Relatório - MIPS Monociclo

Millena Suiani Costa
Departamento de Informática
Universidade Federal do Paraná – UFPR
Curitiba, Brasil
millena.costa@ufpr.br

***Resumo*—O trabalho em questão consistiu no desenvolvimento de um processador subconjunto daquele que se denomina MIPS Monociclo e seus devidos componentes. Foram obtidos como resultado o seu circuito principal – bem como os subcircuitos utilizados em sua constituição –, os hexadecimais das Memórias de Controle e Instruções, além de seu código teste desenvolvido na linguagem *assembly*.**

***Index Terms*—Processador, MIPS, Monociclo, Assembly.**

I. Introdução

O presente trabalho foi elaborado e desenvolvido com base, especialmente, no capítulo 5 do livro *"Computer Organization and Design MIPS Edition"* dos professores universitários e cientistas da computação David Andrew Patterson e John LeRoy Hennessy [1], bibliografia essa que descreve detalhadamente os processos necessários no desenvolvimento de um processador MIPS Monociclo. Com base nas instruções fornecidas, bem como nas operações contidas no MIPS Green Card, foi possível projetar um circuito principal – e seus devidos subcircuitos – no *software Digital*, levando, também como base, a sua documentação em português [2]. Além desses, foram desenvolvidos códigos hexadecimais que objetivaram a obtenção de um parâmetro de teste para a execução de cada uma das operações das memórias ROM contidas no projeto, além da certificação de um correto funcionamento dessas.

II. Unidade de Controle

A elaboração do trabalho teve início através do planejamento da unidade de controle do MIPS com orientação no caminho de dados sugerido na enunciação do projeto. A unidade de controle em si define a união de cada um dos componentes que constituem o processador e nela foram inseridos um *Program Counter (PC)*, dois somadores, cinco multiplexadores, um extensor de sinal, uma porta lógica *and*, dois deslocadores para a esquerda, dois distribuídores de *bits*, múltiplos túneis para a melhor organização do circuito, três memórias ROM – sendo essas uma Memória de Controle, uma Memória de Instruções e uma Memória de Controle da Unidade Lógica e Aritimética (ULA) – e uma memória RAM – sendo essa a Memória de Dados –, além da própria Unidade Lógica e Aritimética e um Bloco de Registradores, sendo, ambos os últimos componentes, subcircuitos.

A. Bloco de Registradores

O Bloco de Registradores consiste em 32 registradores de 32 *bits*, sendo esses o registrador r0 – que, independente do valor que se queira gravar neste, apenas armazena o valor zero – e os registradores r1..r31 que poderão receber demais valores.

O funcionamento do bloco se dá através de duas portas rDT1 (*Read Data 1*) e rDT2 (*Read Data 2*), ambas de 32 *bits*, que dispõem dos conteúdos contidos nos registradores endereçados por RR1 (*Read Register 1*) e RR2 (*Read Register 2*), portas essas de 5 *bits* cada. Esses conteúdos podem ser tanto utilizados nas operações de Lógica e Aritimética – cujo resultado destas pode ser armazenado em um registrador endereçado por WR (*Write Register*) através de uma porta de entrada WD (*Write Data*) quando o sinal de habilitação RW (*Register Write*) está em nível lógico alto, sendo esses respectivamente de 5, 32 e 5 *bits* – quanto utilizados nas operações de controle presentes na Memória de Controle.

B. Unidade Lógica e Aritimética

A Unidade Lógica e Aritimética consiste em um subcircuito no qual estão constadas oito operações a serem realizadas com os valores imputados nas entradas A e B. Há de se esclarecer que, para a correta execução das operações de deslocamento de *bits*, sendo consideradas as especificidades do *software* utilizado, foi necessário aplicar um distribuidor de bits que inserisse 0 no bit mais significativo do *shamt* para que esse, por sua vez, contasse com 6 *bits* ao invés dos 5 habituais e, ainda assim, não resultasse num valor de *input* alterado.

A seleção do resultado á ser direcionado para a saída principal (O), dentre os de todas as operações da ULA, se dá através de um multiplexador com 3 *bits* de seleção, esses que representam a saída de duas memórias de controle particulares da ULA, uma para operações de tipo R e outra para operações do tipo I.

Além da saída principal, essa que, como citado anteriormente, representa o resultado da operação selecionada, há ainda uma segunda saída (Z) que, através de uma porta lógica *nor* cujas entradas são os 32 bits da saída O distribuídos de um a um, indica se o resultado da operação é um resultado nulo, direcionando nível lógico alto para a saída, e nível lógico baixo, caso contrário.

III. Casos Particulares

É detalhada, na presente sessão, a implementação dos casos particulares de cada uma das instruções do MIPS que carecem de uma explicação aprofundada para melhor compreensão de como se deu o planejamento das individualidades do projeto.

