Custom low-latency image compression

Jean-Baptiste Note
Ecole Normale Supérieure, 45 rue d'Ulm, 75005 Paris France.

January 28, 2005

Abstract

used in Sepia ?

1 Theory

(insert here Jean's document. It's on my harddrive at home, and i'll need a week-end to get access to it unfortunately).

2 Code

(Used to show common source)

2.1 standard, direct version

```
//fun SerialReg(n)(x) = SR(x,C(-1),n);

block SerialReg(n)(x) = k {
  k = (n == 0) ? x : Z(SerialReg(n-1)(x));
}

block Ze(s,e) = (ze) {  // Register with enable
  ze = Z(cond(-e,s,ze));
}

var blockSize=16;

block Coder(p,e) = (bc) { // Lossless Signal Coder
// PROBE  @Probe(bc);
  (tr, di, df) := Transform(p,e);
  (d0, e0)  := PixelCoder(tr, di, df, e);
  bc  := BlockCoder(d0,e0);
```

```
  fun dupp(s) = C(1)| shift(s,1);  // s->1+2s

  /* @starttex SerialReg.jzz */

  // Generic Pixel Transform
  block Transform(px, ep) = (tr, di, df) {
// PROBE    @Probe(di);
    df := px-Ze(px,ep);         // Differential signal
    di := lt(C(11), abs(df));   // Decompress indicator
    tr := dupp(px);             // Transformed value
  }

  /* @endtex SerialReg.jzz */

  /* @starttex PixelCoder.mzz */

  block PixelCoder(tr, di, df, e) = (e*v9, e*l9) {  // Pixel Code
    l9 := cond(di, C(9), l8);
    v9 := cond(di, tr, v8);  // Decompressed pixel
    ta := df & C(31);
    v8 := lut(ta, LUTv8);   // LUT(5b)->8b
    l8 := lut(ta, LUTl8);   // LUT(5b)->4b
    // Code LUT: values
    LUTv8 = [n-> (n<16) ? eliasC(n) : (n<32) ? eliasC(n-32) : 0];
    // Code LUT: lengths
    LUTl8 = [n-> (n<16) ? eliasL(n) : (n<32) ? eliasL(n-32) : 0];
    // Elias coding fills both LUTv8 and LUTl8
    fun eliasL(n) = cl { (cn, cl) = elias(n); }
    fun eliasC(n) = cn { (cn, cl) = elias(n); }
    fun elias(n:int) = vl {
      s = n<0 ? 1 : 0;
      a = (n<0) ? -n : n;
      vl = (a>11) ? (0, 0) : (n==0) ? (0, 2) : (2+4*s+8*ca, la+3);
      (ca, la) = (a==1) ? (0,1) : (a<4) ? (1+4*(a&1),3) : (3+4*(a&7),5);
    }
  }

  /* @endtex PixelCoder.mzz */

  /* @starttex BlockCoder.mzz */

  fun BlockCoder(d, e) = (bc) { // (bc, rl, rv) {
    var castrv= [0,2**15-1];
    // Control
    nl := rl + e;          // Adder(4b,4b)->5b
    rl := Z(nl & C(15));   // Reg(4b)
```

```
    nb := shift(nl,-4);     // extract 1 bit
    // Data
    nv := rv | lshift(d, rl);   // Shift-up  24b shift(4b,9b)+or(9b)
    rv := cast(Z(cond(-nb, shift(nv,-blockSize), nv)), castrv); // reg(15b) + mux(1b,15b,15
    bc := nb*dupp(nv & C(2**blockSize-1)); // 17b
  }

  /* @endtex BlockCoder.mzz */

  // Recursive Block Coder
  //  block BlockCoder(d0,e0) = (bc) {
  //// PROBE    @Probe(e1);
  //    (d1, e1) := Stage(0)(d0,e0);
  //    (d2, e2) := Stage(1)(d1,e1);
  //    (d3, e3) := Stage(2)(d2,e2);
  //    (d4, e4) := Stage(3)(d3,e3);
  //    bc = e4*dupp(d4);
  //
  //    block Stage(k':int)(di, ei) = (do, eo) {   // di(n) has ei(n)*k bits
  //      // Control is a Modulo 2 counter
  //      k = 2**k';
  //      se = ei+re;
  //      re = Z(se & C(1));
  //      eo = shift(se, -1);  // enable output
  //      // Pixel operations
  //      //ds = lshift(di, re*C(k));   // shift-up by k*re(n)
  //      ds = cond(-re,shift(di,k),di);   // shift-up by k*re(n)
  //      dr = ds | rd;               // merge with remaining pixel
  //      do = dr & (lshift(C(1),a) - C(1)); // output the low a bits of dr
  //      rd = Z(rshift(dr,a));    // keep the high a(n) bits
