UNIVERSIDADE FEDERAL DE SANTA CATARINA
Centro Tecnológico - CTC

Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Curso de Ciências da Computação

INE5406 – Sistemas Digitais

PROJETO PRÁTICO DE SISTEMAS DIGITAIS:

UNIDADE DE PROCESSAMENTO SIMPLES COM ARQUITETURA BASEADA EM PILHAS

Gabriel Baiocchi de Sant'Anna

Florianópolis
2018

Gabriel Baiocchi de Sant’Anna

PROJETO PRÁTICO DE SISTEMAS DIGITAIS:
UNIDADE DE PROCESSAMENTO SIMPLES COM ARQUITETURA BASEADA EM PILHAS

Trabalho da disciplina “INE5406 – Sistemas Digitais” apresentado ao Curso de Ciências da Computação do Departamento de Informática e Estatística no Centro Tecnológico da Universidade Federal de Santa Catarina.

Professor: Rafael Luiz Cancian, Dr. Eng.

Florianópolis
2018-2

Lista de Algoritmos

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

ADDR – Barramento de endereço da memória externa.
ALU – *Arithmetic Logic Unit* (ou ULA em português).
CIR – *Current Instruction Register*.
CPI – *Cycles per Instruction*.
CPU – *Central Processing Unit*.
DS – *Data Stack*.
FPGA – *Field Programmable Gate Array*.
FSMD – *Finite State Machine with Datapath*.
HDL – *Hardware Description Language*.
I/O – *Input / Output*.
LIFO – *Last In First Out*.
MEM – Barramento de dados da memória externa.
MISC – *Minimal Instruction Set Computer*.
ML0 – Classificação de design de máquina de pilha por Philip J. Koopman.
MMIO – *Memory Mapped I/O*.
MSB – *Most Significant Bit*.
PC – *Program Counter*.
RAM – *Random Access Memory.*
ROM – *Read-Only Memory*.
RS – *Return Stack*.
RTL – *Register-Transfer Level*.

S4PU – *Simple Forth Processing Unit*.

TOS – *Top of Stack*.

VHDL - *Very High Speed Integrated Circuit HDL*.

Sumário

1 Introdução

Este projeto visa desenvolver um sistema digital síncrono que opere como um **processador simples** de **16 bits** utilizando **arquitetura baseada em pilha** otimizada para executar ao menos um subconjunto da linguagem **Forth**. O mesmo será **descrito em HDL** para que possa então ser sintetizado, simulado e **prototipado em FPGA**.

O processador seguirá o **design ML0**, ou seja, uma máquina com múltiplas pilhas com algum sistem de *buffer* em *hardware* e um conjunto de instruções mínimo (MISC) sem campos de operandos. A classificação provém do livro ***Stack Computers: the new wave***, por Philip J. Koopman, que serve de inspiração a este projeto.

Deve ser identificada alguma maneira de programar o sistema e receber respostas do programa. Portanto, serão concebidas uma interface de I/O simples e uma forma de acesso à memória da máquina.

1.1 Pilhas LIFO

Uma pilha ou *stack* nada mais é do que uma forma simplista de guardar dados temporários que serão utilizados por alguma operação. Pode-se visualizar esse esquema imaginando um baralho; retire dele duas cartas quaisquer e as coloque uma após a outra em um monte separado, com os números voltados para cima. Perceba que **a única carta imediatamente acessível na pilha é a última a ter sido colocada ali**, criando o padrão **LIFO** (*Last In First Out)*.

Digamos que deseja-se realizar a operação binária de adição sobre os elementos na pilha. Para isso, é efetuado o seguinte algoritmo (seguindo ainda o exemplo do baralho):

- Retire as duas cartas do topo da pilha e as coloque de volta no baralho.
- Procure no baralho a carta que represente a soma dos valores das duas retiradas no passo anterior. Despeje-a no topo da pilha.

O *stack* será ilustrado como uma sequência de valores indo da esquerda para a direita, sendo o elemento mais à direita o topo da pilha. A operação efetuada também será representada, embora de maneira diferente do tradicional, mas que pode ser considerada mais adequada para a realização de instruções sequenciais: a chamada **Notação Polonesa Inversa** (ou notação pós-fixada). (FSF, 2008)

Supondo que as cartas retiradas do baralho foram 2 e 5, respectivamente, teríamos a seguinte sequência:

- **2 5 +**
- **7**

Assim, uma função aplicada sobre a pilha consome o número de argumentos que necessita de seu topo e deixa ali o(s) seu(s) resultado(s). Caso a máquina recebesse novamente uma instrução de operação binária após o fim da anterior não haveriam operandos o suficiente no *stack* para a sua realização, ocorrendo então um ***stack underflow***. A situação inversa acontece ao admitir que a pilha possui um tamanho finito e uma operação tentasse adicionar um elemento no topo de um *stack* já em seu limite de capacidade, acarretando no chamado ***stack overflow***.

1.2 Arquitetura Baseada em *Stack*

O principal motivo para se utilizar um processador com arquitetura baseada em pilha é a **redução da complexidade do *hardware*** que lida com a seleção de operandos e armazenamento de resultados, pois é seguro supor o topo do *stack* como entrada e saída de (quase) qualquer

operação, possibilitando realocar esses valores para unidades de memória rápidas como *caches* de registradores conectados diretamente à ULA. Além disso, as instruções utilizadas pelo processador podem também ser simplificadas ao serem expressadas apenas por um código de operação de poucos bits, sem campos de operandos; minimizando assim o processo de decodificação e tornando os programas mais compactos. (KOOPMAN, 1989)

Mesmo com tamanha simplicidade, uma máquina de pilha pode ser otimizada para alcançar níveis de performance perfeitamente comparáveis com os de processadores mais complexos. A linguagem Forth, por exemplo, opera sob uma interface com duas pilhas - uma para dados e outra para endereços de retorno – de forma a facilitar e agilizar **chamadas de sub-rotinas, mesmo que recursivas**, sem “sujar” a pilha de dados. (KOOPMAN, 1989)

Uma das desvantagens da utilização deste tipo de arquitetura é encontrada ao tentar acessar elementos colocados mais abaixo no *stack*, pois isso pode levar múltiplos ciclos se não houver alguma forma mais eficiente de acessar o ‘enésimo’ elemento da pilha. Essas e outras complicações são geralmente evitadas por um conjunto de instruções que tenha sido projetado tendo-as em mente.

Estima-se também que, devido à simplicidade da arquitetura, esta seja **ideal para o ensino** de sistemas digitais e organização de computadores em engenharias e nas ciências computação. Criar uma máquina de pilha, seja em *software* (como uma máquina virtual executando *bytecode*) ou em *hardware* (para prototipação em FPGA ou montagem em placas de circuito impresso), deve ser uma tarefa perfeitamente realizável por qualquer estudante nessas áreas.

2 Projeto do Sistema

O projeto inicia-se com a identificação das entradas e saídas, a descrição e captura do comportamento do mesmo e a definição de um conjunto de instruções para o processador, o que é realizado nas seguintes seções deste capítulo.

2.1 Identificação das Entradas e Saídas

O sistema digital proposto implementa uma interface com três sinais principais de entrada de um único bit, quatro saídas indicadoras de erro (*overflow* e *underflow*) nos *stacks* e conexões para acesso de escrita e leitura em uma interface de memória mapeada com **2^{16} (64 Ki) palavras de 16 bits endereçadas consecutivamente**. Pode-se pensar nessa interface de memória como **o sistema no qual o processador está embarcado**.

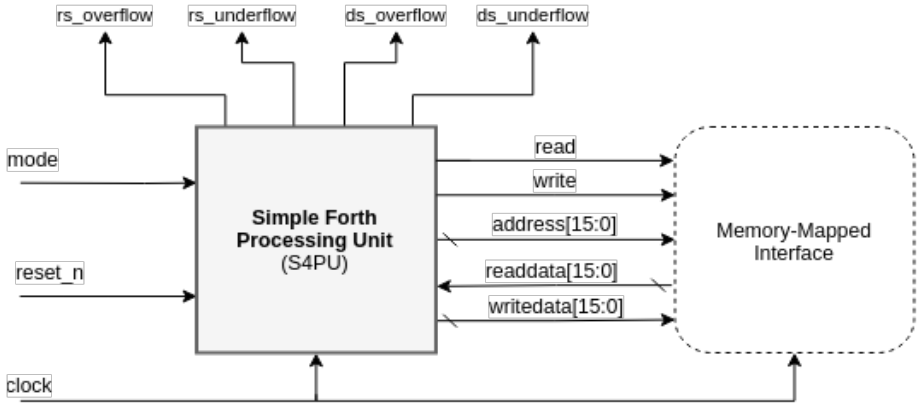


Figura 01 – Interface do Sistema. O componente de memória se faz necessário para a operação correta do processador. Fonte: o autor (2018).

A entrada de *clock* cumpre a função de sincronização da unidade de processamento com outros dispositivos (incluindo a memória) enquanto o sinal *reset_n* (ativo em nível baixo) serve para reiniciar o estado interno do processador de forma síncrona.

O bit *mode* permite ao usuário escolher o modo de operação que o processador tomará após um reset: alternando entre o **modo computador** (0), em que as instruções são lidas da memória principal; e um **modo programador** (1), em que a leitura de instruções ocorre em endereços reservados (memória de programa).

A interface cumpre também a função de comunicação com dispositivos externos, que é realizada pelo processador da mesma maneira com que acessa a memória: operações de leitura (requisitada com *read* em nível alto) ou escrita (sinal *write* em nível alto) em um endereço especificado por *address* cujo fluxo de dados corre pelo respectivo barramento (*readdata* ou *writedata*). Dito isso, segue abaixo o mapeamento sugerido da memória como visto pela CPU:

Função	Faixa de Endereços	Tamanho Reservado
Memória Principal	`0x0000 - 0x7FFF`	32 Ki
Memória de Programa	`0x8000 - 0xA3FF`	9 Ki
Memória de Vídeo	`0xA400 - 0xFDFF`	Aprox. 23 Ki (160x144)
Outros dispositivos MMIO, periféricos, etc.	`0xFE00 - 0xFEFF`	256
Tratamento de *underflow/overflow*.	`0xFF00 - 0xFFFF`	256

Tabela 1 – Mapeamento da Memória. **Os endereços iniciais da memória principal e de programa são fixos**. Os demais dependem da implementação.

2.2 Descrição e Captura do Comportamento

Em ambos os modos de operação, o comportamento do processador deve seguir de acordo com o que está estabelecido na memória, realizando, para cada instrução lida, um ciclo envolvendo sua decodificação, execução, armazenamento de resultado (quando houver) e a leitura da próxima instrução a ser processada.

Um registrador (contador de programa) será responsável por manter o endereço da próxima instrução, simplesmente incrementando o

anterior ou carregando algum valor externo em desvios de controle. Assim, a memória principal fornecerá a instrução a um outro registrador (registrador de instrução atual) para que esta possa ser processada. O comportamento pode ser ilustrado pelo código abaixo.

```
int[] memory = new int[MEMORY_SIZE];
int prog_count = 0 + MODE_OFFSET*mode;

while (true) {
    int curr_instruction = memory[prog_count];
    ++prog_count;
    execute(curr_instruction);
}
```

Algoritmo 1 – Ciclo de Operação do Processador.

A pilha LIFO é implementada como uma memória e um registrador de endereço que realiza duas operações básicas: colocar um valor na pilha (**PUSH**) ou expor o seguinte no próximo ciclo (**POP**); e uma terceira operação (PICK) para copiar o 'enésimo' elemento no fundo do *stack* para o topo. **Tanto pode-se fazer com que a pilha cresça a partir dos endereços mais altos como dos mais baixos**.

```
public class Stack {
    private int[] values = new int[STACK_SIZE];
    private int top_of_stack = STACK_SIZE;

    public void reset() {
        top_of_stack = STACK_SIZE;
    }

    public int pop() {
        int temp = values[top_of_stack];
        ++top_of_stack;
        return temp;
    }

    public void push(int value) {
        --top_of_stack;
        values[top_of_stack] = value;
    }

    public void pick(n) {
        values[top_of_stack] = values[top_of_stack + n + 1];
    }
}
```

Algoritmo 2 – Comportamento de um *Stack* LIFO.

Para cada instrução lida, analisa-se o seu bit menos significativo: quando for 1 os demais bits indicam o código da operação a ser realizada e quando for 0 indica chamada de sub-rotina no endereço dados pelos 15 bits restantes. Assim sendo, **é possível endereçar sub-rotinas em toda a extensão da memória, porém apenas em endereços de valor par**.

Por fim, podemos classificar as instruções do processador de maneira a generalizar sua execução algorítmica:

- **Operações Lógicas e Aritméticas**: Soma, subtração, AND, OR, XOR, NOT, comparação de igualdade, menor, maior, etc.
  - Retira-se do topo da pilha de dados os operandos necessários.
  - Coloca-se o resultado no topo da pilha de dados.
- **Carregamento de valores na pilha**.
  - O próximo elemento na memória é tratado como valor e é colocado no topo da pilha de dados.
- **Acesso à memória mapeada.**
  - Retira-se do topo da pilha de dados o endereço a ser acessado.
  - No endereço de memória, executa a leitura (guardando o resultado no topo da pilha de dados) ou a escrita (retirando o valor do topo da pilha de dados).
- **Tomada de decisões**: código executado condicionalmente.
  - Retira-se o valor do topo da pilha de dados.
  - Se o valor lido for zero (equivale a falso), o próximo elemento da memória é tratado como endereço e colocado no registrador que aponta para a próxima instrução a ser executada, sendo utilizado no próximo ciclo.
  - Caso contrário, incrementa o registrador com o endereço da instrução seguinte.
- **Execução de sub-rotinas**: utiliza a pilha de retorno.
  - Coloca-se no topo da pilha de retorno o endereço da próxima instrução.

- A instrução atual é tratada como endereço e carregada no registrador que aponta para a próxima instrução.
- Ao retornar da sub-rotina, basta retirar o elemento do topo da pilha de retorno e carregá-lo no registrador de endereço de instrução.

- **Outras**: inclui operações de manipulação da pilha de dados e de retorno, entre outras.

```
        public void execute(int instruction) {
            switch (instruction) {
                case ALU_OPERATION:
                    int b = DataStack.pop();
                    int a = DataStack.pop();
                    DataStack.push( operate(a,b) );
                    break;

                case LITERAL:
                    DataStack.push( memory[prog_count] );
                    ++prog_count;
                    break;

                case MEM_STORE:
                    int address = DataStack.pop();
                    memory[address] = DataStack.pop();
                    break;

                case BRANCH_IF_ZERO:
                    int condition = DataStack.pop();
                    if (condition != 0) {
                        ++prog_count;
                    } else {
                        prog_count = memory[prog_count];
                    }
                    break;

                case CALL_SUBROUTINE:
                    ReturnStack.push( prog_count );
                    prog_count = instruction;
                    break;

                case RETURN:
                    prog_count = ReturnStack.pop();
                    break;

                // . . . other instructions . . .
            }
        }
```

Algoritmo 3 – Decodificação e Execução de Instruções.

2.3 Conjunto de Instruções da Arquitetura

Define-se para a máquina um conjunto de 32 instruções simples que tratam de um único tipo numérico, mas que podem ser combinadas para gerar comportamentos mais complexos.

Instrução	Código	Pilha (antes -- depois)	Descrição	Ciclos
CALL addr	`[addr]0`	--	A execução do programa é redirecionada para a sub-rotina apontada pelos 15 bits mais significativos desta instrução.	2
NOP	`0x8001`	--	Consome um ciclo sem operação alguma.	1
@	`0x8003`	A -- X	Lê o valor guardado no endereço de memória A e o coloca no topo do stack.	2
!	`0x8005`	X A --	Escreve o valor X no endereço de memória A.	2
IF	`0x8007`	B --	Se B for falso (zero), a execução do programa é redirecionada para o endereço contido no próximo elemento de memória. Caso contrário, continua com a instrução seguinte.	2
BRANCH	`0x8009`	--	Desvio incondicional para o endereço contido no próximo elemento da memória.	2
EXIT	`0x800D`	--	Retorna da sub-rotina atual.	2
DROP	`0x800F`	X --	Limpa o valor no topo da pilha.	1
LIT	`0x8011`	-- X	O próximo elemento na memória é colocado na pilha de dados.	2
PICK	`0x8013`	$X_N \ldots X_1\ X_0\ N$ -- $X_N \ldots X_1\ X_0\ X_N$	Lê o N-ésimo elemento do stack depois de N e o copia no topo do stack.	2
>R	`0x8015`	X --	Retira X do topo do stack de dados e o coloca no de retorno.	1
R>	`0x8017`	-- X	Retira o valor do topo da pilha de retorno e o coloca na de dados como X.	1
NOT	`0x8019`	X -- ¬X	Inverte bit a bit o valor no topo do stack.	1
OR	`0x801B`	X Y -- (X|Y)	Operação bit a bit OR entre X e Y.	1
AND	`0x801D`	X Y -- (X&Y)	Operação bit a bit AND entre X e Y.	1

XOR	0x801F	X Y -- (X^Y)	Operação bit a bit XOR entre X e Y.	1
+	0x8021	X Y -- (X+Y)	Soma X e Y, a adição é deixado no topo da pilha.	1
-	0x8023	X Y -- (X-Y)	Subtrai Y de X, a diferença é colocada no topo da pilha.	1
1+	0x8025	X -- (X+1)	Incrementa o valor no topo do stack.	1
1-	0x8027	X -- (X-1)	Decrementa o valor no topo do stack.	1
=	0x8029	X Y -- B	Comparação X=Y, o resultado será zero quando a igualdade for falsa e 0xFFFF quando verdadeira.	1
<	0x802B	X Y -- B	Comparação X<Y, o resultado será zero quando falso e 0xFFFF quando verdadeiro.	1
>	0x802D	X Y -- B	Comparação X>Y, o resultado será zero quando falso e 0xFFFF quando verdadeiro.	1
0=	0x8035	X -- B	Comparação X=0, o resultado será zero quando a igualdade for falsa e 0xFFFF quando verdadeira.	1
0<	0x8037	X -- B	Comparação X<0, o resultado será zero quando falso e 0xFFFF quando verdadeiro.	1
0>	0x8039	X -- B	Comparação X>0, o resultado será zero quando falso e 0xFFFF quando verdadeiro.	1
2*	0x8041	X -- (X*2)	Desloca os bits de X uma posição para a esquerda, completando com zero o bit menos significativo.	1
2/	0x8043	X -- (X/2)	Desloca os bits de X uma posição para a direita, mantendo o bit mais significativo.	1
DUP	0x804D	X -- X X	Coloca no topo do stack uma cópia de X.	1
SWAP	0x804F	X Y -- Y X	Troca a ordem dos dois elementos no topo do stack.	2
SBL	0x8051	X -- (X<<8)	Desloca X um octeto para a esquerda, completando com zeros os bits menos significativos.	1
SBR	0x8053	X -- (X>>8)	Desloca X um octeto para a direita, completando com zeros os bits mais significativos.	1

Tabela 2 – Conjunto de Instruções. A última coluna depende da implementação do Bloco de Controle e da especificação da memória externa. A média simples dos seus elementos resulta em um valor de **1,28125 CPI**. (FORTH, 1994)

Uma primeira captura do sistema em FSMD de alto nível reflete o projeto realizado até então:

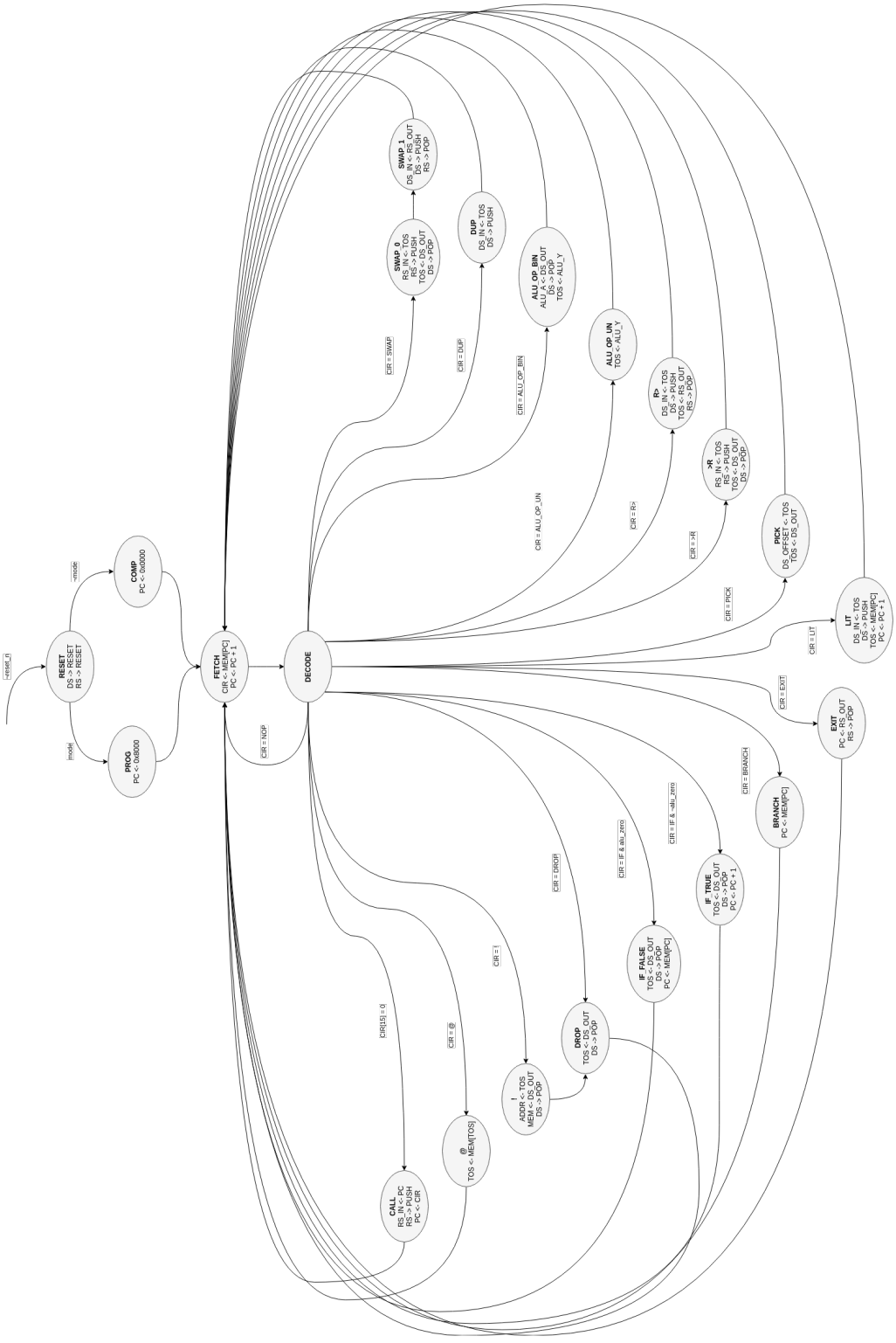


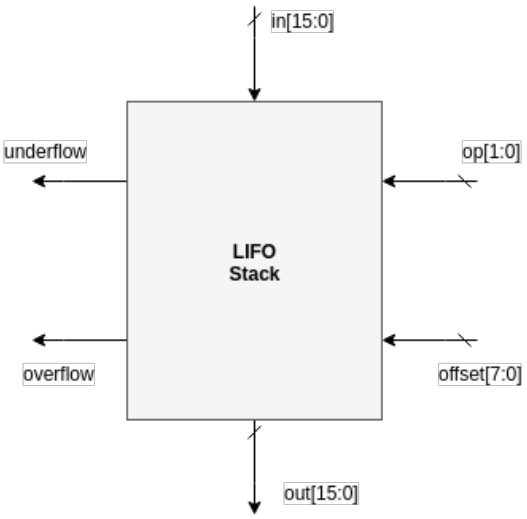
Figura 02 – FSMD Inicial. Implementação sem *pipelining* requer um mínimo de três ciclos para cada instrução. Operações da ULA foram agrupadas em ALU_OP_BIN e ALU_OP_UN; TOS é o registrador de *buffer* do valor do topo da pilha de dados. Fonte: o autor (2018).

