenado em memória.

Por último, o barramento SB está conectado ao registrador AI através da terceira entrada de seu multiplexador para que seja possível a cópia do valor armazenado nos registradores A, X e Y. Essa conexão é essencial para que seja possível a manipulação dos 3 principais registradores accessíveis ao programador, dando suporte às instruções lógicas, aritméticas e de comparação, assim como aos modos de endereçamento indexados por X ou por Y.

Na sequência, observa-se o registrador de estado P, posicionado logo abaixo dos registradores AI e BI. Conforme discutido na Seção 2.1, o registrador P armazena as *flags* de estado do processador, sendo que sua implementação foi realizada através de 8 *flip-flops* a fim de permitir *set/reset* individual de cada bit. Tal abordagem dá suporte às instruções que manipulam diretamente as *flags* (e.g. SEC/CLC). A representação através de um único registrador apresentada tem como objetivo simplificar a construção do diagrama, além de facilitar seu entendimento.

Nota-se que a entrada de dados de P possui duas fontes distintas, onde a primeira consiste nas *flags* V, C, N e Z resultantes das mais diversas operações do processador, como somas, subtrações e

até mesmo carregamento de valores da memória nos registradores A, X e Y. Observa-se que as *flags* V e C são computadas pela ULA, visto que somente operações realizadas por este bloco geram alterações em tais sinais. Já as *flags* N e Z originam-se do barramento SB para que seja possível a atualização desses sinais por instruções em que os dados não passam pela ULA, como no caso de LDA. Durante a execução dessa instrução, o caminho do dado a ser armazenado em A inicia-se a partir do barramento DB após leitura da memória, passa pelo barramento SB (MUX SB) e então é armazenado no registrador destino. A partir da abordagem utilizada todas as possibilidades de alteração de *flags* foram cobertas, sendo que foi necessária a implementação do bloco de processamento de *flags* contendo a lógica destinada a verificar alterações nos sinais que não se originam na ULA.

A segunda entrada de MUX P tem como funcionalidade dar suporte às instruções PLP, RTS e RTI, nas quais valor do registrador P armazenado previamente é recuperado da pilha pelo barramento DB. Para que fosse possível a execução das instruções PHP, JSR e BRK, nas quais o valor de P é armazenado na memória, a saída de tal registrador foi conectada ao barramento DB, através do MUX DB.

A *flag* C obtida da saída do registrador de estado é utilizada pela ULA como fonte do sinal *carry in* de forma a possibilitar a execução das instruções ADC e SBC. Entretanto, nota-se que é possível definir o valor de *carry* a ser enviado para a ULA através de MUX C. A constante 0x01 conectada na primeira posição do multiplexador é utilizada em situações em que é necessário realizar o incremento de algum valor, como por exemplo o incremento de S no instante em que um dado é armazenado na pilha. Já o sinal HC foi implementado através de um *flip-flop*, suprimido do diagrama, e tem como objetivo detectar o chamado *half-carry*, ou seja, o sinal de *carry* gerado por operações internas do processador, como no caso de cruzamento de página citado anteriormente. Tal sinal não deve sobrescrever o valor atual da *flag* C por ser resultante de uma operação interna, justificando a necessidade de HC.

Continuando a análise do bloco de dados construído, destaca-se o fato de que os 3 registradores acessíveis ao programador (X, Y, e A) estão posicionados à direta do bloco da ULA. Nota-se que a única fonte de dados desses registradores é o barramento SB, visto que suas entradas e saídas estão conectadas a ele. Apenas o registrador A possui ligação direta com o barramento DB, através de MUX DB, para que o valor contido em X e Y seja armazenado na memória, deve-se primeiramente enviá-los ao barramento SB através do multiplexador de controle MUX SB.

Os últimos elementos a serem analisados são os multiplexadores que controlam o acesso aos barramentos SB e DB do processador. O denominado MUX SB tem como função principal a transferência de dados entre os registradores X, Y e A, além de oferecer suporte para envio do valor contido em S para que seja incrementado pela ULA. O barramento ADH está conectado nesse

multiplexador de forma a realizar o incremento da parte alta do endereço causado por cruzamentos de página. O MUX DB por sua vez é o caminho de entrada de dados do processador, oferecendo também a conexão para o armazenamento dos valores contidos em A, PCL, PCH e P na memória.

Comparando o diagrama construído com o original, nota-se a ausência dos *latches* e dos chamados *pass mosfets*, circuitos que se assemelham à *buffers tristate*. Esses dois circuitos não são suportados pelo FPGA Spartan 6 (XILINX, 2012) e foram substituídos por *flip-flops* e multiplexadores, os quais possuem a mesma funcionalidade, porém são suportados pelo hardware alvo.

3.2 Bloco de Controle

A partir da conclusão da estrutura do bloco de dados a ser utilizada pelo processador, seguimos para o desenvolvimento das micro operações realizadas em cada ciclo de execução do conjunto de instruções, as quais estão exemplificadas no Apêndice I do presente trabalho. As micro operações facilitaram a implementação do bloco de controle já que definem a operação a ser realizada em cada ciclo bem como o caminho realizado pelos dados, bastando apenas definir quais sinais de controle são necessários para a correta execução das instruções. A estrutura geral da lógica de controle foi baseada na região posicionada à esquerda da Figura 10, resultando no diagrama exposto pela Figura 12.

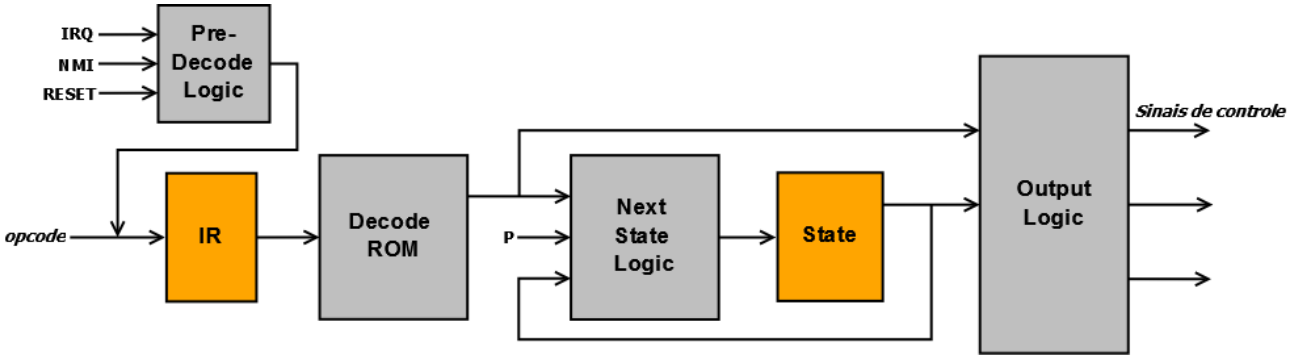

Figura 12 – Diagrama de blocos da lógica de controle.

A sequência de operações de controle inicia-se a partir da atualização do valor contido no registrador de instruções (IR), o qual armazena um *opcode* de tamanho 1 byte. Esse *opcode* é então enviado para a lógica de decodificação (*Decode ROM*) para que seja identificada a instrução e modo de endereçamento aos quais ele se refere. A parir deste instante, é possível definir quantos ciclos serão necessários para a execução da instrução bem como quais são os sinais de controle a serem gerados. A sequência de estados é então tratada pelo bloco *Next State Logic* em conjunto com o registrador *State*, destacando-se que os fatores que influenciam a sequência são o estado atual e o

opcode decodificado. Os valores das *flags* armazenadas no registrador P somente influenciam a sequência de estados durante a execução de instruções de desvio condicional.

Nota-se o bloco *Pre-Decode Logic* posicionado acima do registrador de instruções, sendo que suas entradas correspondem aos sinais de interrupção por hardware. O funcionamento desse bloco relaciona-se com o tratamento dos sinais de interrupção externos, sendo que o mesmo registra o instante em que o sinal de interrupção chega e aguarda até o final da execução da instrução corrente. No ciclo de busca de uma nova instrução, esse bloco interfere na operação da lógica de controle forçando o byte 0x00 na entrada do registrador de instruções. Dessa forma, o processador passa a realizar a sequência de interrupção que corresponde à execução da instrução BREAK (*opcode* = 0x00). Entretanto, cabe ao bloco *Pre-Decode Logic* armazenar qual dos sinais conectados à sua entrada foi o gerador da interrupção para que o vetor de tratamento correspondente seja carregado no PC.

De forma a esclarecer o processo de operação do bloco de controle desenvolvido, usaremos como exemplo a instrução LDA no modo de endereçamento página zero indexado por X. A Figura 13 consiste no diagrama de estados correspondente à execução dessa instrução e ilustra também as micro operações e sinais de controle gerados em cada ciclo.

