Relatório de Projeto Final - Processador

Fernando Henrique Ratusznei Caetano

Sumário

1 Introdução

Esse documento apresenta o projeto final realizado para a disciplina EEB31, Circuitos Digitais, ministrada em conjunto pelos professores Fabio Kurt Schneider e Bertoldo Schneider Junior.

O projeto consiste de um sistema processador simplificado de 8 bits e foi desenvolvido para ser executado em uma placa *Terasic DE10-Lite* utilizando o software *Quartus Prime* versão 18.

Esse projeto foi feito com base no livro "Digital Design and Computer Architecture RISC-V Edition"[1].

2 Conjunto de instruções

O conjunto de instruções é composto de 9 instruções que operam em 4 registradores de uso geral ($R0$, $R1$, $R2$ e $R3$) e em uma memória RAM de 256x8 bits. Todos os operandos são sempre números de 8 bits sem sinal. O programa e os dados ocupam a mesma memória.

Os endereços 0xFF, 0x02, 0x03 e 0x04 da memória são especiais. Ao ler o endereço 0xFF é retornado o valor selecionado pelas chaves 7 até 0 da placa. Ao escrever nos endereços 0x02, 0x03 e 0x04, é simultaneamente escrito em registradores responsáveis por controlar os displays de 7 segmentos da placa. O endereço 0x02 controla os displays 0 e 1, o endereço 0x03 controla os displays 2 e 3, e o endereço 0x04 controla os displays 4 e 5.

Quatro das instruções são muito semelhantes e são agrupadas no Grupo ULA. A seguir são apresentadas as instruções.

2.1 Instruções do Grupo ULA

0	0	OP_0	OP_1	RD_0	RD_1	RS_0	RS_1

Tabela 1: Formato das instruções codificadas no Grupo ULA

OP	00	01	10	11
Nome	*zero*	*add*	*ne*	*less*

Tabela 2: Codificação das operações

Da tabela 1, as instruções do Grupo ULA tem os dois bits mais significativos 00. Os dois bits a seguir codificam a operação a ser realizada conforme a tabela 2. Os quatro últimos bits codificam os registradores destino RD e fonte RS. RD e RS podem ser o mesmo registrador.

A operação *zero* retorna sempre zero, ignorando o conteúdo de RD e RS. A operação *add* realiza a soma entre os dois registradores. A operação *ne* realiza a operação lógica NE entre os dois registradores. A operação *less* realiza a operação $RD < RS$, retornando 0xFF quando verdadeiro e 0x00 quando falso. O resultado de qualquer operação é sempre gravado em RD.

2.2 Instrução ldw

Mnemônico para *load word*. Conforme a tabela 3, os quatro bits mais significativos da instrução codificada são 0100. Logo a seguir são codificados

0	1	0	0	RD_0	RD_1	RP_0	RP_1

Tabela 3: Formato da instrução codificada *ldw*

os registradores destino RD e ponteiro RP. Essa instrução copia um valor da memória RAM para o registrador RD. O endereçamento é indireto utilizando o registrador RP. O nome dessa instrução é baseado na instrução *lw* da arquitetura RISC-V.

2.3 Instrução stw

0	1	0	1	RP_0	RP_1	RS_0	RS_1

Tabela 4: Formato da instrução codificada *ldw*

Mnemônico para *store word*. Conforme a tabela 4, os quatro bits mais significativos da instrução codificada são 0101. Logo a seguir são codificados os registradores ponteiro RP e fonte RS. Essa instrução grava o valor de RS na memória RAM. O endereçamento é indireto utilizando o registrador RP. O nome dessa instrução é baseado na instrução *sw* da arquitetura RISC-V.

2.4 Instrução sei

1	0	IMM_3	IMM_2	RD_0	RD_1	IMM_1	IMM_0

Tabela 5: Formato da instrução codificada *sei*

Mnemônico para *set immediate*. Conforme a tabela 5, os dois bits mais significativos da instrução codificada são 10. Logo a seguir são codificados o valor imediato e o registrador de destino RD. Os bits dos dois valores estão misturados de forma que os bits representado o registrador RD fique nas mesmas posições para todas as instruções que o utilizam. Essa decisão simplifica e hardware necessário para implementação.

2.5 Instrução jz

Mnemônico para *jump if zero*. Conforme a tabela 6, os quatro bits mais significativos da instrução codificada são 1100. Logo a seguir são codificados

1	1	0	0	RS_0	RS_1	RP_0	RP_1

Tabela 6: Formato da instrução codificada *jz*

os registradores fonte *RS* e ponteiro *RP*. Essa instrução realiza um salto se o conteúdo do registrador *RS* for 0x00. O endereçamento é indireto pelo registrador *RP*.

