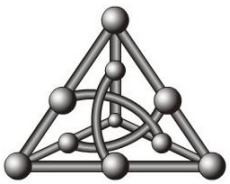
Descrição do Trabalho Prático da Disciplina de Laboratório de Hardware

Prof. Renan Albuquerque Marks

Versão 1.2

Resumo

Este documento contém todas as informações necessárias e suficientes para o desenvolvimento do trabalho prático da disciplina de Laboratório de Hardware. Este trabalho será desenvolvido ao longo deste semestre letivo e constituirá como parte da nota final da disciplina de Laboratório de Hardware. É fortemente recomendado que os estudantes acessem com frequência este documento para esclarecer possíveis dúvidas, estar ciente do cronograma e estar a par de possíveis atualizações/alterações no trabalho.



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul

Histórico de versões

Versão 1.2

- Correção: Firmware deve ser carregado na **`IMEM`** e não na **`DMEM`**.
- Sinais `data_out`/`data_in` da entidade `mem` e `cpu` agora trabalham com 4 bytes de dados em vez de 1 byte.

Versão 1.1

- Pequenos ajustes nas listagens dos códigos das entidades;
- Ajustes nas descrições de funcionamento das entidades `cpu` e `codec`;

Versão 1.0

- Versão inicial;

1 Objetivo

Este trabalho, doravante chamado de "Projeto", tem como objetivo final a modelagem de um hardware denominado SoC : *System on a Chip*. Um SoC é um dispositivo que contém diversos sub-componentes necessários para o funcionamento de um sistema computacional completo, tais como Processador, Memória(s) Principal(ais), Barramentos, entre outros.

O SoC a ser desenvolvido neste projeto deve ser modelado em linguagem VHDL. Adicionalmente, todas as entidades modeladas devem possuir, para cada uma, um circuito de teste denominado *Testbench*, de forma a verificar e validar sua corretude em tempo de simulação.

O SoC a ser desenvolvido nesse projeto, deverá conter os seguintes subcomponentes:

1. Um Processador (CPU) capaz de executar um conjunto de instruções;
2. Duas Memórias Principais:
   - Armazenamento por bytes (8 bits);
   - Endereçadas por byte;
   - Uma memória somente para armazenamento de instruções;
   - Outra memória somente para armazenamento de dados;
3. Um *Codec* (codificador/decodificador) capaz de trabalhar com dados em formato texto, isto é, caracteres ASCII;

Este processador possuirá a capacidade de executar um conjunto reduzido de instruções (RISC), pois quer-se verificar a sua futura viabilidade em desempenho, em menor consumo de recursos físicos como área e energia e seja capaz de ser usado em qualquer dispositivo móvel/embarcado.

2 Datas Importantes

O desenvolvimento desse projeto será a parte desenvolvida em EaD na disciplina e contará como presença. Para cada presença ser computada, os grupos deverão cumprir os seguintes *deadlines*:

- 07/09/2022 a 09/09/2022
  - Prazo para entrega da versão V0 (inicial) do projeto;
- 10/10/2022 a 12/10/2022
  - Prazo para entrega da versão V1 do projeto

- 31/10/2022 a 02/11/2022
  - Prazo para entrega da versão V2 (final) do projeto;

3 O que deve ser feito?

O projeto deve ser realizada por grupos contendo 2 alunos. O projeto deve abranger a modelagem de três sub-componentes e seus respectivos *Testbenches*: Processador, Memória e Codec. Cada um destes componentes será descrito em detalhes nas subseções a seguir.

Além disso, todas as versões (V0 a V2) deverão ser submetidas via disciplina no AVA e satisfazerem os seguintes requisitos:

- Versão V0:
  - Relatório em PDF contendo definição dos integrantes dos grupos;
- Versão V1, deve conter todos os requisitos da V0 incluindo:
  - Relatório atualizado com descrição do que já foi desenvolvido até o momento no projeto;
  - Implementação e *testbenches* das entidades `mem` e `codec`;
- Versão V2, deve conter todos os requisitos da V1 incluindo:
  - Relatório atualizado com descrição do que foi concluído no projeto;
  - Relatório atualizado com descrição do que não foi concluído no projeto;
  - Relatório atualizado com enumeração dos problemas encontrados que foram solucionados no projeto;
  - Relatório atualizado com enumeração dos problemas encontrados que não foram solucionados no projeto;
  - Código dos *firmwares* usados como casos de teste;
  - Implementação e *testbenches* das entidades `soc` e `cpu`;
  - Informações adicionais que julgarem relevantes;

