UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
CONCEPÇÃO ESTRUTURADA DE CIRCUITOS INTEGRADOS

Microarquitetura Multi-cilco MIPS32

THIAGO ALVES DE ARAUJO

2016019787

1 - Microarquitetura Multi-clico

A microarquitetura é a forma como um determinado conjunto de instruções (ISA) é implementado em um processador. Um exemplo de arquitetura que pode ser utilizada é a *monociclo*, onde cada *instrução* é executada em um único ciclo de clock grande o suficiente para acomodar toda a instrução. A grande desvantagem dessa arquitetura é que uma série de etapas correspondentes às operações das unidades funcionais ficam obsoletas enquanto a instrução é executada.

Outro exemplo de arquitetura é a *multi-cilo*. Com ela, cada *etapa* da execução leva um ciclo de clock. Com isso, cada unidade funcional pode ser usada por mais de uma instrução, desde que sejam usadas em ciclos de clock diferentes. Esse compartilhamento ajuda a diminuir a quantidade de hardware necessária para o projeto. A capacidade de permitir que varias instruções sejam executadas "ao mesmo tempo" e a capacidade de compartilhar unidades funcionais dentro da execução de uma única instrução são as principais vantagens deste tipo de projeto.

1.1 – Caminho de dados

Abaixo podemos ver a visão em alto nível do caminho de dados multi-ciclo. Nela podemos ver os elementos-chave do caminho de dados: Uma unidade de memória compartilhada para instruções e para dados, uma única ALU compartilhada entre instruções (ao invés de uma ALU e dois somadores) e as conexões entre essas unidades compartilhadas.

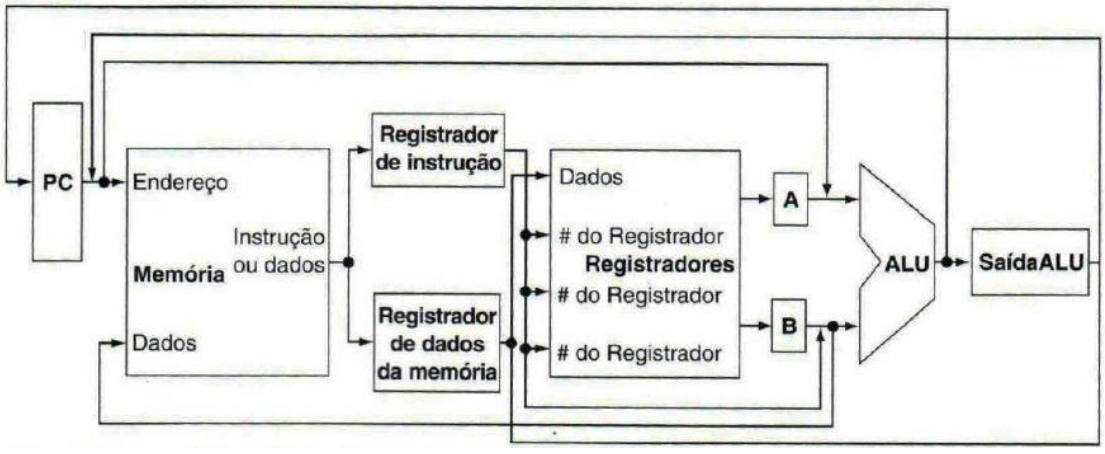

Figura 1 - Caminha de dados multi-ciclo

No projeto multi-ciclo, um ciclo de clock pode realizar no máximo uma das seguintes operações: um acesso a memória, um acesso ao banco de registradores ou uma operação na ALU. Com isso, qualquer dado produzido pela operação em uma dessas unidades precisa ser armazenado em um registrador para poder ser utilizada em um

ciclo posterior. Caso contrário, uma outra operação poderia sobrescrever o dado levando o uso de um dado incorreto.

Com isso, em relação a arquitetura monociclo, são acrescentados os seguintes registradores: Um registrador de instrução (IR) usado para salvar a saída da memória para uma leitura de instrução, um registrador de dados da memória (MDR) usado para salvar a saída da memória para uma leitura de dados, registradores “A” e “B” (figura 1) que são usados para conter os valores dos registradores operandos lidos do banco de registradores e por fim um registrador de saída da ALU que armazena os dados provenientes da ALU.