2.6 Instrução nop

1	1	1	x	x	x	x	x

Tabela 7: Formato da instrução codificada *nop*

Mnemônico para *no operation*. Conforme a tabela 7, os três bits mais significativos da instrução codificada são 111. Os outros bits não importam. Essa instrução não realiza nenhuma operação.

3 Implementação

O projeto foi todo implementado em VHDL de forma modular. A figura 1 lista os componentes e ilustra a hierarquia entre eles.

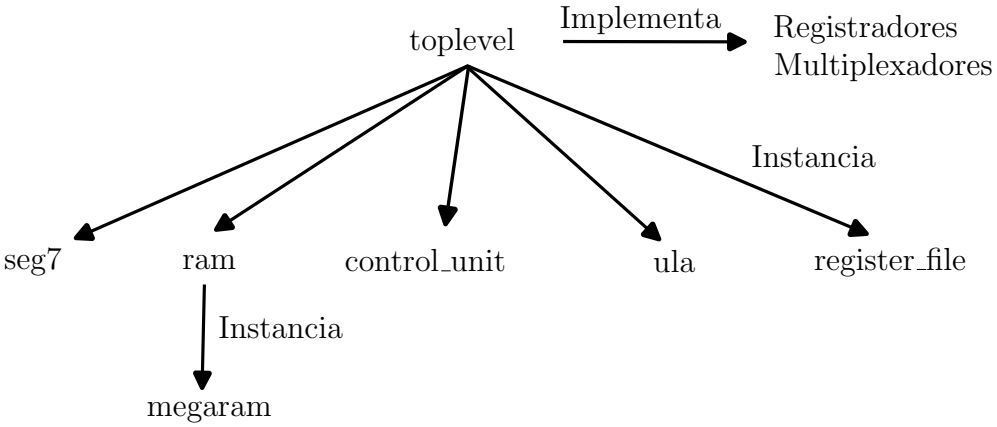

Figura 1: Hierarquia

No topo da hierarquia temos a entidade *toplevel*, que instancia os decodificadores para os displays de 7 segmentos *seg7*, a memória *ram*, a unidade

de controle *control_unit*, a unidade lógico aritmética *ula* e o arquivo de registradores *register_file*, e implementa o caminho de dados com registradores intermediários e multiplexadores utilizando construções em VHDL.

A memória *ram* ainda instancia um componente da biblioteca do Quartus, a *megaram*. O nome foi escolhido pois se trata da ram de uma porta da biblioteca *megafunctions*. Com esse componente podemos também utilizar a ferramenta *In-System Memory Content Editor* para manipular os dados na memória sem necessidade de recompilação.

Cada entidade possui um arquivo VHDL comentado correspondente de mesmo nome com a extensão .vhd na pasta do projeto. A seguir é apresentada a máquina de estados implementada pela unidade de controle.

3.1 Unidade de controle e Máquina de estados

A figura 2 apresenta o diagrama de estados da unidade de controle. Com base nesses estados a unidade de controle ativa sinais de habilitação ou seleção para os outros componentes.

Os estatos com o sufixo *wait* apenas consomem um ciclo de clock enquanto o endereço da memória é propagado. O estado *fetch* carrega a próxima instrução no registrador de instruções, *instr* no código VHDL, e incrementa o contador de programa utilizando a ula. O estado *decode* realiza uma tomada de decisão baseada na instrução atual, a partir desse ponto o fluxo é diferente para cada tipo de instrução.

Para as instruções do Grupo ULA ou a instrução *sei*, os estados *execute ula* e *execute imm* são semelhantes, selecionam os operandos correspondentes na entrada da ula. O estado seguinte *ula wb* (writeback) carrega o resultado no registrador de destino.

Para as instruções *ldw* e *stw*, o estado *mem addr* prepara um dos operandos utilizados, para a *ldw* é o endereço que será lido e para a *stw* é o registrador que será salvo na memória. Os estados seguintes *mem read* e *mem write* preparam o segundo operando de cada instrução. A instrução *lwd* ainda requer um outro estado *mem wb* para realizar a gravação no registrador.

Já para a instrução *jz*, ainda no estado *decode* o endereço de salto já é preparado. Só no próximo estado, *branch zero*, é realizada a decisão se o salto será ou não realizado.