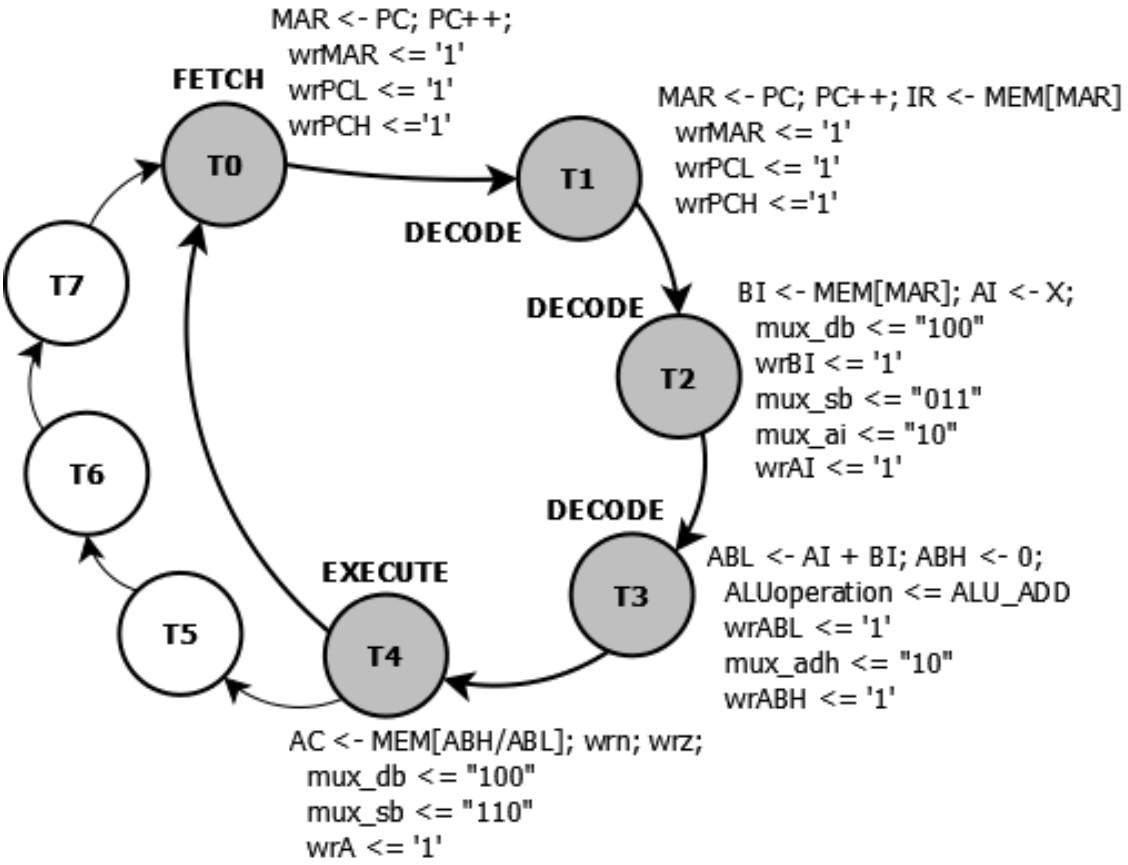

Figura 13 – Máquina de estados finita do bloco de controle.

O estado T0 da máquina de estados construída consiste no ciclo de busca da próxima instrução, por esse motivo, observam-se os sinais destinados a habilitar a escrita dos registradores PC e MAR. Dessa forma, o contador de programa tem seu valor incrementado e os registradores de endereço passam a apontar para o *opcode* que identifica a instrução. O estado de decodificação é o T1, onde o *opcode* lido previamente é armazenado no registrador de instruções para ser enviado ao bloco *Decode ROM*.

Concluída a etapa de decodificação, no estado T2 obtêm-se o byte que identifica o endereço da página zero que servirá como base para o cálculo do endereço efetivo de acesso à memória. Esse byte é lido da memória pelo barramento DB e então escrito em BI. De forma a possibilitar a soma do deslocamento ao endereço base, o registrador X tem seu valor copiado para o registrador AI através do barramento SB.

No estado T3 os operandos base e deslocamento encontram-se devidamente armazenados nos registradores de entrada de dados da ULA, tornando possível a realização da soma entre eles. O resultado obtido por essa operação é então enviado ao registrador de endereços ABL, sendo que ABH recebe a constante 0x00, já que se trata de uma instrução de acesso à página zero da memória.

Por fim, o valor a ser armazenado no registrador A é obtido a partir da leitura do conteúdo armazenado no endereço obtido no ciclo anterior (base + deslocamento). Portanto, para concluir a execução da instrução, o dado é lido através de DB, enviado ao barramento SB e então escrito no registrador acumulador juntamente com a atualização das *flags* N e Z. Observa-se que os estados T5, T6 e T7 são ilustrados pelo diagrama, porém não são utilizados pela combinação de instrução e modo de endereçamento utilizada como exemplo. Esses estados são utilizados na execução de instruções mais longas, como a interrupção por software BREAK, ou instruções em modo de endereçamento indireto, as quais envolvem mais etapas durante sua execução.

Deve-se citar o fato de que é inviável apresentar no diagrama de estados todos os sinais de controle e as condições para ativação de cada um deles devido à grande quantidade de modos de endereçamento. Por esse motivo, as micro operações exemplificadas no Apêndice I do presente trabalho e disponibilizadas por completo no repositório do *github* (github.com/bernardo-andreeti/6502) consistem na melhor forma de documentação possível da implementação do bloco de controle desenvolvida no projeto. A documentação completa não foi inserida no Apêndice I por ser muito extensa, por esse motivo optamos pela disponibilização através do repositório.

3.3 Implementação e simulação VHDL

Com a conclusão dos diagramas do bloco de dados, bloco de controle e definição das micro operações executadas por cada instrução, partimos para a implementação do processador utilizando a linguagem de descrição de hardware VHDL. A descrição iniciou-se com o intuito de construir a

estrutura básica do processador, contendo o bloco de dados completo assim como a lógica de decodificação de instruções. Uma memória de 64 KB também foi implementada de forma a possibilitar a execução de programas e teste do processador em desenvolvimento.

O bloco em que foi investida a maior parcela do tempo de projeto foi o destinado a gerar os sinais de saída da lógica de controle (*Output Logic*), justamente pelo fato de que cada combinação de instrução e modo de endereçamento possui particularidades quanto aos sinais a serem enviados ao bloco de dados. Dessa forma, o suporte aos diferentes grupos de instruções e modos de endereçamento foi construído em etapas e sua validação foi realizada com auxílio do programa de teste *AllSuite.asm*, o qual pode ser consultado no Anexo A. Trata-se de um programa de código aberto, escrito em *assembly* e que tem sua estrutura dividida em 15 testes distintos. Cada teste é destinado a um determinado grupo do conjunto de instruções.

Para que o programa *AllSuite* pudesse ser carregado na memória durante a simulação VHDL, seu código foi montado com auxílio do simulador do processador 6502 (KOWALSKI, 2003), possibilitando a geração da imagem de memória RAM. Essa imagem era então convertida em arquivo texto contendo os bytes de um programa através de software de conversão desenvolvido em linguagem C, também disponibilizado no repositório do projeto. O arquivo texto resultante foi utilizado para inicializar a memória implementada em VHDL, que por sua vez alimentou o processador com o programa a ser executado.

O software QuestaSim, versão 10.0b, foi o escolhido para gerar as simulações do código VHDL implementado por possuir suporte à utilização de *scripts* externos, além de oferecer uma interface de fácil utilização. A partir das simulações realizadas com o software QuestaSim, observamos que a implementação realizada de forma a dar suporte às instruções do grupo de leitura e escrita foram validadas pelo primeiro teste do programa *AllSuite*. Dessa maneira, partimos para a primeira síntese e implementação para o hardware alvo, processo esse que será descrito na seção 4.

4 PROTOTIPAÇÃO

Com a etapa de codificação inicial do processador em VHDL concluída, foi realizada a primeira prototipação para a plataforma de desenvolvimento Nexys 3. Dessa forma, puderam ser levantadas as adaptações necessárias para adequação da arquitetura do processador para a FPGA Spartan 6, já que, por se tratar de um design antigo, muitas técnicas utilizadas no 6502 original não são mais suportadas. Nesta etapa foi possível também a realização de testes pós-síntese, de maneira a verificar a manutenção do funcionamento correto do processador implementado, assim como testes físicos na plataforma de desenvolvimento.

Para que fossem realizados os procedimentos descritos, foram utilizados os softwares contidos no pacote ISE Design Suite, disponibilizado pela empresa Xilinx, destacando-se os programas ISE, Plan Ahead (síntese e implementação) e ChipScope Pro (gravação de *bitstream* e depuração da execução). A partir da validação da primeira versão do projeto, as etapas seguintes corresponderam à alternância entre as etapas de descrição e prototipação, de forma a validar cada alteração realizada na descrição VHDL além de pequenas correções entre as versões. Optamos pela metodologia de descrição e validação incremental para que fosse possível realizar alterações praticamente imediatas caso surgisse algum problema na prototipação do código VHDL para o hardware alvo.

A documentação relacionada ao kit de desenvolvimento Nexys 3 e à FPGA Spartan 6 foi essencial para o sucesso dessa etapa. Pode-se destacar o *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices* (XILINX, 2011), relacionado ao sintetizador utilizado para a plataforma, o *Ise In-Depth Tutorial* (XILINX, 2012) e *ChipScope Pro Software and Cores* (XILINX, 2012), onde são encontradas informações acerca da IDE disponibilizada e do software de transferência do *bitstream* gerado pela implementação para o kit de desenvolvimento, respectivamente.

4.1 Processador e *Block RAM*

A metodologia de validação da prototipação do processador para o kit de desenvolvimento Nexys 3 consistiu na execução do programa de testes utilizado durante a simulação VHDL na própria FPGA. Para que isso fosse possível, algumas mudanças foram necessárias, como por exemplo a implementação de uma memória de forma a utilizar os circuitos de *Block RAM* disponíveis na FPGA. Por se tratar de uma memória de 64 KB, sua implementação através de LUTs torna-se inviável, pois a memória ocuparia recursos lógicos do FPGA que poderiam ser utilizados pelo processador. A utilização de *Block RAM*, que consiste em um bloco de memória dedicado, elimina esse problema além de oferecer um desempenho melhor para o acesso randômico se comparada à implementação de memória distribuída (XILINX, 2011).

Para que o sintetizador inferisse *Block RAM* a partir da descrição VHDL da memória foi necessário torna-la completamente síncrona, ou seja, tanto escritas quanto leituras somente ocorrem em bordas de subida do *clock*. Para a inferência de memórias RAM distribuídas (LUTs) apenas as escritas devem ser síncronas (XILINX, 2011). Outra restrição imposta pelo sintetizador é que apenas arquivos de texto podem ser utilizados para que os dados do arquivo sejam copiados para a *Block RAM*, o que justifica a metodologia citada na seção 3.3 utilizada para obtenção da imagem de memória.

