
**PODER EXECUTIVO**
**MINISTÉRIO DA EDUCAÇÃO**
**UNIVERSIDADE FEDERAL DE RORAIMA**
**DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO**

# ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

# RELATÓRIO DO PROJETO: PROCESSADOR INTELIGENTE

**DISCENTES:**
**Victor Deluca Almirante Gomes – 2201524401**
**Allan Cordeiro Rocha de Araújo – 2201524427**

**Março de 2017**
**Boa Vista/Roraima**


**PODER EXECUTIVO**
**MINISTÉRIO DA EDUCAÇÃO**
**UNIVERSIDADE FEDERAL DE RORAIMA**
**DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO**

# ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

# RELATÓRIO DO PROJETO: PROCESSADOR INTELIGENTE

**Março de 2017**
**Boa Vista/Roraima**

## Resumo

Este trabalho aborda o projeto e a implementação do processador IntelIgente, desenvolvido por Victor Deluca e Allan Cordeiro. Serão descritos aqui os componentes do processador e a forma como eles se conectam no datapath, o formato das instruções a serem executadas no processador e os testes que foram realizados para garantir que todas as unidades funcionam corretamente.

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# 1 Especificações

Nesta seção são apresentados e descritos cada um dos componentes do processador e suas funcionalidades, além de outros itens relativos ao desenvolvimento do projeto.

## 1.1 Plataforma de desenvolvimento

Para a implementação do processador IntelIgente foi utilizada a IDE Quartus Prime 16.1 Lite, e a linguagem utilizada foi VHDL.

## 1.2 Propriedades e limitações

O processador IntelIgente possui 16 registradores REG0, REG1 ... e REG16, dos quais nenhum é um registrador auxiliar ou reservado: É possível ler e escrever em todos eles livremente. A memória ROM é limitada a 16 instruções, cada uma das quais consiste em 16 bits. O sistema opera apenas com inteiros entre -8 e 7, portanto qualquer operação cujo resultado ultrapasse o limite irá gerar um resultado incorreto.

## 1.3 Conjunto de instruções

O processador IntelIgente possui 12 tipos de instruções diferentes divididas em várias categorias descritas abaixo. Os 4 bits mais significativos representam o opcode da instrução, enquanto os 12 outros assumem funções diferentes de acordo com o tipo de instrução.

- **Instruções de três registradores:** São as instruções mais simples, que realizam operações entre dois registradores e armazenam o resultado em um terceiro registrador. Consistem na maior parte das instruções do processador. São instruções de três registradores: ADD, SUB, AND, OR e XOR.

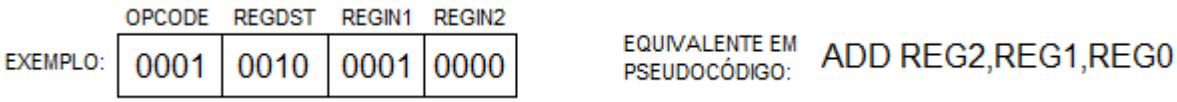


**Figura 1: Formato de uma instrução de três registradores e seu equivalente em pseudocódigo. REGDST simboliza o registrador destino, e os dois outros registradores, REGIN1 e REGIN2 representam as entradas.**

- **Instrução de dois registradores:** Instrução de formato similar ao das instruções de três registradores, porém utiliza apenas o valor de um registrador na entrada. A única instrução que utiliza este formato é o NOT. O segundo registrador de entrada é sempre “0000” por padrão, mas o valor lido nesse registrador nunca é utilizado.

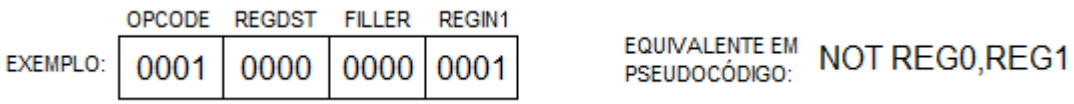


**Figura 2: Formato da instrução NOT e seu equivalente em pseudocódigo. O termo FILLER indica que os bits contidos na região são irrelevantes para a execução do programa.**

- **Instruções de operador constante:** Instruções que operam utilizando um ou dois registradores, e uma constante de 4 bits. As instruções LW e SW utilizam esse formato, com um registrador para a leitura do endereço a partir do offset e um registrador para receber a entrada (SW) ou armazenar o resultado (LW), dependendo da operação. A instrução LI também utiliza esse formato, porém apenas um registrador é utilizado, para armazenar o valor lido. O outro registrador recebe "0000" por padrão, mas nunca é utilizado.