A nota final do trabalho prático será calculada como:

$$\text{Nota Final do Trabalho Prático} = \frac{\text{Nota V1} + \text{Nota V2}}{2}$$

3.1 Processador

O processador a ser implementado possui as características de uma arquitetura MISC (*Minimal Instruction Set Computer*): ele possui poucas instruções e todas as instruções são *simples*, isto é, cada instrução executa somente uma operação. Além disso, este processador trabalha com dados (palavras) de 1 byte (8 bits) de largura, suporta complemento de 2 e acessa duas memórias: uma de instruções (**IMEM**) e uma de dados(**DMEM**), ambas usando um endereço de 16 bits de largura. Em outras palavras: a capacidade máxima de cada memória é de $2^{16} = 65536\,\text{bytes} = 64\,\text{KiB}$.

De modo a manter a arquitetura simples, este processador não possui registradores de propósito geral. Todas as instruções presentes neste processador operam os dados diretamente em memória: em outras palavras, esta arquitetura é uma arquitetura de pilha[1].

Da mesma forma, para manter sua implementação simples, porém não menos robusta, o conjunto de instruções não fornece algumas facilidades ao programador tais como instruções de diferentes tipos e modos de endereçamento.

3.1.1 Modelo de execução

A arquitetura desse processador possui somente dois registradores denominados **IP** e **SP**. O registrador **IP** (*Instruction Pointer*) armazena o endereço da instrução a ser executada. Logo, ele aponta para um endereço da memória de instruções (**IMEM**) do SoC. O registrador **SP** (*Stack Pointer*, ponteiro da pilha) armazena o endereço do topo da pilha. A pilha, é a principal estrutura de dados desta arquitetura sendo armazenada exclusivamente na memória de dados (**DMEM**) do SoC. Por tanto, o registrador **SP** aponta para o último elemento da pilha. Como ambos os registradores são ponteiros para memória, ambos armazenam endereços de 16 bits de largura.

O ciclo de execução de uma instrução nessa arquitetura deve seguir os seguintes passos:

1. CPU acessa **IMEM** no endereço apontado por **IP**;
2. CPU recebe instrução da **IMEM** e a decodifica;
3. CPU executa instrução (aplica operação);
4. CPU altera **IP**
    - Incrementa em uma unidade, caso não seja um desvio;

[1] `https://en.wikipedia.org/wiki/Minimal_instruction_set_computer`

- Com endereço de destino, caso seja um desvio;

5. Volta ao passo 1.

Ao ligar o processador, o mesmo deve inicializar os registradores **IP** e **SP** com valor zero. Dessa forma, o processador irá buscar a primeira instrução a ser executada na memória de instruções: **IMEM**[**IP**], ou seja, **IMEM**[0]. Da mesma forma, o registrador **SP** apontará para a memória de dados na posição: **DMEM**[**SP**], ou seja, **DMEM**[0].

3.1.2 Conjunto de instruções

Cada instrução suportada pelo processador tem tamanho fixo de 1 byte de comprimento. A arquitetura possui somente um formato de instrução, que pode ser visto na Figura 1. O campo "Opcode" contém o código de operação da instrução, possui 4 bytes de comprimento, é único para cada instrução e é localizado nos 4 bits mais significativos (bits 7 a 4) da instrução. A Tabela 1 lista todas as instruções suportadas com os opcodes, como devem funcionar e seus respectivos mneumônicos.

Opcode	Immediate
7 6 5 4	3 2 1 0

Figura 1: Formato da instrução em bits.

Como mencionado anteriormente, a arquitetura deste processador é uma arquitetura de pilha. Dessa forma, quando uma instrução necessitar de vários operandos da pilha para funcionar, o operando denominado "Op1" será o operando presente no topo da pilha, isto é, apontado por **SP**. O operando denominado "Op2" estaria presente abaixo do operando "Op1", isto é, no endereço dado por **SP**-1. Por consequência, o operando denominado "Op3" estaria abaixo de "Op2" na pilha (no endereço **SP**-2), e assim por diante.

Uma excessão são as instruções `PUSHIP`, `JEQ` e `JMP`. Como elas trabalham com operandos que são endereços de memória (16 bits = 2 bytes), elas precisarão armazenar na pilha (ou recuperar dela) 2 bytes de informação. Porém, isso deverá ser feito durante a execução da instrução de forma implícita ao programador, não causando maiores transtornos no desenvolvimento do software para este processador.