Essa alteração teve um impacto no desempenho do processador já que as leituras de dados passaram a ocorrer um ciclo de relógio atrasadas. Para corrigir o problema, optamos por fazer com que o bloco de dados operasse na borda de descida do *clock*, mantendo o bloco de controle operando em borda de subida.

A Figura 14 ilustra a estrutura utilizada para os testes do processador no kit de desenvolvimento Nexys 3.

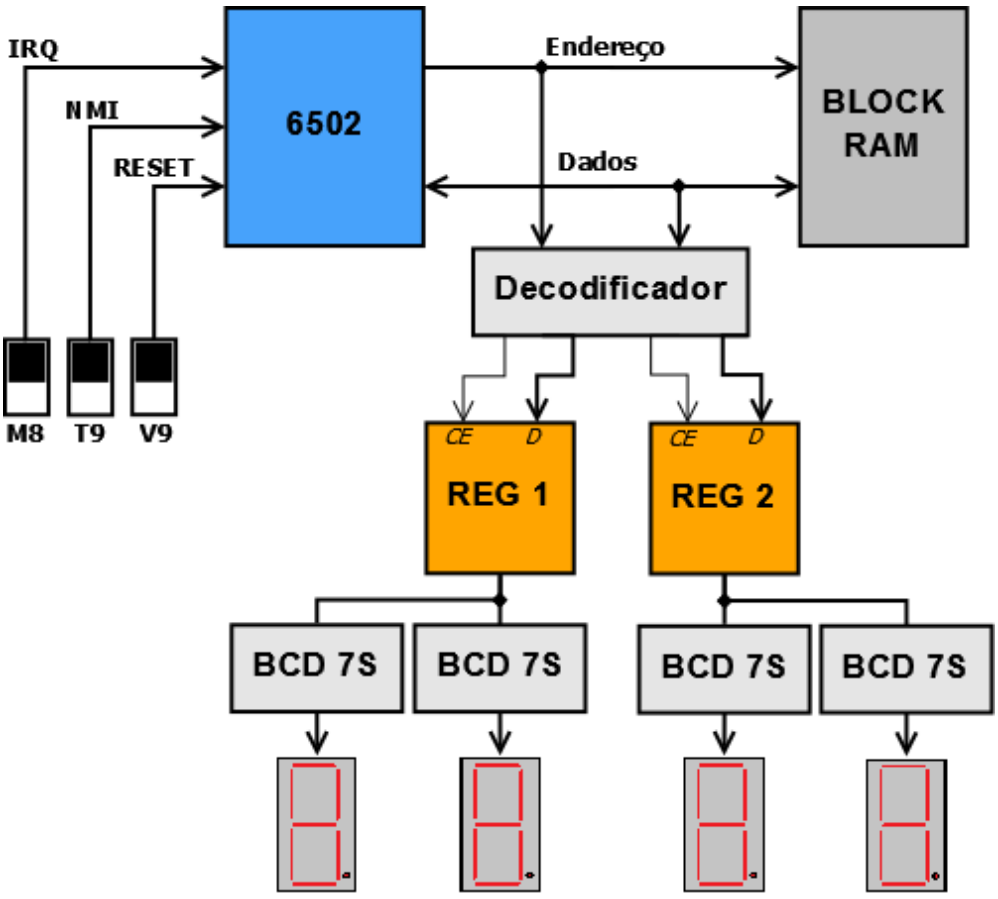

Figura 14 – Esquemático da estrutura de testes de prototipação.

Observa-se que entre o processador 6502 desenvolvido e a unidade de memória RAM foi inserido um decodificador conectado aos barramentos de endereços e de dados. Esse decodificador tem como função monitorar o barramento de endereços em busca de valores específicos que podem

ser definidos no código VHDL que define sua lógica. Quando o endereço buscado é detectado pelo bloco decodificador, o valor sendo transferido pelo barramento de dados é copiado para um dos registradores conectados na saída desse bloco. Os registradores por sua vez armazenam o valor que lhes foi enviado de forma a repassá-lo para uma lógica de decodificação BCD (*Binary Coded Decimal*) para que seja exibido pelos displays de 7 segmentos disponíveis no kit de desenvolvimento.

A estrutura ilustrada foi desenvolvida de forma que fosse possível visualizar os valores armazenados em determinados endereços de memória pelo processador em teste. Essa técnica facilitou muito a depuração da prototipação visto que pudemos verificar se os valores esperados de cada um dos testes feitos pelo programa *AllSuite* foram obtidos pelo processamento realizado no hardware alvo. Dessa forma, garantimos que o comportamento observado em simulação VHDL coincidisse com o do circuito prototipado. A Figura 15 consiste em uma foto do kit de desenvolvimento após a execução completa de todos os testes do programa utilizado.

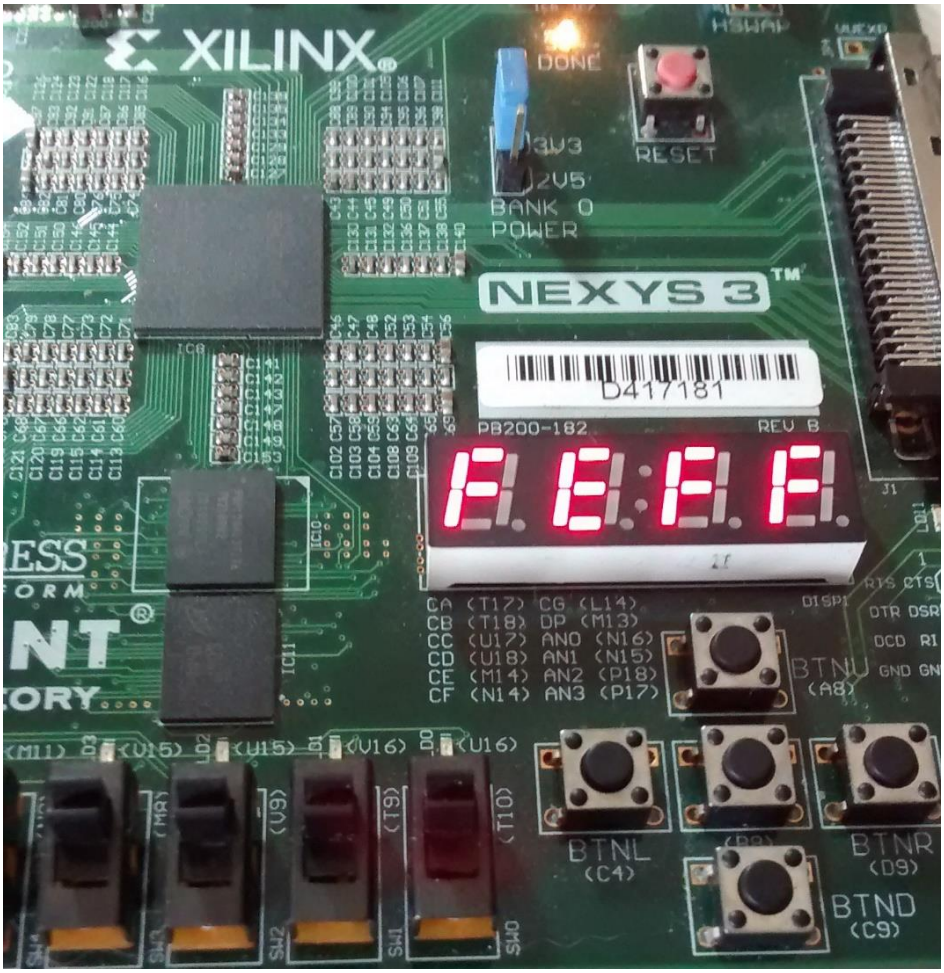

Figura 15 – Resultado do processamento exibido nos displays.

Os valores exibidos pelos displays da Figura 15 correspondem ao seguinte trecho de código do programa *AllSuite*:

```
suiteafinal:
        ; IF $0210 == 0xFE, INCREMENT
        ; (checking that it didn't
        ;  happen to wander off and
        ;  not run our instructions
        ;  to say which tests failed...)
        LDA #$FE
        CMP $0210
        BNE theend
        INC $0210
theend:
        JMP theend
```

Esse trecho de código *assembly* consiste na verificação final realizada pelo programa *AllSuite* após a execução com sucesso dos 15 testes de instruções. Conforme os comentários inseridos no código, para que a validação final ocorra o valor contido no endereço 0x0210 deve ser igual a 0xFE. Esse valor está sendo exibido pelos dois *displays* da esquerda na Figura 15, ou seja, o decodificador foi configurado de forma a monitorar o endereço 0x0210 e copiar o dado para o registrador responsável por esses *displays* (REG 1) no instante em que ocorresse uma leitura nesse endereço. Portanto, o valor esperado que valida a execução correta da bateria de testes foi obtido pelo processador prototipado.

Seguindo a análise do código, caso o valor contido em 0x0210 seja o esperado, ocorre um incremento do mesmo através da instrução INC, resultando em 0xFF. Novamente, observa-se esse mesmo valor sendo exibido pelos *displays* da direita na Figura 15 já que o decodificador foi configurado para copiar para o registrador fonte dos *displays* da direita (REG 2) o valor que fosse escrito no endereço 0x0210. Por fim, o programa entra em um laço infinito até que seja reiniciado por um sinal externo ou o processador sofra uma interrupção.

No que diz respeito a interrupções, a metodologia de testes utilizada fez uso dos *switches* disponibilizados no kit de desenvolvimento. Observa-se na Figura 14 que os sinais de interrupção de hardware IRQ, RESET e NMI estão conectados aos *switches* M8, V9 e T9, respectivamente. Dessa forma, quando o sinal enviado pelos *switches* é o nível lógico 1 (posição superior) o processador segue sua execução normal. Já quando o nível lógico é 0 (posição inferior), ocorre a interrupção do processador causada pelo sinal correspondente à chave que teve sua posição alterada.

