PENINGKATAN PERFORMANSI PROSESOR DLX DENGAN METODE *PIPELINE*

Maman Abdurohman

Jurusan Teknik Informatika Sekolah Tinggi Teknologi Telkom, Bandung
mma@stttelkom.ac.id

Abstrak

Prosesor DLX adalah sebuah prosesor berbasis RISC (*Reduced Instruction Set Computer*) yang dirancang sebagai prosesor tujuan umum (*general purpose processor*). Prosesor ini mempunyai arsitektur *load-store* dengan panjang semua instruksinya 32 bit. Setiap instruksi dieksekusi dalam beberapa siklus waktu (*cycletime*). Secara umum time cycle yang digunakan sebanyak lima tahap yang terdiri dari tahap-tahap : *Instruction Fetch* (IF), *Instruction Decode* (ID), *Execute* (EX), *Memory Access* (MEM), dan *Write Back* (WB). Kelima tahap ini dikerjakan secara berurutan [2]. Sebagai prosesor *multicycle*, DLX mempunyai peluang untuk meningkatkan kinerjanya yang diukur dengan kecepatan proses yang dinyatkan sebagai waktu CPU (*CPU time*). Peningkatan kinerja prosesor DLX dapat diterapkan dengan menggunakan teknik *pipeline*. Pada jurnal ini telah dianalisis peningkatan performansi prosesor DLX dengan menggunakan teknik *pipeline*. Uji coba dilakukan terhadap beberapa program aplikasi yang dieksekusi dengan menggunakan teknik *pipeline* dan tanpa menggunakan teknik *pipeline*. Secara umum terjadi peningkatan kecepatan pada setiap kumpulan instruksi yang dianalisis. Proses pengujian dilakukan dengan menggunakan simulator windlx yang merupakan simulator prosesor DLX.

Kata kunci : Prosesor DLX, RISC, general purpose processor, CPU, Pipeline, windlx

Abstract

DLX processor is a RISC based processor that designed as general purpose processor. Its named load-store processor architecture with 32 bit all instruction long. Each instruction is executed in multicycle time. The stages, generally, used is constist of five process such as Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), dan Write Back (WB), respectively [2]. DLX has an opprotunity, due to its multycycles processes, to increase its performance, measured in CPU time, using *pipeline* technique. This paper tells about increasing performance analysis of DLX processor by implementing *pipeline* technique. Many instructions are tested with and without *pipeline* technique. These results show, in general, there is increasing rate of process after *pipeline* implementation. Windlx simulator is used to test many programs in DLX assembly.

Keywords: DLX processor, RISC, general purpose processor, CPU, Pipeline, windlx

1. Pendahuluan

Prosesor DLX merupakan prosesor dengan arsitektur RISC yang dirancang sebagai *general purpose processor*. DLX memiliki panjang instruksi 32 bit dengan tiga tipe instruksi, yaitu R-type, I-type dan J-type. Setiap instruksi dieksekusi melalui lima tahap. Kelima proses tersebut dilakukan dalam lima siklus waktu (*cycle time*), sehingga masing-masing proses saling bebas satu dengan lainnya. Ketika satu proses sedang diselesaikan, proses yang lainnya sedang menunggu. Demikian juga dengan instruksinya, dikerjakan satu per satu.

Penyelesaian suatu instruksi dalam lima siklus merupakan proses yang tidak efisien, karena pada saat yang bersamaan juga terdapat proses dan instruksi yang sedang menunggu. Menambahkan mekanisme untuk mengeksekusi instruksi secara bersamaan, dengan tidak ada tahapan yang berhenti, dipandang dapat meningkatkan kinerja prosesor tersebut. Peningkatan kinerja yang dimaksud adalah semakin sedikitnya kebutuhan waktu untuk menyelesaikan suatu program yang terdiri dari banyak instruksi.

Metode pengerjaan instruksi secara bersamaan pada proses yang berbeda disebut teknik *pipeline*. Teknik ini ditujukan untuk meningkatkan kinerja prosesor. Kinerja yang baik berbanding terbalik dengan waktu proses. Semakin kecil waktu proses, semakin baiklah kinerjanya.

Rancangan awal prosesor DLX tidak menggunakan metode *pipeline*, dimana setiap instruksi akan dikerjakan satu persatu yang masing-masing memerlukan 5 waktu siklus (*cycletime*). Untuk mengetahui seberapa efektif penerapan teknik *pipeline* terhadap peningkatan kinerja porsesor DLX, perlu dilakukan penelitian lebih lanjut tentang penerapan teknik ini.

Pada jurnal ini dipaparkan penerapan metode *pipeline* dan analisis teknik penerapannya pada prosesor DLX dalam meningkatkan kinerja prosesor tersebut. Analisis dilakukan dengan membandingkan kinerja prosesor DLX dengan *pipeline* terhadap tanpa *pipeline*.

