BAB III

ANALISIS DAN PERANCANGAN

3.1 Analisis Sistem

Analisis adalah tahap aktifitas kreatif dimana analis berusaha memahami permasalahan secara mendalam terhadap metode yang diterapkan. Ini adalah proses *interative* yang terus berjalan hingga permasalahan dapat dipahami dengan benar. Analisis bertujuan untuk mendapatkan pemahaman metode yang diangkat secara keseluruhan tentang sistem yang akan dibuat berdasarkan masukan dari pihak-pihak dan juga pengalaman analis yang berkepentingan dengan sistem tersebut.

3.2 Analisis Permasalahan

Permasalahan yang dibuat dalam tugas akhir ini adalah membuat simulasi algoritma *Lemple Ziv Storer Symanski* (LZSS) dan algortima *Rice Coding* sebagai pembanding untuk mengompresi data, kemudian mengimplementasikannya pada proses pengiriman data. Hasil simulasi algoritma LZSS dan *Rice Coding* yang telah dibuat ini kemudian dilakukan uji coba, sehingga kita bisa melihat hasil kompresi dan simulasi yang telah dibuat. Data dari hasil penelitian dengan algoritma LZSS dan *Rice Coding* ini akan dibandingkan.

3.3 Analisis Tehadap Sebuah *File* Teks

Analisis dilakukan dengan menggunakan *File* teks dengan isi ‘AAAAAAAAAAABCABCAAAAA’. Proses kompresi mula-mula dilakukan dengan metode LZSS dimana pada metode ini isi *File* debaca sebagai sebuah *string* yang ditampung dalam variabel baru yaitu *Siliding Windows* (SW) dan *Read Ahead* (RA). SW merupakan *array* yang dapat menampung 10 *byte* dan RA dapat menampung 11 *byte*. SW kemudian diisi dengan 10 karakter pertama dari *frase* dan RA diisi dengan 11 karakter dimulai dari *index* terakhir SW ditambah 1. Karena SW dan RA merupakan *array* dengan tipe data *byte*, maka dibutuhkan *function* ‘ord’ untuk mengkonversikan nilai *char* ke bentuk *byte*. Terdapat sebuah variabel ‘hasil’ yang akan menyimpan hasil dari kompresi yang telah dilakukan. Pertama-tama, variabel hasil diisi dengan SW.

Frase ‘AAAAAAAAAAABCABCAAAAA’

Hasil ‘AAAAAAAAAA’

Setelah literasi awal dilakukan, kemudian dibandingkan apakah isi dari RA sama atau merupakan bagian dari SW, jika tidak maka *index* akhir sari RA akan dikurangi 1. Proses ini terus dilakukan jika isi dari RA sama dengan 2. Jika RA sama dengan 2 maka variabel hasil akan ditambahkan dengan isi dari RA dan *index* awal SW akan bergeser sebanyak 2 ke kanan.

Frase ‘AAAAAAAAAAABCABCAAAAA’

Hasil ‘AAAAAAAAAAAB’

Jika isi dari RA sama atau terdapat pada SW maka pada variabel hasil akan dicatat 2 buah kode yang terdiri dari *offset* dan *length*. Setelah itu *index* awal SW akan bergeser sebanyak isi dari RA.

Frase ‘AAAAAAAAAAABCABCAAAAA’

Hasil ‘AAAAAAAAAAABCA22’

Pada proses kompresi akan berhenti apabila *index* akhir dari SW sudah sama dengna panjang dari frase. Proses kemudian dilanjutkan dengan menambah *flag* pada setiap 8 *byte* pada variabel hasil. Hasil akhir dari proses kompresi dengan metode LZSS ini ditampung dalam variabel ‘hasil2’.

Frase ‘AAAAAAAAAAABCABCAAAAA’

Hasil ‘AAAAAAAAAAABCA2293’

Hasil2 ‘AAAAAAAA’#3’AAABCA22A93’

Isi dari variabel hasil2 tersebut kemudian dikompresi lagi dengan metode *Huffman*. Langkah pertama dari proses kompresi dengna metode *Huffman* adalah 38 dengan membentuk sebuah *tree* yang berasal dari kumpulan *node-node*. Setiap *node* memiliki variabel data dan *value* dan juga memiliki 1 anak yaitu *left* dan *right*. Nilai dari variabel hasil2 akan dikonversikan menjadi bilangan ASCII(0-225). Nilai dari variabel *value* diperoleh dari jumlah kemunculan karakter tersebut.



Gambar 3.1 *Pohon Huffman Proses Pertama*

Adapun pada Huffman *tree* setiap *leaf* adalah karakter yang terdapat pada sebuah *File* yang akan dikompresi. Setiap mencapai suatu *leaf*, proses akan mencatat alur yang terjadi ke dalam senuah variabel '*path*' dan kemudian akan disimpan ke dalam variabel *dictionary* dapat dilihat pada tabel 3.1

3.4 Analisis Terhadap *File Citra*

Analisis dilakukan terhadap *File* berformat *.bmp* dan *.jpg*. pada tahapan ini perbandingan nilai kompresi pada masing-masing *format* di analisa dan digunakan pada tugas akhir ini adalah dengan melakukan perbandingan dua *format citra* berdasarkan nilai kapasitas masing-masing format kompresi.

Rumus untuk mencari nilai kompresi adalah

100%-(a/b×100%),

Dimana: a = kapasitas *File* setelah dikompresi

b = kapsitas *File* sebelum dikompresi

Ukuran kapasitas besar *File* citra ada 2 contoh yang digunakan. Untuk analisa pada tugas akhir ini bervariasi dimana *format File* citra asalnya adalah

menggunakan format *bmp* (*bit map picture*) dan juga format *jpg*. Sehingga dari analisa semuanya didapatkan nilai masing-masing ukuran *File* setelah dikompresi untuk masing-masing format, hal ini terlihat pada tabel 3.1.

Tabel 3.1 Nilai Masing-Masing Citra Sebelum Dan Sesudah Dikompres

Jenis *File* Citra	**Nama Citra**	**Ukuran Asli (*Byte*)**	**Ukuran Kompres (*Byte*)**
bmp	bunga	228536	214613
jpg	*abstrak*	524817	505728

grayscaling citra adalah kegiatan untuk menyederhanakan model citra. Tiga *layer* pada citra berwarna, R-*layer*, G-*layer*, dan B-*layer* diubah menjadi satu *layer grayscale.* Untuk mengubah citra berwarna yang memiliki nilai matrik masing-masing R, G, dan B menjadi citra *grayscale* dengan nilai Ko, citra berwarna dikonversi dengan memberi nilai bobot pada masing-masing *layer*.

$$Ko = wr\,Ri + wg\,Gi + wb\,Bi \quad (1)$$

Berdasarkan NTSC (*National Television System Committee*), yaitu:

wr = 0.299.

wg = 0.587.

wb = 0.144.

Ko adalah nilai *layer* baru yang didapatkan dari hasil perhitungan ketiga *layer* citra.

3.5 Analisis Metode

Salah satu teknik pemampatan data yang akan digunakan yaitu menggunakan algoritma *Lemple Ziv Strorer Symanski* (LZSS). Kemudian hasilnya akan dibandingkan dengan teknik pemampatan menggunakan algoritma *Rice Coding*. Algoritma *Lemple Ziv Storer Symanski* (LZSS) adalah salah satu kompresi urutan simbol data yang terdiri dari sebuah *literal byte* atau pasangan (*offset, match length*), dengan mengidentifikasi urutan simbol yang berulang dalam masukan yang dikopikan secara langsung ke *output*, sehingga menggantikan urutan-urutan simbol yang lebih kecil. Implementasi *Lemple Ziv Storer Symanski* (LZSS) dapat menyesuaikan jumlah *bit* yang dialokasikan dengan mengganti panjang ukuran *byte* (antara parameter lain). Algoritma *Rice Coding* adalah algoritma yang diciptakan oleh *Robert F. Rice,* yang diturunkan melalui sebuah *subse*t dari *Golomb Coding* untuk menghasilkan sebuah kode sederhana yang memakai *suboptimal* dari kode awalan *Golomb Coding.* Algoritma *Rice Coding* ini digunakan dalam skema pengkodean adaptif, yang dapat mengacu hanya menggunakan sebuah *subset* dari *Golomb Coding*.

Untuk mendapatkan hasil kompresi yang baik, maka kita harus mengetahui tahap-tahap dalam proses pemampatan. Algoritma LZSS itu sendiri terdiri atas dua macam, yaitu:

1. Proses *Encoder* merupakan memisahkan data menjadi suatu bentuk pengkodean dengan berupa *string* biner maupun simbol-simbol. Yang nantinya akan dibentuk kembali menjadi data.

2. Proses *Decoder* yaitu menyusun kembali data dari *string* biner menjadi sebuah karakter kembali.

Sekarang kita akan membahas kedua proses tersebut, pertama penulis akan membahas proses *encoder* pada pemampatan data dengan LZSS, setelah itu penulis membahas proses *decoder*-nya.

3.6 Proses *Encode*

Encoding berarti memisahkan data menjadi suatu bentuk pengkodean dengan berupa *string* biner maupun simbol-simbol. Yang nantinya akan dibentuk kembali menjadi data.

3.6.1 Algoritma Lemple Ziv Storer Symansi (LZSS)

Algoritma yang dihasilkan sebagai berikut:

1. Tempatkan posisi koding pada permulaan input *stream.*
2. Cari *longest match* pada *window* untuk *lookahead buffer*;
   a. P = pointer untuk mencocokan;
   b. L = panjang input yang cocok;
3. Apakah L > = MIN_LENGTH?
   a. Jika YA : outputnya P dan bergerak ke posisi L karakter;
   b. Jika TIDAK : outputnya karakter pertama pada *lookhead buffer* dan bergerak satu posisi karakter ke depan;
4. Jika karakter input *stream* banyak, ulangi langkah 2.

Gambar 3.2 Algoritma *Lemple Ziv Storer Symanski*

Cara kerja algoritma *Lemple Ziv Storer Symanski* (LZSS) terdiri dari sebuah *literal bytes* atau pasangan (*offset, match length*). *Literal bytes* merupakan *input byte* sederhana yang dikopikan secara langsung ke *output.* Pasangan (*offset, match length*) menjelaskan proses kembali (*back*) kejarak *offset* suatu *input* data yang nantinya akan ditemukan sebuah *match length* dari data tersebut, LZSS menggunakan dua *flag* pada *output,* yaitu :

1. Jika *flag* “0”, *byte* berikutnya adalah *literal*.
2. Jika *flag* “1”, merupakan pasangan (*offset, match length)*

Sebagai contoh :

a. *Input* Data= “ABCDABCA”

Mulai dengan *byte* “A”, apakah telah ada sebelumnya? “TIDAK”. Enkodenya adalah sebuah *literal byte* “0” diikuti data “A”. *Output* Data = 0”A”. Proses terus berlangsung sampai *byte* berikutnya sama dengan *byte* sebelumnya. Dengan kata lain output data menjadi : 0 “A “ 0 “ B “ 0” C “ 0 “ D”.

b. *Input* Data = “ABCDABCA”

Byte berikutnya adalah “A”, apakah telah ada sebelumnya ? “YA” 4 *bytes* sebelumnya (*offset = 4*). Ada berapa *bytes* yang sama? 3 *byte* yang sama “ABC”, maka akan dikodekan “1” (untuk mengidentifikasi bahwa ada data yang sama) diikuti (4,3). *Output* Data= 0 “A” 0 “B” 0 “C” 0 “D” 1 (4,3).

c. *Input* Data = “ABCDABCA”

Byte berikutnya adalah “A”, apakah telah ada sebelumnya ? “YA” 3 *bytes* sebelumnya (*offset = 3*). Ada berapa *bytes* yang sama? 1 *byte* yang sama “A”, maka akan dikodekan “1” diikuti (3,1). *Output* Data = 0 “A” 0 “B” 0 “C” 0 “D” 1 (4,3) 1 (3,1).