  //      a = shift(eo,k'+1);
  //      //a = eo*C(2*k);
  //    }
  //  }
}

block Decoder(bc) =  (px, e) {
  (c16, ei) = (shift(bc,-1), bc & C(1));
  (px) := deTransform(oc, e);
  (oc, e) := blockDecoder(c16, ei);

  block deTransform(t, e) = (po) {
    /* THIS HAS BEEN MODIFIED ! */
    var castpo = [0,255];
    t1 = t & C(1);
    t2 = shift(t,-1);
```

```
    tz = t2+Ze(po, e);
//    @Cast(po,castpo);
    //@jprobe(".po");
//    po := cond(-t1,t2,tz);
    po := cast(cond(-t1,t2,tz),castpo);
  }

  block blockDecoder(inCode, inEn) = (outCode, outEn) {
    // Decoder buffer requires 81 bits!
    // This cannot be derived from the Decoder program,
    // since it is a property of the whole Codec.
    var maxol = 56+16; var maxnl = maxol+9;
    var castnl = [0,maxnl]; var castol = [0, maxol];
    var castnc= [0,2**maxnl-1]; var castoc= [0,2**maxol-1];
    (outCode, l) := ValLength(newCode);
    newLength = cast(oldLength + inEn*C(blockSize),  castnl);
    oldLength = cast(Z(newLength-le), castol);

    newCode = cast(oldCode | lshift(inCode, oldLength), castnc);
    oldCode = cast(Z(rshift(newCode,le)), castoc);

    outEn =  C(1)+lt(newLength,l);
    le = l*outEn;

    block ValLength(code) = (val, len) {
      c0 = code & C(1);       // c0=1 <=> decompressed to 9 bits
      c = shift(code, -1);
      val = cond(-c0, code & C(2**9-1), v8);
      len = cond(-c0, C(9), l8);
      al = c & C(15);
      l8 =  lut(al, LUTl8);             // LUT(4b)->4b
      LUTl8 =  [n->deliasL(2*n)];
      sgn = lut(c & C(3), [0,1,0,-1]);
      v8 = sgn * lut(shift(c,-2) & C(2**6-1), LUTv8);
      LUTv8 =  [n->deliasC(2*(1+4*n))];

      // Decoding Elias fills LUTv8 and LUT l8
      fun deliasL(n) =dl { (dl,dn) = delias(n); }
      fun deliasC(n) =dn { (dl,dn) = delias(n); }
      fun delias(n) = (n0==1) ? (9,n.slice(0,8)) :
      (n1==0) ? (2,0) : (n3==0) ? (4,v4) : (n4==0) ? (6,v6) : (8,v8) {
n0 = n.bit(0);  n1 = n.bit(1);
n2 = n.bit(2);  n3 = n.bit(3);
n4 = n.bit(4); // n5 = n.bit(5);
v4 = 2*(1-2*n2);
v6 = v4*(2+n.bit(5));
```

```
v8 = v4*(4+n.slice(5,7)+4*(1-n.bit(7)));
      }
    }
  }
}

block Codec(px, pe) = (qx, qe) {
  (bc1) := Coder(px, pe);
  (qx,qe) := Decoder(bc1);
  @jrename("(bc1).bc","bc");
}
fun testCodec() = jzz0/* format("\n Test the lossless Codec"
 "\n Input \n%a\n%a"
 "\n Internals \n%a\n%a\n%a\n%a\n%a"
 "\n Output \n%a\n%a",
 px, pe,
 d0, e0, d1, e0, bc,
 qx, qe)*/
{
  /* to be automated */
  //@Probe((qx,qe).(bc1).bc);
  dynamic (Streams.prinLength = 512) {

    //design = Minoux.make("CompressDirect",15,2,[qx,qe],0,[]);
    design = Minoux.make("CompressDirect",15,1,[bc1],0,[]);
    jzz0 = design.netzz();
    (cpamdc0,hpamdc0,tpamdc0) = design.pamdc();

    //@Minoux.dumpfile("jazzout.cc",cpamdc0);
    //@Minoux.dumpfile("jazzout.hh",hpamdc0);
    //@Minoux.dumpfile("jazzref.cc",tpamdc0);
  }

  (qx, qe) := Codec(px,pe);
  bc1 := Coder(px,pe);
  pe :=  Input([0,1],inE);
  px :=  Input([0,255],inP);

  var  inP = [22, 12,  12, 12, 12, 12, 12, 12, 12,
0, 15,  30, 45 , 60, 75, 90, 105, 120, 135, 150, 165, 180, 195, n-> Integer.random(16)];
  var inE = [n-> 1];

  //  bc = Probe((qx,qe).bc);
  //  d1 = Probe((qx,qe).(bc1).bc.d1);
  //  d0 = Probe((qx,qe).(bc1).bc.d2);
  //
```

```
  //  e0 = Probe((qx,qe).(bc1).e0);
  //  e1 = Probe((qx,qe).(bc1).e1);
}

@print("\n\t\tCodec on streams of pixels\n%s\n",  testCodec());
```

