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
6 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 de 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.

Foram implementados as instruções de controle de fluxo descritas acima e a instrução *SLT Reg-Reg*, que anteriormente havia apenas a versão imediata da instrução (*SLTI*), que será utilizada no algoritmo de validação da máquina.

Algumas modificações foram feitas a fim de atender o funcionamento da máquina. Tais modificações serão descrita nas seções abaixo.

2 Implementação

Para atender as requisições, fez-se a modificação de vários módulos já implementados anteriormente e houve a inclusão das intruções *BEZ*, *J* e *SLT*. O único módulo que permanenceu inalterado é o do banco de registradores. As subseções abaixos descrevem os módulos e as alterações feitas.

2.1 Memória de Instruções

A memória de instruções é uma memória do tipo ROM (*Read-Only Memory*) com tamanho de palavra de 16-bit e tamanho total da memoria de 4096 palavras. O módulo possui uma entrada para o sinal de *clock*, e uma para o endereço da palava desejada, sendo o endereço com tamanho de 12-bit para endereçar as 4096 palavras, a saída do módulo é a palavra contida no endereço especificado, que é atualizada a cada transição do sinal de *clock*. O diagrama do módulo pode ser visto na Figura **??**.

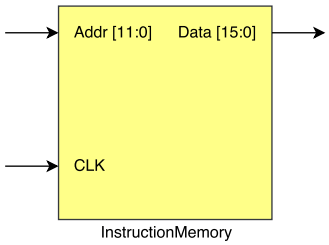

Figura 1: Módulo da Memória de Instruções.

2.2 Decodificador

O decodificador, anteriormente responsável por indicar se a instrução é imediata, agora possui o registrador de instruções (*Instruction Register, IR*), que mantem a instrução que está sendo atualmente executada, apenas encaminhando para a unidade de controle e banco de registradores os endereços dos operandos e *opcode*. O sinal *EN* (*Enable*) é necessário estar em nível lógico alto para que o registro de instruções seja atualizado, atualizando também a saída do módulo. A Figura **??** mostra o diagrama do módulo.

2.3 ULA

Para o módulo da ULA, implementou-se a operação de BEZ, onde é comparado 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

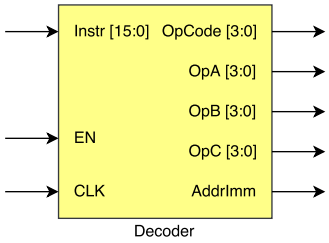

Figura 2: Módulo de decodificação.

ULA para que ela seja completamente combinacional, não havendo mais registros e nem sincronização com o sinal de *clock*. O diagrama do módulo pode ser visto na Figura **??**.

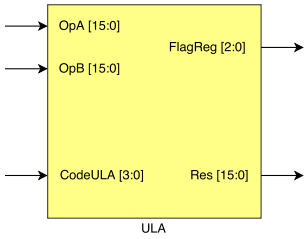

Figura 3: Unidade Lógica Aritmética.

2.4 Controle

A unidade de controle, responsável por coordenar todo o funcionamento do microprocessador, é basicamente uma máquina de estados com transições de estados a cada transição do sinal de *clock*. A máquina de estados pode ser vista na Figura **??**.

