Pontifícia Universidade Católica do Rio Grande do Sul
Arquitetura de Computadores – FACIN
Gabriel F. Susin
Augusto Bergamin

Trabalho Prático – Parte 2

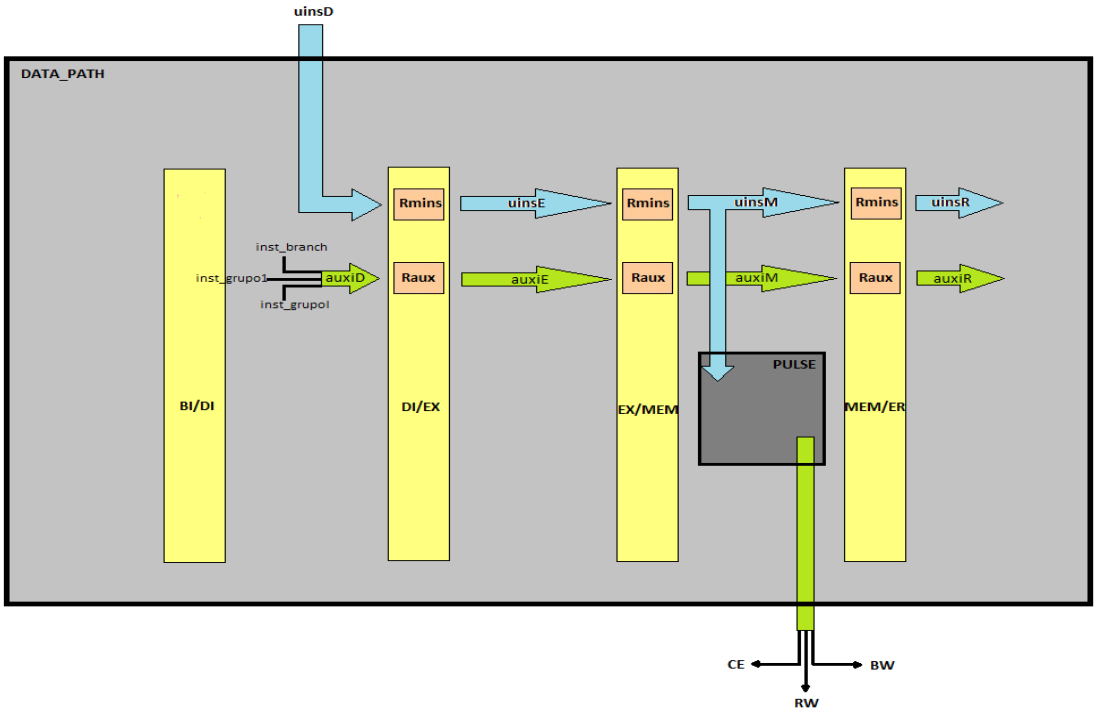
Introdução

No decorrer deste documento estaremos apresentando uma adaptação do MIPS multiciclo que vinha sendo estudado anteriormente. Trata-se da modificação na arquitetura do projeto para fazer com que se torne um modelo pipeline. Temos então como tarefa principal, introduzir barreiras de registradores entre os estágios do projeto. Alem disso, viu-se necessario a retirada da maquina de estados do bloco de controle, fazendo a implatanção de uma lógica combinacional em seu lugar. Para melhor entendimento das mudanças realizadas, será reapresentada a arquitetura do processador, assim como os testes realizados para sua validação e mudanças significativas apresentadas pelo mesmo em comparação direta com o multiciclo utilizado na etapa anterior do projeto.

Arquitetura

O MIPS em sua forma multiciclo funciona com base em uma máquina de estados que comanda o andamento das instruções ao longo dos ciclos. Já no pipeline teremos barreiras de registradores, comandados por um clock, que estarão atuando na propagação das instruções pelo processador. Essas barreiras foram criadas dentro do projeto, sendo implementadas como um par entidade/arquitetura cada uma. São elas: BI/DI, DI/EX, EX/MEM, MEM/ER , sendo BI a busca de instruções, DI a decodificação da instrução, EX a sua execução, MEM o acesso à memória e ER a escrita no banco de registradores.