2.1.1 Pixel coder block

```
  block PixelCoder(tr, di, df, e) = (e*v9, e*l9) {  // Pixel Code
    l9 := cond(di, C(9), l8);
    v9 := cond(di, tr, v8);  // Decompressed pixel
    ta := df & C(31);
    v8 := lut(ta, LUTv8);   // LUT(5b)->8b
    l8 := lut(ta, LUTl8);   // LUT(5b)->4b
    // Code LUT: values
    LUTv8 = [n-> (n<16) ? eliasC(n) : (n<32) ? eliasC(n-32) : 0];
    // Code LUT: lengths
    LUTl8 = [n-> (n<16) ? eliasL(n) : (n<32) ? eliasL(n-32) : 0];
    // Elias coding fills both LUTv8 and LUTl8
    fun eliasL(n) = cl { (cn, cl) = elias(n); }
    fun eliasC(n) = cn { (cn, cl) = elias(n); }
    fun elias(n:int) = vl {
      s = n<0 ? 1 : 0;
      a = (n<0) ? -n : n;
      vl = (a>11) ? (0, 0) : (n==0) ? (0, 2) : (2+4*s+8*ca, la+3);
      (ca, la) = (a==1) ? (0,1) : (a<4) ? (1+4*(a&1),3) : (3+4*(a&7),5);
    }
  }
```

2.1.2 Block coder block

```
  fun BlockCoder(d, e) = (bc) { // (bc, rl, rv) {
    var castrv= [0,2**15-1];
    // Control
    nl := rl + e;           // Adder(4b,4b)->5b
    rl := Z(nl & C(15));    // Reg(4b)
    nb := shift(nl,-4);      // extract 1 bit
    // Data
    nv := rv | lshift(d, rl);   // Shift-up  24b shift(4b,9b)+or(9b)
    rv := cast(Z(cond(-nb, shift(nv,-blockSize), nv)), castrv); // reg(15b) + mux(1b,15b,15
```

```
    bc := nb*dupp(nv & C(2**blockSize-1)); // 17b
  }
```