A. Instruções de Tipo R

As instruções do tipo R implementadas no MIPS Monociclo – sendo essas *and, or, nor, add, subtract (sub), set less than (slt), shift left logical (sll) e shift right logical (srl) e jump register (jr)* – possuem, em geral, um mesmo caminho de dados (salvo a operação *jump register (jr)*, cuja aplicação é detalhada na subseção seguinte). Este trata-se da leitura dos dados de dois registradores, o direcionamento destes à Unidade Lógica e Aritimética, a execução da devida operação e o armazenamento direto de seu resultado em um terceiro registrador.

Todas essas operações contam com um mesmo *ALUOp*, precedido pelo *bit* 1 e seguido por outros três *bits Don't Care (X)*. Diferente das demais operações, no caso das operações de tipo R, os bits em sequência não importam devido às suas operações serem efetivamente selecionadas pelos 6 *bits* menos significativos da instrução, ou seja, os bits de *function*.

B. Jump (j) e Jump Register (jr)

Para as referidas operações de salto, optou-se pela realização de seus controles em um mesmo multiplexador com *bits* seletores advindos dos *bits* de *"Jump"* presentes na memória de controle. Caso não haja salto algum sendo realizado, os *bits* de *"Jump"* serão 00, caso o salto seja da instrução de mesmo nome, *jump (j)* os *bits* de *"Jump"* serão 01 e, caso o salto seja para registrador – ou seja, a operação realizada for a denominada *jump register (jr)* – os *bits* de *"Jump"* serão 11.

C. Jump And Link (jal) e Load Upper Immediate (lui)

Para a implementação da instrução *jump and link (jal)* foi expandido o multiplexador cujo seletor comporta os *bits* de *RegDst* para uma terceira opção que leva a constante 31 para, caso esses *bits* do seletor em questão corresponderem à 10, o resultado de PC+8 será armazenado ao registrador cujo endereço corresponde à constante e, em paralelo, o *program counter (PC)* receberá o *Jump Address*.

Já a instrução *load upper immediate (lui)* desloca um imediato em 16 bits à esquerda e grava o resultado no registrador R[rt] (por, logicamente, tratar-se de uma operação do tipo I), assim, para a sua implementação, foi redirecionado os *bits* de "imediato"à um shifter cujo resultado serve de entrada para um multiplexador, caso os *bits* de controle *MemToReg* sejam 10, o resultado do shifter será armazenado no registrador em questão.

D. Branch On Equal (beq) e Branch On Not Equal (bne)

Como citado na sessão B da Unidade de Controle, as instruções *branch on equal (beq)* e *branch on not equal (bne)* são executadas considerando-se a saída indicadora de resultado nulo da Unidade Lógica Aritimética (ULA). Nela são introduzidos dois valores que se deseja comparar, e a partir deles é realizada uma operação de subtração.

Caso o resultado da subtração for equivalente à zero, significa que ambos os valores são iguais e, assim, é direcionado um sinal de nível alto para a saída "Z", já no caso de valores diferentes, é direcionado um sinal de nível baixo. Esse sinal é encaminhado à duas portas lógicas *and* e *nor* que seguem da ULA. Caso os *bits* de *Branch*, advindos da memória de controle, sejam 01, um multiplexador seleciona a porta lógica equivalente à operação de *branch on equal (beq)*, comparando assim se o sinal lógico resultante de Z é equivalente à 1, caso for, o desvio ocorre. Já no caso dos *bits* de *Branch* serem 10, o multiplexador seleciona a porta lógica equivalente à operação de *branch on not equal (bne)*, e dessa vez, ao contrário da operação anterior, compara se o sinal lógico é equivalente à 0 e, caso for, o desvio ocorre.

E. Load Word (lw) e Store Word (sw)

Na execução da operação *load word (lw)* temos o endereço de um determinado registrador somado à um imediato de sinal extendido na ULA e, em sequência, sendo acessado na memória de dados – devido ao sinal de leitura da memória *MemRead* estar em nível lógico alto – e, por fim, sendo armazenado em um terceiro registrador.

Já na operação *store word (sw)* é aplicada a lógica inversa. Assim como na operação anterior, é somado o endereço de um determinado registrador à um imediato de sinal extendido na ULA, entretanto, ao contrário do que já foi descrito, esse valor não mais será armazenado em um registrador, mas sim, será o endereço de memória cujo valor do terceiro registrador da operação será gravado – devido ao sinal de escrita de memória *MemWrite* estar em nível lógico alto.

