

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

DCC819 - Arquitetura de Computadores

Relatório V - Pipeline

Iuri Silva Castro
João Mateus de Freitas Veneroso
Ricardo Pagoto Marinho

BELO HORIZONTE - MG
5 DE DEZEMBRO DE 2017

1 Introdução

O *pipeline* é uma técnica de *hardware* para promover paralelismo à nível de instrução dentro de um processador. O objetivo da técnica é dividir a execução da instrução em estágios, de forma que, quando a instrução termina o estágio, a próxima já pode ser processada por esse estágio, mantendo então todos os estágios do processador ocupados com alguma instrução pelo máximo de tempo possível. Essa técnica se assemelha a uma linha de montagem, permitindo aumentar consideravelmente o *throughput* do processador em comparação à execução puramente sequencial, pois várias tarefas podem ser executadas em um mesmo ciclo de *clock*.

No entanto, a técnica de *Pipelining* complexifica o controle do processador, uma vez que a execução paralela introduz *Hazards* no caminho de dados, que não existiriam no caso da execução sequencial, como:

- *Hazards Estruturais*: restrições no número de instruções que podem utilizar um módulo do processador ao mesmo tempo, pois apenas uma instrução pode utilizar uma unidade funcional por vez. Pode ser amenizado com o aumento do número de unidades funcionais;
- *Hazards de Dados*: dependência de dados entre instruções. A instrução depende do resultado de uma instrução que ainda não terminou de executar. Pode ser amenizado com técnicas de encaminhamento de dados dentro dos estágios da pipeline;
- *Hazards de Controle*: instruções que fazem desvio do fluxo do programa, alterando o Contador de Programa (*Program Counter*), tornam as próximas instruções indefinidas até que o novo valor do *Program Counter* seja definido/calculado. Pode ser amenizado com técnicas de previsão de *branches*.

Os *Hazards*, quando ocorrem, necessitam que seja introduzido no fluxo do *pipeline* bolhas, ou *stalls*, para resolver esses conflitos.

Para este trabalho, propõe-se a implementação de um *Pipeline* de três estágios sobre o caminho de dados implementado nos trabalhos anteriores. O processador de 16-bits finalizado faz encaminhamento de dados e gera dois ciclos de *stall* ao executar instruções de *branch* e *jump*.

2 Descrição

Nessa seção, descreve-se a organização e a arquitetura do processador proposto neste trabalho.

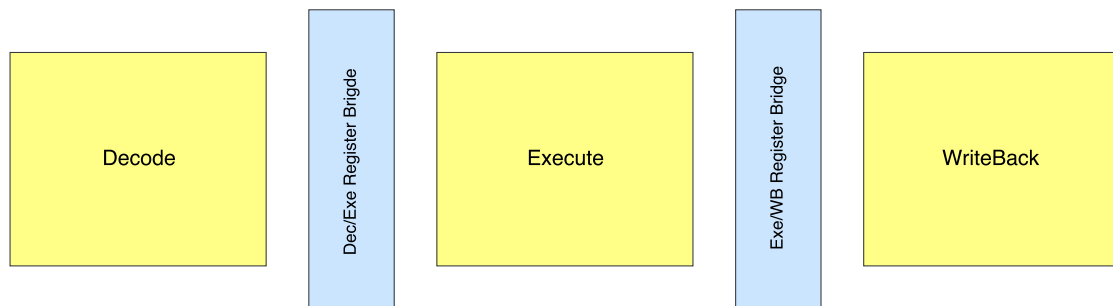
2.1 Organização

O processador desenvolvido nos trabalhos anteriores possuía 5 estágios de execução, sendo, busca de instrução, decodificação, busca de registradores, execução e armazenamento de resultados. Cada estágio requeria um passo de relógio, ou uma transição do sinal de *clock*, e não possuía qualquer paralelismo a nível de instrução.

Para a implementação do *pipeline*, propôs-se uma divisão do processador em 3 estágios: decodificação, execução e armazenamento de resultados. A Figura 1 abaixo mostra a divisão dos estágios.

Entre os estágios estão as *register bridges*, ou registradores de ponte, que são utilizados para passar as informações e sinais de controle de um estágio para o outro. Os estágios, então, serão responsáveis pelas seguintes tarefas:

- *Decode*: busca de instrução e decodificação de instrução;
- *Execute*: busca de registros e execução;
- *WriteBack*: escrita de resultados.

