ID: .....

Desenvolvimento de um Processador em Linguagem de Descrição de Hardware Laboratório de Arquitetura de Computadores

São José dos Campos - Brasil

Julho de 2017

ID: .....

Desenvolvimento de um Processador em Linguagem de Descrição de Hardware
Laboratório de Arquitetura de Computadores

Relatório final do projeto de um processador, parte da disciplina de Laboratório de Sistemas computacionais: Arquitetura e Organização de Computadores da Universidade Federal de São Paulo.

Docente: Prof. Dr. Tiago de Oliveira

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Julho de 2017

Resumo

Computadores, sistemas eletrônicos desenvolvidos para aceitar dados de entrada e retornar valores processados de saída realizando cálculos e tomada de decisão, está presente na vida de todos como um dispositivo auxiliar, de trabalho, ou até mesmo de entretenimento. O principal componente de um computação é sua unidade central de processamento. O objetivo desse relatório é desenvolver um microprocessador completo em linguagem de descrição de *hardware*, testá-lo gerando formas de onda e utilizando placas programáveis FPGA e mostrar a maneira como foram resolvidos problemas de implementação de uma arquitetura baseada em monociclo e sem pipeline.

Palavras-chaves: Verilog. Processador. Sistemas Computacionais. Arquitetura de Computadores. MIPS.

Lista de ilustrações

Lista de tabelas

Lista de Algoritmos

Sumário

1 Introdução

A grande quantidade de informação atualmente nos traz a necessidade de utilizar ferramentas para nos ajudar a lidar com os dados, sistemas computacionais fazem isso muito bem, realizando cálculos e operações lógicas, o que os torna indispensáveis em diferentes aplicações do cotidiano, como (2) cita, em maio de 2017 o número de *smartphones* ativos no Brasil chegará a 168 milhões, o que representa grande parte da população, sem levar em conta que é comum uma pessoa carregar mais de um "computador de bolso", destacando a utilidade do mesmo. O desenvolvimento de um sistema computacional é baseado no microprocessador, a unidade onde são feitos cálculos aritméticos e tomadas de decisão. O microprocessador é programável, ou seja, há uma integração *hardware/software* através de um conjunto de instruções definido em seu mais baixo nível (3). As instruções podem caracterizar o tipo de processador, juntamente com a maneira como ele utiliza memória e o endereçamento de operandos (4), por esse motivo, um dos passos mais importantes é o projeto do conjunto de instruções aliado à maneira como disponibilizar tais funções a partir do *hardware*. A unidade que controla todos os passos tomados em uma instrução é chamada de Unidade de Controle, ela indica o que deve ser feito a partir de uma instrução específica, diferenciada pelos *bits* chamanos *opcode*.

2 Objetivos

2.1 Geral

O objetivo do projeto, junto aos relatórios anteriores é desenvolver um microprocessador, utilizar uma linguagem de descrição de hardware para descrevê-lo e gerar simulações que demonstram o funcionamento do microprocessador.

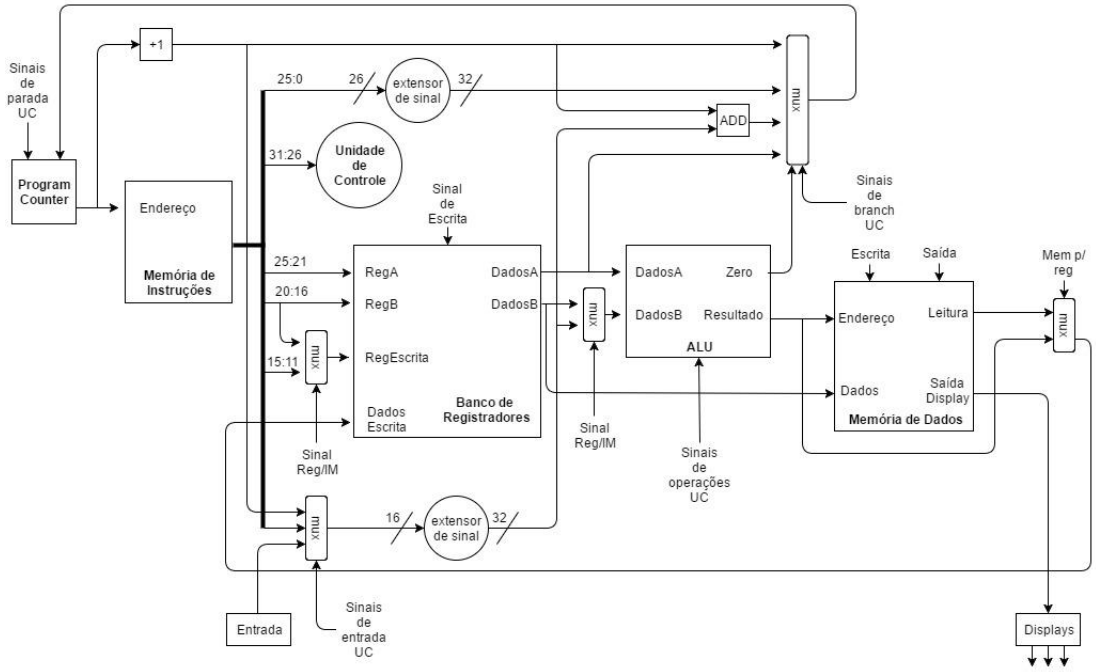
2.2 Específico

Nos relatórios anteriores foram desenvolvidos o conjunto de instruções, o esquemático do mesmo e a implementação de todos os módulos da arquitetura interligados (em linguagem de descrição de *hardware* ***Verilog***) com excessão da Unidade de Controle. O objetivo desse relatório então é mostrar a implementação da Unidade de Controle em ***Verilog***, juntando-a com os módulos já desenvolvidos, os resultados serão mostrados a partir de formas de onda e testes feitos em placas FPGA (5). Por ser o relatório final da disciplina, torna-se interessante elaborar um conteúdo menos específico, abordando objetivos de relatórios anteriores.

3 Fundamentação Teórica

Para desenvolver um processador a partir do esquemático mostrado na Figura 1, é necessário entender como deve funcionar cada módulo, que age de acordo de acordo com suas entradas, por exemplo, a Unidade Lógica e Aritmética deve fornecer os resultados para operações aritméticas e lógicas ou comparar dois números (3), a resposta pode servir como base para definir se um *branch* será tomado ou não. A possibilidade de executar todas instruções do conjunto no esquemático apresentado já foi garantida no relatório anterior (6), agora deve-se ajustar os módulos da arquitetura base MIPS (7) para que possibilitem a implementação delas, inserindo entradas e saídas necessárias.

Figura 1 – Esquemático



Fonte: Relatório 1 (6)

3.1 MIPS - Caminho de Dados

3.1.1 Unidade Lógica e Aritmética

A ULA (Unidade Lógica e Aritmética ou *ALU*, do inglês), recebe dois operandos de entrada e sinais de função, os operandos são utilizados para realizar a operação que os sinais de função indicam. As saídas são o resultado da operação e uma *flag* chamada zero, que indica se o resultado foi zero ou não (Utilizado para a realização de *branches*) (3).

3.1.2 Banco de Registradores

O Banco de Registradores nada mais é do que componentes de memória organizados, com endereços exclusivos. As entradas do banco de registradores são: 3 endereços de registradores, sendo dois deles exclusivamente para leitura, o terceiro é o registrador de destino. O sinal de escrita, que controla se dados serão escritos no registrador de destino. Os dados de escrita, que só são utilizados se o sinal de escrita for 1, se sim, serão guardados no registrador de destino. As saídas desse módulo são apenas duas, os dados dos dois registradores de leitura endereçados pelas entradas (3).

3.1.3 Memória de Dados

A Memória de Dados, também chamada de memória principal do computador é um meio externo de guardar dados (normalmente utiliza uma série de capacitores, diferente do Banco de Registradores, que utiliza portas lógicas para armazenas dados) quando a unidade de processamento não está utilizando-o, assim outros dados podem ser guardados no banco de registradores, que tem mais limitações de espaço. em sua entrada se encontra uma *flag* de escrita, um endereço e uma entrada para dados. Quando a *flag* de escrita é 1, o endereço sinaliza a parte da memória onde serão gravados os dados de entrada, senão o endereço informa o endereço dos dados de leitura na memória. Há também um componente de saída, que são os dados de leitura (3).

3.1.4 Memória de Instruções

A Memória de Instruções segue o mesmo conceito da Memória de Dados (na maioria dos casos é parte dela), porém guarda apenas instruções, que são carregadas ao início da execução de um programa, recebem um endereço de entrada e disponibiliza uma instrução na saída (3).

3.1.5 Contador de Programa

O contador de programa indica à memória de instruções qual é a instrução atual a ser lida, ou se a execução deve ser descontinuada. Como entrada tem um endereço e como saída também tem um endereço (3).

3.1.6 extensores de Sinal

Os extensores de Sinal são normalmente utilizados para adaptar operandos para que sejam a entrada de algum módulo que só admite um maior número de bits, por exemplo, imediatos que vêm diretamente na instrução normalmente utilizam 16 bits, mas a ULA só admite operando com 32 bits, então eles recebem um dado e disponibilizam o

mesmo dado, porém estendido com zeros (no caso de números positivos) ou uns (no caso de números negativos) (3).

3.1.7 Multiplexadores

Multiplexadores, comumente chamados apenas de *"muxes"* (ou *"mux"* no singular), são basicamente seletores, com duas ou mais entradas (fontes de dados), disponibilizando na saída os dados da entrada selecionada a partir de um sinal ou conjunto de sinais (8).

3.2 Linguagem de Descrição de *Hardware*

O projeto digital moderno é feito usando linguagens de descrição de *hardware* e ferramentas de síntese auxiliadas por computador que podem criar projetos de *hardware* detalhados das descrições usando bibliotecas e síntese de lógica (3). Uma linguagem de descrição de *hardware* bastante utilizada é a *Verilog*. Essas linguagens diferem de linguagens de programação *software* por incluir meios de descrever tempo de propagação e força do sinal, mas Verilog se assemelha muito à liguagem C, apesar de trazer um conceito diferente (9). Informações sobre como utilizar *Verilog* encontradas em (10) e (5).

A vantagens de utilizar linguagens como Verilog é basicamente tempo, a quantidade de tempo necessária para projetar um sistema através de esquemáticos é muito maior do que escrever a algumas linhas de código, já que o "trabalho duro"já foi implementado através de bibliotecas, além da maior legibilidade do projeto.

