**UNIVERSIDADE FEDERAL DE SERGIPE**
**DEPARTAMENTO DE COMPUTAÇÃO**

**PROCESSO SELETIVO CORETECH**
**GRUPO 6**

# DOCUMENTAÇÃO PQP VERILOG
# (DESAFIO A/B)

**DISCENTES:**
**DAVÍ ANTONIO MARTINS RIBEIRO**
**DAVI ARAÚJO DO NASCIMENTO**
**GUSTAVO GOMES TAVARES**
**MATEUS ARANHA ROCHA**
**PEDRO JOAQUIM SILVA SILVEIRA**

**São Cristóvão - SE**
**03/11/2025**

# 1. INTRODUÇÃO

Os processadores são peças fundamentais para área da computação, mas muitas vezes os conceitos e técnicas que os formam são deixados de lado no aprendizado dessa área do conhecimento, em favor do ensino de linguagens de programação de alto nível e formas de se estruturar e organizar um software. Visto a necessidade de se conhecer a fundo os funcionamentos desse pequeno pedaço de metal complexo, o grupo 6 do processo seletivo da Liga Acadêmica CoreTech, também conhecido como “Flasco”, resolveu implementar um processador de uma arquitetura denominada PicoQuickProcessor (PQP). Essa arquitetura simples caracterizada por poucas instruções e ciclo único é perfeita para os integrantes do grupo que estão adentrando o mundo do hardware voltado à computação.

O Flasco implementou sua instância da PQP na linguagem de descrição de hardware Verilog, ela é usada para projetar e simular circuitos digitais utilizando de uma sintaxe concisa e clara, dando um controle e poder imenso àqueles que a usam.

Este documento contém todas as informações necessárias para uso e entendimento da arquitetura e implementação. O código Verilog também estará disponível para uso, inspeção e modificação, permitindo a avaliação do projeto perante à bancada da Liga Acadêmica CoreTech e aos demais interessados.

# 2. DETALHES DA IMPLEMENTAÇÃO

## 2.1. Especificações básicas

A arquitetura PicoQuickProcessor é de 32 bits, que utiliza ordem *little endian* para o armazenamento e interpretação de dados. Há 16 registradores de 32 bits, bem como uma memória de 256 bytes (2048 bits), que segue a arquitetura Von Neumann, de forma que a memória de instruções e a memória de dados são compartilhadas.

O processador é implementado como ciclo-único, de forma que todas as etapas de uma instrução são executadas em um ciclo de *clock*.

## 2.2. Instruções e opcodes

A implementação atual tem compatibilidade com apenas 9 das 16 instruções especificadas pela documentação oficial da PQP, com adição de uma especial. Elas estão especificadas abaixo:

### 2.2.1. Modelo da instrução

As instruções da PQP são palavras de 32 bits divididas em 4 campos, o primeiro contém 8 bits que armazenam o opcode (código de operação) da instrução. Logo em seguida vem dois campos de 4 bits, ambos são usados na maioria das instruções para indexar dois dos 16 registradores. E por último um campo de 16 bits que pode ser usado para passar um inteiro com sinal também de 16 bits, também chamado de imediato. Nenhuma das instruções usam todos os campos de uma vez, quando o campo é inutilizado recebe um valor de don't care, ou na tradução "não se importe", ou seja, independente do valor presente, ele não será usado.

### 2.2.2. MOV $r_x$, $i_{16}$

Move o imediato da instrução para registrador usando extensão de sinal na conversão dos 16 para 32 bits.

| | | | |
|---|---|---|---|
| 0x00 | [0x0 .. 0xF] | don't care | [-0x8000 .. 0x7FFF] |

### 2.2.3. MOV $r_x$, $r_y$

Move o valor do registrador da instrução de $r_x$ para $r_y$.

| | | | |
|---|---|---|---|
| 0x01 | [0x0 .. 0xF] | [0x0 .. 0xF] | don't care |

### 2.2.4. MOV $r_x$, $[r_y]$

Move o valor na memória representado pelo valor de $r_y$ para o registrador $r_x$, correspondendo a uma operação de carregamento da memória.

