Trabalho Prático 2
Organização de Computadores I

Vinicius Silva Gomes - 2021421869[1],
Mirna Mendonça e Silva - 2021421940[2],
Matheus Vaz Leal Lima - 2020109985[3]

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brazil

{vinicius.gomes, mirnamendonca, vazleal}@dcc.ufmg.br

Resumo. *Este artigo documenta o processo de modificação do datapath de uma implementação do processador RISC-V na linguagem de especificação de hardware Verilog. As modificações foram feitas para que ele suporte mais instruções, sendo elas: ANDI (and immediate), SRLI (shift right logical immediate), J (jump) e BGT (branch on greater than). Serão especificadas todas as mudanças necessárias ao longo do datapath para fazer cada uma delas funcionar e as saídas esperadas/obtidas.*

1. Introdução

O processador é o componente principal de qualquer computador, indo desde os computadores mais genéricos e primitivos que conhecemos até os mais modernos e pequenos, que encontramos nos mais variados dispositivos e equipamentos do dia-a-dia. Dentre os componentes do processador, podemos destacar o *datapath*, uma coleção de unidades funcionais, como Unidades Lógica e Aritmética (ALU), comparadores, multiplicadores, etc; que performam as operações de processamento dos dados até a saída do processador.

O *datapath*, portanto, é composto por uma série de componentes diferentes e pode ser implementado de inúmeras maneiras, incluindo técnicas avançadas para gerenciamento do fluxo e segmentação de dados/instruções por ele, como *pipeline*, etc.

Os componentes que compõem o *datapath* irão dizer quais operações esse processador será capaz de fazer e vão orientar o conjunto de instruções em cima desse processador. Todavia, um conjunto de instruções pode não incluir todas as instruções possíveis implementadas sobre aquela arquitetura, dado o hardware do processador em que ele está implementado. Sendo assim, uma tarefa que pode ser vital para um sistema de hardware se tornar mais otimizado é a inclusão de novas operações no conjunto de instruções.

Com isso, chegamos ao objetivo principal desse trabalho: dado uma implementação parcial de um processador em Verilog, linguagem conhecida de especificação de hardware, devemos alterar o *datapath* desse processador, modificando as operações implementadas pela ALU, os sinais de controle enviados pela Unidade de Controle (CU), etc; para que esse processador seja capaz de executar novas instruções a partir disso. As instruções a serem adicionadas são: ANDI (and immediate), SRLI (shift right logical immediate), J (jump) e BGT (branch on greater than).

Como citado anteriormente, o *datapath* do processador pode ser organizado de diferentes formas. No entanto, para o caso desse trabalho, será utilizado uma arquitetura

mista baseada em duas outras implementações conhecidas, o RISC-V e o MIPS. Essa arquitetura, além disso, implementa uma estratégia de *pipeline* em 5 estágios. Esses estágios do *pipeline* e a estrutura parcial do processador usado pode ser observada na figura 1.