É importante notar que, exceto o registrador IR, todos os registradores contem apenas dados entre um par de ciclos de clock adjacentes, logo não precisam de um sinal de controle de escrita. Já o IR precisa conter a instrução até o fim da execução da instrução, e, portanto, ele precisa de um sinal de controle para a escrita.

Agora, como uma única memória é usada para instruções e para dados, é preciso acrescentar multiplexadores para selecionar para qual endereço de memória a informação deve ir (PC ou saída da ALU). Com isso, são acrescentados os seguintes multiplexadores: Um multiplexador na primeira entrada da ALU e escolhe entre o registrador A e o PC e outro multiplexador na segunda entrada da ALU que escolhe entre a constante 4 (usada para incrementar o PC) e o campo offset com um sinal estendido e deslocado (usado no calculo do endereço de desvio).

Abaixo podemos ver o caminho de dados completo com a adição dos registrados e multiplexadores necessários. Este caminho de dados já é bem mais completo que o dá figura 1, porém ele manipula apenas as instruções básicas do MIPS. A figura também mostra os sinais de controle que serão aplicados para os múltiplos ciclos de clock por instrução funcionarem corretamente.

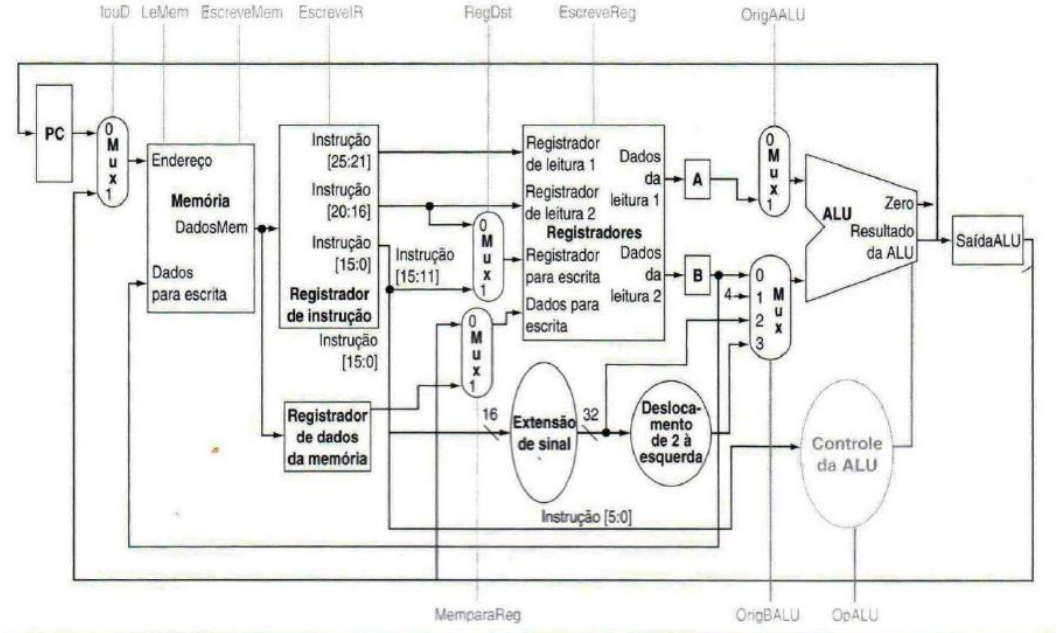

Figura 2 - Caminho de dados multi-ciclo com sinas de controle indicados

Este caminho de dados ainda necessita de alterações para suportar as operações de desvio e jump. Com a instrução jump e a instrução *branch*, o valor do PC pode ser escrito pelas seguintes unidades: A saída de ALU que escreve PC+4 durante a busca de instrução, o registrador saída ALU que armazena o endereço destino do desvio após ele ser calculado e os 26 bits menos significativos do registrador de instrução IR deslocados dois a esquerda.

O valor de PC pode ser escrito incondicionalmente e condicionalmente. Para um incremento normal ou um jump, o valor é escrito automaticamente. Se a instrução é um desvio condicional, o valor de PC é substituído pelo valor em saída ALU se os dois registradores designados forem iguais. Portanto, dois sinais de controle são utilizados, o *EscrevePC* que causa uma escrita incondicional e o *EscrevePCCond* que escreve no PC caso a condição de desvio seja verdadeira. Esses dois sinais são conectados por meio de um circuito combinacional que utiliza portas AND e OR determinar o caminho do dado. Abaixo podemos ver o caminho de dados completo para a implementação multicilco.