Pengujian peningkatan performansi prosesor idealnya dilakukan dengan menerapkan langsung pada perangkat keras prosesor DLX. Karena saat ini belum ada representasi prosesor DLX yang sempurna, maka uji coba dilakukan pada simulator DLX yaitu windlx.

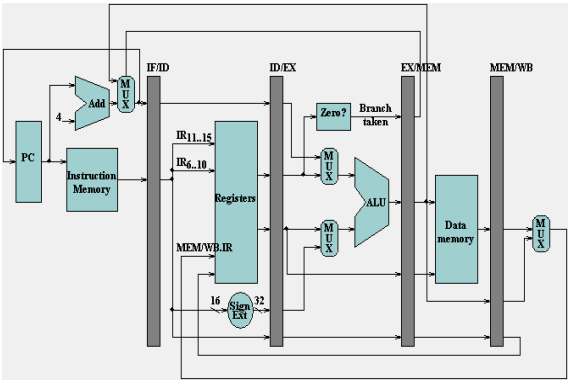
2. Prosesor DLX (Delux)

2.1 Jalur Data dan Kendali

Prosesor DLX menggunakan arsitektur ambil-simpan (*load-store*) dengan lima tahap *pipeline* untuk menyelesaikan instruksi. Kelima tahap tersebut adalah: Pengambilan Instruksi – *Instruction Fetch* (IF), Penerjemahan Kode – *Instruction Decode* (ID), Eksekusi – *Execute* (EX), Akses Memori – *Memory Access* (MEM), dan Menulis Balik – *Write Back* (WB). Panjang semua instruksi DLX adalah 32 bit. Dengan rancangan ini, maka proses *fetch* instruksi dari memori akan lebih mudah. Pengambilan instruksi ini dilakukan dengan membaca 4-*byte* data dalam memory.

Arsitektur prosesor DLX terdiri dari 5 komponen utama, sebagaimana diperlihatkan pada Gambar 1, yaitu terdiri dari:

a. PC (*Program Counter*) : register yang berfungsi menampung nilai yang digunakan untuk menunjuk alamat instruksi yang akan dieksekusi.
b. Memori Instruksi : Memori tempat menyimpan sejumlah instruksi yang akan dijalankan.
c. Register : Kumpulan Register 32 bit tempat menyimpan nilai sementara.
d. ALU (*Arithmatic and Logical Unit*) : tempat dilakukannya operasi aritmetik dan logic.
e. Memori Data : Memori untuk menyimpan data.



Gambar 1. Data Path Prosesor DLX

Prosesor DLX memiliki 32 buah register dengan masing-masing panjangnya 32 bit. Dua buah register mempunyai fungsi khusus, yaitu register 0 selalu bernilai nol. Register ini digunakan sebagai *operand* sumber jika memerlukan nilai nol. Register 31 disediakan oleh suatu instruksi DLX. DLX juga memiliki *program counter* dengan panjang 32 bit.

Salah satu ciri arsitektur RISC adalah memiliki panjang instruksi yang sama. DLX memiliki panjang instruksi 32 bit dengan 5 tahap *pipeline* yaitu : IF, ID, EX, MEM, dan WB.

Tahapan WB (*Write Back*) menyediakan saluran bagi DLX sehingga dapat menyimpan kembali register untuk tahap eksekusi (EX). Hal ini akan mempercepat pelaksanaan operasi register-ke-register oleh unit logic dan aritmetik yang berada dalam tahap eksekusi.

Operasi register-ke-register meningkatkan keseluruhan kecepatan saluran dimana tahapan tidak diperlambat oleh operasi *fetch/write* memori. Hal ini sangat penting dalam peningkatan kecepatan yang merupakan salah satu bagian penting arsitektur RISC. Arsitektur ini menuntut agar seluruh instruksi harus selesai dalam satu siklus yang sempurna.

Penjelasan rinci kelima tahap tersebut adalah sebagai berikut :

a. Siklus fetch instruksi (IF)
   `IR ← MEM[PC]`
   `PC ← PC + 4`
b. Siklus decode/Register Fetch (ID)
   `A ← Regs[IR 6..10]`
   `B ← Regs[IR 11..15]`
   `Imm ← ((IR16)16 ## IR16..31)`
c. Siklus eksekusi (EX)
   Mengacu memori :
   `ALUOutput ← A + Imm`
   Instruksi ALU register-register :
   `ALUOutput ← A op B`
   Instruksi ALU register-immediate :
   `ALUOutput ← A op Imm`
   Pencabangan :
   `ALUOutput ← NPC + Imm`
   `Cond ← (A op 0)`
d. Siklus akses memori/pencabangan (MEM)
   Mengacu memori :
   `LMD ← Mem [ALUOutput]`
   Pencabangan :
   `if (cond) ← ALUOutput`
   `else PC ← NPC`
e. Siklus tulis-balik (WB)
   Instruksi ALU register-register :
   `Regs[IR 6..20] ← ALUOutput`
   Instruksi ALU register-immediate :
   `Regs[IR 11..15]← ALUOutput`
   Instruksi Load :
   `Regs[IR 11..15]← LMD`