| | | | |
|---|---|---|---|
| 0x02 | [0x0 .. 0xF] | [0x0 .. 0xF] | don't care |

### 2.2.5. **MOV** $[r_x]$**,** $r_y$

Move o valor de $r_y$ para o valor na memória correspondente a $r_x$, correspondendo a uma operação de salvamento na memória.

| | | | |
|---|---|---|---|
| 0x03 | [0x0 .. 0xF] | [0x0 .. 0xF] | don't care |

### 2.2.6. **JMP** $\mathbf{i_{16}}$

Adiciona ao *program counter* (PC) o valor 4, representando o comprimento da instrução, junto ao valor do imediato, que passa pela conversão de sinal de 16 para 32 bits .

| | | | |
|---|---|---|---|
| 0x05 | don't care | don't care | [-0x8000 .. 0x7FFF] |

### 2.2.7. **ADD** $r_x$**,** $r_y$

Adiciona ao valor do registrador $r_x$ o valor de $r_y$**.**

| | | | |
|---|---|---|---|
| 0x09 | [0x0 .. 0xF] | [0x0 .. 0xF] | don't care |

### 2.2.8. **SUB** $r_x$**,** $r_y$

Subtrai do valor do registrador $r_x$ o valor de $r_y$**.**

| | | | |
|---|---|---|---|
| 0x0A | [0x0 .. 0xF] | [0x0 .. 0xF] | don't care |

### 2.2.9. **AND** $r_x$**,** $r_y$

Realiza uma operação lógica AND *bitwise* entre os valores de $r_x$ e $r_y$**,** armazenando o resultado em $r_x$**.**

| | | | |
|---|---|---|---|
| 0x0B | [0x0 .. 0xF] | [0x0 .. 0xF] | don't care |

### 2.2.10. **OR** $r_x$**,** $r_y$

Realiza uma operação lógica OR *bitwise* entre os valores de $r_x$ e $r_y$**,** armazenando o resultado em $r_x$**.**

| | | | |
|---|---|---|---|
| 0x0C | [0x0 .. 0xF] | [0x0 .. 0xF] | don't care |

#### 2.2.11. HALT

Termina a execução das instruções do programa.

| | | | |
|---|---|---|---|
| 0xFF | don't care | don't care | don't care |

### 2.3. Ciclo de instrução

O ciclo de instrução do processador foi implementado da seguinte forma:

1. O endereço da instrução é carregado no pc, que é então passado para o módulo "memory.v".
2. Por meio deste endereço, a instrução é passada para o módulo "top.v", que imediatamente passa o início da instrução, contendo o opcode, para a unidade de controle
3. Com o *opcode*, os sinais de controle da memória, dos registros, e outros, são definidos por "control.v" de acordo com a instrução passada.
4. Os endereços dos registradores, ou valores imediatos, contidos na instrução, são enviados para o módulo "registradores.v", bem como os sinais de escrita.
5. Caso a operação envolva a Unidade Lógica e Aritmética, os endereços dos registradores são enviados para "ula.v", bem como o *opcode*, que determina a operação realizada. Caso a operação seja de pular instruções, o módulo "top.v" atualiza o PC para o endereço desejado.
6. Na operação de carregamento da memória, o módulo "memory.v" recebe o endereço para obter o dado, bem como o registrador cujo dado será salvo, retornando o valor para esse registrador. Caso seja uma operação de salvamento, ela recebe o valor do registrador correspondente, e o endereço a ser salvo.
7. Por fim, os dados são escritos no banco de registradores pelo endereço lido na decodificação da instrução, finalizando o ciclo de instrução.

### 2.4. Lógica de decodificação e controle

Para o processo de decodificação de instruções, partes da instrução são enviadas para diferentes locais no módulo "top.v". Uma instrução é separada da seguinte forma:

| [31:24] (8 bit) | [23:20] (4 bit) | [19:16] (4 bit) | [15:0] (16 bit) |
|---|---|---|---|
| $OPCODE$ | $R_x$ | $R_y$ | $Immediate$ |

O opcode é enviado para a unidade de controle, para a ativação dos sinais de controle necessários, bem como para a Unidade Lógica e Aritmética, para a realização necessária baseado na instrução.