Dibawah ini adalah tabel proses *Encoding* yang disajikan pada tabel 3.1 dan tabel 3.2.

Tabel 3.2 *Input Stream For Coding*

Pos	1	2	3	4	5	6	7	8	9	10	11
Char	A	A	B	B	C	B	B	A	A	B	C

Input stream : barisan karakter yang akan di kompres;

Pos : posisi *coding*;

Char : deretan karakter / elemen data dalam *input stream*;

Proses *Encoding* : proses pengkodean input *Stream*;

Step : langkah proses pengkodean;

Match : karakter yang sama yang ditemukan;

Tabel 3.3 Proses *Encoding* (MIN_LENGTH = 2)

Step	***Pos***	***Match***	***Output***
1	1	--	A
2	2	A	A
3	3	--	B
4	4	B	B
5	4	--	C
6	6	BB	(3,2)
7	8	AAB	(7,3)
8	11	C	C

Proses *Decoding* : proses pengkodean *output Stream*;

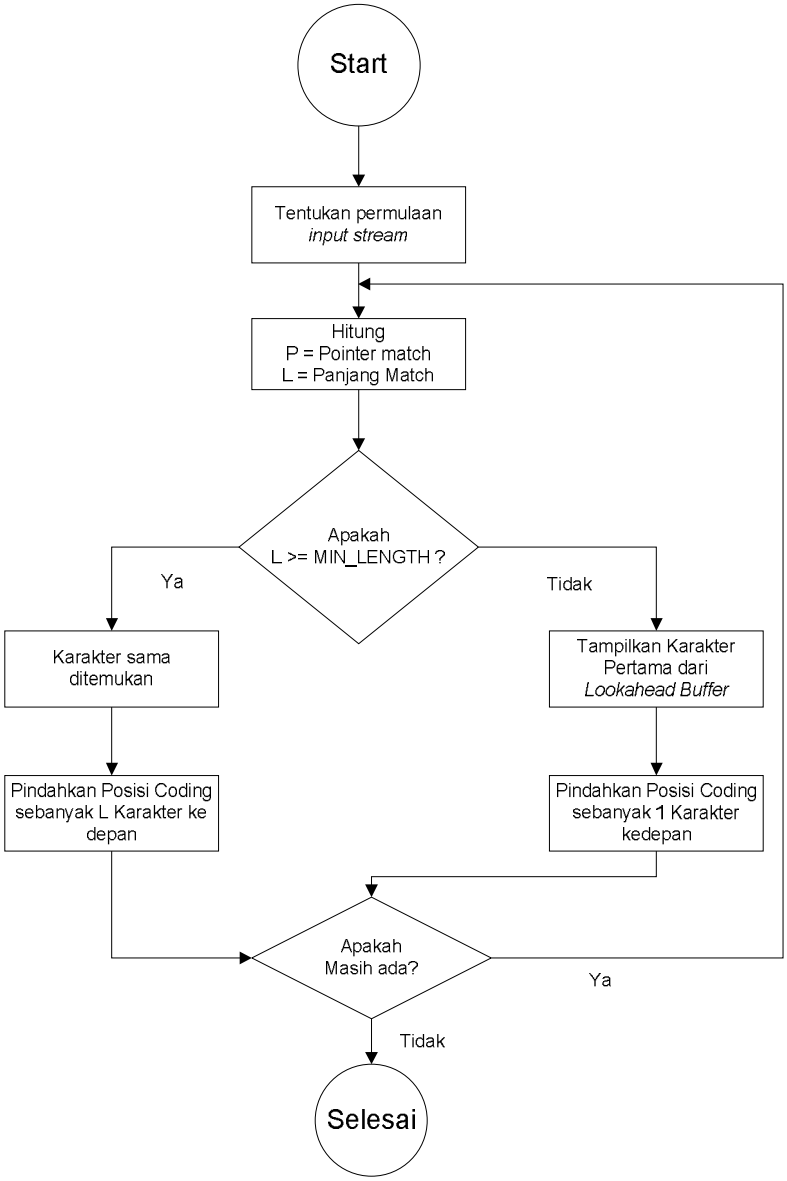
Output Stream : barisan *simbol* hasil proses dekompresi;

Window : ukuran W yang mengandung W karakter dari posisi koding;

Pointer : menunjukan pada *match* dalam *Window*;

Pada proses *decoding*, *window* akan memunculkan *output stream* dengan cara yang sama, dengan algoritman *encoding* pada saat memproses *input stream. Eksplisit* karakter merupakan *output* langsung, dan ketika menunjukan *output* pada *window*.

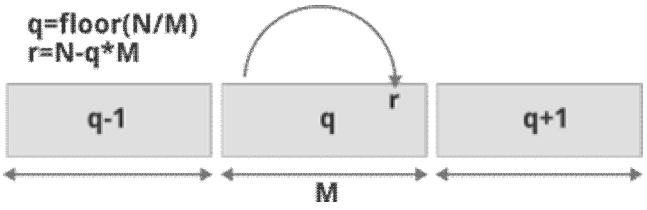
Dari uraian diatas, maka algoritma *encode* untuk pemampatan data dengan menggunakan LZSS dapat disajikan dengan diagram alir sebagai berikut :



Gambar 3.3 Diagram Alir Proses pemampatan data (*encode*)

3.6.2 Algoritma Rice Coding

Algoritma *Rice Coding* ini digunakan dalam skema pengkodean adaptif, yang akan ditunjukan pada gambar 3.3 yang dapat mengacu hanya menggunakan sebuah *subset* dari turunan *Golomb Coding*.



Gambar 3.4 Pengkodean *Integer* pada *Rice Coding*

Pada gambar diatas *Rice coding* dapat dianggap sebagai kode yang menunjukkan pada posisi *(q),* dan *offset* dalam *(r).* Sehingga posisi *q*, dan *r offset* untuk pengkodean integer *N* pada *Rice Coding* yang menggunakan parameter *M.*

Secara *formal*, dua bagian yang diberikan oleh ekspresi berikut, di mana x adalah nomor yang disandikan :

$$q = \left\lfloor \frac{(x-1)}{M} \right\rfloor$$ Dan $r = x - q\,M - 1$ Hasil akhir terlihat seperti : $(q+1)\,r$ (2.4)

Perhatikan bahwa *r* dapat dari berbagai jumlah bit, dan secara khusus bit *b* hanya untuk *Rice Coding,* dan menghubungkan antara b-1 dan b bit untuk kode *Golomb* (M yaitu bukan kelipatan dari 2) :

$$b = \lceil \log_2(M) \rceil \tag{2.5}$$

Jika $0 \leq r < 2^b - M$ gunakan -1 *b* bit untuk mengkodekan *r*.

Jika , $2^b - M \leq r < M$ gunakan bit untuk mengkodekan *r.*

b = log $_2$ (*M*) Jika M adalah hasil kedua yang dapat dikodekan semua nilai r dengan bit b.

Algoritma untuk *Rice Coding* adalah pengkodean biner secara sederhana. Contoh dari algoritma *Rice Coding* berikut:

1. Tetapkan nilai M sebagai integer.
2. Lakukan perulangan dari N, sampai jumlah yang akan di encode cari:
    1. q = integer [N/M]
    2. r = N mod M
3. Tampilkan (*generate*) *codeword*
    1. Format kode : (q) (r), dimana
    2. Code q
        a. Tulis sebanyak panjang *string* q 1 bit
        b. Tulis sebuah 0 bit
    3. Codc r
        1. Jika M adalah kelipatan dari 2, code r adalah format biner. Jadi diperlukan sebanyak log $_2$ (*M*) bit.
        2. Jika M adalah bukan kelipatan dari 2,
            a. Jika $r < 2^b - M$ code r adalah biner akan digunakan sebanyak b-1 bit.
            b. Jika jumlah codenya $r + 2^b - M$ ditulis dalam biner yang direpresentasikan sebanyak b bit.

Gambar 3.5 Algoritma *Rice Coding*

Sebagai Contoh :

Set *M* = 10. Demikian $b = \lceil \log_2(10) \rceil = 4$ dan 2 ^{b} − *M* = 16 − 10 = 6 kemudian 2 ^{b} - *M* = 16 - 10 = 6.

Tabel 3.4 Tabel *Encoding* Hasil bagi

***Encoding* Hasil bagi**	
q	**Bits hasil**
0	0
1	10
2	110
3	1110
4	11110
5	111110
6	1111110
:	:
N	(N mengulang pada 1) 0

Tabel 3.5 Tabel *encoding* Sisa hasil

***Encoding* Bagian Sisa**			
r	***Offset***	**Biner**	***Output* bit**
0	0	0000	000
1	1	0001	001
2	2	0010	010
3	3	0011	011
4	4	0100	100
5	5	0101	101
6	6	1100	1100

7	7	1101	1101
8	8	1110	1110
9	9	1111	1111

Sebagai contoh:

Catatan: kode ini mengasumsikan dasar *Rice Coding*, yaitu, satu dengan M setara dengan pangkat 2, tetapi tidak seperti kasus diatas ketika bit *encoding* terpotong dari sisa pembagiannya maka akan lebih baik (bila M adalah bukan kelipatan dari 2, seperti pada contoh sebelumnya).