2.2 Implementasi Dasar DLX

Setiap instruksi DLX dapat diimplementasikan paling banyak dalam 5 siklus clock. Kelima siklus clock tersebut adalah sebagai berikut :

a. Siklus pengambilan (*fetch*) instruksi (IF):
   `IR ← Mem[PC]`
   `NPC ← PC + 4`
   Operasi: mengeluarkan nilai PC dan mengambil isi memori pada alamat PC di simpan ke dalam register instruksi (IR), nilai PC ditambah 4 untuk menunjuk pada instruksi selanjutnya. IR digunakan untuk menyimpan instruksi yang diperlukan pada siklus clock selanjutnya; begitu juga register NPC digunakan untuk menyimpan nilai PC berikutnya.

b. Siklus decode instruksi (ID), pengambilan isi register:

```
A ← Regs [IR6 .. 10];
B ← Regs [IR11..15];
Imm ← ((IR16)16##IR16..31)
```

Operasi: decode instruksi dan mengakses file register untuk membaca register. Keluaran dari register-register tujuan-umum disimpan dalam register sementara (A dan B) untuk digunakan pada siklus selanjutnya. 16 bit bawah pada IR juga adalah *sign-extended* dan disimpan ke dalam register sementara Imm, untuk digunakan pada siklus berikutnya.

Proses decode dilakukan bersamaan dengan pembacaan register. Hal ini dimungkinkan, karena *field*-nya berada pada lokasi yang tetap dalam format instruksi DLX. Teknik ini dinamakan decode *field*-tetap (*fixed-field decoding*). Dengan catatan, kita mungkin membaca register yang tidak digunakan, yang tidak membantu dan tidak merugikan juga. Karena porsi *immediate* setiap instruksi DLX berada pada tempat yang sama, *sign-extented immediate* juga dihitung selama siklus ini yang mungkin diperlukan pada siklus berikutnya.

c. Siklus eksekusi (EX), alamat efektif:
ALU melaksanakan salah satu jenis fungsi dari 4 fungsi operasi dengan operand berasal dari siklus sebelumnya. Operasi yang dilaksanakan oleh ALU ditentukan oleh tipe intstruksinya :

- Mengacu pada memori

```
ALUOutput ← A + Imm;
```

Operasi: ALU menjumlahkan operand-operandnya untuk mendapatkan alamat efektif dan menempatkan hasilnya ke dalam register ALUOutput.

- Instruksi register-register

```
ALUOutput ← A op B;
```

Operasi: ALU melaksanakan operasi yang ditentukan oleh opcode terhadap nilai register A dan nilai register B. Hasilnya disimpan sementara di register ALUOutput.

- Instruksi register-immediate

```
ALUOutput ← A op Imm;
```

Operasi: ALU melaksanakan operasi yang ditentukan oleh opcode terhadap nilai register A dan nilai pada register Imm. Hasilnya disimpan sementara di register ALUOutput.

- Pencabangan

```
ALUOutput ← NPC + Imm;
Cond ← (A op 0)
```

Operasi: ALU menjumlahkan NPC dengan nilai sign-extended immediate dalam register Imm untuk menghasilkan alamat tujuan pencabangan. Register A, yang telah dibaca pada siklus sebelumnya, diperiksa untuk menentukan apakah terjadi pencabangan. Operasi perbandingan op adalah operator relasi yang ditentukan oleh opcode pencabangan, contoh op sama dengan "==" untuk instruksi BEQZ.

DLX disebut arsitektur load-store artinya bahwa siklus alamat efektif dan eksekusi dapat dikombinasikan ke dalam satu siklus clock, karena tidak ada instruksi yang secara bersamaan menghitung alamat data, menghitung alamat tujuan instruksi, dan melaksanakan operasi pada data.

d. Siklus akses memori (MEM), penyelesaian pencabangan:
Instruksi DLX yang aktif pada siklus ini adalah instruksi-instruksi *load*, *store* dan *branche*.

- Mengacu pada memori :

```
LMD←Mem[ALUOutput] atau
Mem[ALUOutput] ← B
```

Operasi: akses memori jika diperlukan. Jika instruksi di load, data dari memori ditempatkan pada register LMD (*load memory data*), jika instruksinya store maka data dari register B akan disimpan dalam memori.