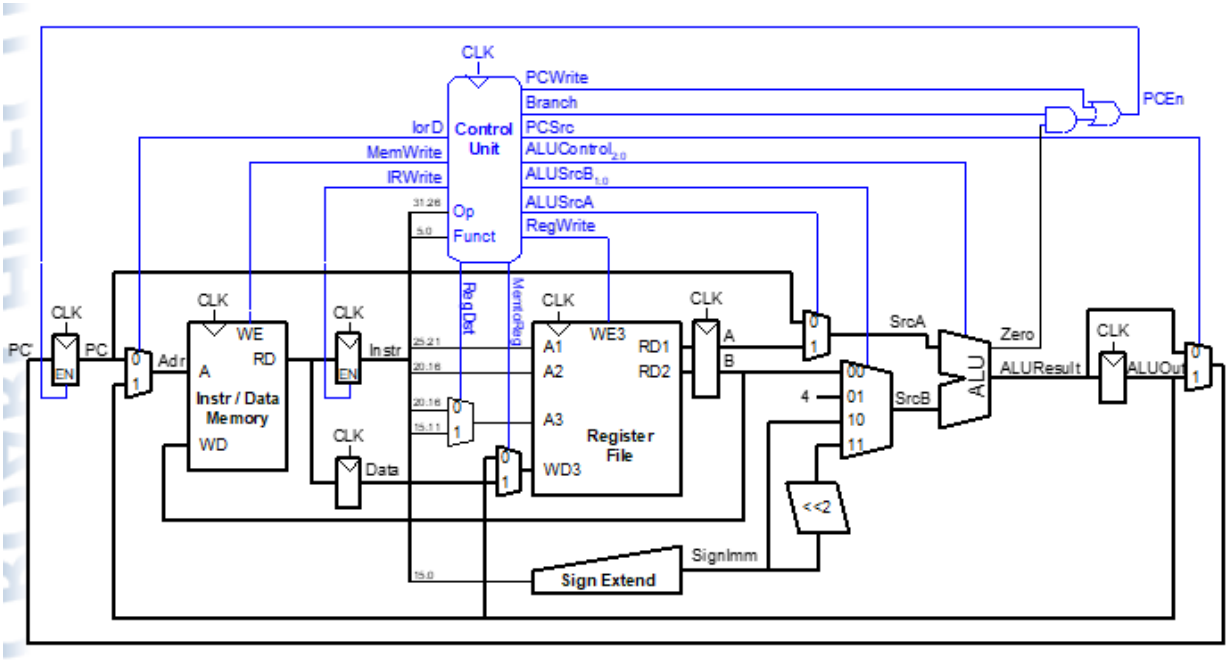

Figura 3 – Caminho de dados completo para a arquitetura multi-ciclo

1.2 – Etapas da execução

Agora que o caminho de dados está pronto, podemos descrever quais são as etapas da execução de uma instrução a cada ciclo de clock. Como cada etapa contém apenas uma operação, o ciclo de clock pode ser tão curto quanto a mais longa operação. Cada instrução no MIPS precisa de três a cinco etapas. A seguir, segue as etapas de execução e suas ações.

1.2.1 – Etapa de busca da instrução

Busca a instrução da memoria e calcula o endereço da próxima instrução. Nela, o valor de PC é enviado para a memoria como endereço. Após isso, a instrução é obtida através de uma leitura e escrita no registrador de instrução (IR). Além disso ela acrescenta o valor do PC em 4

IR <= Mem[PC]

PC <= PC +4

1.2.2 – Etapa de decodificação da instrução e busca dos registradores

Nesta etapa, lemos os valores dos registradores indicados pelos campos de instrução *rs* e *rt* e armazenamos nos registradores temporários A e B. Além disso, calculamos com a ALU o endereço de destino do desvio e salvamos *em SaidaALU.*

A <= Reg[IR[25:21]]

B <= Reg[IR[20:16]]

SaidaAlu <= PC + (sinal estendido IR[15-0] << 2)

1.2.3 – Execução, calculo do endereço de memoria ou conclusão do desvio

Nesta etapa a ALU realiza uma de quatro funções, dependendo da classe da instrução, atuando sobre os operandos preparados na etapa de decodificação e busca.