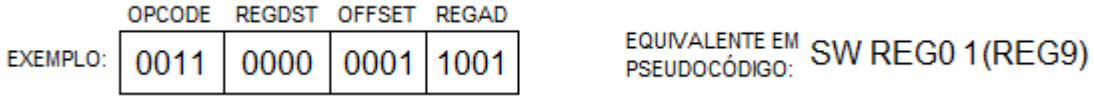


**Figura 3: Formato de uma instrução de operador constante com dois registradores. "OFFSET" é uma constante de 4 bits que representa o offset do endereço do valor a ser armazenado na memória RAM, sendo que o endereço é calculado da seguinte forma: (OFFSET + VALOR EM REGAD) MOD 16. O LI, instrução de operador constante com registrador único, tem a mesma estrutura, porém os últimos quatro bits (Na região do REGAD) nunca são utilizados.**

- **Instruções de salto:** Instruções que podem alterar o valor do PC, saltando a execução para uma instrução específica desejada. Existem dois tipos de salto: O salto incondicional (J) e o salto condicional (BEQ, BNE). No caso do salto condicional, a instrução possui dois registradores e os 4 bits menos significativos representam a "distância" do salto, ou seja, quantas instruções serão puladas a partir da próxima instrução, e o salto será realizado apenas quando a condição para o salto for cumprida. No caso do salto incondicional, o salto possui 12 bits. É importante notar que, como o salto é maior que o número total de bits, o endereço da nova instrução é dado por: (PC + 1 + distância do salto) mod 16.
- **Instrução de encerramento:** Última linha de comando do programa. Seu único elemento utilizado é o Opcode, sendo que os outros bits recebem '0' por padrão. Desabilita o PC, impedindo que o programa continue sua execução, conforme veremos mais adiante. É a instrução END.

São descritas abaixo, em detalhe, todas as instruções do processador:

| Instrução | Opcode | Exemplo | Equivalente em pseudocódigo |
|---|---|---|---|
| **ADD** | **0000** | **0000 0000 0001 0010** | **ADD R0, R1, R2** |
| Descrição: Executa uma soma entre os valores armazenados em R1 e R2, e armazena o resultado em R0. | | | |
| **SUB** | **0001** | **0001 0001 0000 0010** | **SUB R1, R0, R2** |
| Descrição: Subtrai o valor armazenado em R2 do valor armazenado em R0, e armazena o resultado em R1. | | | |
| **LW** | **0010** | **0010 0111 0011 0010** | **LW R7,3(R2)** |
| Descrição: Armazena em R7 o valor lido no endereço (3 + R2) MOD 16 da memória RAM | | | |
| **SW** | **0011** | **0011 0110 0010 0100** | **SW R6,2(R4)** |
| Descrição: Armazena no endereço (2 + R4) MOD 16 da memória RAM o valor lido em R6 | | | |
| **J** | **0100** | **0100 000000001000** | **J JUMP** |
| Descrição: Ignora as próximas 8 instruções | | | |
| **BEQ** | **0101** | **0100 0000 0001 1000** | **BEQ R0,R1,JUMP** |
| Descrição: Se o valor armazenado em R0 e o valor armazenado em R1 forem iguais, as próximas 8 instruções serão ignoradas | | | |
| **OR** | **0110** | **0110 0000 0001 1000** | **OR R0,R1,R8** |
| Descrição: Executa uma operação OR entre R1 e R8, e armazena o resultado em R0 | | | |
| **AND** | **0111** | **0111 0100 1000 1001** | **AND R4,R8,R9** |
| Descrição: Executa uma operação AND entre R8 e R9, e armazena o resultado em R4 | | | |