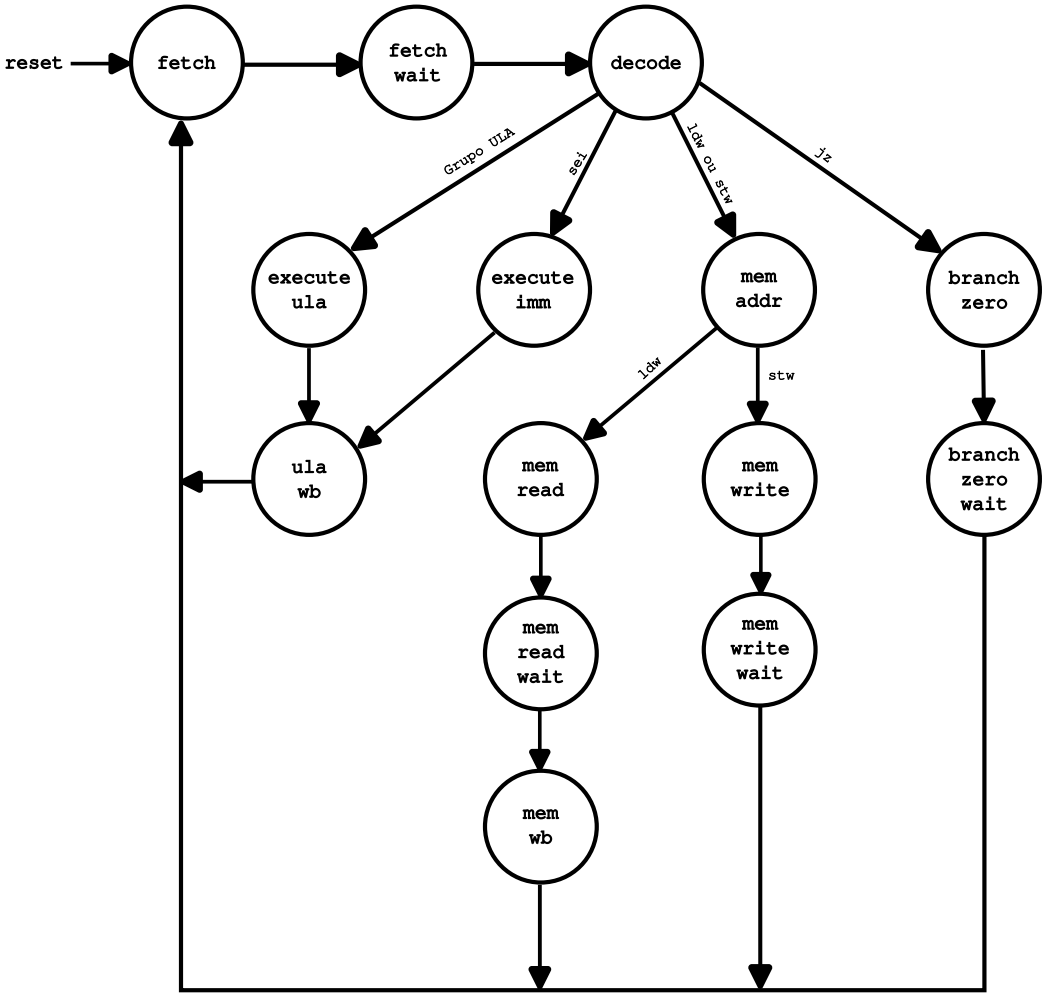

Figura 2: Diagrama de estados, as transições anotadas são as únicas com alguma dependência e dependem da instrução atual

4 Programa exemplo

Foi desenvolvido um programa exemplo para ser executado. O programa realiza o algoritmo de divisão por subtrações sucessivas entre dois operandos, apresentado o quociente e o resto nos displays da placa.

O programa está salvo no diretório *programs* do projeto. O arquivo p1.ascii contém as instruções em texto e foi utilizado para criação do código de máquina de forma manual. A seguir é apresentado um trecho desse arquivo, a primeira coluna é o endereço, o segundo é o conteúdo que será gravado na memória e o resto são comentários.

Nesse trecho de código é realizado um salto para o começo do programa

no endereço 0xf. Os dados gravados nos endereços 0x00 e 0x01 são instruções codificadas e os dados gravados nos endereços 0x02, 0x03 são os argumentos do programa, e o endereço 0x04 é onde será armazenado o resultado. No caso o programa irá calcular o quociente entre os valores hexadecimais 0x64 e 0x32.

```
00000000: b3  sei r0 0xf     1011 0011 -- pula para o
00000001: c4  jz r1 r0       1100 0100 -- comeco do programa
00000002: 64  operando x
00000003: 32  operando y
00000004: 00  quociente q = x/y
```

Existem também nesse diretório um Makefile que cria o arquivo no formato intel hex para gravação da memória. Para executar esse Makefile é necessário instalação dos utilitários de linha de comando xxd e srec. Esse script já está configurado para preparar o arquivo para ser carregado para rodar na placa ou para simulação pelo modelsim.