A seguir podemos ver um diagrama de blocos exemplificando isso:



Dessa forma teremos uma parte de uma instrução em cada setor do processador, aumentando bastante a eficiencia dele pois a vazão(throughput) de instruções será muito maior, priniplamente quando comparado com o multiciclo que para iniciar uma nova intrução necessitava acabar a anterior.

Dado a fato de que as barreiras são compostas de registradores, foi necessário a implementação de dois novos registradores, um para microinstruções e outro para sinais. Além disso tivemos de fazer algumas modificações no formato do registrador "regnbit" para possibilitar o uso do "mestre-escravo" assim como poder mudar o tamanho do registrador nos usando "generic maps". Ficando dessa forma:

```
49
50  --+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
51  -- Generic multibit register
52  --+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
53  library IEEE;
54  use IEEE.std_logic_1164.all;
55
56  entity regnbit is
57            generic(REG_SIZE : integer := 31; INIT_VALUE : STD_LOGIC_VECTOR(31 downto 0) := (others=>'0');  SENSIBILITY : std_logic := '1');
58            port(  ck, rst, ce : in std_logic;
59                   D : in  STD_LOGIC_VECTOR (REG_SIZE downto 0);
60                   Q : out STD_LOGIC_VECTOR (REG_SIZE downto 0)
61                 );
62  end regnbit;
63
64  architecture regnbit of regnbit is
65  begin
66
67    process(ck, rst)
68    begin
69         if rst = '1' then
70               Q <= INIT_VALUE(REG_SIZE downto 0);
71         elsif ck'event and ck = SENSIBILITY then
72             if ce = '1' then
73                Q <= D;
74             end if;
75         end if;
76    end process;
77
78  end regnbit;
79
```

Seguido dos dois registradores genericos criados, para sinal e microinstrução, respectivamente:

```
80  --+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
81  -- Generic bit register
82  --+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
83  library IEEE;
84  use IEEE.std_logic_1164.all;
85
86  entity regbit is
87             generic(SENSIBILITY : STD_LOGIC := '1'; INIT_VALUE : STD_LOGIC := '0');
88             port(  ck, rst, ce : in std_logic;
89                    D : in  STD_LOGIC;
90                    Q : out STD_LOGIC );
91  end regbit;
92
93  architecture regbit of regbit is
94  begin
95
96    process(ck, rst)
97    begin
98         if rst = '1' then
99               Q <= INIT_VALUE;
100        elsif ck'event and ck = SENSIBILITY then
101            if ce = '1' then
102               Q <= D;
103            end if;
104        end if;
105   end process;
106
107 end regbit;
108
```