3.3 Aritmética Computacional

Para que o programa rode corretamente ou sinalize erros, é necessário entender um pouco de aritmética computacional e detectar erros de *overflow* ou aritméticos (como divisão por zero). Um *overflow* ocorre quando o resultado de uma operação necessita de mais bits do que estão disponíveis para o mesmo (3), gerando uma resposta errada, existem maneiras de sinalizar ou contornar esses problemas.

3.3.1 Adição e Subtração

A Tabela 1 indica quando um resultado indica *overflow* levando em consideração os dois operandos e o tipo de operação (no caso, adição ou subtração).

Tabela 1 – Condições de Overflow para Adição e Subtração.

Operação	Operando A	Operando B	Resultado Indicando Overflow
A+B	≥ 0	≥ 0	< 0
A+B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

Fonte: (3)

3.3.2 Multiplicação e Divisão

No caso da multiplicação, é possível que o resultado necessite do dobro do número de bits dos operandos, ou seja, se os operando têm 16 bits cada, para que não haja *overflow* são necessários 32 bits para o resultado no pior caso, uma maneira de contornar o *overflow* da multiplicação é restringir os operando a ter apenas metade dos bits, garantindo assim que não ocorram problemas. Na divisão, verifica-se o segundo operando para sinalizar ou não divisão por zero (3).

3.4 Entrada e Saída de Dados

3.4.1 Entrada de Dados

Quando pensamos na entrada de dados, precisamos levar em consideração que há uma interface humano-computador e que o usuário não segue o mesmo ritmo do computador, ou seja, o computador deve esperar o usuário indicar que terminou de ajustar a entrada de dados e que o processo pode continuar, isso pode ser feito utilizando um botão que determina a continuação do programa.

3.4.2 Saída de Dados

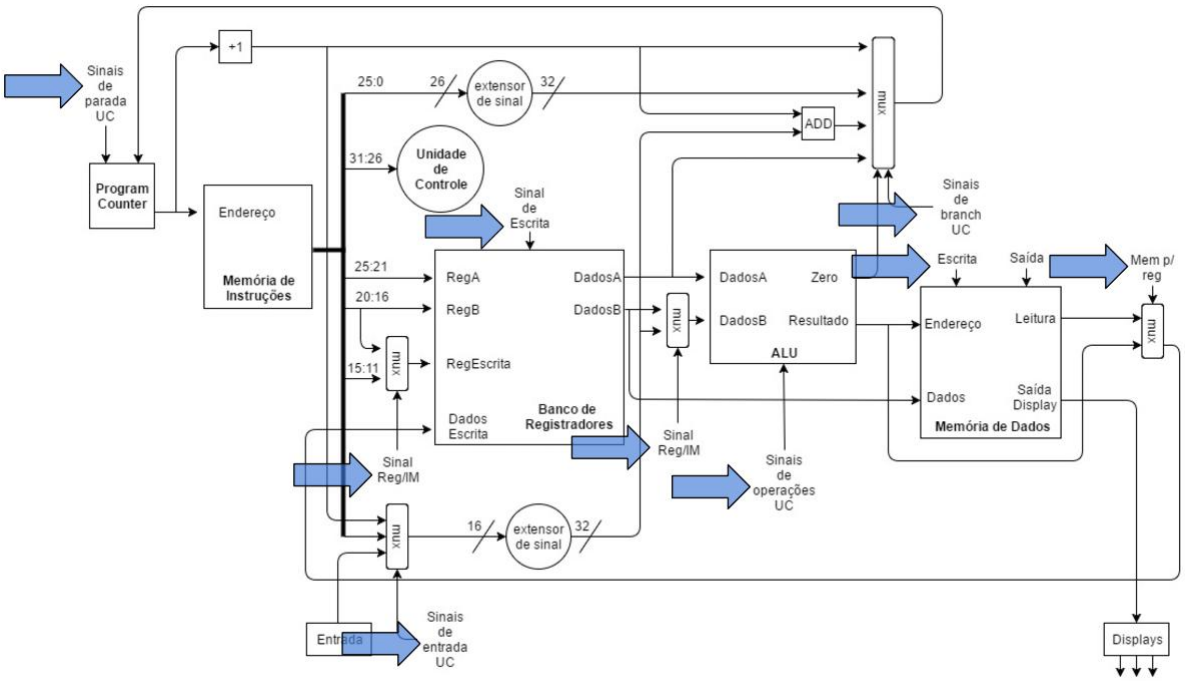
A saída de dados, pensando no ritmo computacional, deve permanecer nos *displays*, para que o usuário consiga observar os resultados. É recomendável "traduzir"os números de binário para decimal ao mostrar ao usuário, facilitando na hora de ler, além disso, é necessário tratar números negativos de alguma maneira, linguagens de descrição de *hardware* normalmente trabalham com complemento de 2 (11), então seria necessário transformar esse número para positivo e indicar que o mesmo é negativo para mostrar nos *displays*.

3.5 A Unidade de Controle

A unidade de controle é a estrutura do processador que controla e direciona a operação do mesmo, ela diz aos componentes da unidade de processamento como reagir às

instruções de programa recebidas, caracterizando o fluxo busca, decodificação e execução. No contexto MIPS, a Unidade de Controle é um circuito combinacional que recebe em torno de 6 bits que representam uma instrução específica do conjunto de instruções, gerando saídas que serão direcionadas à estruturas da unidade de processamento que precisam ser controladas, indicadas na Figura 2 (extensores de sinal não necessariamente são controlados pela UC)(12).

Figura 2 – Sinais da Unidade de Controle



Fonte: Relatório 1 (6)

4 Desenvolvimento

A partir do esquemático da arquitetura MIPS, o esquemático mostrado na Figura 1 e com base nas instruções e funções da ALU desenvolvidas no relatório (6) (Tabela 2), o objetivo é juntar os dois conceitos e criar soluções de implementação em *Verilog*. É interssante apontar que o processador desenvolvido e suas instrução contam com modos de endereçamento de operandos dos tipos *Imediato, por registrador e por registrador-base (para endereçar memória essencialmente)* e os formatos de instruções estão descritos na Tabela 3.

Tabela 2 – Tabela de Instruções.

	OPCODE	INSTRUÇÃO	FUNÇÃO
0	000000	NOP	-
1	000001	ADD	RE ← RA + RB
2	000010	ADDI	RE ← RA + IM
3	000011	ADD1	RE ← RA + 1
4	000100	SUB	RE ← RA - RB
5	000101	SUBI	RE ← RA - IM
6	000110	SUB1	RE ← RA - 1
7	000111	AND	RE ← RA & RB
8	001000	OR	RE ← RA | RB
9	001001	XOR	RE ← RA ^RB
10	001010	NOT	RE ← ∼RA
11	001011	MOD	RE ← RA % RB
12	001100	SHIFTL	RE ← RA «SHAMT
13	001101	SHIFTR	RE ← RA »SHAMT
14	001110	SLT	RE ← 1 se RA < RB | RE ← 0 se RA≥ RB
15	001111	SHT	RE ← 1 se RA > RB | RE ← 0 se RA ≤ RB
16	010000	SEQ	RE ← 1 se RA = RB | RE ← 0 se RA≠RB
17	010001	BEQ	PC ← PC+1+IM se RA = RB PC ← PC+1 se RA ≠ RB
18	010010	BNE	PC ← PC+1+IM se RA ≠ RB PC ← PC+1 se RA = RB
19	010011	J	PC ← IM+1
20	010100	JR	PC ← RA+1
21	010101	HALT	FIM DE EXECUÇÃO
22	010110	IN	RE ← RA + IN
23	010111	OUT	OUT ← RB
24	011000	LOAD	RE ← M[RA+IM]
25	011001	LOADI	RE ← RA + IM
26	011010	LOADA	RE ← PC + 1
27	011011	MULT	RE ← RA * RB
28	011100	DIV	RE ← RA / RB
29	011101	STORE	M[RA + IM] ← RB
30	011110	RESET	PC ← 0

Fonte: (6)

Tabela 3 – Formatos de Instruções

Número de bits	6	5	5	5	5	6
R	*opcode*	ra	rb	re	deslocamento	-
I	*opcode*	ra	rb	imediato (16 bits)		
J	*opcode*	imediato (26 bits)				
JR	*opcode*	ra	-			
JR (variação)	*opcode*	-	rb	-		
H	*opcode*	-				

Fonte: (6)

4.1 A implementação dos Componentes

Cada componente foi desenvolvido e testado separadamente, para depois ser testado dentro de todo o conjunto, formando o processador.

4.1.1 Unidade Lógica e Aritmética

A ULA desenvolvida nesse projeto é um pouco diferente da ULA utilizada nas arquiteturas MIPS simples (normalmente estudadas em cursos de arquitetura de computadores), a diferença é que foram adicionadas funções de comparação ("menor que", "maior que"e "igual a"), funções que calculam resto de divisão, multiplicação, divisão inteira e funções lógicas *and*, *or*, *xor*. Todas as funções da ULA foram listadas em Tabela 4.

Tabela 4 – Tabela de Funções da ULA.

	ALU OPCODE	FUNÇÃO
0	0000	OUT ← A + B
1	0001	OUT← A - B
2	0010	OUT ← A + 1
3	0011	OUT ← A - 1
4	0100	OUT ← A & B
5	0101	OUT ← A | B
6	0110	OUT ← A ^B
7	0111	OUT ← ∼A
8	1000	OUT ← A «SHAMT
9	1001	OUT ← A »SHAMT
10	1010	se A <B OUT ← 1 | senão OUT ← 0
11	1011	se A = B OUT ← 1 | senão OUT ← 0
12	1100	se A >B OUT ← 1 | senão OUT ← 0
13	1101	OUT ← A * B
14	1110	OUT ← A / B
15	1111	OUT ← A % B

Fonte: (6)

A implementação de uma ULA, mostrada no Algoritmo 4.1 recebe como entrada a operação(*operation*) a ser realizada, dois operandos(*dataA, dataB*) e um operando exclusivo

para um possível deslocamento de bits(*shamt*). Suas saídas são o resultado(*saida*), um sinal de overflow(*of*) e um sinal de resposta zero(*zero*).