2.4 Projeto do Bloco Operativo

Tendo o comportamento e o conjunto de instruções do sistema definidos, dá-se início ao projeto do bloco operativo (*datapath*) com a identificação das interfaces e comportamento de seus principais componentes. Sinais de *clock* foram omitidos nas figuras abaixo para facilitar a sua representação.

Pilhas LIFO

A cada início de pulso de relógio a saída *out* deve disponibilizar o valor lido no ciclo anterior a um número *offset* (também afeta operações de escrita) de posições a partir do topo da pilha. Os sinais de *underflow* e *overflow* indicam erros na pilha. Os bits de *op* definem a operação a ser completada pelo bloco até o próximo pulso de relógio.



op	Operação realizada
00	RESET – Limpa a pilha.
01	POP – Remove o valor do topo da pilha.
10	PUSH – Insere o valor lido em *in* no topo do *stack*.
11	NOP – Nenhuma operação.

Tabela 3 – Operações da Pilha.

Figura 03 – Interface dos *Stacks*. Fonte: o autor (2018).

Unidade Lógica-Aritmética

A ULA da máquina opera sempre com algum valor externo (A) e o valor do topo da pilha de dados, portanto este último é mantido em um registrador (TOSREG) conectado diretamente à segunda entrada de operandos do bloco. Como o resultado de uma operação é sempre salvo

no topo do *stack*, a saída da ULA também é redirecionada a este mesmo registrador. É desta forma que, nesta implementação, **o valor vindo da pilha de dados representa, na verdade, o segundo elemento a partir do topo do *stack***. Essa otimização permite realizar operações lógicas e aritméticas em um único ciclo pois ambos os operandos estão sempre imediatamente disponíveis.

O bloco inclui sinais de saída indicando resultado nulo (zero) e o resultado da última operação (TOS). Funções aritméticas supõem que todos os **valores são inteiros representados em complemento de dois com 16 bits**. Além disso, '**testes' lógicos levam todos os bits da saída a nível alto quando o teste for positivo e baixo caso contrário**.

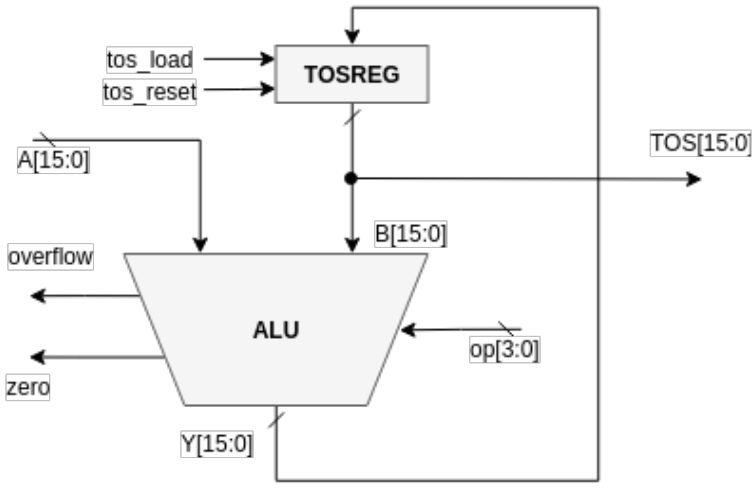


Figura 04 – ULA e *buffer* do topo da pilha. Fonte: o autor (2018).

op	Saída (*Y*)
0000	A
0001	A + B
0010	A - B
0011	A – B, testa zero
0100	A – B, testa negativo
0101	A – B, testa positivo
0110	A, testa zero
0111	A, testa negativo

1000	A, testa positivo
1001	A ^ B
1010	A | B
1011	A & B
1100	A << 1
1101	A >> 1, mantém MSB
1110	A << 8
1111	A >> 8

Tabela 4 – Operações da ULA.

Contador de Programa e Registrador de Instrução

São necessários outros dois elementos de memória persistente entre estados da máquina: o contador de programa (PC) que requer ainda um incrementador independente da ULA; e o registrador de instrução (CIR) que é utilizado para armazenar dados lidos da memória em instruções que tomam mais de um ciclo e é especialmente importante ao chamar funções, pois o endereço da sub-rotina está contido na própria instrução e precisa ser preservado entre estados.

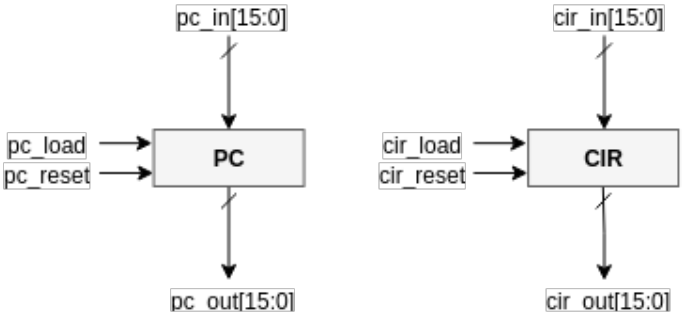


Figura 05 – Registradores PC e CIR. Fonte: o autor (2018).

Seletores de entrada

Haja vista que os componentes descritos necessitam trocar dados entre si, a entrada externa de cada um deles deve ser selecionada por sinais vindos do bloco de controle. Outras implementações normalmente empregariam um ou mais barramentos utilizando lógica *tri-state*, mas a

solução dos multiplexadores não somente permite atribuições em paralelo, como também facilita a síntese do projeto na FPGA.

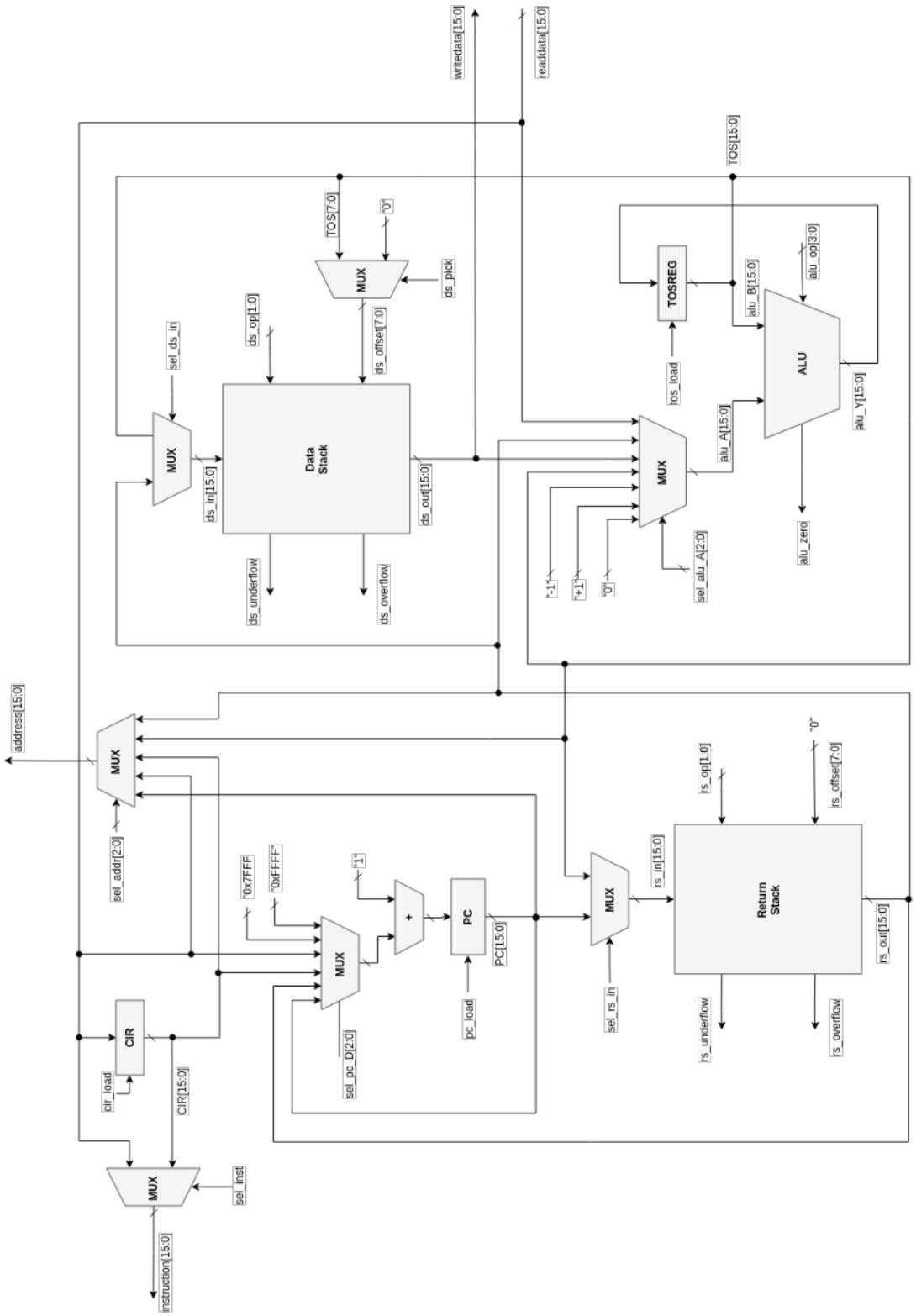


Figura 06 – Bloco Operativo. Seu comportamento é dado pela composição dos componentes descritos anteriormente e os sinais de controle. Fonte: O autor (2018).

2.5 Projeto do Bloco de Controle

O projeto do Bloco de Controle deve levar em conta os sinais indo e vindo do Bloco Operativo, além das entradas e saídas de controle da interface. São sinais de controle:

- `instruction`: Indica próxima instrução a ser executada;
- `alu_zero`: Sinaliza que a saída da ULA é zero;
- `ds_underflow`: Sinaliza *underflow* na pilha de dados;
- `ds_overflow`: Sinaliza *overflow* na pilha de dados;
- `rs_underflow`: Sinaliza *underflow* na pilha de retorno;
- `rs_overflow`: Sinaliza *overflow* na pilha de retorno;
- `ds_op`: Controla a operação da pilha de dados. Vide Tabela 3;
- `sel_ds_in`: Seleciona a entrada da pilha de dados;
  - **0**: TOS, **1**: RS;
- `ds_pick`: Seleciona offset de endereço da pilha de dados;
  - **0**: “0”, **1**: oito bits menos significativos de TOS.
- `rs_op`: Controla a operação da pilha de retorno. Vide Tabela 3;
- `sel_rs_in`: Seleciona a entrada da pilha de retorno;
  - **0**: PC, **1**: TOS;
- `alu_op`: Controla a operação da ULA. Vide Tabela 4;
- `sel_alu_A`: Seleciona a entrada de operando da ULA;
  - **000**: “0”, **001**: “+1”, **010**: “-1”, **011**: TOS, **100**: DS, **101**: RS, **110**: MEM, **111**: indefinido;
- `tos_load`: Controla a carga do registrador TOSREG;
- `pc_load`: Controla a carga do registrador PC;
- `sel_pc_D`: Seleciona a entrada do registrador PC;
  - **000**: “0x0000”, **001**:”0x8000”, **010**: PC + 1, **011**: CIR + 1, **100**: MEM + 1, **101**: RS + 1, outros: indefinido;
- `sel_addr`: Seleciona a saída para o barramento de endereço;
  - **000**: PC, **001**: TOS, **010**: MEM, **011**: CIR, **100**: RS, outros: indefinido;

- `read`: Controla operações de leitura da memória externa;
- `write`: Controla operações de escrita da memória externa;
- `cir_load`: Controla a carga do registrador CIR;
- `sel_inst`: Seleciona o valor tomado como próxima instrução;
  - **0**: MEM, **1**: CIR.

Considerando a estrutura do bloco operativo, as decisões de projeto tomadas e os nomes de sinais estabelecidos, novamente captura-se o funcionamento do sistema com uma máquina de estados. Agora, entretanto, toma-se o cuidado para otimizar a performance do processador reduzindo ao máximo o número de ciclos ao introduzir **saídas de Mealy** (ou seja, que dependem não somente do estado atual mas também de uma ou mais entradas do Bloco de Controle) e operações paralelas de acesso a memória e outras atribuições para criar **segmentação de instruções (*pipelining*)**.

Todos os sinais são considerados ativos em nível alto (com exceção da entrada externa *reset_n)* e quanto ao acesso à memória, supõe-se que basta direcionar o endereço desejado à saída *address* por um período inteiro (enquanto mantém ativo o comando *read)* e os dados lidos estarão disponíveis em *readdata* pela duração do próximo ciclo; no caso de escrita o dado em questão deve ser mantido na saída *writedata* até a próxima subida do *clock* (com *write* em nível alto). (INTEL)

Cada estado inicial de instrução supõe que no início do período de relógio estarão disponíveis nas pilhas os seus respectivos valores do topo, na entrada de dados lidos da memória a próxima instrução a ser executada, no registrador PC o endereço da seguinte e no registrador CIR a instrução atual; portanto deve preparar o sistema para que o próximo possa partir das mesmas suposições.

A Tabela 5, na página a seguir, expõe a maneira como isso é realizado, possibilitando o *pipelining* das instruções uma após a outra e alcançando os valores de ciclos por instrução dados na Tabela 2.

---- / Estado atual	Saídas / ----	ds_op	sel_ds_in	ds_pick	rs_op	sel_ rs_in	alu_op	sel_alu_A	tos_load	pc_load	sel_pc_D	sel_addr	read / write	cir_load	sel_inst	Próximo estado
RESET		RESET	-	0	RESET	-	A	0	1	1	*h (mode)*	-	NOP	-	-	FETCH
FETCH		NOP	-	0	NOP	-	A	TOS	-	1	PC + 1	PC	READ	-	-	DECODE
DECODE / NOP		NOP	-	0	NOP	-	A	TOS	-	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
CALL		NOP	-	0	PUSH	PC	A	TOS	-	1	CIR + 1	CIR	READ	-	-	DECODE
@_0		NOP	-	0	NOP	-	A	TOS	-	0	-	TOS	READ	1	-	@_1
@_1		NOP	-	0	NOP	-	A	MEM	1	*g (inst)*	PC + 1	PC	READ	0	CIR	*f (inst, zero)*
!_0		POP	-	0	NOP	-	A	TOS	-	0	-	TOS	WRITE	1	-	!_1
!_1		POP	-	0	NOP	-	A	DS	1	*g (inst)*	PC + 1	PC	READ	0	CIR	*f (inst, zero)*
IF_FALSE		POP	-	0	NOP	-	A	DS	1	1	MEM + 1	MEM	READ	-	-	DECODE
IF_TRUE		POP	-	0	NOP	-	A	DS	1	1	PC + 1	PC	READ	-	-	DECODE
BRANCH		NOP	-	0	NOP	-	A	TOS	-	1	MEM + 1	MEM	READ	-	-	DECODE
EXIT		NOP	-	0	POP	-	A	TOS	-	1	RS + 1	RS	READ	-	-	DECODE
DROP		POP	-	0	NOP	-	A	DS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
LIT		PUSH	TOS	0	NOP	-	A	MEM	1	1	PC + 1	PC	READ	-	-	DECODE
PICK_0		NOP	-	TOS	NOP	-	A	TOS	-	0	-	-	NOP	1	-	PICK_1
PICK_1		NOP	-	0	NOP	-	A	DS	1	*g (inst)*	PC + 1	PC	READ	0	CIR	*f (inst, zero)*
>R		POP	-	0	PUSH	TOS	A	DS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
R>		PUSH	TOS	0	POP	-	A	RS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
NOT		NOP	-	0	NOP	-	A - TOS	-1	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
OR		POP	-	0	NOP	-	A | TOS	DS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
AND		POP	-	0	NOP	-	A & TOS	DS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
XOR		POP	-	0	NOP	-	A ^ TOS	DS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
+		POP	-	0	NOP	-	A + TOS	DS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
-		POP	-	0	NOP	-	A - TOS	DS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
1+		NOP	-	0	NOP	-	A + TOS	1	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
1-		NOP	-	0	NOP	-	A + TOS	-1	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
=		POP	-	0	NOP	-	A = TOS	DS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
<		POP	-	0	NOP	-	A < TOS	DS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
>		POP	-	0	NOP	-	A > TOS	DS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
0=		NOP	-	0	NOP	-	A = 0	TOS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
0<		NOP	-	0	NOP	-	A < 0	TOS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
0>		NOP	-	0	NOP	-	A > 0	TOS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
2*		NOP	-	0	NOP	-	A << 1	TOS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
2/		NOP	-	0	NOP	-	A >> 1	TOS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
DUP		PUSH	TOS	0	NOP	-	A	TOS	-	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
SWAP_0		POP	-	0	PUSH	TOS	A	DS	1	0	-	-	NOP	1	-	SWAP_1
SWAP_1		PUSH	RS	0	POP	-	A	TOS	-	*g (inst)*	PC + 1	PC	READ	0	CIR	*f (inst, zero)*
SBL		NOP	-	0	NOP	-	A << 8	TOS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*
SBR		NOP	-	0	NOP	-	A >> 8	TOS	1	*g (inst)*	PC + 1	PC	READ	1	rdata	*f (inst, zero)*

Tabela 5 – FSMD com *Pipelining*. Estão destacadas as saídas de Mealy.

A Tabela 5 representa as saídas para cada estado do autômato do Bloco de Controle e transição para o próximo estado. Estão destacadas as saídas cujo valor é dado por uma função combinacional que depende de um ou mais sinais de entrada (além do estado atual). **É necessário atentar-se para que tais funções não tomem como argumento sinais combinacionais que dependam da própria saída da função**. São elas:

- *h (mode)*: Relacionada com o sinal *sel_pc_D*, deve escolher o valor "0x0000" quando *mode* for zero e "0x8000" quando for um.
- *f (instruction, alu_zero)*: Realiza a decodificação de *instruction,* seu resultado deve ser o código para o estado correspondente. Se a instrução for "IF", seleciona o estado "IF_FALSE" ou "IF_TRUE" dependendo do sinal *alu_zero*.
- *g (instruction)*: Controla a carga do contador de programa, deve tomar o valor do bit menos significativo da instrução, que indica quando há chamada de sub-rotina.

Uma **outra entrada que sempre interfere no próximo estado é o sinal de *reset***, que quando em nível lógico baixo faz com que o estado tomado no próximo ciclo (*reset* síncrono) pela máquina seja, invariavelmente, o estado "RESET". Essa propriedade não foi representada na tabela.

Por fim, obtém-se um processador multiciclo eficiente descrito com um *hardware* relativamente simples sem precisar recorrer a algumas das técnicas empregadas em outros processadores para tal fim e que podem vir a limitar a velocidade do sistema (especificar uma memória que opere a uma taxa duas vezes mais rápida que o *clock*, por exemplo, como é feito na arquitetura didática do MIPS monociclo*)*.

2.6 Projeto da Unidade de Processamento

Com os projetos do Bloco Operativo e do Bloco de Controle prontos, expõe-se o sistema digital completo: a Unidade de Processamento Simples com Arquitetura Baseada em Pilhas. O circuito final, ilustrado pela Figura 07, apenas integra ambos os blocos mencionados respeitando a especificação da interface do sistema.

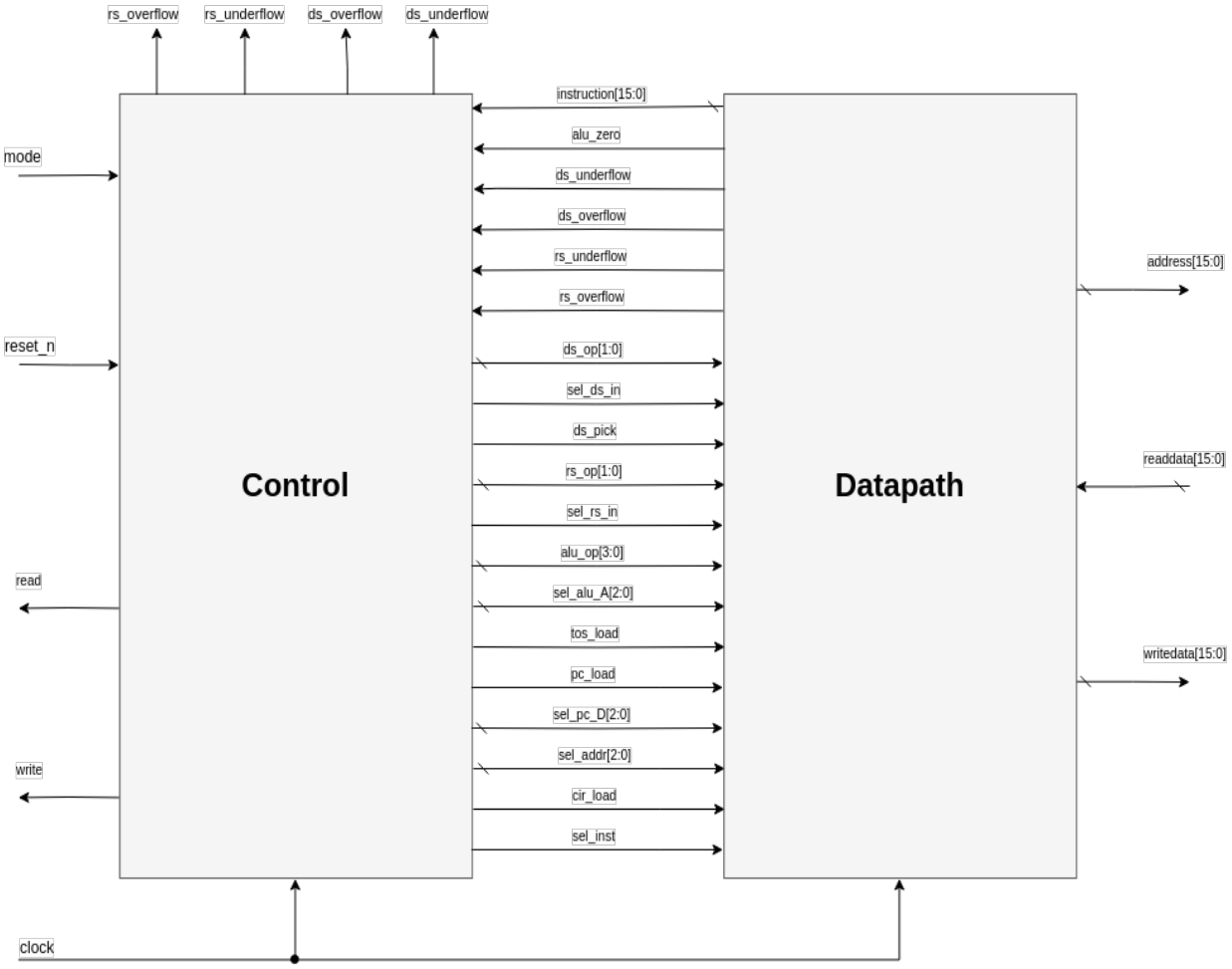


Figura 07 – Unidade de Processamento. Integração do Bloco Operativo (*Datapath*) com o Bloco de Controle (*Control*). Fonte: o autor (2018).

3 Desenvolvimento

Levando em conta que o projeto visa implementação em FPGA (mais especificamente, em um chip **EP2C35F672C6** da família **Cyclone II**), serão abstraídos detalhes de características eletrônicas do circuito; possíveis otimizações a níveis de portas lógicas e transistores; e outros tais aspectos da síntese. Assim, o desenvolvimento do sistema e suas partes consistirá na descrição de seu *hardware* em **VHDL** no *software* **Quartus II** (Altera / Intel).

