**Instituto Politécnico Nacional  
 Escuela Superior de Cómputo  
Ingeniería en   
Sistemas Computacionales**

**Unidad de Aprendizaje:**

**VISIÓN POR COMPUTADORA**

**Grupo: 7CV6**

**Práctica 4: Transformaciones logicas**

**Alumno:**

**Bautista Ríos Alfredo**

Docente: Saul De la O Torres

**Introducción**

Las operaciones lógicas entre imágenes binarias son una forma directa de construir máscaras: AND para intersección, OR para unión y XOR para detectar diferencias. Además, las operaciones relacionales permiten comparar dos imágenes píxel a píxel y generar una máscara donde se cumple una condición. En esta práctica se implementa una interfaz en GTK3 que carga dos imágenes (A y B), opcionalmente fuerza su binarización (umbral 128), y aplica operaciones lógicas (AND/OR/XOR con NOT A/NOT B) o relacionales (==, !=, >, >=, <, <=), produciendo una salida estrictamente binaria.

**Objetivos**

* Crear la interfaz necesaria para cargar dos imágenes (A y B) y visualizar el resultado.
* Implementar AND, OR, XOR y las variantes NOT A / NOT B.
* Implementar ==, !=, >, >=, <, <= entre A y B (imágenes binarias).
* Asegurar que el resultado sea binario (0/255) y validar el comportamiento con casos simples.
* Permitir guardar el resultado como archivo PNG.

**Marco teórico**

**Imágenes binarias y umbralización**

Una imagen binaria solo contiene dos estados (0/1). En representación de 8 bits se usan 0 y 255. Para forzar binarización se utiliza un umbral fijo T=128: si I(x,y)≥T → 255, en otro caso → 0.

**Operaciones lógicas (por píxel)**

Para A,B∈{0,1}: AND=1 solo si A=1 y B=1; OR=1 si A=1 o B=1; XOR=1 si A≠B. NOT invierte el bit (0↔1) antes de aplicar el operador principal.

**Operaciones relacionales (por píxel)**

Las comparaciones (==, !=, >, >=, <, <=) regresan una máscara binaria: R(x,y)=1 si la relación se cumple entre A(x,y) y B(x,y), en otro caso R(x,y)=0. Si A y B ya están binarizadas (0/255), entonces por ejemplo A>B equivale a “A=255 y B=0”.

**Metodología y desarrollo**

**Arquitectura del proyecto (MVC)**

• Vista: MainWindow (GTK Builder + Glade) crea la interfaz y expone widgets al controlador.  
• Controlador: AppController conecta callbacks (botones) y decide qué operación ejecutar.  
• Modelo: ImageModel gestiona carga, binarización/normalización, reescalado y produce el resultado.  
• Núcleo de operaciones: BinaryOps implementa AND/OR/XOR y comparaciones relacionales píxel a píxel.

**Interfaz (Glade)**

* Cargar A / Cargar B: selecciona imágenes de entrada.
* Forzar binarización (umbral 128): asegura entradas binarias aunque el archivo sea escala de grises o RGB.
* Selección de operación: radio-buttons para AND/OR/XOR y para ==, !=, >, >=, <, <=.
* NOT A / NOT B: invierte A o B antes de la operación lógica.
* Aplicar: calcula y muestra el Resultado. Guardar: exporta resultado.png (PNG).

**Preprocesamiento aplicado por el modelo**

1) Conversión a gris (luma): cuando la entrada es RGB, se calcula luminancia y se replica en R=G=B para tratarla como gris.  
2) Binarización: umbral 128 → {0,255}.  
3) Alineación de tamaño: si A y B tienen distinta resolución, se reescala B al tamaño de A con INTERP\_NEAREST (vecino más cercano) para no crear valores intermedios.

**Compilación y ejecución (Linux)**

Dependencias: gtkmm-3.0 (pkg-config).  
Comandos:  
 mkdir -p build  
 cmake -S . -B build -DCMAKE\_BUILD\_TYPE=Release  
 cmake --build build -j  
 ./build/imlogic

CMake copia assets/ al build/ para que el ejecutable encuentre assets/ui.glade.