Algoritmo 4.1 – ULA

```
1  module ALU(operation, dataA, dataB, saida, zero, shamt, of);
2    input [3:0] operation;
3    input [4:0] shamt;
4    input [31:0] dataA, dataB;
5    output reg [31:0] saida;
6    output zero;
7    output reg of;
8
9    always@(*)
10   begin
11     of = 0;
12   case(operation[3:0])
13     4'b0000:
14     begin
15     saida = dataA + dataB;
16     if(~dataA[31] && ~dataB[31] && saida[31])
17       of = 1;
18     else if(dataA[31] && dataB[31] && ~saida[31])
19       of = 1;
20     end
21     4'b0001:
22     begin
23       saida = dataA - dataB;
24       if(~dataA[31] && dataB[31] && saida[31])
25       of = 1;
26       else if(dataA[31] && ~dataB[31] && ~saida[31])
27       of = 1;
28     end
29     4'b0010: saida = dataA + 1;
30     4'b0011: saida = dataA - 1;
31     4'b0100: saida = dataA & dataB;
32     4'b0101: saida = dataA | dataB;
33     4'b0110: saida = dataA ^ dataB;
34     4'b0111: saida = ~dataA;
35     4'b1000: saida = dataA << shamt;
36     4'b1001: saida = dataA >> shamt;
37     4'b1010: saida = dataA < dataB ? 1 : 0;
38     4'b1011: saida = dataA == dataB ? 1 : 0;
39     4'b1100: saida = dataA > dataB ? 1 : 0;
40     4'b1101: saida = dataA[15:0] * dataB[15:0];
41     4'b1110:
42     begin
43       saida = dataA / dataB;
```

```
44      if(dataB == 32'd0)
45        of = 1;
46      end
47      4'b1111:
48      begin
49      saida = dataA % dataB;
50      if(dataB == 32'd0)
51        of = 1;
52      end
53      default: saida = 32'b0;
54    endcase
55    end
56    assign zero = (saida == 0);
57  endmodule
```

É interessante ressaltar que essa ULA (Algoritmo 4.1) só multiplica números de 16 bits, para evitar *overflow*, além disso ela indica divisão por zero e *overflow* (adição e subtração). As condições de *overflow* em adição e subtração foram tiradas da Tabela 1.

4.1.2 Banco de Registradores

A implementação do Banco de Registradores (Algoritmo 4.2) é exatamente igual ao MIPS, não foi necessário realizar modificação. As entradas são: endereços de dois registradores a serem lidos(*readregA, readregB*), endereço do registrador de escrita(*writereg*), a *flag* de escrita(*writeflag*), os dados de escrita(*writedata*) e o *clock*(*clk*). As saídas são: dados dos registradores de leitura(*dataA, dataB*).

Algoritmo 4.2 – Banco de Registradores

```
1     module RegBank(readregA, readregB, writereg, writedata,
2             writeflag, dataA, dataB, clk);
3
4   input [4:0] readregA, readregB, writereg;
5   input writeflag, clk;
6   input [31:0] writedata;
7   output [31:0] dataA, dataB;
8   reg [31:0] REGS [31:0];
9
10  always@(posedge clk)
11  begin
12    if(writeflag)
13      REGS[writereg] = writedata;
14    REGS[0] = 32'b0;
15  end
16
17  assign  dataA = REGS[readregA];
18  assign  dataB = REGS[readregB];
```

```
19
20 endmodule
```

Nesse código(Algoritmo 4.2), o registrador número zero não pode ser alterado, isso é útil para que ele sirva como base para comparações e saltos lógicos.

4.1.3 Memória de Instruções

A memória de instruções (Algoritmo 4.3) foi implementada como um vetor com 32 bits em cada posição (semelhante ao Banco de Registradores 4.1.2). As entradas são: Endereço da instrução(*address*) e o *clock*(*clk*). A saída é a instrução de 32 bits(*out*).

Algoritmo 4.3 – Memória de Instruções

```
1  module InstMem(address, clk, out);
2    input[15:0] address;
3    input clk;
4    output reg[31:0] out;
5    integer firstclock = 0;
6
7    reg[31:0] mem[11:0];
8
9    always@(posedge clk)
10   begin
11   if(firstclock == 0)
12   begin
13     // instrucoes
14
15         //Fatorial
16         //nop
17         mem[0] = 32'b00000000000000000000000000000000;
18         //in R1
19         mem[1] = 32'b01011000000000010000000000000000;
20         //add R1 R0 R2
21         mem[2] = 32'b00000100001000000001000000000000;
22         //addi R0 R3 1
23         mem[3] = 32'b00001000000000011000000000000001;
24         //mult R2 R3 R3
25         mem[4] = 32'b01101100010000110001100000000000;
26         //subi R2 R2 1
27         mem[5] = 32'b00010100010000100000000000000001;
28         //sht R2 R0 R4
29         mem[6] = 32'b00111100010000000010000000000000;
30         //beq R0 R4 1
31         mem[7] = 32'b01000100000001000000000000000001;
32         //j 4
33         mem[8] = 32'b01001100000000000000000000000100;
34         //store R0 R3 1
```

```
35          mem[9] = 32'b01110100000000110000000000000001;
36          //load R0 R5 1
37          mem[10] = 32'b01100000000001010000000000000001;
38          //out R5
39          mem[11] = 32'b01011100000001010000000000000000;
40          //reset
41          mem[12] = 32'b01111000000000000000000000000000;
42
43        firstclock <= 1;
44      end
45      end
46
47      always@(address)
48      begin
49        out = mem[address];
50      end
51
52  endmodule
```

Foi utilizado como exemplo uma sequência de instruções que representa um algoritmo que calcula fatoriais e será testado no Capítulo 5.

4.1.4 Memória de Dados

A diferença entre a memória de dados (Algoritmo 4.4) e o Banco de Registradores - 4.1.2 (nessa implementação) é que há apenas um endereço de entrada (que corresponde a posições de memória), porém, foi colocado um registrador a mais dentro da memória para manter registrada a saída para o módulo de saída (subseção 4.1.10). As entradas são: Endereço da posição de memória(*address*), dados a serem escritos(*datain*), *flag* de escrita na memória(*writeflag*), *flag* de escrita no registrador do *display*(*displayflag*). As saídas são: Dados de leitura(*dataout*) e Dados do *display*(*display*).

Algoritmo 4.4 – Memória de Dados

```
1  module DataMem(clk, writeflag, address, datain,
2          dataout, displayflag, display);
3    input writeflag, clk, displayflag;
4    input[31:0] address;
5    input[31:0] datain;
6    output [31:0] dataout, display;
7
8    reg[31:0] mem[9:0];
9    reg[31:0] displayreg;
10
11   always@(negedge clk)
12   begin
13     if(writeflag)
```

```
14        mem[address] = datain;
15      if(displayflag)
16        displayreg = datain;
17    end
18
19    assign dataout = mem[address];
20    assign display = displayreg;
21
22  endmodule
```

Nota-se que a memória de dados atualiza seus valores só na descida de *clock*, isso evita com que haja algum erro por causa de atraso da leitura de registradores recém escritos.

4.1.5 Contator de Programa

O contador de programa (Algoritmo 4.5) foi implementado de maneira símples, ele recebe um endereço e coloca na saída quando o *clock* muda de 0 para 1, verificando se há algum sinal de *reset*(PC $\leftarrow$ 0) ou de *halt*(*pause*). Entradas: Próximo endereço(*inaddress*), *clock*(*clk*), *reset*(*reset*), *halt*(*halt*). Saída: endereço de saída(*outaddress*.

Algoritmo 4.5 – Contador de Programa

```
1  module PC(inaddress, outaddress, halt, clk, reset);
2    input clk, reset;
3    input[15:0] inaddress;
4    input halt;
5    reg [15:0] novo;
6    output reg [15:0] outaddress;
7
8    always@(*)
9    begin
10   novo = inaddress;
11   end
12
13   always@(posedge clk)
14   begin
15     if(halt)
16       begin //faz nada
17       end
18     else if(reset)
19       outaddress = 0;
20     else
21       outaddress = novo;
22   end
23 endmodule
```

4.1.6 extensores de Sinal

No projeto há dois extensores de Sinal, um que estende a entrada de 16 para 32 bits(Algoritmo 4.6) e outro que estende a entrada de 26 para 32 bits(Algoritmo 4.7). A implementação é de certa forma simles, porém deve-se levar em consideração a expansão de números negativos (complemento de 2).

Algoritmo 4.6 – extensor de Sinal 16-32

```
module SE16(datain, dataout);

  input[15:0] datain;
  output reg[31:0] dataout;

  always@(*)
  begin
    if(datain[15])
    dataout = {{16{1'b1}},datain};
    else
    dataout = {{16{1'b0}},datain};
  end

endmodule
```

Algoritmo 4.7 – extensor de Sinal 26-32

```
module SE26(datain, dataout);

  input[25:0] datain;
  output reg[31:0] dataout;

  always@(*)
  begin
    if(datain[25])
    dataout = {{6{1'b1}},datain};
    else
    dataout = {{6{1'b0}},datain};
  end

endmodule
```

4.1.7 Multiplexadores

Para este projeto, foram desenvolvidos 4 tipos de multiplexadores, apesar de que todos seguem o mesmo conceito, servir como seletor. O *mux* de 5 bits tem duas entradas de 5 bits(*dataA, dataB*), uma entrada *flag* para selecionar entre as duas entradas(*flag*) e uma saída de 5 bits(*out*), é utilizado para selecionar o endereço do registrador de escrita

no Banco de Registradores (subseção 4.1.2). O *mux* de 16 bits recebe 3 entradas de 16 bits cada(*PC*, *switches* e *immediate*) e uma entrada *flag*(*flag*) de 2 bits, sua saída tem 16 bits, serve para selecionar entre *switches*, imediato ou endereço da instrução. O *mux* de 32 bits recebe 2 entradas de 32 bits cada(*dataA* e *dataB*) e uma *flag*(*flag*), serve para selecionar o operando de entrada de dados no banco de registradores ou o operando número 2 da ULA. Já o *mux* do PC, não é um seletor simples, ele gera qual será o próximo endereço de instrução, para isso recebe sinais da UC para saber se haverá algum tipo de desvio (*control*), recebe um sinal da ULA para verificar se haverá salto condicional(*zero*) e mais 4 entradas, as que serão selecionadas(*branch, PCin, jimmediate e jreg*), sua saída é *PCout*, de 15 bits.