- Pencabangan:

```
If (cond) PC ← ALUOutput
else PC ← NPC
```

Operasi: untuk jenis instruksi pencabangan, maka nilai PC ditimpa dengan alamat tujuan pencabangan yang ada pada register ALUOutput, jika tidak maka nilai PC digantikan nilai register NPC.

e. Siklus tulis-balik (WB)

- Instruksi Register-Register

```
Regs[IR16..20] ← ALUOutput;
```

- Instruksi Register-Immediate

```
Regs[IR11..15] ← ALUOutput;
```

- Instruksi Load

```
Regs[IR11..15] ← LMD;
```

Operasi: hasil akhir dari instruksi dituliskan ke dalam file register, baik yang berasal dari sistem memori (pada LMD) atau dari ALU (pada ALUOutput). Register tujuan juga berada pada salah satu posisi field tergantung opcode.

Pada akhir setiap siklus clock, setiap nilai yang dihitung selama siklus clock tersebut dan diperlukan pada siklus clock berikutnya (baik untuk instruksi sekarang atau berikutnya) ditulis ke dalam perangkat penyimpan, yang mungkin berupa memori, register tujuan-umum, PC atau register sementara (seperti LMD, Imm, A, B, IR, NPC, ALUOutput atau Cond). Register sementara menyimpan nilai atnara siklus clock untuk satu instruksi, sementara elemen penyimpan lainnya adalah menyimpan nilai untuk instruksi berikutnya.

Pada implementasi ini, instruksi pencabangan memerlukan empat siklus dan semua instruksi lain memerlukan lima siklus.

2.3 Format Instruksi

Dalam DLX terdapat tiga buah format instruksi yaitu: tipe-R, tipe-I dan tipe-J. Semua format instruksi dibedakan oleh kode operasinya (operation

code - opcode). Informasi lain pada setiap format juga berbeda-beda. Instruksi tipe-R (register) membutuhkan dua register sumber dan satu register tujuan. Instruksi tipe-I (immediate) membutuhkan satu buah register sumber, satu buah register tujuan dan nilai 16-bit. Instruksi tipe-J (jump) terdiri dari opcode dan 26 bit operand. Format instruksi DLX dijelaskan pada Tabel 1. Set instruksi pada Prosesor DLX dapat dilihat dalam daftar pada Lampiran.

Tabel 1. Format Instruksi DLX

Format	Bit					
	31 26	25 21	20 16	15 11	10 6	5 0
Tipe-R	0x0	Rs1	Rs2	Rt	no	Op
Tipe-I	Op	Rs1	Rt	*immediate*		
Tipe-J	Op	*Nilai*				

3. Penerapan Teknik *Pipeline* Pada Prosesor DLX dan Kendalanya

3.1 Definisi

Pipelining adalah teknik implementasi dengan beberapa instruksi dieksekusi dalam waktu yang bersamaan. Saat ini, *pipelining* merupakan teknik implementasi kunci yang digunakan untuk meningkatkan kecepatan CPU. *Pipeline* mirip dengan jalur perakitan. Pada jalur perakitan mobil, terdapat beberapa tahap, masing-masing tahap memberikan kontribusi bagi konstruksi mobil. Setiap tahap beroperasi secara paralel dengan tahap lain untuk mobil yang berbeda. Dalam *pipeline* komputer, setiap tahap dalam *pipeline* menyelesaikan satu bagian instruksi. Masing-masing tahap ini dinamakan tahap *pipe* atau segmen *pipe*. Tahapan-tahapan terhubung satu dengan berikutnya untuk melakukan satu rangkaian pekerjaan.

Throughput didefinisikan sebagai jumlah instruksi yang keluar dari *pipeline* per satuan waktu. Karena *pipe stage* berkaitan satu sama lain, seluruh tahapan harus siap untuk memproses pada waktu bersamaan. Waktu yang diperlukan antara perpindahan instruksi satu tahap *pipeline* disebut siklus mesin. Karena semua tahap bekerja simultan, maka panjang siklus mesin ditentukan oleh waktu yang diperlukan oleh tahap *pipe* yang paling lambat. Dalam komputer, siklus mesin ini biasanya satu siklus clock (kadang-kadang dua atau lebih), walaupun clock mungkin mempunyai banyak fase.

Tujuan perancang *pipeline* adalah agar panjang setiap tahap *pipeline* seimbang. Jika tahap-tahap tersebut seimbang secara sempurna, maka waktu per instruksi pada mesin dengan *pipeline* – asumsi kondisi ideal – sama dengan: pembagian Waktu per instruksi pada mesin tanpa *pipeline* oleh Jumlah tahap *pipe*. Dengan kondisi ini, peningkatan kecepatan sama dengan jumlah tahap pipe. Biasanya, bagaimanapun, tahapan-tahapan tidak akan seimbang secara sempurna; lebih lanjutnya *pipeline* akan melibatkan *overhead*.

Pipelining menghasilkan pengurangan rata-rata waktu eksekusi per instruksi. Pengurangan ini akan terlihat sebagai penurunan jumlah siklus clock per instruksi (CPI), sebagai penurunan waktu siklus clock atau kombinasinya. Jika titik awalnya mesin yang mengambil banyak siklus clock per instruksi, maka pipelining biasanya dilihat sebagai pengurangan CPI. Jika titik awalnya adalah mesin yang mengambil satu siklus clock per instruksi, maka pipelining menurunkan waktu siklus clock.