3.1 Desenvolvimento do Bloco Operativo

Seguindo o esquemático da Figura 06, o Bloco Operativo é gerado pela composição de alguns multiplexadores, registradores, um incrementador, uma ULA e os dois *stacks*. A descrição do *hardware* é totalmente estrutural pois seu comportamento provém dos seus componentes.

```
     ------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;     -- type conversion
use ieee.math_real.all;       -- log2 & ceil


entity S4PU_Datapath is       -- S4PU Operative Unit
       generic (ARCH: positive := 16);
       port (
              -- CLOCK --
              clock: in std_logic;

              -- EXTERNAL INPUTS --
              readdata: in std_logic_vector(ARCH-1 downto 0);

              -- COMMANDS --
              ds_op: in std_logic_vector(1 downto 0);
              sel_ds_in: in std_logic;
              ds_pick: in std_logic;
              rs_op: in std_logic_vector(1 downto 0);
              sel_rs_in: in std_logic;
              alu_op: in std_logic_vector(3 downto 0);
              sel_alu_A: in std_logic_vector(2 downto 0);
              tos_load: in std_logic;
              pc_load: in std_logic;
              sel_pc_D: in std_logic_vector(2 downto 0);
              sel_addr: in std_logic_vector(2 downto 0);
```

```
                cir_load: in std_logic;
                sel_inst: in std_logic;

                -- STATUS --
                instruction: out std_logic_vector(ARCH-1 downto 0);
                alu_zero: out std_logic;
                rs_overflow, rs_underflow: out std_logic;
                ds_overflow, ds_underflow: out std_logic;

                -- EXTERNAL OUTPUTS --
                address: out std_logic_vector(ARCH-1 downto 0);
                writedata: out std_logic_vector(ARCH-1 downto 0)
        );
end entity;


architecture operative_v0 of S4PU_Datapath is       -- default
        -- COMPONENTS --
        component Multiplexer is      -- Mux
                generic (
                        WIDTH: positive := 1;
                        FAN_IN: positive := 2 -- no. of WIDTH size inputs
                );
                port (
                        sel: in std_logic_vector(natural(ceil(log2(real(FAN_IN))))-1 downto 0);
                        mux_in: in std_logic_vector((WIDTH*FAN_IN)-1 downto 0);
                        mux_out: out std_logic_vector(WIDTH-1 downto 0)
                );
        end component;

        component Reg is      -- Register with asynchronous reset
                generic (WIDTH: positive := 8);
                port (
                        -- CLOCK & CONTROL --
                        clock: in std_logic;
                        reset, enable: in std_logic;  -- asynchronous reset

                        -- DATA --
                        D: in std_logic_vector(WIDTH-1 downto 0);
                        Q: out std_logic_vector(WIDTH-1 downto 0)
                );
        end component;

        component ALU_16 is   -- 16-operation ALU
                generic (WORD: positive := 16);      -- should be greater than 8
                port (
                        -- CONTROL --
                        op: in std_logic_vector(3 downto 0);

                        -- DATA --
                        A, B: in std_logic_vector(WORD-1 downto 0);
                        Y: out std_logic_vector(WORD-1 downto 0);

                        -- SIGNALS --
                        zero, overflow: out std_logic
                );
        end component;

        component LIFO_Stack is       -- Push-down LIFO Stack
                generic (
                        WORD: positive := 16;
                        ADDR: positive := 8   -- address vector size
                );
                port (
                        -- CLOCK --
                        clock: in std_logic;

                        -- CONTROL --
                        op: in std_logic_vector(1 downto 0);
                        offset: in std_logic_vector(ADDR-1 downto 0);

                        -- DATA --
                        stack_in: in std_logic_vector(WORD-1 downto 0);
                        stack_out: out std_logic_vector(WORD-1 downto 0);
```

```
                -- STACK ERRORS --
                overflow, underflow: out std_logic
        );
    end component;


    -- CONSTANTS --
constant ADDR: positive := 8;
constant UNDEFINED: std_logic_vector(ARCH-1 downto 0) := (others => '-');


    -- SIGNALS --
    signal alu_A_sig, alu_Y_sig, tos_sig,
           ds_in_sig, ds_out_sig,
           rs_in_sig, rs_out_sig,
           pc_D_sig, pc_Q_sig, final_pc_D_sig,
           cir_sig: std_logic_vector(ARCH-1 downto 0);

    signal ds_offset_sig: std_logic_vector(ADDR-1 downto 0);

    signal alu_overflow_sig: std_logic;  -- @note unused

    signal sel_ds_offset_sig, sel_ds_in_sig,
           sel_rs_in_sig, sel_inst_sig: std_logic_vector(0 downto 0);

    signal mux_in_alu_A: std_logic_vector((ARCH*8)-1 downto 0);
    signal mux_in_ds_offset: std_logic_vector((ADDR*2)-1 downto 0);
    signal mux_in_ds_in, mux_in_rs_in, mux_in_inst: std_logic_vector((ARCH*2)-1 downto 0);
    signal mux_in_pc_D, mux_in_addr: std_logic_vector((ARCH*8)-1 downto 0);


    -- BEHAVIOUR --
    begin
           ALU: ALU_16
                  generic map (WORD => ARCH)
                  port map (
                         op => alu_op,

                         A => alu_A_sig,
                         B => tos_sig,
                         Y => alu_Y_sig,

                         zero => alu_zero,
                         overflow => alu_overflow_sig
                  );

           MUX_ALU_A: Multiplexer
                  generic map (
                         WIDTH => ARCH,
                         FAN_IN => 8
                  )
                  port map (
                         sel => sel_alu_A,
                         mux_in => mux_in_alu_A,
                         mux_out => alu_A_sig
                  );
           mux_in_alu_A <= (
                  UNDEFINED &                                  -- 111
                  readdata &                                   -- 110
                  rs_out_sig &                                 -- 101
                  ds_out_sig &                                 -- 100
                  tos_sig &                                    -- 011
                  std_logic_vector(to_signed(-1, ARCH)) &      -- 010
                  std_logic_vector(to_signed(1, ARCH)) &       -- 001
                  std_logic_vector(to_signed(0, ARCH))         -- 000
           );

           TOS_REG: Reg
                  generic map (WIDTH => ARCH)
                  port map (
                         clock => clock,
                         reset => '0',
                         enable => tos_load,

                         D => alu_Y_sig,
```

```
                Q => tos_sig
        );

DATA_STACK: LIFO_Stack
        generic map (
                WORD => ARCH,
                ADDR => ADDR
        )
        port map (
                clock => clock,

                op => ds_op,
                offset => ds_offset_sig,

                stack_in => ds_in_sig,
                stack_out => ds_out_sig,

                overflow => ds_overflow,
                underflow => ds_underflow
        );

writedata <= ds_out_sig;

MUX_DS_OFFSET: Multiplexer
        generic map (
                WIDTH => ADDR,
                FAN_IN => 2
        )
        port map (
                sel => sel_ds_offset_sig,
                mux_in => mux_in_ds_offset,
                mux_out => ds_offset_sig
        );
sel_ds_offset_sig(0) <= ds_pick;
mux_in_ds_offset <= (
        tos_sig(ADDR-1 downto 0) &           -- 1
        std_logic_vector(to_signed(0, ADDR)) -- 0
);

MUX_DS_IN: Multiplexer
        generic map (
                WIDTH => ARCH,
                FAN_IN => 2
        )
        port map (
                sel => sel_ds_in_sig,
                mux_in => mux_in_ds_in,
                mux_out => ds_in_sig
        );
sel_ds_in_sig(0) <= sel_ds_in;
mux_in_ds_in <= (
        rs_out_sig &   -- 1
        tos_sig        -- 0
);

RETURN_STACK: LIFO_Stack
        generic map (
                WORD => ARCH,
                ADDR => ADDR
        )
        port map (
                clock => clock,

                op => rs_op,
                offset => std_logic_vector(to_unsigned(0, ADDR)),

                stack_in => rs_in_sig,
                stack_out => rs_out_sig,

                overflow => rs_overflow,
                underflow => rs_underflow
        );

MUX_RS_IN: Multiplexer
        generic map (
```

```
                WIDTH => ARCH,
                FAN_IN => 2
        )
        port map (
                sel => sel_rs_in_sig,
                mux_in => mux_in_rs_in,
                mux_out => rs_in_sig
        );
sel_rs_in_sig(0) <= sel_rs_in;
mux_in_rs_in <= (
        tos_sig &       -- 1
        pc_Q_sig        -- 0
);

PC_REG: Reg
        generic map (WIDTH => ARCH)
        port map (
                clock => clock,
                reset => '0',
                enable => pc_load,

                D => final_pc_D_sig,
                Q => pc_Q_sig
        );
final_pc_D_sig <= std_logic_vector(unsigned(pc_D_sig) + 1);

MUX_PC_D: Multiplexer
        generic map (
                WIDTH => ARCH,
                FAN_IN => 8
        )
        port map (
                sel => sel_pc_D,
                mux_in => mux_in_pc_D,
                mux_out => pc_D_sig
        );
mux_in_pc_D <= (
        UNDEFINED &     -- 111
        UNDEFINED &     -- 110
        rs_out_sig &    -- 101
        readdata &      -- 100
        cir_sig &       -- 011
        pc_Q_sig &      -- 010
        x"7FFF" &       -- 001
        x"FFFF"         -- 000
);

CIR_REG: Reg
        generic map (WIDTH => ARCH)
        port map (
                clock => clock,
                reset => '0',
                enable => cir_load,

                D => readdata,
                Q => cir_sig
        );

MUX_INST: Multiplexer
        generic map (
                WIDTH => ARCH,
                FAN_IN => 2
        )
        port map (
                sel => sel_inst_sig,
                mux_in => mux_in_inst,
                mux_out => instruction
        );
sel_inst_sig(0) <= sel_inst;
mux_in_inst <= (
        cir_sig &       -- 1
        readdata        -- 0
);

MUX_ADDR: Multiplexer
```

```
                generic map (
                        WIDTH => ARCH,
                        FAN_IN => 8
                )
                port map (
                        sel => sel_addr,
                        mux_in => mux_in_addr,
                        mux_out => address
                );
        mux_in_addr <= (
                UNDEFINED &             -- 111
                UNDEFINED &             -- 110
                UNDEFINED &             -- 101
                rs_out_sig &            -- 100
                cir_sig &               -- 011
                readdata &              -- 010
                tos_sig &               -- 001
                pc_Q_sig                -- 000
        );

end architecture;
      ---------------------------------------------------------------------
```

- Temporização
  - *Clock to output*: 20,592 ns.
- Área
  - Funções combinacionais: 501 elementos.
  - Registradores: 64 bits.
  - Blocos de memória: 8.192 bits.

Como pode-se observar pelo esquemático RTL gerado pela ferramenta de síntese, na Figura 08, o Bloco Operativo segue à risca o projeto realizado no Capítulo anterior.

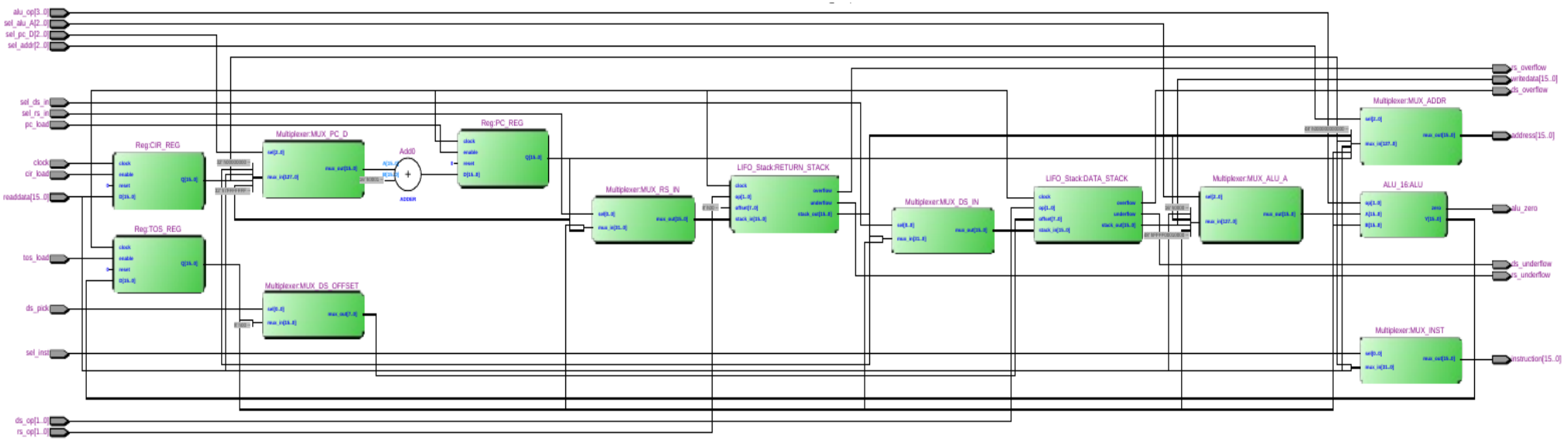


Figura 08 - Bloco Operativo Sintetizado. Fonte: o autor (2018).

3.1.1 - Multiplexador

Visando criar um multiplexador genérico, utilizou-se uma interface com um único vetor formado pela concatenação (externa) de suas entradas, onde a saída é igual a um intervalo do vetor de entrada tomado a partir de uma posição que depende do valor no sinal de seleção.

```
      ----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;     -- type conversion
use ieee.math_real.all;       -- log2 & ceil


entity Multiplexer is -- Mux
       generic (
              WIDTH: positive := 1;
              FAN_IN: positive := 2 -- no. of WIDTH size inputs
       );
       port (
              sel: in std_logic_vector(natural(ceil(log2(real(FAN_IN))))-1 downto 0);
              mux_in: in std_logic_vector((WIDTH*FAN_IN)-1 downto 0);
              mux_out: out std_logic_vector(WIDTH-1 downto 0)
       );
       begin
              assert (2**sel'length = FAN_IN)
                     report "Invalid mux fan in: should be a power of two."
                     severity ERROR;
end entity;


architecture vectorial of Multiplexer is     -- default
       -- BEHAVIOUR --
       begin
              out_gen: for i in mux_out'range generate
                     mux_out(i) <= mux_in(to_integer(unsigned(sel))*WIDTH + i);
              end generate;
end architecture;
      ----------------------------------------------------------------------
```

- Temporização
  - Atraso de propagação: 5,271 ns.
- Área
  - Funções combinacionais: 1 elemento (para *mux* 2x1).
  - Registradores e blocos de memória: 0 bits.

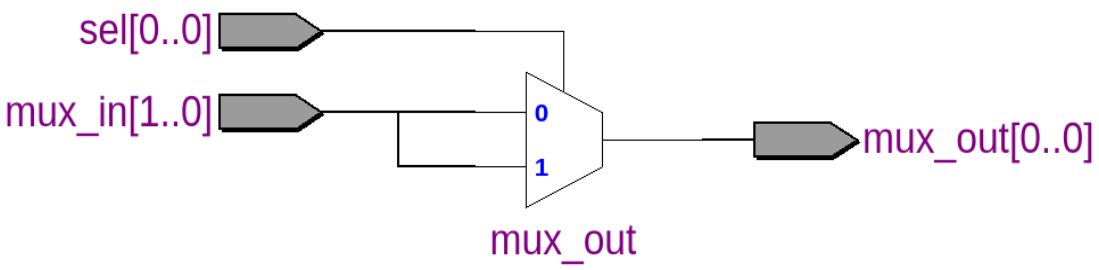


Figura 09 - Multiplexador Sintetizado. Fonte: o autor (2018).

3.1.2 – Registrador

Implementou-se um registrador genérico de carga paralela completo com sinal de carga ativo em nível alto, *reset* assíncrono ativo em nível alto e carregamento de dados na borda ascendente do relógio.

```
      ----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;


entity Reg is  -- Register with asynchronous reset
       generic (WIDTH: positive := 8);
       port (
              -- CLOCK & CONTROL --
              clock: in std_logic;
              reset, enable: in std_logic;  -- asynchronous reset

              -- DATA --
              D: in std_logic_vector(WIDTH-1 downto 0);
              Q: out std_logic_vector(WIDTH-1 downto 0)
       );
end entity;


architecture canonical of Reg is     -- default
       -- INTERNAL STATE --
       subtype InternalState is std_logic_vector(WIDTH-1 downto 0);
       signal curr_state, next_state: InternalState;


       -- BEHAVIOUR --
       begin
              -- next-state logic : Combinatorial
              next_state <= D when enable = '1' else curr_state;

              -- memory element : Sequential
              ME: process (clock, reset) is
                     begin
                            if (reset = '1') then
                                   curr_state <= (others => '0');
                            elsif (rising_edge(clock)) then
                                   curr_state <= next_state;
                            end if;
              end process;

              -- output logic
              Q <= curr_state;
end architecture;
      ----------------------------------------------------------------------
```

- Temporização
  - *Setup*: 3,279 ns.
  - *Clock to output*: 8,673 ns.
- Área
  - Funções combinacionais: 0 elementos.
  - Registradores: 8 bits (para registrador de 8 bits).
  - Blocos de memória: 0 bits.

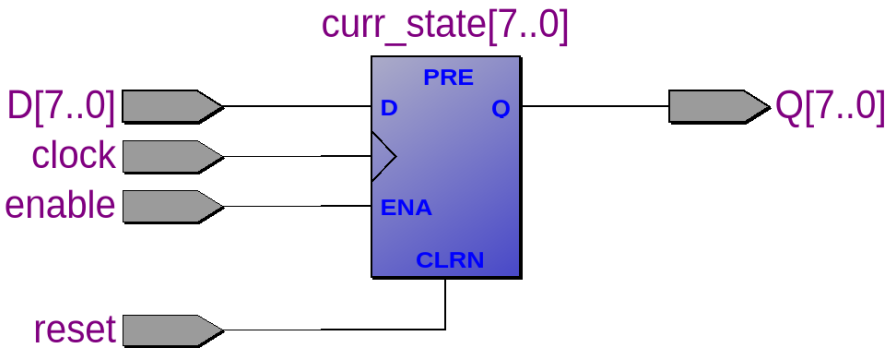


Figura 10 - Registrador Sintetizado. Fonte: o autor (2018).

3.1.3 – Unidade Lógica e Aritmética

A ULA projetada realiza todas as suas 16 operações em paralelo e então escolhe a saída desejada com o sinal de operação (respeitando a especificação de códigos da Tabela 4). O *overflow* aritmético pode ocorrer em operações de soma, subtração e deslocamento aritmético para a esquerda.

```
      ----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;     -- type conversion


entity ALU_16 is      -- 16-operation ALU
        generic (WORD: positive := 16);      -- should be greater than 8
        port (
                -- CONTROL --
                op: in std_logic_vector(3 downto 0);

                -- DATA --
                A, B: in std_logic_vector(WORD-1 downto 0);
                Y: out std_logic_vector(WORD-1 downto 0);

                -- SIGNALS --
                zero, overflow: out std_logic
        );
        begin
                assert (WORD > 8)
                        report "Invalid word size: Should be >=8."
                        severity FAILURE;
end entity;


architecture parallel of ALU_16 is   -- default
        -- SIGNALS --
        signal result,
                nop, add, sub,
                op_xor, op_or, op_and,
        op_sla, op_sra, op_sll, op_srl: std_logic_vector(WORD downto 0);

        signal comp_equal, comp_less, comp_greater,
                A_zero, A_negative, A_positive: std_logic;


        -- CONSTANT OPCODES --
```

```
constant ALU_OP_NOP: std_logic_vector(3 downto 0) :=       "0000";
constant ALU_OP_ADD: std_logic_vector(3 downto 0) :=       "0001";
constant ALU_OP_SUB: std_logic_vector(3 downto 0) :=       "0010";
constant ALU_OP_CET: std_logic_vector(3 downto 0) :=       "0011";
constant ALU_OP_CLT: std_logic_vector(3 downto 0) :=       "0100";
constant ALU_OP_CGT: std_logic_vector(3 downto 0) :=       "0101";
constant ALU_OP_ZER: std_logic_vector(3 downto 0) :=       "0110";
constant ALU_OP_NEG: std_logic_vector(3 downto 0) :=       "0111";
constant ALU_OP_POS: std_logic_vector(3 downto 0) :=       "1000";
constant ALU_OP_XOR: std_logic_vector(3 downto 0) :=       "1001";
constant ALU_OP_OR: std_logic_vector(3 downto 0) :=        "1010";
constant ALU_OP_AND: std_logic_vector(3 downto 0) :=       "1011";
constant ALU_OP_SAL: std_logic_vector(3 downto 0) :=       "1100";
constant ALU_OP_SAR: std_logic_vector(3 downto 0) :=       "1101";
constant ALU_OP_SBL: std_logic_vector(3 downto 0) :=       "1110";
constant ALU_OP_SBR: std_logic_vector(3 downto 0) :=       "1111";


-- BEHAVIOUR --
begin
	-- A
	nop <= '0' & A;

	-- A + B
	add <= std_logic_vector(signed(A(A'left) & A) + signed(B(B'left) & B));

	-- A - B
	sub <= std_logic_vector(signed(A(A'left) & A) - signed(B(B'left) & B));

	-- A = B
	comp_equal <= '1' when signed(sub) = 0 else '0'; – <-> A-B=0

	-- A < B
	comp_less <= sub(sub'left);   -- A < B <-> A-B < 0

	-- A > B
	comp_greater <= comp_equal nor comp_less;    -- <-> ¬(A=B) & ¬(A<B)

	-- A = 0
	A_zero <= '1' when signed(A) = 0 else '0';

	-- A < 0
	A_negative <= A(A'left);

	-- A > 0
	A_positive <= A_zero nor A_negative;

	-- A xor B
	op_xor <= '0' & (A xor B);

	-- A or B
	op_or <= '0' & (A or B);

	-- A and B
	op_and <= '0' & (A and B);

	-- arithmetic shift A 1 bit left
	op_sla <= A & '0';

	-- arithmetic shift A 1 bit right
	op_sra <= A(A'left) & A(A'left) & A(A'left downto A'right+1);

	-- logical shift A 8 bits left
	op_sll(7 downto 0) <= (others => '0');
	op_sll(WORD downto 8) <= A(A'left-7 downto A'right);

	-- logical shift A 8 bits right
	op_srl(WORD downto WORD-8) <= (others => '0');
	op_srl(WORD-9 downto 0) <= A(A'left downto A'right+8);


	-- temporary result
	with op select result <=
		add                                 when ALU_OP_ADD,
		sub                                 when ALU_OP_SUB,
```

```
                (others => comp_equal)       when ALU_OP_CET,
                (others => comp_less)        when ALU_OP_CLT,
                (others => comp_greater)     when ALU_OP_CGT,
                (others => A_zero)           when ALU_OP_ZER,
                (others => A_negative)       when ALU_OP_NEG,
                (others => A_positive)       when ALU_OP_POS,
                op_xor                       when ALU_OP_XOR,
                op_or                        when ALU_OP_OR,
                op_and                       when ALU_OP_AND,
                op_sla                       when ALU_OP_SAL,
                op_sra                       when ALU_OP_SAR,
                op_sll                       when ALU_OP_SBL,
                op_srl                       when ALU_OP_SBR,
                nop                          when others; -- ALU_OP_NOP

        -- overflow detect
        with op select overflow <=
                result(result'left) xor result(result'left-1)
                        when ALU_OP_ADD|ALU_OP_SUB|ALU_OP_SAL, -- overflow
                '0'     when others;

        -- zero detect
        zero <= '1' when signed(result(WORD-1 downto 0)) = 0 else '0';

        -- ALU out
        Y <= result(WORD-1 downto 0);

end architecture;
-------------------------------------------------------------------------
```

- Temporização
  - Atraso de propagação: 20,295 ns.
- Área
  - Funções combinacionais: 208 elementos.
  - Registradores e blocos de memória: 0 bits.