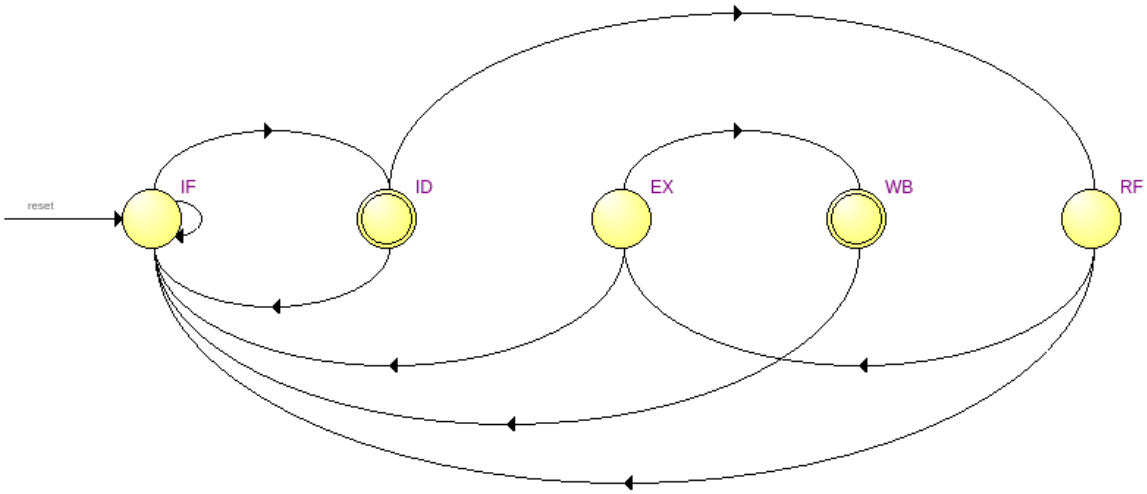

Figura 4: Máquina de estados da unidade de controle.

Definiu-se os estágios da máquina como *Instruction Fetch (IF)*, *Instruction Decode (ID)*, *Register Fetch (RF)*, *Execution (EX)* e *Write Back (WB)*.

- *IF*: Estágio de busca de instrução na memória de instrução. O registro *Program Counter* (*PC*) é responsável em apontar qual é o endereço da próxima instrução a ser buscada para execução;
- *ID*: Estágio de decodificação da instrução. A instrução retornada pela memória de instruções é decodificada e quebrada em partes, alimentando dados a unidade de controle e banco de registradores;
- *RF*: Estágio de busca de registradores no banco de registradores. Os registros necessários para a execução da instrução são buscados nesse estágio;
- *EX*: Estágio de execução da instrução. A unidade de controle informa a ULA qual operação deve ser feita e encaminha os operandos corretos para a unidade;
- *WB*: Estágio de escrita dos resultados e modificação do estado da máquina. O resultado da ULA pode ser guardado no banco de registradores, atualiza-se o *PC* e retorna-se para o estágio de *IF*.

O diagrama do módulo de controle pode ser visto na Figura **??**.

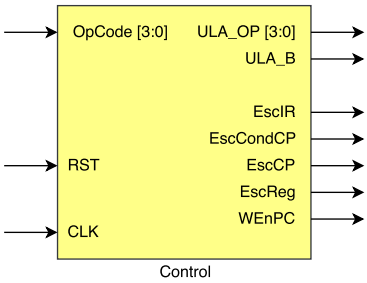

Figura 5: Módulo de controle.

O sinal *ULAOP* indica à ULA qual operação deverá ser feita. *ULAB* indica se o dado é um imediato ou o conteúdo do registrador B. *EscIR* é o sinal de escrita para o registrador de instruções, localizado dentro do módulo de decodificação. Os sinais *EscCondCP* e *EscCP* são responsáveis por indicar se a instrução é um salto condicional (*Branch*) ou um salto incondicional *Jump*. *EscReg* é o sinal para escrita no banco de registradores e *WEnPC* é o sinal para escrita no registrador *PC*.

3 Integração

Após a implementação dos módulos, faz-se a integração do sistema (microprocessador). O diagrama do sistema pode ser visto na Figura **??**, considere que os sinais de *clock (CLK)* e *reset (RST)* são globais, e conectados a todos os módulos que os possuem.

Há a utilização de 3 multiplexadores, sendo um para a seleção entre registro e imediato (operando B) e os outros dois trabalham em conjunto para definir o próximo valor do registro *PC*, podendo ser ele um valor de um registro, o endereço imediato ou o incremento do próprio registro *PC* mais 1. Para a opção de incremento do registro *PC*, optou-se pela utilização de um somador dedicado, diminuindo a complexidade da unidade de controle.

O sistema é comandado pelo módulo de controle que, através dos sinais de controle, consegue ditar toda a execução das instruções.