Os endereços dos registradores contidos na instrução são enviados para o banco de registradores, e caso necessário, são enviados para a memória para carregamento ou salvamento de dados.

Para qualquer operação realizada com um valor imediato passado na instrução, seu valor de 16 bit é estendido para 32 bits.

A lógica de controle, definida no módulo “control.v”, define os sinais a serem ativados ou desativados, e, consequentemente, os caminhos a serem seguidos no *datapath*. Os sinais de controle definidos para cada instrução estão disponíveis no Anexo I.

## 2.5. Diagrama de bloco

O diagrama de blocos é uma forma visual de entender como um sistema digital funciona. Os blocos possuem entradas e saídas com conexões entre si, que dizem de onde as informações saem e para onde vão, sendo esse fluxo representado por uma seta. Além disso, bifurcações nos fios são representadas por um ponto acima da bifurcação. Abaixo temos o diagrama de blocos do *datapath* da PQP:

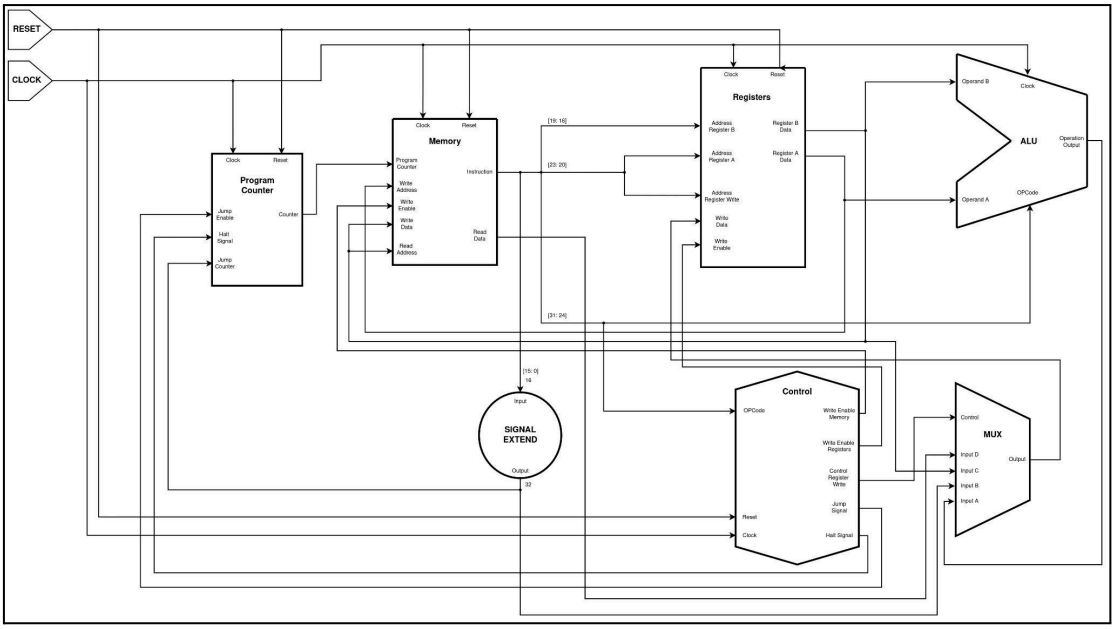



Diagrama de blocos do PQP

# 3. MÓDULOS VERILOG

## 3.1. Registradores

O módulo “registradores.v” descreve o banco de registradores utilizado para executar as instruções do processador. Há um total de 16 registradores de 32 bits.

O módulo possui entradas para o *clock* e para resetar o banco de registradores para o estado inicial, bem como entradas para os endereços dos registradores $r_x$ e $r_y$ a serem lidos, o endereço do registrador em que o dado será escrito, o dado a ser escrito no registrador e um sinal de habilitação de escrita. As saídas do módulo contém os valores armazenados em $r_x$ e $r_y$ que serão manipulados.

Quando o sinal de *reset* é ativado, o banco de registradores é reiniciado para o estado original, em que, para todos os registradores, o valor definido é 0. Caso a borda do *clock* seja detectada e a escrita esteja habilitada, o dado contido na entrada de dados do banco de registradores será armazenado no endereço selecionado.