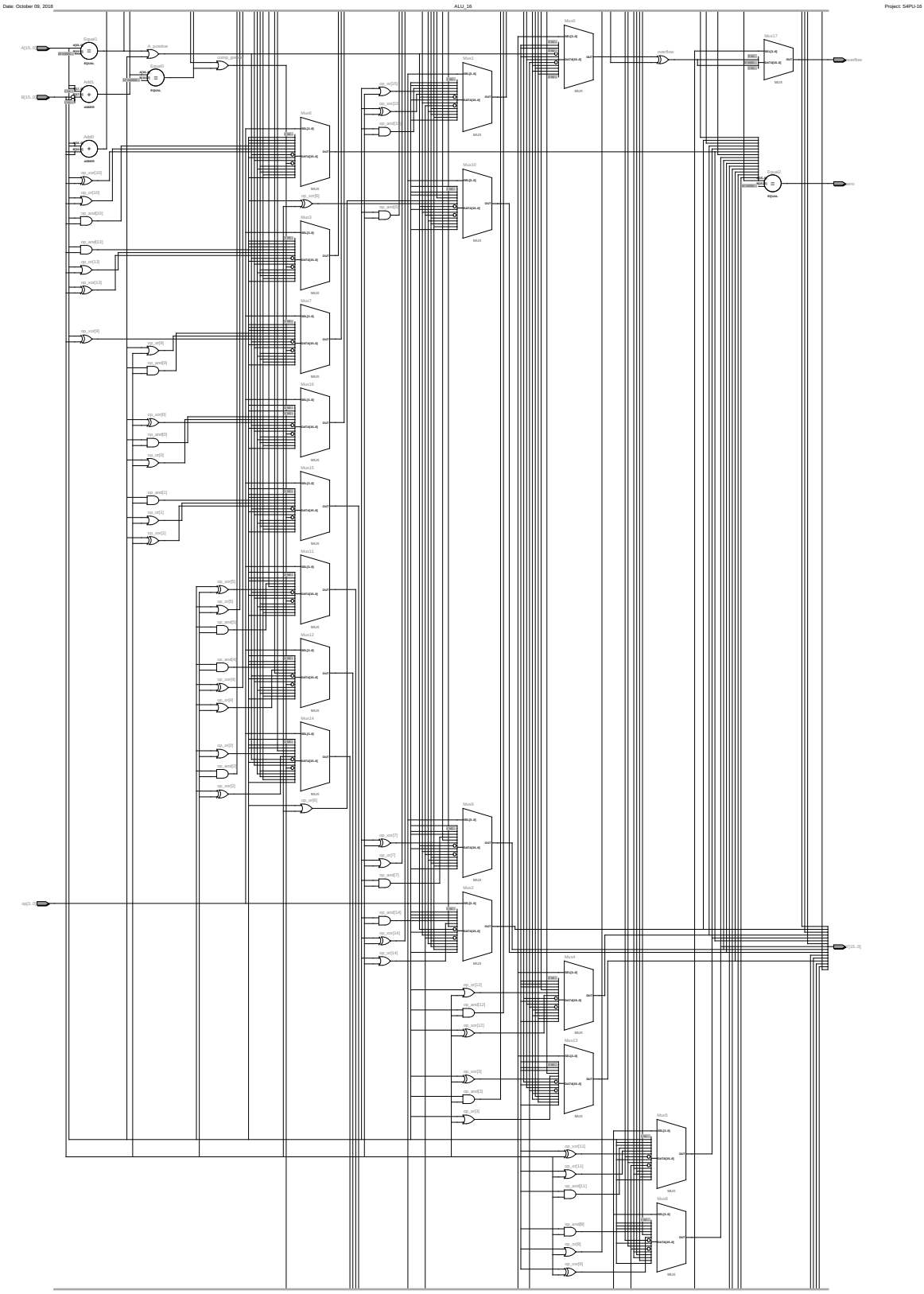


Figura 11 - ULA Sintetizada. Fonte: o autor (2018).

3.1.4 – Pilha LIFO

O *stack* opera dados de forma síncrona, disponibilizando no ciclo seguinte o topo da pilha após a operação atual - seguindo a codificação apresentada na Tabela 3 e considerando a entrada de deslocamento (*offset*) de endereços a partir do topo; enquanto as **saídas sinalizadoras de erro são assíncronas**, ou seja, vão para nível lógico alto apenas durante o ciclo da própria operação causadora de *underflow/ overflow*. Um detalhe da implementação que não se expõe na interface é que a pilha cresce a partir dos endereços mais altos da memória.

Por sua simplicidade, a descrição deste componente é principalmente estrutural: utiliza um registrador para armazenar o endereço de topo do *stack;* uma memória RAM *single-port* (gerada pelo Quartus) com 256 palavras de 16 bits endereçadas consecutivamente para os dados da pilha; e dois multiplexadores para selecionar qual valor será carregado no registrador de endereço e qual será utilizado para acessar a memória neste ciclo.

Outra técnica de implementação abandonada neste projeto envolveria agregar as duas pilhas em um único componente, utilizando uma **memória *dual-port* compartilhada entre *stacks*** de dados e de endereços de retorno, o primeiro iniciaria nos endereços mais elevados enquanto o segundo cresceria de baixo para cima; assim, havendo um registrador de endereço para cada uma das pilhas seria possível obter um **aproveitamento mais flexível da memória** livre entre os topos dos *stacks.* (KOOPMAN, 1989)

```
      ----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;     -- type conversion
use ieee.math_real.all;       -- log2 & ceil


entity LIFO_Stack is  -- Push-down LIFO Stack with from-the-top offset
       generic (
              WORD: positive := 16;
              ADDR: positive := 8          -- address vector size
       );
       port (
              -- CLOCK --
              clock: in std_logic;

              -- CONTROL --
```

```
		op: in std_logic_vector(1 downto 0);
		offset: in std_logic_vector(ADDR-1 downto 0);

		-- DATA --
		stack_in: in std_logic_vector(WORD-1 downto 0);
		stack_out: out std_logic_vector(WORD-1 downto 0);

		-- STACK ERRORS --
		overflow, underflow: out std_logic
	);
end entity;


architecture onchip_quartus_ram of LIFO_Stack is    -- default
	-- COMPONENTS --
	component onchip_ram IS       -- Altera Quartus wizard generated 1-port RAM
		PORT
		(
			address : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
			clock		: IN STD_LOGIC  := '1';
			data		: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
			wren		: IN STD_LOGIC ;
			q		: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
		);
	END component;

	component Reg is      -- Register with asynchronous reset
		generic (WIDTH: positive := 8);
		port (
			-- CLOCK & CONTROL --
			clock: in std_logic;
			reset, enable: in std_logic;  -- asynchronous reset

			-- DATA --
			D: in std_logic_vector(WIDTH-1 downto 0);
			Q: out std_logic_vector(WIDTH-1 downto 0)
		);
	end component;

	component Multiplexer is      -- Mux
		generic (
			WIDTH: positive := 1;
			FAN_IN: positive := 2 -- no. of WIDTH size inputs
		);
		port (
			sel: in std_logic_vector(natural(ceil(log2(real(FAN_IN))))-1 downto 0);
			mux_in: in std_logic_vector((WIDTH*FAN_IN)-1 downto 0);
			mux_out: out std_logic_vector(WIDTH-1 downto 0)
		);
	end component;


	-- CONSTANTS --
	constant UNDEFINED: std_logic_vector(ADDR-1 downto 0) := (others => '-');

	-- opcodes
	constant STACK_RESET: std_logic_vector(1 downto 0) :=      "00";
	constant STACK_POP: std_logic_vector(1 downto 0) :=        "01";
	constant STACK_PUSH: std_logic_vector(1 downto 0) :=       "10";
	constant STACK_NOP: std_logic_vector(1 downto 0) :=        "11";

	-- SIGNALS --
	signal address_sig, final_address_sig,
		tosp_D_sig, tosp_Q_sig,
		pushed, popped: std_logic_vector(ADDR-1 downto 0);
	signal wren_sig, reset_sig, update_tosp: std_logic;

	signal sel_tosp_D: std_logic_vector(0 downto 0);
	signal tosp_mux_in: std_logic_vector((ADDR*2)-1 downto 0);
	signal sel_address: std_logic_vector(1 downto 0);
	signal address_mux_in: std_logic_vector((ADDR*4)-1 downto 0);

	-- COMPOSITE BEHAVIOUR --
	begin
		-- @note fixed data and address length on wizard-generated RAM
```

```
        assert ((WORD >= 16) and (ADDR >= 8))
        report "On-chip RAM is incompatible with LIFO parameters."
        severity FAILURE;

        MEMORY: onchip_ram
                port map (
                        address => final_address_sig(7 downto 0),
                        clock => clock,
                        data => stack_in(15 downto 0),
                        wren => wren_sig,
                        q => stack_out(15 downto 0)
                );

        TOS_POINTER_REG: Reg
                generic map (WIDTH => ADDR)
                port map (
                        clock => clock,
                        reset => reset_sig,
                        enable => update_tosp,
                        D => tosp_D_sig,
                        Q => tosp_Q_sig
                );

        MUX_TOSP: Multiplexer
                generic map (
                        WIDTH => ADDR,
                        FAN_IN => 2
                )
                port map (
                        sel => sel_tosp_D,
                        mux_in => tosp_mux_in,
                        mux_out => tosp_D_sig
                );

        MUX_ADDR: Multiplexer
                generic map (
                        WIDTH => ADDR,
                        FAN_IN => 4
                )
                port map (
                        sel => sel_address,
                        mux_in => address_mux_in,
                        mux_out => address_sig
                );

        wren_sig <= op(1) and (not op(0));   -- write <-> PUSH ("10")
        reset_sig <= op(1) nor op(0);        -- RESET <-> op="00"
        update_tosp <= op(1) xor op(0);      -- update TOSP <-> op="01"|"10"

        pushed <= std_logic_vector(unsigned(tosp_Q_sig) - 1);
        popped <= std_logic_vector(unsigned(tosp_Q_sig) + 1);

        sel_tosp_D(0) <= op(1);
        tosp_mux_in <= (
                pushed &       -- 1 -> PUSH
                popped         -- 0 -> POP
        );
        sel_address <= op;
        address_mux_in <= (
                tosp_Q_sig &   -- 11 -> read at current tosp
                pushed &       -- 10 -> write at next of stack
                popped &       -- 01   -> read at next of stack
                UNDEFINED      -- 00
        );

        final_address_sig <=std_logic_vector(unsigned(address_sig) + unsigned(offset));

        overflow <= '1' when (op = STACK_PUSH) and (unsigned(tosp_Q_sig) = 1)
                        else '0';

        underflow <= '1' when (unsigned(tosp_Q_sig) = 0) and (op = STACK_POP)
        else '0';

end architecture;
      ---------------------------------------------------------------------
```

- Temporização
  - *Clock to output*: 10,243 ns.
- Área
  - Funções combinacionais: 32 elementos.
  - Registradores: 8 bits.
  - Blocos de memória: 4.096 bits.

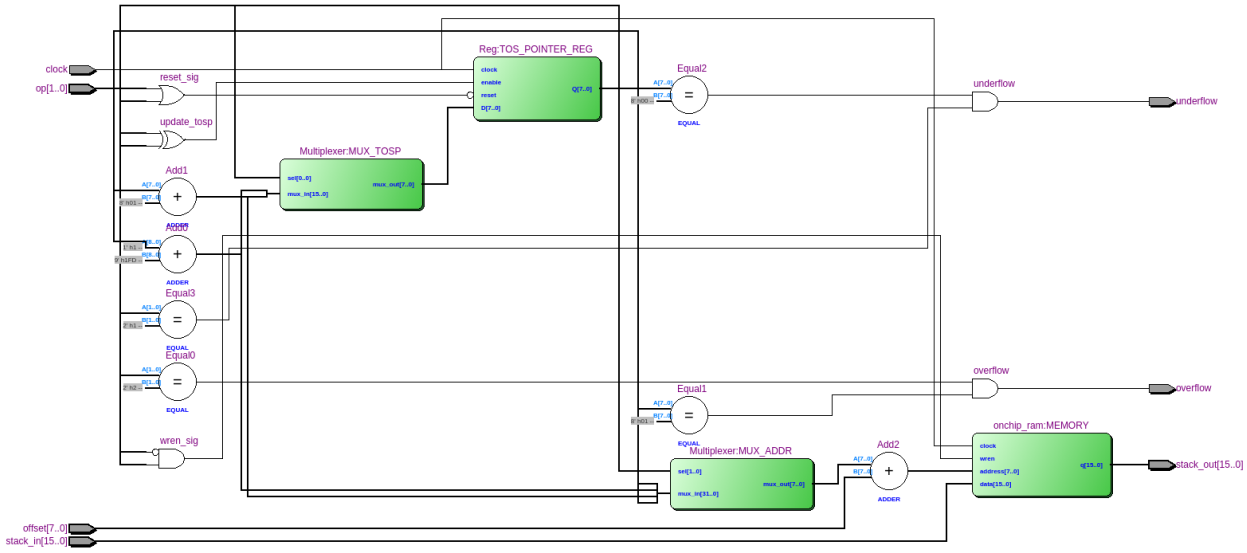


Figura 12 - Pilha Sintetizada. Fonte: O autor (2018).

3.2 Desenvolvimento do Bloco de Controle

O Bloco de Controle descreve a máquina de estados dada pela Tabela 5 com a adição de *reset* síncrono ativo em nível baixo e as saídas de Mealy descritas no capítulo anterior.

Percebe-se que **nenhuma lógica é aplicada sobre os sinais de erro dos *stacks***, que são simplesmente roteados das entradas vindas do Bloco Operativo para as saídas especificadas na interface do sistema; isso ocorre pois optou-se por deixar livre à implementação (considerando que o processador será embarcado em um outro sistema digital) a forma como esses erros serão tratados.

```
-----------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;


entity S4PU_Control is      -- S4PU Control Unit
	generic (ARCH: positive := 16);     -- considered 16-bit
	port (
		-- CLOCK --
		clock: in std_logic;

		-- EXTERNAL INPUTS --
		reset_n: in std_logic;         -- active low, synchronous
		mode: in std_logic;

		-- STATUS --
		instruction: in std_logic_vector(ARCH-1 downto 0);
		alu_zero: in std_logic;
		i_rs_overflow, i_rs_underflow: in std_logic;
		i_ds_overflow, i_ds_underflow: in std_logic;

		-- COMMANDS --
		ds_op: out std_logic_vector(1 downto 0);
		sel_ds_in: out std_logic;
		ds_pick: out std_logic;
		rs_op: out std_logic_vector(1 downto 0);
		sel_rs_in: out std_logic;
		alu_op: out std_logic_vector(3 downto 0);
		sel_alu_A: out std_logic_vector(2 downto 0);
		tos_load: out std_logic;
		pc_load: out std_logic;
		sel_pc_D: out std_logic_vector(2 downto 0);
		sel_addr: out std_logic_vector(2 downto 0);
		cir_load: out std_logic;
		sel_inst: out std_logic;

		-- EXTERNAL OUTPUTS --
		o_rs_overflow, o_rs_underflow: out std_logic;
		o_ds_overflow, o_ds_underflow: out std_logic;
		read, write: out std_logic
	);
	begin
		assert (ARCH >= 16)
			report "Architecture should be at least 16-bits."
			severity FAILURE;
end entity;


architecture fsm_v0 of S4PU_Control is       -- default
	-- INTERNAL STATE --
	type InternalState is (
		RESET,          FETCH,      DECODE,       CALL,
		LOAD_0,         LOAD_1,     STORE_0,      STORE_1,
		IF_FALSE,       IF_TRUE,    BRANCH,       RET,
		DROP,           LIT,        PICK_0,       PICK_1,
		TO_R,           R_FROM,     ALU_NOT,      ALU_OR,
		ALU_AND,        ALU_XOR,    ALU_ADD,      ALU_SUB,
		ALU_INC,        ALU_DEC,    ALU_EQUAL,    ALU_LESS,
		ALU_GREATER,    ALU_ZER,    ALU_NEG,      ALU_POS,
		ALU_SAL,        ALU_SAR,    DUP,          SWAP_0,
		SWAP_1,         ALU_SBL,    ALU_SBR
	);
	-- @todo attribute enum_encoding ... : string ... of InternalState: type is ...
	signal curr_state, next_state: InternalState;


	-- INSTRUCTION DECODING FUNCTION: f(instruction, alu_zero)
	function NEXT_STATE_DECODE (
			inst: std_logic_vector(15 downto 0);
			zero: std_logic
	) return InternalState is
		variable goto_state: InternalState;
			begin
```

```
if (inst(inst'right) = '0') then
        goto_state := CALL;

elsif (inst = x"8001") then
        goto_state := DECODE;

elsif (inst = x"8003") then
        goto_state := LOAD_0;

elsif (inst = x"8005") then
        goto_state := STORE_0;

elsif (inst = x"8007") then
        if (zero = '1') then
                goto_state := IF_FALSE;
        else
                goto_state := IF_TRUE;
        end if;

elsif (inst = x"8009") then
        goto_state := BRANCH;

elsif (inst = x"800D") then
        goto_state := RET;

elsif (inst = x"800F") then
        goto_state := DROP;

elsif (inst = x"8011") then
        goto_state := LIT;

elsif (inst = x"8013") then
        goto_state := PICK_0;

elsif (inst = x"8015") then
        goto_state := TO_R;

elsif (inst = x"8017") then
        goto_state := R_FROM;

elsif (inst = x"8019") then
        goto_state := ALU_NOT;

elsif (inst = x"801B") then
        goto_state := ALU_OR;

elsif (inst = x"801D") then
        goto_state := ALU_AND;

elsif (inst = x"801F") then
        goto_state := ALU_XOR;

elsif (inst = x"8021") then
        goto_state := ALU_ADD;

elsif (inst = x"8023") then
        goto_state := ALU_SUB;

elsif (inst = x"8025") then
        goto_state := ALU_INC;

elsif (inst = x"8027") then
        goto_state := ALU_DEC;

elsif (inst = x"8029") then
        goto_state := ALU_EQUAL;

elsif (inst = x"802B") then
        goto_state := ALU_LESS;

elsif (inst = x"802D") then
        goto_state := ALU_GREATER;
```

```
                                elsif (inst = x"8035") then
                                        goto_state := ALU_ZER;

                                elsif (inst = x"8037") then
                                        goto_state := ALU_NEG;

                                elsif (inst = x"8039") then
                                        goto_state := ALU_POS;

                                elsif (inst = x"8041") then
                                        goto_state := ALU_SAL;

                                elsif (inst = x"8043") then
                                        goto_state := ALU_SAR;

                                elsif (inst = x"804D") then
                                        goto_state := DUP;

                                elsif (inst = x"804F") then
                                        goto_state := SWAP_0;

                                elsif (inst = x"8051") then
                                        goto_state := ALU_SBL;

                                elsif (inst = x"8053") then
                                        goto_state := ALU_SBR;

                                -- undefined instruction code
                                else
                                        goto_state := RESET;
                                end if;

                                return goto_state;
end NEXT_STATE_DECODE;


-- CONSTANT OPCODES --
-- stacks
constant STACK_RESET: std_logic_vector(1 downto 0) :=       "00";
constant STACK_POP: std_logic_vector(1 downto 0) :=         "01";
constant STACK_PUSH: std_logic_vector(1 downto 0) :=        "10";
constant STACK_NOP: std_logic_vector(1 downto 0) :=         "11";
-- alu
constant ALU_OP_NOP: std_logic_vector(3 downto 0) :=        "0000";
constant ALU_OP_ADD: std_logic_vector(3 downto 0) :=        "0001";
constant ALU_OP_SUB: std_logic_vector(3 downto 0) :=        "0010";
constant ALU_OP_CET: std_logic_vector(3 downto 0) :=        "0011";
constant ALU_OP_CLT: std_logic_vector(3 downto 0) :=        "0100";
constant ALU_OP_CGT: std_logic_vector(3 downto 0) :=        "0101";
constant ALU_OP_ZER: std_logic_vector(3 downto 0) :=        "0110";
constant ALU_OP_NEG: std_logic_vector(3 downto 0) :=        "0111";
constant ALU_OP_POS: std_logic_vector(3 downto 0) :=        "1000";
constant ALU_OP_XOR: std_logic_vector(3 downto 0) :=        "1001";
constant ALU_OP_OR: std_logic_vector(3 downto 0) :=         "1010";
constant ALU_OP_AND: std_logic_vector(3 downto 0) :=        "1011";
constant ALU_OP_SAL: std_logic_vector(3 downto 0) :=        "1100";
constant ALU_OP_SAR: std_logic_vector(3 downto 0) :=        "1101";
constant ALU_OP_SBL: std_logic_vector(3 downto 0) :=        "1110";
constant ALU_OP_SBR: std_logic_vector(3 downto 0) :=        "1111";


-- BEHAVIOUR --
begin
        -- FSM next state logic
        NSL: process (curr_state, reset_n, instruction, alu_zero) is
                begin
                        if (reset_n = '0') then         -- synchronous reset
                                next_state <= RESET;
                        else
                                case curr_state is
                                        when FETCH|CALL|IF_FALSE|IF_TRUE|
                                        BRANCH|RET|LIT =>
                                                next_state <= DECODE;
```

```
                                                when RESET =>
                                                        next_state <= FETCH;

                                                when LOAD_0 =>
                                                        next_state <= LOAD_1;

                                                when STORE_0 =>
                                                        next_state <= STORE_1;

                                                when PICK_0 =>
                                                        next_state <= PICK_1;

                                                when SWAP_0 =>
                                                        next_state <= SWAP_1;

                                                when others =>
                                                        next_state <=
NEXT_STATE_DECODE(instruction(15 downto 0), alu_zero); -- f(instruction, alu_zero)
                                        end case;
                                end if;
                end process;

                -- memory element
                ME: process (clock) is
                        begin
                                if (rising_edge(clock)) then
                                        curr_state <= next_state;
                                end if;
                end process;

                -- output logic
                with curr_state select ds_op <=
                        STACK_RESET when RESET,
                        STACK_POP when STORE_0|STORE_1|IF_FALSE|IF_TRUE|DROP|
                                TO_R|ALU_OR|ALU_AND|ALU_XOR|ALU_ADD|
                                ALU_SUB|ALU_EQUAL|ALU_LESS|ALU_GREATER|SWAP_0,
                        STACK_PUSH when LIT|R_FROM|DUP|SWAP_1,
                        STACK_NOP when others;

                with curr_state select sel_ds_in <=
                        '1' when SWAP_1,            -- RS
                        '0' when others;            -- TOS

                with curr_state select ds_pick <=
                        '1' when PICK_0,            -- TOS(7 downto 0)
                        '0' when others;            -- "0"

                with curr_state select rs_op <=
                        STACK_RESET when RESET,
                        STACK_POP when RET|R_FROM|SWAP_1,
                        STACK_PUSH when CALL|TO_R|SWAP_0,
                        STACK_NOP when others;

                with curr_state select sel_rs_in <=
                        '0' when CALL,   -- PC
                        '1' when others;  -- TOS

                with curr_state select alu_op <=
                        ALU_OP_ADD when ALU_ADD|ALU_INC|ALU_DEC,
                        ALU_OP_SUB when ALU_NOT|ALU_SUB,
                        ALU_OP_CET when ALU_EQUAL,
                        ALU_OP_CLT when ALU_LESS,
                        ALU_OP_CGT when ALU_GREATER,
                        ALU_OP_ZER when ALU_ZER,
                        ALU_OP_NEG when ALU_NEG,
                        ALU_OP_POS when ALU_POS,
                        ALU_OP_XOR when ALU_XOR,
                        ALU_OP_OR when ALU_OR,
                        ALU_OP_AND when ALU_AND,
                        ALU_OP_SAL when ALU_SAL,
                        ALU_OP_SAR when ALU_SAR,
                        ALU_OP_SBL when ALU_SBL,
```

```
                ALU_OP_SBR when ALU_SBR,
                ALU_OP_NOP when others;

        with curr_state select sel_alu_A <=
                "000" when RESET,                       -- "0"
                "001" when ALU_INC,                     -- "+1"
                "010" when ALU_NOT|ALU_DEC,             -- "-1"
                "100" when STORE_1|IF_FALSE|IF_TRUE|DROP|PICK_1|
                        TO_R|ALU_OR|ALU_AND|ALU_XOR|ALU_ADD|                    – DS
                        ALU_SUB|ALU_EQUAL|ALU_LESS|ALU_GREATER|SWAP_0,
                "101" when R_FROM,                      -- RS
                "110" when LOAD_1|LIT,                  -- MEM
                "011" when others;                      -- TOS

        tos_load <= '1';

        with curr_state select pc_load <=
                '0' when LOAD_0|STORE_0|PICK_0|SWAP_0,
                '1' when RESET|FETCH|CALL|IF_FALSE|
                        IF_TRUE|BRANCH|RET|LIT,
                instruction(instruction'right) when others; -- g(instruction)

        with curr_state select sel_pc_D <=
                ("00" & mode) when RESET,       -- h(mode)
                "011" when CALL,                -- CIR + 1
                "100" when IF_FALSE|BRANCH,     -- MEM + 1
                "101" when RET,                 -- RS + 1
                "010" when others;              -- PC + 1

        with curr_state select sel_addr <=
                "001" when LOAD_0|STORE_0,      -- TOS
                "010" when IF_FALSE|BRANCH,     -- MEM
                "011" when CALL,                -- CIR
                "100" when RET,                 -- RS
                "000" when others;              -- PC

        with curr_state select read <=
                '0' when STORE_0|RESET|PICK_0|SWAP_0,
                '1' when others;

        with curr_state select write <=
                '1' when STORE_0,
                '0' when others;

        with curr_state select cir_load <=
                '0' when LOAD_1|STORE_1|PICK_1|SWAP_1,
                '1' when others;

        with curr_state select sel_inst <=
                '1' when LOAD_1|STORE_1|PICK_1|SWAP_1,          -- CIR
                '0' when others;                                -- MEM

        -- route stack error status to external output
        o_rs_overflow <= i_rs_overflow;
        o_rs_underflow <= i_rs_underflow;
        o_ds_overflow <= i_ds_overflow;
        o_ds_underflow <= i_ds_underflow;

end architecture;
        ------------------------------------------------------------------------
```

- Temporização
  - *Clock to output*: 10,204 ns.
- Área
  - Funções combinacionais: 118 elementos.
  - Registradores: 39 bits.