Figura 1: Estrutura de *pipeline* de 3 estágios proposta.

Para aplicação do *pipeline* necessita-se que os estágios utilizem o mesmo tempo de execução, assim cada estágio executa em duas transições do sinal de *clock*, mesmo o estágio de *WriteBack* que ficará então uma transição *idle* para adequar ao tempo dos outros estágios.

O *pipeline* insere no sistema *Hazards*, como descrito anteriormente, e para isso precisa-se utilizar de algumas técnicas para eliminar ou amenizar o problema. Eliminou-se o *hazard* de dados utilizando a técnica de encaminhamento, assim dados que são encaminhados da saída do estágio de execução para a entrada do mesmo estágio, não havendo *stalls*. O sistema não possui *hazards* estruturais, pois todos os estágios requerem o mesmo tempo de execução e o *Banco de Registradores* possui duas portas de leitura e uma de escrita. Além disso o estágio de execução possui uma unidade dedicada para executar multiplicações, conseguindo, assim, executar instruções de multiplicação em um ciclo. Para instruções de desvio de fluxo *hazards* de controle acontecem, e são gerados dois *stalls* quando tais instruções são detectadas.

2.2 Arquitetura

O conjunto final de instruções do processador está descrito na tabela 1. Todas as instruções aritméticas e lógicas recebem o operando A do banco de registradores e o operando B pode ser um imediato de 4-bit ou um registrador, dependendo da instrução. A instrução *BEZ* não utiliza os bits 11-8 e as instruções *GHI* e *GLO* não utilizam os bits de 7-0. A instrução *J* altera o *PC* para um valor imediato de 12-bit que comporta qualquer endereço da memória de 4096 posições.

Instrução	Opcode	Bits 11-8	Bits 7-4	Bits 3-0	Descrição
ADD	0000	C	B	A	$\text{Reg}(C) = \text{Reg}(A) + \text{Reg}(B)$
SUB	0001	C	B	A	$\text{Reg}(C) = \text{Reg}(A) - \text{Reg}(B)$
SLTI	0010	C	Imm	A	$\text{Reg}(C) = \text{Reg}(A) > \text{Imm}$
AND	0011	C	B	A	$\text{Reg}(C) = \text{Reg}(A) \text{ AND } \text{Reg}(B)$
OR	0100	C	B	A	$\text{Reg}(C) = \text{Reg}(A) \text{ OR } \text{Reg}(B)$
XOR	0101	C	B	A	$\text{Reg}(C) = \text{Reg}(A) \text{ XOR } \text{Reg}(B)$
ANDI	0110	C	Imm	A	$\text{Reg}(C) = \text{Reg}(A) + \text{Imm}$
ORI	0111	C	Imm	A	$\text{Reg}(C) = \text{Reg}(A) \text{ OR } \text{Imm}$
XORI	1000	C	Imm	A	$\text{Reg}(C) = \text{Reg}(A) \text{ XOR } \text{Imm}$
ADDI	1001	C	Imm	A	$\text{Reg}(C) = \text{Reg}(A) + \text{Imm}$
SUBI	1010	C	Imm	A	$\text{Reg}(C) = \text{Reg}(A) - \text{Imm}$
J	1011	Imm			$\text{PC} = \text{Imm}$
BEZ	1100	-	B	A	If $(\text{Reg}(A) = 0)$ $\text{PC} = \text{Reg}(B)$
MUL	1101	C	B	A	$\text{Reg}(C) = \text{Reg}(A) * \text{Reg}(B)$
GHI	1110	C	-	-	$\text{Reg}(C) = \text{HI}$
GLO	1111	C	-	-	$\text{Reg}(C) = \text{LO}$