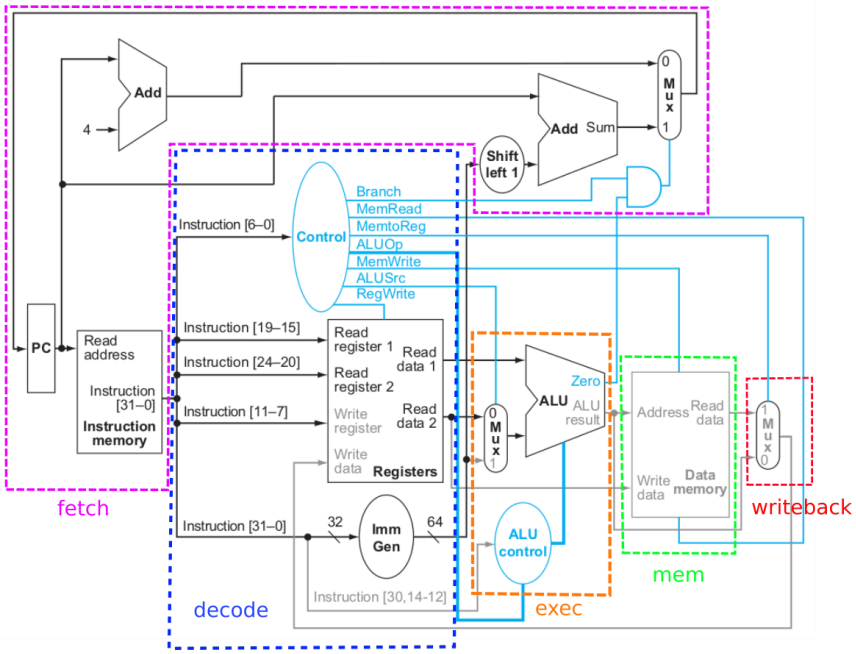


Figura 1. Datapath do RISC-V com 5 estágios de pipeline

2. Implementação das Instruções

Nessa seção, todas as instruções adicionadas serão melhor destrinchadas, dando ênfase para as modificações necessárias para que elas funcionem e quais eram os resultados esperados/obtidos.

2.1. ANDI (*and immediate*)

A instrução ANDI (*and immediate*) realiza a operação lógica *AND* entre um registrador de propósito geral e o imediato informado no código da instrução. O formato da instrução ANDI pode ser observado na tabela 1.

Para implementar essa instrução no *assembly*, foram necessárias poucas modificações. Como instruções que utilizam imediatos já haviam sido previamente implementadas, como o ADDI, que utiliza o mesmo *opcode* que a intrução ANDI, o fluxo de sinais de controle para instruções deste tipo já estava correto, bastando ser feito modificações necessárias na ALU para que seja capaz de executar essa função, e algumas modificações para que o *funct3* seja interpretado e diferenciado de outras operações com imediato. A figura 2 mostra como ficou o código da Unidade Lógica Aritmética após a instrução ANDI ser incluída.

Além disso, também foi necessário modificar a Unidade de Controle, uma vez que a instrução ADDI, que já tinha sido implementada, possui o mesmo *opcode*, desta forma foi necessário diferenciar as duas operações pelo *funct3*. Para isso foi criado um *switch case* que gerencia a diferença de *funct3* para essas instruções. A figura 3 mostra como ficou o código que realiza esse condicionamento.

```
always @(*) begin
  case(funct[3:0])
    4'd0:  _funct = 4'd2;   /* add */
    4'd5:  _funct = 4'd1;   /* or */
    4'd6:  _funct = 4'd13;  /* xor */
    4'd7:  _funct = 4'd0;   /* andi */
    4'd8:  _funct = 4'd6;   /* sub */
    4'd10: _funct = 4'd7;   /* slt */
    default: _funct = 4'd0;
  endcase
end
```

Figura 2. Bloco de código da ALU incluindo a instrução ANDI

```
7'b0010011: begin /* addi or andi or srli */
    case (f3)
     3'b000: begin /* addi */
        aluop[1:0] <= 2'b00;
        ImmGen     <= {{20{inst[31]}}, inst[31:20]};
      end
     3'b101: begin /* srli */
        aluop[1:0] <= 2'b11;
        ImmGen     <= {{20{inst[31]}}, inst[31:20]};
      end
     3'b111: begin /* andi */
        aluop[1:0] <= 2'b10;
        ImmGen     <= {{20{inst[31]}}, inst[31:20]};
      end
    endcase
  alusrc <= 1'b1;  /* use imm */
end
```

Figura 3. Bloco de código da Unidade de Controle incluindo a instrução ANDI

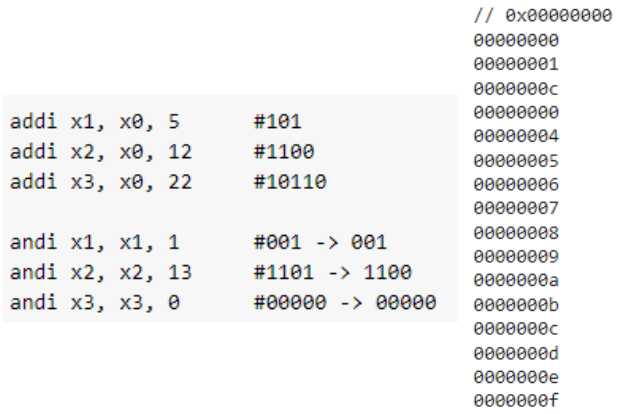
imm[11:0]	RS1	funct3	RD	opcode
Imediato	Registrador 1	111	Registrador destino	0010011

Tabela 1. Formato da instrução ANDI

Foram criados dois casos para testar a implementação de ANDI. No primeiro caso, os registradores $x1$, $x2$ e $x3$ são inicializados com os valores 5, 12 e 22, respectivamente, escolhidos de forma arbitrária. Realizamos a operação ANDI entre estes registradores e os imediatos 1, 13 e 0, respectivamente, mas escolhidos de forma a representar situações diferentes. O imediato 0, por exemplo, deve garantir que o resultado seja 0.