Algoritmo 4.8 – Multiplexador 5 bits

```
1  module Mux5(flag, dataA, dataB, out);
2
3    input flag;
4    input[4:0] dataA, dataB;
5    output reg[4:0] out;
6
7    always@(*)
8    begin
9      if(flag)
10     out = dataB;
11     else
12     out = dataA;
13   end
14
15 endmodule
```

Algoritmo 4.9 – Multiplexador 16 bits

```
1  module Mux16(flag, PC, switches, immediate, out);
2
3    input[1:0] flag;
4    input[15:0] PC;
5    input[15:0] switches, immediate;
6    output reg[15:0] out;
7
8  always@(*)
9    begin
10   case(flag)
11     default: //immediate
12     out = immediate;
13     2'b01: //PC
14     out = PC;
15     2'b10: //switches
16     out = switches;
```

```
17   endcase
18   end
19 endmodule
```

Algoritmo 4.10 – Multiplexador 32 bits

```
1  module Mux32(flag, dataA, dataB, out);
2
3    input flag;
4    input[31:0] dataA, dataB;
5    output reg[31:0] out;
6
7    always@(*)
8    begin
9    if(flag)
10     out = dataB;
11   else
12     out = dataA;
13   end
14 endmodule
```

Algoritmo 4.11 – Multiplexador PC

```
1  module MuxPC(zero, control, branch, jimmediate,
2          jreg, PCin, PCout);
3    input zero;
4    input[2:0] control;
5    input[31:0] branch, jimmediate, jreg;
6    input [15:0] PCin;
7    reg [15:0] in;
8    output reg[15:0] PCout;
9
10   always@(*)
11   begin
12     in = PCin + 16'd1;
13     case(control)
14       default: //incremento
15       begin
16     PCout = in;
17       end
18       3'b001: //BEQ
19       begin
20     if(zero)
21       PCout = in + branch[15:0];
22     else
23       PCout = in;
24       end
25       3'b010: //BNE
26       begin
```

```
27      if(~zero)
28        PCout = in + branch[15:0];
29      else
30        PCout = in;
31        end
32        3'b011: //Jump
33        begin
34      PCout = jimmediate[15:0];
35        end
36        3'b100: //Jump to reg
37        begin
38      PCout = jreg[15:0];
39        end
40      endcase
41    end
42  endmodule
```

4.1.8 Display

O módulo de Display(Algoritmo 4.12) faz parte da saída do processador, quando um número é gravado na memória de saída, ele é apresentado em *displays*. O display implementado recebe um valor de zero a nove em binário de 4 bits (*Entrada*) e organiza os 7 bits de saída(*Saida*) para que o número apareça de maneira correta na placa FPGA (5) utilizada para simular o circuito.

Algoritmo 4.12 – Display

```
1  module Display (Entrada, Saida);
2
3  input [3:0] Entrada;
4  output reg [6:0] Saida;
5
6  always@(*)
7  begin
8      case (Entrada)
9          4'b0000: Saida = 7'b0000001;
10         4'b0001: Saida = 7'b1001111;
11         4'b0010: Saida = 7'b0010010;
12         4'b0011: Saida = 7'b0000110;
13         4'b0100: Saida = 7'b1001100;
14         4'b0101: Saida = 7'b0100100;
15         4'b0110: Saida = 7'b0100000;
16         4'b0111: Saida = 7'b0001111;
17         4'b1000: Saida = 7'b0000000;
18         4'b1001: Saida = 7'b0001100;
19         default: Saida = 7'b1111111;
20     endcase
```

```
21 end
22 endmodule
```

4.1.9 Binário para BCD

Para que um número em binário apareça com mais de um caractere, utilizando mais de um *display*, é interessante utilizar um algoritmo conversor de binário para BCD(A codificação binária decima), e utilizar *displays* padrão de 0 a 9 para cada caractere. O algoritmo (Algoritmo 4.13) foi baseado em um algoritmo apresentado em (1), ele recebe um binário de até 8 bits *in* e retorna 3 binários de 4 bits *centena, dezena, unidade* e um sinal de negativo *negative*. Foi necessário tratar a saída de números negativos, como o verilog trata-os como complemento de 2, verifica-se se o bit mais significativo é 1 (negativo), caso seja, a *flag negative* recebe 1, os bits do número são invertidos e o número 1 é somado à ele, fazendo com que ele seja reconhecido pelos displays.

Algoritmo 4.13 – Binário para BCD baseado em (1)

```
1  module bin_bcd(in, centena, dezena, unidade, negative);
2
3    input [7:0] in;
4    reg [7:0] in2;
5    output reg [3:0] centena, dezena, unidade;
6    output reg negative;
7
8    integer i;
9
10   always@(in)
11   begin
12       if(in[7])
13     begin
14       in2 = ~in + 1;
15       negative = 1;
16     end
17     else
18     begin
19       in2 = in;
20       negative = 0;
21     end
22
23
24     centena = 4'd0;
25     dezena = 4'd0;
26     unidade = 4'd0;
27
28     for(i=7; i>=0; i=i-1)
29     begin
```

```
30      //add 3 to columns >=5
31
32      if(centena >= 5)
33      centena = centena + 4'd3;
34      if(dezena >= 5)
35      dezena = dezena + 4'd3;
36      if(unidade >= 5)
37        unidade = unidade + 4'd3;
38
39      // shift left one
40      centena = centena << 1;
41      centena[0] = dezena[3];
42      dezena = dezena << 1;
43      dezena[0] = unidade[3];
44      unidade = unidade << 1;
45      unidade[0] = in2[i];
46    end
47    end
48 endmodule
```

4.1.10 Módulo de Saída

O módulo de saída(Algoritmo 4.14) junta os displays e o conversor de binário para BCD, recebe o número guardado na memória para a saída e devolve 3 conjuntos de 4 bits e um bit indicando negativo que serão a saída para a placa FPGA.

Algoritmo 4.14 – Módulo de Saída

```
1  module OutputModule(in, out1, out2, out3, negative);
2
3    input [32:0] in;
4    wire [3:0] centena, dezena, unidade;
5    output wire negative;
6    output wire [6:0] out1, out2, out3;
7
8    bin_bcd translate(.in(in[7:0]), .centena(centena),
9        .dezena(dezena), .unidade(unidade),
10       .negative(negative));
11
12   Display d1(.Entrada(centena), .Saida(out1));
13   Display d2(.Entrada(dezena), .Saida(out2));
14   Display d3(.Entrada(unidade), .Saida(out3));
15
16 endmodule
```

4.2 Entrada de Dados

A entrada de dados é feita através de *switches* da placa FPGA, a espera da entrada de dados, que é implementada na Unidade de Controle, utiliza um sinal de parada no PC para esperar a entrada de dados, quando o sinal *ready* (alavanca no sistema FPGA) recebe valor 1, a Unidade de Controle indica a execução da instrução atual e retira o sinal de parada do PC.

4.3 A Unidade de Controle

A unidade de controle foi implementada como um circuito combinacional (Algoritmo A.1), apenas utilizando o clock para verificar o tempo certo em que será feita a continuação quando há um sinal de pausa relacionado a entrada de dados. Como entrada apresenta apenas os sinais *clk (clock), ready e opcode* e todos os sinais de controle da UC representados em Figura 1. Os sinais são: halt (sinal de parada), reset (sinal de novo início), sreg (sinal de escrita no banco de registradores), smux16 (sinal de escolha do multiplexadore de 16 bits), smux5 (sinal de escolha do multiplexadore de 5 bits), smux32 (sinal de escolha do multiplexadore de 32 bits da entrada da ULA), smuxPC (sinal de escolha do multiplexadore do PC), salu (sinal de controle de função a ULA), smem (sinal de escrita na memória de dados), sdisplay (sinal de escrita no registrador de saída do *display*), smemtoreg (sinal de escolha do multiplexador de 32 bits da entrada de dados no banco de registradores). Logo abaixo há um trecho do algoritmo da UC.

Algoritmo 4.15 – Trecho da Unidade de Controle

```
1  module ControlUnit(clk, ready, opcode, halt, reset, sreg, smux5, smux16,
       smux32, smuxPC, salu, smem, sdisplay, smemtoreg);
2
3    input [5:0] opcode;
4    input ready, clk;
5    output reg halt, reset, sreg, smux5, smem, sdisplay, smemtoreg, smux32;
6    output reg [1:0] smux16;
7    output reg [2:0] smuxPC;
8    output reg [3:0] salu;
9
10   always@(opcode or ready)
11   begin
12     case(opcode)
13       6'b000000:
14       begin
15         halt =0;
16         reset =0;
17         sreg =0;
18         smux5 =0;
```

```
19          smem =0;
20          sdisplay =0;
21          smemtoreg =0;
22          smux32 =0;
23          smux16 =2'b00; //1:0  2'b
24          smuxPC =3'b000; //2:0 3'b
25          salu =4'b0000;   //3:0  4'b
26        end
27        6'b000001:
28        begin
29          halt =0;
30          reset =0;
31          sreg =1;
32          smux5 =0;
33          smem =0;
34          sdisplay =0;
35          smemtoreg =0;
36          smux32 =0;
37          smux16 =2'b00; //1:0
38          smuxPC =3'b000; //2:0
39          salu =4'b0000;   //3:0
40        end
41        6'b000010:
42        begin
43          halt =0;
44          reset =0;
45          sreg =1;
46          smux5 =1;
47          smem =0;
48          sdisplay =0;
49          smemtoreg =0;
50          smux32 =1;
51          smux16 =2'b00; //1:0
52          smuxPC =3'b000; //2:0
53          salu =4'b0000;   //3:0
54        end
55      .
56      .
57      .
```

4.4 Implementação da CPU

Este é o módulo que organiza todos os outros módulos juntos, deixando como entrada o *clock*, os *switches* e o sinal *ready*, os sinais de saída são *out1, out2, out3, negative* (representam as saídas dos dispays) e *of* (representa *overflow* em alguma operação da

ULA), as demais saídas representadas no código (*muxMemtoreg_reg, pc_instmen e display*) estão sendo utilizados para analisar a corretude dos resultados através das formas de onda, que serão mostradas no Capítulo 5. Ao rodar o código na placa FPGA, os sinais de saída adicionais foram retirados, trabalhando apenas como fios internos e um *DeBouncer*(13) foi instanciado para filtrar oscilações indesejadas do botão de *clock*.