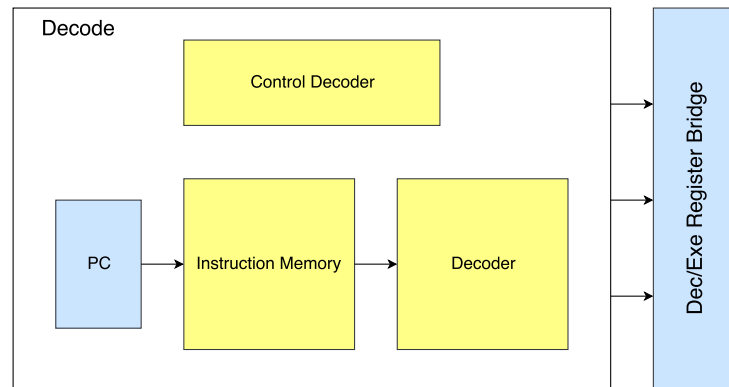
Tabela 1: Instruções

3 Implementação

Foram reutilizados os módulos desenvolvidos nos trabalhos anteriores, fazendo-se apenas pequenas alterações nos módulos. A maior parte das alterações foram feitas nos caminhos dos dados, utilizando multiplexadores e os registros de ponte entre os estágios. As subseções abaixo descrevem a implementação dos três estágios definidos.

3.1 Decode

O estágio de decodificação mantém os módulos da memória de instruções e do decodificador de instruções, além do registrador Contador de Programa. Uma máquina de estados é utilizada para controlar o funcionamento do estágio. A Figura 2 mostra uma versão simplificada de como é a organização do estágio.

Figura 2. Diagrama simplificado do estágio *Decode* do *pipeline*.

No primeiro ciclo de *clock* busca-se a instrução na memória de instruções. No segundo ciclo, atualiza-se o valor do *PC* e carrega os dados no registrador de ponte para o estágio de execução (*Execute*). Para os casos em que o estágio estiver em *stall*, o *PC* não é atualizado e nem o registrador de ponte, até que o controle saia da condição de *stall*.

A seleção entre os valores do *PC* será entre o incremento ($PC + 1$) ou, caso haja uma condição de desvio (*branch* ou *jump*), um valor externo.

3.2 Execute

O estágio de execução é responsável pela busca de registradores no *Banco de Registradores* e a execução da instrução pela *Unidade Lógica Aritmética* (ULA) ou pela unidade de multiplicação. A Figura 3 mostra uma versão simplificada de como é a organização do estágio.

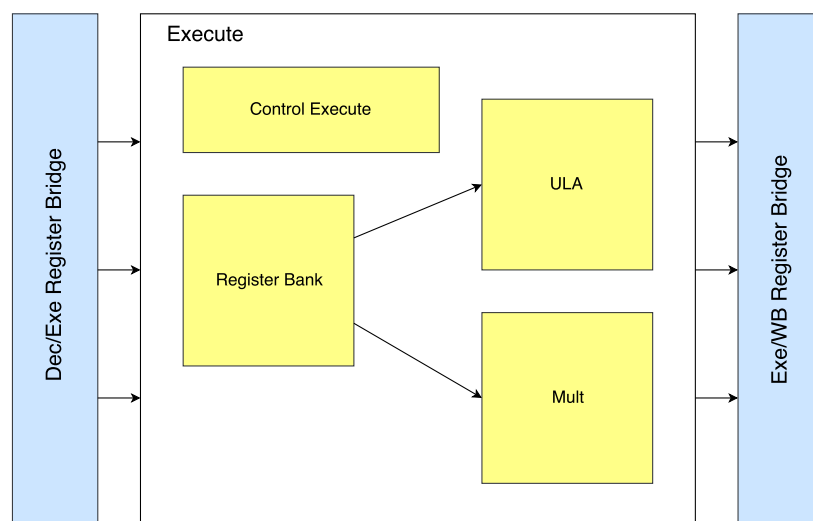


Figura 3. Diagrama simplificado do estágio *Execute* do *pipeline*.

O estágio também possui uma máquina de estados para comandar a execução do mesmo. No primeiro ciclo de *clock*, busca-se os registradores no *Banco de Registradores* ou no registrador de ponte, caso a instrução atual necessite de dados calculados pela instrução que está sendo executada, fazendo o encaminhamento. No segundo ciclo os valores recuperados dos registros são utilizados pela ULA ou pelo módulo de multiplicação. A decisão de qual módulo será executado vem de sinais de controle enviados através do registrador de ponte com o estágio de *Decode*.