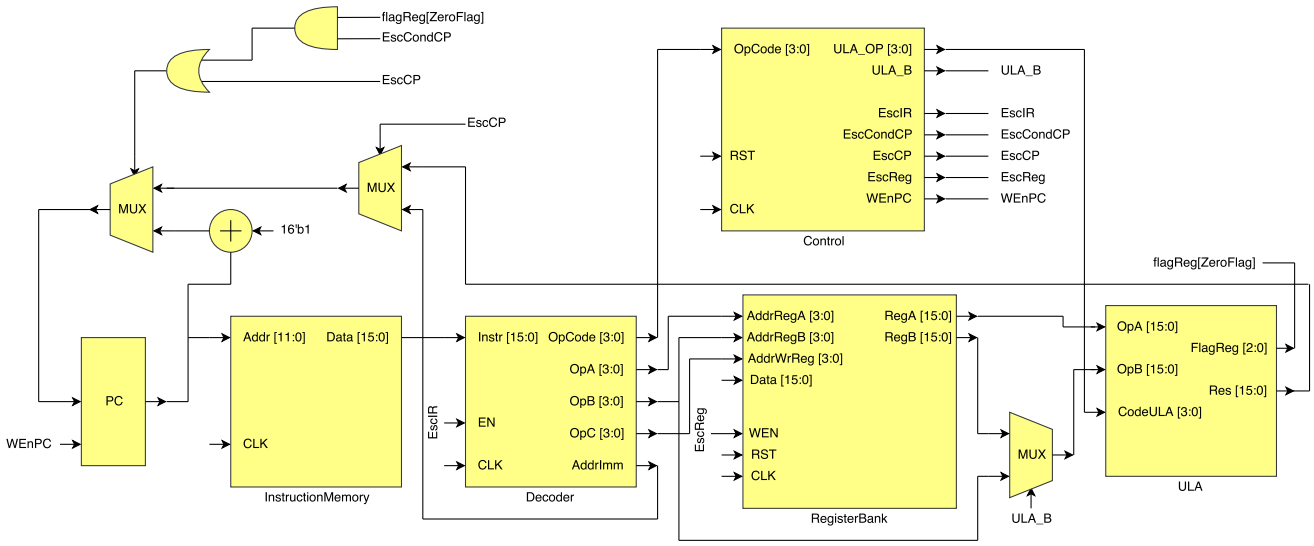

Figura 6: Sistema completo e conexões.

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.

A instrução *SLT Reg-Reg* é utilizada para fazer a comparação entre os registros. Utiliza-se os registradores R8 e R10 como auxiliares. Ao fim o programa fica preso em um laço infinito, simbolizando o fim da execução do mesmo. O código pode ser visualizado abaixo.

```
START:
    XOR   R8, R8, R8     // Zera contador de swaps
    SLT   R10, R11, R12
    BEZ   NOSWAP1, 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   NOSWAP2, 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   NOSWAP3, 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
```

```
27      ADDI   R14, 0, R15
28      ADDI   R15, 0, R10
29      ADDI   R8, 1, R8      // Incrementa contador de swaps
30  TEST:
31      BEZ    END, R8        // Teste se houve algum swap
32      J      START          // Houve, retornar para inicio
33  END:
34      J      END            // Fim de execucao, loop infinito
```

A instrução *BEZ* (*Branch if Equals Zero*) requer que o ponteiro para o 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 Tabela **??** relaciona os labels com suas posições e registros.

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.

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

Converteu-se o código assembly para binário utilizando um pequeno assembler desenvolvido para auxiliar no trabalho, para que então possa ser gravado no arquivo de inicialização de memória (.mif).

```
1  0101100010001000
2  1101101010111100
3  1100000000011010
4  1001101000001011
5  1001101100001100
6  1001110000001010
7  1001100000011000
8  1101101011001101
9  1100000000101010
10 1001101000001100
11 1001110000001101
12 1001110100001010
13 1001100000011000
14 1101101011011110
15 1100000000111010
16 1001101000001101
17 1001110100001110
18 1001111000001010
19 1001100000011000
20 1101101011101111
21 1100000001001010
22 1001101000001110
23 1001111000001111
24 1001111100001010
25 1001100000011000
26 1100000001011000
27 1011000000000000
28 1011000000011011
```

Fez-se a simulação do sistema rodando o algoritmo acima no ModelSim. Pode-se ver as formas de onda na Figura **??**.