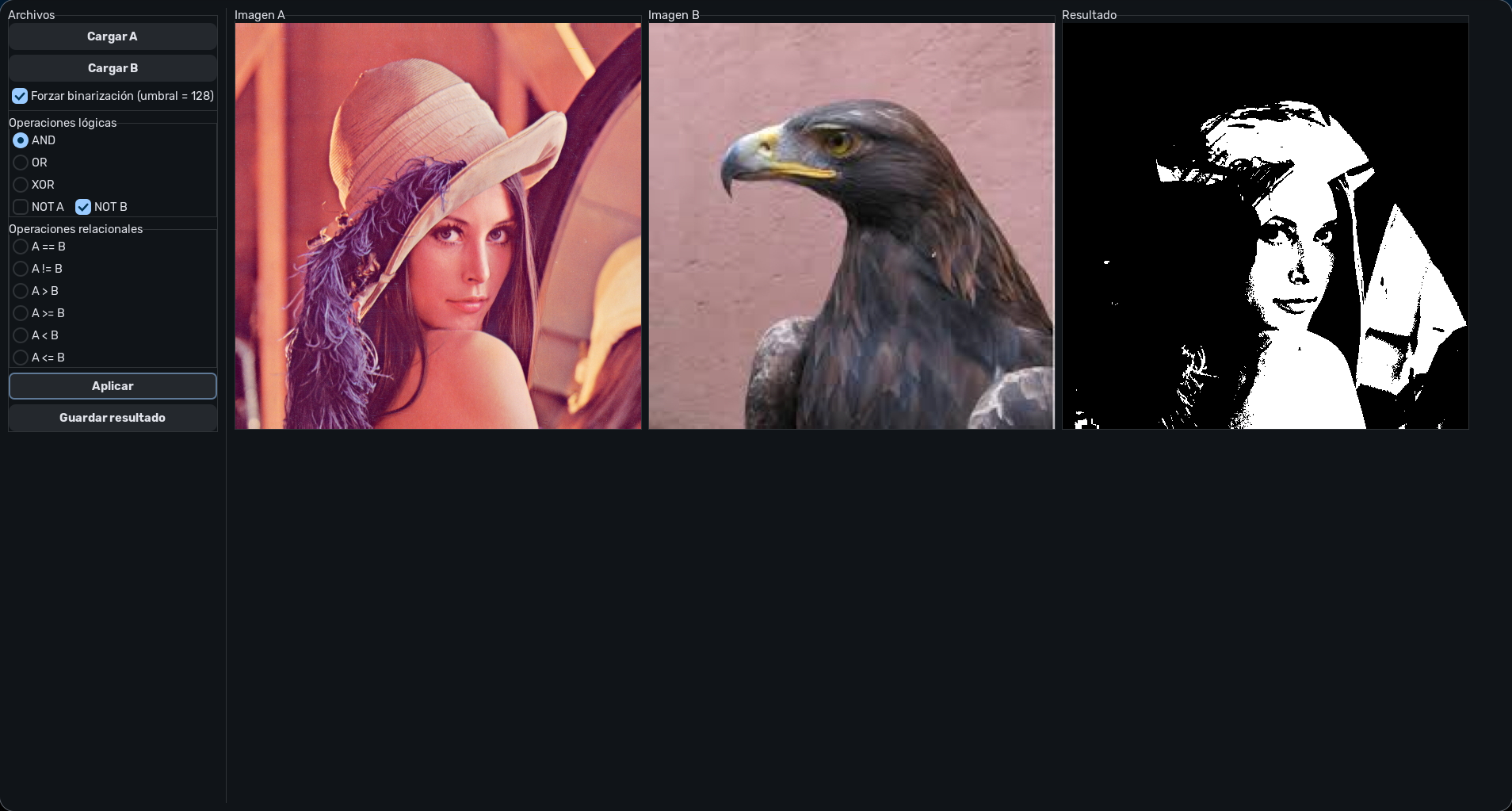