Referência à memória – SaidaAlu <= A + sinal estendido (IR[15:0])

Instrução logica ou aritmética – SaidaAlu <= A op B

Desvio – if(A==B) PC <= SaidaAlu

Jump – PC <= {PC [31:28], IR([25;0]], 2'b00) } (concatena PC com IR)

1.2.4 – Etapa de acesso à memória ou conclusão de instrução do tipo R

Nesta etapa uma instrução *load* ou store acessa a memória e uma instrução logica ou aritmética escreve o seu resultado. Quando um valor é lido da memória, ele é armazenado no registrador de dados da memória (MDR).

Referência a memória – MDR <= Mem[SaidaAlu] ou Mem[SaidaAlu] <= B

Instrução lógica ou aritmética do tipo R – Reg[IR[15:11]] <= SaidaAlu

1.3 – Controle de dados

A unidade de controle consiste em um conjunto de tabelas verdades que especificam a definição dos sinais de controle com base na classe da instrução. Para o caminho de dados multi-ciclo, o controle precisa especificar os sinas a serem definidos em qualquer etapa de execução, assim como a próxima etapa na sequência. Para especificar o

controle podem ser utilizados duas técnicas: A primeira é chamada de maquinas de estados finitas. A outra é chamada de microprogamação.

1.3.1 – Máquinas de estados finitas

A máquina de estados finitos consiste em um conjunto de estados e escolhas em como mudar de estado. Estas escolhas são definidas por uma função de próximo estado que mapeia o estado atual e as entradas para um novo estado.

Abaixo podemos ver uma versão de alto nível dessa máquina de estados finitos.

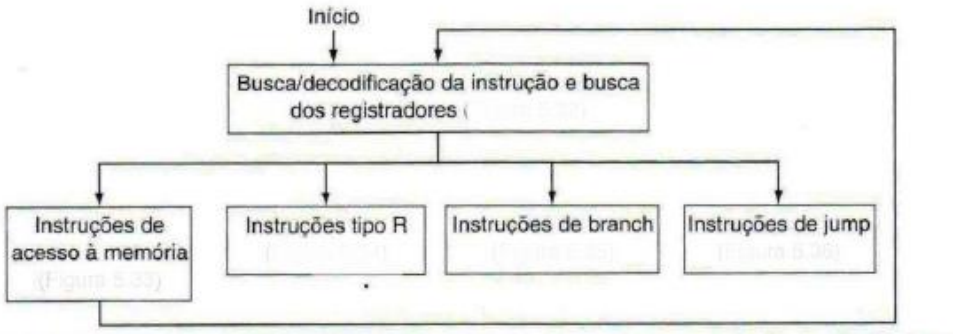

Figura 4 – Maquina de estados finita

Agora, iniciamos o estuda da máquina de estados finita mostrando os dois primeiros estados dela. O estado zero corresponde a etapa 1 mostrada anteriormente. Este é o estado inicial da máquina. Os sinais ativados em cada valor são mostrados dentro do círculo. Os próximos estados estão rotulados com condições que selecionam um próximo estado especifico quando mais de um próximo estado é possível. Após o estado 1, os sinais ativados dependem da classe de instrução. Logo, a maquina de estados finitos possui quatro próximos estados (que correspondem as quatro classes de instruções) partindo do estado 1.