Algoritmo 4.16 – CPU

```
1  module CPU(clk, ready, switches, out1, out2, out3, negative, of,
       muxMemtoreg_reg, pc_instmem, display);
2
3    input clk, ready;
4    wire  clk;
5    wire  halt, reset, sreg, smux5, smem, sdisplay, smemtoreg, smux32;
6    wire [1:0] smux16;
7    wire [2:0] smuxPC;
8    wire [3:0] salu;
9    wire [15:0] newpc;
10   input [15:0] switches;
11   output wire [15:0] pc_instmem;
12   wire [31:0] mux32_alu;
13   wire [31:0] inst;
14   wire [4:0] mux5_reg;
15   wire [15:0] mux16_se16;
16   wire [31:0] se26_muxpc, se16_muxpc;
17   wire [31:0] saidaalu;
18   wire [31:0] A, B;
19   wire [31:0] mem_muxMemtoreg;
20   output wire [31:0] muxMemtoreg_reg;
21   output wire [31:0] display;
22   wire zero;
23   output wire [6:0] out1, out2, out3;
24   output wire negative, of;
25
26   //DeBounce db(.DB_out(clk), .clk(clock), .n_reset(1'b1), .button_in(~
       ndclk));
27
28   PC instPC(.inaddress(newpc), .outaddress(pc_instmem), .halt(halt), .clk(
       clk), .reset(reset));
29
30   ControlUnit instControlUnit(.clk(clk), .ready(ready), .opcode(inst
       [31:26]), .halt(halt), .reset(reset), .sreg(sreg), .smux5(smux5), .
       smux16(smux16), .smux32(smux32), .smuxPC(smuxPC), .salu(salu), .smem(
       smem), .sdisplay(sdisplay), .smemtoreg(smemtoreg));
31
32   InstMem instInstMem(.address(pc_instmem), .clk(clk), .out(inst));
33
```

```
34  Mux5 instMux5(.flag(smux5), .dataA(inst[15:11]), .dataB(inst[20:16]), .
      out(mux5_reg));
35
36  Mux16 instMux16(.flag(smux16), .PC(pc_instmem), .switches(switches), .
      immediate(inst[15:0]), .out(mux16_se16));
37
38  Mux32 instMux32(.flag(smux32), .dataA(B), .dataB(se16_muxpc), .out(
      mux32_alu));
39
40  RegBank instRegBank(.writeflag(sreg), .writedata(muxMemtoreg_reg), .clk(
      clk), .readregA(inst[25:21]),
41              .readregB(inst[20:16]), .writereg(mux5_reg), .dataA(A),
42              .dataB(B));
43
44  SE26 instSE26(.datain(inst[25:0]), .dataout(se26_muxpc));
45
46  SE16 instSE16(.datain(mux16_se16), .dataout(se16_muxpc));
47
48  ALU instALU(.operation(salu), .dataA(A), .dataB(mux32_alu), .saida(
      saidaalu), .zero(zero), .shamt(inst[10:6]), .of(of));
49
50  DataMem instDataMem(.clk(clk), .writeflag(smem), .address(saidaalu), .
      datain(B), .dataout(mem_muxMemtoreg), .displayflag(sdisplay), .display(
      display));
51
52  Mux32 Memtoreg(.flag(smemtoreg), .dataA(saidaalu), .dataB(mem_muxMemtoreg
      ), .out(muxMemtoreg_reg));
53
54  MuxPC instMuxPC(.zero(zero), .control(smuxPC), .branch(se16_muxpc), .
      jimmediate(se26_muxpc), .jreg(A), .PCin(pc_instmem), .PCout(newpc));
55
56  OutputModule instOutputModule(.in(display), .out1(out1), .out2(out2), .
      out3(out3), .negative(negative));
57
58
59 endmodule
```

5 Resultados Obtidos e Discussões

Os resultados foram separados em duas partes. A primeira parte é relacionada aos testes feitos em forma de onda (simulações no software *Quartus*), onde foram testados dois algoritmos, o de Fatorial e um algoritmo de teste com as instruções que não foram testadas no Fatorial.

5.1 Simulações Forma de Onda

O primeiro programa testado em forma de onda foi o mostrado no Algoritmo 4.3 (binário e "*Assembly*"), que calcula o fatorial do número de entrada. Sua simulação pode ser vista na imagens 3 e 4, que resultaram nas saídas 2 e 6 respectivamente.

Algoritmo 5.1 – Fatorial equivalente em *C*

```
int a,b,c,d;
inicio:
scanf(%d, &a);
b = a;
c = 1;
laco:
c = c * b;
b--;
if(b > 0)
  d=1;
else
  d=0;
if(d)
  goto laco;
printf(%d, c);
goto inicio;
```

O Segundo programa testado tem como objetivo mostrar que as outras instruções que não foram usadas no programa Fatorial também funcionam, o código em *assembly* pode ser visto no Algoritmo 5.2, o código em binário pode ser visto no Algoritmo 5.3, e sua execução na Figura 5, que resultou na saída 9.

Algoritmo 5.2 – Programa de teste em *Assembly*

```
nop
addi 0 1 8
add1 2
sub 1 2 1
sub1 2
```

```
and 1 1 3
or 1 1 3
xor 1 1 3
not 1 0 3
mod 1 1 3
div 1 1 3
shiftl 1 1 1
shiftr 1 1 1
sht 1 2 3
seq 1 2 3
bne 1 2 1
add 1 1 1
loadi 0 11 3
add 0 0 6
loada 5
addi 6 6 1
addi 0 7 1
bne 6 7 1
jr 5
subi 0 5 10
out 5
halt
```

Algoritmo 5.3 – Programa de teste em binário

```
mem[0] = 32'b00000000000000000000000000000000;
mem[1] = 32'b00001000000000010000000000001000;
mem[2] = 32'b00001100000000000000000000000000;
mem[3] = 32'b00010000001000100000100000000000;
mem[4] = 32'b00011000000000000000000000000000;
mem[5] = 32'b00011100001000010001100000000000;
mem[6] = 32'b00100000001000010001100000000000;
mem[7] = 32'b00100100001000010001100000000000;
mem[8] = 32'b00101000001000000011100000000000;
mem[9] = 32'b00101100001000010001100000000000;
mem[10] = 32'b01110000001000010001100000000000;
mem[11] = 32'b00110000001000000000100001000000;
mem[12] = 32'b00110100001000000000100001000000;
mem[13] = 32'b00111100001000100001100000000000;
mem[14] = 32'b01000000001000100001100000000000;
mem[15] = 32'b01001000001000100000000000000001;
mem[16] = 32'b00000100001000010000100000000000;
mem[17] = 32'b01100100000010110000000000000011;
mem[18] = 32'b00000100000000000011000000000000;
mem[19] = 32'b01101000000001010000000000000000;
mem[20] = 32'b00001000110001100000000000000001;
mem[21] = 32'b00001000000001110000000000000001;
mem[22] = 32'b01001000110001110000000000000001;
```

```
24 mem[23] = 32'b01010000101000000000000000000000;
25 mem[24] = 32'b00010100000001010000000000001010;
26 mem[25] = 32'b01011100000001010000000000000000;
27 mem[26] = 32'b01010100000000000000000000000000;
```

Figura 3 – Forma de onda do algoritmo Fatorial com entrada 2

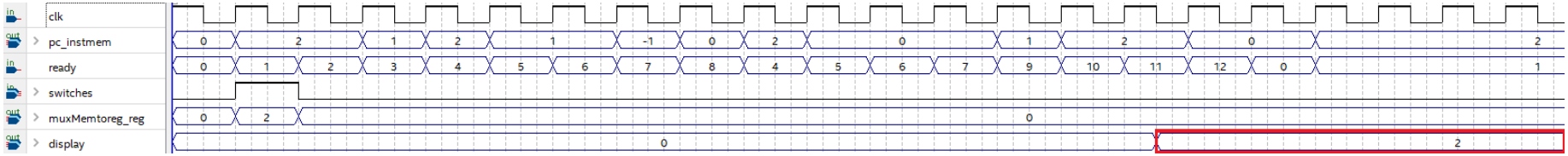


Figura 4 – Forma de onda do algoritmo Fatorial com entrada 3

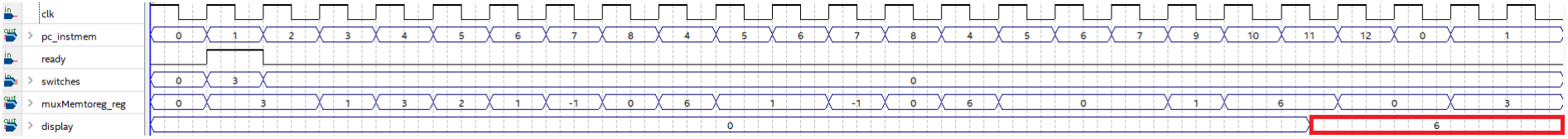
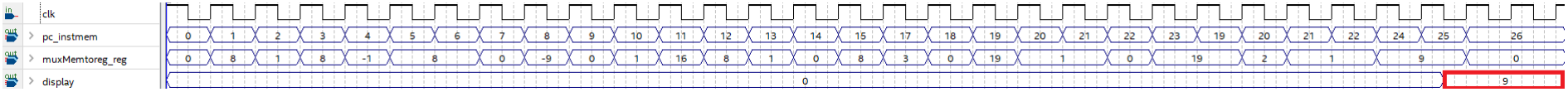


Figura 5 – Algoritmo de Teste



5.2 Simulações FPGA

Nas simulações feitas em FPGA foram utilizados os mesmos algoritmos, porém o algoritmo de fatorial foi executado com as entradas 3, 4 (Figura 9) e 5 (Figura 10), a entrada 3 foi simulada na Figura 4 e gera o mesmo valor de saída do executado em FPGA, que pode ser visto na Figura 8. A saída do algoritmo de teste com as instruções não utilizadas no fatorial também gerou o mesmo resultado, como se pode ver na Figura 5 e na Figura 11. A Figura 6 mostra o comportamento do FPGA ao começar a execução do programa, os *displays* ficam zerados, já a Figura 7 mostra como é feito para colocar um número de entra na instrução **IN**.