3.2 *Pipeline* pada prosesor DLX

Pipeline dapat dibuat pada jalur data DLX dengan sedikit perubahan dengan memulai instruksi baru pada setiap siklus clock. Setiap siklus clock dari bagian sebelumnya menjadi *pipe stage* salah satu siklus dalam *pipeline*. Pola eksekusinya diperlihatkan seperti pada Tabel 2.

Tabel 2. Proses *Pipeline*

Instruksi	1	2	3	4	5	6	7
Ins i	IF	ID	EX	MEM	WB	WB	WB
Ins i+1		IF	ID	EX	MEM	MEM	
Ins i+2			IF	ID	EX		

Setiap instruksi mengambil lima siklus clock sampai selesai. Selama setiap siklus clock, perangkat keras menginisiasi satu instruksi baru dan mengeksekusi bagian dari lima instruksi berbeda.

Pada kenyataannya *pipeline* tidak sesederhana kondisi di atas. Ada beberapa perintah yang tidak bisa dijalankan secara paralel karena ada sumber daya yang salin tergantung misalnya sehingga satu instruksi tidak dapat berjalan sebelum instruksi hasil instruksi sebelumnya diperoleh. Ini salah satu masalah yang harus diselesaikan agar tujuan dari *pipeline* tercapai.

Untuk mengetahui lebih rinci permasalahan-nya, harus diketahui apa yang terjadi pada setiap siklus clock mesin dan harus dipastikan bahwa tidak dilakukan dua operasi yang berbeda dengan sumber datapath yang sama pada siklus clock yang sama. Contoh, sebuah ALU tidak dapat diminta untuk menghitung alamat efektif dan melakukan operasi pengurangan secara bersamaan. Jadi, harus dipastikan tumpang tindih instruksi dalam *pipeline* tidak menyebabkan terjadinya konflik seperti itu.

3.3 Hambatan Pipeline

Terdapat situasi, disebut hazard (resiko), yang mencegah instruksi berikutnya dalam rangkaian instruksi eksekusi selama siklus clock yang ditandainya. Ada tiga kelas hazard yang menurunkan kinerja kecepatan ideal *pipelining*, yaitu:

a. ***Hazard* struktur** akibat dari konflik sumber daya pada saat perangkat keras tidak dapat mendukung semua kemungkinan kombinasi instruksi dalam eksekusi yang simultan.

b. ***Hazard* data** yang timbul pada saat instruksi tergantung pada hasil instruksi sebelumnya.

c. ***Hazard* kendali** akibat *pipelining* pencabangan dan instruksi lain yang mengubah PC.

Hazard dalam *pipeline* mengakibatkan *pipeline* harus melakukan stall. Dalam mengatasi hazard serting kali memerlukan beberapa instruksi dalam *pipeline* diizinkan untuk diproses sementara instruksi lain menunggu (*delay*). Pada saat sebuah instruksi di-*stall*, semua instruksi yang keluar sesudah instruksi tersebut di stall juga. Instruksi yang muncul sebelum instruksi yang distall harus terus *dipipeline*. Hasilnya, tidak instruksi baru yang difetch selama *stall*.

4. Kinerja Prosesor DLX dengan *Pipeline*

4.1 Skenario Pengujian

Pengujian dilakukan dengan menggunakan beberapa aplikasi yaitu :

- gccdlx: cross compiler DLX pada gcc. Fungsinya meng-compile sumber dalam bahasa *.c atau *.cpp ke dalam bahasa assembly DLX.
- dlxcc: cross compiler DLX untuk file c atau cpp.
- dlxsim: simulator DLX yang meng-assembly bahasa tingkat rendah dalam bahasa assembly DLX ke dalam bahasa mesin untuk dijalankan.
- windlx: simulator DLX dalam window yang berfungsi untuk menjalankan aplikasi dalam bahasa assembly DLX kemudian ditampilkan berbagai hasil pengerjaan file tersebut [1].

Pengujian dimulai dengan melihat program dalam bahasa c atau cpp kemudian dicompile menjadi bahasa assembly DLX dengan menggunakan gccdlx atau dlxcc [3]. Output dari gccdlx atau dlxcc dijalankan dalam windlx[4], sehingga jumlah instruksi yang dieksekusi dapat diketahui dan berapa banyak cycle time yang diperlukan setelah diterapkannya *pipeline*.

Pengujian di atas kemudian dibandingkan dengan menjalankan instruksi tanpa *pipeline*, setiap instruksi dijalankan dalam 5 siklus clock.