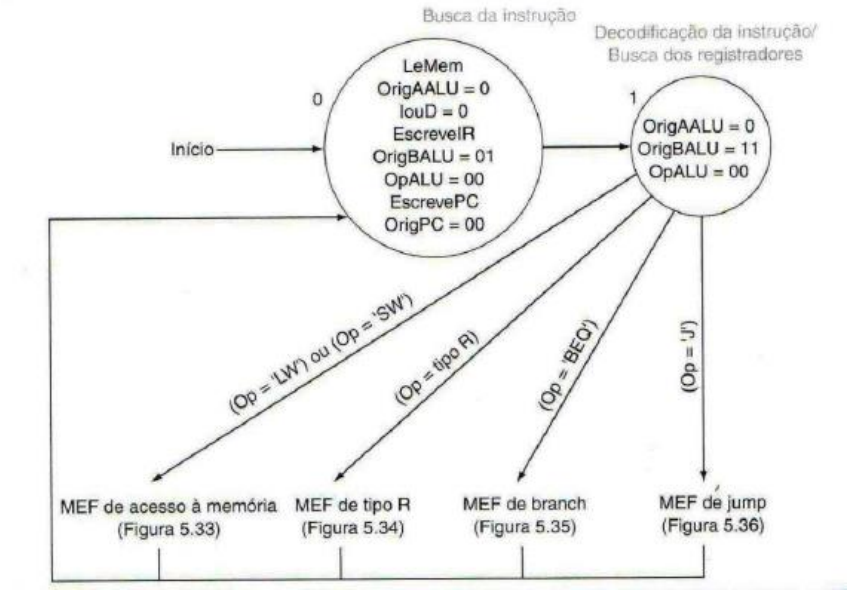

Figura 5 – Estes estados correspondem ao retângulo superior da figura 4

A próxima imagem mostra a parte da maquina de estados correspondente a parte necessária para implementar as instruções de acesso a memória. Para isso, o primeiro os registradores calculam o endereço de memoria (estado 2). Após este cálculo, a memória deve ser lida ou escrita (por isso existem dois estados possíveis). Se o *opcode* da instrução é l2 o estado 3 faz a leitura da memória. Se a instruções é *load,* um outro estado é necessário para escrever este resultado na memória (estado 4).

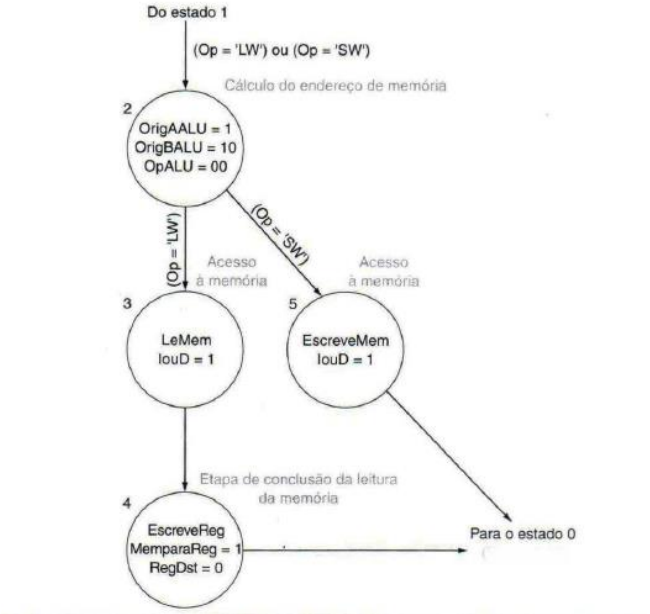

Figura 6 – Segunda parte da máquina de estados finita

As instruções do tipo R exigem dois estados correspondentes às etapas 3 e 4. A próxima figura mostra essa parte. Esses estados correspondem ao retângulo chamado de "instruções do tipo R" da figura 4. O primeiro estado faz com que a operação da ALU ocorra, enquanto o segundo estado faz com que o resultado da ALU seja escrito no banco de registradores.

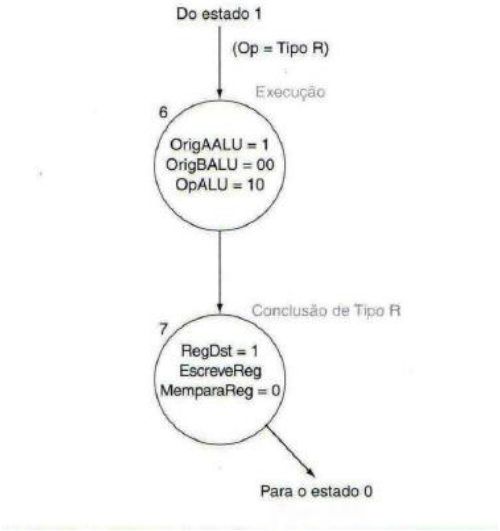

Figura 7 – Instruções do tipo R

As próximas partes da maquina de estados correspondem a instrução *Branch* e *Jump* que possuem apenas um estado cada.