Figura 6 – Início de um algoritmo

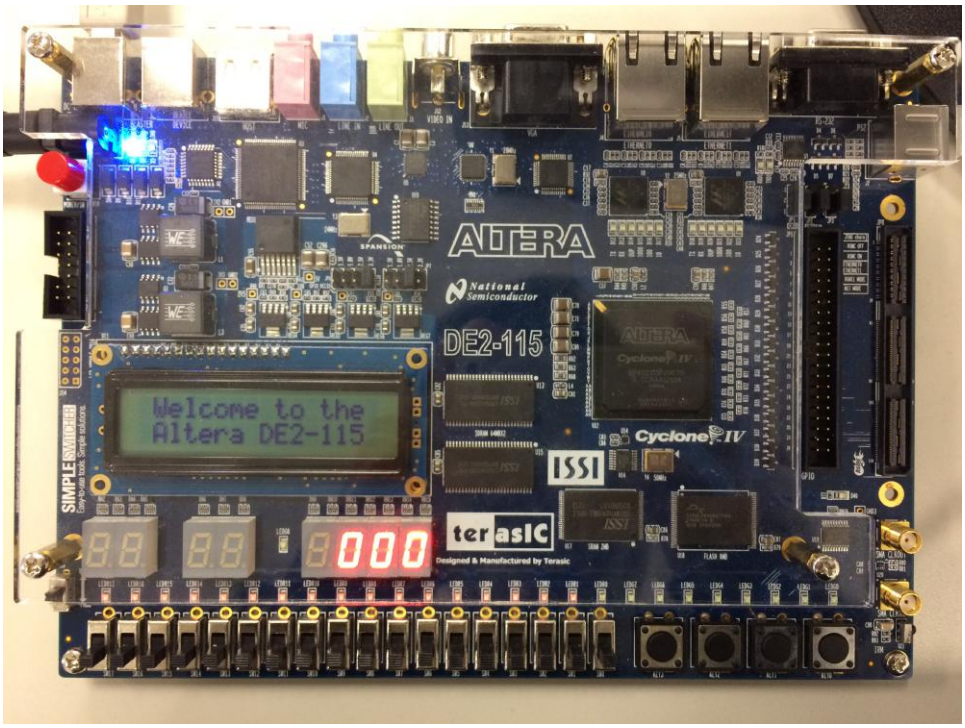


Figura 7 – Entrada no algoritmo fatorial

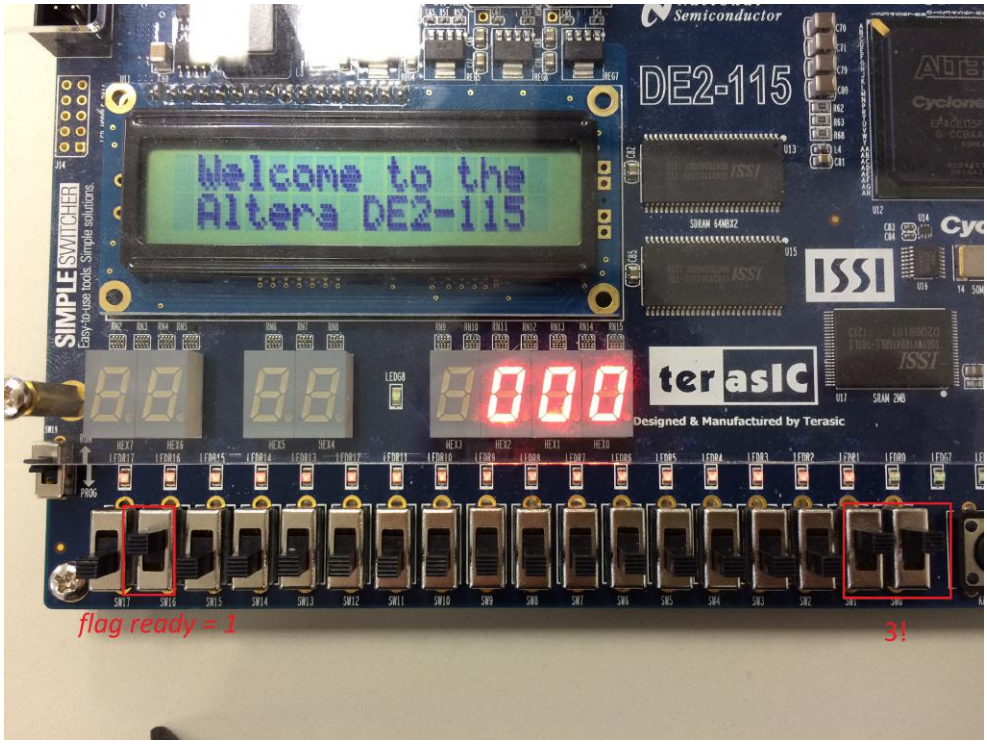


Figura 8 – Saída fatorial de 3

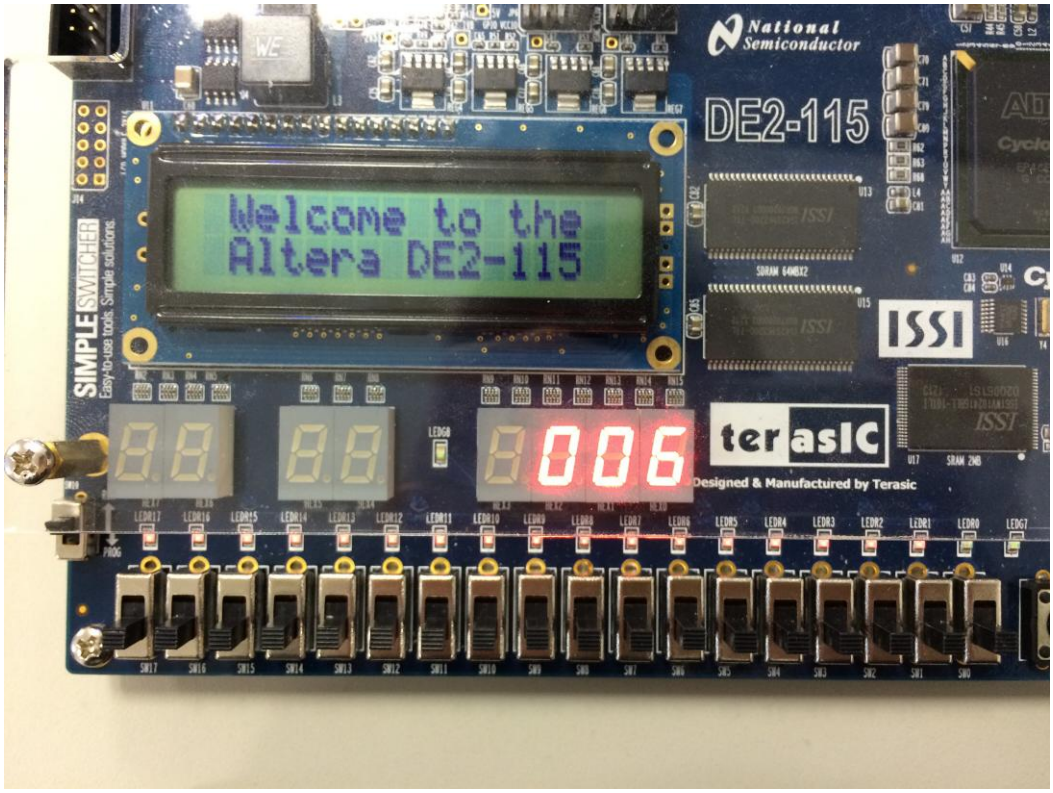


Figura 9 – Saída fatorial de 4

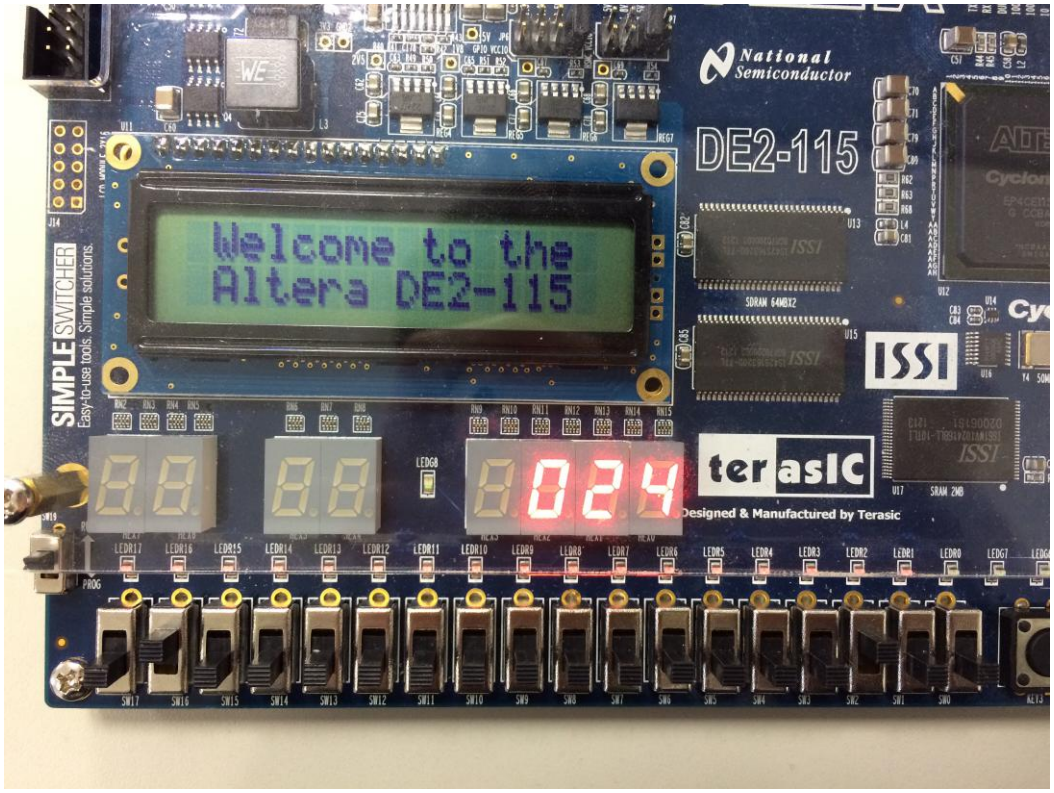


Figura 10 – Saída fatorial de 5

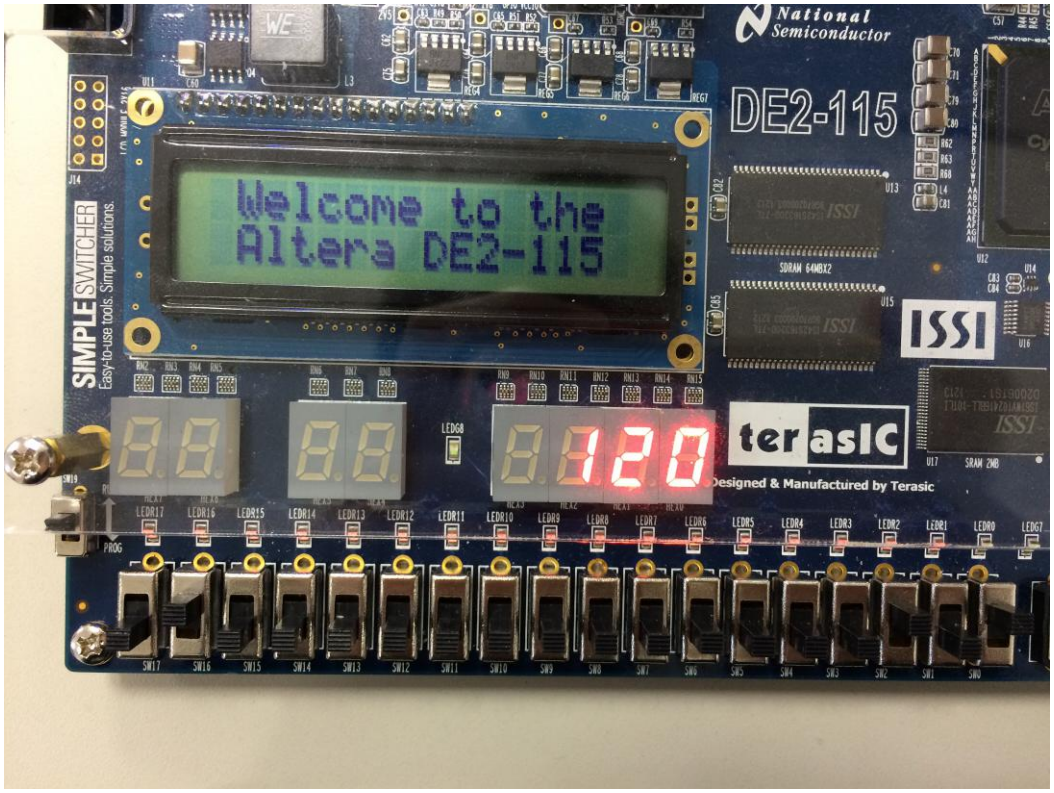
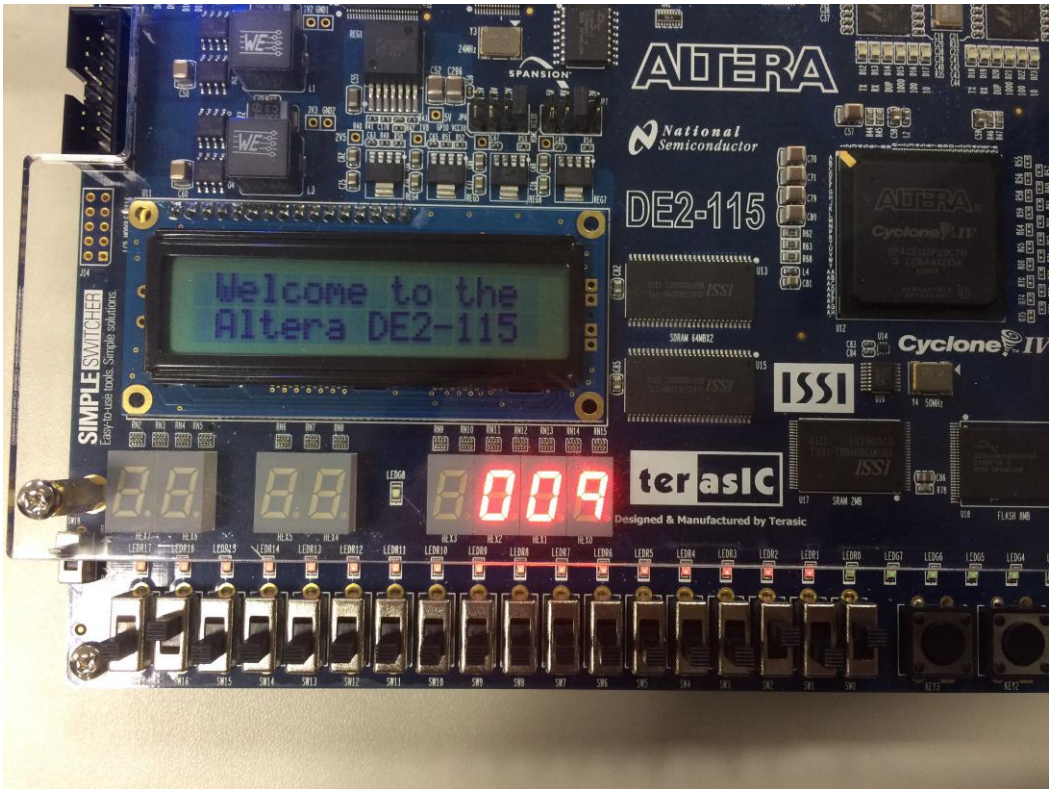


Figura 11 – Saída programa de teste



5.3 Discussões

Todos os resultados foram satisfatórios para todos os tipos de instruções, os resultados por simulação foram os que mais ajudaram a aprimorar a forma como as instruções estavam sendo executadas no processo de desenvolvimento, os testes em placas FPGA confirmaram o sucesso.

6 Considerações Finais

A elaboração desse trabalho trouxe ao autor profundo entendimento da arquitetura MIPS e gerou um aprendizado da linguagem de descrição de *hardware Verilog*, o maior aprendizado se dá quando problemas vêm a tona, forçando o engenheiro a entender realmente o que está acontecendo com o circuito no momento. A aplicação prática desse processador complementou a abordagem teórica de uma maneira única, tanto na experiência de gestão e desenvolvimento de projetos na área de engenharia de computação como elaboração de relatórios técnicos. Os próximos passos são desenvolver sistemas baseados no microprocessador desenvolvido e se necessário adicionar novas instruções e funções ao mesmo, já que sua implementação traz essa facilidade.

Referências