Instruções que não utilizem o campo Immediate devem preenche-lo sempre com o valor zero. Por exemplo, uma instrução `NAND`, cujo opcode é `0xA`, será codificada em binário como $\texttt{10100000}_2$. Já a instrução `PUSH 3`, cujo opcode é `0x7`, será codificada em binário como $\texttt{01000111}_2$.

Apesar do conjunto de instruções não contemplar instruções aritméticas mais complexas como divisão e resto, essas mesmas operações podem ser executadas algoritmicamente com instruções de soma, subtração e laços. O mesmo vale para outras instruções lógicas, como NOT, AND e OR, que podem ser feitas através da operação lógica universal NAND.

Tabela 1: Listagem do conjunto de instruções suportadas pelo processador. Operandos (Op1, Op2, Op3) sempre tem 1 byte a não ser quando indicado em contrário.

Opcode	Mneumônico	Significado
`0x0`	`HLT`	Interrompe execução indefinidamente.
`0x1`	`IN`	Empilha um byte recebido do codec.
`0x2`	`OUT`	Desempilha um byte e o envia para o codec.
`0x3`	`PUSHIP`	Empilha o endereço armazenado no registrador **`IP`**(2 bytes, primeiro MSB[2]e depois LSB[3]).
`0x4`	`PUSH imm`	Empilha um byte contendo imediato (armazenado nos 4 bits menos significativos da instrução)
`0x5`	`DROP`	Elimina um elemento da pilha.
`0x6`	`DUP`	Reempilha o elemento no topo da pilha.
`0x8`	`ADD`	Desempilha Op1 e Op2 e empilha (Op1 + Op2).
`0x9`	`SUB`	Desempilha Op1 e Op2 e empilha (Op1 − Op2).
`0xA`	`NAND`	Desempilha Op1 e Op2 e empilha `NAND`(Op1, Op2).
`0xB`	`SLT`	Desempilha Op1 e Op2 e empilha (Op1 < Op2).
`0xC`	`SHL`	Desempilha Op1 e Op2 e empilha (Op1 ≪ Op2).
`0xD`	`SHR`	Desempilha Op1 e Op2 e empilha (Op1 ≫ Op2).
`0xE`	`JEQ`	Desempilha Op1(1 byte), Op2(1 byte) e Op3(2 bytes); Verifica se (Op1 = Op2), caso positivo soma Op3 no registrador **`IP`**.
`0xF`	`JMP`	Desempilha Op1(2 bytes) e o atribui no registrador **`IP`**.

A instrução *Halt* (`HLT`) interrompe o ciclo de execução indefinidamente. O processador só volta a execução após um *hard-reset*, isto é, após o mesmo ser desligado e ligado novamente através do sinal de entrada `halt`.

3.1.3 Interface da Entidade `cpu`

A Listagem a seguir contém a definição da interface da entidade VHDL denominada "`cpu`". A entidade define dois parâmetros genéricos e sinais de

[2]Most Significant Byte: byte mais significativo.
[3]Least Significant Byte: byte menos significativo.

entrada e saída.

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

entity cpu is
    generic (
        addr_width: natural := 16; -- Memory Address Width (in bits)
        data_width: natural := 8   -- Data Width (in bits)
    );
    port (
        clock: in std_logic; -- Clock signal
        halt : in std_logic; -- Halt processor execution when '1'

        ---- Begin Memory Signals ---
        -- Instruction byte received from memory
        instruction_in  : in  std_logic_vector(data_width-1 downto 0);
        -- Instruction address given to memory
        instruction_addr: out std_logic_vector(addr_width-1 downto 0);

        data_read : out std_logic; -- When '1', read data from memory
        data_write: out std_logic; -- When '1', write data to memory
        -- Data address given to memory
        data_addr : out std_logic_vector(addr_width-1 downto 0);
        -- Data sent from memory when data_read = '1' and data_write = '0'
        data_in  : out  std_logic_vector((data_width*4)-1 downto 0);
        -- Data sent to memory when data_read = '0' and data_write = '1'
        data_out  : in  std_logic_vector(data_width-1 downto 0);
        ---- End Memory Signals ---

        ---- Begin Codec Signals ---
        codec_interrupt: out std_logic; -- Interrupt signal
        codec_read: out std_logic; -- Read signal
        codec_write: out std_logic; -- Write signal
        codec_valid: in std_logic; -- Valid signal