Pengkodean

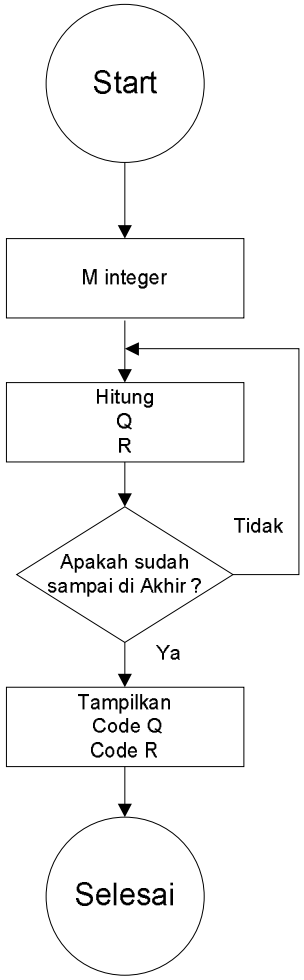
```
void golombEncode (sumber char *, char * dest, int M)
{
   IntReader intreader (sumber);
   BitWriter bitwriter (dest);
   sementara (intreader. hasLeft ())
   {
      intreader int = num). getInt (;
      int q = num / M;
      for (int i = 0; i <q; i + +)
         bitwriter;. putBit (true) / / menulis yang q
      bitwriter;. putBit (false) / / menulis satu nol
      int v = 1;
      for (int i = 0; i <log2 (M); i + +)
      {
         bitwriter). putBit (v & num;
         v = v <<1;
      }
   }
   bitwriter). dekat (;
   intreader). dekat (;
}
```

Gambar 3.6 Dasar Pengkodean Algoritma *Rice Coding*

```
Decoding
 void golombDecode (sumber char *, char * dest, int M)
 {
    BitReader bitreader (sumber);
    IntWriter intwriter (dest);
    int q = 0;
    int nr = 0;
    sementara (bitreader. hasLeft ())
    {
       nr = 0;
       q = 0;
       sementara (bitreader. getBit ()) q + +; / / berpotensi berbahaya dengan file format yang salah.
       for (int a = 0; sebuah <log2 (M): a + +) / / membacakan berurutan log2 (M) bit
          if (bitreader. getBit ())
             nr + = 1 <<a;
       nr + = q * M; / / tambahkan bit dan beberapa M
       intwriter;. putInt (nr) / / menulis nilai
    }
    bitreader). dekat (;
    intwriter (. close)
 }
```

Gambar 3.7 Dasar *Decoding* Algoritma *Rice Coding*

Dari uraian diatas, maka algoritma *encode* untuk pemampatan data dengan menggunakan *Rice Coding* dapat disajikan dengan diagram alir sebagai berikut :



Gambar 3.8 Diagram Alir Proses pemampatan data (*encode*)

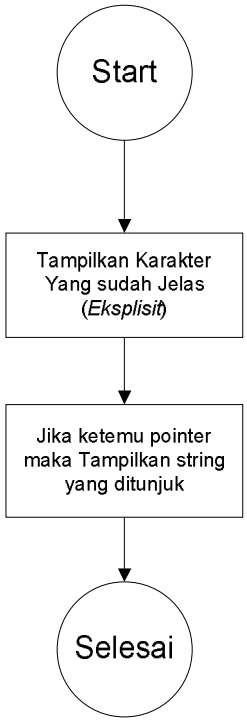
3.7 Proses *Decode*

Decoding merupakan kebalikan dari *encoding*. *Decoding* berarti menyusun kembali data dari *string* biner menjadi sebuah karakter kembali.

3.7.1 Lemple Ziv Storer Symanski (LZSS)

Pada metode *Lemple Ziv Storer Symanski* (LZSS), Pertama tampilkan karakter yang sudah eksplisit, kemudian jika bertemu dengan pointer

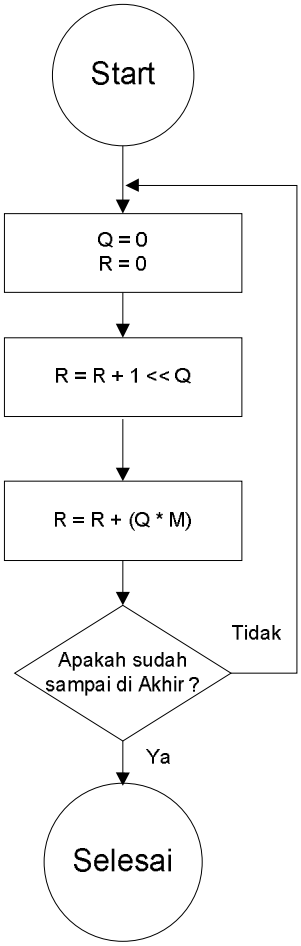
maka tampilkan string yang ditunjuk. Dari uraian diatas, maka algoritma *decode* untuk pemampatan data dapat disajikan dengan diagram alir sebagai berikut :



Gambar 3.9 Diagram Alir Proses pemampatan data (*Decode*)

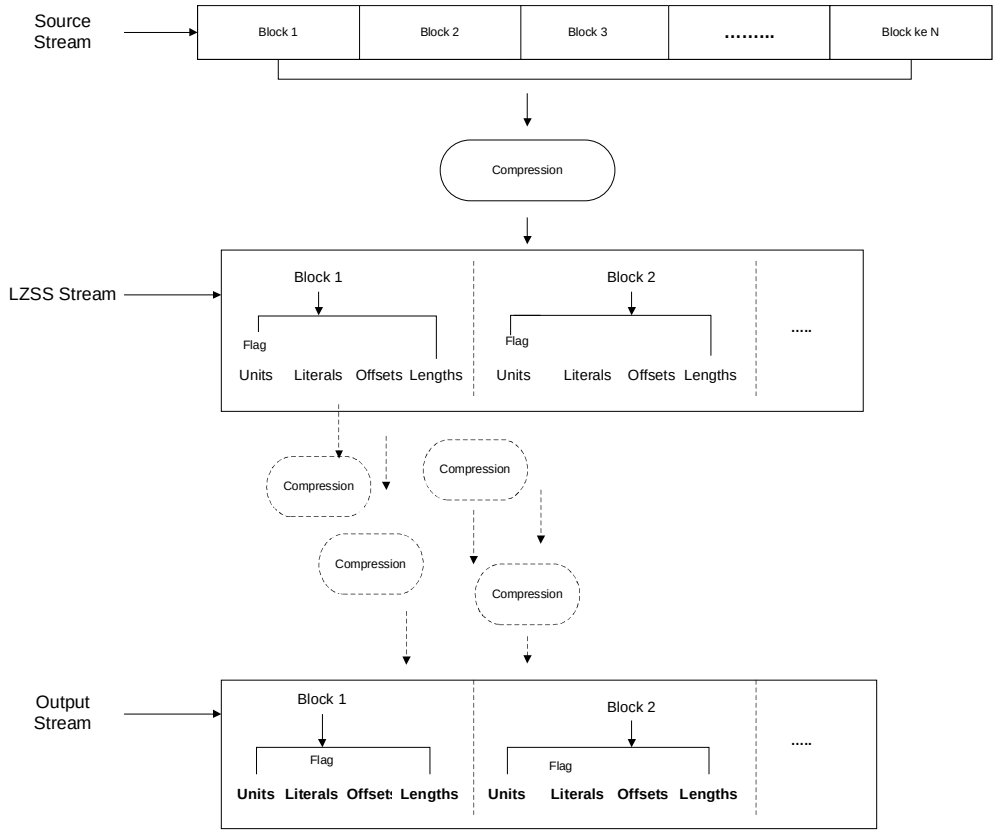
3.7.2 Algoritma Rice Coding

Decoding pada *Rice Coding* dengan menggunakan pengkodean biner yaitu dari proses *encode* diatas, maka untuk proses *decode* pada *Rice Coding* adalah kebalikan dari *encode* itu sendiri. Sehingga pada metode *Rice Coding* untuk proses *decode* pemampatan data dapat disajikan dengan diagram alir sebagai berikut :



Gambar 3.10 Diagram Alir Proses pemampatan data (*Decode*)

3.8 Pemisahan Masing-masing *Block* Data



Gambar 3.11 Pemisahan Masing-masing Block Data

Algoritma kompresi LZSS menghasilkan beberapa blok data yang masing-masingnya tediri dari blok pertama sampai blok data ke N. Pada Semua kelompok blok data yang berasal dari *source stream* akan dikompresi berdasarkan masing-masing perblok data, yang kemudian akan ditampung di LZSS *stream* perblok data tersebut didalamnya terdapat *flag unit, literal, offser, lengths.* Setelah dari LZSS *stream* lalu ke *output stream* yang nantinya akan ditampilkan blok yang telah dipisahkan berdasarkan kapasitas masing-masing blok data.

3.9 Window dan Match Length

Ada beberapa aturan agar sistem dapat bekerja yaitu:

a. Untuk *flag* mengidentifikasi sebuah *literal* atau *match.*

b. Seberapa jauh data sebelumnya dapat dicocokan.

c. Jumlah *bytes* maksimum yang dapat dicocokan.

Untuk aturan yang pertama hanya membutuhkan dua *flag* yaitu *literal* atau *match*, jadi hanya menggunakan bit tunggal. Jika menggunakan bit tunggal maka seluruh *bytes* tidak selalu ditulis tapi hanya sebagian saja. Sebelum proses pemampatan dilakukan, dibutuhkan suatu fungsi kode untuk membaca dan menulis jumlah *variable* bit dari atau ke data *stream.*

Sebagai contoh : batas *offset* menggunakan 4 *bits*, jadi range-nya antara 0-31. Sedangkan untuk len menggunakan 3 bits jadi range-nya antara 0-7. Outputnya akan ditulis menjadi (4,3) dalam byte menjadi “1 0100 011”. 1 bit pertama menunjukan *flag,* diikuti 4 bits berikutnya yang menunjukan *offset,* kemudian 3 bits berikutnya menunjukan *len.*

Contoh penggunaannya :

Input “1231231” dalam biner ditulis menjadi 56 bits dalam pengkodean ASCII:

00000001 00000010 00000011 00000001

00000010 00000011 00000001

Dikodekan menjadi :

1. *Input* Data = “1231231”

   Mulai dengan *byte* “1”. Apakah pernah ada sebelumnya?

   “TIDAK”, maka akan dienkodekan sebagai *literal*. *Output* Data = 0 0 0000001

2. *Input* Data = “1231231”

   Byte berikutnya “2”. Apakah pernah ada sebelumnya? “TIDAK”, maka dienkodekan sebagai *literal*.

   Output Data = 0 0 0000001 0 00000010

3. *Input* Data = “1231231”

   Byte berikutnya “3”. Apakah pernah ada sebelumnya? “TIDAK”, maka dienkodekan sebagai *literal*.

   Output Data = 0 0000001 0 00000010 0 00000011

4. *Input* Data = “1231231”

   Byte berikutnya “1”. Apakah pernah ada sebelumnya? “YA”, 3 bytes sebelumnya. Berapa bytes yang sama ? 3 bytes “123”, maka outputnya untuk *flag* “1” (1 *bit*) diikuti *offset* “3” (4 *bits*), dan untuk *match lenght* “3” (3 *bits*). *Output* Data = 0 0000001 0 0000001 0 00000010 0 00000011 1 0011 011.

5. *Input* Data = “1231231”

   Byte berikutnya “1”. Apakah pernah ada sebelumnya? “YA”, 3 bytes sebelumnya. Berapa *bytes* yang sama ? 1 *bytes* “1”, maka outputnya untuk *flag* “1” (1 *bit*) diikuti *offset* “3” (4 *bits*), dan untuk *lenght* “1” (3 *bits*). Output Data = 0 0000001 0 00000010

0 00000011 1 0011 011 1 0011 001. *Input* data yang tadinya 56 bits, *output* datanya menjadi 43 bits.

00000011 1 0011 011 1 0011 001. *Input* data yang tadinya 56 bits, *output* datanya menjadi 43 bits.