Para que pudéssemos testar as interrupções no mesmo contexto em que foi validada a implementação do conjunto de instruções, o programa *AllSuite* foi modificado de forma a conter as

rotinas de interrupção correspondentes a cada um dos sinais, resultando no trecho de código inserido logo após a validação final citada anteriormente. A Figura 16 expõe o resultado de uma interrupção NMI causada pela alteração da posição da chave T9.

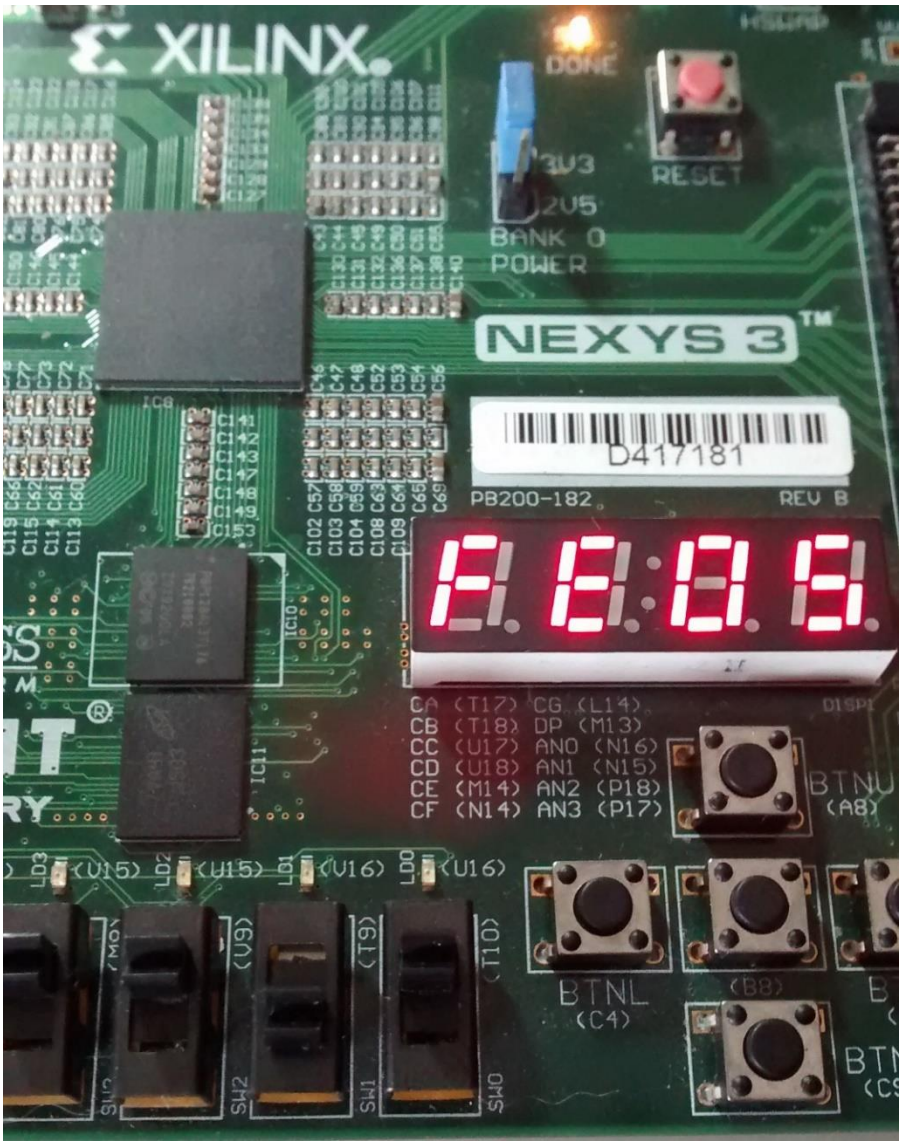

Figura 16 – Resultado obtido após execução de rotina de interrupção.

Como a rotina de interrupção atribuída para o sinal NMI opera sobre o valor armazenado no endereço 0x0210, a lógica do decodificador não precisou ser alterada. Trata-se de uma rotina bastante simples que carrega o valor 0x05 no registrador A e o armazena no referido endereço, resultando na alteração observada nos *displays* da direita na Figura 16. Apesar de se tratar de um teste simples, a chamada da rotina relacionada ao sinal de interrupção foi validada, bem como sua execução, sendo que o mesmo se aplica aos sinais IRQ e RESET.

Deve-se destacar que a lógica contida no bloco *Pre-Decode Logic* da Figura 12 garante que a rotina de interrupção seja tratada apenas uma vez após a alteração do nível lógico de alto para baixo,

evitando que ocorra um laço enquanto o sinal é mantido nesse nível. Para que esse comportamento fosse obtido, as entradas de interrupção foram definidas como sensíveis à borda de descida e não ao nível lógico baixo.

Outra característica importante da implementação é a utilização de uma interface destinada a realizar o *debounce* dos sinais enviados pelas chaves do kit de desenvolvimento. Caso esses sinais não sejam tratados, ocorre uma oscilação indesejada até que se estabilize no nível definido pela posição da chave. O circuito de *debounce* tem como objetivo corrigir esse fenômeno fazendo com que apenas uma alteração de nível seja enviada ao processador a cada mudança de posição dos *switches*, filtrando as imperfeições geradas pelo componente mecânico (GREENSTED, 2010).

Finalizada a etapa de prototipação, buscamos a obtenção de dados que caracterizassem o desempenho e eficiência do processador desenvolvido. Essa tarefa foi facilitada pelo fato de que durante o processo de síntese e implementação, o software disponibilizado pela Xilinx oferece relatórios completos que trazem todas informações que o projetista deseja. A Figura 17 consiste no gráfico exibido pelo software Plan Ahead, o qual revela informações relacionadas aos recursos da FPGA utilizados pelo projeto.

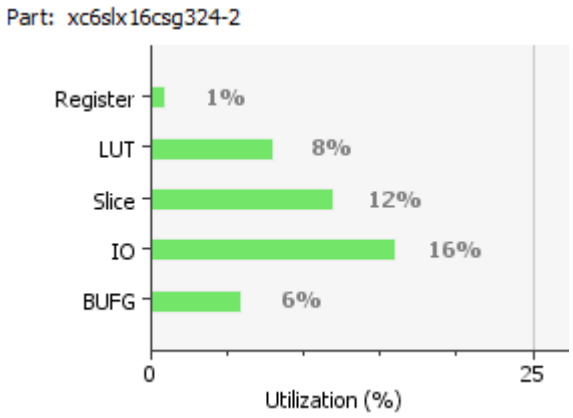

Figura 17 – Recursos utilizados pelo processador desenvolvido.

Observa-se que apenas 1% dos registradores disponíveis em chip foram utilizados, valor que corresponde à 125 unidades sequenciais. Com relação às *look-up tables* (LUTs), componentes que compõem efetivamente a lógica implementada, o resultado de 8% de utilização corresponde à 846 unidades ocupadas. Os *slices* consistem em grupos de 4 LUTs e 8 *flip-flops* no circuito reprogramável Spartan 6 (XILINX, 2012), sendo que 277 unidades deste componente foram ocupadas. Quanto às unidades de entrada e saída (IO), 39 delas foram utilizadas e apenas 1 *buffer* global de *clock* (BUFG) foi necessário.

Com relação às características de desempenho, o período mínimo de *clock* obtido foi de 29,476 *nS*, o que corresponde à uma frequência máxima de 33,926 MHz. A análise da dissipação de

potência do projeto foi realizada a partir do software X Power Analizer, sendo que o valor resultante foi de 30 mW. Deve-se destacar que os resultados citados correspondem apenas à prototipação do processador desenvolvido, sem levar em conta a unidade de memória utilizada na fase de testes.

4.2 Integração ao Projeto "*fpga_nes*"

A motivação de integrar o processador desenvolvido ao projeto que descreve o console NES encontra-se no fato de proporcionar validação final de seu funcionamento em uma aplicação real do processador 6502. Além disso a computação resultante da execução de jogos é muito maior que a oferecida pelo programa de testes utilizado na etapa de implementação e prototipação. Desse modo, uma combinação maior de instruções, dados e condições de operação puderam ser impostas ao processador, assim como sua interação em sistema com diversos circuitos periféricos também pode ser avaliada. A oportunidade de investigar o funcionamento do console NES também contribuiu para a escolha por se tratar de um dos dispositivos de maior sucesso dentre os que aplicaram o processador 6502 em sua estrutura.

Para que o processo ocorresse da melhor forma possível, foi necessário o estudo da arquitetura básica do console, observando os pontos principais da interação do processador com os demais circuitos. Uma das principais fontes de conhecimento para tal foi a documentação disponibilizada pelos desenvolvedores do emulador FCEUX (ADELIKAT, 2009), o qual simula o funcionamento do console em software. Além da documentação, o software oferece ferramentas de depuração que foram importantes para o entendimento da sequência de operações realizada durante a execução dos jogos.

O conjunto de informações obtidas a partir do estudo da documentação e utilização do emulador culminaram na construção do diagrama exposto na Figura 18, o qual ilustra de forma simplificada a arquitetura do console, destacando seus principais componentes.