- Blocos de memória: 0 bits.

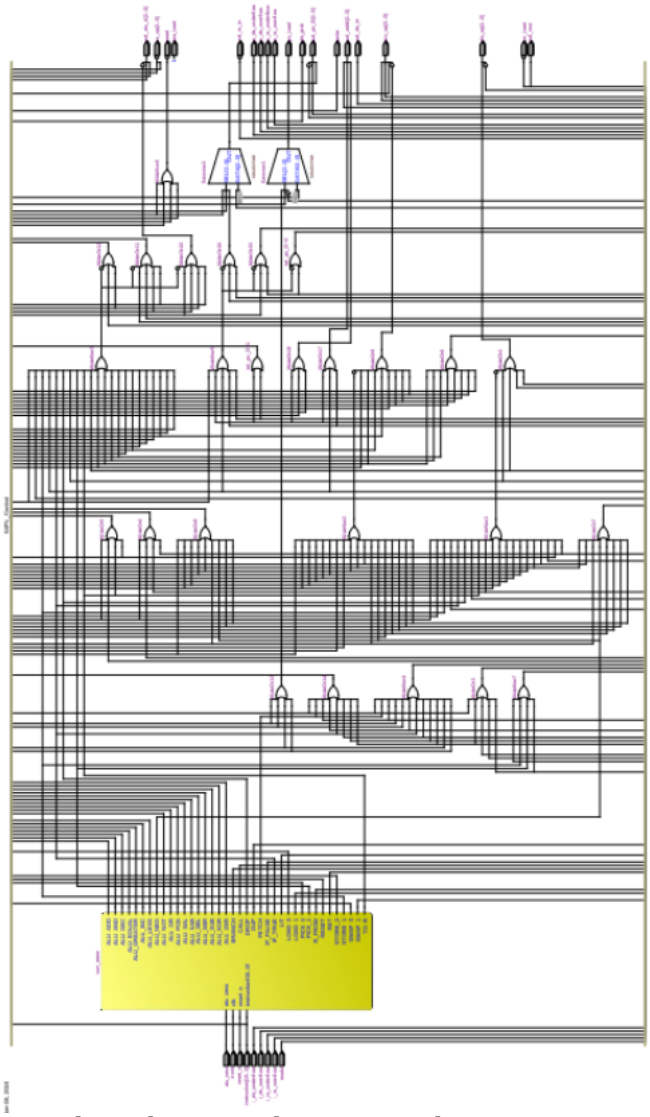


Figura 13 - Bloco de Controle Sintetizado. Fonte: o autor (2018).

3.3 Desenvolvimento da Unidade de Processamento

Como projetado no capítulo anterior, a estrutura da Unidade de Processamento Simples com Arquitetura Baseada em Pilhas simplesmente integra Bloco Operativo e Bloco de Controle, finalizando o desenvolvimento do sistema digital proposto.

```
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
```

```
entity S4PU is -- Simple Forth Processing Unit
	generic (ARCH: positive := 16);
	port (
		-- CLOCK --
		clock: in std_logic;

		-- SYSTEM CONTROL --
		reset_n: in std_logic;-- active low, synchronous
		mode: in std_logic;

		-- STACK ERRORS --
		rs_overflow, rs_underflow: out std_logic;
		ds_overflow, ds_underflow: out std_logic;

		-- PUPPET/SLAVE MEMORY (Avalon spec.) --
		read, write: out std_logic;
		address: out std_logic_vector(ARCH-1 downto 0);
		readdata: in std_logic_vector(ARCH-1 downto 0);
		writedata: out std_logic_vector(ARCH-1 downto 0)
	);
end entity;


architecture composite of S4PU is    -- default
	-- SIGNALS --
	signal instruction_sig: std_logic_vector(ARCH-1 downto 0);
	signal alu_zero_sig: std_logic;
	signal rs_overflow_sig, rs_underflow_sig: std_logic;
	signal ds_overflow_sig, ds_underflow_sig: std_logic;
	signal ds_op_sig: std_logic_vector(1 downto 0);
	signal sel_ds_in_sig: std_logic;
	signal ds_pick_sig: std_logic;
	signal rs_op_sig: std_logic_vector(1 downto 0);
	signal sel_rs_in_sig: std_logic;
	signal alu_op_sig: std_logic_vector(3 downto 0);
	signal sel_alu_A_sig: std_logic_vector(2 downto 0);
	signal tos_load_sig: std_logic;
	signal pc_load_sig: std_logic;
	signal sel_pc_D_sig: std_logic_vector(2 downto 0);
	signal sel_addr_sig: std_logic_vector(2 downto 0);
	signal cir_load_sig: std_logic;
	signal sel_inst_sig: std_logic;


	-- COMPONENTS --
	component S4PU_Control is     -- S4PU Control Unit
		generic (ARCH: positive := 16);      -- considered 16-bit
		port (
			-- CLOCK --
			clock: in std_logic;

			-- EXTERNAL INPUTS --
			reset_n: in std_logic;-- active low, synchronous
			mode: in std_logic;

			-- STATUS --
			instruction: in std_logic_vector(ARCH-1 downto 0);
			alu_zero: in std_logic;
			i_rs_overflow, i_rs_underflow: in std_logic;
			i_ds_overflow, i_ds_underflow: in std_logic;

			-- COMMANDS --
			ds_op: out std_logic_vector(1 downto 0);
			sel_ds_in: out std_logic;
			ds_pick: out std_logic;
			rs_op: out std_logic_vector(1 downto 0);
			sel_rs_in: out std_logic;
			alu_op: out std_logic_vector(3 downto 0);
			sel_alu_A: out std_logic_vector(2 downto 0);
			tos_load: out std_logic;
			pc_load: out std_logic;
			sel_pc_D: out std_logic_vector(2 downto 0);
			sel_addr: out std_logic_vector(2 downto 0);
```

```
                cir_load: out std_logic;
                sel_inst: out std_logic;

                -- EXTERNAL OUTPUTS --
                o_rs_overflow, o_rs_underflow: out std_logic;
                o_ds_overflow, o_ds_underflow: out std_logic;
                read, write: out std_logic
        );
end component;

component S4PU_Datapath is    -- S4PU Operative Unit
        generic (ARCH: positive := 16);
        port (
                -- CLOCK --
                clock: in std_logic;

                -- EXTERNAL INPUTS --
                readdata: in std_logic_vector(ARCH-1 downto 0);

                -- COMMANDS --
                ds_op: in std_logic_vector(1 downto 0);
                sel_ds_in: in std_logic;
                ds_pick: in std_logic;
                rs_op: in std_logic_vector(1 downto 0);
                sel_rs_in: in std_logic;
                alu_op: in std_logic_vector(3 downto 0);
                sel_alu_A: in std_logic_vector(2 downto 0);
                tos_load: in std_logic;
                pc_load: in std_logic;
                sel_pc_D: in std_logic_vector(2 downto 0);
                sel_addr: in std_logic_vector(2 downto 0);
                cir_load: in std_logic;
                sel_inst: in std_logic;

                -- STATUS --
                instruction: out std_logic_vector(ARCH-1 downto 0);
                alu_zero: out std_logic;
                rs_overflow, rs_underflow: out std_logic;
                ds_overflow, ds_underflow: out std_logic;

                -- EXTERNAL OUTPUTS --
                address: out std_logic_vector(ARCH-1 downto 0);
                writedata: out std_logic_vector(ARCH-1 downto 0)
        );
end component;


-- COMPOSITE BEHAVIOUR --
begin
        CONTROL_BLOCK: S4PU_Control
                generic map (ARCH => ARCH)
                port map (
                        clock => clock,
                        reset_n => reset_n,
                        mode => mode,
                        instruction => instruction_sig,
                        alu_zero => alu_zero_sig,
                        i_rs_overflow => rs_overflow_sig,
                        i_rs_underflow => rs_underflow_sig,
                        i_ds_overflow => ds_overflow_sig,
                        i_ds_underflow => ds_underflow_sig,
                        ds_op => ds_op_sig,
                        sel_ds_in => sel_ds_in_sig,
                        ds_pick => ds_pick_sig,
                        rs_op => rs_op_sig,
                        sel_rs_in => sel_rs_in_sig,
                        alu_op => alu_op_sig,
                        sel_alu_A => sel_alu_A_sig,
                        tos_load => tos_load_sig,
                        pc_load => pc_load_sig,
                        sel_pc_D => sel_pc_D_sig,
                        sel_addr => sel_addr_sig,
                        cir_load => cir_load_sig,
                        sel_inst => sel_inst_sig,
                        o_rs_overflow => rs_overflow,
```

```
                        o_rs_underflow => rs_underflow,
                        o_ds_overflow => ds_overflow,
                        o_ds_underflow => ds_underflow,
                        read => read, write => write
                );

        OPERATIVE_BLOCK: S4PU_Datapath
                generic map (ARCH => ARCH)
                port map (
                        clock => clock,
                        readdata => readdata,
                        instruction => instruction_sig,
                        alu_zero => alu_zero_sig,
                        rs_overflow => rs_overflow_sig,
                        rs_underflow => rs_underflow_sig,
                        ds_overflow => ds_overflow_sig,
                        ds_underflow => ds_underflow_sig,
                        ds_op => ds_op_sig,
                        sel_ds_in => sel_ds_in_sig,
                        ds_pick => ds_pick_sig,
                        rs_op => rs_op_sig,
                        sel_rs_in => sel_rs_in_sig,
                        alu_op => alu_op_sig,
                        sel_alu_A => sel_alu_A_sig,
                        tos_load => tos_load_sig,
                        pc_load => pc_load_sig,
                        sel_pc_D => sel_pc_D_sig,
                        sel_addr => sel_addr_sig,
                        cir_load => cir_load_sig,
                        sel_inst => sel_inst_sig,
                        address => address,
                        writedata => writedata
                );
end architecture;
        ------------------------------------------------------------------------
```

- Temporização
  - *Clock to output*: 12,570 ns.
- Área
  - Funções combinacionais: 618 elementos.
  - Registradores: 103 bits.
  - Blocos de memória: 8.192 bits.

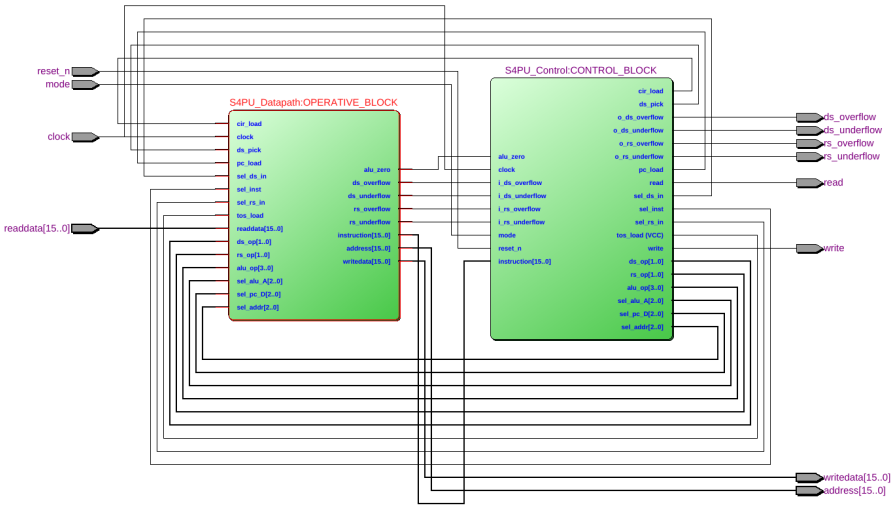


Figura 14 - Processador Sintetizado. Fonte: o autor (2018).

4 Testes e Validação

Os componentes do projeto devem ser testados e validados antes de partir para a etapa de prototipação. Logo, procura-se desenvolver um ***testbench*** ou uma simulação através de estímulos para cada subsistema, pois assegurando-se de seu funcionamento é possível aumentar a abstração nos próximos níveis da hierarquia do sistema, facilitando o procedimento de *debugging*.

Os processos de testes do sistema digital e suas partes serão realizados no ambiente do *software* **ModelSim** (Mentor Graphics).

4.1 Validação do Bloco Operativo

Sendo o Bloco Operativo apenas a integração de componentes mencionada no capítulo anterior, percebe-se que os sinais de sua interface são simplesmente roteados indo e vindo diretamente desses mesmos componentes, de forma que as saídas do bloco não revelam muito sobre seu estado interno. Assim, havendo confiabilidade no funcionamento de suas partes, justifica-se deixar de lado testes exaustivos do *datapath*.

4.1.1 - Multiplexador

Testou-se o multiplexador genérico com uma instância de 4 entradas de 16 bits cada, ou seja, um multiplexador 64 para 16. As entradas são mantidas fixas enquanto varia-se o valor no sinal de seleção; a saída deve reagir após o tempo de atraso do componente e tomar o valor da entrada correspondente.

```
      ----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;     -- type conversion
use ieee.math_real.all;       -- log2 & ceil


entity TB_Mux is
end entity;


architecture testbench_64x16 of TB_Mux is   -- default
       -- DUT INTERFACE --
       component Multiplexer is      -- Mux
              generic (
                     WIDTH: positive := 1;
                     FAN_IN: positive := 2 -- no. of WIDTH size inputs
              );
              port (
                     sel: in std_logic_vector(natural(ceil(log2(real(FAN_IN))))-1 downto 0);
                     mux_in: in std_logic_vector((WIDTH*FAN_IN)-1 downto 0);
                     mux_out: out std_logic_vector(WIDTH-1 downto 0)
              );
       end component;


       -- CONSTANTS --
       constant LAG: time := 8 ns;

       constant N: natural := 16;
       constant ADDR: positive := 2;
       constant FAN: positive := 2**ADDR;

       constant A: std_logic_vector(N-1 downto 0) := x"BABE";
       constant B: std_logic_vector(N-1 downto 0) := x"BEEF";
       constant C: std_logic_vector(N-1 downto 0) := x"CAFE";
       constant D: std_logic_vector(N-1 downto 0) := x"FEED";


       -- SIGNALS --
       signal sel: std_logic_vector(ADDR-1 downto 0);
       signal mux_in: std_logic_vector((N*FAN)-1 downto 0);
       signal mux_out: std_logic_vector(N-1 downto 0);


       -- TESTING --
       begin
              UUT: Multiplexer
                     generic map (
                            WIDTH => N,
                            FAN_IN => FAN
                     )
                     port map (
                            sel => sel,
                            mux_in => mux_in,
                            mux_out => mux_out
                     );

              mux_in <= (
                     D &     -- 11
                     C &     -- 10
                     B &     -- 01
                     A       -- 00
              );

              STIMULUS: process is
                     begin
                            wait for LAG;

                            sel <= "00";
                            wait for LAG;
                            assert (mux_out=A)
                                   report "Multiplexing error when select '00'"
                                   severity ERROR;
```

```
                        sel <= "01";
                        wait for LAG;
                        assert (mux_out=B)
                                report "Multiplexing error when select '01'"
                                severity ERROR;

                        sel <= "10";
                        wait for LAG;
                        assert (mux_out=C)
                                report "Multiplexing error when select '10'"
                                severity ERROR;

                        sel <= "11";
                        wait for LAG;
                        assert (mux_out=D)
                                report "Multiplexing error when select '11'"
                                severity ERROR;

                        -- TOTAL TIME : 5 * LAG
        end process;

end architecture;
    ---------------------------------------------------------------------
```

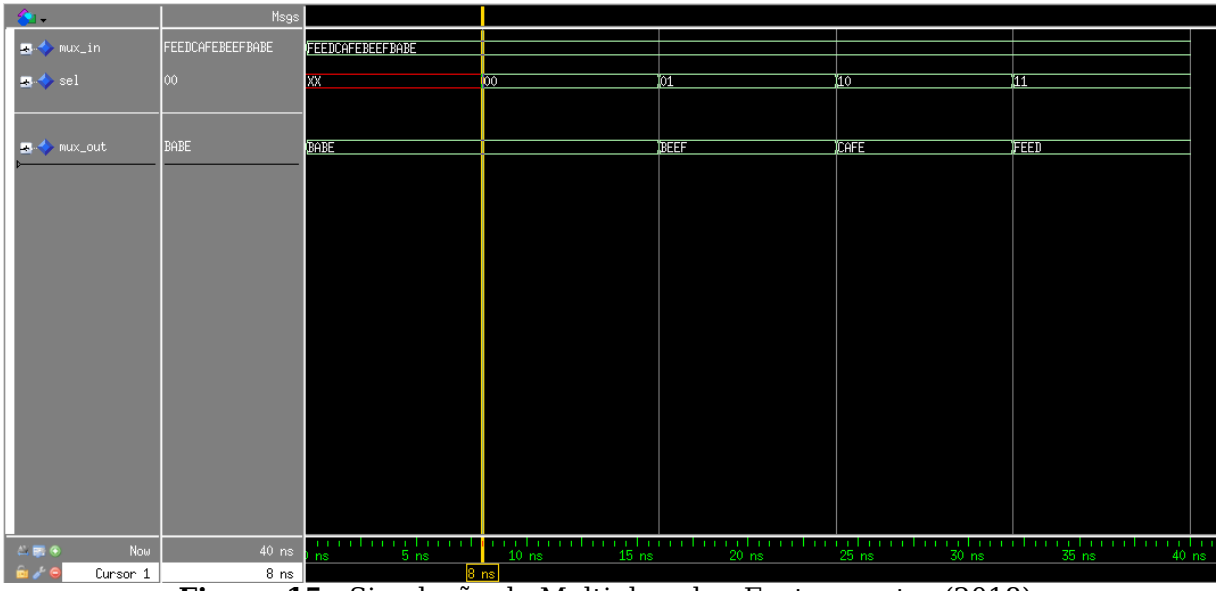


Figura 15 - Simulação do Multiplexador. Fonte: o autor (2018).

4.1.2 - Registrador

Foi utilizado um registrador de 8 bits para o *testbench* do componente. Este envolve testar a carga do registrador, seu *reset* assíncrono e a desabilitação de carga, avaliando a saída que reflete seu estado interno após cada operação.

```
    ---------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
```

```
entity TB_Reg is
end entity;


architecture testbench of TB_Reg is  -- default
	-- DUT INTERFACE --
	component Reg is	-- Register with asynchronous reset
		generic (WIDTH: positive := 8);
		port (
			-- CLOCK & CONTROL --
			clock: in std_logic;
			reset, enable: in std_logic;  -- asynchronous reset

			-- DATA --
			D: in std_logic_vector(WIDTH-1 downto 0);
			Q: out std_logic_vector(WIDTH-1 downto 0)
		);
	end component;


	-- CONSTANTS --
	constant T: time := 20 ns;
	constant LAG: time := 5 ns;
	constant N: natural := 8;


	-- SIGNALS --
	signal clock, reset, enable: std_logic;
	signal D, Q: std_logic_vector(N-1 downto 0);


	-- TESTING --
	begin
		UUT: Reg
			generic map (WIDTH => N)
			port map (
				clock => clock,
				reset => reset,
				enable => enable,
				D => D,
				Q => Q
			);

		CLK: process is -- 50% duty
			begin
				clock <= '0';
				wait for (T/2);
				clock <= '1';
				wait for (T/2);
		end process;

		STIMULUS: process is
			begin
				wait until rising_edge(clock);
				wait for LAG;

				reset <= '0';
				enable <= '1';
				D <= x"d0";
				wait until rising_edge(clock);
				wait for LAG;
				assert (Q=x"d0")
					report "Register Load failed"
					severity ERROR;

				reset <= '1';
				enable <= '0';
				wait for LAG;
				assert (Q=x"00")
					report "Register Asynchronous Reset failed"
					severity ERROR;

				reset <= '0';
				enable <= '0';
				D <= x"FA";
```

```
                        wait until rising_edge(clock);
                        wait for LAG;
                        assert (Q=x"00")
                                report "Register Disable failed"
                                severity ERROR;

                        enable <= '1';
                        D <= x"FA";
                        wait until rising_edge(clock);
                        wait for LAG;
                        assert (Q=x"FA")
                                report "Register Load failed"
                                severity ERROR;

                        -- TOTAL TIME : 4 * T
        end process;

end architecture;
    ---------------------------------------------------------------------
```

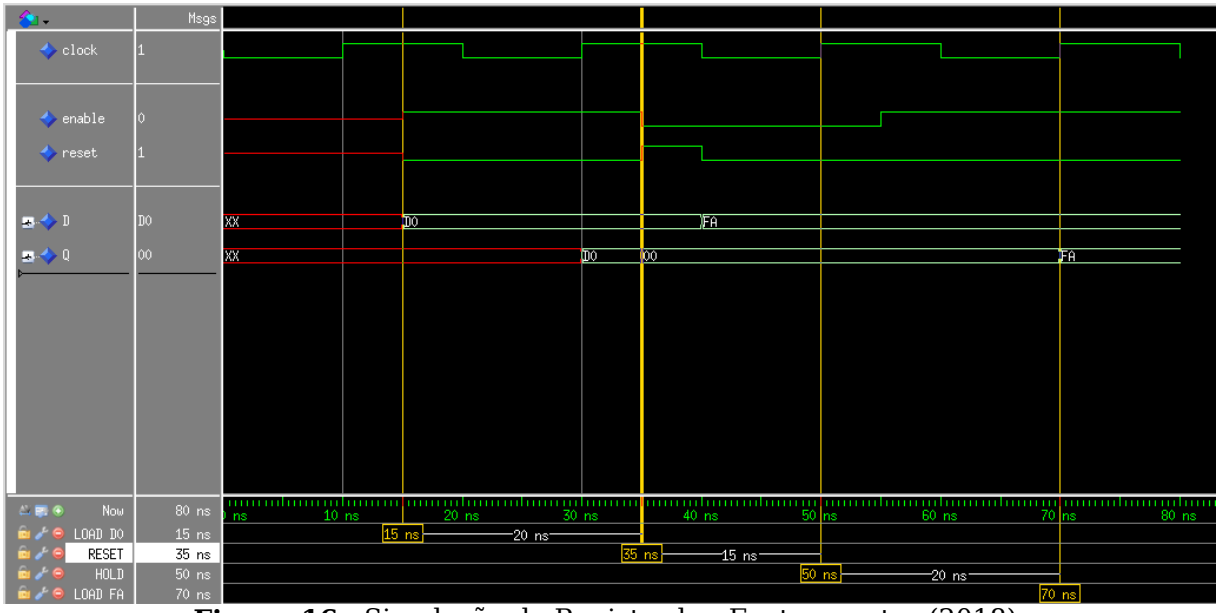


Figura 16 - Simulação do Registrador. Fonte: o autor (2018).