3.10 Struktur dari *CAR File*

Struktur dari *CAR File* sangat sederhana yaitu terdiri dari daftar *sequential block File header* yang diikuti oleh *File* data. Barisan atau *sequence* ini terus berulang tidak terbatas sampai sebuah *header* khusus menjumpai *File name null.* Seperti table di bawah ini.

Tabel 3. 6 Struktur dari *CAR File*

File 1 Header
File 1 Data
File 2 Header
File 2 Data
File 3 Header
File 3 Data
-
-
-
EOF Header

Struktur *sequential* dibawah ini memiliki kelebihan dan kekurangan. Dalam data mentah berikutnya maksudnya pencarian dan

penambahan arsip tidak dapat dilakukan dengan menggunakan *random access method.* Oleh karena itu pencarian linier dan pengkopian digunkan, akan tetapi setiap waktu *File* dalam arsip akan dimodifikasi maksudnya semua arsip akan dikopikan dari *File* aslinya ke *File* versi terbaru.

3.11 Analisis Kebutuhan Non Fungsional

Analisis kebutuhan non fungsional merupakan bagian penting agar program aplikasi dapat berjalan dengan baik. untuk mengetahui elemen-elemen yang berhubungan dengan sistem yang sedang berjalan.

Program aplikasi pemampatan data ini membutuhkan spesifikasi perangkat keras dan perangkat lunak pendukung dengan spesifikasi sebagai berikut:

3.11.1 Analisis Perangkat Keras

Analisis perangkat keras merupakan spesifikasi *hardware* yang digunakan agar program aplikasi pemampatan pada data dapat berjalan dengan baik yang telah difungsikan.

3.11.1.1 Perangkat keras pendukung penelitian

Perangkat keras yang digunakan untuk mengimplementasikan dan pengujian sistem ini adalah sebagai berikut:

1. *Processor* : *AMD AtlonX2 5000 + 2,6 GHz*
2. *Memory* : 2 GB DDR
3. *Harddisk* : 120 GB
4. *VGA* : 256 MB
5. Monitor *LG 17 inch*.

6. *Keyboard* dan *mouse*

3.11.1.2 Perangkat keras *spesifikasi* terendah aplikasi

Spesifikasi terendah dari perangkat keras yang dapat digunakan untuk mengimplementasikan dan pengujian sistem ini adalah sebagai berikut:

a. *Processor* : *Intel Pentium* IV 1.3 *GHz*
b. *Memory* : 128 MB DDR
c. *Harddisk* : 80 GB
d. *VGA on Board* : 32 MB

3.11.2 Analisis Perangkat lunak

Analisis perangkat lunak adalah untuk mengetahui spesifikasi perangkat lunak minimum yang dibutuhkan dalam membangun aplikasi pemampatan.

3.11.2.1 Perangkat lunak pendukung penelitian

Spesifikasi perangkat lunak yang digunakan untukn membuat serta untuk mengimplementasikan program aplikasi pemampatan data ini adalah sebagai berikut :

a. Sistem Operasi menggunakan *Windows XP Profesional SP2*.
b. Bahasa pemrograman *Borland Delphi 7.0*

3.11.3 Analisis Kebutuhan Pengguna (*User*)

Pengguna dari aplikasi ini adalah orang yang sudah berpengalaman dalam menggunakan komputer atau yang telah mengikuti *training* aplikasi ini. Aplikasi yang dibangun digunakan oleh dua pengguna yaitu *server* dan *client*. *Server* berfungsi sebagai pengguna untuk mengencodekan (kompresi) sedangkan *client* berfungsi untuk mendecodekan (dekompresi).

Tabel 3.7 Tabel Spesifikasi *User*

Pengguna	Tanggung Jawab	Hak Akses	Tingkat Pendidikan (Usia)	Tingkat Keterampilan	Pengalaman
user	-Melakukan proses kompresi (*encode*) data, kemudian pilih data yang akan dikirim -Melakukan dekompresi (*decode*) data yang telah di	-Hanya melakukan proses kompresi (*encode)* dan menerima koneksi dari *client* -Hanya melakukan proses dekompresi(*de code)* dan	Sekolah Menengah Atas Usia minimal 16 tahun s/d maksimal 45 tahun.	Mengerti Jaringan dan memasukan data	Pelatihan Aplikasi dan Mengerti Jaringan

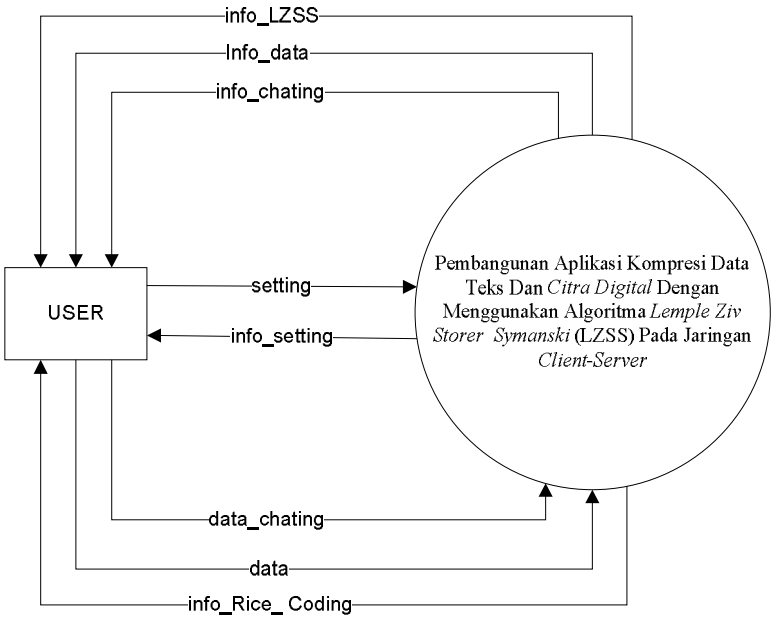
	kompresi	menghubungkan koneksi ke *server*			

3.12 Analisis Kebutuhan Fungsional

Analisis kebutuhan fungsional ini meliputi *Data Flow Diagram* (DFD). Data *Flow Diagram server* dibawah ini merupakan alat yang dapat menggambarkan suatu proses di dalam sistem secara terstruktur dan jelas, serta dokumentasi dari sistem yang baik. DFD suatu level dapat diturunkan menjadi *level* berikutnya.

3.12.1 Diagram Konteks

Diagram konteks menggambarkan hubungan antara sistem dengan entitas luarnya. Diagram konteks berfungsi sebagai transformasi dari satu proses yang melakukan transformasi data *input* menjadi data data *output.*

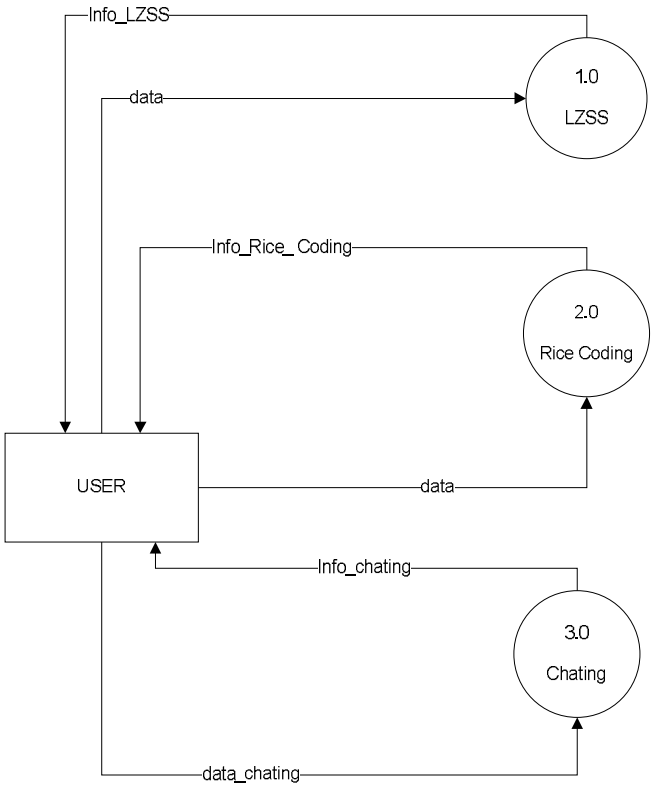


Gambar 3.12 Diagram Konteks

3.12.2 Kebutuhan Fungsional *Server*

Kebutuhan fungsional *server* membahas mengenai fungsional yang ada di dalam aplikasi yang digunakan oleh *server*. Sehingga dapat digambarkan oleh gambar – gambar berikut ini :

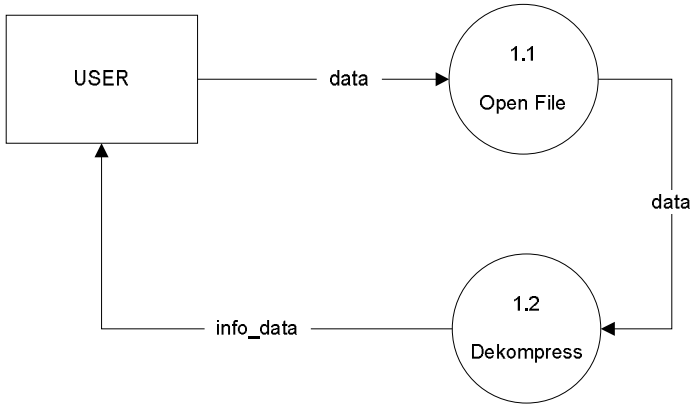
3.12.2.1 DFD Level 1



Gambar 3.13 DFD Level 1 (*Server*)

3.12.2.2 DFD Level 2 Proses 1 (*decode*)

Proses *decode* digunakan oleh *server* yang akan menerima dan mendekompresi data pada saat data dikirim melalui *client* yang telah terkoneksi melalui jaringan.