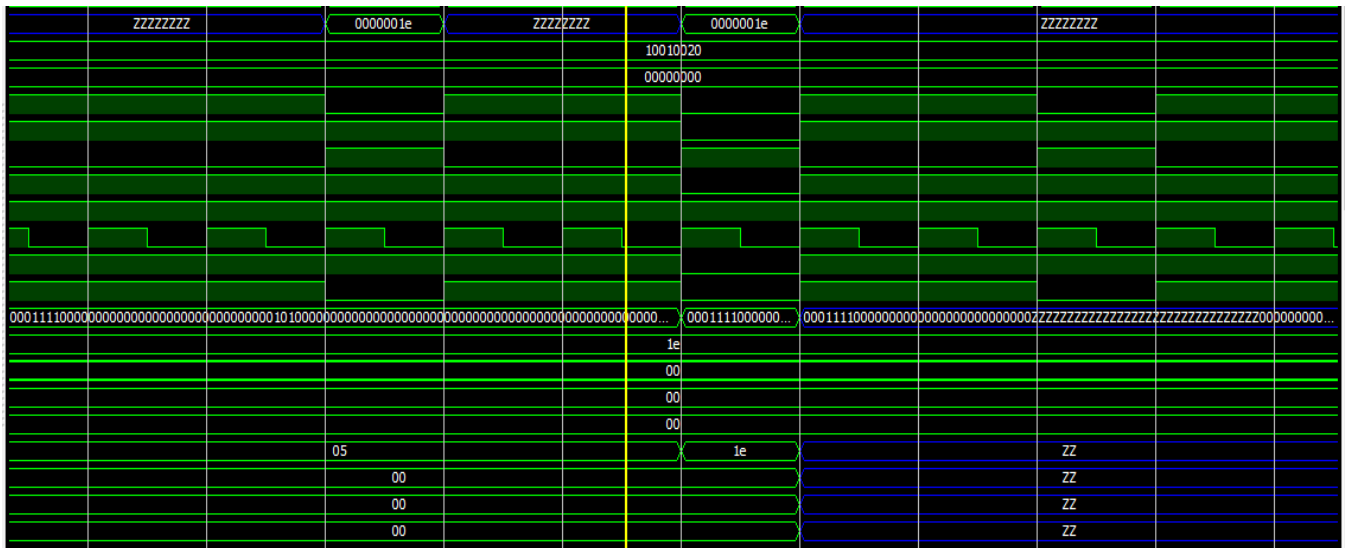
```
112  library IEEE;
113  use IEEE.std_logic_1164.all;
114  use work.p_MRstd.all;
115
116  entity regmins is
117            generic(SENSIBILITY : STD_LOGIC := '1'; INIT_VALUE : inst_type := invalid_instruction);
118            port(  ck, rst, ce : in std_logic;
119                   D : in  microinstruction;
120                   Q : out microinstruction );
121  end regmins;
122
123  architecture regmins of regmins is
124  begin
125
126    process(ck, rst)
127    begin
128         if rst = '1' then
129                Q.i <= INIT_VALUE;
130                Q.CY1 <= '0';
131                Q.CY2 <= '0';
132                Q.walu <= '0';
133                Q.wmdr <= '0';
134                Q.wpc <= '0';
135                Q.wreg <= '0';
136                Q.ce <= '0';
137                Q.rw <= '0';
138                Q.bw <= '1';
139         elsif ck'event and ck = SENSIBILITY then
140             if ce = '1' then
141                Q <= D;
142             end if;
143         end if;
144    end process;
145
146  end regmins;
```

Uma das principais mudanças feitas foi a retirada da maquina de estados no bloco de controle pois no pipeline o andamento das instruções no processador é dado pelas barreiras. Assim no bloco de controle temos apenas um código combinacional que define os sinais do “uins” além da decodificação da instrução que não foi alterada.

```
724      ---------------------------------------------------------------------------------------
725      -- BLOCK (2/3) - DATAPATH REGISTERS load control signals generation.
726      ---------------------------------------------------------------------------------------
727      uins.CY1   <= '1';
728
729      uins.CY2   <= '1';
730
731      uins.walu  <= '1';
732
733      uins.wmdr  <= '1' when i=LBU  or i=LW else '0';
734
735      uins.wreg   <= '0' when i=SB or i=SW or i=BEQ or i=BGEZ or i=BLEZ or i=BNE or i=J or i=JR or i=NOP else   '1';
736
737      uins.rw    <= '0' when i=SB or i=SW else  '1';
738
739      uins.ce    <= '1' when i=LBU  or i=LW or i=SB or i=SW  else '0';
740
741      uins.bw    <= '0' when i=SB or i=LBU  else '1';
742
743      uins.wpc   <= '1';
744
745  end control_unit;
746
```

Para cada um dos estágios foi necessario criar um “uins” diferente, assim como todos os dados que precisavam ser passados a diante. Isso se da pois cada um dos estágios estará com instruções diferentes e por consequencia os sinais mudam também.