Com o programa carregado na placa podemos utilizar o *In-System Memory Content Editor*, acessado pelos menus conforme figura 3, para editar a memória e alterar os argumentos do algoritmo. Depois basta resetar o programa por qualquer um dos botões na placa para realizar outra divisão.

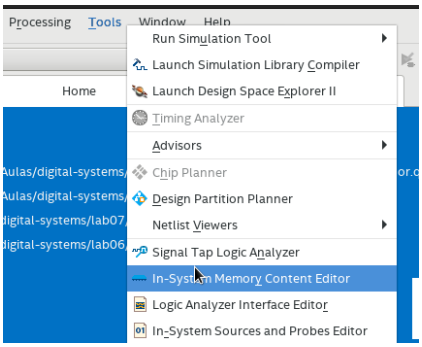

Figura 3: Acesso ao editor de memória pelo menu do Quartus

Para acessar o editor de memória foi necessário habilitar a funcionalidade durante a criação da entidade da ram de uma porta pelo catálogo IP. Essa opção está marcada com um "x"vermelho na figura 4.

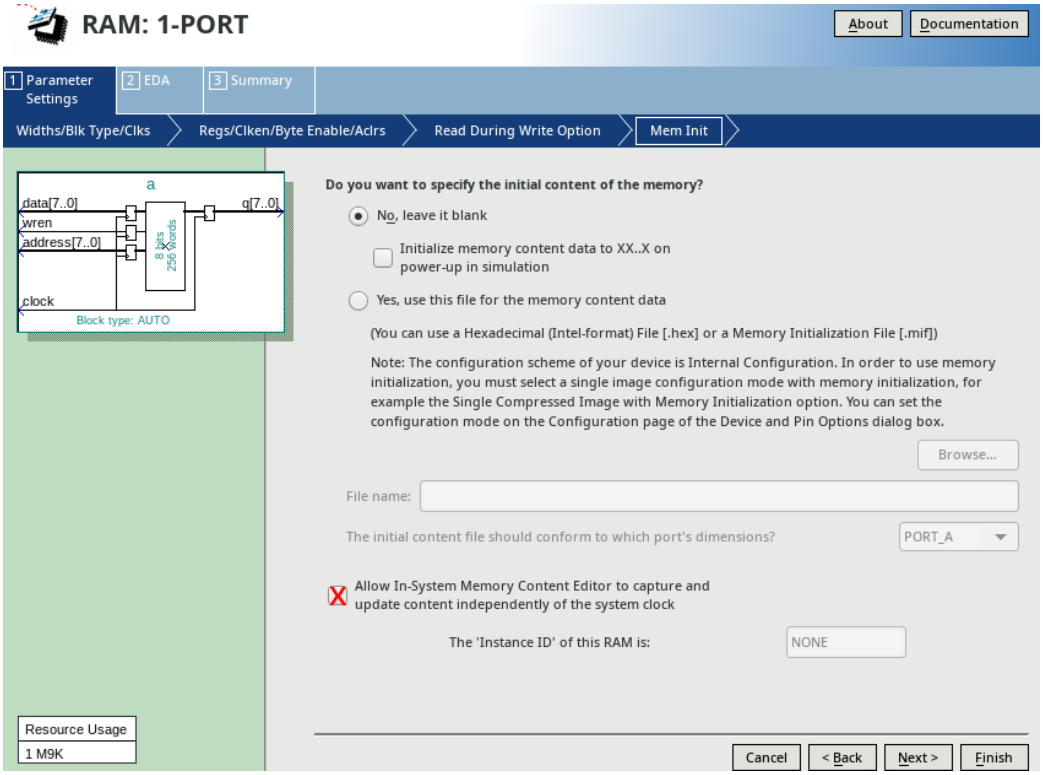

Figura 4: Opção para habilitar o editor de memória

5 Conclusão e comentários adicionais

Foi utilizado nesse projeto a maioria dos conceitos estudados durante o semestre. Conhecimento sobre memórias, lógica combinacional, registradores e máquinas de estado foram necessários para esse projeto. O processador desenvolvido é um modelo simplificado, porém já serve como base para o estudo mais aprofundado que é ementa de disciplinas futuras sobre arquitetura de computadores.

Os códigos, descrições VHDL e programa de exemplo, estão disponíveis no diretório do projeto.

Referências

[1] Sarah L. Harris e David Harris. *Digital Design and Computer Architecture RISC-V Edition*. Morgan Kaufmann, 2022.