4.2 Metode Pengujian

Pengujian dilakukan pada beberapa program untuk mengukur peningkatan kecepatan (*speedup*) setelah menggunakan *pipeline*. Program uji terdiri lima buah jenis aplikasi berbeda yang terdiri dari: operasi array, data hazard, bilangan prima, penjumlahan nilai skalar dan memori. Masing-masing aplikasi akan dijalankan dengan menggunakan metoda *pipeline* untuk diukur total siklus yang diperlukan sampai program selesai.

Hasil pengujian ini akan dibandingkan dengan banyaknya siklus yang diperlukan tanpa *pipeline*. Dengan membandingkan keduanya akan diperoleh peningkatan kecepatan (*Speedup*). Semakin tinggi nilai *speedup* berarti *pipeline* semakin efektif pada jenis program tersebut.

4.3 Program Aplikasi Sebagai Materi Uji

- **Aplikasi Pertama – Operasi *Array***

```
#define MAXVSIZE 1000
#include <stdio.h>
void loop(double c[], double a[], double
b[], int n)
{
   int i;
   for (i=0;i<n;i++)
     c[i] = a[i] + b[i];
}
void main() {
double a[MAXVSIZE], b[MAXVSIZE],
c[MAXVSIZE];
     int i,n;
     printf("Masukan ukuran vektor:  ");
     scanf("%d",&n);
     for (i=0;i<n;i++) {
       a[i] = i;
       b[i] = n-i;
     }
loop(c,a,b,n);
    for (i=0;i<n;i++)
      printf("c[%d] = %g\n",i,c[i]);
}
```

Data setelah dcompile oleh dlxcc

```
.data
Size: .word    5
A:.double      1.0,2.0,3.0,4.0,5.0
B:.double      5.0,4.0,3.0,2.0,1.0
C:.double      0.0,0.0,0.0,0.0,0.0
.text
.global main
main:
lw      r2,Size(r0)
addi    r1,r0,0         ;r1 = i = 0
Loop:   seq             r3,r1,r2
bnez    r3,Exit
sll     r4,r1,3         ; r4 = 8*i
ld      f2,A(r4)        ; load A[i]
ld      f4,B(r4)        ; load B[i]
addd    f0,f2,f4
sd      C(r4),f0        ; store C[i]
addi    r1,r1,1         ; i++
j       Loop
Exit:
Finish:         ;*** end
trap            0
```

Aplikasi ini memerlukan 75 cycletime dengan jumlah instruksi yang dieksekusi 50 instruksi.

- **Aplikasi Kedua – Data Hazard**

```
.data
temp:   .word 1,2,3,4,5
.text
.global main
main:
        addi R1, R0, #1
        addi R2, R0, #2
        addi R3, R0, #3
        addi R4, R0, #4
        addi R5, R0, #5
        addi R6, R0, #6
        addi R7, R0, #7
        addi R8, R0, #8
        addi R9, R0, #9
        addi R10, R0, #10
        addi R11, R0, #11
        add  R1, R2, R3
        sub  R4, R1, R5
        and  R6, R1, R7
        or   R8, R1, R9
        xor  R10, R1, R11
```

```
        ; RAW data hazard
        addi R2, R0, temp
        addi R3, R0, #4
        add  R1, R2, R3
        lw   R4, 0(R1)
        sw   8(R2), R4
        ; RAW data hazard
        lw   R1, 0(R2)
        sub  R4, R1, R5
        and  R6, R1, R7
        or   R8, R1, R9
        trap 0
```

Aplikasi ini memerlukan 32 cycletime dengan jumlah instruksi yang dieksekusi 26 instruksi.

- **Aplikasi Ketiga – Tabel Bilangan Prima**

```
.data
.global Count
Count:
.word          10
.global Table
Table:
.space  Count*4
.text
.global main
main:
        addi    r1,r0,0
        addi    r2,r0,2
NextValue:
        addi    r3,r0,0
Loop:   seq     r4,r1,r3
        bnez    r4,IsPrim
        lw      r5,Table(R3)
        divu    r6,r2,r5
        multu   r7,r6,r5
        subu    r8,r2,r7
        beqz    r8,IsNoPrim
        addi    r3,r3,4
        j       Loop
IsPrim:
        sw      Table(r1),r2
        addi    r1,r1,4
        lw      r9,Count
        srli    r10,r1,2
        sge     r11,r10,r9
        bnez    r11,Finish
IsNoPrim:
        addi    r2,r2,1
        j       NextValue
Finish:
        trap           0
```

Aplikasi ini memerlukan 2605 cycletime dengan jumlah instruksi yang dieksekusi 750 instruksi.

- **Aplikasi Keempat – Penjumlahan Nilai Skalar**

```
Loop :  LD F0,0(R1)
        ADDD F4, F0, F2
        SD 0(R1), F4
        SUBI R1, R1, 8
        BNEZ R1, Loop
```

Analisis dengan memperhatikan Hazard dan melakukan stall pipeline :

```
Loop :  LD F0,0(R1)
        Stall
        ADDD F4, F0, F2
        Stall
        Stall
        SD 0(R1), F4
        SUBI R1, R1, 8
        BNEZ R1, Loop
        Stall
```

Waktu yang diperlukan adalah 9 clock cycle.