3.3 Writeback

O processador finalizado conta com quatro módulos principais, além de uma série de módulos de controle secundários e multiplexadores. Os módulos principais são:

- *Decoder*: recebe a instrução de 16 bits e decodifica o *Opcode*, identificando os operandos e preparando os registradores que sinalizam se a instrução é uma multiplicação, se é um *Jump*, se haverá *Stall*, se haverá *Write Back*, qual registrador da multiplicação será armazenado se for o caso e se o segundo operando é um imediato.
- *Register Bank*: o banco de registradores conta com 16 registradores de 16 bits, duas portas de leituras para os operandos A e B e uma porta de escrita.

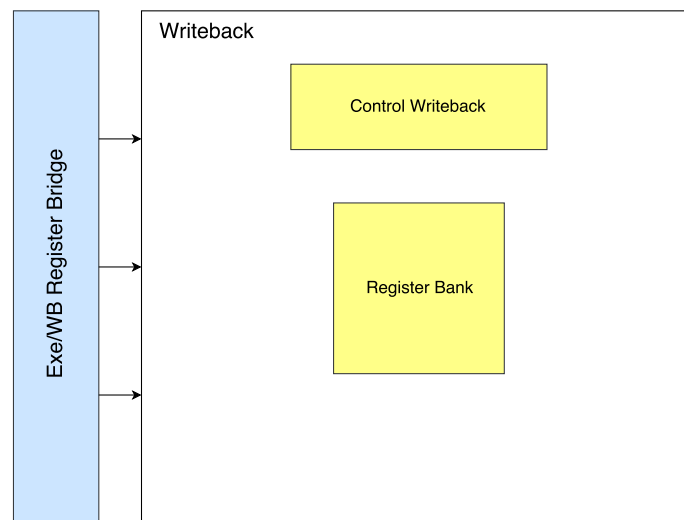


Figura 4. Diagrama simplificado do estágio *Writeback* do *pipeline*.

- *Unidade Lógica Aritmética*: a unidade lógica aritmética executa as instruções: ADD, SUB, SLT, AND, OR, XOR e BEZ.
- *Unidade multiplicadora*: a unidade multiplicadora recebe dois operandos de 16-bits e executa uma multiplicação produzindo um resultado de 32-bits que é armazenado em dois registradores: HI, que armazena os 16-bits mais significativos e LO, que armazena os 16-bits menos significativos. O resultado da operação armazenado nos registradores pode ser acessado por meio das instruções GHI e GLO, que escrevem o conteúdo dos registradores HI e LO, respectivamente, em um registrador do banco de registradores.

4 Integração

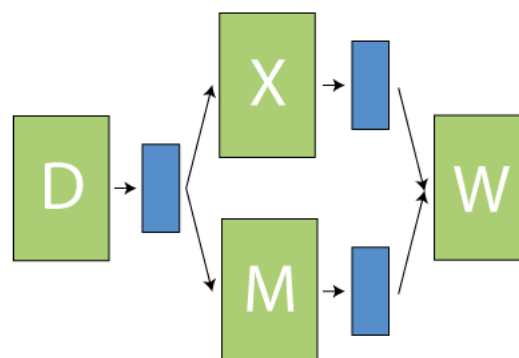


Figura 5. Diagrama da *pipeline*

A pipeline implementada possui três estágios: Decodificação, Execução e *Write Back*. O estágio de execução é dividido entre dois módulos independentes: o módulo de multiplicação e a Unidade Lógica Aritmética. O diagrama na figura 5 descreve a *pipeline*.

Cada um dos estágios da *pipeline* tem uma máquina de estados associada que executa as operações necessárias em múltiplos ciclos de *clock*. Ao fim de cada estágio, cada módulo define os valores de um conjunto de registradores que passa os dados para o próximo estágio por meio de um *buffer*. Na nossa implementação existem dois *buffers*: Decode/Execute e Execute/Write Back.