Durante o desenvolvimento do processador notou-se problemas para executar a instrução SW (store word) e SB (store byte). Quando a instrução está no segmento responsável por uso da memória, é fornecido ao sinal “data” o valor do registrador que será salvo na memória, e é ativo o os sinais de controle da memória (ce, rw, bw), estes permanecem ativos durante todo um ciclo de clock. Mas como a memória RAM do projeto é assíncrona, com a troca de instrução o valor de “data” era reiniciado adquirindo o valor (ZZZZZZZZ) e durante este curto período de tempo a RAM salvava este valor por cima do valor esperado.



Para contorno deste problema foi criado uma entidade que faz o controle dos sinais que serão enviados para a memória RAM, ele é composto por uma máquina de estados que faz o controle do sinal “ce” para ativar ou não a memória. Quando é detectado que a instrução é um store, o sinal “ce” é acionado e logo depois na decida do clock é desativado para que não seja substituído o valor salvo.

Isto foi implementado da seguinte forma:

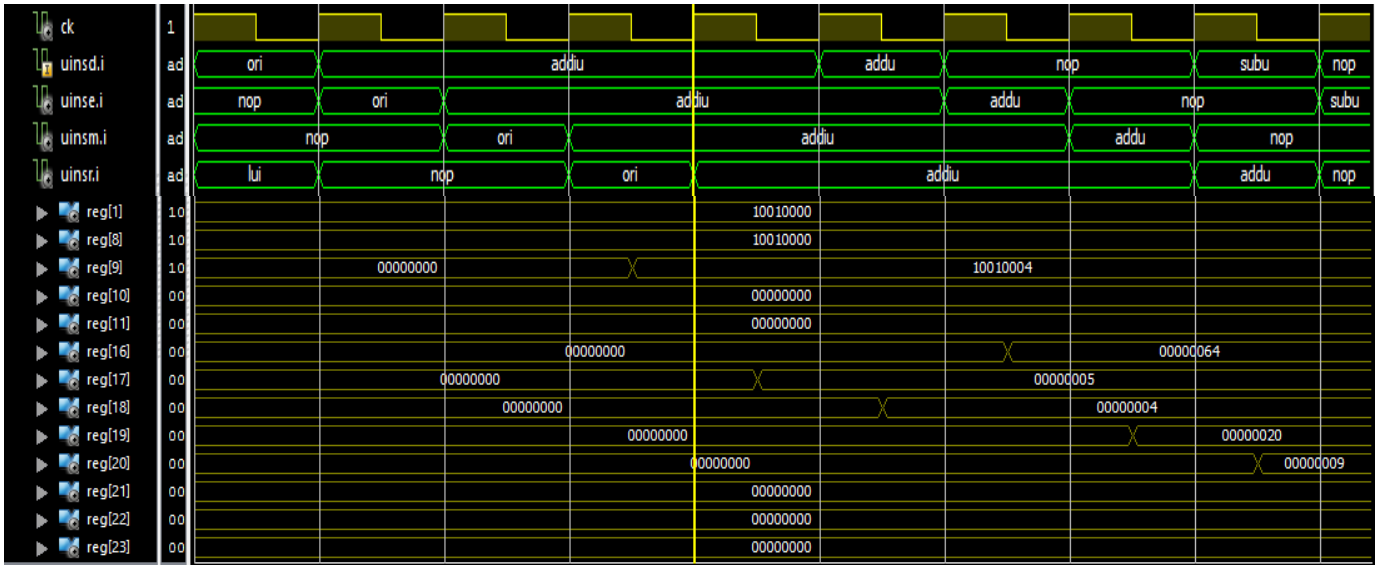
```
155  entity pulsesig is
156            port(  ck, rst : in std_logic;
157                   D : in  microinstruction;
158                   Q : out STD_LOGIC_VECTOR (2 downto 0));
159  end pulsesig;
160
161  architecture pulsesig of pulsesig is
162     type type_state is (Sstore, Sload, Send);
163     signal PST, NST : type_state;
164  begin
165
166    process(rst, ck)
167      begin
168         if rst='0' then
169              PST <= Sstore;
170         elsif ck'event then
171
172              PST <= NST;
173
174         end if;
175     end process;
176
177      process(PST, D.i)
178      begin
179         case PST is
180
181              when Sstore => if D.i = SB or D.i = SW then
182                                NST <= Send;
183                             elsif D.i = LBU or D.i = LW then
184                                NST <= Sload;
185                             else
186                                NST <= Sstore;
187                             end if;
188
189              when Sload => NST <= Sstore;
190              when Send =>  NST <= Sstore;
191         end case;
192      end process;
193
194      Q(2) <= '1' when (PST = Sstore and (D.i = SW or D.i = SB or D.i = LBU or D.i = LW)) or PST = Sload else '0';
195      Q(1) <= D.rw;
196      Q(0) <= D.bw;
197
198  end pulsesig;
```