Penjadwalan ulang program di atas menyebabkan perbaikan sebagai berikut:

```
Loop :  LD F0, 0(R1)
        Stall
        ADDD    F4, F0, F2
        SUBI    R1, R1, #8
        BNEZ    R1, LOOP
        SD 8(R1), F4
```

diperlukan 6 cycletime, jumlah instruksi 5.

- **Aplikasi Kelima – Memori**

```
Loop :  LD F0, 0(R1)
        ADDD F4, F0, F2
        SD 0(R1), F4
        LD F6, -8(R1)
        ADDD F8, F6, F2
        SD -8(R1), F8
        LD F10, -16(R1)
        ADDD F12, F10, F2
        SD -16 (R1), F12
        LD F14, -24(R1)
        ADDD F16, F14, F2
        SD -24(R1), F16
        SUBI R1, R1, #32
        BNEZ R1, LOOP
```

Analisis dengan memperhatikan Hazard dan melakukan stall pipeline :

```
Loop :  LD F0, 0(R1)
        Stall
        ADDD F4, F0, F2
        Stall
        Stall
        SD 0(R1), F4
        LD F6, -8(R1)
        Stall
        ADDD F8, F6, F2
        Stall
        Stall
        SD -8(R1), F8
        LD F10, -16(R1)
        Stall
        ADDD F12, F10, F2
        Stall
        Stall
        SD -16 (R1), F12
        LD F14, -24(R1)
        Stall
        ADDD F16, F14, F2
        Stall
        Stall
        SD -24(R1), F16
        SUBI R1, R1, #32
        BNEZ R1, LOOP
        Stall
```

Waktu yang diperlukan adalah 27 clock cycle.
Setelah dilakukan penjadwalan:

```
Loop :  LD F0, 0(R1)
        LD F6, -8(R1)
        LD F10, -16(R1)
        LD F14, -24 (R1)
        ADDD F4, F0, F2
        ADDD F8, F6, F2
        ADDD F12, F10, F2
        ADDD F16, F14, F2
        SD 0(R1), F4
        SD -8(R1), F8
        SD -16(R1), F12
        SUBI R1, R1, #32
        BNEZ R1, LOOP
        SD 8(R1), F16
```

Waktu yang diperlukan adalah 14 clock cycle, jumlah instruksi 14.

4.4 Hasil Simulasi

Hasil pengujian dari simulasi di atas ditunjukkan pada Tabel 3 di bawah ini.

Tabel 3. Hasil Pengujian

Program	Σ Cycle Non *Pipeline*	ΣCycle *Pipeline*	*Speedup*
Aplikasi 1	250	75	3,33
Aplikasi 2	130	32	4,06
Aplikasi 3	3750	2605	1,44
Aplikasi 4	25	6	4,11
Aplikasi 5	70	14	5
Rata-rata peningkatan kecepatan			3,59

4.5 Analisis Kinerja Prosesor DLX dengan Penerapan Pipeline.

Dari Tabel 3 di atas dapat diamati bahwa seluruh aplikasi yang dijalankan dengan menggunakan simulator DLX *Pipeline* menunjukan peningkatan kinerja 1 – 5 kali, dengan rata-rata peningkatan secara keseluruhan 3,59 kali. Ini membuktikan peningkatan kinerja yang signifikan dari penerapan metode *Pipeline* pada prosesor DLX.

Asumsi dasar menyatakan bahwa peningkatan kinerja menggunakan metode *Pipeline* dapat mencapai maksimal = jumlah tahapan yaitu 5. Dalam ujicoba ini terlihat bahwa yang sesuai dengan asumsi tersebut adalah Aplikasi 5 yang di dalamnya tidak terdapat *stall*. Sedangkan *pipeline stall* terjadi pada Aplikasi 1 sampai Aplikasi 4, yaitu siklus clock tambahan karena terdapat dua tahap yang memerlukan *resource* yang sama. Semakin banyak *stall*, peningkatan kecepatan prosesor DLX semakin turun. Pengujian pada Aplikasi 3, dengan stall terbanyak, menghasilkan peningkatan performansi sistem paling rendah, yaitu hanya 1,44 kali dari kondisi tanpa *pipeline*.

5. Kesimpulan dan Saran

5.1 Kesimpulan

1. Penerapan metode *Pipeline* pada prosesor DLX telah meningkatkan performansi sistem sampai 3,59 kali dari kondisi tanpa *pipeline*.
2. Peningkatan kinerja yang ideal adalah 5 sesuai dengan jumlah tahapan. Tidak tercapainya kondisi tersebut disebabkan oleh *pipeline stall*. Semakin banyak *stall* seperti, peningkatan kecepatan prosesor DLX semakin turun.