4.1.3 - Unidade Lógica e Aritmética

Ao validar a ULA projetada, contemplou-se suas 16 possíveis operações com valores de entrada pré-determinados, amostrando as saídas após o tempo de atraso do componente e levando em conta possíveis *overflows* aritméticos.

```
    ---------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;     -- type conversion


entity TB_ALU is
end entity;
```

```
architecture testbench_16 of TB_ALU is      -- default
	-- DUT INTERFACE --
	component ALU_16 is   -- 16-operation ALU
		generic (WORD: positive := 16);      -- should be greater than 8
		port (
			-- CONTROL --
			op: in std_logic_vector(3 downto 0);

			-- DATA --
			A, B: in std_logic_vector(WORD-1 downto 0);
			Y: out std_logic_vector(WORD-1 downto 0);

			-- SIGNALS --
			zero, overflow: out std_logic
		);
	end component;


	-- CONSTANTS --
	constant LAG: time := 20 ns;

	constant N: natural := 16;

	-- opcodes
	constant ALU_OP_NOP: std_logic_vector(3 downto 0) :=       "0000";
	constant ALU_OP_ADD: std_logic_vector(3 downto 0) :=       "0001";
	constant ALU_OP_SUB: std_logic_vector(3 downto 0) :=       "0010";
	constant ALU_OP_CET: std_logic_vector(3 downto 0) :=       "0011";
	constant ALU_OP_CLT: std_logic_vector(3 downto 0) :=       "0100";
	constant ALU_OP_CGT: std_logic_vector(3 downto 0) :=       "0101";
	constant ALU_OP_ZER: std_logic_vector(3 downto 0) :=       "0110";
	constant ALU_OP_NEG: std_logic_vector(3 downto 0) :=       "0111";
	constant ALU_OP_POS: std_logic_vector(3 downto 0) :=       "1000";
	constant ALU_OP_XOR: std_logic_vector(3 downto 0) :=       "1001";
	constant ALU_OP_OR: std_logic_vector(3 downto 0) :=        "1010";
	constant ALU_OP_AND: std_logic_vector(3 downto 0) :=       "1011";
	constant ALU_OP_SAL: std_logic_vector(3 downto 0) :=       "1100";
	constant ALU_OP_SAR: std_logic_vector(3 downto 0) :=       "1101";
	constant ALU_OP_SBL: std_logic_vector(3 downto 0) :=       "1110";
	constant ALU_OP_SBR: std_logic_vector(3 downto 0) :=       "1111";


	-- SIGNALS --
	signal op: std_logic_vector(3 downto 0);
	signal A, B, Y: std_logic_vector(N-1 downto 0);
	signal zero, overflow: std_logic;


	-- TESTING --
	begin
		UUT: ALU_16
			generic map (WORD => N)
			port map (
				op => op,
				A => A,
				B => B,
				Y => Y,
				zero => zero,
				overflow => overflow
			);

		STIMULUS: process is
			begin
				wait for LAG;

				-- Y = A
				op <= ALU_OP_NOP;
				A <= x"C0DE";
				wait for LAG;
				assert ((Y=x"C0DE") and (zero='0') and (overflow='0'))
						report "ALU error on operation NOP"
						severity ERROR;
				op <= ALU_OP_NOP;
```

```
A <= x"0000";
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation NOP"
                severity ERROR;

-- Y = A + B
op <= ALU_OP_ADD;
A <= std_logic_vector(to_signed(0, N));
B <= std_logic_vector(to_signed(0, N));
wait for LAG;
assert ((signed(Y)=0) and (zero='1') and (overflow='0'))
                report "ALU error on operation ADD"
                severity ERROR;
op <= ALU_OP_ADD;
A <= std_logic_vector(to_signed(32767, N));
B <= std_logic_vector(to_signed(1, N));
wait for LAG;
assert ((signed(Y)=-32768) and (zero='0') and (overflow='1'))
                report "ALU error on operation ADD"
                severity ERROR;
op <= ALU_OP_ADD;
A <= std_logic_vector(to_signed(-32768, N));
B <= std_logic_vector(to_signed(-1, N));
wait for LAG;
assert ((signed(Y)=32767) and (zero='0') and (overflow='1'))
                report "ALU error on operation ADD"
                severity ERROR;

-- Y = A - B
op <= ALU_OP_SUB;
A <= std_logic_vector(to_signed(0, N));
B <= std_logic_vector(to_signed(0, N));
wait for LAG;
assert ((signed(Y)=0) and (zero='1') and (overflow='0'))
                report "ALU error on operation SUB"
                severity ERROR;
op <= ALU_OP_SUB;
A <= std_logic_vector(to_signed(32767, N));
B <= std_logic_vector(to_signed(-1, N));
wait for LAG;
assert ((signed(Y)=-32768) and (zero='0') and (overflow='1'))
                report "ALU error on operation SUB"
                severity ERROR;
op <= ALU_OP_SUB;
A <= std_logic_vector(to_signed(-32768, N));
B <= std_logic_vector(to_signed(1, N));
wait for LAG;
assert ((signed(Y)=32767) and (zero='0') and (overflow='1'))
                report "ALU error on operation SUB"
                severity ERROR;

-- Y = ?(A = B)
op <= ALU_OP_CET;
A <= x"C0DE";
B <= x"1337";
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation EQ"
                severity ERROR;
op <= ALU_OP_CET;
A <= x"FEED";
B <= x"FEED";
wait for LAG;
assert ((Y=x"FFFF") and (zero='0') and (overflow='0'))
                report "ALU error on operation EQ"
                severity ERROR;

-- Y = ?(A < B)
op <= ALU_OP_CLT;
A <= std_logic_vector(to_signed(42, N));
B <= std_logic_vector(to_signed(-55, N));
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation LESS"
```

```
                severity ERROR;
op <= ALU_OP_CLT;
A <= std_logic_vector(to_signed(-42, N));
B <= std_logic_vector(to_signed(55, N));
wait for LAG;
assert ((Y=x"FFFF") and (zero='0') and (overflow='0'))
                report "ALU error on operation LESS"
                severity ERROR;
op <= ALU_OP_CLT;
A <= std_logic_vector(to_signed(7, N));
B <= std_logic_vector(to_signed(7, N));
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation LESS"
                severity ERROR;

-- Y = ?(A > B)
op <= ALU_OP_CGT;
A <= std_logic_vector(to_signed(42, N));
B <= std_logic_vector(to_signed(-55, N));
wait for LAG;
assert ((Y=x"FFFF") and (zero='0') and (overflow='0'))
                report "ALU error on operation GREATER"
                severity ERROR;
op <= ALU_OP_CGT;
A <= std_logic_vector(to_signed(-55, N));
B <= std_logic_vector(to_signed(42, N));
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation GREATER"
                severity ERROR;
op <= ALU_OP_CGT;
A <= std_logic_vector(to_signed(7, N));
B <= std_logic_vector(to_signed(7, N));
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation GREATER"
                severity ERROR;

-- Y = ?(A = 0)
op <= ALU_OP_ZER;
A <= x"AAAA";
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation ZER"
                severity ERROR;
op <= ALU_OP_ZER;
A <= x"0000";
wait for LAG;
assert ((Y=x"FFFF") and (zero='0') and (overflow='0'))
                report "ALU error on operation ZER"
                severity ERROR;

-- Y = ?(A < 0)
op <= ALU_OP_NEG;
A <= std_logic_vector(to_signed(0, N));
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation NEG"
                severity ERROR;
op <= ALU_OP_NEG;
A <= std_logic_vector(to_signed(5, N));
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation NEG"
                severity ERROR;
op <= ALU_OP_NEG;
A <= std_logic_vector(to_signed(-5, N));
wait for LAG;
assert ((Y=x"FFFF") and (zero='0') and (overflow='0'))
                report "ALU error on operation NEG"
                severity ERROR;

-- Y = ?(A > 0)
op <= ALU_OP_POS;
```

```
A <= std_logic_vector(to_signed(0, N));
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation POS"
                severity ERROR;
op <= ALU_OP_POS;
A <= std_logic_vector(to_signed(5, N));
wait for LAG;
assert ((Y=x"FFFF") and (zero='0') and (overflow='0'))
                report "ALU error on operation POS"
                severity ERROR;
op <= ALU_OP_POS;
A <= std_logic_vector(to_signed(-5, N));
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation POS"
                severity ERROR;

-- Y = A xor B
op <= ALU_OP_XOR;
A <= x"AAAA";
B <= x"5555";
wait for LAG;
assert ((Y=x"FFFF") and (zero='0') and (overflow='0'))
                report "ALU error on operation XOR"
                severity ERROR;
op <= ALU_OP_XOR;
A <= x"BBBB";
B <= x"BBBB";
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation XOR"
                severity ERROR;

-- Y = A or B
op <= ALU_OP_OR;
A <= x"FAFA";
B <= x"F5F5";
wait for LAG;
assert ((Y=x"FFFF") and (zero='0') and (overflow='0'))
                report "ALU error on operation OR"
                severity ERROR;
op <= ALU_OP_OR;
A <= x"0000";
B <= x"CCCC";
wait for LAG;
assert ((Y=x"CCCC") and (zero='0') and (overflow='0'))
                report "ALU error on operation OR"
                severity ERROR;

-- Y = A and B
op <= ALU_OP_AND;
A <= x"AAAA";
B <= x"5555";
wait for LAG;
assert ((Y=x"0000") and (zero='1') and (overflow='0'))
                report "ALU error on operation AND"
                severity ERROR;
op <= ALU_OP_AND;
A <= x"8000";
B <= x"FFFF";
wait for LAG;
assert ((Y=x"8000") and (zero='0') and (overflow='0'))
                report "ALU error on operation AND"
                severity ERROR;

-- Y = A * 2
op <= ALU_OP_SAL;
A <= std_logic_vector(to_signed(-160, N));
wait for LAG;
assert ((signed(Y)=-320) and (zero='0') and (overflow='0'))
                report "ALU error on operation SAL"
                severity ERROR;
op <= ALU_OP_SAL;
A <= std_logic_vector(to_signed(16384, N));
```

```
                wait for LAG;
                assert ((signed(Y)=-32768) and (zero='0') and (overflow='1'))
                        report "ALU error on operation SAL"
                        severity ERROR;

                -- Y = A / 2
                op <= ALU_OP_SAR;
                A <= std_logic_vector(to_signed(320, N));
                wait for LAG;
                assert ((signed(Y)=160) and (zero='0') and (overflow='0'))
                        report "ALU error on operation SAR"
                        severity ERROR;
                op <= ALU_OP_SAR;
                A <= std_logic_vector(to_signed(-3, N));
                wait for LAG;
                assert ((signed(Y)=-2) and (zero='0') and (overflow='0'))
                        report "ALU error on operation SAR"
                        severity ERROR;

                -- Y = A << 8
                op <= ALU_OP_SBL;
                A <= x"ABCD";
                wait for LAG;
                assert ((Y=x"CD00") and (zero='0') and (overflow='0'))
                        report "ALU error on operation SBL"
                        severity ERROR;

                -- Y = A >> 8
                op <= ALU_OP_SBR;
                A <= x"ABCD";
                wait for LAG;
                assert ((Y=x"00AB") and (zero='0') and (overflow='0'))
                        report "ALU error on operation SBR"
                        severity ERROR;

                -- TOTAL TIME : 37 * LAG
        end process;

end architecture;
    ---------------------------------------------------------------------------
```

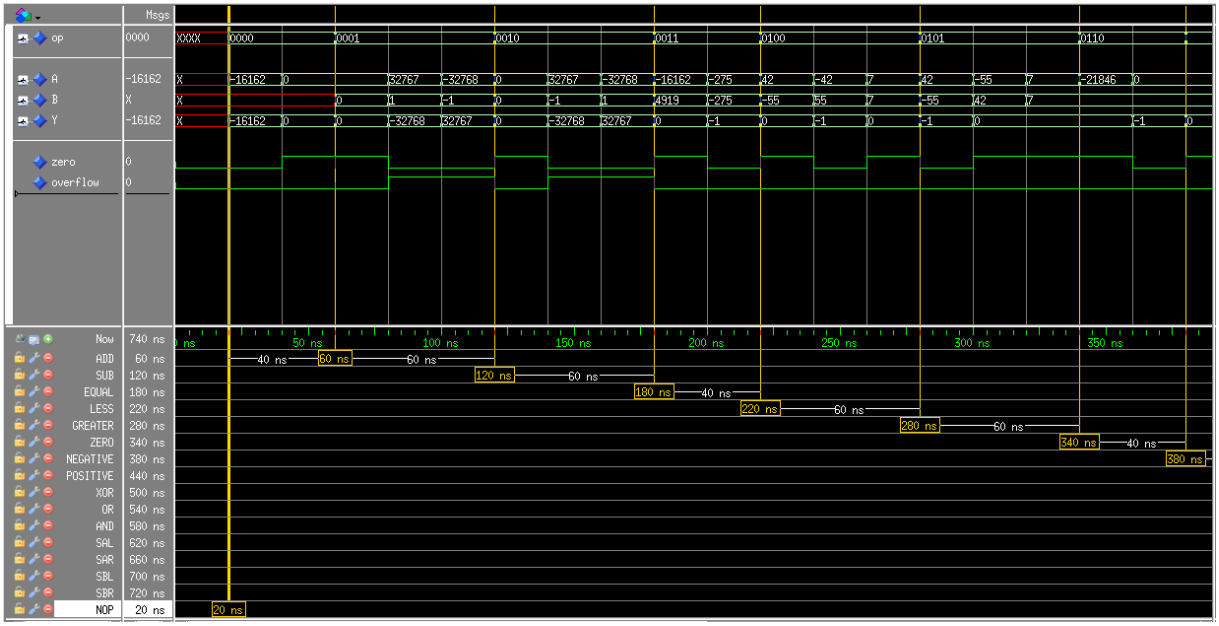


Figura 17 - Simulação da ULA (1). Fonte: o autor (2018).

Figura 18 - Simulação da ULA (2). Fonte: o autor (2018).

4.1.4 - Pilha LIFO

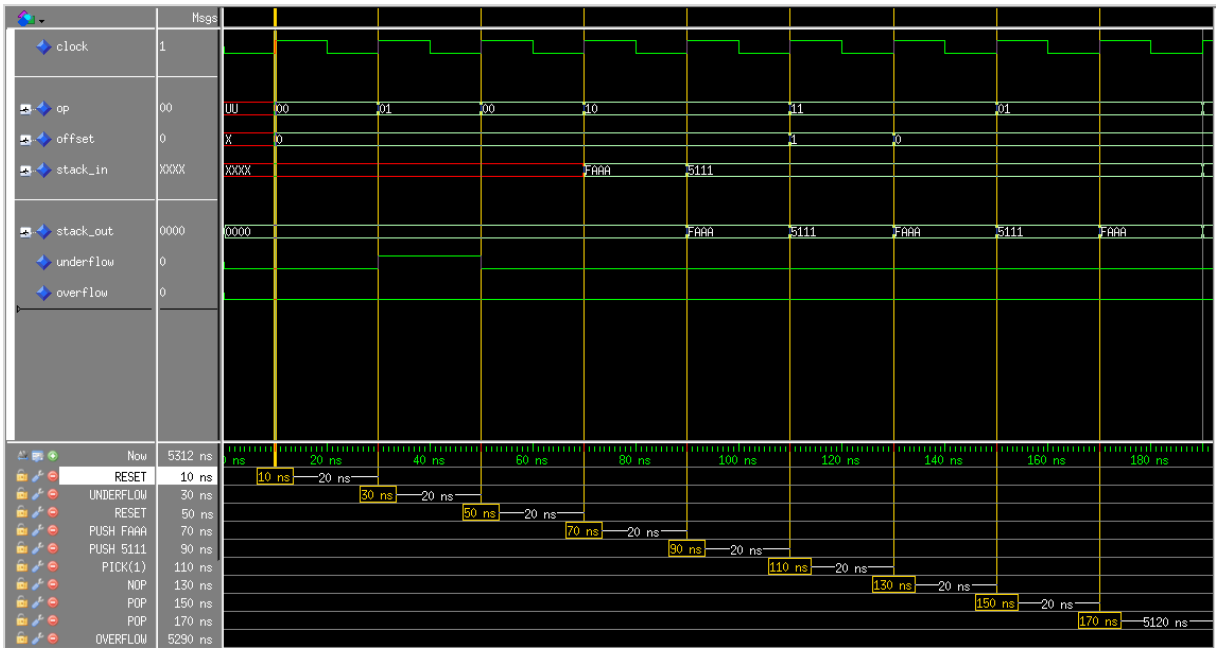
Para testar o comportamento da pilha foram executadas sequências das suas quatro operações básicas (incluindo leitura de elementos abaixo do topo utilizando o *offset* disponível), sempre avaliando o valor da saída de dados após o início do período de relógio seguinte.

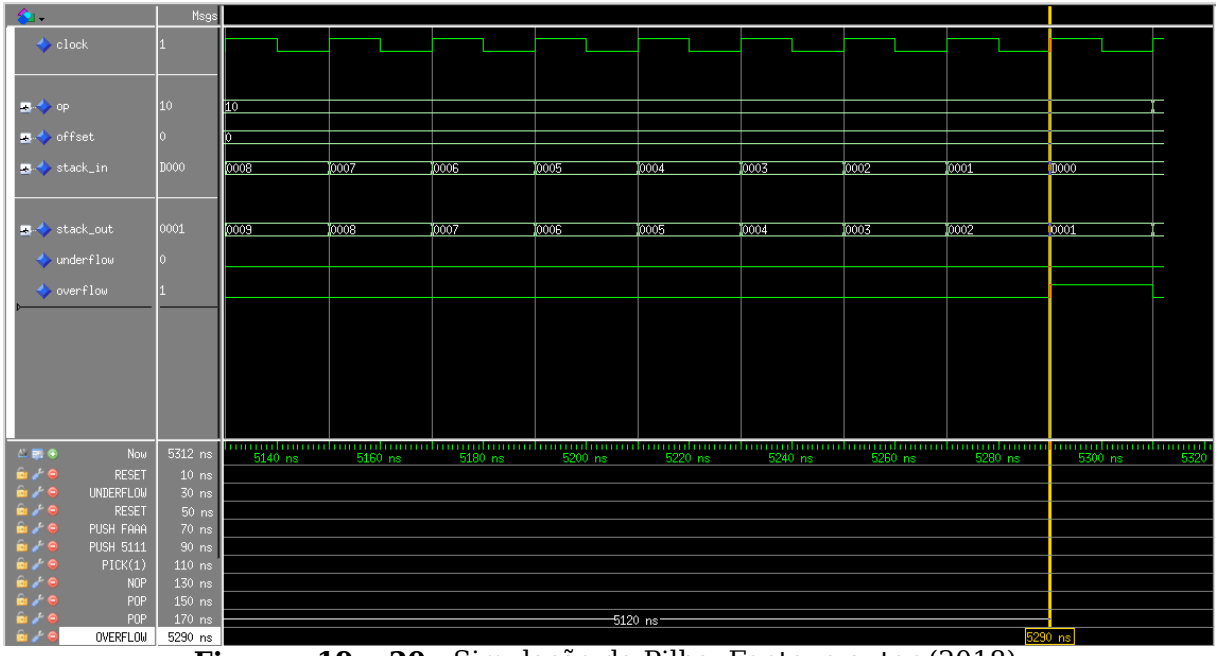
Além disso, para verificar o funcionamento devido dos indicadores de erro de *stack*, propositalmente causou-se um *underflow* e um *overflow* da maneira descrita no Capítulo 1.

```
    ----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;     -- type conversion
use ieee.math_real.all;       -- log2 & ceil


entity TB_Stack is
end entity;


architecture testbench of TB_Stack is        -- default
       -- DUT INTERFACE --
       component LIFO_Stack is       -- Push-down LIFO Stack
              generic (
                     WORD: positive := 16;
                     ADDR: positive := 8           -- address vector size
              );
              port (
                     -- CLOCK --
                     clock: in std_logic;
```

```
		-- CONTROL --
		op: in std_logic_vector(1 downto 0);
		offset: in std_logic_vector(ADDR-1 downto 0);

		-- DATA --
		stack_in: in std_logic_vector(WORD-1 downto 0);
		stack_out: out std_logic_vector(WORD-1 downto 0);

		-- STACK ERRORS --
		overflow, underflow: out std_logic
	);
end component;


-- CONSTANTS --
constant T: time := 20 ns;
constant LAG: time := 12 ns;

constant N: natural := 16;
constant ADDR: positive := 8;

-- opcodes
constant STACK_RESET: std_logic_vector(1 downto 0) :=	"00";
constant STACK_POP: std_logic_vector(1 downto 0) :=	"01";
constant STACK_PUSH: std_logic_vector(1 downto 0) :=	"10";
constant STACK_NOP: std_logic_vector(1 downto 0) :=	"11";


-- SIGNALS --
signal clock, overflow, underflow: std_logic;
signal op: std_logic_vector(1 downto 0);
signal offset: std_logic_vector(ADDR-1 downto 0);
signal stack_in, stack_out: std_logic_vector(N-1 downto 0);


-- TESTING --
begin
	UUT: LIFO_Stack
		generic map (
			WORD => N,
			ADDR => ADDR
		)
		port map (
			clock => clock,
			op => op,
			offset => offset,
			stack_in => stack_in,
			stack_out => stack_out,
			overflow => overflow,
			underflow => underflow
		);

	CLK: process is		-- 50% duty
		begin
			clock <= '0';
			wait for (T/2);
			clock <= '1';
			wait for (T/2);
	end process;

	STIMULUS: process is
		begin
			wait until rising_edge(clock);

			op <= STACK_RESET;
			offset<= x"00";
			wait for LAG;
			assert ((overflow='0') and (underflow='0'))
				report "Reported false Stack error"
				severity ERROR;
			wait until rising_edge(clock);

			op <= STACK_POP;
			wait for LAG;
			assert ((overflow='0') and (underflow='1'))
```

```
                report "Didn't report Stack underflow"
                severity ERROR;
            wait until rising_edge(clock);

            op <= STACK_RESET;
            wait until rising_edge(clock);

            op <= STACK_PUSH;
            stack_in <= x"FAAA";
            wait for LAG;
            assert ((overflow='0') and (underflow='0'))
                report "Reported false Stack error"
                severity ERROR;
            wait until rising_edge(clock);

            op <= STACK_PUSH;
            stack_in <= x"5111";
            wait for LAG;
            assert (stack_out=x"FAAA")
                report "Wrong TOS after last PUSH"
                severity ERROR;
            wait until rising_edge(clock);

            op <= STACK_NOP;
            offset <= std_logic_vector(to_unsigned(1, ADDR));
            wait for LAG;
            assert (stack_out=x"5111")
                report "Wrong TOS after last PUSH"
                severity ERROR;
            wait until rising_edge(clock);

            op <= STACK_NOP;
            offset <= x"00";
            wait for LAG;
            assert (stack_out=x"FAAA")
              report "Wrong value read on Stack after PICK(1)"
              severity ERROR;
            wait until rising_edge(clock);

            op <= STACK_POP;
            wait for LAG;
            assert (stack_out=x"5111")
                report "Wrong TOS value"
                severity ERROR;
            wait until rising_edge(clock);

            op <= STACK_POP;
            wait for LAG;
            assert (stack_out=x"FAAA")
                report "Wrong TOS value read after last POP"
                severity ERROR;
            wait until rising_edge(clock);

            -- push until overfloing
            for i in (2**ADDR)-1 downto 1 loop
              op <= STACK_PUSH;
              stack_in <= std_logic_vector(to_unsigned(i, N));
              wait until rising_edge(clock);
            end loop;

            op <= STACK_PUSH;
            stack_in <= x"D000";
            wait for LAG;
            assert ((overflow='1') and (underflow='0'))
                report "Didn't report Stack overflow"
                severity ERROR;

            -- TOTAL TIME : (255+10)*T + LAG
        end process;

end architecture;
```





Figuras 19 e 20 - Simulação da Pilha. Fonte: o autor (2018).