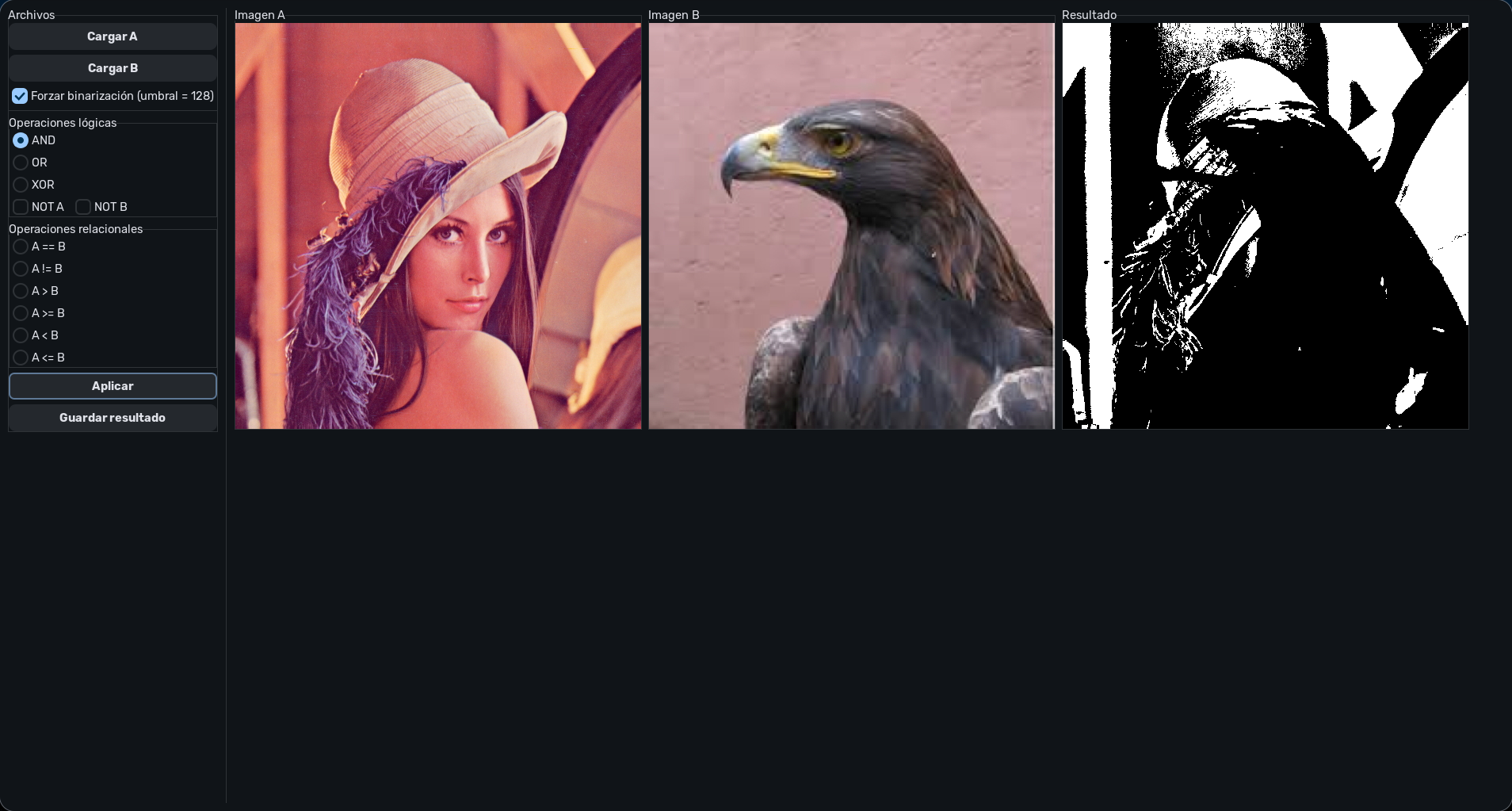
**Resultados y validación**