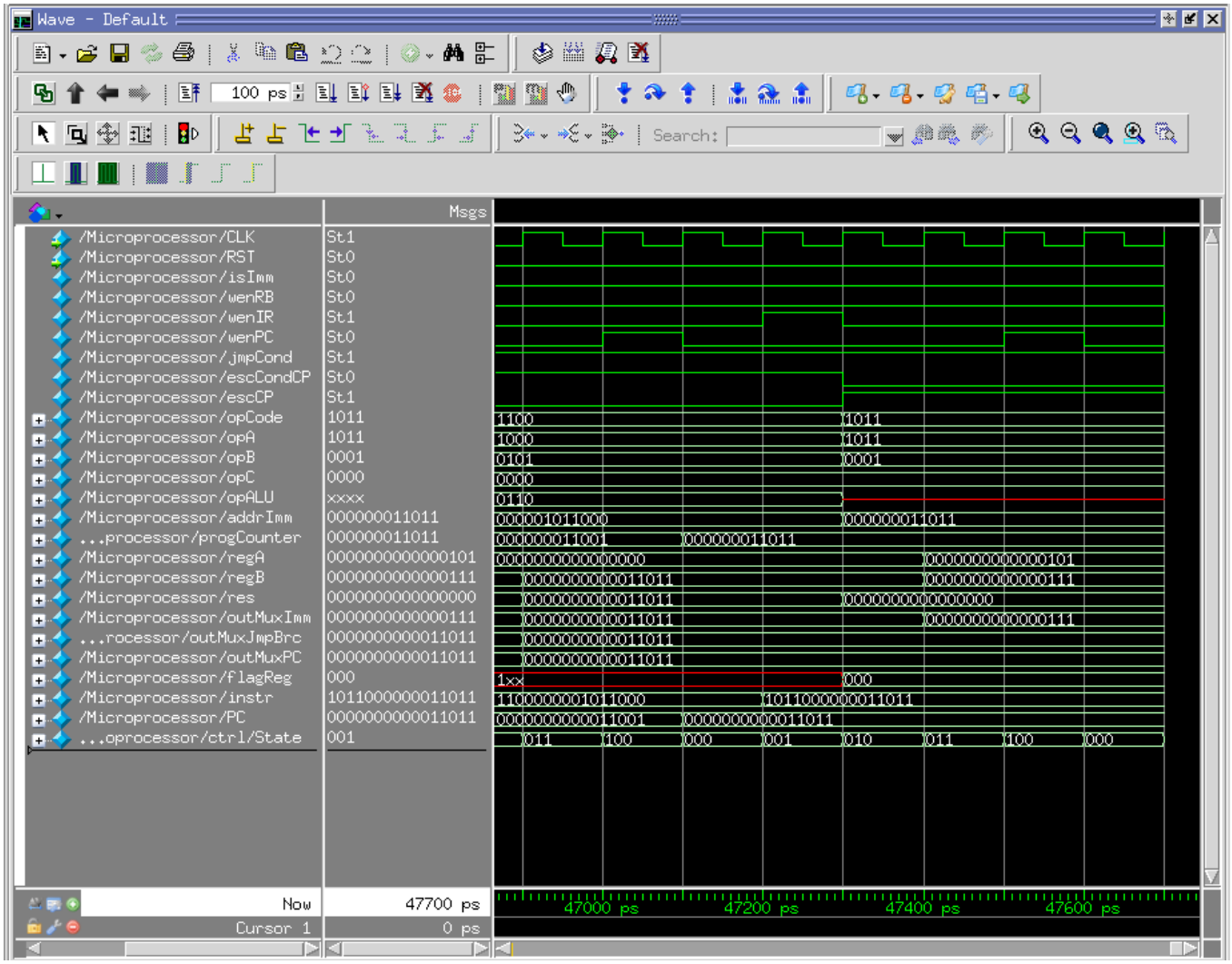

Figura 7. Simulação no ModelSim.