Ao fim do módulo, estão as partes responsáveis por fornecer os dados contidos nos registradores selecionados baseados no endereço.

## 3.2. Memória

O módulo “memory.v” representa a memória do sistema, possuindo 256 bytes de armazenamento, possibilitando que sejam armazenadas até 64 palavras de 32 bits. Seguindo a arquitetura Von Neumann, a memória é utilizada tanto para armazenar o conjunto de instruções quanto para carregar e salvar dados.

O módulo possui entradas para o *clock* e *reset*. Junto a isso, há uma entrada para o *program counter* (PC), contendo o endereço para a próxima instrução, bem como entradas para os endereços de carregamento e salvamento de dados na memória e para o dado a ser guardado no endereço. Por fim, o módulo possui um sinal para habilitar e desabilitar a escrita.

As saídas são compostas pela instrução buscada com o PC, e pela palavra obtida na operação de carregamento da memória.

O funcionamento se dá de forma similar ao banco de registradores, de forma que, caso o sinal de *reset* seja habilitado, a memória é retornada para o estado inicial, bem como, caso o sinal de escrita seja habilitado, na borda do *clock*, o dado desejado é escrito no endereço passado na entrada.

Os endereços de memória são de 32 bits, sendo utilizados os valores do bit na nona posição ao bit na segunda posição, possibilitando a representação de 256 endereços diferentes, de forma que cada leitura ou escrita ocorre de 32 em 32 bits (uma palavra inteira).

### 3.3. Unidade Lógica e Aritmética

O módulo “ula.v” descreve, com modelagem dataflow, a Unidade Lógica e Aritmética, responsável por operações utilizadas em determinadas instruções, baseado no opcode dessas instruções.

As entradas são compostas pelo *clock*, o opcode da instrução para a seleção da operação, as entradas de dados a serem operados, e a saída do dado resultante da operação.

As operações implementadas foram a soma, a subtração, o AND e o OR lógicos *bitwise*, selecionados pelos seus respectivos opcodes.

### 3.4. Controle

O módulo “control.v” controla as operações a serem realizadas no processador a partir do opcode das instruções. Com o opcode, os sinais de escrita nos registros e na memória, bem como os sinais para as instruções JMP e HALT são controlados.

As entradas são compostas pelo clock, pelo sinal de reset, e pelo opcode da instrução. As saídas são os sinais para habilitar escrita na memória, escrita nos registradores, o que será escrito no registrador, bem como o sinal de HALT e de JMP. A tabela contendo o sinal de cada instrução está disponível no Anexo I.

### 3.5. Top level module

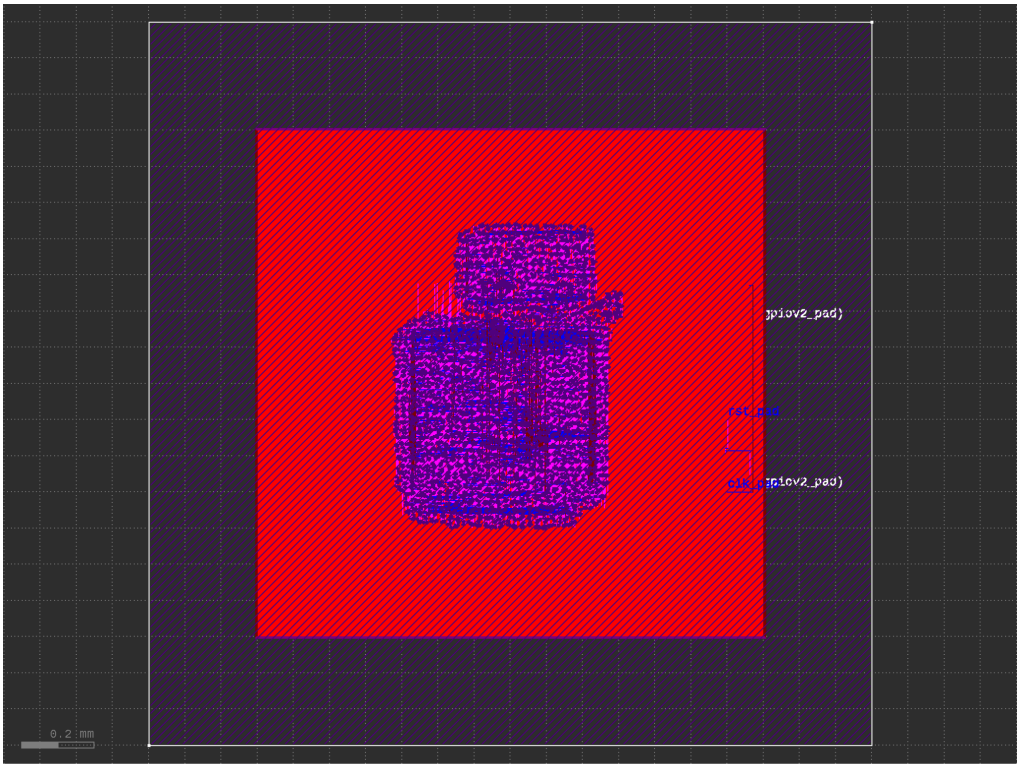
O módulo “top.v” é o módulo responsável por interligar todos os módulos do *datapath* do processador, bem como atualizar o PC baseado na execução sequencial das instruções, da instrução JMP.