Gambar 3.14 DFD Level 2 Proses 1.0 (*server*)

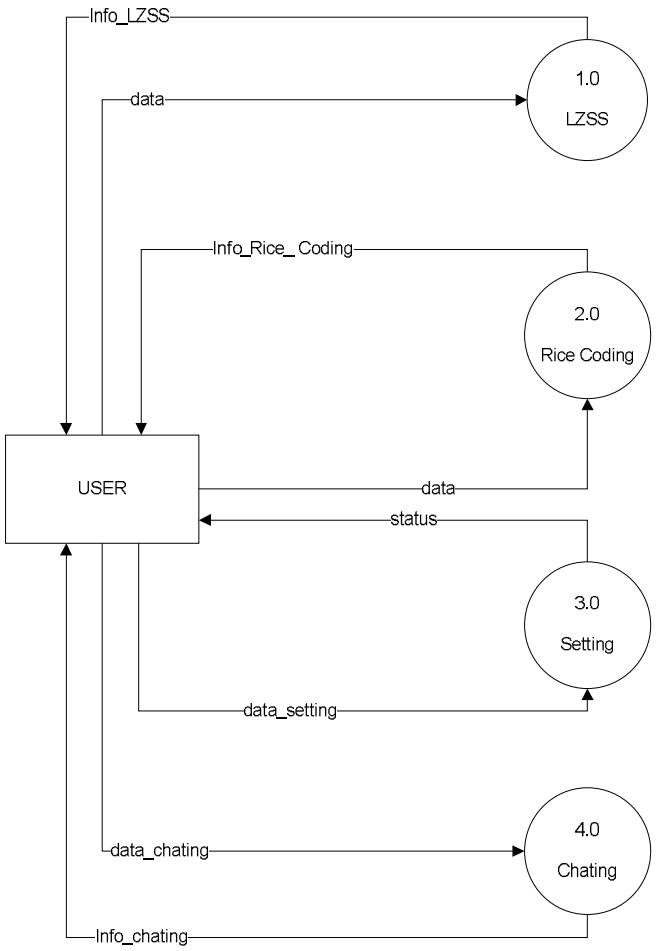
Tabel 3.8 Spesifikasi Proses *Server*

No. proses	*Input*	Proses	*Output*
Proses 1.0	LZSS	Menampilkan metode LZSS	Info LZSS
Proses 2.0	*Rice Coding*	Menampilkan metode *Rice Coding*	Info *Rice Coding*
Proses 3.0	*Chatting*	menampilkan pesan dari *client*	Info data *user*
Proses 1.1	*Open File*	Ambil data berupa teks dan citra digital	Info data
Proses 1.2	*Dekompress*	Ambil data berupa teks dan citra digital Data hasil *encode* akan di *decode* dengan LZSS.	Info data
Proses 2.1	*Open File*	Ambil data berupa teks dan citra digital	Info data
Proses 2.2	*Dekompress*	Ambil data berupa teks dan citra digital Data hasil *encode* akan di *decode* dengan *Rice Coding.*	Info data

3.13 Kebutuhan Fungsional *Client*

Kebutuhan fungsional *client* merupakan alat yang dapat menggambarkan suatu proses di dalam aplikasi yang digambarkan oleh gambar – gambar berikut ini :

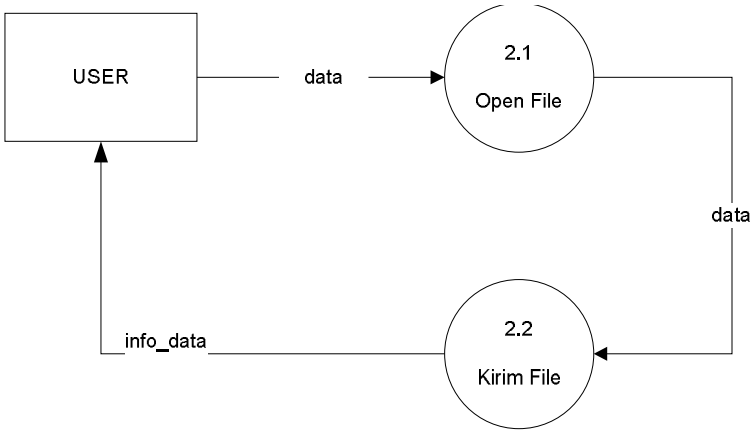
3.13.1 DFD Level 1



Gambar 3.15 DFD Level 1 (*Client*)

3.13.2 DFD Level 2 Proses 2 (*encode*)

Proses kompresi (*encode)* dilakukan oleh *client* yang akan mengkompresi data pada saat data dikirim melalui jaringan yang telah terkoneksi antara *server* dan *client*.



Gambar 3.16 DFD Level 2 Proses 2.0 (*Client*)

Tabel 3.9 Spesifikasi Proses *Client*

No. proses	*Input*	Proses	*Output*
Proses 1.0	LZSS	Menampilkan metode LZSS	Info LZSS
Proses 2.0	*Rice Coding*	Menampilkan metode *Rice Coding*	Info *Rice Coding*
Proses 3.0	*Setting*	untuk mengkoneksikan dari *client* ke *server*	Info data *user*
Proses 4.0	*Chatting*	menampilkan pesan dari *client*	Info data *user*
Proses 1.1	*Open File*	Ambil data berupa teks dan citra *digital*	Info data
Proses 1.2	Kirim *File*	Data yang akan di kirim hasil *encode* dengan LZSS	Info data
Proses 2.1	*Open File*	Ambil data berupa teks dan	Info data

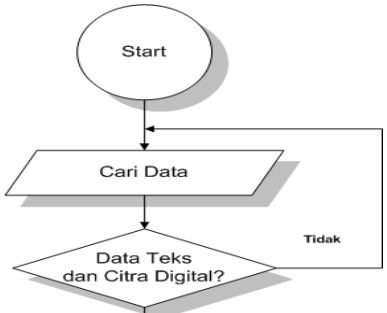
		citra digital	
Proses 2.2	Kirim *File*	Data hasil yang akan di *encode* dan kirim dengan metode *Rice Coding*	Info data

3.14 Perancangan Sistem

Perancangan merupakan tahap lanjutan dari analisis sistem dimana pada perancangan sistem digambarkan rancangan sistem yang akan dibangun sebelum dilakukan pengkodean kedalam suatu bahasa pemrograman.

3.14.1 Gambaran umum sistem

Berikut ini adalah diagram alir yang menggambarkan langkah-langkah penyelesaian masalah tersebut :



Gambar 3.17 Diagram Alir Gambaran umum sistem

Berikut ini adalah penjelasan dari diagram alir langkah-langkah penyelesaian masalah:

1. Ambil Data

   Langkah pertama yang dilakukan adalah mengambil data dari media penyimpanan. Kemudian data tersebut akan diperiksa tipe *File*nya, untuk data teks apakah bertipe *doc,docx, pdf, rtf* dan untuk citra *digital* apakah bertipe *jpg, bmp* atau bukan? Jika ”ya” maka langkah selanjutnya dilakukan proses *Encode.* Jika ”tidak” maka kembali lagi pada langkah pertama yaitu mengambil citra yang sesuai.

2. *Encode*

Data yang berekstensi diatas tersebut, kemudian digunakan untuk proses *encode* (proses pemampatan terjadi) yang hanya dilakukan oleh *client*.

3. Decode

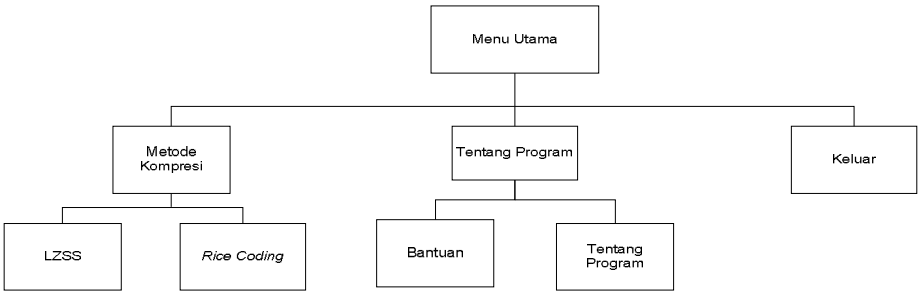
Decode merupakan proses yang dilakukan oleh *server,* yaitu lanjutan dari proses *encoding*. Tujuan dari proses ini adalah menampilkan data yang telah dimampatkan oleh *client* dalam bentuk aslinya.

4. Data *Rekonstruksi*

Data ini merupakan hasil keluaran dari proses *decode*, yaitu data yang telah dimampatkan.

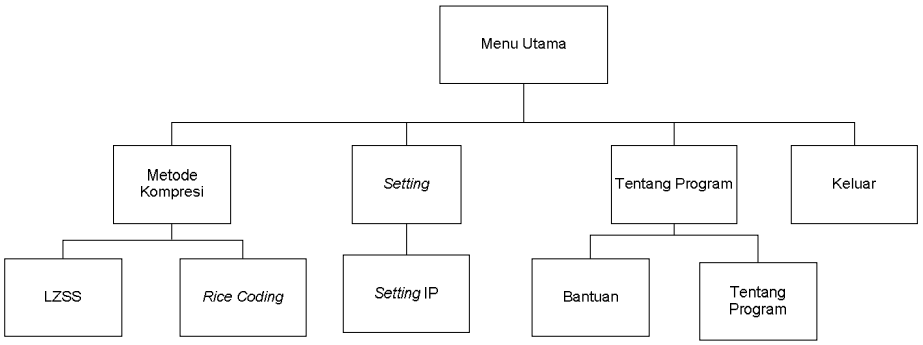
3.15 Perancangan Struktu Menu

1. Tampilan struktur menu *Server*



Gambar 3.18 Struktur Menu Utama *Server*

2. Tampilan struktur menu *Client*



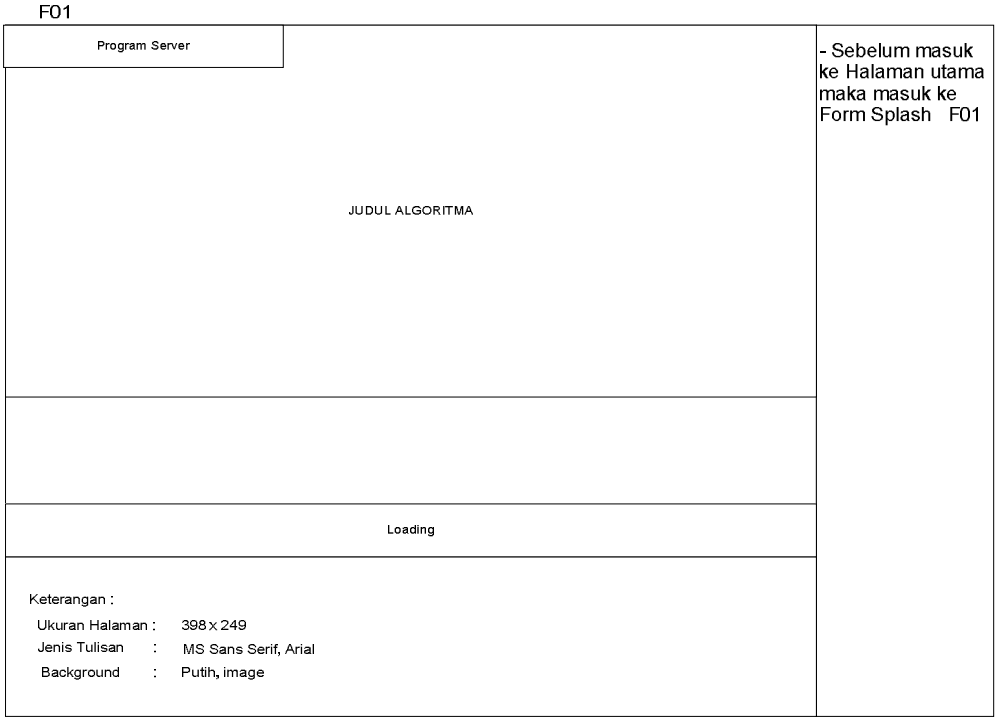
Gambar 3.19 Struktur Menu Utama *Client*

Perancangan antarmuka digunakan untuk memberikan gambaran mengernai proses-proses apa saja yang bisa dilakukan pengguna dalam aplikasi yang akan dibangun, yang didasarkan hasil *identifikasi* dan penelitian yaitu bagaimana aplikasi akan memberikan fasilitas-fasilitas dengan harapan dapat mempermudah dalam mengakses dan menggunakan program Pembagunan Aplikasi Kompresi Data teks dan *Citra Digital* Dengan Menggunakan Algoritma *Lemple Ziv Storer Symansi* (LZSS) Pada Jaringan *Client-Server.*