O estágio *Decoder* (D) recebe a próxima instrução da memória de instruções e define os valores dos registradores a seguir com base no *Opcode* da instrução, atualizando o *buffer* Decode/Execute:

- OpULA: o opcode da instrução para a ULA com 4-bits que indica a operação aritmética a ser realizada. Por exemplo: ADDI e ADD tem o mesmo OpULA.
- OpA, OpB, OpC: os operandos A, B e C.
- IsImm: indica se o segundo operando da instrução é um imediato.
- IsJump: indica se a instrução é um *Jump* (J).
- HasWB: indica se a instrução faz *Write Back*.
- HasStall: indica se a instrução gera *Stall*.
- IsMult: indica se a instrução vai utilizar a unidade multiplicadora.
- HiLo: indica qual registrador da multiplicação vai ser armazenado.
- StoreHiLo: indica que vai armazenar um registrador da unidade multiplicadora.
- AddrImm: endereço de memória de 12-bits para o *Jump*.

O estágio *Execução* (X e M) pode ocorrer em dois módulos separados: a ULA e a unidade multiplicadora. As entradas dos módulos são os registradores de saída da etapa anterior descritos acima. O resultado da ULA é armazenado no registrador Res e as flags da operação são armazenadas no registrador FlagReg. Já a unidade de multiplicação produz um resultado que é armazenado nos registradores Hi e Lo de 16-bits. O *buffer* Execute/Write Back conta com os seguintes registradores:

- Res: indica o resultado da operação na ULA ou os registradores Hi ou Lo da unidade multiplicadora.
- FlagReg: indica as flags da ULA.
- RegDest: indica o registrador de destino da operação de *Write Back*.
- HasWB: indica se o *Write Back* vai ser executado.
- AddrImm: indica o endereço do imediato da operação *Jump* (J).
- HasJumped: indica se um *Jump* foi executado.
- HasStall: indica se houve *Stall*.

Um multiplexador redireciona a saída da ULA e da unidade multiplexadora para o estágio de *WriteBack*. Perceba que a operação de multiplicação não passa efetivamente pelo estágio de *WriteBack* uma vez que o resultado da operação fica armazenado apenas nos registradores Hi e Lo, que podem ser escritos no banco de registradores por meio da execução de mais duas instruções: GHI e GLO.

5 Simulação e Testes

Os testes realizados procuraram medir a melhora de desempenho após a implementação da *pipeline*. Para isso, um programa de teste foi executado no processador antigo sem *pipeline* e no processador novo com a *pipeline* de três estágios. Perceba que a ordem dos operandos está invertida no *Assembly*. Isso ocorre porque no nosso processador o operando 2, que pode ser um imediato, é o operando referente aos bits 7-4 da instrução e não 3-0. Portanto, o código foi alterado para refletir a característica das instruções armazenadas na memória.

ADDI R1, 8, R0	R1 = 8
ADDI R2, 8, R0	R2 = 8
MUL -, R2, R1	$R1 * R2 = 64$
GLO R1, -, -	R1 = 64.
ADDI R2, 15, R0	R2 = 15.
ADDI R2, 1, R2	$R2 = 15 + 1$.
MUL -, R1, R2	$R1 * R2 = 1024$.
GLO R1, -, -	R1 = 1024.
ADDI R2, 10, R0	R2 = 10.
MUL -, R2, R2	$R2 * R2 = 100$.
GLO R2, -, -	R2 = 100.
ADDI R3, 0, R1	$R3 = R1 = 1024$.
ADDI R4, 0, R0	R4 = 0.
ADDI R6, 15, R0	R6 = 15.
ADDI R6, 8, R6	R6 = 23.
ADDI R4, 1, R4	$R4 = R4 + 1$
SUB R3, R2, R3	$R3 = R3 - R2$.
SLTI R5, 9, R4	$R5 = R4 > 9$
BEZ -, R6, R5	If (R5 == 0) jump to #R6

Tabela 2. Programa de teste

O programa de teste realiza a divisão de 1024 por 100, calculando o resto por meio de um loop e armazenando o resultado no registrador R3. O programa está descrito na tabela 2.

A figura XXX mostra o resultado da simulação no processador antigo e a figura YYY mostra o resultado da simulação no processador novo. Como percebemos pelos resultados, o processador novo executou o programa em Y *clocks* e o processador antigo executou o programa em X *clocks*. Portanto, a *pipeline* obteve uma melhora de 99% nesse caso específico.

Além deste, fizemos um programa para ordenar um vetor de 5 valores. O código do programa está descrito a seguir. Para melhor visualização e entendimento do código, ele foi dividido em blocos de 12 instruções. Os blocos são equivalentes, o que muda são os valores dos registradores.