1 BINARY to BCD Conversion Algorithm. Apr 2017. Disponível em: <http://www.eng.utah.edu/~nmcdonal/Tutorials/BCDTutorial/BCDConversion.html>. Citado 2 vezes nas páginas 5 e 30.

2 NúMERO de smartphones em uso no Brasil chega a 168 milhões. Apr 2017. Disponível em: <http://folha.com/no1761310>. Citado na página 9.

3 PATTERSON, D. A.; HENNESY, J. L. *Computer Organization and Design*. 5th edition. ed. Waltham/MA, EUA: Morgan Kaufmann, 2007. Citado 5 vezes nas páginas 9, 13, 14, 15 e 16.

4 RISC vs. CISC. Apr 2017. Disponível em: <https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/>. Citado na página 9.

5 LEARNING FPGA And Verilog A Beginner's Guide. Apr 2016. Disponível em: <https://docs.numato.com/kb/learning-fpga-verilog-beginners-guide-part-1-introduction/>. Citado 3 vezes nas páginas 11, 15 e 29.

6 LúCIO, V. *Ponto de Checagem 1 - Processador*. [S.l.], 2017. Disponível em: <https://docs.google.com/document/d/1slDH0XpmfC2M3-FA579gJKlpXGvSNSphlUofoTj__Zo/edit?usp=sharing>. Citado 4 vezes nas páginas 13, 17, 19 e 20.

7 MIPS Achitecture. Apr 2017. Disponível em: <https://en.wikipedia.org/wiki/MIPS_architecture>. Citado na página 13.

8 MULTIPLEXADOR. Apr 2017. Disponível em: <https://pt.wikipedia.org/wiki/Multiplexador>. Citado na página 15.

9 VERILOG. Apr 2017. Disponível em: <https://en.wikipedia.org/wiki/Verilog>. Citado na página 15.

10 VERILOG HDL Reference Manual. [S.l.], 1999. Disponível em: <http://www.csit-sun.pub.ro/resources/xilinx/synver.pdf>. Citado na página 15.

11 COMPLEMENTO para dois. Mar 2017. Disponível em: <https://pt.wikipedia.org/wiki/Complemento_para_dois>. Citado na página 16.