5.2 Riset Selanjutnya

Untuk mencapai kondisi ideal perlu diimplementasikan mekanisme penanganan *Pipeline stall* baik yang disebabkan oleh *data hazard*, *struktur hazard* atau *control hazard*.

Daftar Pustaka

[1] Adams, George, *DLX View: A Window Simulator of DLX*, Purdue University's School of Electrical and Computer Engineering.

[2] John L, Hennessy & David A. Patterson, *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publisher, Ind. San Mateo. USA.

[3] Khosravipour, Maizar, *Windows – DLX Simulator*, Univ. of Technology Vienna. 1992.

[4] Larry, B. Hostetler & Brian Mirtich, *DLXsim – a Simulator for DLX*. http://hardy.ocs.mq.edu.au/mpce_course/dlxsim/report.html

Lampiran

Set Instruksi pada Prosesor DLX

```
1.  ADD : add (R-0x20), Rt = Rs1 +  Rs2

2.  ADDI : add immediate (I-0x80), Rt = Rs1
    + extend (Immediate)

3.  AND : and (R-0x24), Rt = Rs1 & Rs2

4.  ANDI : and immediate, (I-0x0c), Rt =
    Rs1 & immediate.

5.  BEQZ : branch if equal to zero, (I-
    0x04), PC+=(Rs1==0?extend(immediate):0)

6.  BNEZ : branch if not equal to zero,

7.  (I-0x05), PC+ = (Rs1 !=0? Extend
    (immediate) :0)

8.  J : jump, (J-0x02), PC+ = extend
    (nilai)

9.  JAL : jump and link, (J-0x03),
    R31=PC+4; PC=extend(nilai)

10. JALR : jump and link register, (I-
    0x13), R31=PC+4, PC = Rs1

11. JR : jump register, (I-0x12), PC = Rs1

12. LHI : load high bits, (I-0x0f), Rt
    =immediate << 16

13. LW : load word, (I-0x23), Rt =Mem[Rs1 +
    extend(immediate)]

14. OR : or, (R-0x25), Rt = Rs1 | Rs2

15. ORI : or immediate, (I-0x0d), Rt = Rs1
    | immediate

16. SEQ : set if equal, (R-0x28), Rt = (Rs1
    == Rs2 ? 1 : 0)

17. SEQI : set if equal to immediate, (I-
    0x18), Rt = (Rs1==extend(immediate)?
    1:0)

18. SLE : shift left if less than or equal,
    (R-0x2c), Rt = (Rs1<=Rs2 ? 1:0)
```

19. SLEI : shift left if less than immediate, (I-0x1c), Rt=(Rs1<=extend(immediate)?1:0)

20. SLL : shift left logical, (R-0x04), Rt=Rs1 << (Rs2 % 8)

21. SLLI : shift left logical immediate, (I-0x14), Rt=Rs1 << (immediate%8)

22. SLT : set if less than, (R-0x2a), Rt = (Rs1<Rs2 ? 1: 0)

23. SLTI : set if less than immediate, (I-0x1a), Rt = (Rs1<extend(immediate) ? 1: 0)

24. SNE : set if not equal, (R-0x29), Rt = (Rs1!= Rs2 ? 1 : 0)

25. SNEI : set if not equal to immediate, (I-0x19),Rt=(Rs1!= extend(immediate)?1: 0)

26. SRA : shift right arithmetic, as SRL & see below

27. SRAI : shift right arithmetic immediate, as SRLI and see below

28. SRL : shift right logical, (R-0x06), Rt = Rs1 >> (Rs2 % 8)

29. SRLI : shift right logical immediate, (I-0x16), Rt = Rs1 >> (immediate % 8)

30. SUB : substract, (R-0x22), Rt = Rs1 – Rs2

31. SUBI : substract immediate, (I-0x0a), Rt = Rs1 – extend (immediate)

32. SW : store word, (I-0x2b), Mem [Rs1 + extend(immediate)] = Rd

33. XOR : exclusive or, (R-0x26), Rt = Rs1^Rs2

34. XORI : exclusive or immediate, (I-0x0e), Rt = Rs1 ^ immediate.

Catatan:

- JAL dan JALR menggunakan Rs1 saja, nilai immediate dan Rt dibuang.

- SHI dipakai untuk mengambil bit-bit sebelah atas dari konstanta 32-bit.

- SRA dan SRAI adalah penggeseran kanan aritmetik. Bit tanda operand akan digandakan, bukan menggantinya dengan nol. SRL dan SRA akan sama jika Rs1 bernilai positif. Jika Rs1 negatif (bit 31 == 1), maka 1 disisipkan dari kiri untuk SRA dan SRAI.

- SW menggunakan Rt untuk register sumber (nilai yang disimpan di memori berasal dari Rt).