No segundo caso de teste, os registradores $x4$, $x5$ e $x6$ são inicializados com os mesmos valores iniciais do caso anterior. Dessa vez, é realizada a operação ANDI entre estes registradores e os imediatos 6, 15 e 31, respectivamente, no qual o 31, por exemplo, é escolhido por ter todos os bits positivos e, por isso, o resultado deve ser o próprio número do registrador usado.

As figuras 4 e 5 mostram as instruções que realizam cada um dos testes e qual foi a saída obtida nos registradores de propósito geral.

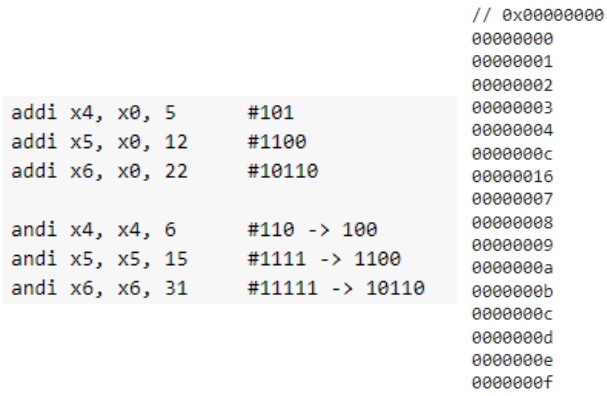


```
addi x1, x0, 5      #101
addi x2, x0, 12     #1100
addi x3, x0, 22     #10110

andi x1, x1, 1      #001 -> 001
andi x2, x2, 13     #1101 -> 1100
andi x3, x3, 0      #00000 -> 00000
```

```
// 0x00000000
00000000
00000001
0000000c
00000000
00000004
00000005
00000006
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

Figura 4. Sequência de instruções e resultado do teste 1 do ANDI



```
addi x4, x0, 5      #101
addi x5, x0, 12     #1100
addi x6, x0, 22     #10110

andi x4, x4, 6      #110 -> 100
andi x5, x5, 15     #1111 -> 1100
andi x6, x6, 31     #11111 -> 10110
```

```
// 0x00000000
00000000
00000001
00000002
00000003
00000004
0000000c
00000016
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

Figura 5. Sequência de instruções e resultado do teste 2 do ANDI

2.2. SRLI (*shift right logical immediate*)

A instrução SRLI (*shift right logical immediate*) realiza a operação lógica *shift* com o dado presente em um registrador de propósito geral, deslocando o dado do registrador o número de posições informadas pelo imediato da instrução. O formato da instrução SRLI pode ser observado na tabela 2.

funct5	funct2	RS2	RS1	funct3	RD	opcode
00000	00	shamt[4:0]	Registrador 1	111	Registrador destino	0010011

Tabela 2. Formato da instrução SRLI

Foi necessário implementar a instrução SRLI, pois a ALU existente não possuía esta operação. Logo, bastou adicionar uma linha que realizasse uma operação de *shift* para a direita e então adaptar a ALU para que ela reconhecesse o *funct3* desta instrução de valor 101. As figuras 6 e 7 mostram como ficou o código da ALU após essas alterações.

Também foi necessário modificar a Unidade de Controle, tendo em vista que o *opcode* das instruções ADDI, ANDI e SRLI são idênticos. Como já havíamos implementado uma lógica para verificar o *funct3*, somente precisamos adicionar um *case* especifico para SRLI, como pode ser visto na figura 3.