12 CONTROL Unit. Jul 2017. Disponível em: <https://en.wikipedia.org/wiki/Control_unit>. Citado na página 17.

13 DEBOUNCER. Jul 2017. Disponível em: <https://eewiki.net/pages/viewpage.action?pageId=13599139>. Citado na página 34.

Apêndices

APÊNDICE A – Unidade de Controle

Algoritmo A.1 – Unidade de Controle

```
1  module ControlUnit(clk, ready, opcode, halt, reset, sreg, smux5, smux16,
       smux32, smuxPC, salu, smem, sdisplay, smemtoreg);
2
3    input [5:0] opcode;
4    input ready, clk;
5    output reg halt, reset, sreg, smux5, smem, sdisplay, smemtoreg, smux32;
6    output reg [1:0] smux16;
7    output reg [2:0] smuxPC;
8    output reg [3:0] salu;
9
10   always@(opcode or ready)
11   begin
12     case(opcode)
13       6'b000000:
14       begin
15         halt =0;
16         reset =0;
17         sreg =0;
18         smux5 =0;
19         smem =0;
20         sdisplay =0;
21         smemtoreg =0;
22         smux32 =0;
23         smux16 =2'b00; //1:0  2'b
24         smuxPC =3'b000; //2:0 3'b
25         salu =4'b0000;   //3:0  4'b
26       end
27       6'b000001:
28       begin
29         halt =0;
30         reset =0;
31         sreg =1;
32         smux5 =0;
33         smem =0;
34         sdisplay =0;
35         smemtoreg =0;
36         smux32 =0;
37         smux16 =2'b00; //1:0
38         smuxPC =3'b000; //2:0
39         salu =4'b0000;   //3:0
40       end
```

```
41        6'b000010:
42        begin
43          halt =0;
44          reset =0;
45          sreg =1;
46          smux5 =1;
47          smem =0;
48          sdisplay =0;
49          smemtoreg =0;
50          smux32 =1;
51          smux16 =2'b00; //1:0
52          smuxPC =3'b000; //2:0
53          salu =4'b0000;    //3:0
54        end
55        6'b000011:
56        begin
57          halt =0;
58          reset =0;
59          sreg =1;
60          smux5 =0;
61          smem =0;
62          sdisplay =0;
63          smemtoreg =0;
64          smux32 =0;
65          smux16 =2'b00; //1:0
66          smuxPC =3'b000; //2:0
67          salu =4'b0010;    //3:0
68        end
69        6'b000100:
70        begin
71          halt =0;
72          reset =0;
73          sreg =1;
74          smux5 =0;
75          smem =0;
76          sdisplay =0;
77          smemtoreg =0;
78          smux32 =0;
79          smux16 =2'b00; //1:0
80          smuxPC =3'b000; //2:0
81          salu =4'b0001;    //3:0
82        end
83        6'b000101:
84        begin
85          halt =0;
86          reset =0;
87          sreg =1;
```

```
88          smux5 =1;
89          smem =0;
90          sdisplay =0;
91          smemtoreg =0;
92          smux32 =1;
93          smux16 =2'b00; //1:0
94          smuxPC =3'b000; //2:0
95          salu =4'b0001;    //3:0
96        end
97        6'b000110:
98        begin
99          halt =0;
100         reset =0;
101         sreg =1;
102         smux5 =0;
103         smem =0;
104         sdisplay =0;
105         smemtoreg =0;
106         smux32 =0;
107         smux16 =2'b00; //1:0
108         smuxPC =3'b000; //2:0
109         salu =4'b0011;    //3:0
110       end
111       6'b000111:
112       begin
113         halt =0;
114         reset =0;
115         sreg =1;
116         smux5 =0;
117         smem =0;
118         sdisplay =0;
119         smemtoreg =0;
120         smux32 =0;
121         smux16 =2'b00; //1:0
122         smuxPC =3'b000; //2:0
123         salu =4'b0100;    //3:0
124       end
125       6'b001000:
126       begin
127         halt =0;
128         reset =0;
129         sreg =1;
130         smux5 =0;
131         smem =0;
132         sdisplay =0;
133         smemtoreg =0;
134         smux32 =0;
```

```
135         smux16 =2'b00; //1:0
136         smuxPC =3'b000; //2:0
137         salu =4'b0101;    //3:0
138       end
139       6'b001001:
140       begin
141         halt =0;
142         reset =0;
143         sreg =1;
144         smux5 =0;
145         smem =0;
146         sdisplay =0;
147         smemtoreg =0;
148         smux32 =0;
149         smux16 =2'b00; //1:0
150         smuxPC =3'b000; //2:0
151         salu =4'b0110;    //3:0
152       end
153       6'b001010:
154       begin
155         halt =0;
156         reset =0;
157         sreg =1;
158         smux5 =0;
159         smem =0;
160         sdisplay =0;
161         smemtoreg =0;
162         smux32 =0;
163         smux16 =2'b00; //1:0
164         smuxPC =3'b000; //2:0
165         salu =4'b0111;    //3:0
166       end
167       6'b001011:
168       begin
169         halt =0;
170         reset =0;
171         sreg =1;
172         smux5 =0;
173         smem =0;
174         sdisplay =0;
175         smemtoreg =0;
176         smux32 =0;
177         smux16 =2'b00; //1:0
178         smuxPC =3'b000; //2:0
179         salu =4'b1111;    //3:0
180       end
181       6'b001100:
```

```
182         begin
183           halt =0;
184           reset =0;
185           sreg =1;
186           smux5 =0;
187           smem =0;
188           sdisplay =0;
189           smemtoreg =0;
190           smux32 =0;
191           smux16 =2'b00; //1:0
192           smuxPC =3'b000; //2:0
193           salu =4'b1000;    //3:0
194         end
195         6'b001101:
196         begin
197           halt =0;
198           reset =0;
199           sreg =1;
200           smux5 =0;
201           smem =0;
202           sdisplay =0;
203           smemtoreg =0;
204           smux32 =0;
205           smux16 =2'b00; //1:0
206           smuxPC =3'b000; //2:0
207           salu =4'b1001;    //3:0
208         end
209         6'b001110:
210         begin
211           halt =0;
212           reset =0;
213           sreg =1;
214           smux5 =0;
215           smem =0;
216           sdisplay =0;
217           smemtoreg =0;
218           smux32 =0;
219           smux16 =2'b00; //1:0
220           smuxPC =3'b000; //2:0
221           salu =4'b1010;    //3:0
222         end
223         6'b001111:
224         begin
225           halt =0;
226           reset =0;
227           sreg =1;
228           smux5 =0;
```

```
229         smem =0;
230         sdisplay =0;
231         smemtoreg =0;
232         smux32 =0;
233         smux16 =2'b00; //1:0
234         smuxPC =3'b000; //2:0
235         salu =4'b1100;    //3:0
236       end
237       6'b010000:
238       begin
239         halt =0;
240         reset =0;
241         sreg =1;
242         smux5 =0;
243         smem =0;
244         sdisplay =0;
245         smemtoreg =0;
246         smux32 =0;
247         smux16 =2'b00; //1:0
248         smuxPC =3'b000; //2:0
249         salu =4'b1011;    //3:0
250       end
251       6'b010001:
252       begin
253         halt =0;
254         reset =0;
255         sreg =0;
256         smux5 =0;
257         smem =0;
258         sdisplay =0;
259         smemtoreg =0;
260         smux32 =0;
261         smux16 =2'b00; //1:0
262         smuxPC =3'b001; //2:0
263         salu =4'b0001;    //3:0
264       end
265       6'b010010:
266       begin
267         halt =0;
268         reset =0;
269         sreg =0;
270         smux5 =0;
271         smem =0;
272         sdisplay =0;
273         smemtoreg =0;
274         smux32 =0;
275         smux16 =2'b00; //1:0
```

```
276         smuxPC =3'b010; //2:0
277         salu =4'b0001;    //3:0
278       end
279       6'b010011:
280       begin
281         halt =0;
282         reset =0;
283         sreg =0;
284         smux5 =0;
285         smem =0;
286         sdisplay =0;
287         smemtoreg =0;
288         smux32 =0;
289         smux16 =2'b00; //1:0
290         smuxPC =3'b011; //2:0
291         salu =4'b0000;    //3:0
292       end
293       6'b010100:
294       begin
295         halt =0;
296         reset =0;
297         sreg =0;
298         smux5 =0;
299         smem =0;
300         sdisplay =0;
301         smemtoreg =0;
302         smux32 =0;
303         smux16 =2'b00; //1:0
304         smuxPC =3'b100; //2:0
305         salu =4'b0000;    //3:0
306       end
307       6'b010101:
308       begin
309         halt =1;
310         reset =0;
311         sreg =0;
312         smux5 =0;
313         smem =0;
314         sdisplay =0;
315         smemtoreg =0;
316         smux32 =0;
317         smux16 =2'b00; //1:0
318         smuxPC =3'b000; //2:0
319         salu =4'b0000;    //3:0
320       end
321       6'b010110: //IN
322       begin
```

```
323         if(ready==1)
324           begin
325           halt =0;
326           reset =0;
327           sreg =1;
328           smux5 =1;
329           smem =0;
330           sdisplay =0;
331           smemtoreg =0;
332           smux32 =1;
333           smux16 =2'b10;  //1:0
334           smuxPC =3'b000; //2:0
335           salu =4'b0000;   //3:0
336           end
337         else if(clk==1)
338           begin
339             halt =1;
340             reset =0;
341             sreg =0;
342             smux5 =0;
343             smem =0;
344             sdisplay =0;
345             smemtoreg =0;
346             smux32 =0;
347             smux16 =2'b00;  //1:0 2'b
348             smuxPC =3'b000; //2:0 3'b
349             salu =4'b0000;   //3:0  4'b
350           end
351       end
352       6'b010111:
353       begin
354         halt =0;
355         reset =0;
356         sreg =0;
357         smux5 =0;
358         smem =0;
359         sdisplay =1;
360         smemtoreg =0;
361         smux32 =0;
362         smux16 =2'b00; //1:0
363         smuxPC =3'b000; //2:0
364         salu =4'b0000;   //3:0
365       end
366       6'b011000:
367       begin
368         halt =0;
369         reset =0;
```

```
370         sreg =1;
371         smux5 =1;
372         smem =0;
373         sdisplay =0;
374         smemtoreg =1;
375         smux32 =1;
376         smux16 =2'b00; //1:0
377         smuxPC =3'b000; //2:0
378         salu =4'b0000;    //3:0
379       end
380       6'b011001:
381       begin
382         halt =0;
383         reset =0;
384         sreg =1;
385         smux5 =1;
386         smem =0;
387         sdisplay =0;
388         smemtoreg =0;
389         smux32 =1;
390         smux16 =2'b00; //1:0
391         smuxPC =3'b000; //2:0
392         salu =4'b0000;    //3:0
393       end
394       6'b011010:
395       begin
396         halt =0;
397         reset =0;
398         sreg =1;
399         smux5 =1;
400         smem =0;
401         sdisplay =0;
402         smemtoreg =0;
403         smux32 =1;
404         smux16 =2'b01; //1:0
405         smuxPC =3'b000; //2:0
406         salu =4'b0000;    //3:0
407       end
408       6'b011011:
409       begin
410         halt =0;
411         reset =0;
412         sreg =1;
413         smux5 =0;
414         smem =0;
415         sdisplay =0;
416         smemtoreg =0;
```

```
417         smux32 =0;
418         smux16 =2'b00; //1:0
419         smuxPC =3'b000; //2:0
420         salu =4'b1101;    //3:0
421       end
422       6'b011100:
423       begin
424         halt =0;
425         reset =0;
426         sreg =1;
427         smux5 =0;
428         smem =0;
429         sdisplay =0;
430         smemtoreg =0;
431         smux32 =0;
432         smux16 =2'b00; //1:0
433         smuxPC =3'b000; //2:0
434         salu =4'b1110;    //3:0
435       end
436       6'b011101:
437       begin
438         halt =0;
439         reset =0;
440         sreg =0;
441         smux5 =1;
442         smem =1;
443         sdisplay =0;
444         smemtoreg =0;
445         smux32 =1;
446         smux16 =2'b00; //1:0
447         smuxPC =3'b000; //2:0
448         salu =4'b0000;    //3:0
449       end
450       default:
451       begin
452         halt =0;
453         reset =1;
454         sreg =0;
455         smux5 =0;
456         smem =0;
457         sdisplay =0;
458         smemtoreg =0;
459         smux32 =0;
460         smux16 =2'b00; //1:0
461         smuxPC =3'b000; //2:0
462         salu =4'b0000;    //3:0
463       end
```

```
464       endcase
465     end
466
467 endmodule
```