3.16 Perancangan Antarmuka *Server*

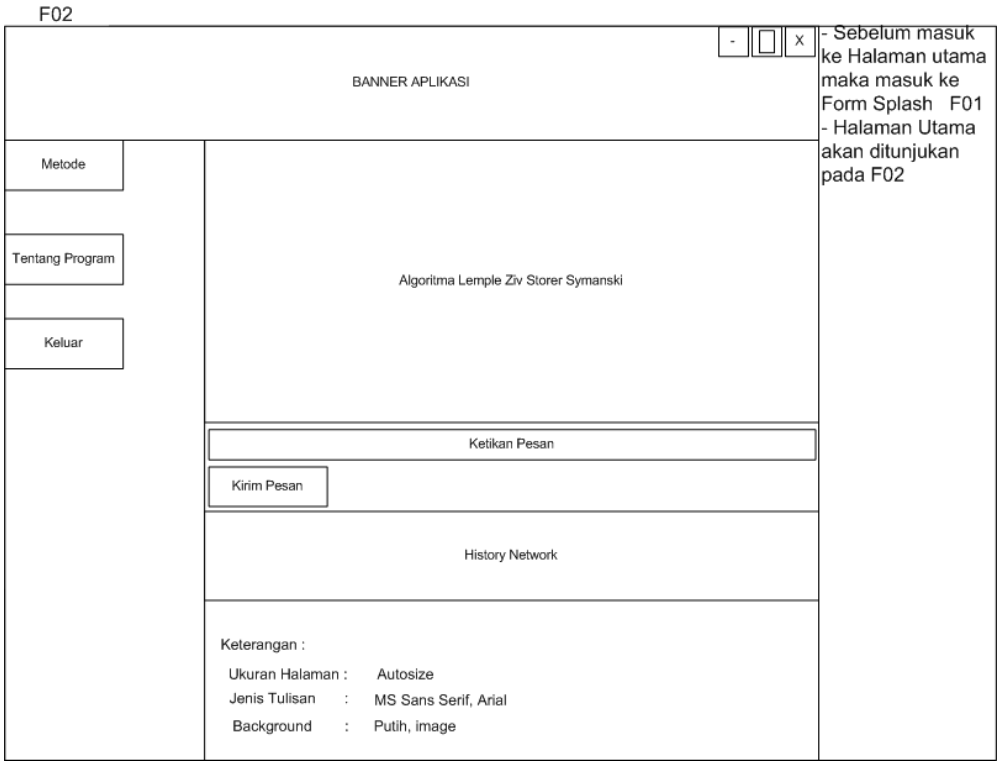
Perancangan antarmuka aplikasi yang dilakukan oleh pada *server* yaitu :

1. Perancangan antar muka *Form Splash.*



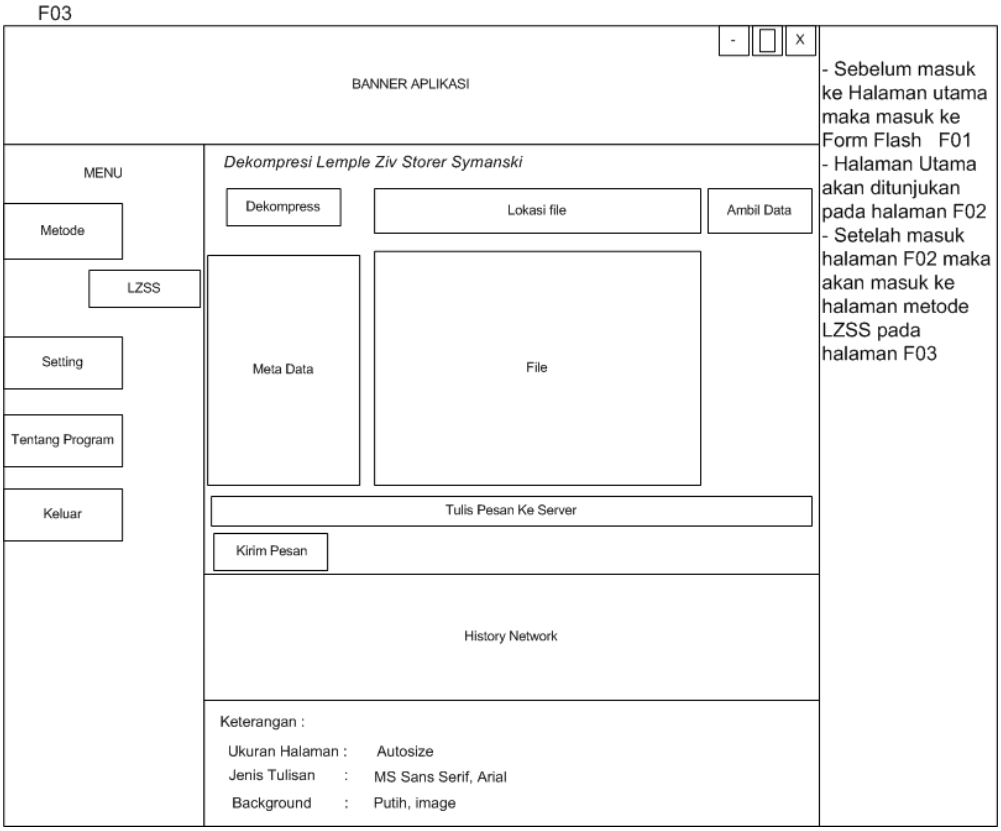
Gambar 3.20 Antarmuka *Form Splash*

2. Perancangan antarmuka Menu Utama *Server*



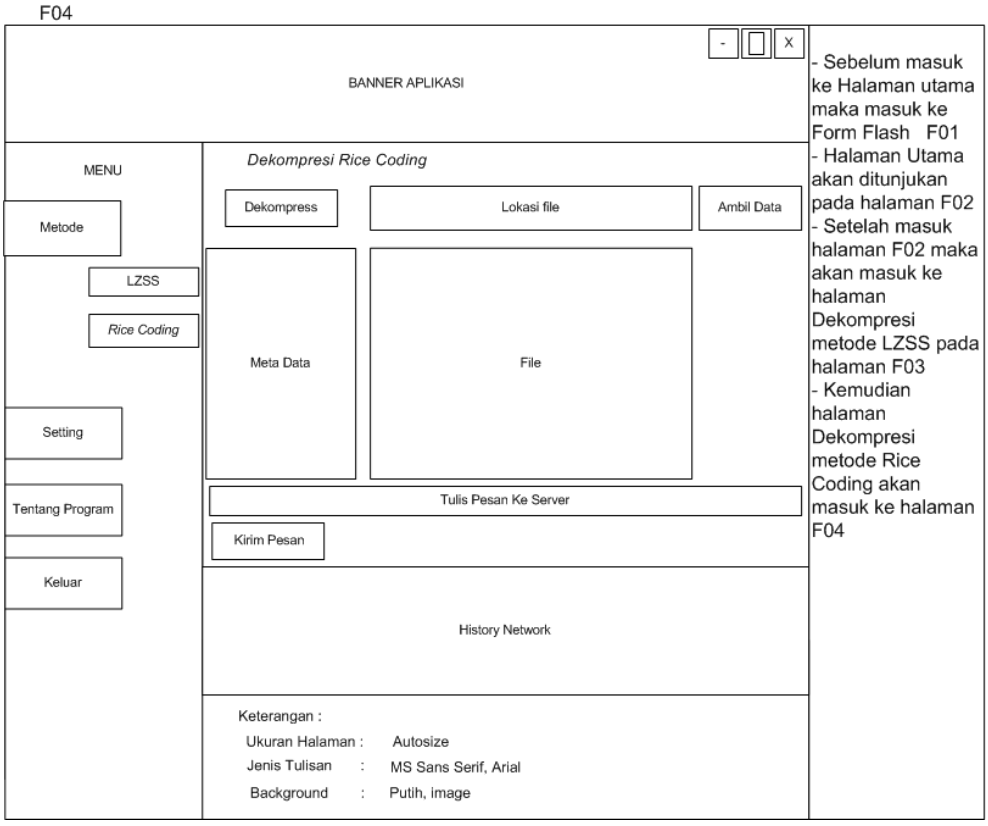
Gambar 3.21 Antarmuka Menu Utama

3. Perancangan antarmuka Kompresi Metode *Lemple Ziv Storer Symanski* (LZSS).



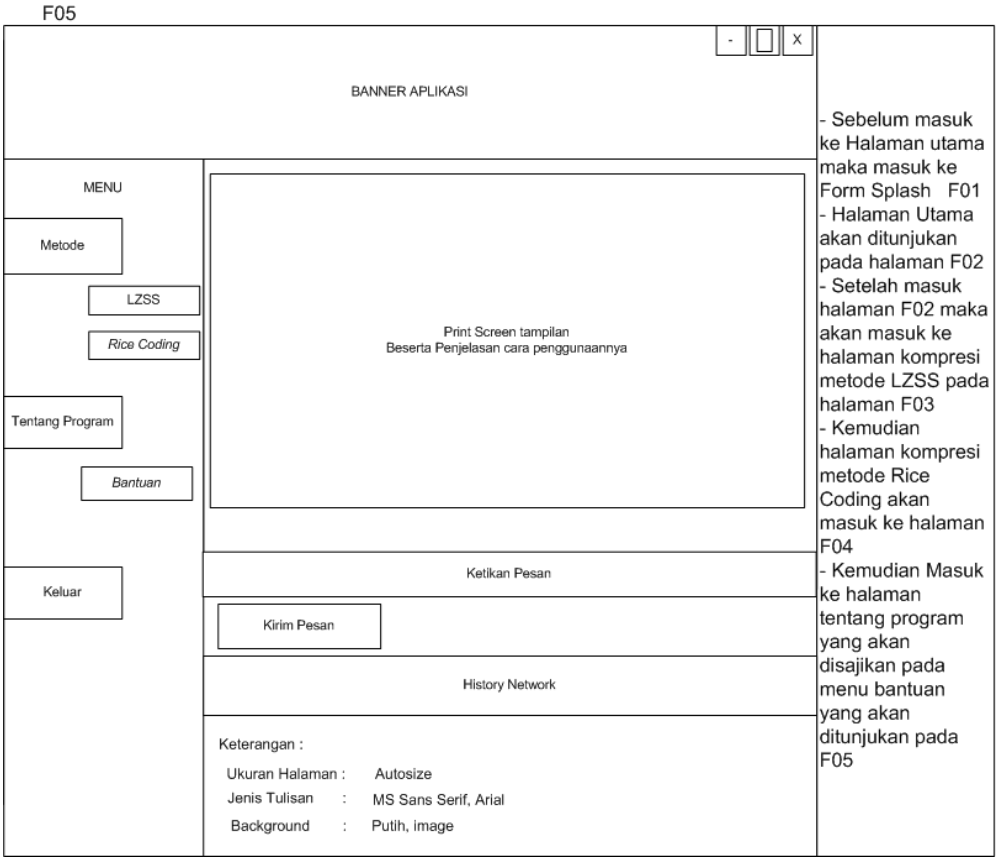
Gambar 3.22 Antarmuka Kompresi Metode LZSS