O vetor está armazenado nos registradores 11 a 15. A cada bloco, o valor de um registrador é trocado com outro caso o seguinte seja menor do que ele. As trocas começam no registrador 11 (R11) e vão até o 14 (R14). No R11, primeiro verificamos se o valor de R12 é menor do que o dele, caso seja, troca os valores. Então o mesmo processo é feito com o R13, R14 e R15. Depois de finalizada as comparações do R11, olhamos para o R12 e fazemos o mesmo processo, *i.e.*, comparamos primeiro com o R13, depois com o R14 e por último com o R15. Este processo se repete até fazermos a última comparação de R14 e R15. Após isso, o vetor está ordenado.

Devido ao limitado número de instruções disponíveis, algumas adaptações precisaram ser feitas. A primeira é que não existe comparação de valores de registradores, logo para fazer isso, os valores a serem comparados são armazenados nos registradores R1 e R2. Para saber se devemos trocar,

fazemos a subtração desses valores e armazenamos em R3. A partir daí, devemos saber se o número é positivo ou negativo. Caso seja positivo, $R2 > R1$ e a troca não deve acontecer. Caso negativo, $R2 < R1$ e a troca deve ocorrer. Porém, a instrução de comparação SLTI não funciona com valores negativos, logo não podemos simplesmente comparar o valor de R3 com 0. Para isso, utilizamos o registrador R9 que em seu bit mais significativo possui o valor 1 e nos demais o valor 0. Com este registrador, fazemos um AND entre R9 e R3, dessa forma, sabemos o sinal de R3. Agora podemos comparar R3 com 0 e descobrir se o resultado é positivo ou negativo. Repare que caso $R3 = 1$, a subtração foi negativa, logo a troca deve ocorrer. A decisão de se devemos ou não trocar os valores é armazenada em R4 que é utilizado em um *branch*. Se $R4 = 0$, então $R3 = 0$ e a troca não deve ocorrer, logo o *branch* deve ser tomado.

Para realizar a troca, utilizamos o próprio R3 como registrador auxiliar. Se o *branch* precisar ser tomado, o programa pula para o valor armazenado em R8. Para encontrar o valor de R8, utilizamos 3 registradores: R5, R6 e R7. R5 armazena em qual bloco de 13 instruções estamos: no primeiro ele recebe o valor 1 e é incrementado em 1 a cada início de bloco. R6 armazena a multiplicação de R5 e R7, cujo valor é 13 (tamanho do bloco de instruções). Por fim, R8 pega os bits menos significativos da multiplicação e descobre para qual endereço deve pular caso necessário.

```

1 ADDI R5,1,R0
2 ADDI R7,13,R0
3 MULL R6,R5,R7
4 GLO R8,-,-
5 ADD R1,R11,R0
6 ADD R2,R12,R0
7 SUB R3,R1,R2
8 AND R3,R9,R3
9 SLTI R4,0,R3
10 BEZ -,R8,R4
11 ADD R3,R11,R0
12 ADD R11,R12,R0
13 ADD R12,R3,R0
14 # 2
15 ADDI R5,1,R5
16 ADDI R7,13,R0
17 MULL R6,R5,R7
18 GLO R8,-,-
19 ADD R1,R11,R0
20 ADD R2,R13,R0
21 SUB R3,R1,R2
22 AND R3,R9,R3
23 SLTI R4,0,R3
24 BEZ -,R8,R4
25 ADD R3,R11,R0
26 ADD R11,R13,R0
27 ADD R13,R3,R0
28 # 3
29 ADDI R5,1,R5
30 ADDI R7,13,R0
31 MULL R6,R5,R7
32 GLO R8,-,-
33 ADD R1,R11,R0
34 ADD R2,R14,R0
35 SUB R3,R1,R2
36 AND R3,R9,R3
37 SLTI R4,0,R3
38 BEZ -,R8,R4
39 ADD R3,R11,R0
40 ADD R11,R14,R0
41 ADD R14,R3,R0
42 # 4
43 ADDI R5,1,R5