Ao analisarmos os *bits* de controle das operações em questão temos que, em ambas as operações, tanto é efetuada a mesma operação de soma *add* implementada na ULA e, por conseguinte, tanto possuem os mesmos *bits* de *ALUOp* quanto apresentam *bit* 1 para o ALUSrc por serem instruções do tipo I, ou seja, contam com "imediatos"em uma das entradas das operações de soma. Já a diferença nos *bits* dessas operações se dá pelas operações de acesso à valores na memória de dados *(MemRead)* e a gravação destes em um determinado registrador (MemToReg), apenas serem habilitados durante o *load word (lw)*. Ademais, a operação de gravação de valores na memória de dados (MemWrite) apenas é habilitada na operação de *store word (sw)*.

IV. Acesso Às Memórias

O circuito do MIPS Monociclo dispõe de quatro memórias ROM, duas Memórias para Controle da Unidade Lógica e Aritimética – sendo essas a que representa instruções do tipo R (com 6 *bits* de endereços expressos pelo campo funct e 3 *bits* de dados) e a que representa instruções do tipo I (com 3 *bits* de endereços expressos pelos *bits* de *ALUOp* e 3 *bits* de dados) –, uma Memória de Controle (com 6 *bits* de endereços de 16 *bits* de dados) que armazena os *bits* de controle para cada operação, e a Memória de Instruções (com endereços de 24 *bits* e dados de 32 *bits*) na qual é armazenado o código de teste do MIPS Monociclo.

No desenvolvimento dos hexadecimais para ambas as Memórias de Controle da ULA foram levadas em consideração as 16 operações que a constam em seus caminhos de dados.

Primeiramente, foram estabelecidos 4 *bits* de *ALUOp* – desses, o mais significativo representa o *"enable"* de ambas as memórias – e, em seguida, separadas as memórias e realizados os devidos ajustes com distribuidores de *bits*. Por fim, foi elaborada a Tabela I, cujas primeiras 8 linhas representam a primeira memória (para o controle de instruções de tipo R) e as 8 últimas representam a segunda memória (para controle de instruções de tipo I).

Tabela I
DISTRIBUIÇÃO DE BITS NAS MEMÓRIAS DE CONTROLE DA ULA

	Type	ALUOP	func	saída
and	R	1000	000000	000
or	R	1000	000001	001
nor	R	1000	000010	010
add	R	1000	000011	011
sub	R	1000	000100	100
slt	R	1000	000101	101
sll	R	1000	000110	110
srl	R	1000	000111	111
andi	I	0000	000000	000
ori	I	0001	000000	001
slti	I	0010	000000	101
addi	I	0011	000000	011
lw	I	0011	000000	011
sw	I	0011	000000	011
beq	I	0100	000000	100
bne	I	0100	000000	100

Já para desenvolver o hexadecimal da Memória de Controle, foi estruturada a Tabela II, contendo o nome de cada operação e os valores binários utilizados pelos *bits* de seleção de cada uma delas e, por fim, traduzido o binário resultante para hexadecimal.

Tabela II
DISTRIBUIÇÃO DE BITS NA MEMÓRIA DE CONTROLE

	RegDst	Jump	Branch	MemToReg	MemRead	MemWrite	ALUOp	ALUSrc	RegWrite
and	01	00	00	00	0	0	1000	0	1
andi	00	00	00	00	0	0	0000	1	1
or	01	00	00	00	0	0	1000	0	1
ori	00	00	00	00	0	0	0001	1	1
nor	01	00	00	00	0	0	1000	0	1
add	01	00	00	00	0	0	1000	0	1
addi	00	00	00	00	0	0	0011	1	1
sub	01	00	00	00	0	0	1000	0	1
slt	01	00	00	00	0	0	1000	0	1
slti	00	00	00	00	0	0	0101	1	1
sll	01	00	00	00	0	0	1000	0	1
srl	01	00	00	00	0	0	1000	0	1
lw	00	00	00	01	1	0	0011	1	1
sw	00	00	00	00	0	1	0011	1	0
beq	00	00	01	00	0	0	0100	0	0
bne	00	00	10	00	0	0	0100	0	0
j	00	01	00	00	0	0	0000	0	0
jal	10	01	00	11	0	0	0000	0	1
jr	00	11	00	00	0	0	0000	0	0
lui	00	00	00	10	0	0	0000	0	1

V. MEMÓRIA DE INSTRUÇÕES E PROGRAMA TESTE

O programa teste para cada uma das instruções – cujo binário é alocado na Memória de Instruções – foi idealizado através da construção da Tabela III, na qual foram inseridos, nas linhas os códigos *assembly* de cada operação e nas colunas a conversão destes para digitos binários.