4. Perancangan antarmuka Kompresi Metode *Rice Coding*



Gambar 3.23 Antarmuka Kompresi metode *Rice Coding*

5. Perancangan antarmuka Menu Bantuan

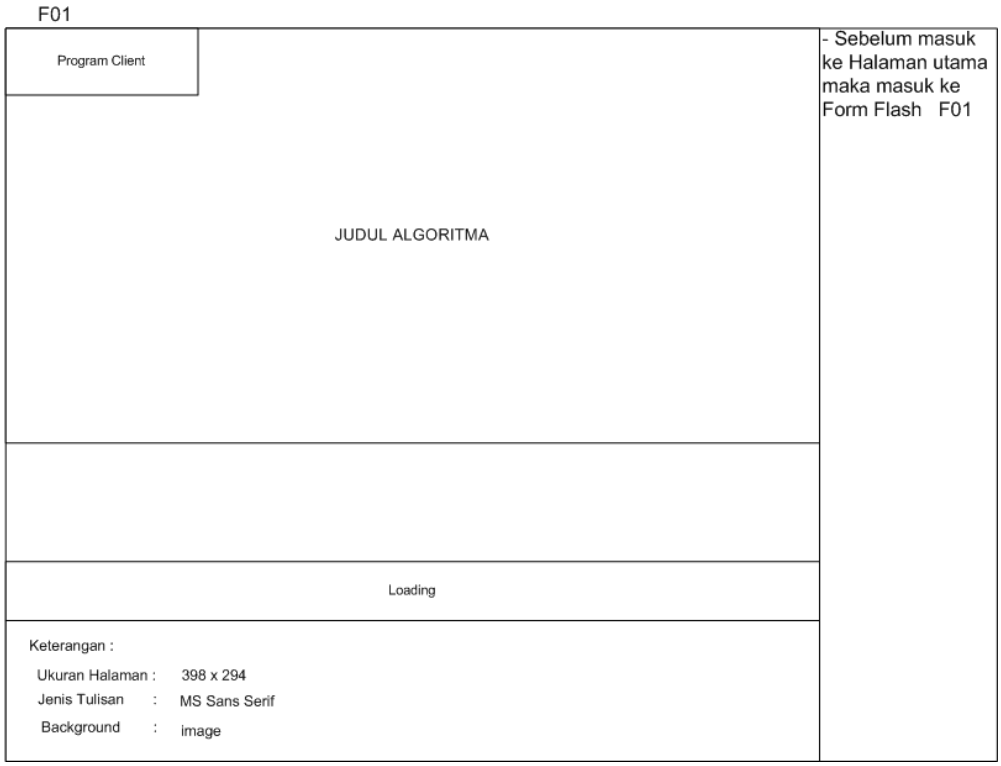


Gambar 3.24 Antarmuka Menu Bantuan

3.17 Perancangan Antarmuka *Client*

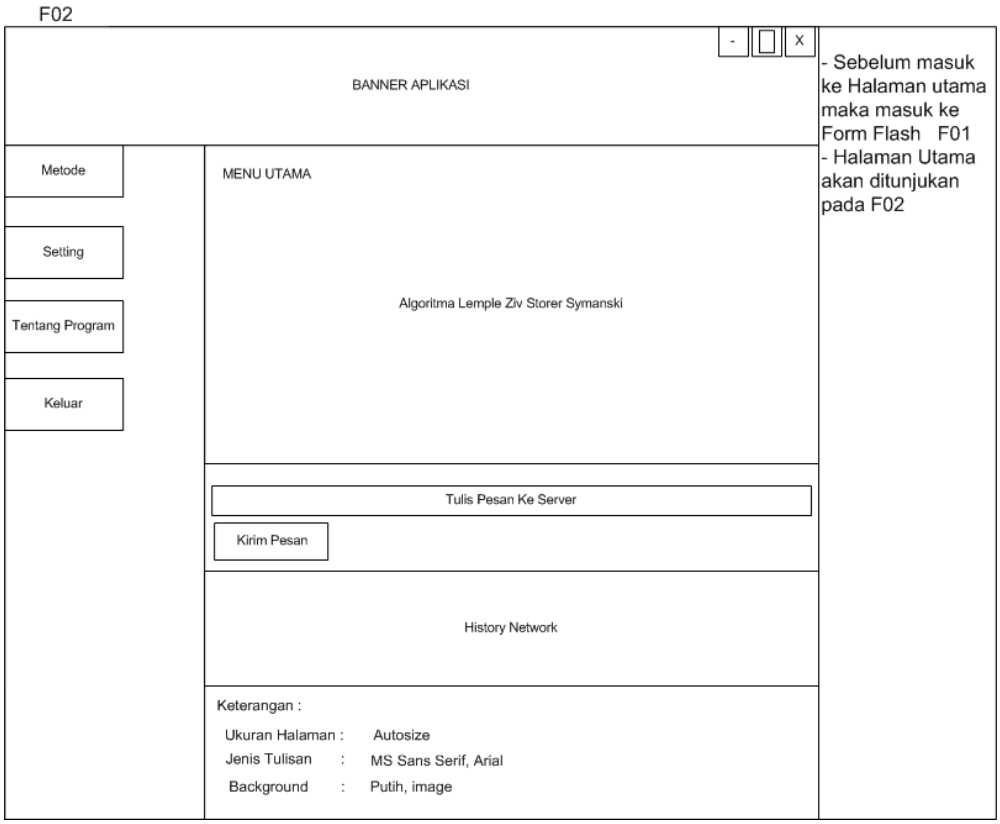
Perancangan antarmuka aplikasi yang dilakukan oleh pada *Client* yaitu :