Validação

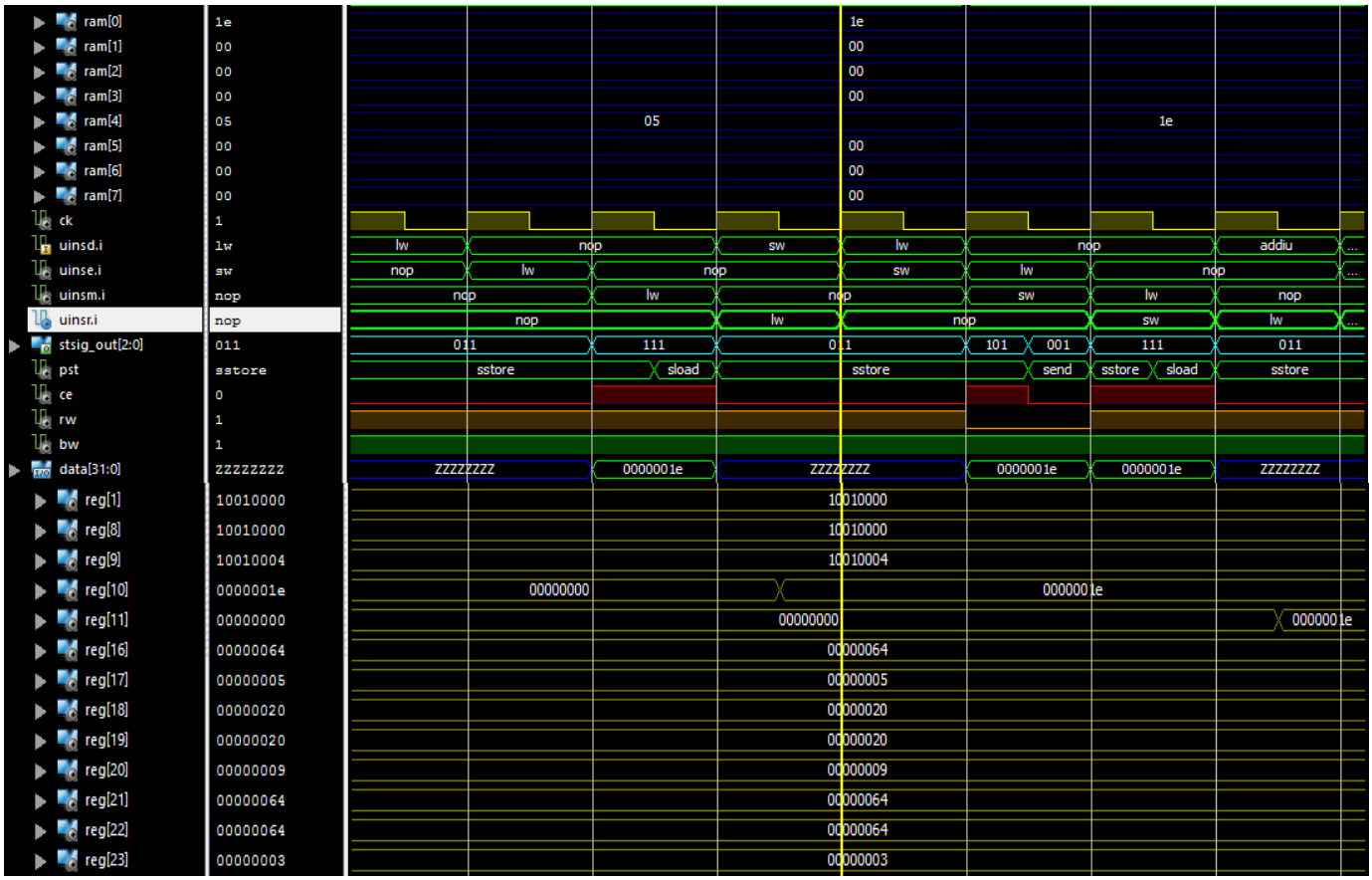
Assim como na parte anterior a validação do código se da por meio de testbenchs. O código a ser validado é feito em assambly, utilizando o MARS, e é feito o dump da memoria para o testbench no ISE. O código em questão é o mesmo utilizado na parte 1 do trabalho porém dessa vez foi necessário colocar “nop” entre algumas instruções já que a parte de conflitos do processador ainda não foi implementada.

```
main:
        lui     $at,0x00001001
        nop
        nop
        ori     $t0,$at,0x00000000
        lui     $at,0x00001001
        nop
        nop
        ori     $t1,$at,0x00000004


        li      $s1,5
        li      $s2,4
        li      $s0,100
        li      $s3,32

        addu    $s4,$s1,$s2
        nop
        nop
        subu    $s5,$s0,$s4
        nop
        nop
        addu    $s6,$s4,$s5
loop:
        beq     $s2,$s3,loop2
        nop
        nop
        nop
        nop
        sll     $s2,$s2,1
        addiu   $s7,$s7,1
        j       loop
        nop
        nop
        nop
        nop

loop2:
        addiu   $s5,$s5,1
        nop
        nop
        bne     $s5,$s0,loop2
        nop
        nop
        nop
        nop

fim:
        lw      $t2,0($t0)
        nop
        nop
        sw      $t2,0($t1)
        lw      $t3,0($t1)
        nop
        nop
        addiu   $t3,$t3,8
        nop
        nop
        sw      $t3,0($t0)
end:
        j       end

        .data

A:      .word   30
B:      .word   5
```