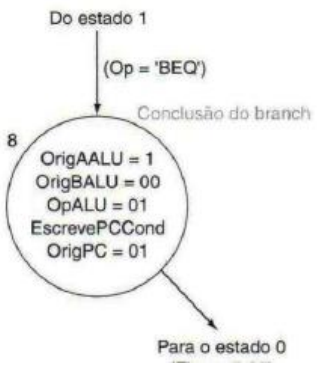

Figura 8 – Instrunção Branch

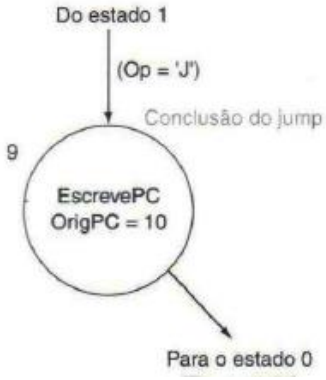

Figura 9 – Instrução Jump

Com isso, todo o controle do caminho de dados da arquitetura multiciclo foi implementado com a maquina de estados finita. Abaixo podemos ver a maquina de estados completa onde os "rotulos" em cada arco são condições testadas para determinar qual é o proximo estado. Também é importante lembrar que os rotulos detro de cada estado indicam os sinais de saida ativados durante aquele estado.