1. Perancangan antar muka *Form Splash.*



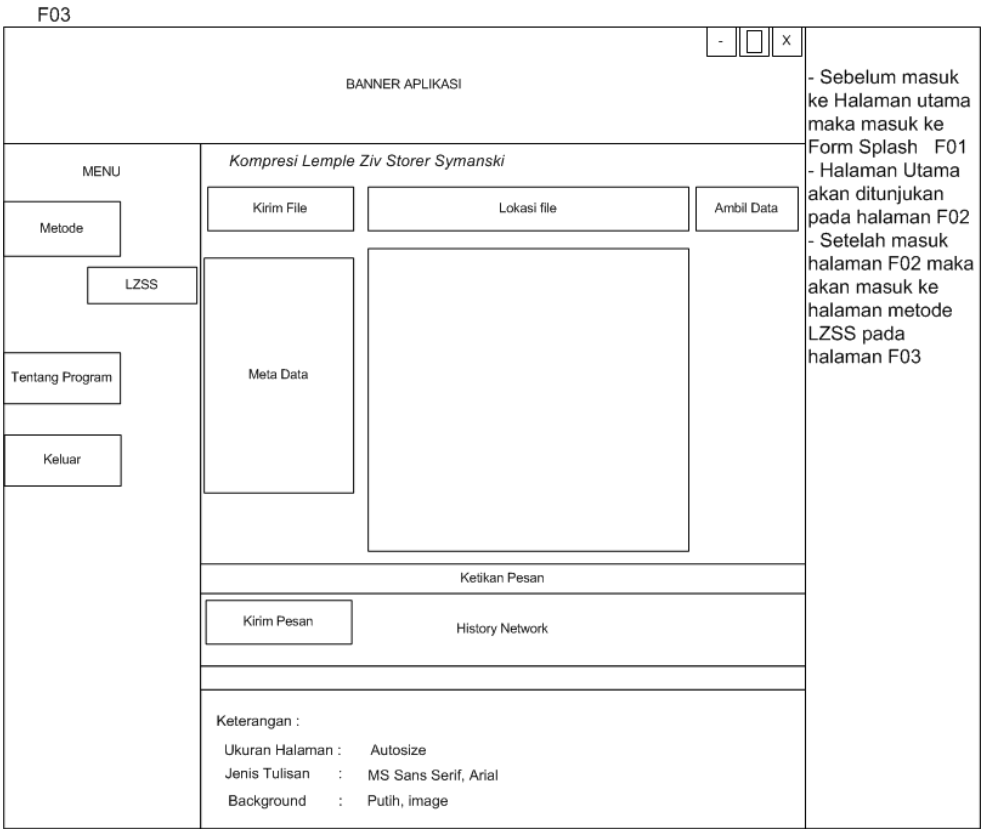
Gambar 3.25 Antarmuka *Form Splash*

2. Perancangan antarmuka Menu Utama *Client*



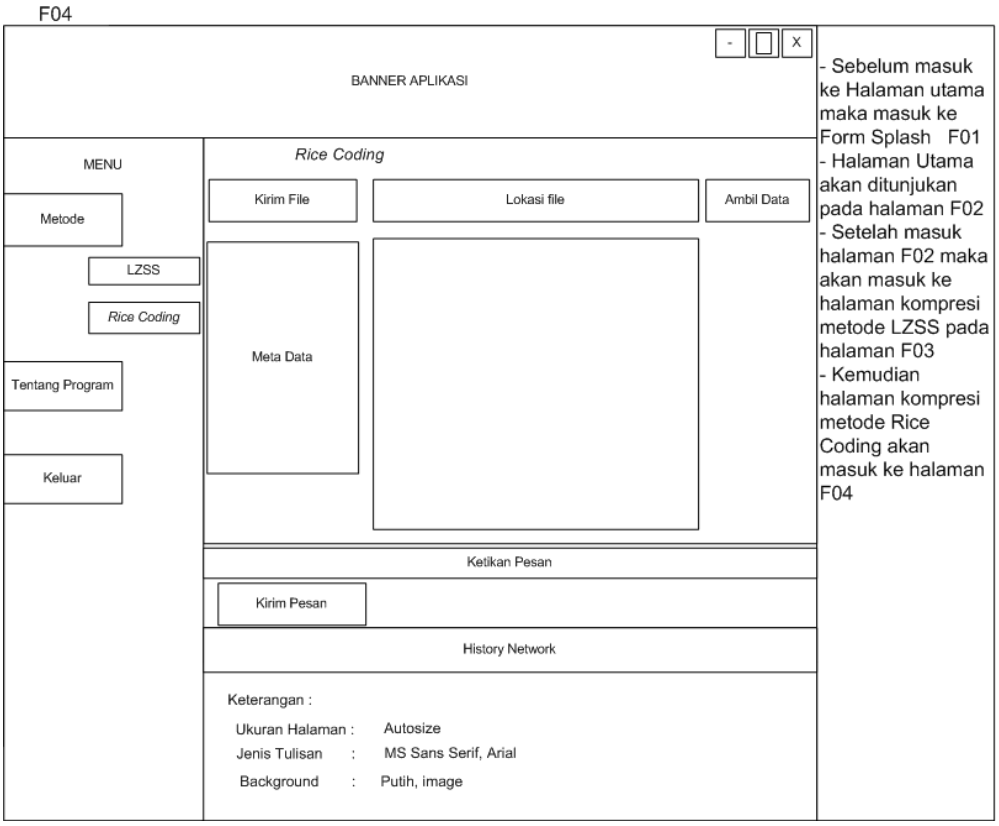
Gambar 3.26 Antarmuka Menu Utama

3. Perancangan antarmuka Menu Dekompresi metode LZSS



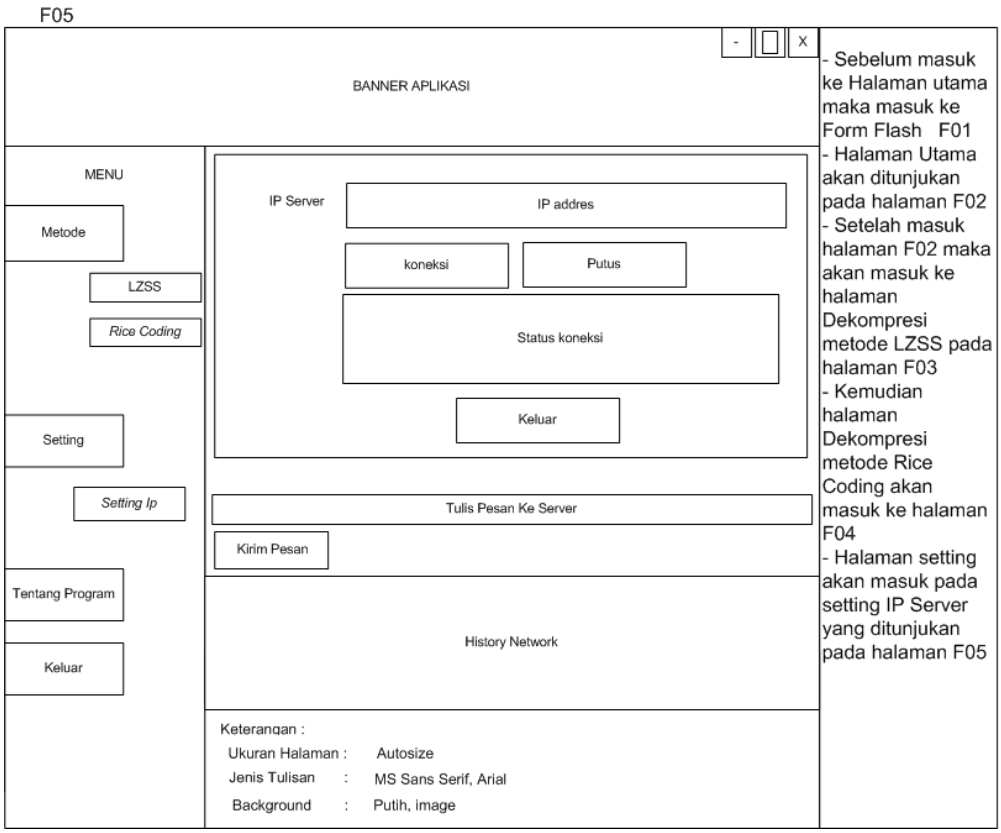
Gambar 3.27 Antarmuka Menu Dekompresi metode LZSS

4. Perancangan antarmuka Menu Dekompresi Metode *Rice Coding*



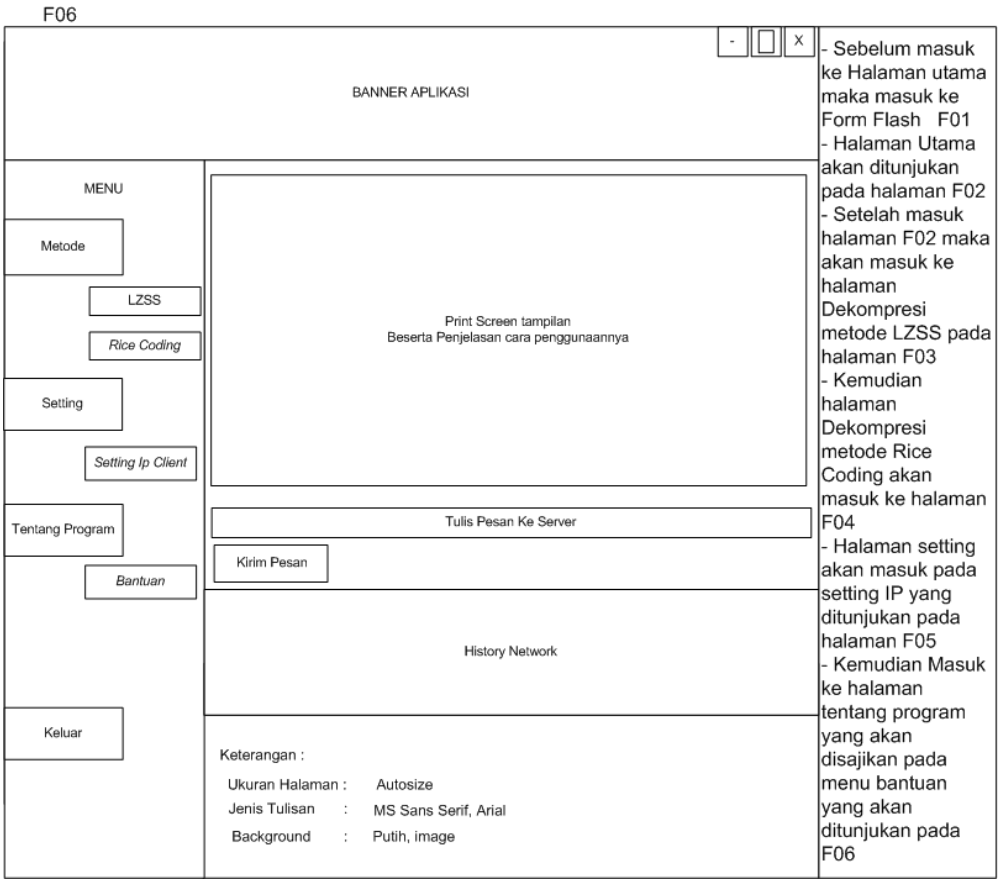
Gambar 3.28 Antarmuka menu Dekompresi Metode *Rice Coding*

5. Perancangan Antarmuka Menu *Setting* IP ke *Server*



Gambar 3.29 Antarmuka Menu *Setting* IP *Address* ke *Server*

6. Perancangan Antarmuka Menu Bantuan



Gambar 3.30 Antarmuka Menu Bantuan

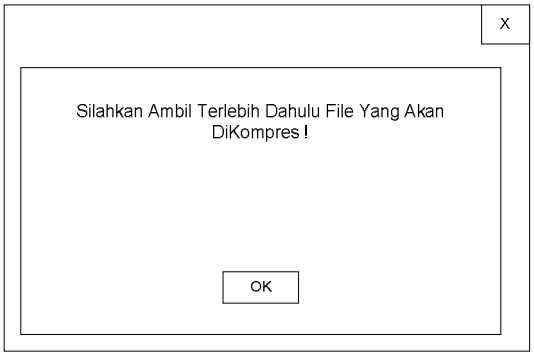
7. Perancangan Antarmuka Pesan



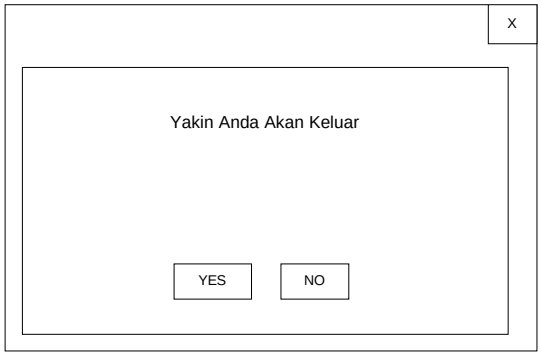
Gambar 3.31 Pesan Belum Mengisi IP *Address*



Gambar 3.32 Pesan *Client* Tidak Ditemukan



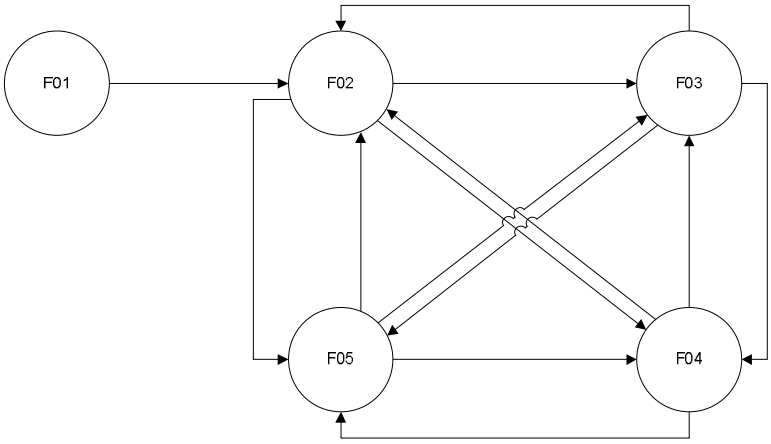
Gambar 3.33 Pesan Data Belum di Isi



Gambar 3.34 Pesan Keluar Aplikasi

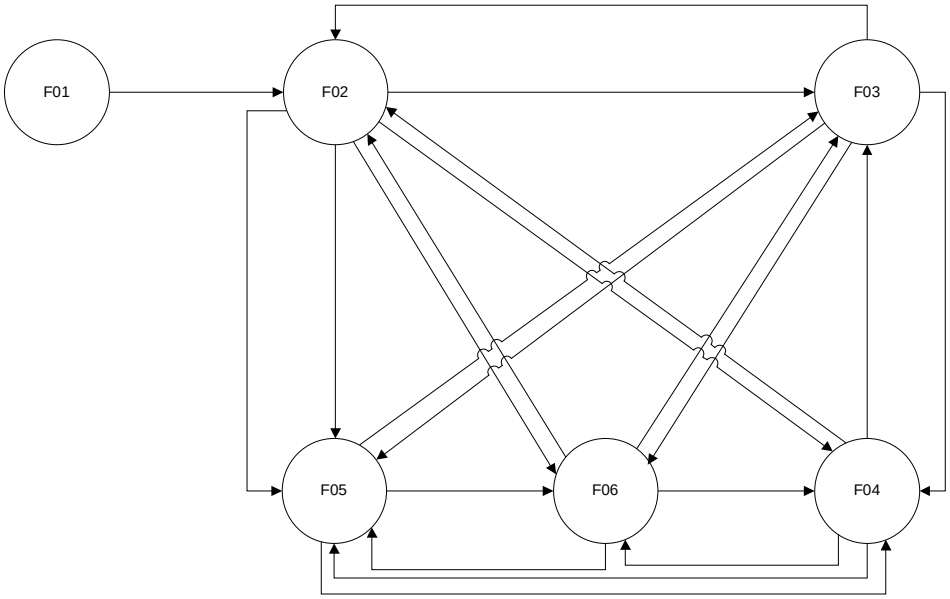
3.18 Jaringan Semantik

1. Struktur Jaringan Semantik *Server*



Gambar 3.35 Jaringan Semantik *Server*

2. Struktur Jaringan Semantik *Client*



Gambar 3.36 Jaringan Semantik