Há de se destacar que, para correto funcionamento do programa, foram inseridas instruções sem efeito prático, conforme as instruções enunciadas para o desenvolvimento do projeto. Ademais, para fins de teste, foram inseridas pontas de prova ao longo do circuito para monitorar as linhas de operação e os resultados decorrentes de cada uma delas, bem como as operações que os provocam. Todos os resultados previstos ao longo do desenvolvimento do código *assembly* foram devidamente obtidos.

Tabela III
DEFINIÇÃO DOS BINÁRIOS PARA O CÓDIGO DE TESTE

	Assembly	OpCode	RS(Addr)	RT(Addr)	RD(Addr/Imm)	Shamt(Addr/Imm)	Funct(Addr/Imm)
L0	addi r1, r0, Imm(12)	000110	00000	00001	00000	00000	001100
L1	addi r2, r0, Imm(16)	000110	00000	00010	00000	00000	010000
L2	addi r3, r0, Imm(4)	000110	00000	00011	00000	00000	000100
L3	j Imm(8)	010000	00000	00000	00000	00000	001000
L4	add r0, r0, r0	000101	00000	00000	00000	00000	000011
L5	sub r4, r2, r3	000111	00010	00011	00100	00000	000100
L6	beq r4, r1, Imm(2)	001110	00100	00001	00000	00000	000010
L7	add r0, r0, r0	000101	00000	00000	00000	00000	000011
L8	jr r2	010010	00010	00000	00000	00000	000000
L9	add r0, r0, r0	000101	00000	00000	00000	00000	000011
L10	add r5, r3, r3	000101	00011	00011	00101	00000	000011
L11	add r5, r5, r1	000101	00101	00001	00101	00000	000011
L12	slti r6, r3, Imm(10)	001001	00011	00110	00000	00000	001010
L13	bne r6, r0, Imm(2)	001111	00110	00000	00000	00000	000010
L14	add, r0, r0, r0	000101	00000	00000	00000	00000	000011
L15	sub r6, r6, r2	000111	00110	00010	00110	00000	000100
L16	andi r7, r2, Imm(0)	000001	00010	00111	00000	00000	000000
L17	and r7, r7, r6	000000	00111	00110	00111	00000	000000
L18	sw r1, Imm(8), r7	001101	00001	00111	00000	00000	001000
L19	lw r8, r0, Imm(20)	001100	00000	01000	00000	00000	010100
L20	or r8, r7, r0	000010	00111	00000	01000	00000	000001
L21	ori r9, r8, Imm(0)	000011	01000	01001	00000	00000	000000
L22	nor r9, r9, r2	000100	01001	00010	01001	00000	000010
L23	sll r10, r3, Imm(4)	001010	00000	00011	01010	00100	000110
L24	srl r11, r1, Imm(3)	001011	00000	00001	01011	00011	000111
L25	slt r12, r11, r10	001000	01011	01010	01100	00000	000101
L26	lui r1, Imm(2)	010011	00000	00001	00000	00000	000010
L27	jal L1	010001	00000	00000	00000	00000	000001

VI. CONCLUSÃO

A realização do projeto se deu através das bases fornecidas e seguindo as instruções apresentadas para tal, assim, foi possível o desenvolvimento de um circuito funcional (assim como os subcircuitos que o constituem) e do dump em hexadecimal das memórias nele contidas. Sua funcionalidade foi testada através de um código *assembly* convertido para hexadecimal, focado na verificação das funções projetadas exclusivamente para o MIPS Monociclo. Ao fim do projeto concluiu-se que o trabalho finalizado apresentou resultados satisfatórios, executando as instruções solicitadas de maneira correta.

VII. GLOSSÁRIO

As abreviações *bit* ou *bits* advém de *Binary Digit(s)*, em tradução livre do inglês, Digitos Binários.

A abreviação *opcode* advém de *Operational Code*, em tradução livre do inglês, Código Operacional.

O acrônimo *ROM* advém de *Read-Only Memory*, em tradução livre do inglês, Memória Somente de Leitura.

O acrônimo *RAM* advém de *Random Access Memory,*, em tradução livre do inglês, Memória de Acesso Randômico.

As expressões foram amplamente utilizadas nas principais referências utilizadas como base para o desenvolvimento do presente trabalho, o livro de David A. Patterson e John L. Hennessy [1] e o MIPS Green Card [3].

REFERÊNCIAS

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Morgan Kaufmann, 2013, ch. 5.

[2] "Digital - some minor improvements and bug fixes (doc-portugues.pdf)," Helmut Neemann, 02 2022. [Online]. Available: https://github.com/hneemann/Digital/releases/tag/v0.29

[3] "Mips reference data card ("green card")," urlhttps://inst.eecs.berkeley.edu/ cs61c/resources/MIPS$_{G}reen_{S}heet.pdf$, 2009.