```

```
44 ADDI R7,13,R0
45 MULL R6,R5,R7
46 GLO R8,-,-
47 ADD R1,R11,R0
48 ADD R2,R15,R0
49 SUB R3,R1,R2
50 AND R3,R9,R3
51 SLTI R4,0,R3
52 BEZ -,R8,R4
53 ADD R3,R11,R0
54 ADD R11,R15,R0
55 ADD R15,R3,R0
56 ##### 5
57 ADDI R5,1,R5
58 ADDI R7,13,R0
59 MULL R6,R5,R7
60 GLO R8,-,-
61 ADD R1,R12,R0
62 ADD R2,R13,R0
63 SUB R3,R1,R2
64 AND R3,R9,R3
65 SLTI R4,0,R3
66 BEZ -,R8,R4
67 ADD R3,R12,R0
68 ADD R12,R13,R0
69 ADD R13,R3,R0
70 # 6
71 ADDI R5,1,R5
72 ADDI R7,13,R0
73 MULL R6,R5,R7
74 GLO R8,-,-
75 ADD R1,R12,R0
76 ADD R2,R14,R0
77 SUB R3,R1,R2
78 AND R3,R9,R3
79 SLTI R4,0,R3
80 BEZ -,R8,R4
81 ADD R3,R12,R0
82 ADD R12,R14,R0
83 ADD R14,R3,R0
84 # 7
85 ADDI R5,1,R5
86 ADDI R7,13,R0
87 MULL R6,R5,R7
88 GLO R8,-,-
89 ADD R1,R12,R0
90 ADD R2,R15,R0
91 SUB R3,R1,R2
92 AND R3,R9,R3
93 SLTI R4,0,R3
94 BEZ -,R8,R4
95 ADD R3,R12,R0
96 ADD R12,R15,R0
97 ADD R15,R3,R0
98 ##### 8
99 ADDI R5,1,R5
100 ADDI R7,13,R0
101 MULL R6,R5,R7
102 GLO R8,-,-
103 ADD R1,R13,R0
104 ADD R2,R14,R0
105 SUB R3,R1,R2
106 AND R3,R9,R3
```

```

107 SLTI R4,0,R3
108 BEZ -,R8,R4
109 ADD R3,R13,R0
110 ADD R13,R14,R0
111 ADD R14,R3,R0
112 # 9
113 ADDI R5,1,R5
114 ADDI R7,13,R0
115 MULL R6,R5,R7
116 GLO R8,-,-
117 ADD R1,R13,R0
118 ADD R2,R15,R0
119 SUB R3,R1,R2
120 AND R3,R9,R3
121 SLTI R4,0,R3
122 BEZ -,R8,R4
123 ADD R3,R13,R0
124 ADD R13,R15,R0
125 ADD R15,R3,R0
126 ##### 10
127 ADDI R5,1,R5
128 ADDI R7,13,R0
129 MULL R6,R5,R7
130 GLO R8,-,-
131 ADD R1,R14,R0
132 ADD R2,R15,R0
133 SUB R3,R1,R2
134 AND R3,R9,R3
135 SLTI R4,0,R3
136 BEZ -,R8,R4
137 ADD R3,R14,R0
138 ADD R14,R15,R0
139 ADD R15,R3,R0

```

A Figura 6 mostra como o vetor está distribuído nos registradores:

- R11 = 4
- R12 = 3
- R13 = 2
- R14 = 5
- R15 = 1

Repare que R9 possui o bit mais significativo com valor 1 e os outros bits 0, como dito anteriormente. A Figura 7 mostra o fim da simulação. Note que o vetor está ordenado e que a simulação terminou com 28300 ps.

Para mostrar a melhoria ao introduzir o pipeline, executamos o mesmo programa na implementação que não possuía o pipeline. A Figura 8 mostra o resultado desta execução. Note que, sem o pipeline, a execução termina somente por volta de 60800 ps, significando que com o pipeline, neste programa, o ganho foi de aproximadamente 56% de tempo.