O script de inicialização da simulação e ajuste dos dados no banco de registradores pode ser visto abaixo.

```
## Arquivo de simulacaoo para o ModelSIM
vsim -L altera_mf_ver -L lpm_ver -L cycloneiii_ver -L cycloneii_ver Microprocessor

add wave -position end  sim:/Microprocessor/CLK
add wave -position end  sim:/Microprocessor/RST
add wave -position end  sim:/Microprocessor/isImm
add wave -position end  sim:/Microprocessor/wenRB
add wave -position end  sim:/Microprocessor/wenIR
add wave -position end  sim:/Microprocessor/wenPC
add wave -position end  sim:/Microprocessor/jmpCond
```

```
add wave -position end  sim:/Microprocessor/escCondCP
add wave -position end  sim:/Microprocessor/escCP
add wave -position end  sim:/Microprocessor/opCode
add wave -position end  sim:/Microprocessor/opA
add wave -position end  sim:/Microprocessor/opB
add wave -position end  sim:/Microprocessor/opC
add wave -position end  sim:/Microprocessor/opALU
add wave -position end  sim:/Microprocessor/addrImm
add wave -position end  sim:/Microprocessor/progCounter
add wave -position end  sim:/Microprocessor/regA
add wave -position end  sim:/Microprocessor/regB
add wave -position end  sim:/Microprocessor/res
add wave -position end  sim:/Microprocessor/outMuxImm
add wave -position end  sim:/Microprocessor/outMuxJmpBrc
add wave -position end  sim:/Microprocessor/outMuxPC
add wave -position end  sim:/Microprocessor/flagReg
add wave -position end  sim:/Microprocessor/instr
add wave -position end  sim:/Microprocessor/PC
add wave -position end  sim:/Microprocessor/ctrl/State

force -freeze sim:/Microprocessor/CLK 1 0, 0 {50 ps} -r 100
force -freeze sim:/Microprocessor/RST 1 0
run
force -freeze sim:/Microprocessor/RST 0 0

## Inicializacao do banco de registros para teste: R11 = 1, R12 = 2, R13 = 3, R14 = 4, R15 = 5
mem load -skip 0 -filltype value -filldata 0001 -fillradix symbolic -startaddress 11 -endaddress 11 /Microprocessor/regBank/RegBank
mem load -skip 0 -filltype value -filldata 0010 -fillradix symbolic -startaddress 12 -endaddress 12 /Microprocessor/regBank/RegBank
mem load -skip 0 -filltype value -filldata 0011 -fillradix symbolic -startaddress 13 -endaddress 13 /Microprocessor/regBank/RegBank
mem load -skip 0 -filltype value -filldata 0100 -fillradix symbolic -startaddress 14 -endaddress 14 /Microprocessor/regBank/RegBank
mem load -skip 0 -filltype value -filldata 0101 -fillradix symbolic -startaddress 15 -endaddress 15 /Microprocessor/regBank/RegBank

## Inicializacao dos ponteiros para os branches no banco de registros
mem load -skip 0 -filltype value -filldata 00111 -fillradix symbolic -startaddress 1 -endaddress 1 /Microprocessor/regBank/RegBank
mem load -skip 0 -filltype value -filldata 01101 -fillradix symbolic -startaddress 2 -endaddress 2 /Microprocessor/regBank/RegBank
mem load -skip 0 -filltype value -filldata 10011 -fillradix symbolic -startaddress 3 -endaddress 3 /Microprocessor/regBank/RegBank
mem load -skip 0 -filltype value -filldata 11001 -fillradix symbolic -startaddress 4 -endaddress 4 /Microprocessor/regBank/RegBank
mem load -skip 0 -filltype value -filldata 11011 -fillradix symbolic -startaddress 5 -endaddress 5 /Microprocessor/regBank/RegBank

## Passo da instrucao: 5 ciclos (500ps). ~95 execucoes de instrucoes para ordenacao com os valores atuais
run 47600
```

5 Discussões

Teve-se problemas para conseguir simular corretamente o módulo de memória no software ModelSim, principalmente 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.

A instrução *SLT Reg-Reg* foi implementada apenas para atender o programa de ordenação. O somador dedicado para o registro de contador de programa foi implementado para diminuir a complexidade do controle da máquina e o número de estágios.