| | | | |
|---|---|---|---|
| **XOR** | **1000** | **1000 0000 0001 0010** | **XOR R0,R1,R2** |
| Descrição: Executa uma operação XOR entre R1 e R2, e armazena o resultado em R0 | | | |
| **NOT** | **1001** | **1001 0001 0000 0000** | **NOT R1,R0** |
| Descrição: Executa uma operação NOT sobre o registrador R0, e armazena o resultado em R1 | | | |
| **LI** | **1010** | **1010 0010 0100 0000** | **LI R2,4** |
| Descrição: Armazena o número 4 no registrador R2. | | | |
| **BNE** | **1011** | **1011 0000 0100 0010** | **BNE R0,R4,JUMP** |
| Descrição: Se os valores armazenados em R0 e R4 forem diferentes, as próximas 2 instruções serão ignoradas | | | |
| **END** | **1100** | **1100 0000 0000 0000** | **END** |
| Descrição: Desabilita o PC, encerrando a execução do programa. | | | |

**Tabela 1: Descrição das instruções do processador e como cada bit delas é utilizado, com exemplos.**

## 1.4. Descrição do Hardware

Nesta seção, serão descritos os componentes que formam o processador IntelIgente, bem como suas funções dentro do sistema.

### 1.4.1. PC

O componente PC (Program Counter) é o mais simples da estrutura, porém de vital importância para o funcionamento do sistema. É através dele que o processador reconhece qual é a próxima instrução a ser executada. O PC é iniciado em 0 e, a menos que uma instrução de salto seja executada, tem seu valor incrementado em 1 a cada ciclo de clock. Suas entradas são o próprio clock e a instrução atual, inicializada com 0, e sua única saída é a própria instrução atual.

### 1.4.2. Offsomador

O componente Offsomador é o responsável pelo incremento do valor de PC em 1 a cada ciclo de clock. Sua entrada principal é a instrução atual, mas ele também possui uma outra entrada regida pela unidade de controle (Descrita mais adiante), que define se o Program Counter deverá continuar avançando ou se o processador já chegou ao fim do programa. Essa entrada é chamada *enable*.

### 1.4.3. MemoriaROM

A memória ROM (MemoriaROM) é o local onde fica armazenado o programa a ser executado, sob a forma de um conjunto de instruções na forma binária. Sua entrada é o PC, e sua saída é a instrução correspondente ao valor de PC.

### 1.4.4. UnidadeControle

A unidade de controle (unidadeControle) é o componente que coordena os outros componentes do hardware para executarem a instrução desejada. Tal coordenação é obtida através de sinais de controle, que serão descritos abaixo:

- *ula_op* define a operação a ser executada na ULA (Que será descrita mais adiante). A maioria das instruções executa uma soma (ADD, LW, LI) ou uma subtração (SUB, BEQ, BNE), mas algumas instruções específicas de três registradores podem requisitar um OR, um AND, um XOR ou mesmo um NOT.
- *memtoreg* define se a instrução precisa ler algum valor na memória.

- *memwrite* define se a instrução precisa escrever algum valor na memória.
- *branch* define se a instrução é um salto condicional.
- *jump* define se a instrução é um salto incondicional.
- *ula_otherinp* define se a primeira entrada da ULA (Mais tarde descrita como *a*) recebe o valor lido no primeiro registrador ou 0. Em quase todas as instruções, a ULA recebe o valor do primeiro registrador, porém a instrução LI nada mais é que uma soma entre 0 e uma constante, e nesse caso a primeira entrada será 0.
- *ula_inp* define se a segunda entrada da ULA (Mais tarde descrita como *b*) recebe o valor lido no segundo registrador ou uma constante.
- *regwrite* define se a instrução escreve no registrador de destino. Instruções como o SW e o BEQ não escrevem em registradores, por exemplo.
- *negate* define se a instrução é um BEQ ou um BNE, caso ela seja um branch (Caso contrário, o sinal retorna '0' por padrão, porém seu valor não importa).
- *enable* define se o PC deverá continuar avançando ou se o programa já chegou ao fim de sua execução.