Uma vez feita a simulação com o ISim destaca-se alguns pontos importantes a se observar, ordenando os “uins.i” (que carregam o nome das instruções), o formato de pipeline já em atuação, muito parecido com os diagramas feitos em aula.



Um dos pontos mais críticos na composição do pipeline é a execução correta dos loads e stores. É preciso vários sinais nos seus valores corretos, como pode ser visto abaixo:



Avaliação

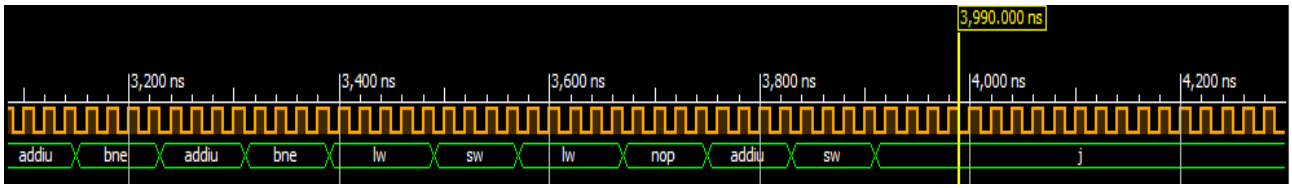
Já em uma visão teórica de pipeline é possível ver que teria-se um ganho de desempenho em relação ao multiciclo. Como já foi falado anteriormente para um modelo pipeline temos uma vazão de instruções muito mais ampla, isso se da pelo paralelismo imposto no modelo atual que aproveita as unidades de forma mais inteligente.