Foram criados dois casos para simular a implementação de SRLI. No primeiro

```
always @(*) begin
  case (ctl)
    4'd0:  out <= a & b;        /* and */
    4'd1:  out <= a | b;        /* or */
    4'd2:  out <= add_ab;       /* add */
    4'd4:  out <= a >> b;       /* srli */
    4'd6:  out <= sub_ab;       /* sub */
    4'd7:  out <= {{31{1'b0}}, slt};  /* slt */
    4'd12: out <= ~(a | b);       /* nor */
    4'd13: out <= a ^ b;        /* xor */
    default: out <= 0;
  endcase
end
```

Figura 6. Bloco de código da ALU incluindo a instrução SRLI

```
always @(*) begin
  case(aluop)
    2'd0: aluctl = 4'd2;  /* add */
    2'd1: aluctl = 4'd6;  /* sub */
    2'd2: aluctl = _funct;
    2'd3: aluctl = 4'd4;  /* srli */
    default: aluctl = 0;
  endcase
end
```

Figura 7. Bloco de código da ALU incluindo a instrução SRLI

caso, os registradores $x1$, $x2$ e $x3$ são inicializados com os valores 5, 12 e 22, respectivamente, escolhidos de forma arbitrária. Realizamos a operação SRLI entre estes registradores e o imediato 1, ou seja, realizamos um *shift* de uma casa em cada número.

No segundo caso de teste, os registradores $x4$, $x5$ e $x6$ são inicializados com os mesmos valores iniciais do caso anterior. Dessa vez, é realizada a operação SRLI entre estes registradores e os imediatos 2, 3 e 4, respectivamente. Com isso, realizamos *shifts* de tamanhos diferentes entre os números e verificamos seus resultados.

As figuras 8 e 9 mostram as instruções que realizam cada um dos testes e qual foi a saída obtida nos registradores de propósito geral.

```
addi x1, x0, 5      #101
addi x2, x0, 12     #1100
addi x3, x0, 22     #10110

slri x1, x1, 1      #2
slri x2, x2, 1      #6
slri x3, x3, 1      #11
```

```
// 0x00000000
00000000
00000002
00000006
0000000b
00000004
00000005
00000006
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

Figura 8. Sequência de instruções e resultado do teste 1 do SRLI

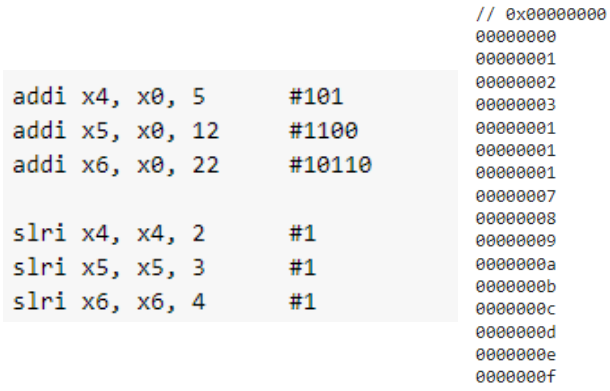


Figura 9. Sequência de instruções e resultado do teste 2 do SRLI

2.3. J (*jump*)

A instrução J (*jump*) realiza um desvio incondicional no conjunto de instruções que estão sendo executadas pelo processador. O *jump* encaminha o contador de programas (PC) para o trecho informado como *label* na chamada da instrução. Esse *alias* é traduzido como um imediato e indica para qual linha o PC vai ser transferido. O formato da instrução J pode ser observado na tabela 3.

imm[31:7]	opcode
Imediato	1101111

Tabela 3. Formato da instrução J

Para implementar essa instrução no *assembly*, foram necessárias modificações na Unidade de Controle e no Código de Decodificação. O valor do *opcode* previamente escrito na versão base do trabalho para a instrução *jump* estava incorreto, então, corrigimos o valor para que a instrução funcione corretamente. E se fez necessário alterar o Imediato que estava na versão base MIPS para RISC-V.

```
7'b1101111: begin /* j jump */
  ImmGen <= {{11{inst[31]}}, inst[31], inst[20], inst[30:21], inst[19:12], 1'b0};
  jump <= 1'b1;
end
```

Figura 10. Bloco de código da Unidade de Controle incluindo a instrução JUMP

Como dito, foi necessário modificação no Código de Decodificação, para que o valor de PC no momento do salto fosse PC + Imediato.

Foram criados dois casos de teste para simular a implementação de J. No primeiro caso, os registradores $x1$, $x2$, $x3$ são inicializados com os valores 5, 12 e 22, respectivamente, escolhidos de forma arbitrária. Então, é chamado o *jump*, que deve redirecionar o código para um *label* que se encontra na penúltima linha do teste. Existem três instruções entre o *jump* e o *label*, que adicionam um aos registradores previamente mencionados. Após o *label*, existe uma instrução que inicializa o registrador $x7$ com o valor 2. Dessa forma, se o caso tiver sucesso, os três registradores iniciais devem permanecer com os valores que foram inicializados e o $x7$ deve ter o valor 2.