4.2 Validação do Bloco de Controle

Almejando a validação do Bloco de Controle, foram induzidos sequencialmente cada um dos estados da FSM utilizando a entrada de instrução para em seguida verificar os valores lidos nas saídas relevantes

a cada ciclo. Valores de referência usados no *testbench* seguem a Tabela 5 e as convenções de codificação definidas no Capítulo 2.

```
      ----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;


entity TB_Control is
end entity;


architecture testbench_truthtable of TB_Control is  -- default
       -- DUT INTERFACE --
       component S4PU_Control is     -- S4PU Control Unit
              generic (ARCH: positive := 16);      -- considered 16-bit
              port (
                     -- CLOCK --
                     clock: in std_logic;

                     -- EXTERNAL INPUTS --
                     reset_n: in std_logic;-- active low, synchronous
                     mode: in std_logic;

                     -- STATUS --
                     instruction: in std_logic_vector(ARCH-1 downto 0);
                     alu_zero: in std_logic;
                     i_rs_overflow, i_rs_underflow: in std_logic;
                     i_ds_overflow, i_ds_underflow: in std_logic;

                     -- COMMANDS --
                     ds_op: out std_logic_vector(1 downto 0);
                     sel_ds_in: out std_logic;
                     ds_pick: out std_logic;
                     rs_op: out std_logic_vector(1 downto 0);
                     sel_rs_in: out std_logic;
                     alu_op: out std_logic_vector(3 downto 0);
                     sel_alu_A: out std_logic_vector(2 downto 0);
                     tos_load: out std_logic;
                     pc_load: out std_logic;
                     sel_pc_D: out std_logic_vector(2 downto 0);
                     sel_addr: out std_logic_vector(2 downto 0);
                     cir_load: out std_logic;
                     sel_inst: out std_logic;

                     -- EXTERNAL OUTPUTS --
                     o_rs_overflow, o_rs_underflow: out std_logic;
                     o_ds_overflow, o_ds_underflow: out std_logic;
                     read, write: out std_logic
              );
       end component;


       -- CONSTANTS --
       constant T: time := 20 ns;
       constant LAG: time := 10 ns;

       -- instruction opcodes
       constant c_CALL: std_logic_vector(15 downto 0) :=          x"0000";
       constant c_NOP: std_logic_vector(15 downto 0) :=           x"8001";
       constant c_LOAD: std_logic_vector(15 downto 0) :=          x"8003";
       constant c_STORE: std_logic_vector(15 downto 0) :=         x"8005";
       constant c_IF: std_logic_vector(15 downto 0) :=            x"8007";
       constant c_BRANCH: std_logic_vector(15 downto 0) :=        x"8009";
       constant c_RET: std_logic_vector(15 downto 0) :=           x"800D";
       constant c_DROP: std_logic_vector(15 downto 0) :=          x"800F";
       constant c_LIT: std_logic_vector(15 downto 0) :=           x"8011";
       constant c_PICK: std_logic_vector(15 downto 0) :=          x"8013";
       constant c_TO_R: std_logic_vector(15 downto 0) :=          x"8015";
       constant c_R_FROM: std_logic_vector(15 downto 0) :=        x"8017";
       constant c_ALU_NOT: std_logic_vector(15 downto 0) :=       x"8019";
```

```
constant c_ALU_OR: std_logic_vector(15 downto 0) :=        x"801B";
constant c_ALU_AND: std_logic_vector(15 downto 0) :=       x"801D";
constant c_ALU_XOR: std_logic_vector(15 downto 0) :=       x"801F";
constant c_ALU_ADD: std_logic_vector(15 downto 0) :=       x"8021";
constant c_ALU_SUB: std_logic_vector(15 downto 0) :=       x"8023";
constant c_ALU_INC: std_logic_vector(15 downto 0) :=       x"8025";
constant c_ALU_DEC: std_logic_vector(15 downto 0) :=       x"8027";
constant c_ALU_CET: std_logic_vector(15 downto 0) :=       x"8029";
constant c_ALU_CLT: std_logic_vector(15 downto 0) :=       x"802B";
constant c_ALU_CGT: std_logic_vector(15 downto 0) :=       x"802D";
constant c_ALU_ZER: std_logic_vector(15 downto 0) :=       x"8035";
constant c_ALU_NEG: std_logic_vector(15 downto 0) :=       x"8037";
constant c_ALU_POS: std_logic_vector(15 downto 0) :=       x"8039";
constant c_ALU_SAL: std_logic_vector(15 downto 0) :=       x"8041";
constant c_ALU_SAR: std_logic_vector(15 downto 0) :=       x"8043";
constant c_DUP: std_logic_vector(15 downto 0) :=           x"804D";
constant c_SWAP: std_logic_vector(15 downto 0) :=          x"804F";
constant c_ALU_SBL: std_logic_vector(15 downto 0) :=       x"8051";
constant c_ALU_SBR: std_logic_vector(15 downto 0) :=       x"8053";

-- stack opcodes
constant STACK_RESET: std_logic_vector(1 downto 0) :=      "00";
constant STACK_POP: std_logic_vector(1 downto 0) :=        "01";
constant STACK_PUSH: std_logic_vector(1 downto 0) :=       "10";
constant STACK_NOP: std_logic_vector(1 downto 0) :=        "11";

-- alu opcodes
constant ALU_OP_NOP: std_logic_vector(3 downto 0) :=       "0000";
constant ALU_OP_ADD: std_logic_vector(3 downto 0) :=       "0001";
constant ALU_OP_SUB: std_logic_vector(3 downto 0) :=       "0010";
constant ALU_OP_CET: std_logic_vector(3 downto 0) :=       "0011";
constant ALU_OP_CLT: std_logic_vector(3 downto 0) :=       "0100";
constant ALU_OP_CGT: std_logic_vector(3 downto 0) :=       "0101";
constant ALU_OP_ZER: std_logic_vector(3 downto 0) :=       "0110";
constant ALU_OP_NEG: std_logic_vector(3 downto 0) :=       "0111";
constant ALU_OP_POS: std_logic_vector(3 downto 0) :=       "1000";
constant ALU_OP_XOR: std_logic_vector(3 downto 0) :=       "1001";
constant ALU_OP_OR: std_logic_vector(3 downto 0) :=        "1010";
constant ALU_OP_AND: std_logic_vector(3 downto 0) :=       "1011";
constant ALU_OP_SAL: std_logic_vector(3 downto 0) :=       "1100";
constant ALU_OP_SAR: std_logic_vector(3 downto 0) :=       "1101";
constant ALU_OP_SBL: std_logic_vector(3 downto 0) :=       "1110";
constant ALU_OP_SBR: std_logic_vector(3 downto 0) :=       "1111";


-- SIGNALS --
signal clock, reset_n, mode, alu_zero,
       sel_ds_in, ds_pick, sel_rs_in,
       tos_load, pc_load, cir_load, sel_inst,
       o_rs_overflow, o_rs_underflow,
       o_ds_overflow, o_ds_underflow,
       read, write: std_logic;

signal instruction: std_logic_vector(15 downto 0);

signal ds_op, rs_op: std_logic_vector(1 downto 0);

signal alu_op: std_logic_vector(3 downto 0);

signal sel_alu_A, sel_pc_D, sel_addr: std_logic_vector(2 downto 0);


-- TESTING --
begin
       UUT: S4PU_Control
              generic map (ARCH => 16)
              port map (
                     clock => clock,

                     reset_n => reset_n,
                     mode => mode,

                     instruction => instruction,
                     alu_zero => alu_zero,
                     i_rs_overflow => '-',
```

```
                i_rs_underflow => '-',
                i_ds_overflow => '-',
                i_ds_underflow => '-',

                ds_op => ds_op,
                sel_ds_in => sel_ds_in,
                ds_pick => ds_pick,
                rs_op => rs_op,
                sel_rs_in => sel_rs_in,
                alu_op => alu_op,
                sel_alu_A => sel_alu_A,
                tos_load => tos_load,
                pc_load => pc_load,
                sel_pc_D => sel_pc_D,
                sel_addr => sel_addr,
                cir_load => cir_load,
                sel_inst => sel_inst,

                o_rs_overflow => o_rs_overflow,
                o_rs_underflow => o_rs_underflow,
                o_ds_overflow => o_ds_overflow,
                o_ds_underflow => o_ds_underflow,
                read => read,
                write => write
        );

CLK: process is       -- 50% duty
        begin
                clock <= '0';
                wait for (T/2);
                clock <= '1';
                wait for (T/2);
end process;

STIMULUS: process is
        begin
                wait until rising_edge(clock);

                reset_n <= '0';
                mode <= '1';
                wait until rising_edge(clock);
                wait for LAG;
                assert ((ds_op=STACK_RESET) and (ds_pick='0') and
                        (rs_op=STACK_RESET) and
                        (alu_op=ALU_OP_NOP) and (sel_alu_A="000") and
                        (pc_load='1') and (sel_pc_D="001") and (write='0'))
                                report "Dirty state: RESET .1"
                                severity ERROR;

                reset_n <= '0';
                mode <= '0';
                wait until rising_edge(clock);
                wait for LAG;
                assert ((ds_op=STACK_RESET) and (ds_pick='0') and
                        (rs_op=STACK_RESET) and
                        (alu_op=ALU_OP_NOP) and (sel_alu_A="000") and
                        (pc_load='1') and (sel_pc_D="000") and (write='0'))
                                report "Dirty state: RESET .0"
                                severity ERROR;

                reset_n <= '1';
                wait until rising_edge(clock);
                wait for LAG;
                assert ((ds_op=STACK_NOP) and (ds_pick='0') and
                        (rs_op=STACK_NOP) and
                        (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
                        (pc_load='1') and (sel_pc_D="010") and
                        (sel_addr="000") and (write='0') and (read='1'))
                                report "Dirty state: FETCH"
                                severity ERROR;
                wait until rising_edge(clock);

                instruction <= c_NOP;
                wait for LAG;
                assert ((ds_op=STACK_NOP) and (ds_pick='0') and
```

```
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: DECODE after FETCH"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_CALL;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
        (pc_load='0') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: NOP(DECODE) after DECODE"
                severity ERROR;
wait until rising_edge(clock);

wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_PUSH) and (sel_rs_in='0') and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
        (pc_load='1') and (sel_pc_D="011") and
        (sel_addr="011") and (write='0') and (read='1'))
                report "Dirty state: CALL after DECODE"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_LOAD;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: DECODE after CALL"
                severity ERROR;
wait until rising_edge(clock);

wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
        (pc_load='0') and
        (sel_addr="001") and (write='0') and (read='1') and
        (cir_load='1'))
                report "Dirty state: LOAD_0 after DECODE"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_STORE;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="110") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='0') and (sel_inst='1'))
                report "Dirty state: LOAD_1"
                severity ERROR;
wait until rising_edge(clock);

wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
        (pc_load='0') and
        (sel_addr="001") and (write='1') and (read='0') and
        (cir_load='1'))
                report "Dirty state: STORE_0 after LOAD"
                severity ERROR;
```

```
wait until rising_edge(clock);

instruction <= c_IF;
alu_zero <= '1';
wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
       (rs_op=STACK_NOP) and
       (alu_op=ALU_OP_NOP) and (sel_alu_A="100") and
       (pc_load='1') and (sel_pc_D="010") and
       (sel_addr="000") and (write='0') and (read='1') and
       (cir_load='0') and (sel_inst='1'))
              report "Dirty state: STORE_1"
              severity ERROR;
wait until rising_edge(clock);

wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
       (rs_op=STACK_NOP) and
       (alu_op=ALU_OP_NOP) and (sel_alu_A="100") and
       (pc_load='1') and (sel_pc_D="100") and
       (sel_addr="010") and (write='0') and (read='1'))
              report "Dirty state: IF_FALSE after STORE"
              severity ERROR;
wait until rising_edge(clock);

instruction <= c_IF;
alu_zero <= '0';
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
       (rs_op=STACK_NOP) and
       (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
       (pc_load='1') and (sel_pc_D="010") and
       (sel_addr="000") and (write='0') and (read='1') and
       (cir_load='1') and (sel_inst='0'))
              report "Dirty state: DECODE after IF(false)"
              severity ERROR;
wait until rising_edge(clock);

wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
       (rs_op=STACK_NOP) and
       (alu_op=ALU_OP_NOP) and (sel_alu_A="100") and
       (pc_load='1') and (sel_pc_D="010") and
       (sel_addr="000") and (write='0') and (read='1'))
              report "Dirty state: IF_TRUE after DECODE"
              severity ERROR;
wait until rising_edge(clock);

instruction <= c_BRANCH;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
       (rs_op=STACK_NOP) and
       (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
       (pc_load='1') and (sel_pc_D="010") and
       (sel_addr="000") and (write='0') and (read='1') and
       (cir_load='1') and (sel_inst='0'))
              report "Dirty state: DECODE after IF(true)"
              severity ERROR;
wait until rising_edge(clock);

wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
       (rs_op=STACK_NOP) and
       (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
       (pc_load='1') and (sel_pc_D="100") and
       (sel_addr="010") and (write='0') and (read='1'))
              report "Dirty state: BRANCH after DECODE"
              severity ERROR;
wait until rising_edge(clock);

instruction <= c_RET;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
       (rs_op=STACK_NOP) and
       (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
```

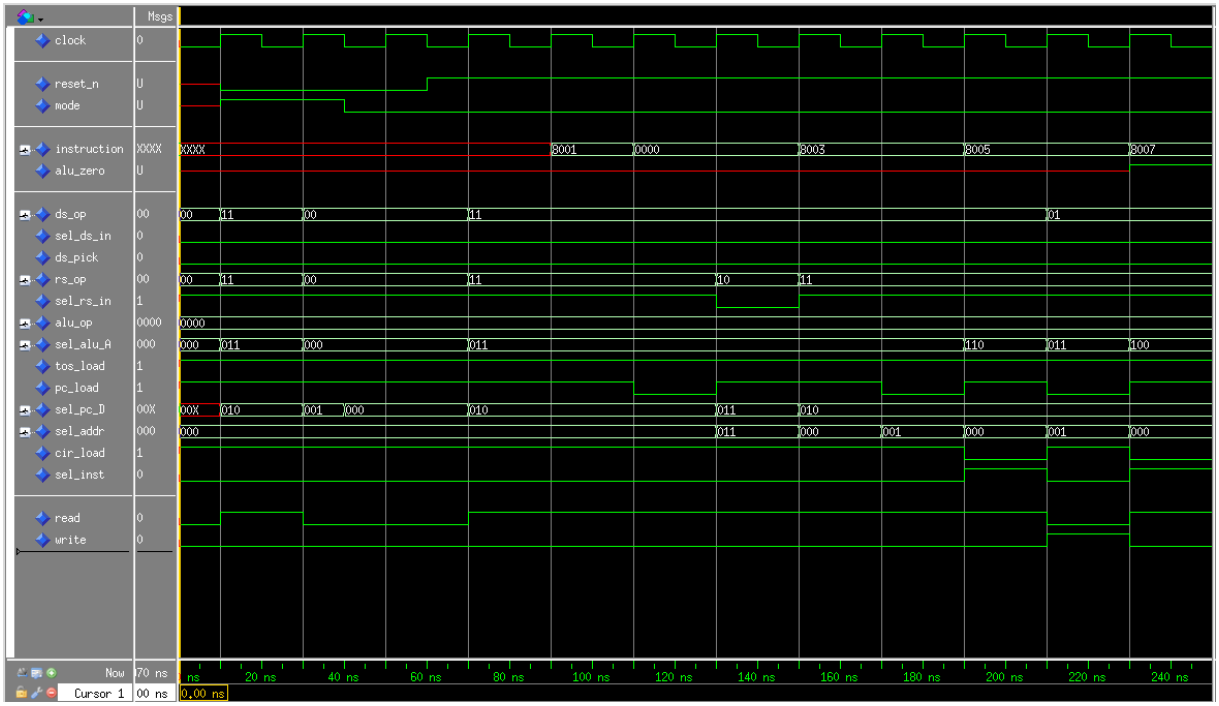
```
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: DECODE after BRANCH"
                severity ERROR;
wait until rising_edge(clock);

wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_POP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
        (pc_load='1') and (sel_pc_D="101") and
        (sel_addr="100") and (write='0') and (read='1'))
                report "Dirty state: RET after DECODE"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_DROP;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: DECODE after RET"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_LIT;
wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="100") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: DROP after DECODE"
                severity ERROR;
wait until rising_edge(clock);

wait for LAG;
assert ((ds_op=STACK_PUSH) and (sel_ds_in='0') and
        (ds_pick='0') and (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="110") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1'))
                report "Dirty state: LIT after DROP"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_PICK;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: DECODE after LIT"
                severity ERROR;
wait until rising_edge(clock);

wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='1') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
        (pc_load='0') and (write='0') and (cir_load='1'))
                report "Dirty state: PICK_0 after DECODE"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_TO_R;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
```

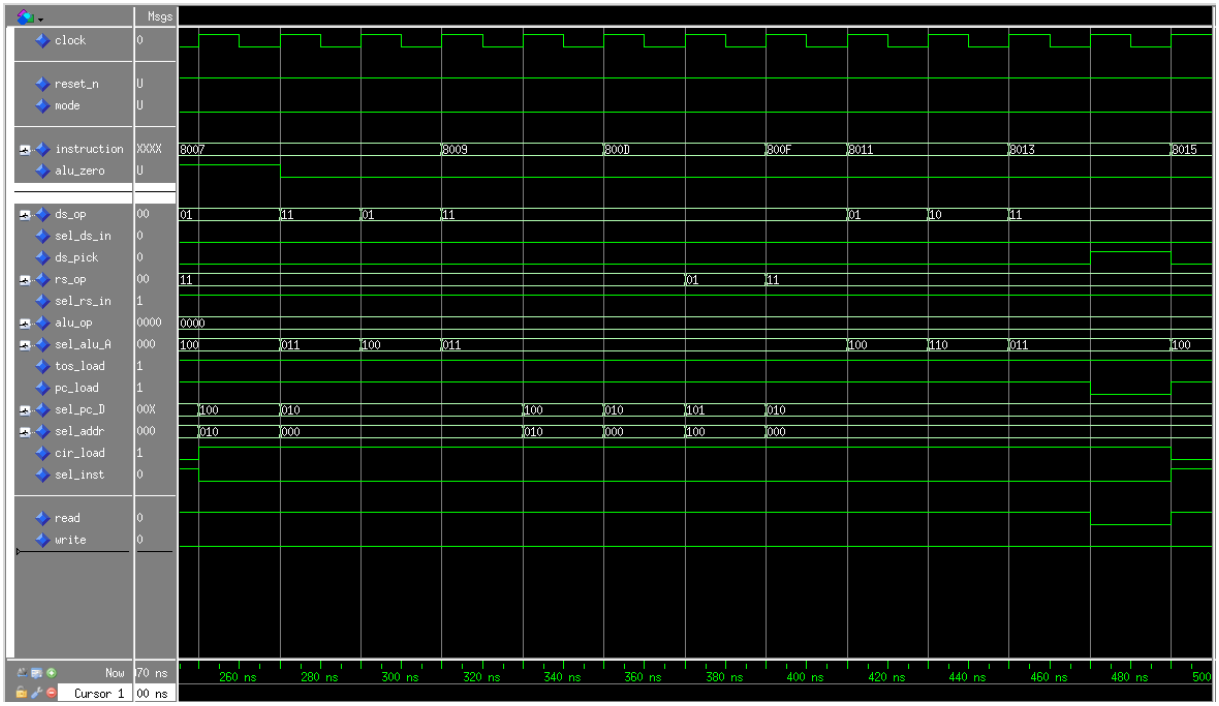
```
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="100") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='0') and (sel_inst='1'))
               report "Dirty state: PICK_1"
               severity ERROR;
wait until rising_edge(clock);

instruction <= c_R_FROM;
wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
        (rs_op=STACK_PUSH) and (sel_rs_in='1') and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="100") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
               report "Dirty state: TO_R after PICK"
               severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_NOT;
wait for LAG;
assert ((ds_op=STACK_PUSH) and (sel_ds_in='0') and
        (ds_pick='0') and (rs_op=STACK_POP) and
        (alu_op=ALU_OP_NOP) and (sel_alu_A="101") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
               report "Dirty state: R_FROM after TO_R"
               severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_OR;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_SUB) and (sel_alu_A="010") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
               report "Dirty state: ALU_NOT after R_FROM"
               severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_AND;
wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_OR) and (sel_alu_A="100") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
               report "Dirty state: ALU_OR after ALU_NOT"
               severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_XOR;
wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_AND) and (sel_alu_A="100") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
               report "Dirty state: ALU_AND after ALU_OR"
               severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_ADD;
wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_XOR) and (sel_alu_A="100") and
        (pc_load='1') and (sel_pc_D="010") and
```

```
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_XOR after ALU_AND"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_SUB;
wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_ADD) and (sel_alu_A="100") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_ADD after ALU_XOR"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_INC;
wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_SUB) and (sel_alu_A="100") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_SUB after ALU_ADD"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_DEC;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_ADD) and (sel_alu_A="001") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_INC after ALU_SUB"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_CET;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_ADD) and (sel_alu_A="010") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_DEC after ALU_INC"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_CLT;
wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_CET) and (sel_alu_A="100") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_CET after ALU_DEC"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_CGT;
wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_CLT) and (sel_alu_A="100") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_CLT after ALU_CET"
```

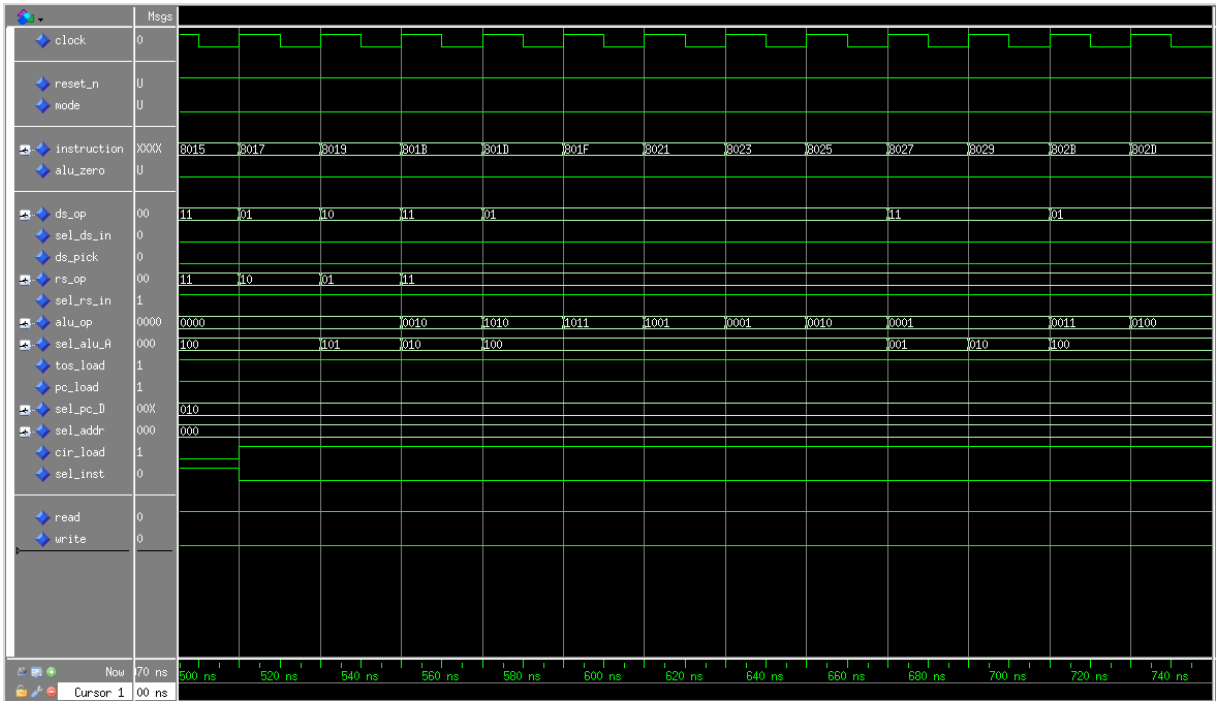
```
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_ZER;
wait for LAG;
assert ((ds_op=STACK_POP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_CGT) and (sel_alu_A="100") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_CGT after ALU_CLT"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_NEG;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_ZER) and (sel_alu_A="011") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_ZER after ALU_CGT"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_POS;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_NEG) and (sel_alu_A="011") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_NEG after ALU_ZER"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_SAL;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_POS) and (sel_alu_A="011") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_POS after ALU_NEG"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_SAR;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_SAL) and (sel_alu_A="011") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_SAL after ALU_POS"
                severity ERROR;
wait until rising_edge(clock);

instruction <= c_ALU_SBL;
wait for LAG;
assert ((ds_op=STACK_NOP) and (ds_pick='0') and
        (rs_op=STACK_NOP) and
        (alu_op=ALU_OP_SAR) and (sel_alu_A="011") and
        (pc_load='1') and (sel_pc_D="010") and
        (sel_addr="000") and (write='0') and (read='1') and
        (cir_load='1') and (sel_inst='0'))
                report "Dirty state: ALU_SAR after ALU_SAL"
                severity ERROR;
wait until rising_edge(clock);
```