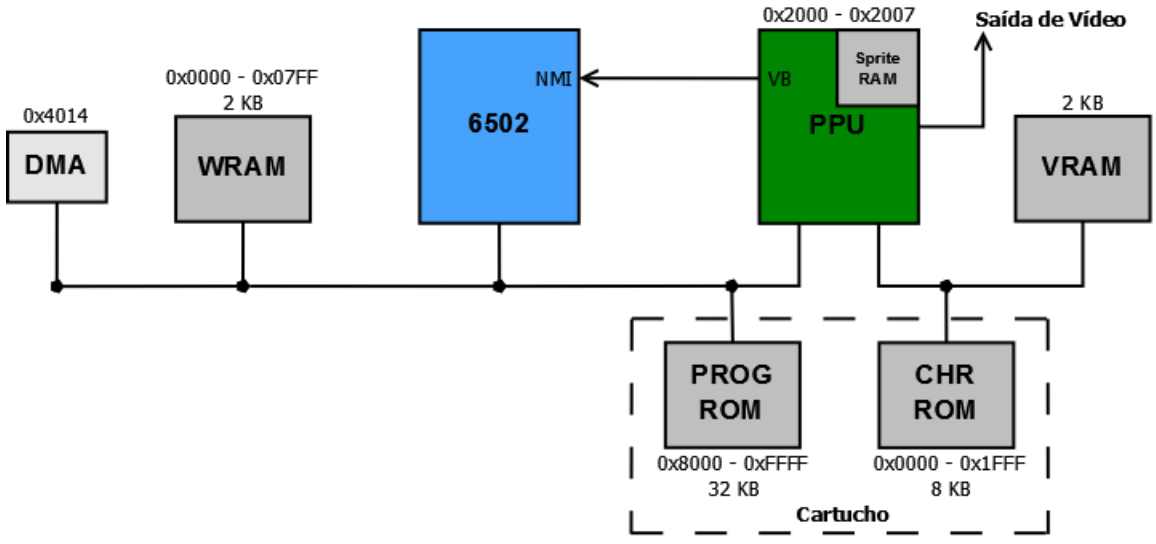

Figura 18 – Diagrama da arquitetura do console NES.

Para esclarecer o funcionamento do sistema e de seus componentes, a sequência de processamento de um *frame* será abordada. Um *frame* corresponde a uma imagem gerada pelo sistema que é enviada ao dispositivo destinado a exibi-la, seja ele um monitor ou um aparelho de televisão. No caso de um jogo, essa imagem representa a situação atual do mesmo de maneira estática, ou seja, o *frame* assemelha-se a uma foto de um determinado instante da execução do jogo. Para que o usuário de sistemas que usam *frames* para compor a imagem tenha a noção de que a exibição é um processo contínuo, os *frames* são renderizados em grande quantidade durante um curto período de tempo, de forma a atualizar a imagem sem que o usuário perceba sua discretização. Uma métrica comum que aponta a velocidade de geração de imagens de um sistema computacional é a taxa de *frames* por segundo (FPS), já para o dispositivo destinado a exibi-la é utilizada a medida em Hz, que indica sua frequência de atualização.

No caso do console NES, o processo de execução dos jogos começa pelo processamento inicial do código por parte do processador. O código de programa é armazenado na memória contida nos cartuchos de jogos do vídeo game, correspondendo ao bloco PROG ROM. Conforme indicado no diagrama, o intervalo de endereços que contém o código de programa inicia-se em 0x8000 e vai até 0xFFFF.

No início da execução é comum que a memória identificada como WRAM (*work RAM*) seja inicializada com as constates e dados necessários para o prosseguimento da execução. No caso deste sistema, o processador tem disponível 2 KB de memória de acesso randômico, compreendidos entre os endereços 0x0000 e 0x07FF. Essa área de memória é utilizada para armazenar as variáveis dos jogos como por exemplo localização atual dos personagens na tela, pontuação, tempo, número de vidas, etc.

Concluída a etapa de inicialização, o processador envia ao circuito denominado PPU (*Picture Processing Unit*) um sinal que habilita seu funcionamento. Esse bloco, como o próprio nome indica, é destinado ao processamento das imagens geradas e envio dos sinais que formam essa imagem para a saída de vídeo. O fato de conter uma unidade com propósito específico de processamento de imagem foi responsável pela qualidade gráfica alcançada pelo console NES, além de possibilitar o desenvolvimento de jogos mais complexos para a época, já que o processador possuía mais tempo de execução livre para processamento do código de programa.

Deve-se destacar que a comunicação entre CPU e PPU é realizada através de registradores mapeados na memória do 6502. Esses registradores fazem parte da PPU e servem para controlar o seu funcionamento. Eles são acessados pela CPU a partir dos endereços contidos no intervalo de 0x2000 a 0x2007.

Após habilitar o bloco de processamento de imagens, o processador entra em estado de espera até que a PPU envie um sinal de interrupção através da entrada NMI. Esse sinal é conhecido como

Vertical Blank e tem como objetivo indicar que um novo *frame* deve ser processado. No início da rotina de interrupção ocorre o carregamento das *sprites*, que são blocos de imagem formados por 64 *pixels* (8x8) destinados a compor os elementos móveis da cena. Os bytes correspondentes às *sprites* são enviados para uma memória interna da PPU (*Sprite* RAM) através do circuito DMA (*Direct Memory Access*), que é ativado a partir da escrita no endereço 0x4014. Esse circuito permite a cópia de 256 bytes em uma fração do tempo que esse processo levaria caso fosse realizado pelo processador através de instruções de leitura e escrita.

Logo após o termino da cópia de *sprites* através do DMA, o processador passa a executar a denominada *engine*, que consiste no trecho de código em que todas as funcionalidades do jogo são descritas, como a detecção de colisões entre os elementos, processamento de áudio, leitura dos comandos enviados através dos *joysticks*, entre outras. Ou seja, a parcela mais importante do processamento dos jogos ocorre no curto intervalo de tempo compreendido entre cada *frame*, fator que define a complexidade do programa desenvolvido, visto que esse processamento não pode exceder o limite de tempo em que a próxima cena deve ser exibida.

Parte do processamento da *engine* consiste na atualização da memória VRAM utilizada pela PPU. Essa memória também tem capacidade de 2 KB e é modificada pelo processador através do endereço 0x2007, sendo que é destinada ao armazenamento do plano de fundo e das cores que compõem a imagem. O plano de fundo é composto a partir da combinação de pequenos padrões de imagem chamados *tiles* (8x8 *pixels*). Esses dados são específicos para cada jogo e estão contidos na memória CHR ROM, que faz parte da estrutura do cartucho. A PPU então agrupa e processa os dados contidos nas memórias VRAM e *sprite* RAM, enviando os sinais destinados a compor a imagem renderizada para a saída de vídeo do console, concluindo o processo.

Terminada a renderização do *frame*, o processador entra em estado de espera novamente até que um novo sinal *Vertical Blank* seja emitido pela PPU. Deve-se destacar que todo o processamento descrito se repete 60 vezes em um segundo para os consoles que utilizam o sistema de cores NTSC e 50 vezes por segundo para o sistema PAL, de forma a coincidir com a taxa de atualização de tela imposta por esses sistemas.

Partindo do conhecimento da arquitetura do console NES, a seguinte etapa da integração consistiu na análise da implementação realizada na linguagem Verilog do projeto *fpga_nes* (BENNET, 2012). Esse projeto descreve todos os circuitos do console apresentados na Figura 19. O objetivo era simplesmente substituir a descrição Verilog do processador 6502 pela descrição VHDL implementada. Essa etapa acabou sendo dificultada pela ausência de documentação complementar do projeto citado, sendo necessário o estudo do código fonte em busca de esclarecimentos sobre seu funcionamento.

O primeiro obstáculo encontrado foi o fato de o processador 6502 desenvolvido no projeto *fpga_nes* operar em uma frequência de 1,785 MHz e o resto do sistema operar a uma frequência de 100 MHz. O fato de o processador trabalhar na frequência mencionada já era de nosso conhecimento, visto que coincide com o projeto original do console, porém durante nossa implementação não levamos em conta a possibilidade de haver divergência entre a frequência de memória e da CPU. Dessa forma, através de uma simulação VHDL/Verilog (6502 em VHDL e restante do sistema em Verilog), observamos que primeira tentativa de substituição do processador pelo nosso fracassou devido à diferença de frequência de operação dos componentes.

Analisando as formas de onda resultantes da simulação, observamos que as diferentes frequências de operação causavam um problema de sincronia da comunicação entre CPU e WRAM. Esse problema ocorria principalmente durante a execução de instruções de escrita de dados, onde o dado era armazenado pela unidade de memória antes que o processador concluísse o processamento do endereço de destino. Uma situação semelhante ocorria durante as operações de IO, nas quais o processador acessa os registradores internos da PPU.

A solução encontrada foi o desenvolvimento de uma interface de sincronia externa ao processador que atuasse sobre os barramentos de dados e de endereços de forma a resolver os problemas citados. Essa abordagem proporcionou a correção da comunicação entre os diferentes blocos do sistema sem que fosse necessária alguma alteração interna no processador desenvolvido, o qual já tinha seu funcionamento validado. A interface foi desenvolvida de forma a garantir que o dado a ser armazenado, endereço de destino e sinal que habilita escrita fossem enviados aos demais blocos no mesmo ciclo de *clock*, corrigindo tanto a comunicação com a memória quanto leituras e escritas nos registradores internos da PPU.

Novamente a simulação VHDL/Verilog do sistema completo foi o mecanismo inicial de validação da interface de sincronia desenvolvida. Nessa etapa, o programa de testes *AllSuite* utilizado anteriormente como imagem de memória foi substituído pelo jogo Super Mario Bros. Entretanto, o processo de obtenção do arquivo texto com a imagem a ser carregada foi mantido, sendo que a única operação adicional necessária era a separação dos dados destinados à PROG ROM, correspondentes ao programa em si, dos dados da CHR ROM.

Através das formas de onda obtidas após as correções de sincronia aplicadas observamos que o processamento do jogo Super Mario Bros estava correto, já que a interação entre o processador e os demais chips passou a funcionar como o esperado. Pudemos observar os momentos quem que o sinal *Vertical Blank* era enviado pela PPU, gerando a interrupção no processador que ocasionava o processamento de um novo *frame*. Diante do sucesso obtido em simulação, partimos para a prototipação do sistema e execução dos jogos no hardware alvo. A Figura 19 ilustra o jogo Super Mario Bros sendo executado no kit de desenvolvimento Nexys 3.