Para que possamos ver melhor o ganho de desempenho do pipeline em relação ao multiciclo, fizemos dois códigos assembly semelhantes para serem rodados nos dois modelos. Fala-se em semelhantes pois no caso do pipeline foi necessario colocar os “nops” para evitar os conflitos e se mantivessemos eles no multiciclo não teriamos uma comparação justa, afinal queremos o melhor desempenho de cada um deles.

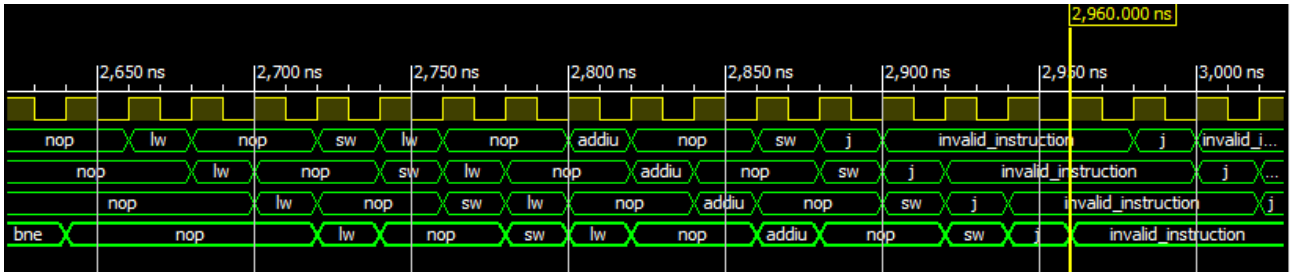
Executando ambos os assemblys em seus devidos processadores podemos ver claramente que o mips pipeline é mais eficiente, acabando todas as instruções 940.000 ns antes. Isso se torna ainda mais surprientente quando observamos que ainda não foi implementado tratamento de conflitos assim como previsão se salto fazendo com que em cada salto sejam usados 4 nops e assim aumentando ainda mais o tempo de execução.

Segue os dois testbenchs usados para comparação nas suas ultimas instruções:

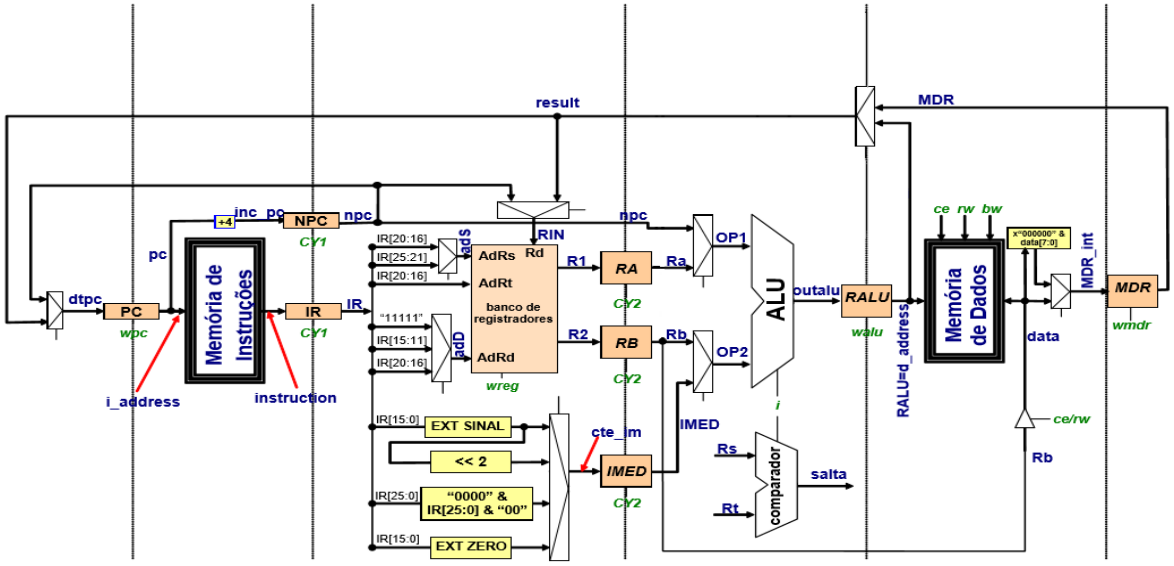
MULTICICLO



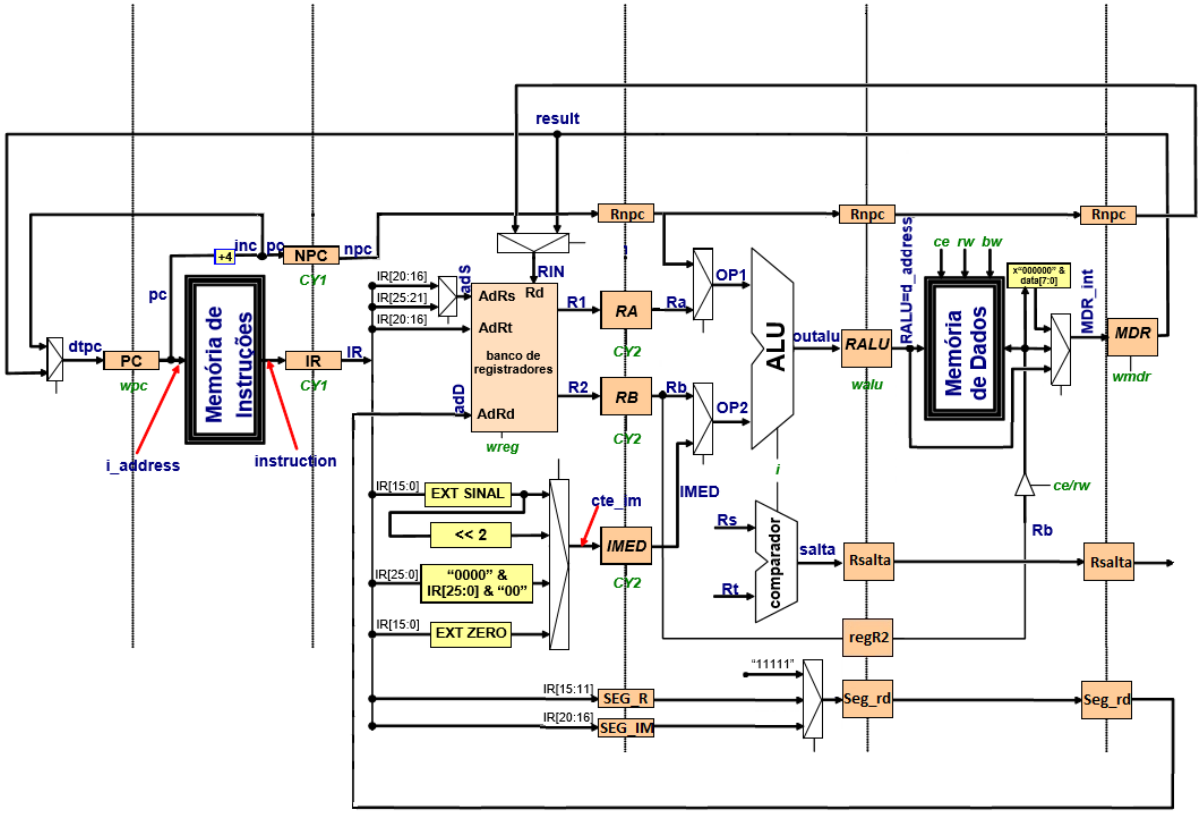
PIPELINE



Para termos uma visão mais clara da diferença entre a arquitetura dos dois é interessante observar os dois diagramas de blocos:



E do pipeline:



Com isso é possível ver mais claramente as mudanças de arquitetura feitas e descritas ao longo deste documento.