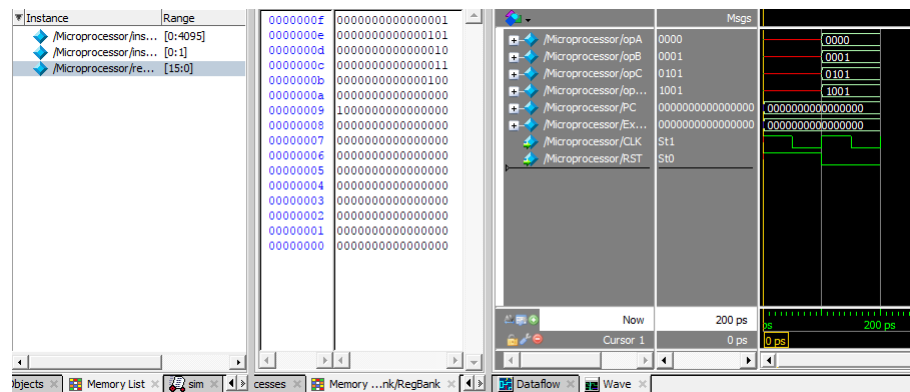


Figura 6. Início da ordenação

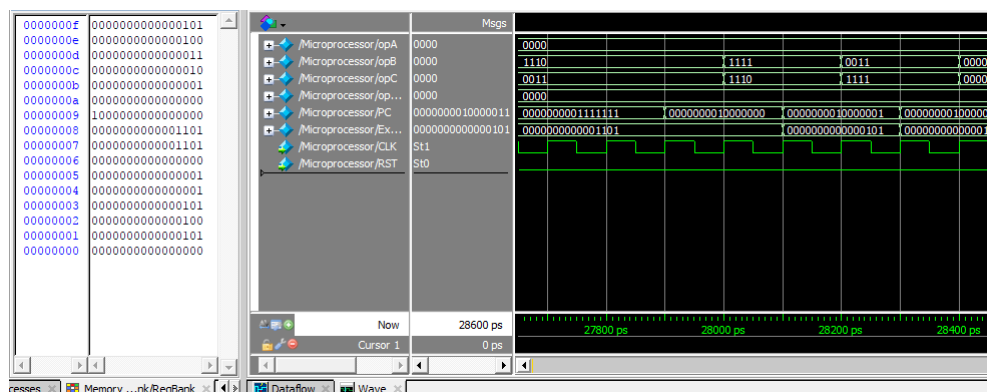


Figura 7. Fim da ordenação

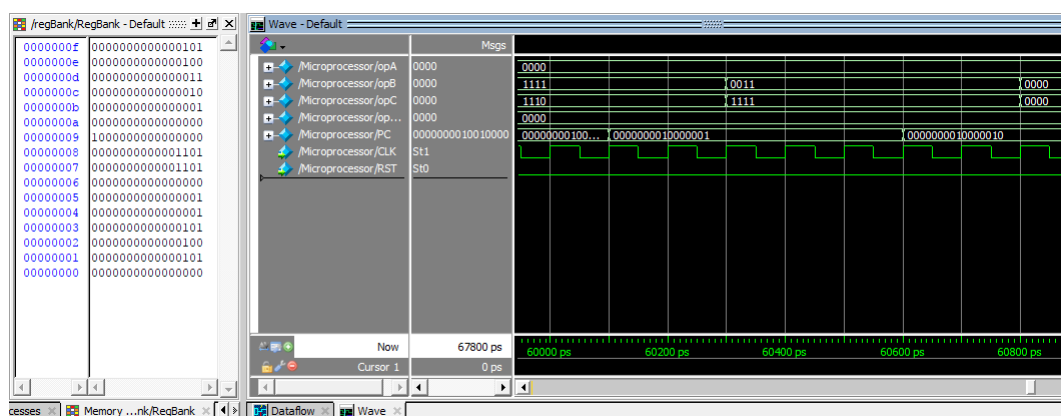


Figura 8. Fim da ordenação sem pipeline

6 Conclusão

A incorporação da *pipeline* de três estágios exigiu a introdução de *buffers* e unidades de controle adicionais no processador, complexificando consideravelmente o projeto. Os *branches* exigiram a introdução de um ciclo de *stall* para esperar pelo novo *Program Counter*, o que limitou os ganhos de velocidade de execução. A lógica de encaminhamento foi implementada para evitar *stalls* devido à

Hazards de dados. Ao final, o ganho de desempenho obtido foi de cerca de 99% no cenário de teste, o que mostra claramente a importância dos *pipelines* em processadores de alto desempenho.