Observa-se que a saída VGA disponível no kit de desenvolvimento foi utilizada para enviar as imagens renderizadas pelo sistema para o monitor. O projeto *fpga_nes* também oferece suporte para conexão dos *joysticks* do console original através dos pinos PMOD de entrada e saída do kit, de forma a dar suporte à interação do usuário com o jogo. Há suporte também para o áudio, sendo que a conexão também é realizada pela interface PMOD, e a funcionalidade de *reset* do console foi atribuída ao botão C9.

Figura 19 – Execução de jogos do console NES.

Observa-se a presença de um cabo conectado na entrada USB UART do kit de desenvolvimento. Essa interface é utilizada para o carregamento dos jogos a serem executados, os quais são enviados através do software NESDBG, desenvolvido pelo autor do projeto original. Deve-se destacar que os jogos suportados são os mesmos utilizados pelo emulador FCEUX, que possuem a extensão NES.

Dessa forma, chegamos ao final do projeto proposto, visto que obtivemos uma descrição funcional do processador 6502. Essa descrição foi validada através de simulações VHDL, prototipação em FPGA e aplicação no projeto que descreve o console NES, resultando na contemplação de todos os objetivos iniciais de projeto.

5 CONCLUSÃO

O processador 6502 representou um marco na história da computação, sendo utilizado em larga escala nas décadas de 1970 e 1980. Grande parte de seu sucesso pode ser atribuída à relação entre custo e desempenho que o circuito integrado apresentava, algo que foi colocado como objetivo principal de seu desenvolvimento. A partir de sua concepção, os processadores de propósito geral tiveram sua utilização alavancada nos dispositivos de produção em massa, já que foi demonstrado que era possível o desenvolvimento de tais componentes a custos acessíveis.

O presente trabalho descreveu o processo de revisitação do processador 6502 a partir de uma implementação na linguagem VHDL e prototipação em FPGA. Obviamente, durante o curso do projeto realizamos adaptações, como a substituição de *latches* que compunham a estrutura original do circuito integrado por *flip-flops*, de forma a adequar o processador à processos modernos de síntese. Porém, após a conclusão de todas as etapas de desenvolvimento, obtivemos um processador funcional com suporte ao conjunto de instruções do projeto original. Além da documentação detalhada sobre a execução das instruções, os códigos desenvolvidos também estão disponíveis no repositório do sistema de controle de versões *github* (github.com/bernardo-andreeti/6502).

Deve-se destacar a complexidade da etapa de verificação que envolve o desenvolvimento de circuitos VLSI (*Very Large Scale Integration*). Após a integração VHDL/Verilog discutida na seção 4.2, bem como o desenvolvimento da camada de sincronização entre o processador e os demais componentes do sistema, ainda pudemos identificar problemas em algumas instruções a partir da depuração com o código *assembly* do jogo Super Mario Bros, que puderam ser corrigidos. Entretanto, alguns jogos do console NES ainda não são executados de maneira correta, sendo que a origem do problema não pode ser investigada por falta de tempo de projeto. Conclui-se que a etapa de verificação é de extrema importância e deve ser realizada a partir de metodologias específicas, principalmente quando se trata do desenvolvimento de ASICs, onde não há possibilidade de correções pós fabricação.

Outro aspecto importante abordado pelo trabalho relaciona-se com a utilização do código de terceiros. Nesse caso, utilizamos o projeto que descreve o console NES por completo para aprofundar a validação do processador desenvolvido. A utilização de códigos de outros desenvolvedores é algo comum em equipes de desenvolvimento, tanto de hardware quanto de software, sendo que é imprescindível a elaboração de documentação complementar para que esse processo seja facilitado.

Com relação às características de eficiência e desempenho, será realizada uma comparação entre o processador implementado em VHDL, identificado como “P6502.vhd” na Figura 20, e o processador original do projeto *fpga_nes*, identificado como “cpu.v”.

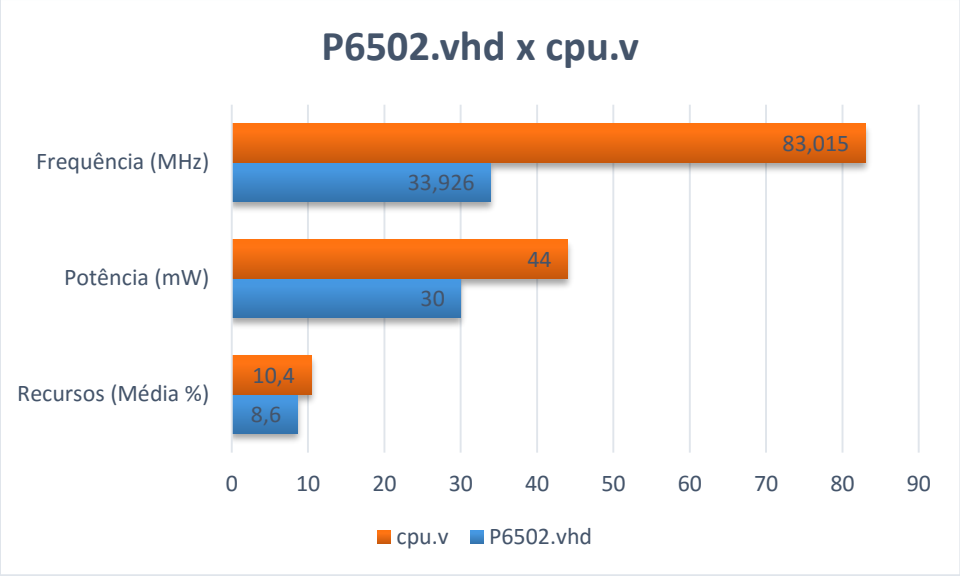

Figura 20 – Comparativo entre implementação Verilog e VHDL.

Observa-se uma vantagem da descrição Verilog no que se relaciona à frequência de operação máxima do circuito prototipado (83,015 MHz contra 33,926 MHz). O motivo pelo qual obtivemos uma frequência máxima menor foi o fato de o bloco de dados operar em uma borda de c*lock* diferente do bloco de controle, o que dificulta as etapas de posicionamento e roteamento do sintetizador (CHU, 2006). Analisando os dados de potência dissipada e média de recursos utilizados da FPGA, conclui-se que o processador descrito em VHDL é mais eficiente. Os resultados obtidos podem ser atribuídos ao fato de que a implementação Verilog apresenta um nível de abstração mais alto que a realizada em VHDL. Dessa forma, a ferramenta de síntese possui um grau de liberdade maior na prototipação do circuito para o hardware alvo, sendo que claramente houve priorização do desempenho em função da dissipação de potência e recursos ocupados.

Como trabalhos futuros relacionados ao projeto desenvolvido, deve-se buscar a solução dos problemas que impedem a execução de alguns jogos do console NES, sejam eles relacionados à implementação das instruções ou à camada de sincronia entre os componentes do sistema. É notável a ausência da técnica de *pre-fetch* no processador desenvolvido, a qual foi uma das responsáveis pelo sucesso do projeto original, sendo que deve ser aplicada em busca da redução do número de ciclos necessários para execução das instruções suportadas.

Outra alteração relevante seria a busca por uma alternativa para a abordagem utilizada em que o bloco de controle e o bloco de dados operam em bordas de *clock* distintas, de forma a compensar o atraso causado pela memória RAM síncrona. Apesar de resultar na solução do problema, tal alteração tem impacto negativo na frequência máxima de operação do circuito prototipado.

6 REFERÊNCIAS

BENNET, Brian. **FPGA_NES Project**. 2012. Disponível em:
< https://github.com/brianbennett/fpga_nes > Acesso em: 10 de Junho de 2015.

BEREGNYEI, Balazs. **6502 Reverse Engineering**. 2001. Disponível em:
<http://www.visual6502.org/wiki/index.php?title=Balazs%27_schematic_and_documents> Acesso em: 06 de Agosto de 2015.

NABEREZNY, Mike. **6502.org - the 6502 microprocessor resource**. 2006. Disponível em:
<http://6502.org/> Acesso em: 12 de Agosto de 2015.

MOS TECHNOLOGY, INC. **MCS6500 Microcomputer Family Programming Manual**. 1976. Disponível em:
<http://users.telenet.be/kim1-6502/6502/proman.html> Acesso em: 13 de Agosto de 2015.

MOS TECHNOLOGY, INC. **MCS6500 Microcomputer Family Hardware Manual**. 1976. Disponível em:
<http://archive.6502.org/books/mcs6500_family_hardware_manual.pdf> Acesso em: 13 de Agosto de 2015.

MOS TECHNOLOGY, INC. **MCS6500 Microprocessors Data Sheet**. 1976. Disponível em:
<http://archive.6502.org/datasheets/mos_6500_mpu_preliminary_may_1976.pdf> Acesso em: 15 de Agosto de 2015.

JAMES, Greg; SILVERMAN, Barry; SILVERMAN, Brian; SPITTLES, Ed. **The Visual 6502**. 2009. Disponível em: <http://www.visual6502.org/> Acesso em 08: de Agosto de 2015.

CHEN, Yu; CHOI, Jaebin; SUNDARAM, Arthy; ERLINGER, Anthony. **Reconstruction of the MOS 6502 on the Cyclone II FPGA**. 2010. Disponível em:
<http://www.cs.columbia.edu/~sedwards/classes/2013/4840/reports/6502.pdf> Acesso em: 03 de Agosto de 2015.

XILINX. **XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices.** 2011. Disponível em: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/xst_v6s6.pdf> Acesso em: 01 de Agosto de 2015.

XILINX. **PlanAhead User Guide.** 2012. Disponível em: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/PlanAhead_UserGuide.pdf > Acesso em: 02 de Agosto de 2015.