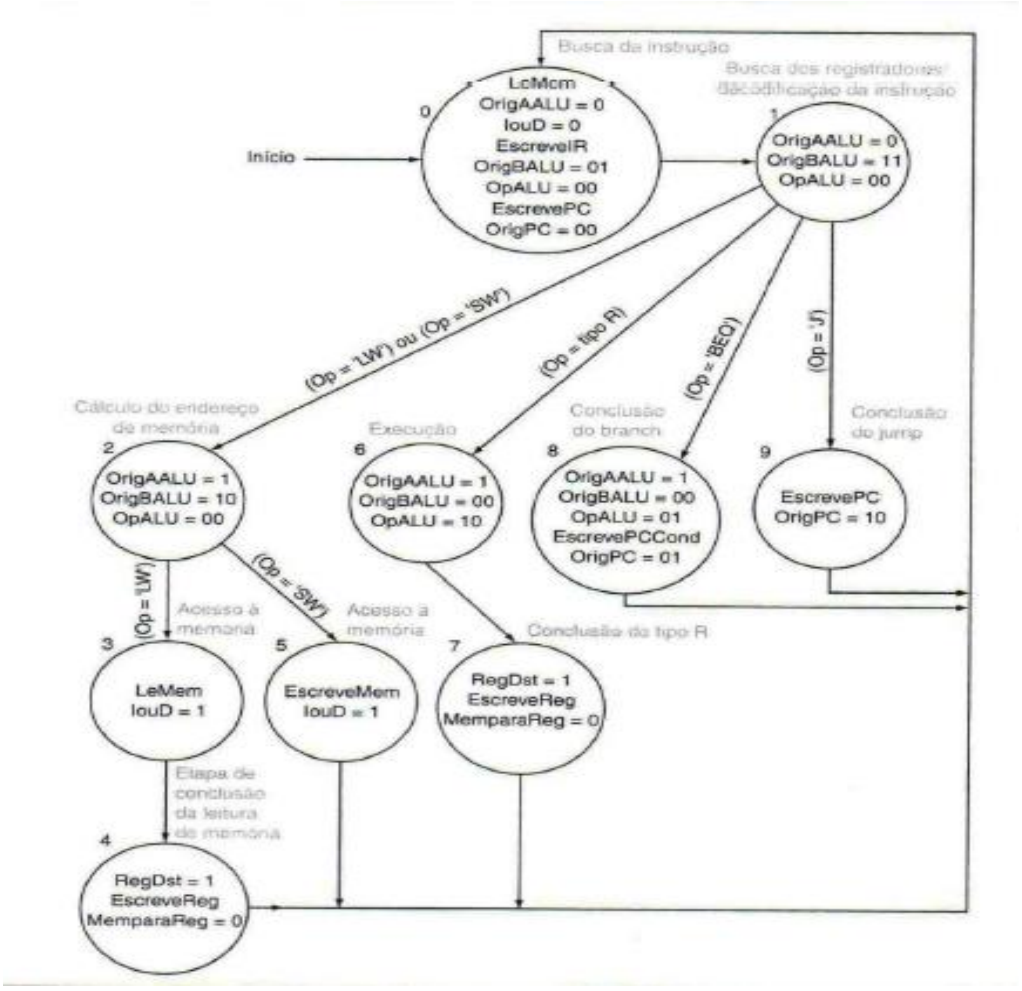

Figura 10 – Maquina de estados finita completa

2 – Conjunto de instruções do MIPS

Abaixo seguimos com o conjunto de instruções do MIPS, assim como sua descrição, operação, syntax e codificação. Os registradores de uso geral são indicados pelo sinal de dolar $.

ADD (com overflow)
Descrição: Soma dois registradores e guarda o resultado em um registrador
Operação: $d = $s + $t; advance_pc (4);
Sintaxe: add $d, $s, $t
Codificação: 0000 00ss ssst tttt dddd d000 0010 0000

ADDI imediato (com overflow)
Descrição: Soma um registrador a um valor imediato e armazena o resultado em um regsitrador
Operação: $t = $s + imm; advance_pc (4);
Sintaxe: addi $t, $s, imm
Codificação: 0010 00ss ssst tttt iiii iiii iiii iiii

ADDIU sem assinatura imediata (sem overflow)
Descrição: Soma um registrador a um valor imediato e armazena o resultado em um regsitrador
Operação: $d = $s + $t; advance_pc (4);
Sintaxe: aaddu $d, $s, $t
Codificação: 0000 00ss ssst tttt dddd d000 0010 0001

AND (Bit a Bit)
Descrição: Faz um and bit a bit de dois registradores e armazena em um registrador
Operação: $ d = $ s & $ t; advance_pc (4);
Sintaxe: e $ d, $ s, $ t
Codificação: 0000 00ss ssst tttt dddd d000 0010 0100

ANDI (Bit a Bit imediato)
Descrição: Faz um and bit a bit entre um regitrador e um valor imediato e guarda em um registrador
Operação: $ t = $ s & imm; advance_pc (4);
Sintaxe: andi $ t, $ s,imm
Codificação: 0011 00ss ssst tttt iiii iiii iiii iiii

BEQ (Branch on equal)
Descrição: Faz um branch se os dois registradores forem iguais
Operação: if $s == $t advance_pc (offset << 2)); else advance_pc (4);
Sintaxe: beq $s, $t, offset
Codificação: 0001 00ss ssst tttt iiii iiii iiii iiii

BGEZ (Branch em maior ou igual a zero)
Descrição: Faz um branch se os dois registradores forem maiores ou iguais a zero
Operação: if $s >= 0 advance_pc (offset << 2)); else advance_pc (4);
Sintaxe: bgez $s, offset
Codificação: 0000 01ss sss0 0001 iiii iiii iiii iiii

BGEZAL
Descrição: Faz um branch se os dois registradores forem maiores ou iguais a zero e salva o retorno no endereço $31
Operação: if $s >= 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2));
else advance_pc (4);
Sintaxe: bgezal $s, offset
Codificação: 0000 01ss sss1 0001 iiii iiii iiii iiii

BGTZ
Descrição: Faz um branch se o registrador for maior que zero
Operação: if $s > 0 advance_pc (offset << 2)); else advance_pc (4);
Sintaxe: bgtz $s, offset
Codificação: 0001 11ss sss0 0000 iiii iiii iiii iiii

BLEZ
Descrição: Faz um branch se o registrador for menor ou igual a zero
Operação: if $s <= 0 advance_pc (offset << 2)); else advance_pc (4)
else advance_pc (4);
Sintaxe: blez $s, offset
Codificação: 0001 10ss sss0 0000 iiii iiii iiii iiii

BGEZAL
Descrição: Faz um branch se o registrador é menor que zero
Operação: if $s < 0 advance_pc (offset << 2)); else advance_pc (4)
else advance_pc (4);
Sintaxe: bltz $s, offset
Codificação: 0000 01ss sss0 0000 iiii iiii iiii iiii

BNE
Descrição: Faz um branch se os dois registradores não forem iguais a zero
Operação: if $s != $t advance_pc (offset << 2)); else advance_pc (4)
Sintaxe: bne $s, $t, offset
Codificação: 0001 01ss ssst tttt iiii iiii iiii iiii