O módulo conta com entradas para o *clock* e *reset,* para restaurar o processador ao estado inicial, bem como saídas para depuração do PC, das instruções, e da saída da Unidade Lógica e Aritmética.

Aqui, são instanciados os módulos da ULA, do banco de registradores, da memória, e da unidade de controle, bem como fios para as interconexões entre eles. Nesse módulo, também é definido de onde o valor a ser escrito em um registrador virá, seja da ULA, seja de um valor imediato de 16 bits com extensão de sinal para 32 bits, de outro registrador, ou da memória. Os códigos para definição desses valores estão disponíveis no Anexo II e complementam as operações de controle do Anexo I.

## 4. MODELO FÍSICO DO PROCESSADOR

Com os módulos do processador prontos, é possível produzir um arquivo de um modelo físico dele. Os arquivos resultantes do *Place and Route*, bem como a versão final estão disponíveis na pasta “Físico” do repositório do processador, e podem ser acessados com o *software* Klayout. Abaixo, é possível visualizar uma imagem do modelo físico do processador pronto para fabricação (arquivo .gds):



Processador pronto para fabricação visualizado no Klayout

## 5. FORMA DE USO

### 5.1. Dependências

Para executar a simulação e visualizar seu resultado se faz necessária a instalação das seguintes dependências:

- Icarus Verilog
- GTKWave
- GNU Make
- Git

Todos esses programas estão disponíveis para os sistemas operacionais Linux e Windows. Eles podem ser facilmente instalados usando os gerenciadores de pacotes da maioria das distribuições Linux, ou podem ser instalados manualmente, um a um, utilizando-se dos instaladores para Windows.

### 5.2. Execução

O código-fonte do projeto vem junto de um arquivo makefile, um script que resume a instalação a poucos comandos. Esses estão listados abaixo, um por linha, eles devem ser executados em ordem, de cima para baixo. O primeiro comando se faz necessário apenas se o diretório com os arquivos Verilog do projeto ainda não foram baixados. Não há necessidade de limpeza após a execução, pois arquivos residuais são automaticamente deletados pelo GNU Make.