XILINX. **ChipScope Pro Software and Cores.** 2012. Disponível em: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/chipscope_pro_sw_cores_ug029.pdf> Acesso em: 05 de Agosto de 2015.

HANSON, Donald F. **A VHDL Conversion Tool for Logic Equations with Embedded D Latches**. 1995. Disponível em: <https://www.ncsu.edu/wcae/WCAE1/hanson.pdf> Acesso em: 20 de Outubro de 2015.

PURCARU, John B. **Games vs. Hardware. A history of PC gaming: The 80's**. 2014. Disponível em:<https://books.google.com.br/books?id=lB4PAwAAQBAJ&printsec=frontcover&hl=pt-BR#v=onepage&q&f=false> Acesso em: 12 de Novembro de 2015.

BURCH, Carls. **Software Logisim**. 2005. Disponível em: < http://www.cburch.com/logisim/pt/index.html> Acesso em: 30 de Março de 2015.

KOWALSKI, Michal. **6502 Macro Assembler & Simulator**. 2003. Disponível em: < http://exifpro.com/utils.html> Acesso em: 20 de Junho de 2015.

GREENSTED, Andrew. **The Lab Book Pages – An online collection of electronics information**. 2010. Disponível em: < http://www.labbookpages.co.uk/electronics > Acesso em: 15 de Novembro de 2015.

ADELIKAT. **The all in one NES/Famicom Emulator**. 2013. Disponível em: < http://www.fceux.com/web/home.html> Acesso em: 30 de Julho de 2015.

CHU, Pong P. **RTL HARDWARE DESIGN USING VHDL**. 1. Ed. Hoboken, New Jersey: John Wiley & Sons, Inc. 2006.

MATTHEWS, Ian. **Commodore 64 – The Best Selling Computer in History**. 2003. Disponível em: < http://www.commodore.ca/commodore-products/commodore-64-the-best-selling-computer-in-history/ > Acesso em: 15 de Novembro de 2015.

COMPUTER HISTORY MUSEUM. **The Apple II**. 1996 - 2015. Disponível em: < http://www.computerhistory.org/revolution/personal-computers/17/300 > Acesso em: 10 de Novembro de 2015.

7 ANEXO A – TRECHO DE CÓDIGO *ASSEMBLY* DO PROGRAMA *ALLSUITE.ASM*

```
        .ORG $4000
start
; EXPECTED FINAL RESULTS: $0210 = FF
; (any other number will be the
;  test that failed)
; initialize:
        LDA #$00
        STA $0210
        ; store each test's expected
        LDA #$55
        STA $0200
        LDA #$AA
        STA $0201
        LDA #$FF
        STA $0202
        LDA #$6E
        STA $0203
        LDA #$42
        STA $0204
        LDA #$33
        STA $0205
        LDA #$9D
        STA $0206
        LDA #$7F
        STA $0207
        LDA #$A5
        STA $0208
        LDA #$1F
        STA $0209
        LDA #$CE
        STA $020A
        LDA #$29
        STA $020B
        LDA #$42
        STA $020C
        LDA #$4C   ; Changed from 6C (Simulator Result)
        STA $020D
        LDA #$42
        STA $020E


; expected result: $022A = 0x55
test00
        LDA #85
        LDX #42
        LDY #115
        STA $81
        LDA #$01
        STA $61
        LDA #$7E
        LDA $81
        STA $0910
        LDA #$7E
        LDA $0910
        STA $56,X
        LDA #$7E
```

```
        LDA $56,X
        STY $60
        STA ($60),Y
        LDA #$7E
        LDA ($60),Y
        STA $07ff,X
        LDA #$7E
        LDA $07ff,X
        STA $07ff,Y
        LDA #$7E
        LDA $07ff,Y
        STA ($36,X)
        LDA #$7E
        LDA ($36,X)
        STX $50
        LDX $60
        LDY $50
        STX $0913
        LDX #$22
        LDX $0913
        STY $0914
        LDY #$99
        LDY $0914
        STY $2D,X
        STX $77,Y
        LDY #$99
        LDY $2D,X
        LDX #$22
        LDX $77,Y
        LDY #$99
        LDY $08A0,X
        LDX #$22
        LDX $08A1,Y
        STA $0200,X

; CHECK test00:
        LDA $022A
        CMP $0200
        BEQ test00pass
        JMP theend
test00pass
        LDA #$FE
        STA $0210
…
; Código completo disponível em: github.com/bernardo-andreeti/6502
```

8 ANEXO B – TABELA DE INSTRUÇÕES IMPLEMENTADAS

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK	ORA ind, X			**TSB zpg**	ORA zpg	ASL zpg		PHP	ORA imm	ASL A		**TSB abs**	ORA abs	ASL abs		0
1	BPL rel	ORA ind, Y	**ORA ind**		**TRB zpg**	ORA zpg, X	ASL zpg, X		CLC	ORA abs, Y	**INC A**		**TRB abs**	ORA abs, X	ASL abs, X		1
2	JSR abs	AND ind, X			BIT zpg	AND zpg	ROL zpg		PLP	AND imm	ROL A		BIT abs	AND abs	ROL abs		2
3	BMI rel	AND ind, Y	**AND ind**		**BIT zpg,X**	AND zpg, X	ROL zpg, X		SEC	AND abs, Y	**DEC A**		**BIT abs,X**	AND abs, X	ROL abs, X		3
4	RTI	EOR ind, X				EOR zpg	LSR zpg		PHA	EOR imm	LSR A		JMP abs	EOR abs	LSR abs		4
5	BVC rel	EOR ind, Y	**EOR ind**			EOR zpg, X	LSR zpg, X		CLI	EOR abs, Y	**PHY**			EOR abs, X	LSR abs, X		5
6	RTS	ADC ind, X			**STZ zpg**	ADC zpg	ROR zpg		PLA	ADC imm	ROR A		JMP ind	ADC abs	ROR abs		6
7	BVS rel	ADC ind, Y	**ADC ind**		**STZ zpg,X**	ADC zpg, X	ROR zpg, X		SEI	ADC abs, Y	**PLY**		**JMP ind,X**	ADC abs, X	ROR abs, X		7
8	**BRA rel**	STA ind, X			STY zpg	STA zpg	STX zpg		DEY	**BIT imm**	TXA		STY abs	STA abs	STX abs		8
9	BCC rel	STA ind, Y	**STA ind**		STY zpg, X	STA zpg, X	STX zpg, Y		TYA	STA abs, Y	TXS		**STZ abs**	STA abs, X	**STZ abs,X**		9
A	LDY imm	LDA ind, X	LDX imm		LDY zpg	LDA zpg	LDX zpg		TAY	LDA imm	TAX		LDY abs	LDA abs	LDX abs		A
B	BCS rel	LDA ind, Y	**LDA ind**		LDY zpg, X	LDA zpg, X	LDX zpg, Y		CLV	LDA abs, Y	TSX		LDY abs, X	LDA abs, X	LDX abs, Y		B
C	CPY imm	CMP ind, X			CPY zpg	CMP zpg	DEC zpg		INY	CMP imm	DEX		CPY abs	CMP abs	DEC abs		C
D	BNE rel	CMP ind, Y	**CMP ind**			CMP zpg, X	DEC zpg, X		CLD	CMP abs, Y	**PHX**			CMP abs, X	DEC abs, X		D
E	CPX imm	SBC ind, X			CPX zpg	SBC zpg	INC zpg		INX	SBC imm	NOP		CPX abs	SBC abs	INC abs		E
F	BEQ rel	SBC ind, Y	**SBC ind**			SBC zpg, X	INC zpg, X		SED	SBC abs, Y	**PLX**			SBC abs, X	INC abs, X		F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Note: ■ = New Op Codes

NOTA: *Opcodes* em branco representam o conjunto de instruções original do processador 6502, sendo que todas foram implementadas neste projeto. Os *opcodes* destacados em preto representam instruções adicionadas nas revisões do processador como a versão 65C02 e por esse motivo não foram implementadas.

9 APÊNDICE I – EXEMPLO DE DOCUMENTAÇÃO DE MICRO OPERAÇÕES

LOAD AND STORE GROUP:

- Addressing Modes: Absolute, Zero Page, Immediate, Absolute_X, Absolute_Y, Zero Page_X, Indexed Indirect, Indirect Indexed;
- Lenght: 2 to 3 bytes;
- Cycles: 3 to 7;

States shared between different addressing modes:

```
ALL
----
   T0: MAR <- PC ;  PC++                          # Fetch
   T1: MAR <- PC ; IR <- MEM[MAR] ; PC++;         # First Decode step, IR receives opcode
```