2.2 standard, direct version, retimed at 4 ns

```
fun SerialReg(n)(x) = SR(x,C(-1),n);
fun Zr(i) = Z(i); // retiming register

var blockSize=16;

block Ze(s,e) = (ze) {  // Register with enable
  ze = Z(cond(-e,s,ze));
}

block Coder(p',e') = (bc) { // Lossless Signal Coder
// PROBE  @Probe(bc);
  p := Zr(Zr(p'));
  e := Zr(Zr(e'));
  (tr, di, ta) := Transform(p,e);
  (d0, e0)     := PixelCoder(tr, di, ta, Zr(Zr(Zr(e))));
  bc           := BlockCoder(d0,e0);
  fun dupp(s) = C(1)| shift(s,1);  // s->1+2s

  // Generic Pixel Transform
  block Transform(px, ep) = (Zr(tr),Zr(di),Zr(ta)) {
// PROBE    @Probe(di);
    df' = px-Ze(px,ep);// Differential signal
    df := Zr(df');
    ta := Zr(df) & C(31);
    di := lt(C(11), Zr(abs(df)));   // Decompress indicator
    tr := Zr(dupp(Zr(px)));             // Transformed value
  }

  block PixelCoder(tr, di', df, e') = (Zr(cond(-e,v9,C(0))), Zr(cond(-e,l9,C(0)))) {  // Pi
    e := Zr(e');
    di := Zr(di');
    l9 := cond(di, C(9), Zr(l8));
    v9 := cond(di, Zr(tr), Zr(v8));  // Decompressed pixel

    v8 := lut(ta, LUTv8);   // LUT(5b)->8b
    l8 := lut(ta, LUTl8);   // LUT(5b)->4b
    // Code LUT: values
    LUTv8 = [n-> (n<16) ? eliasC(n) : (n<32) ? eliasC(n-32) : 0];
    // Code LUT: lengths
```

```
  LUT18 = [n-> (n<16) ? eliasL(n) : (n<32) ? eliasL(n-32) : 0];
  // Elias coding fills both LUTv8 and LUT18
  fun eliasL(n) = cl { (cn, cl) = elias(n); }
  fun eliasC(n) = cn { (cn, cl) = elias(n); }
  fun elias(n:int) = vl {
    s = n<0 ? 1 : 0;
    a = (n<0) ? -n : n;
    vl = (a>11) ? (0, 0) : (n==0) ? (0, 2) : (2+4*s+8*ca, la+3);
    (ca, la) = (a==1) ? (0,1) : (a<4) ? (1+4*(a&1),3) : (3+4*(a&7),5);
  }
}

fun BlockCoder(d, e) = (Zr(bc)) { // (bc, rl, rv) {
  var castrv= [0,2**15-1];
  // Control
  nl := rl + e;          // Adder(4b,4b)->5b
  rl := Z(nl & C(15));   // Reg(4b)
  @Probe(nb);
  nb := shift(Zr(nl),-4);     // extract 1 bit
  // Data
  nv := rv | Zr(lshift(d, rl));   // Shift-up  24b shift(4b,9b)+or(9b)
  rv := cast(Z(cond(-nb, shift(nv,-blockSize), nv)), castrv); // reg(15b) + mux(1b,15b,15
  bc := cond(Zr(-nb),dupp(Zr(nv) & C(2**blockSize-1)),C(0)); // 17b
}

// Recursive Block Coder
//  block BlockCoder(d0,e0) = (Zr(bc)) {
//// PROBE    @Probe(e1);
//    (d1, e1) := Stage(0)(d0,e0);
//    (d2, e2) := Stage(1)(d1,e1);
//    (d3, e3) := Stage(2)(d2,e2);
//    (d4, e4) := Stage(3)(d3,e3);
//    // e4 is one bit, so mux should be ok there
//    //bc = e4*dupp(d4);
//    bc := cond(-e4,dupp(d4),C(0));
//
//    block Stage(k':int)(di, ei) = (Zr(do), Zr(Zr(eo))) {   // di(n) has ei(n)*k bits
//      // Control is a Modulo 2 counter
//      k = 2**k';
//      se := ei+re;
//      re := Z(se & C(1));
//      eo := Zr(shift(se, -1));  // enable output
//      // Pixel operations
//      ds := Zr(cond(-re,shift(di,k),di));
//      //ds' := lshift(di, re*C(k));   // shift-up by k*re(n)
//      dr := ds | rd;              // merge with remaining pixel
```

```
  //      do := Zr(dr) & mask; // output the low a(n) bits
  //      mask := Zr(lshift(C(1),a)) - C(1);
  //      rd := Z(rshift(dr,a));    // keep the high a(n) bits
  //      a := shift(eo,k'+1);
  //      //a := cond(-eo,C(2*k),C(0));
  //    }
  //  }
}

block Decoder(bc) =  (px, e) {
  (c16, ei) = (shift(bc,-1), bc & C(1));
  (px) := deTransform(oc, e);
  (oc, e) := blockDecoder(Zr(c16), Zr(ei));

  block deTransform(t, e) = (po) {
    /* THIS HAS BEEN MODIFIED ! */
    var castpo = [0,255];
    t1 = t & C(1);
    t2 = shift(t,-1);
    tz = t2+Ze(po, e);
    @Cast(po,castpo);
    //@jprobe(".po");
    po := cond(-t1,t2,tz);
//    po := Cast(cond(-t1,t2,tz),castpo);
  }

  block blockDecoder(inCode, inEn) = (outCode, outEn) {
    // Decoder buffer requires 81 bits!
    // This cannot be derived from the Decoder program,