DIV
Descrição: Divide $s por $t e armazena o quociente em $LO e o restante em $HI
Operação: $LO = $s / $t; $HI = $s % $t; advance_pc (4);
Sintaxe: div $s, $t
Codificação: 0000 00ss ssst tttt 0000 0000 0001 1010

DIVU
Descrição: Divide $s por $t e armazena o quociente em $LO e o restante em $HI
Operação: $LO = $s / $t; $HI = $s % $t; advance_pc (4);
Sintaxe: divu $s, $t
Codificação: 0000 00ss ssst tttt 0000 0000 0001 1011

JUMP
Descrição: Pula para o endereço calculado
Operação: PC = nPC; nPC = (PC & 0xf0000000) | (target << 2)
Sintaxe: j target
Codificação: 0000 10ii iiii iiii iiii iiii iiii iiii

JR
Descrição: Pula para o endereco contido no registrador $s
Operação: PC = nPC; nPC = $s;
Sintaxe: jr $s
Codificação: 0000 00ss sss0 0000 0000 0000 0000 1000

LW (Load Word)
Descrição: Uma palavra é carregada no registrador especificado no endereço
Operação: $t = MEM[$s + offset]; advance_pc (4);
Sintaxe: lw $t, offset($s)
Codificação: 1000 11ss ssst tttt iiii iiii iiii iiii

MULT (Multiplica)
Descrição: Multiplica $s por $t e guarda o resultado em $LO
Operação: $LO = $s * $t; advance_pc (4);
Sintaxe: mult $s, $t
Codificação: 0000 00ss ssst tttt 0000 0000 0001 1000

NOOP
Descrição: Não realiza nenhuma operação
Operação: advance_pc (4);
Sintaxe: noop
Codificação: 0000 0000 0000 0000 0000 0000 0000 0000

OR (bit a bit)
Descrição: Faz um or bit a bit e guarda o resultado em um registrador
Operação: $d = $s | $t; advance_pc (4);
Sintaxe: or $d, $s, $t
Codificação: 0000 00ss ssst tttt dddd d000 0010 0101

SB (store byte)
Descrição: O byte menos significativo de $ t é armazenado no endereço especificado.
Operação: MEM[$s + offset] = (0xff & $t); advance_pc (4);
Sintaxe: sb $t, offset($s)
Codificação: 1010 00ss ssst tttt iiii iiii iiii iiii

SLL (shift left logico)
Descrição: Desloca um valor de registro deixado pelo valor do turno listado na instrução e coloca o resultado em um terceiro registro. Zeros são deslocados em
Operação: $d = $t << h; advance_pc (4);
Sintaxe: sll $d, $t, h
Codificação: 0000 00ss ssst tttt dddd dhhh hh00 0000

SRA (shift right aritmético)
Descrição: Desloca um valor de registro diretamente pelo valor de deslocamento e coloca o valor no registro de destino.
Operação: $d = $t >> h; advance_pc (4);
Sintaxe: sra $d, $t, h
Codificação: 0000 00-- ---t tttt dddd dhhh hh00 0011

SRL (shift right logico)
Descrição: Desloca um valor de registro diretamente pelo valor de deslocamento e coloca o valor no registro de destino.
Operação: $d = $t >> h; advance_pc (4);
Sintaxe: srl $d, $t, h
Codificação: 0000 00-- ---t tttt dddd dhhh hh00 0010

SUB
Descrição: Subtrai dois registradores e salva o resultado em um registrador
Operação: $d = $s - $t; advance_pc (4);
Sintaxe: sub $d, $s, $t
Codificação: 0000 00ss ssst tttt dddd d000 0010 0010

SW (Store word)
Descrição: O conteúdo de $ t é armazenado no endereço especificado.
Operação: MEM[$s + offset] = $t; advance_pc (4);
Sintaxe: sw $t, offset($s)
Codificação: 1010 11ss ssst tttt iiii iiii iiii iiii

XOR (xor bit a bit)
Descrição: Faz a operação xor entre dois registradores e salva o resultado em um registrador
Operação: $d = $s ^ $t; advance_pc (4);
Sintaxe: xor $d, $s, $t
Codificação: 0000 00ss ssst tttt dddd d--- --10 0110