Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DCC819 - Arquitetura de Computadores

Relatório III - Controle e Memória de Instruções

Guilherme Batista Santos
Iuri Silva Castro
João Mateus de Freitas Veneroso
Ricardo Pagoto Marinho

Belo Horizonte - MG
5 de novembro de 2017

1 Descrição

Deseja-se, agora, que o microprocessador seja capaz de ler instruções de uma memória de instruções e que as execute que forma sequencial. Instruções de controle de fluxo como *BEZ (Branch if Equals Zero)* e *J (Jump)* são necessárias para o controle de execução dos programas. Para tais tarefas necessita-se de um sistema de controle robusto, mais complexo do que o anteriormente implementado.

Para a validação do sistema utilizou-se de um algoritmo de ordenação de números inteiros positivos. Viu-se a necessidade de implementar a instrução *SLT Reg-Reg*, para fazer a comparação entre registros, anteriormente implementada apenas a versão imediatada ta instrução (SLTI Reg-Imm).

2 Implementação

Para atender as requisições, fez-se a modificação de vários módulos já implementados em relatórios passados. Adicionado instrução de SLT Reg-Reg.

O único módulo que permanenceu inalterado é o do banco de registradores.

2.1 Memória de Instruções

A memória de instruções é uma memória do tipo ROM (Read-Only Memory).

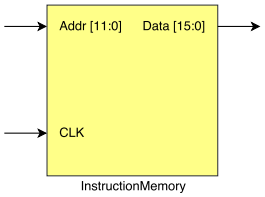

Figura 1: Módulo da Memória de Instruções.

2.2 Decodificador

O decodificador mantem o registrador de instruções (Instruction Register, IR), que mantem a instrução que está sendo atualmente executada.

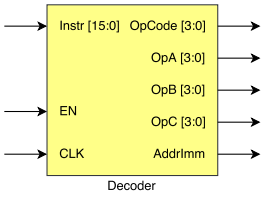

Figura 2: Unidade Lógica Aritmética.

2.3 ULA

Para o módulo da ULA, implementou-se a operação de BEZ, onde ela compara o valor do operando A com zero (OpA == 0?), retornando como resultado o valor do operando B (res = OpB) e ajustando o campo Zero do registro de flags de acordo com a comparação. Modificou-se também a ULA para que ela seja completamente combinacional, não havendo mais registros e nem sincronização com o sinal de clock.

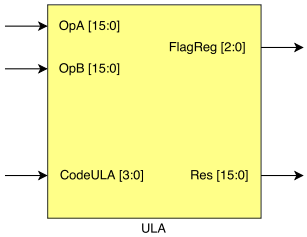

Figura 3: Unidade Lógica Aritmética.

2.4 Controle

Estágios da máquina de controle. Sinais de controle adicionados, sinais de controle não utilizados.

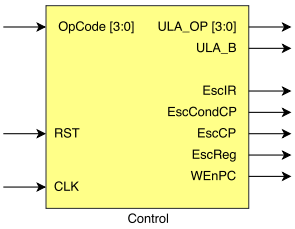

Figura 4: Unidade Lógica Aritmética.

3 Integração

Mux's para seleção de PC. Adder externo para incremento de PC. Considerar que os sinais de clock (CLK) e reset (RST) são globais, e estão conectados em todos os módulos que os têm. O diagrama do sistema pode ser visto na figura **??**.