        -- Byte written to codec
        codec_data_out   : in  std_logic_vector(7 downto 0);
        -- Byte read from codec
        codec_data_in  : out  std_logic_vector(7 downto 0)
        ---- End Codec Signals ---
    );
end entity;
```

A seguir, cada um desses parâmetros e sinais terá sua função descrita. Os seguintes genéricos fazem parte da entidade `cpu`:

- `addr_width`: Largura do endereço de memória (16 bits);
- `data_width`: Largura da palavra de dados do processador (8 bits);

Os seguintes sinais de entrada e saída fazem parte da entidade `cpu`:

- `clock`: Sinal de clock usado no processador;
- `halt`: Sinal de controle usado para suspender a execução do processador quando estiver em nível '1';

- `instruction_in`: Vetor de bits representando 1 byte recebido da memória de instruções;
- `instruction_addr`: Vetor de bits representando endereço enviado à memória de instruções;
- `data_read`: Sinal de controle enviado a memória comandando uma leitura de dado;
- `data_write`: Sinal de controle enviado a memória comandando uma escrita de dado;
- `data_addr`: Vetor de bits representando endereço enviado à memória de dados;
- `data_in`: Vetor de bits representando 4 bytes recebidos da memória de dados;
- `data_out`: Vetor de bits representando 1 byte enviado à memória de dados;
- `codec_interrupt`: Sinal de interrupção para ativar o codec;
- `codec_valid`: Sinal de validade do codec, informa que comando foi executado com sucesso;
- `codec_read`: Gera sinal de controle para `codec` ler 1 byte de informação;
- `codec_write`: Gera sinal de controle para `codec` escrever 1 byte de informação;
- `codec_data_in`: Vetor de bits representando 1 byte recebido do codec.
- `codec_data_out`: Vetor de bits representando 1 byte enviado ao codec.

As entidades `mem`, `codec` e `soc` possuem sinais relativos aos presentes na entidade `cpu` e que devem ser interligados entre si.

3.2 Memória

A definição VHDL da entidade `mem` está presente na Listagem abaixo. Ela possui os mesmos parâmetros genéricos da entidade `cpu`. Os sinais da entidade `cpu` prefixados com "`data_`" devem ser interligados com os sintais "`data_`" da entidade `mem`.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity memory is
5      generic (
6          addr_width: natural := 16; -- Memory Address Width (in bits)
7          data_width: natural := 8   -- Data Width (in bits)
8      );
9      port (
10         clock: in std_logic; -- Clock signal; Write on Falling-Edge
11
12         data_read : in std_logic; -- When '1', read data from memory
13         data_write: in std_logic; -- When '1', write data to memory
14         -- Data address given to memory
15         data_addr : in std_logic_vector(addr_width-1 downto 0);
16         -- Data sent from memory when data_read = '1' and data_write = '0'
17         data_in   : in  std_logic_vector(data_width-1 downto 0);
18         -- Data sent to memory when data_read = '0' and data_write = '1'
19         data_out  : out  std_logic_vector((data_width*4)-1 downto 0)
20     );
21 end entity;
```

Esta entidade deverá ser instanciada duas vezes dentro da entidade `soc` de forma a representar a memórias **`IMEM`** e a **`DMEM`** do processador. Ambas as memórias, então, deverão ser conectadas à entidade `cpu`.

Por padrão, o barramento de dados lidos representado pelo sinal `data_out` retorna 4 bytes lidos a partir do endereço enviado no sinal `data_addr`. Por exemplo: se o endereço enviado é o endereço `0x10`, seriam lidos 4 bytes a partir do endereço `0x10`, que estariam armazenados nos endereços `0x10`, `0x11`, `0x12` e `0x13`.

3.3 Codec

A definição VHDL da entidade `codec` está presente na Listagem abaixo. Ela é a entidade que fará a interface de comunicação de entrada e saída com o processador. No caso, esta entidade trabalhará com bytes lidos de um arquivo texto e transmitidos à CPU (*read*) e também receberá bytes da CPU (*write*) que serão impressos em um arquivo de saída.

Por causa do comportamento descrito, a arquitetura dessa entidade, **excepcionalmente**, não será sintetizável, pois irá simular o comportamento de um dispositivo de entrada/saída de hardware utilizando arquivos de texto.