O conjunto desses sinais, enviados para os outros componentes, é o que define o tipo de instrução a ser executada. Para tanto, a unidade de controle recebe o tipo de instrução a ser executada, ou seja, os 4 bits mais significativos da instrução, e retorna como saída todos os seus sinais de controle. A tabela abaixo descreve os valores atribuídos aos sinais para cada instrução:

| | Memwrite | Branch | Jump | Ula_otherinp | Ula_inp | Regwrite | Negate | Enable | Ula_op | Memtoreg |
|---|---|---|---|---|---|---|---|---|---|---|
| ADD | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 000 | 0 |
| SUB | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 001 | 0 |
| LW | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 000 | 1 |
| SW | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 000 | x |
| J | 0 | 0 | 1 | X | X | 0 | 0 | 1 | XXX | X |
| BEQ | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 001 | 0 |
| OR | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 010 | 0 |
| AND | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 011 | 0 |
| XOR | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 100 | 0 |
| NOT | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 100 | 0 |
| LI | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 000 | 0 |
| BNE | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 001 | 0 |
| END | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 000 | 0 |

**Tabela 2: Sinais de controle correspondentes a cada instrução**

### 1.4.5. BancoReg

O banco de registradores (BancoReg) contém todos os registradores do processador. É nele que serão armazenados os resultados das operações de soma, leitura imediata, subtração e outras, e é nele que são lidos os operadores de grande parte das instruções do processador.

O banco de registradores consiste em um vetor de 16 elementos, cada um dos quais pode armazenar 4 bits. Recebe como entrada um sinal de controle, três endereços de registradores e um input.

Os três registradores endereçados são o registrador destino (*regout*) e os dois registradores a serem lidos (*reg1* e *reg2*). O sinal de controle define se o input deverá ou não ser escrito no registrador destino, e é necessário para evitar que valores indesejados sejam armazenados em algum registrador durante a execução de instruções como *SW*, *BEQ* e *J*. Por fim, o input (*inp*) define o valor a ser escrito no registrador destino. O banco de registradores possui três saídas principais, que são os valores lidos em *reg1*, *reg2* e *regout*, e 16 saídas adicionais que permitem a visualização do banco de registradores a partir da Waveform, cada saída representando o valor armazenado em um registrador.

### 1.4.6. ULA

A Unidade Lógica e Aritmética (ULA) realiza uma operação entre dois valores de 4 bits (*a* e *b*). Esta operação pode ser uma soma, uma subtração, ou uma operação lógica (AND, OR, XOR ou NOT). Caso a operação seja um NOT, o segundo input é ignorado. A operação é definida pela unidade de controle.

A ULA trata todas as suas entradas como *Signed Ints*, o que significa que suas operações são limitadas a inteiros de 4 bits incluindo o bit de sinal (Uma vez que suas entradas sempre possuem apenas 4 bits). Qualquer saída com valor superior a esse estará incorreta. Além do resultado da operação, a ULA possui uma segunda saída, o *zero*, que define os dois valores são iguais.

Seu uso depende completamente da instrução, podendo servir para calcular um endereço ou mesmo executar uma operação direta entre valores lidos nos registradores. Além disso, sua saída *zero* desempenha papel fundamental na execução de instruções Branch.

### 1.4.7. MemoriaRAM

A memória RAM (MemoriaRAM) consiste em um vetor de 16 elementos (Não poderia ser maior, uma vez que a ULA retorna apenas valores de 4 bits, e $2^4 = 16$), cada um dos quais armazena 4 bits. Sua entrada consiste em um valor de input, dois endereços, um para leitura e um para escrita, e um sinal de controle que define se o input deverá ser armazenado no endereço de escrita ou não. Sua saída é o valor lido no endereço de leitura.

### 1.4.8. Multiplexadores

O processador IntelIgente conta com uma vasta gama de multiplexadores. Uma vez que muitas saídas não serão utilizadas devido a serem especificamente destinadas a uma instrução, são necessários multiplexadores que definam quais saídas deverão ser selecionadas para que a instrução seja executada corretamente. Esses multiplexadores são geridos pela Unidade de Controle, e estão listados a seguir:

- *memux* garante que o *input* do banco de registradores sempre receberá o valor certo, decidindo de acordo com a instrução se sua saída deverá ser o valor lido na memória RAM ou o resultado da operação executada na ULA.
- *zeromux* garante o funcionamento da instrução *LI*, decidindo se a primeira entrada da ULA receberá 0 ou o valor lido no primeiro registrador do banco.
- *ulainpmux* garante o funcionamento das instruções de operador constante, decidindo se a ULA receberá o valor lido no segundo registrador do banco ou uma constante dada pelo endereço do segundo registrador do banco.
- *branchmux* verifica se uma instrução é um branch ou não. Caso ela seja, o valor de PC poderá ser alterado se a condição para o branch ocorrer for verificada, pulando o número de instruções definido pelo salto.
- *jumpmux* verifica se uma instrução é um jump ou não. Caso ela seja, o valor de PC será alterado, pulando o número de instruções definido pelo salto. Tanto este como o *branchmux* recebem a instrução atual somada a 1 e a instrução atual com salto tomado, e sua função é decidir qual dessas será a próxima instrução.
- *eqnoteqmux* verifica se a instrução é um BEQ ou um BNE. Caso não seja nenhum dos dois, é inferida uma instrução BEQ por padrão, porém seu salto não é utilizado em nenhum momento, e portanto tal inferência não tem efeito sobre o comportamento do processador. Sua entrada é a saída *zero* da ULA e, caso a instrução seja um BNE, a saída é invertida (NOT *zero*).

### 1.4.9. Extensores de sinal

O processador IntelIgente conta com dois extensores de sinal, *ExtensorBranch* e *ExtensorJump* que transformam os 4 (Ou 12) bits do salto de uma instrução Branch ou Jump em 16 bits, para que este salto possa ser aplicado ao PC.

### 1.4.10. BranchSomador

O processador IntelIgente conta com duas instâncias de um componente chamado *BranchSomador*, cuja função é aplicar um salto *Branch* e um salto *Jump* ao PC. Ambos os somadores utilizam a exata mesma arquitetura, porém são instâncias diferentes, com entradas e saídas também diferentes.

### 1.4.11. Descrição do Datapath

Por fim, o Datapath tem a função de unir todos esses componentes para fazer o processador funcionar como deve. Para tanto, é necessário ligar as saídas e as entradas de cada componente corretamente.

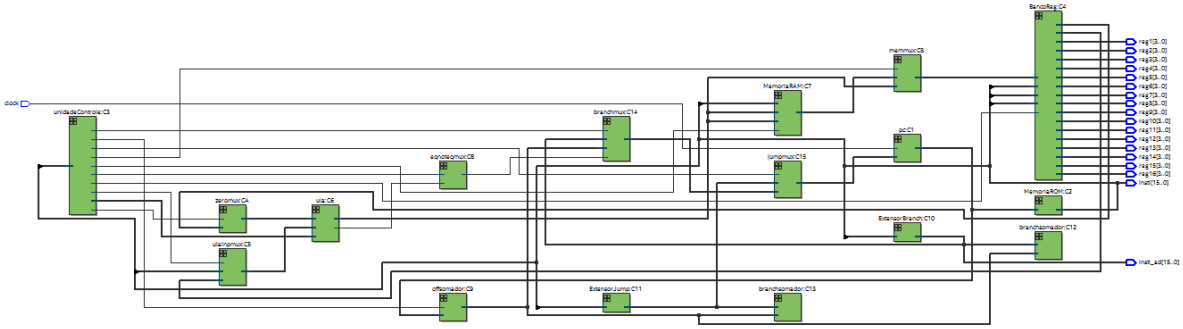

**Figura 4: RTL viewer**

A imagem acima descreve a RTL do componente (É possível visualizar uma versão amplificada no arquivo “rtl.png”, em anexo), e pode ajudar na visualização do Datapath. A maioria das conexões já foi explicada nas seções anteriores: PC envia o endereço da próxima instrução para a memória ROM, que retorna a próxima instrução. Os 4 bits mais significativos da primeira instrução serão utilizados para determinar que tipo de instrução deverá ser executado, enquanto os 12 outros bits assumem os mais diversos usos em cada componente:

- O banco de registradores os recebe como endereços de registradores em blocos de quatro bits. Os 4 bits mais significativos do grupo representam o registrador de destino, enquanto os outros 8 representam os registradores de entrada.
- O multiplexador *ulainpmux* recebe os bits de posição 8 até a posição 11 na instrução (Da direita para a esquerda) como possível constante.
- O extensor *branch* recebe os 4 bits menos significativos da instrução como possível salto.
- O extensor *jump* recebe os 12 bits menos significativos da instrução como possível salto.