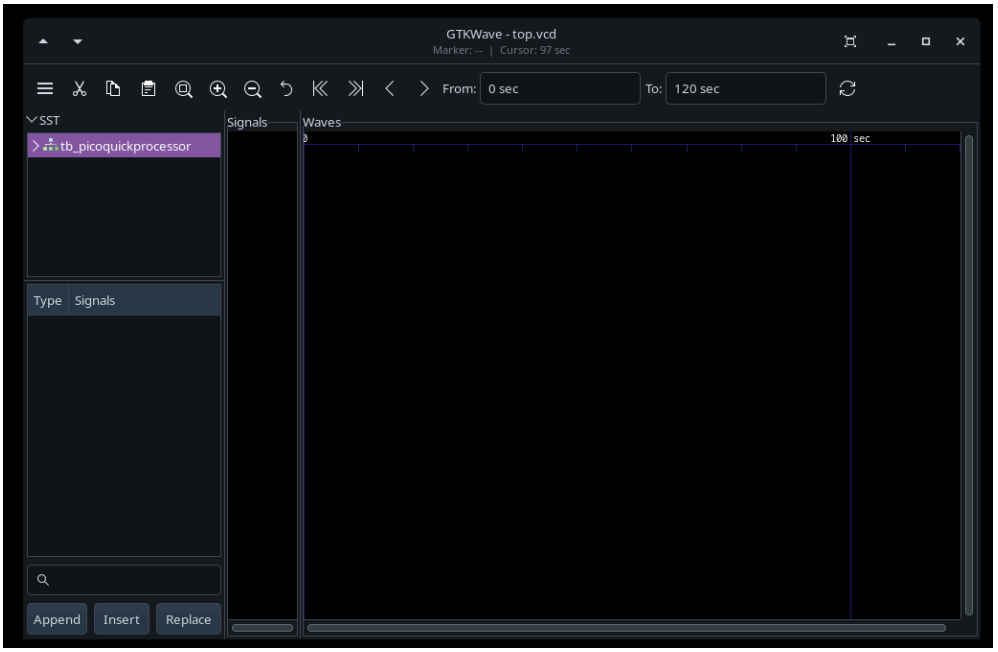
```
git clone https://github.com/PedroJSS05/PicoQuickProcessor_verilog
–depth 1

cd PicoQuickProcessor_verilog

make
```

### 5.3. Visualizando os resultados

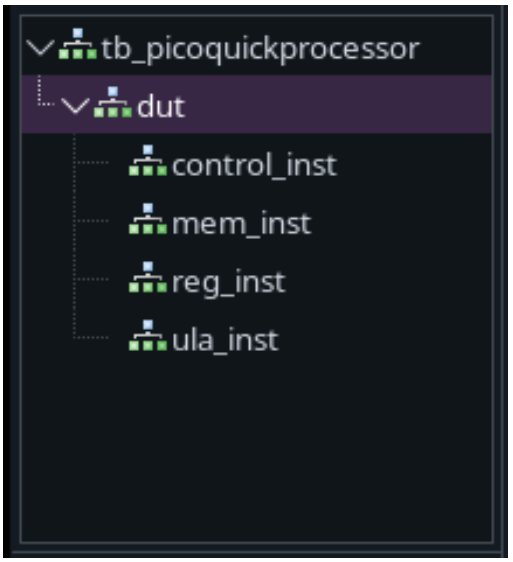
Após a execução do makefile, a janela do GTKWave se abrirá, ela deve aparecer de forma similar à figura abaixo:




Tela inicial do GTKWave

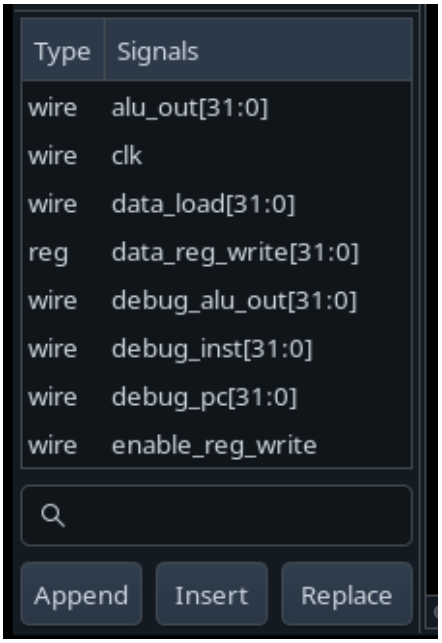
O centro da janela é onde os sinais e valores serão exibidos, a barra no topo possui algumas opções para ajustar a visualização e na barra à esquerda existem os módulos e seus componentes para serem adicionados à janela central para visualização e análise de seus sinais.

Os componentes na esquerda podem ser expandidos para revelar mais sobre a hierarquia de subcomponentes que os compõem.




Componentes disponíveis para verificação no GTKWave

Ao clicar em qualquer um desses componentes, na seção imediatamente abaixo, aparecerão os sinais para serem analisados, junto dos seus respectivos nomes e tipo.