    // since it is a property of the whole Codec.
    var maxol = 56+16; var maxnl = maxol+9;
    var castnl = [0,maxnl]; var castol = [0, maxol];
    var castnc= [0,2**maxnl-1]; var castoc= [0,2**maxol-1];
    var blockSize = 16;
    (outCode, l) := ValLength(newCode);
    newLength = cast(oldLength + inEn*C(blockSize),  castnl);
    oldLength = cast(Z(newLength-le), castol);

    newCode = cast(oldCode | lshift(inCode, oldLength), castnc);
    oldCode = cast(Z(rshift(newCode,le)), castoc);

    outEn =  C(1)+lt(newLength,l);
    le = l*outEn;

    block ValLength(code) = (val, len) {
      c0 = code & C(1);       // c0=1 <=> decompressed to 9 bits
```

```
      c = shift(code, -1);
      val = cond(-c0, code & C(2**9-1), v8);
      len = cond(-c0, C(9), l8);
      al = c & C(15);
      l8 =  lut(al, LUTl8);             // LUT(4b)->4b
      LUTl8 =  [n->deliasL(2*n)];
      sgn = lut(c & C(3), [0,1,0,-1]);
      v8 = sgn * lut(shift(c,-2) & C(2**6-1), LUTv8);
      LUTv8 =  [n->deliasC(2*(1+4*n))];

      // Decoding Elias fills LUTv8 and LUT l8
      fun deliasL(n) =dl { (dl,dn) = delias(n); }
      fun deliasC(n) =dn { (dl,dn) = delias(n); }
      fun delias(n) = (n0==1) ? (9,n.slice(0,8)) :
      (n1==0) ? (2,0) : (n3==0) ? (4,v4) : (n4==0) ? (6,v6) : (8,v8) {
n0 = n.bit(0);  n1 = n.bit(1);
n2 = n.bit(2);  n3 = n.bit(3);
n4 = n.bit(4); // n5 = n.bit(5);
v4 = 2*(1-2*n2);
v6 = v4*(2+n.bit(5));
v8 = v4*(4+n.slice(5,7)+4*(1-n.bit(7)));
      }
    }
  }
}

block Codec(px, pe) = (qx, qe) {
  (bc1) := Coder(px, pe);
  (qx,qe) := Decoder(bc1);
  @jrename("(bc1).bc","bc");
}
fun testCodec() = jzz0/* format("\n Test the lossless Codec"
 "\n Input \n%a\n%a"
 "\n Internals \n%a\n%a\n%a\n%a\n%a"
 "\n Output \n%a\n%a",
 px, pe,
 d0, e0, d1, e0, bc,
 qx, qe)*/
{
  /* to be automated */
  //@Probe((qx,qe).(bc1).bc);

  dynamic (Streams.prinLength = 512) {
    //((cpamdc0,hpamdc0,tpamdc0),jzz0,(vcd0,cfg0)) = ( Minoux.compile(1,[bc1],0,[]) );
    //((cpamdc0,hpamdc0,tpamdc0),jzz0,(vcd0,cfg0)) = ( Minoux.compile(2,[qx,qe],0,[]) );
```

```
    design = Minoux.make("CompressDirect4ns",10.0,15,1,[bc1],0,[]);
    jzz0 = design.netzz();
    /* dumps the design in pamdc form */
    @design.pamdc();
    /* dumps the calculus done on the design */
    @design.iosample();
  }

  (qx, qe) := Codec(px,pe);
  bc1 := Coder(px,pe);
  pe :=  Input([0,1],inE);
  px :=  Input([0,255],inP);

  var  inP = [22, 12,  12, 12, 12, 12, 12, 12, 12,
0, 15,  30, 45 , 60, 75, 90, 105, 120, 135, 150, 165, 180, 195, n-> Integer.random(16)];
  var inE = [n-> 1];

  //  bc = Probe((qx,qe).bc);
  //  d1 = Probe((qx,qe).(bc1).bc.d1);
  //  d0 = Probe((qx,qe).(bc1).bc.d2);
  //
  //  e0 = Probe((qx,qe).(bc1).e0);
  //  e1 = Probe((qx,qe).(bc1).e1);
}

@print("\n\t\tCodec on streams of pixels\n%s\n",  testCodec());
```

2.3 recursive version

We only change a few blocks to get to this version.

2.3.1 Recursive Block Coder block

```
  // Recursive Block Coder
  block BlockCoder(d0,e0) = (bc) {
// PROBE    @Probe(e1);
    (d1, e1) := Stage(0)(d0,e0);
    (d2, e2) := Stage(1)(d1,e1);
    (d3, e3) := Stage(2)(d2,e2);
    (d4, e4) := Stage(3)(d3,e3);
    bc = e4*dupp(d4);

    block Stage(k':int)(di, ei) = (do, eo) {   // di(n) has ei(n)*k bits
      // Control is a Modulo 2 counter
```

```
      k = 2**k';
      se := ei+re;
      re := Z(se & C(1));
      eo := shift(se, -1);  // enable output
      // Pixel operations
      //ds = lshift(di, re*C(k));   // shift-up by k*re(n)
      ds := cond(-re,shift(di,k),di);   // shift-up by k*re(n)
      dr := ds | rd;              // merge with remaining pixel
      do := dr & (lshift(C(1),a) - C(1)); // output the low a bits of dr
      rd := Z(rshift(dr,a));    // keep the high a(n) bits
      a := shift(eo,k'+1);
      //a = eo*C(2*k);
    }
  }
}
```