```
assign imm      = inst_s2[15:0];
assign shamt    = inst_s2[10:6];
assign jaddr_s2 = pc4_s2 + ImmGen;
assign seimm  = {{16{inst_s2[15]}}, inst_s2[15:0]};
```

Figura 11. Bloco de código do Código de Decodificação para o JUMP

No segundo caso, os registradores $x4$, $x5$ e $x6$ são inicializados com os mesmos valores do caso anterior. Dessa vez, é chamado o *jump* e o *label* está na linha seguinte. Três instruções, então, são chamadas, adicionando um aos registradores previamente mencionados. Com isso, se o caso tiver sucesso, os três registradores iniciais devem ter os valores iniciais incrementados por um.

As figuras 12 e 13 mostram as instruções que realizam cada um dos testes e qual foi a saída obtida nos registradores de propósito geral.

```
addi x1, x0, 5      #101
addi x2, x0, 12     #1100
addi x3, x0, 22     #10110

j end

addi x1, x1, 1      #110
addi x2, x2, 1      #1101
addi x3, x3, 1      #10111

end:
addi x7, x0, 2      #10
```

```
// 0x00000000
00000000
00000005
0000000c
00000016
00000004
00000005
00000006
00000002
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

Figura 12. Sequência de instruções e resultado do teste 1 do J

```
addi x4, x0, 5      #101
addi x5, x0, 12     #1100
addi x6, x0, 22     #10110

j endtwo
endtwo:
addi x4, x4, 1      #110
addi x5, x5, 1      #1101
addi x6, x6, 1      #10111
```

```
// 0x00000000
00000000
00000001
00000002
00000003
00000006
0000000d
00000017
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

Figura 13. Sequência de instruções e resultado do teste 2 do J

2.4. BGT (*branch on greater than*)

A instrução BGT (*branch on greater than*) realiza um desvio condicional baseado na comparação de dois registradores de propósito geral diferentes. A comparação realizada

é se o primeiro registrador fonte é maior que o segundo registrador fonte. O formato da instrução BGT pode ser observado na tabela 4.

imm[12|10 : 5]	RS2	RS1	funct3	imm[4 : 1|11]	opcode
Imediato	Registrador 2	Registrador 1	100	Imediato	1100011

Tabela 4. Formato da instrução BGT

As implementações clássicas da instrução BGT em arquiteturas conhecidas a incluem como uma pseudo-instrução baseada na instrução BLT. Para utilizar a instrução *branch on greater than*, o compilador chama a instrução *branch on less than* com os registradores recebidos da BGT invertidos. Dessa forma, na prática, a operação de BGT que será realizada e o resultado desejado será obtido. A tabela 5 mostra como é o formato de instrução do BGT se visualizado como uma pseudo-instrução, construído a partir da instrução BLT.

imm[12|10 : 5]	RS1	RS2	funct3	imm[4 : 1|11]	opcode
Imediato	Registrador 1	Registrador 2	100	Imediato	1100011

Tabela 5. Formato da instrução BGT baseado na BLT (RS1 e RS2 invertidos)

Com isto, não se fez necessário nenhuma inclusão nos códigos da implementação fornecida, somente utilizar normalmente a instrução desejada com esse pequeno detalhe que foi descrito.

Para testar o BGT, foram criados três testes diferentes que cobriam todas as possíveis situações da comparação: $RS1 > RS2$, $RS1 = RS2$ e $RS1 < RS2$. No primeiro caso, os registradores $x1$, $x2$, $x3$ são inicializados com os valores 5, 12 e 22, respectivamente, escolhidos de forma arbitrária. Então, é chamado o *branch*, que deve redirecionar o código para um *label* que se encontra na penúltima linha do teste caso o registrador $x3$ seja maior que o $x1$, o que é verdade. Existem três instruções entre o *jump* e o *label*, que adicionam um aos registradores previamente mencionados. Após o *label*, existe uma instrução que inicializa o registrador $x10$ com o valor 2. Dessa forma, se o caso tiver sucesso, os três registradores iniciais devem permanecer com os valores que foram inicializados e o $x10$ deve ter o valor 2.

No segundo caso, os registradores $x4$, $x5$ e $x6$ são inicializados com os mesmos valores do caso anterior. Então, é chamado o *branch*, que deve redirecionar o código para um *label* que se encontra na penúltima linha do teste caso o registrador $x4$ seja maior que o $x6$, o que é falso. Existem três instruções entre o *jump* e o *label*, que adicionam um aos registradores previamente mencionados. Após o *label*, existe uma instrução que inicializa o registrador $x11$ com o valor 2. Dessa forma, se o caso tiver sucesso, os três registradores iniciais devem ter os valores incrementados por um e o $x11$ deve ter o valor 2.

No terceiro caso, os registradores $x7$, $x8$, $x9$ são inicializados com os valores 5, 22 e 22, respectivamente. Então, é chamado o *branch*, que deve redirecionar o código para um *label* que se encontra na penúltima linha do teste caso o registrador $x8$ seja maior que o $x9$, o que é falso, visto que são iguais. Existem três instruções entre o *jump* e o *label* que adicionam um aos registradores previamente mencionados. Após o *label*, existe uma instrução que inicializa o registrador $x12$ com o valor 2. Dessa forma, se o caso tiver

sucesso, os três registradores iniciais devem ter os valores incrementados por um e o $x12$ deve ter o valor 2.

As figuras 14, 15 e 16 mostram as instruções que realizam cada um dos testes e qual foi a saída obtida nos registradores de propósito geral.