As duas primeiras saídas do banco de registradores tornam-se operadores da ULA, enquanto a terceira saída é o valor a ser armazenado na memória RAM, caso *memwrite* esteja ativo. A saída principal da ULA torna-se o endereço a ser lido *e* escrito na memória RAM (Como nunca se lê e escreve ao mesmo tempo na memória RAM, não há problemas em usar o mesmo valor para ambos; Um deles sempre estará desabilitado), enquanto o *zero* torna-se um dos argumentos do *branchmux*, que recebe também o PC

atual e a instrução a ser executada caso o *branch* ocorra, e escolhe qual deles será o próximo PC. A saída do *branchmux* é aplicada no *jumpmux*, que recebe também a instrução a ser executada caso o *jump* ocorra, e assim o processador decide se a próxima instrução será PC + 1, um desvio *branch* ou um *jump* incondicional. Ao mesmo tempo, o *memmux* decide qual valor deverá ser armazenado no registrador de destino, e sua saída será o input do banco de registradores. Todas as instruções do processador funcionam dessa forma.

O datapath representa a entidade “Processador IntelIgente” como um todo, e portanto também possui pinos de entrada e saída. Sua única entrada é o clock, que fará o PC mudar de valor a cada ciclo. Não é necessária qualquer outra entrada, já que todas as instruções estão armazenadas na memória ROM. Suas saídas, por outro lado, são numerosas:

- *reg1,reg2,...,reg16* representam os valores armazenados em cada registrador. Esta é a saída principal do processador, afinal quando se está programando no MIPS é através dos registradores que todas as operações são executadas. Através dos valores armazenados em cada registrador é possível saber se o programa está funcionando corretamente ou não.
- *insti* representa o valor do PC após o ciclo de clock, enquanto *insti_ad* representa a instrução lida na memória de instruções a partir do PC. Ambos são pinos inicialmente utilizados para debugging que foram mantidos para proporcionar uma melhor visualização da execução do programa na waveform.

## 2. Simulações e testes

Todas as simulações e testes foram realizados com o auxílio da ferramenta “University Program Waveform” da IDE Quartus Prime. Os componentes foram testados com diversas entradas individualmente, e em seguida testados em conjunto através do datapath. O processador foi testado através de vários programas simples que consistiam de instruções de todos os tipos.

O primeiro teste foi um simples programa de 3 linhas. Sua Waveform pode ser visualizada no arquivo “Waveform1.png”, e seu código consistia em:

```
Binário                  -- Pseudocódigo
"1010000000010000" -- LI REG1,1
"1010000100100000" -- LI REG2,2
"0000001000010000" -- ADD REG3,REG2,REG1
```

Através dele, foi possível testar o funcionamento das instruções LI e ADD. Em seguida, foi aplicado um teste mais complicado:

```
Binário                  -- Pseudocódigo
"1010000000000000" -- LI REG1,0
"1010000100010000" -- LI REG2,1
"0011000100010000" – SW REG2,0(REG1)
"0010001100010000" -- LW REG2,0(REG1)
"1100000000000000" -- END
```

Nesse segundo teste, foi verificado o funcionamento das funções SW, LW e END. Todas funcionaram de acordo, e os resultados podem ser verificados no arquivo “Waveform2.png”.

O último teste realizado consistiu na verificação do funcionamento das operações lógicas AND, OR, NOT e XOR. Seus resultados podem ser verificados no arquivo “Waveform3.png”, e o seu código é mostrado abaixo:

```
Binário
"1010000000000000" -- LI REG1,0
"1010000100010000” -- LI REG2,1
"0110001000010000" -- OR REG3,REG2,REG1
```

"0111001100010000" -- AND REG4,REG2,REG1
"1000011000010000" -- XOR REG7,REG2,REG1
"1001010100010000" -- NOT REG6,REG2
"1100000000000000" – END

Entretanto, os testes para as instruções JUMP e BEQ falharam. Embora todas as saídas e testes individuais tenham gerado o output esperado, as instruções geram loop infinito quando testadas na waveform, por razões desconhecidas. Estas são as únicas instruções do processador que não funcionam.

## 3. Considerações finais

Este trabalho apresentou o projeto e implementação do processador de 16 bits denominado IntelIgente. Embora seja um projeto simples e falho, demandou uma imensa quantia de esforço e dedicação, mas também gerou aos seus participantes uma certa quantia de conhecimento. Foi uma experiência interessante.