Specific States:

```
LDA
----
   - IMM: LDA #01 (a9 01)                         # AC <- 01
      T2: AC <- MEM[MAR]; wrn; wrz;               # Via DB and SB

   - Z-PAGE: LDA $33 (a5 33)                      # AC <- MEM[33h]
      T2: MAR <- MEM[MAR];                        # DB holds the address of value to be written into AC
      T3: AC <- MEM[MAR]; wrn; wrz;               # Via DB and SB

   - Z-PAGE,X: LDA $13 (b5 13)                    # AC <- MEM[13h+X]
      T2: BI <- MEM[MAR]; AI <- X
      T3: ABL <- AI + BI;
      T4: AC <- MEM[ABH/ABL]; wrn; wrz;           # Via DB and SB

   - (IND,X): LDA ($36,X) (a1 36)                 # AC <- MEM[MEM[36h+X]] : Results in zero page address!
      T2: BI <- MEM[MAR]; AI <- X
      T3: ABL <- AI + BI; ADH <- 0
      T4: ABL <- AI + BI + 1; BI <- MEM[ABH/ABL]; AI <- 0;
      T5: ABL <- AI + BI; ABH <- MEM[ABH/ABL]
      T6: AC <- MEM[ABH/ABL]; wrn; wrz;           # Via DB and SB

   - (IND),Y: LDA ($A5),Y (b1 a5)                 # AC <- MEM[MEM[a5h]+Y]
      T2: MAR <- MEM[MAR]; BI <- MEM[MAR]; AI <- 0
      T3: MAR <- AI + BI + 1; BI <- MEM[MAR]; AI <- Y
      T4: ABL <- AI + BI; BI <- MEM[MAR]; AI <- 0
      T5: ABH <- AI + BI + hc;
      T6: AC <- MEM[ABH/ABL]; wrn; wrz;           # Via DB and SB

   - ABS: LDA $0001 (ad 01 00)                    # AC <- MEM[0001h]
      T2: ABL <- MEM[MAR]; MAR <- PC; PC++;
      T3: ABH <- MEM[MAR];
      T4: AC <- MEM[ABH/ABL]; wrn; wrz;           # Via DB and SB

   - ABS,X: LDA $12F0,X (bd F0 12)                # AC <- MEM[12F0h+X]
      T2: BI <- MEM[MAR]; AI <- X; MAR <- PC; PC++;   # BI <- LOW ADDRESS BYTE
      T3: ABL <- AI + BI; BI <- MEM[MAR]; AI <- 0;    # BI <- HIGH ADDRESS BYTE
      T4: ABH <- AI + BI + hc;                        # ABH & ABL <- [12F0h+X]
      T5: AC <- MEM[ABH/ABL]; wrn; wrz;               # Via DB and SB

   - ABS,Y: LDA $12F0,Y (b9 F0 12)            # AC <- MEM[12F0h+Y]
```

```
T2: BI <- MEM[MAR]; AI <- Y; MAR <- PC; PC++;      # BI <- LOW ADDRESS BYTE
T3: ABL <- AI + BI; BI <- MEM[MAR]; AI <- 0;       # BI <- HIGH ADDRESS BYTE
T4: ABH <- AI + BI + hc;                           # ABH & ABL <- [12F0h+Y]
T5: AC <- MEM[ABH/ABL]; wrn; wrz;                  # Via DB and SB
…
# Documentação completa disponível em: github.com/bernardo-andreeti/6502
```

10 APÊNDICE II – TUTORIAL PARA A PROTOTIPAÇÃO DO PROCESSADOR E *BLOCK RAM*

- Baixar arquivos do repositório 6502: https://github.com/bernardo-andreeti/6502
- Descompactar arquivo ZIP baixado
- Criar um novo projeto no software ISE ou PlanAhead e importar **todo** o diretório "VHDL" (contido nos arquivos do repositório) na etapa de importação de códigos fonte
- Arquivo de *constraints* (.ucf) e imagem de memória (.txt) encontram-se no diretório "**..VHDL/SuportePrototipacao**"
- Importar o arquivo **P6502_BRAM.ucf** durante a criação do projeto
- Copiar arquivo de imagem de memória para o diretório "**..VHDL/SuportePrototipacao**" do projeto criado para que a *Block RAM* seja inicializada
- **IMPORTANTE:** Certifique-se de definir o arquivo P6502_RAM_tb.vhd para ser utilizado **somente** em simulação, dessa forma o arquivo P6502_RAM.vhd assume a posição de entidade de topo automaticamente!
- Executados corretamente os passos anteriores, deve-se obter a seguinte estrutura:

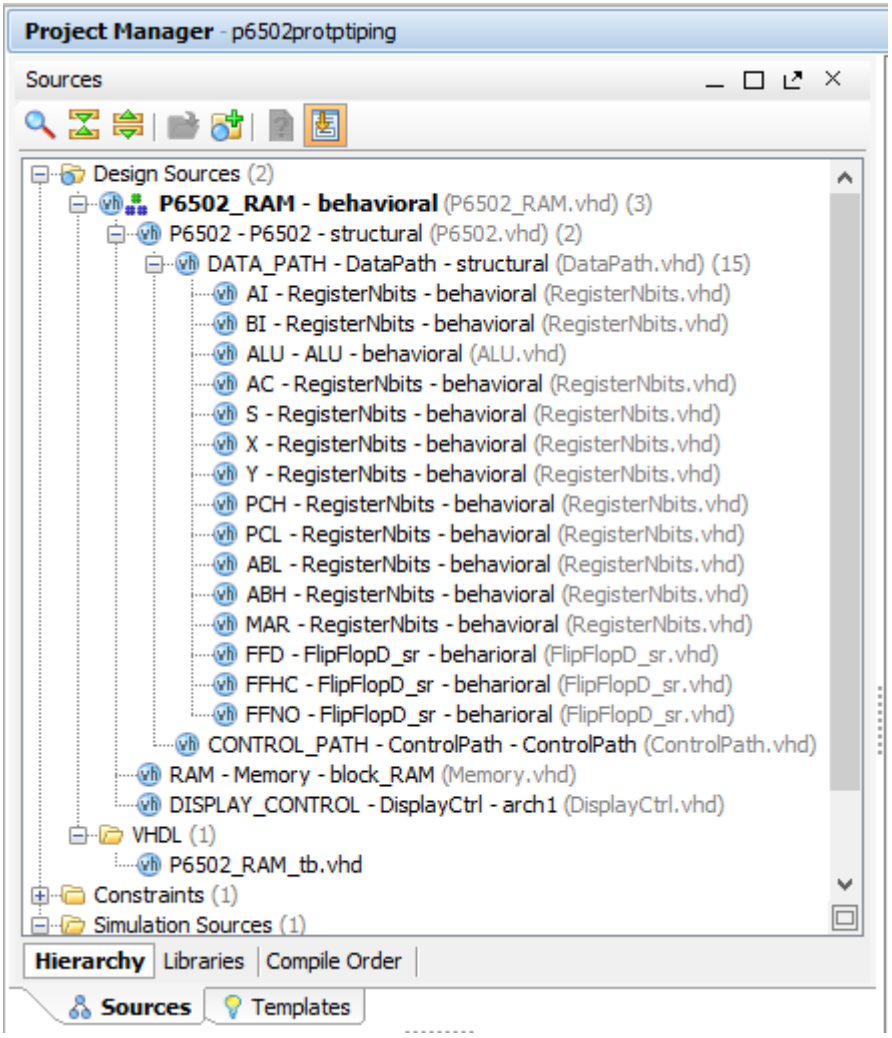

Figura 21 – Estrutura de arquivos obtida em prototipação.

- Proceder com as etapas de síntese, implementação e geração de *bitsream* (Dica: clicar diretamente na geração de *bitsream*, as etapas anteriores são realizadas automaticamente)
- Enviar *bitstream* para a FPGA utilizando o software Chip Scope Pro
- Interagir com o processador conforme a estrutura da Figura 14

11 APÊNDICE III – TUTORIAL DE UTILIZAÇÃO DO PROJETO *FPGA_NES*

- Baixe os arquivos do repositório 6502: https://github.com/bernardo-andreeti/6502
- Baixe os arquivos do repositório fpga_nes: https://github.com/bernardo-andreeti/fpga_nes
- Copie **todo** o conteúdo do diretório "**../6502/VHDL/P6502**" para o diretório "**../fpga_nes/hw/src/cpu**"
- Abrir o projeto do software ISE no diretório "**../fpga_nes/hw/ise**"
- **IMPORTANTE:** Antes de iniciar o processo de síntese, altere o valor inicial de **PCH** no arquivo DataPath.vhd para **128** de forma que o endereço inicial de PC seja 0x8000 (início da memória de programa do NES)
- Sintetizar, implementar e gerar *bitstream*
- Enviar *bitstream* para a FPGA utilizando o software Chip Scope Pro
- Montar a seguinte estrutura de conexões:

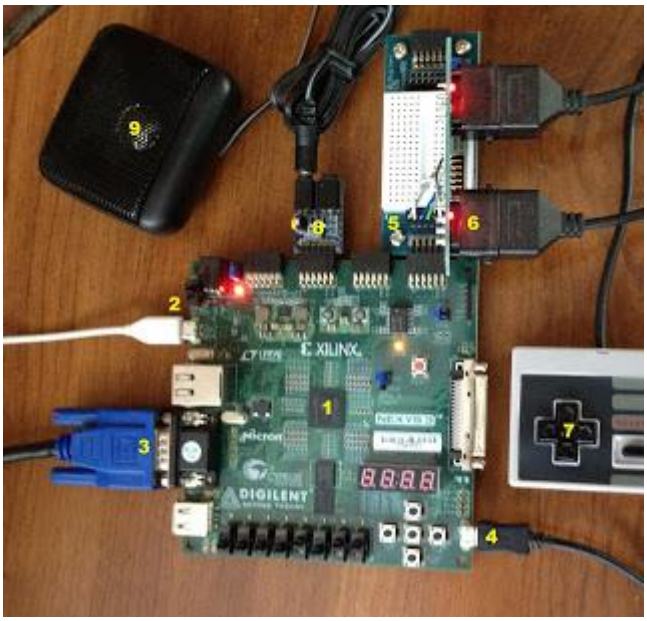

Figura 22 – Estrutura de conexões do projeto *fpga_nes*.

- Identificação das conexões:
    - 2 – Cabo USB de envio de *bitstream*
    - 3 – Cabo VGA com sinal de saída de vídeo
    - 4 – Cabo USB para envio de ROMS
    - 5, 6 e 7 – Interface de *Joysticks*
    - 8 e 9 – Interface de áudio
- Deve-se destacar que são obrigatórias apenas as conexões **2**, **3** e **4** para o funcionamento básico do console, sendo que as demais representam apenas o suporte para periféricos
- Montada a estrutura de conexões básica, pode-se carregar os jogos em ROMS através do software NESDBG, disponível no diretório "**../fpga_nes/sw/NESDBG Built**"
- Para carregar as ROMS, basta acessar o menu *File*, clicar na opção *Load ROM*, e escolher o jogo no diretório "**..fpga_nes/sw/roms/game_roms/supported**"