A entidade `codec` deverá ser instanciada dentro da entidade `soc` e conectada à entidade `cpu`.

```
1  library ieee, std;
2  use ieee.std_logic_1164.all;
3  use std.textio.all;
4
5  entity codec is
6      port (
7          clock: in std_logic;
8
9          interrupt: in std_logic; -- Interrupt signal
10         read_signal: in std_logic; -- Read signal
11         write_signal: in std_logic; -- Write signal
12         valid: out std_logic; -- Valid signal
13
14         -- Byte written to codec
15         codec_data_in   : in  std_logic_vector(7 downto 0);
16         -- Byte read from codec
17         codec_data_out  : out  std_logic_vector(7 downto 0)
18     );
19 end entity;
```

Quando o processador executar a instrução `IN`, o processador deve suspender a execução, atribuir os valores "`read_signal <= '1';`" conjuntamente com "`write_signal <= '0';`" e emitir um pulso no sinal `interrupt` para a entidade `codec`. Após essa comunicação, a entidade `codec` gerará um pulso no sinal `valid` informando à entidade `cpu` que um byte foi lido com sucesso e seu valor encontra-se disponível para leitura no sinal `codec_data_out`. Nesse momento, então, a `cpu` sairá da suspenção e continuará a execução normalmente.

Quando o processador executar a instrução `OUT`, o mecanismo será semelhante, porém contrário: ele deve suspender a execução, atribuir atribuir os valores "`read_signal <= '0';`" conjuntamente com "`write_signal <= '1';`" e emitir um pulso no sinal `interrupt` para a entidade `codec`. Após essa comunicação, a entidade `codec` gerará um pulso no sinal `valid` informando à entidade `cpu` que o byte enviado pelo sinal `codec_data_in` foi escrito com sucesso. Nesse momento, então, a `cpu` sairá da suspenção e continuará a execução normalmente.

3.4 SoC

A definição VHDL da entidade `soc` está presente na Listagem a seguir. Ela é a entidade de mais alto nível da hierarquia e será ela que irá conter todas as outras entidades previamente mencionadas. Em outras palavras: a entidade `soc` encapsulará todas outras as entidades, de forma a interligá-las.

A entidade `soc` possui somente dois sinais de entrada: o sinal "`clock`", que receberá o pulso de clock gerado por um circuito auxiliar externo, e o sinal "`started`" que deve iniciar a execução quando colocado no valor '1'.

O parâmetro genérico denominado "`firmware_filename`" será usado para informar o nome de arquivo que contém o firmware (software) que será carregado na memória **`IMEM`** a partir do endereço zero quando a entidade `soc` for instanciada. Dessa forma, ao disparar a execução com "`started <= '1';`", o processador (entidade `cpu`) irá buscar a primeira instrução do firmware no endereço zero da **`IMEM`**.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity soc is
5      generic (
6          firmware_filename: string := "firmware.bin"
7      );
8      port (
9          clock: in std_logic; -- Clock signal
10         started: in std_logic -- Start execution when '1'
11     );
12 end entity;
```

4 Informações Importantes

- Trabalhos que **não** compilarem e/ou simularem receberão nota **zero**;
- **Atenção:** Não serão aceitas entregas de trabalho atrasadas.

5 Dicas e Sugestões

- Inicie o trabalho o **quanto antes**. O tempo **voa**!
- Retire as dúvidas quanto ao entendimento das entidades que compõem o circuito. Junto com a implementação dos *testbenches*, isso possibilitará detectar possíveis falhas na implementação logo cedo.
- Trabalhem em equipe e dividam a responsabilidade da implementação: um membro na modelagem VHDL, outro na implementação dos firmwares que serão usados para teste, etc.
- Frequentemente testem ambas as implementações juntas, equipes no mercado de trabalho que desenvolvem tanto hardware quanto software (Apple, Tesla, Intel, Google, Samsung, etc) utilizam uma metodologia de trabalho similar;

- Organize seu código de modo a que ele tenha partes simples e reusáveis: componha entidades complexas a partir de entidades mais simples;
- Utilize os pacotes da biblioteca IEEE e a modelagem algorítimica e dataflow ao seu favor!
- Se você já cursou (ou cursa) as disciplinas de Engenharia de Software aproveite para exercitar os conceitos de engenharia de software que facilitem seu trabalho: métodos ágeis, programação em pares, controle de versão (Git, SVN, etc);
- Aproveitem as aulas para tirar dúvidas com o professor.