Sinais disponíveis em um componente no GTKWave

Após a seleção de qualquer um desses sinais, eles podem ser adicionados ao visualizador de ondas usando os botões Append, Insert e Replace.


Visualização de sinais obtidos a partir dos estímulos

### 5.4. Escrevendo programas customizados

Dentro da pasta Main do projeto é possível encontrar um arquivo chamado “program.mem”. Esse arquivo inicializa a memória da simulação da arquitetura PQP permitindo escrever os programas que serão carregados e executados durante a simulação.

No arquivo são escritas as representações hexadecimais do que será carregado na memória. Não há necessidade de alguma formatação específica dos bytes, porém, é interessante que a cada palavra escrita, ou seja, 4 bytes, seja feita uma quebra de linha. A seção 2.2 deste documento pode servir como referência para a escrita das instruções em hexadecimal. Após a escrita do programa e salvamento do arquivo, basta executar o makefile.

## 6. DESAFIOS ENCONTRADOS

Durante o andamento do projeto, alguns desafios foram encontrados e superados, como, por exemplo, a criação dos módulos e a depuração deles, que primeiro recebeu uma avaliação manual de cada parte antes da criação do *testbench* de avaliação do processador como um todo.

Outro desafio encontrado foi a criação do testbench, com a necessidade de um aprendizado mais profundo do Verilog, bem como o entendimento sobre como cada módulo do processador funciona e como eles se relacionam entre si, para que assim a estrutura dos testes fossem definidas. Nesse processo, a maior dificuldade foi compreender conflitos para a escrita de dados na memória, crucial para a execução do *testbench* como um todo. Por fim, a comparação entre os resultados do modelo de referência e do *DUT* tornou necessária uma verificação manual de cada resultado.

Para a produção da documentação, o mais complexo foi ler as instruções, interpretar o passo-a-passo de cada instrução, para assim, determinar o funcionamento de cada uma, bem como descrever o comportamento entre os módulos.

## 7. CONCLUSÃO

Em conclusão, o projeto foi concluído por inteiro, com todos os módulos, documentação e *testbenches* funcionais, alcançando as

expectativas de todos os integrantes do grupo, e proporcionando aprendizado a todos. As dificuldades foram tratadas como situações que podem vir a ser recorrentes no desenvolvimento de *Hardware* e de *Software*, e assim, contribuíram para a formação de todos.

**ANEXO I - Sinais da Unidade de controle por instrução**

| Instrução | Opcode | Escrita na memória | Escrita nos registros | Control op | HALT | Jump Enable |
|---|---|---|---|---|---|---|
| **MOV** $r_x$, $i_{16}$ | 0x00 | 0 | 1 | 01 | 0 | 0 |
| **MOV** $r_x$, $r_y$ | 0x01 | 0 | 1 | 10 | 0 | 0 |
| **MOV** $r_x$, $[r_y]$ | 0x02 | 0 | 1 | 11 | 0 | 0 |
| **MOV** $[r_x]$, $r_y$ | 0x03 | 1 | 0 | don't care | 0 | 0 |
| **JMP** $i_{16}$ | 0x05 | don't care | don't care | don't care | 0 | 1 |
| **ADD** $r_x$, $r_y$ | 0x09 | 0 | 1 | 00 | 0 | 0 |
| **SUB** $r_x$, $r_y$ | 0x0A | 0 | 1 | 00 | 0 | 0 |
| **AND** $r_x$, $r_y$ | 0x0B | 0 | 1 | 00 | 0 | 0 |
| **OR** $r_x$, $r_y$ | 0x0C | 0 | 1 | 00 | 0 | 0 |
| **HALT** | 0xFF | don't care | don't care | don't care | 1 | 0 |

**ANEXO II - Controle das entradas do registrador**

| Valor | Valor a ser escrito |
|---|---|
| **00** | Saída da ALU |
| **01** | Valor imediato convertido para 32 bits |
| **10** | Valor do registro $R_y$ em $R_x$ |
| **11** | Valor contido no endereço apontado na memória por $R_y$ em $R_x$ |