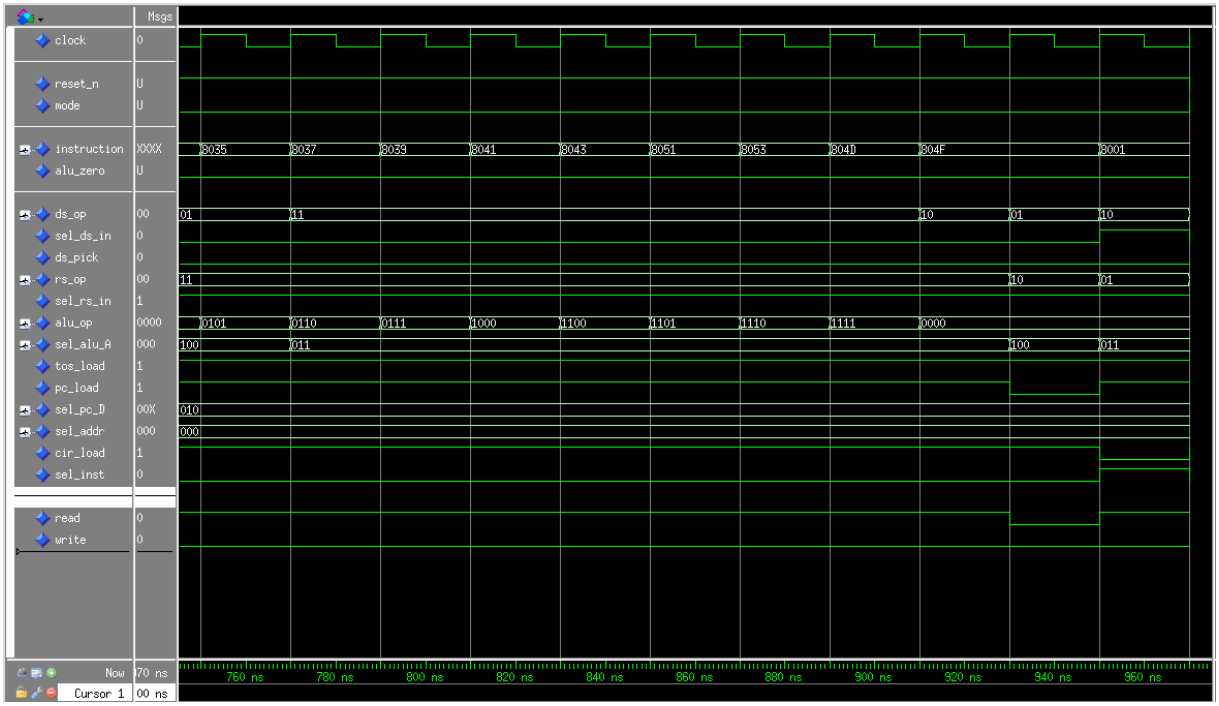
```
                                instruction <= c_ALU_SBR;
                                wait for LAG;
                                assert ((ds_op=STACK_NOP) and (ds_pick='0') and
                                       (rs_op=STACK_NOP) and
                                       (alu_op=ALU_OP_SBL) and (sel_alu_A="011") and
                                       (pc_load='1') and (sel_pc_D="010") and
                                       (sel_addr="000") and (write='0') and (read='1') and
                                       (cir_load='1') and (sel_inst='0'))
                                              report "Dirty state: ALU_SBL after ALU_SAR"
                                              severity ERROR;
                                wait until rising_edge(clock);

                                instruction <= c_DUP;
                                wait for LAG;
                                assert ((ds_op=STACK_NOP) and (ds_pick='0') and
                                       (rs_op=STACK_NOP) and
                                       (alu_op=ALU_OP_SBR) and (sel_alu_A="011") and
                                       (pc_load='1') and (sel_pc_D="010") and
                                       (sel_addr="000") and (write='0') and (read='1') and
                                       (cir_load='1') and (sel_inst='0'))
                                              report "Dirty state: ALU_SBR after ALU_SBL"
                                              severity ERROR;
                                wait until rising_edge(clock);

                                instruction <= c_SWAP;
                                wait for LAG;
                                assert ((ds_op=STACK_PUSH) and (sel_ds_in='0') and
                                       (ds_pick='0') and (rs_op=STACK_NOP) and
                                       (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
                                       (pc_load='1') and (sel_pc_D="010") and
                                       (sel_addr="000") and (write='0') and (read='1') and
                                       (cir_load='1') and (sel_inst='0'))
                                              report "Dirty state: DUP after ALU_SBR"
                                              severity ERROR;
                                wait until rising_edge(clock);

                                wait for LAG;
                                assert ((ds_op=STACK_POP) and (ds_pick='0') and
                                       (rs_op=STACK_PUSH) and (sel_rs_in='1') and
                                       (alu_op=ALU_OP_NOP) and (sel_alu_A="100") and
                                       (pc_load='0') and (write='0') and (cir_load='1'))
                                              report "Dirty state: SWAP_0 after DUP"
                                              severity ERROR;
                                wait until rising_edge(clock);

                                instruction <= c_NOP;
                                wait for LAG;
                                assert ((ds_op=STACK_PUSH) and (sel_ds_in='1') and
                                       (ds_pick='0') and (rs_op=STACK_POP) and
                                       (alu_op=ALU_OP_NOP) and (sel_alu_A="011") and
                                       (pc_load='1') and (sel_pc_D="010") and
                                       (sel_addr="000") and (write='0') and (read='1') and
                                       (cir_load='0') and (sel_inst='1'))
                                              report "Dirty state: SWAP_1"
                                              severity ERROR;

                                -- TOTAL TIME : 48*T + LAG
                end process;

end architecture;
      ---------------------------------------------------------------------
```





Figuras 21 e 22 - Simulação do Bloco de Controle (1 e 2). Fonte: o autor (2018).





Figuras 23 e 24 - Simulação do Bloco de Controle (3 e 4). Fonte: o autor (2018).

4.3 Validação da Unidade de Processamento

Havendo sido definida no início deste projeto a necessidade de uma memória externa para o processador, não há sentido testá-lo sem ela. Desta forma, **projetou-se uma interface que implementa a parte fixa do mapeamento de memória** dado na Tabela 1, utilizando um bloco de memória RAM como memória principal, outro de ROM para a memória de programa e fazendo a extensão dos demais endereços em sua interface externa. Destaca-se que a FPGA Cyclone II – na qual pretende-se realizar a prototipação - possui células de memória limitadas, levando à necessidade de reduzir o tamanho instanciado dos blocos RAM e ROM.

A validação do sistema se dá utilizando este componente que o instancia e iniciando o processador no 'modo programador' para que execute as instruções lidas na memória de programa sabendo que esta é carregada a partir de um arquivo de inicialização com uma rotina de teste montada a partir da linguagem *assembly* da máquina, ou seja, o subconjunto de Forth definido na Tabela 2.

O mapeamento de endereço da interface consiste no direcionamento dos sinais indo ao processador de cada uma das memórias instanciadas (ou ao exterior do componente, como uma extensão da interface) dependendo da faixa de valores em que o sinal de endereço se encontra a cada operação.

```
      ------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;     -- type conversion

entity S4PU_Daughterboard is
       generic (ARCH: positive := 16);      -- Architecture word size
       port (
              -- CLOCK --
              clock: in std_logic;

              -- CPU CONTROL --
              reset_n: in std_logic;-- active low, synchronous
              mode: in std_logic;

              -- MEMORY EXTENSION (Avalon spec.) --
              read, write: out std_logic;
              address: out std_logic_vector(ARCH-1 downto 0);
              readdata: in std_logic_vector(ARCH-1 downto 0);
              writedata: out std_logic_vector(ARCH-1 downto 0)
       );
end entity;
```

```
architecture embedded_v0 of S4PU_Daughterboard is   -- default
	-- COMPONENTS --
	component S4PU is     -- Simple Forth Processing Unit
		generic (ARCH: positive := 16);
		port (
			-- CLOCK --
			clock: in std_logic;

			-- SYSTEM CONTROL --
			reset_n: in std_logic;-- active low, synchronous
			mode: in std_logic;

			-- STACK ERRORS --
			rs_overflow, rs_underflow: out std_logic;
			ds_overflow, ds_underflow: out std_logic;

			-- PUPPET/SLAVE MEMORY (Avalon spec.) --
			read, write: out std_logic;
			address: out std_logic_vector(ARCH-1 downto 0);
			readdata: in std_logic_vector(ARCH-1 downto 0);
			writedata: out std_logic_vector(ARCH-1 downto 0)
		);
	end component;

	component main_ram IS -- Altera Quartus wizard-generated single-port RAM
		PORT
		(
			address : IN STD_LOGIC_VECTOR (13 DOWNTO 0);
			clock   : IN STD_LOGIC  := '1';
			data    : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
			wren    : IN STD_LOGIC ;
			q       : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
		);
	END component;

	component prog_rom IS -- Altera Quartus wizard-generated 1-port ROM ./memory/prog.mif
		PORT
		(
			address : IN STD_LOGIC_VECTOR (12 DOWNTO 0);
			clock   : IN STD_LOGIC  := '1';
			q       : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
		);
	end component;


	-- CONSTANT MEMORY RANGE MAPPING --
	constant MAIN_MEM_START: natural := 0;
	constant PROG_MEM_START: natural := 32768;
	constant EXT_MEM_START: natural := 41984;

	-- @XXX DE2 COULDN'T FIT THE DESIGN SO BOARD MEMORY WAS CUT IN HALF
	constant MAIN_MEM_END: natural := MAIN_MEM_START + 16383;
	constant PROG_MEM_END: natural := PROG_MEM_START + 4607;
	constant EXT_MEM_END: natural := EXT_MEM_START + 23551;

	-- SIGNALS --
	signal main_mem_wren,
		cpu_reset_n, cpu_read, cpu_write,
		rs_overflow, rs_underflow,
		ds_overflow, ds_underflow: std_logic;

	signal cpu_address, address_range,
		cpu_readdata, cpu_writedata,
		main_mem_address, main_mem_q,
		prog_mem_address, prog_mem_q: std_logic_vector(ARCH-1 downto 0);

	-- INTERNAL STATE --
	subtype InternalState is std_logic_vector(ARCH-1 downto 0);
	signal curr_state: InternalState;

	-- DESCRIPTION --
	begin
		-- @note fixed data'length and address'length on wizard-generated memory
		assert (ARCH >= 16)
```

```
                report "Architecture incompatible with instantiated memory"
                severity FAILURE;

        -- COMP. INSTANTIATION --
        CPU: S4PU
                generic map (ARCH => ARCH)
                port map (
                        clock => clock,

                        reset_n => cpu_reset_n,
                        mode => mode,

                        rs_overflow => rs_overflow,
                        rs_underflow => rs_underflow,
                        ds_overflow => ds_overflow,
                        ds_underflow => ds_underflow,

                        read => cpu_read,
                        write => cpu_write,
                        address => cpu_address,
                        readdata => cpu_readdata,
                        writedata => cpu_writedata
                );

        MAIN_MEMORY: main_ram
                port map (
                        address => main_mem_address(13 downto 0),
                        clock => clock,
                        data => cpu_writedata(15 downto 0),
                        wren => main_mem_wren,
                        q => main_mem_q(15 downto 0)
                );

        PROGRAM_MEMORY: prog_rom
                port map (
                        address => prog_mem_address(12 DOWNTO 0),
                        clock => clock,
                        q => prog_mem_q(15 DOWNTO 0)
                );


        -- BEHAVIOUR --
        writedata <= cpu_writedata;

        -- @note stack errors are treated as critical errors that reset the cpu
        cpu_reset_n <= reset_n and not(rs_overflow or rs_underflow or ds_overflow or
ds_underflow);

        -- address range conversion
        main_mem_address <= std_logic_vector(unsigned(cpu_address) - MAIN_MEM_START);
        prog_mem_address <= std_logic_vector(unsigned(cpu_address) - PROG_MEM_START);
        address <= std_logic_vector(unsigned(cpu_address) - EXT_MEM_START);

        -- memory mapping
        main_mem_wren <= '1' when cpu_write='1' and (unsigned(cpu_address) >=
MAIN_MEM_START) and (unsigned(cpu_address) <= MAIN_MEM_END)
                                else '0';

        read <= '1' when cpu_read='1' and (unsigned(cpu_address) >= EXT_MEM_START) and
(unsigned(cpu_address) <= EXT_MEM_END) else '0';
        write <= '1' when cpu_write='1' and (unsigned(cpu_address) >= EXT_MEM_START)
and (unsigned(cpu_address) <= EXT_MEM_END) else '0';

        -- address register
        ME: process (clock, reset_n) is
                begin
                        if (reset_n = '0') then
                                curr_state <= (others => '0');
                        elsif (rising_edge(clock)) then
                                curr_state <= cpu_address;
                        end if;
        end process;

        -- output logic
        address_range <= curr_state when cpu_write='1'
```

```
                              else cpu_address;   -- bypass register when not writing

            cpu_readdata <= main_mem_q when
                                    (unsigned(address_range) >= MAIN_MEM_START) and
                                    (unsigned(address_range) <= MAIN_MEM_END) else
                                prog_mem_q when (unsigned(address_range) >= PROG_MEM_START) and
                                    (unsigned(address_range) <= PROG_MEM_END) else
                                readdata when (unsigned(address_range) >= EXT_MEM_START) and
                                    (unsigned(address_range) <= EXT_MEM_END)
                                else (others => '1'); -- undefined

end architecture;
```

Para a simulação, criou-se um arquivo de estímulos simples para geração de clock, inicialização correta da máquina no modo programador e *feedback* dos valores de saída do algoritmo:

```
restart -f

force /clock 0 0ns, 1 10ns -r 20ns
force /reset_n 0 0ns, 1 15ns
force /mode 1 0ns

run @1814730ns

set t 0
for { set i 0 } { $i < 19 } { incr i } {
    echo $i "th EMIT"
    set t [scan [lindex [expr [searchlog -expr {cpu_write == '1'} [expr $t]ns]] 0] "%d"]
    set f [examine -time [expr $t]ns /cpu_writedata]
    echo $f
}
```

O conteúdo inicializado na memória consiste em um programa em *loop* com algumas sub-rotinas utilitárias, testes de aritmética e uma função recursiva para gerar números da sequência de Fibonacci. Resultados dos testes e do algoritmo são “escritos” na interface de memória e detectados pelo *script tcl* acima.

Segue, portanto, devidamente validado o funcionamento da Unidade de Processamento Simples com Arquitetura baseada em Pilhas, com memória reprogramável, conjunto de instruções com suporte a chamadas recursivas, desvios condicionais e comunicação com dispositivos externos através da interface de memória mapeada: conclui-se o projeto do processador como sistema digital.

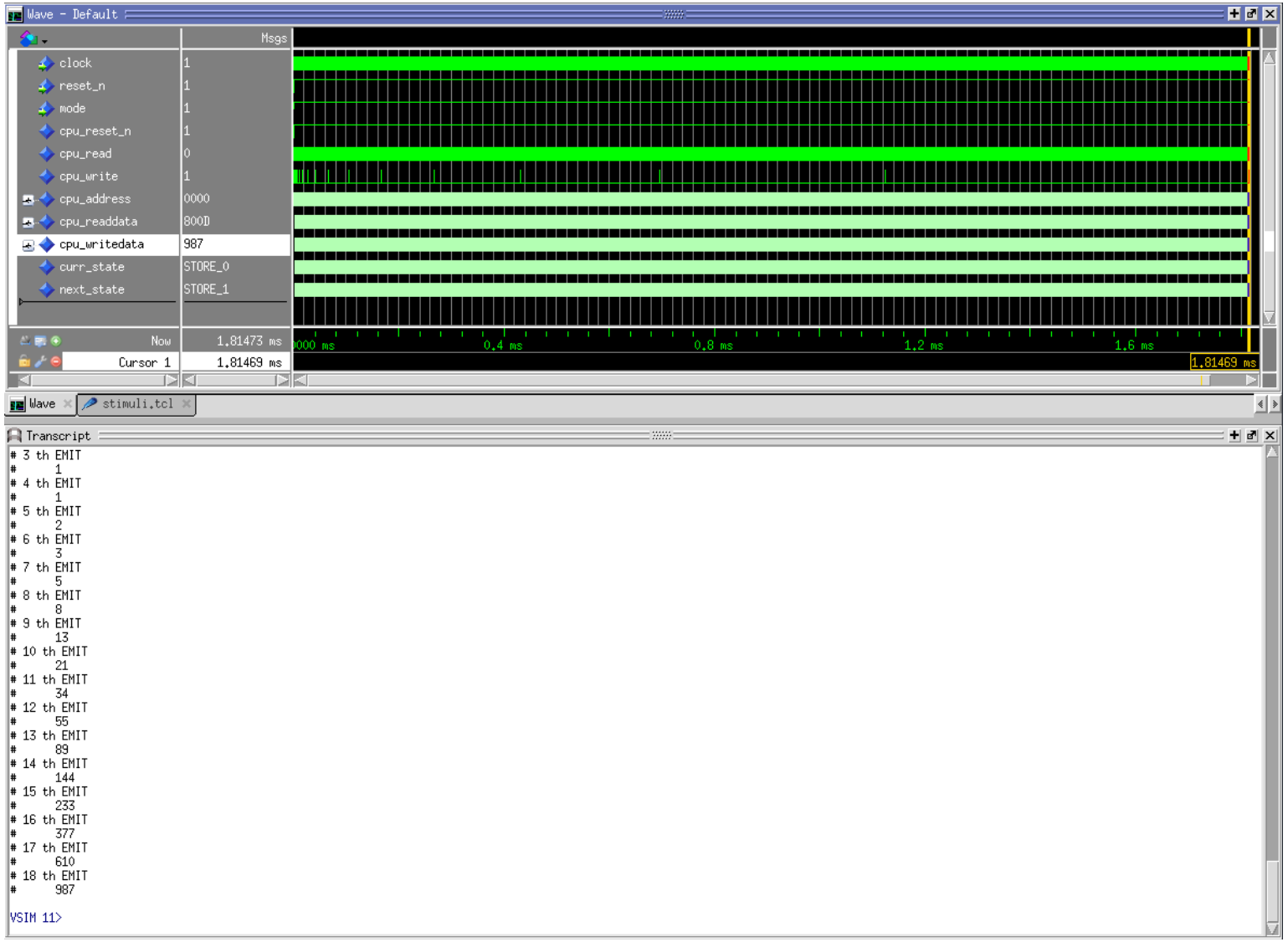


Figura 25 - Simulação do Sistema Digital. Valores enviados pelo processador são mostrados na janela *Transcript*, demonstrando os números de Fibonacci sendo gerados pelo algoritmo recursivo. Fonte: o autor (2018).

5 Conclusões

O projeto de uma CPU simples a nível RT permitiu **compreender melhor o funcionamento de um processador moderno** e a lógica por trás de técnicas empregadas para acesso a periféricos, controle de memória, convenções de programação, otimizações de desempenho, comunicação por barramentos, entre muitos outros aspectos estudados ao longo do processo.

Além disso, induziu a absorção de uma **nova maneira de pensar sequencialmente** dada na linguagem Forth e na arquitetura baseada em pilha; introduzindo ainda a necessidade de ferramentas para facilitar a programação do conjunto de instruções criado: **compiladores e *assemblers***.

O produto final, batizado *Simple Forth Processing Unit* (S4PU), sendo um processador de uso geral, é capaz de atuar em áreas diversas; e por ser uma máquina de pilha, apresenta-se como ótimo candidato em aplicações que precisem de boa performance combinada com *hardware* de baixa complexidade; a área de sistemas de controle embarcados torna-se um excelente exemplo por seus requisitos:

- **Tamanho e peso**: Relacionados principalmente com o tamanho da placa de circuito impresso, são reduzidos em processadores simples que utilizam poucos componentes externos e memórias pequenas.
- **Dissipação térmica**: O gasto energético de um circuito está relacionado com o número de transistores chaveando, e em máquinas de pilha de 16 bits esse valor é no mínimo duas vezes menor do que em processadores 32 bits com arquitetura baseada em registradores.
- **Robustez**: Em algumas condições de operação de controle embarcado, o sistema deve lidar com vibrações, impactos, temperaturas extremas ou até mesmo radiação. Para isso, normalmente recomenda-se manter a contagem de componentes

internos e de pinos a mais baixa possível, como é o caso em arquiteturas minimais.

- **Consistência de execução**: Processadores que utilizam técnicas mais avançadas para melhorar sua performance média (como *cache* de instruções, predição de desvios e *pipelining* fora de ordem) acabam por não poder garantir o tempo exato de um dado programa. A arquitetura baseada em pilhas se coloca como um meio termo que troca esse nível de otimização por consistência na execução de instruções.
- **Preço**: Pelos mesmos fatores mencionados anteriormente, máquinas de pilha obtêm vantagem no seu custo de produção.

(KOOPMAN, 1989)

Quanto ao desempenho do processador, entende-se que este pode variar dependendo do programa executado e **são necessários testes com *benchmarks* para obter suas reais métricas de performance**; mas existem muitas operações que podem ser realizadas em menos tempo do que, por exemplo, na arquitetura do processador MIPS multiciclo estudada na disciplina de Sistemas Digitais (INE5406), tendo em mente a diferença no número de ciclos por instrução:

Classe da Instrução	Ciclos por Instrução (CPI)	
	MIPS multiciclo (arquitetura de registradores)	**S4PU** com *pipelining* (arquitetura de pilhas)
Leitura de memória	5	2
Escrita na memória	4	2
Aritmética ou lógica	4	1
Desvio de controle	3	2

Tabela 6 – Comparação com o MIPS. Pode-se comparar o desempenho das arquiteturas através do número de ciclos devido à similaridade da especificação de uso da memória – um ciclo para leitura ou escrita – que levam-nas a períodos mínimos de *clock* parecidos.

(HENNESSY; PATTERSON, 2014)

Dito isso, existem ainda diversos aspectos do sistema nos quais pode-se trabalhar para otimizar performance e recursos; e melhorar compatibilidade e integração:

- Juntar as pilhas em uma mesma memória como considerado no capítulo 3;
- Otimizar o tamanho dos *stacks* ao encontrar o melhor termo entre economia de memória e prevenção de *overflow*;
- Substituir os registradores completos onde uma versão mais simples poderia ser utilizada;
- Projetar interfaces compatíveis com componentes integrados disponíveis no mercado (para a ULA, por exemplo);
- Alterar a codificação dos sinais de controle para a otimização lógica dos códigos de operação (utilizando algoritmos como o *Espresso* de Berkeley);
- Adicionar mais instruções e tipos numéricos tornando a arquitetura compatível com o padrão internacional da linguagem Forth;
- Flexibilizar a especificação de atraso de operações na memória externa;
- Tornar a interface do sistema compatível com barramentos modernos utilizando lógica *tri-state*;
- Possibilitar interrupções do processador e aplicar tratamento de erros como *overflow* e *underflow*;
- Implementar outras técnicas de aumento de desempenho da CPU.

Por fim, pode-se até mesmo imaginar o sistema digital aqui projetado (e futuramente incrementado) sendo prototipado fisicamente na forma de um chip integrado visando **aplicações na área de computação embarcada**; ou mesmo no **ensino**, caso o projeto venha a tornar-se parte do domínio público.

Referências

KOOPMAN, Philip J., Jr. **Stack Computers:** the new wave. Pittsburgh, USA: Ellis Horwood, 1989. Disponível em <https://users.ece.cmu.edu/~koopman/stack_computers/index.html>. Acesso em: 01 de setembro de 2018.

HENNESSY John L; PATTERSON, David A. **Computer Organization and Design: the Hardware/Software Interface**. 5ª edição. Oxford: Morgan Kaufmann, 2014.

CHU, Pong P. **RTL Hardware Design Using VHDL**: Coding for Efficiency, Portability, and Scalability. Hoboken: John Wiley & Sons, 2006.

FORTH. **ANS Forth Standard**. Revisão de 2013. The Forth Standardisation Committee, 1994. Disponível em <http://forth-standard.org/standard/core>. Acesso em: 01 de setembro de 2018.

INTEL. **Avalon Interface Specifications**. Disponível em <https://www.intel.com/content/www/us/en/programmable/documentation/nik1412467993397.html#nik1412467940376>. Acesso em: 06 de setembro de 2018.

FSF. **Gforth**. v0.7.0. Free Software Foundation, 2008. Disponível em <http://www.complang.tuwien.ac.at/forth/gforth/Docs-html/index.html>. Acesso em: 01 de setembro de 2018.

BRODIE, Leo. **Starting FORTH**. 2ª edição. Hermosa Beach, California: Forth Inc; 1981. Disponível em <https://www.forth.com/starting-forth/>. Acesso em: 01 de setembro de 2018.