```
addi x1, x0, 5      #101
addi x2, x0, 12     #1100
addi x3, x0, 22     #10110

bgt x3, x1, maior

addi x1, x1, 1      #110
addi x2, x2, 1      #1101
addi x3, x3, 1      #10111

maior:
addi x10, x0, 2      #10
```

```
// 0x00000000
00000000
00000005
0000000c
00000016
00000004
00000005
00000006
00000007
00000008
00000009
00000002
0000000b
0000000c
0000000d
0000000e
0000000f
```

Figura 14. Sequência de instruções e resultado do teste 1 do BGT

```
addi x4, x0, 5      #101
addi x5, x0, 12     #1100
addi x6, x0, 22     #10110

bgt x4, x6, maiortwo

addi x4, x4, 1      #110
addi x5, x5, 1      #1101
addi x6, x6, 1      #10111

maiortwo:
addi x11, x0, 2      #10
```

```
// 0x00000000
00000000
00000001
00000002
00000003
00000006
0000000d
00000017
00000007
00000008
00000009
0000000a
00000002
0000000c
0000000d
0000000e
0000000f
```

Figura 15. Sequência de instruções e resultado do teste 2 do BGT

```
addi x7, x0, 5      #101
addi x8, x0, 22     #10110
addi x9, x0, 22     #10110

bgt x8, x9, maiorthree

addi x7, x7, 1      #110
addi x8, x8, 1      #10111
addi x9, x9, 1      #10111

maiorthree:
addi x12, x0, 2      #10
```

```
// 0x00000000
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000006
00000017
00000017
0000000a
0000000b
00000002
0000000d
0000000e
0000000f
```

Figura 16. Sequência de instruções e resultado do teste 3 do BGT

3. Considerações Finais

De modo geral, esse trabalho foi bastante pertinente para que conhecêssemos melhor como funciona o *datapath* de um processador, e pudéssemos, com isso, ter um contato mais próximo e aplicado dos conceitos vistos em sala de aula.

Apesar de grande parte do *datapath* já ter sido previamente implementada, para adicionar novas instruções, muito tempo foi investido no estudo de como os circuitos e códigos funcionam. Isso fez com que todos do grupo entendessem bem como o *datapath* foi pensado e implementado e quais deveriam ser as modificações necessárias nele para que as novas instruções fossem incluídas.

Dessa forma, com o desenvolvimento do trabalho, muitos conceitos vistos em sala de aula sobre *datapath* e a estrutura do processador, no geral, puderam ser consolidados e tivemos um contato maior e mais próximo com a linguagem de especificação de hardware Verilog, que é muito utilizada, juntamente com a linguagem VHDL, para definir o fluxo de funcionamento e especificações dos mais variados dispositivos de hardware.