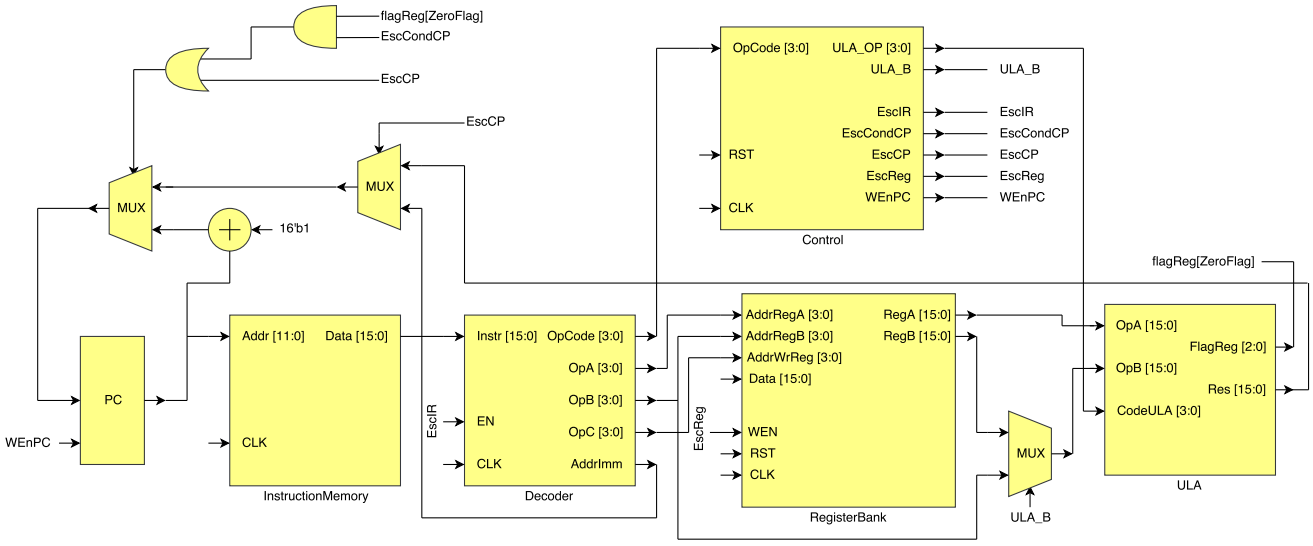

Figura 5: Sistema completo e conexões.

4 Simulação e Testes

Integrado o microprocessador passa-se para o processo de simulação e teste para validar a implementação. Para tal, gerou-se um algoritmo para fazer a ordenação de um vetor, utilizando os registros R11, R12, R13, R14 e R15 como elementos do vetor. O algoritmo faz a ordenação de forma decrescente, colocando o elemento de maior valor em R11 e o de menor valor em R15, trabalhando apenas com valores inteiros positivos.

```
START:
    XOR   R8, R8, R8     // Zera contador de swaps
    SLT   R10, R11, R12
    BEZ   N0SWAP1, R10
    ADDI  R10, 0, R11
    ADDI  R11, 0, R12
    ADDI  R12, 0, R10
    ADDI  R8, 1, R8      // Incrementa contador de swaps
NOSWAP1:
    SLT   R10, R12, R13
    BEZ   N0SWAP2, R10
    ADDI  R10, 0, R12
    ADDI  R12, 0, R13
    ADDI  R13, 0, R10
    ADDI  R8, 1, R8      // Incrementa contador de swaps
NOSWAP2:
    SLT   R10, R13, R14
    BEZ   N0SWAP3, R10
    ADDI  R10, 0, R13
    ADDI  R13, 0, R14
    ADDI  R14, 0, R10
    ADDI  R8, 1, R8      // Incrementa contador de swaps
NOSWAP3:
    SLT   R10, R14, R15
    BEZ   TEST, R10
    ADDI  R10, 0, R14
    ADDI  R14, 0, R15
    ADDI  R15, 0, R10
    ADDI  R8, 1, R8      // Incrementa contador de swaps
TEST:
    BEZ   END, R8        // Teste se houve algum swap
    J     START          // Houve, retornar para inicio
END:
    J     end            // Fim de execucao, loop infinito
```

A instrução BEZ (Branch if Equals Zero) requer que o ponteiro para qual o programa será desviado caso ocorra o branch seja armazenado em um registrador. Assim, precisa-se calcular a posição das labels do programa e carregar tais valores no banco de registradores durante a inicialização do programa. A tabeka abaixo relaciona os labels com suas posições e registros.

Não atribui-se registro ao label *START* pois não há instrução *BEZ* que faz desvio desvio para sua posição, apenas instrução *J* que utiliza imediato ao invés de ponteiro.

Converteu-se o código assembly para binário, para que então possa ser gravado no arquivo de inicialização de memória (.mif).

```
0101100010001000
1101101010111100
1100000000011010
1001101000001011
1001101100001100
1001110000001010
1001100000011000
```

Label	Posição	Registro
START	0	-
NOSWAP1	7	R1
NOSWAP2	13	R2
NOSWAP3	19	R3
TEST	25	R4
END	27	R5

Tabela 1: Labels e posições na memória de instrução.

```
1101101011001101
1100000000101010
1001101000001100
1001110000001101
1001110100001010
1001100000011000
1101101011011110
1100000000111010
1001101000001101
1001110100001110
1001111000001010
1001100000011000
1101101011101111
1100000001001010
1001101000001110
1001111000001111
1001111100001010
1001100000011000
1100000001011000
1011000000000000
1011000000011011
```

5 Discussões

Houve-se problemas para conseguir simular corretamente o módulo de memória no software ModelSim, principlamente problemas relacionados com o arquivo de inicialização de memória (.mif). Para contornar tais problemas, modificou-se o arquivo "InstrMemory.v", colocando o caminho completo do arquivo de inicialização. Pede-se ao leitor que ao fazer a reprodução da simulação atente-se a esse detalhe e que faça a modificação do caminho do arquivo para atender às configurações da máquina em que está trabalhando. Devido as limitações das formas de endereçamento atual da máquina, é necessário calcular os ponteiros dos labels ao qual os branches farão o salto no programa, armazenando-os anteriormente no banco de registradores.