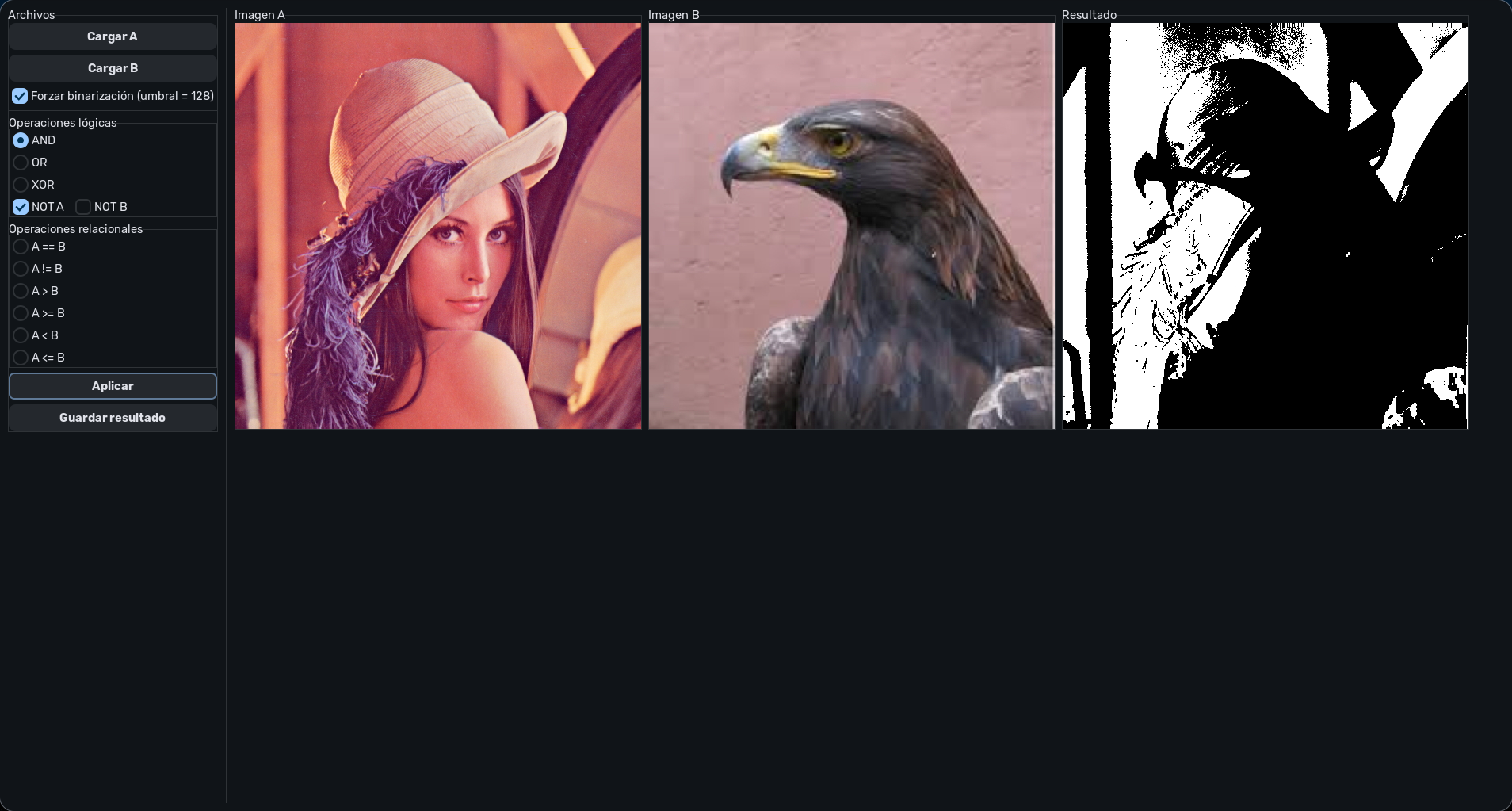
Criterios de validación recomendados:

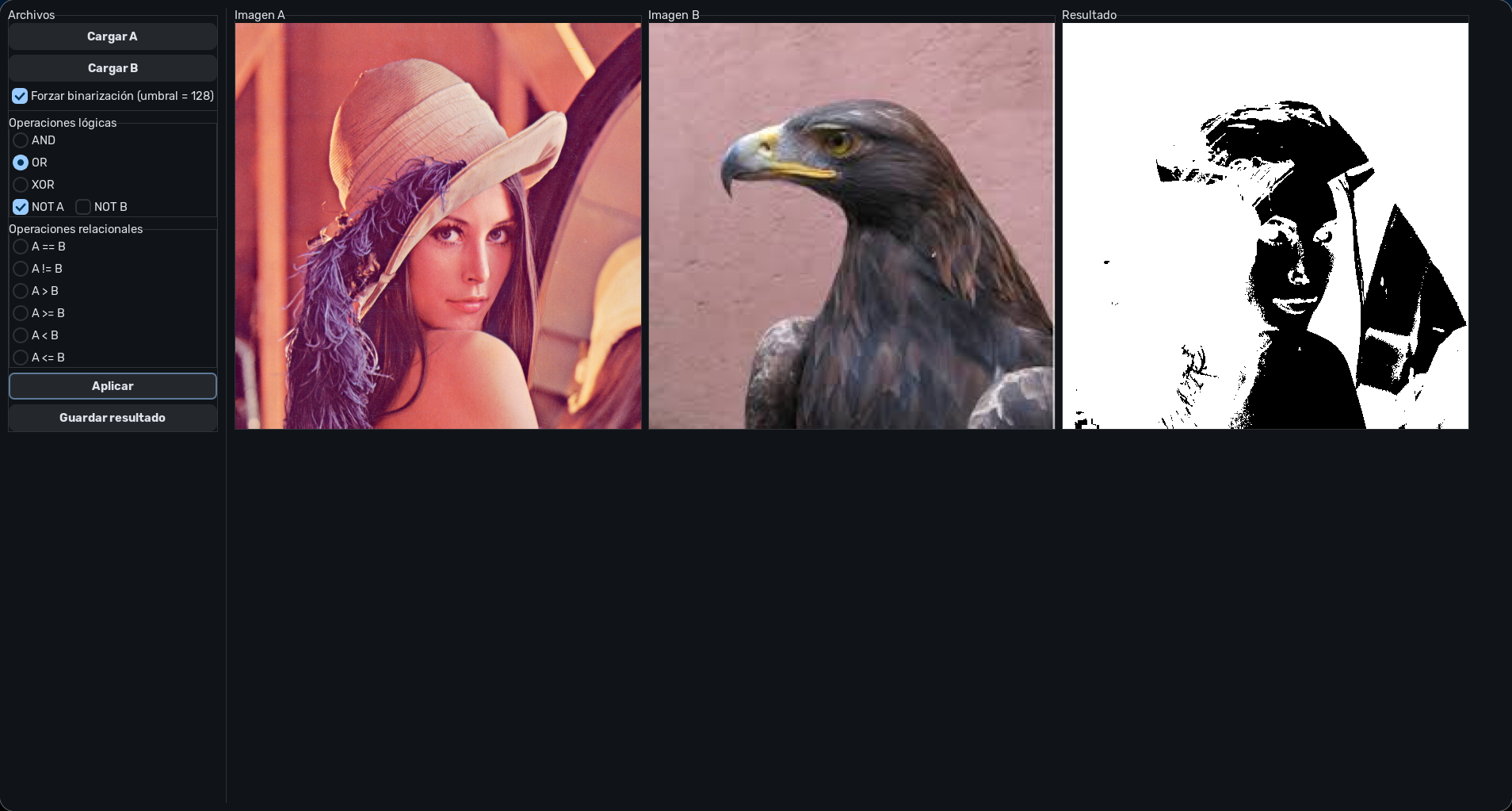
* Salida binaria: el Resultado debe contener únicamente 0 o 255.
* AND: blanco solo en la intersección de blancos (A=255 y B=255).
* OR: blanco en la unión de blancos (A=255 o B=255).
* XOR: blanco donde difieren (A≠B).
* NOT A / NOT B: al activar, se debe invertir la máscara correspondiente antes de AND/OR/XOR.
* Relacionales: == marca coincidencias; != marca diferencias; >,>=,<,<= se interpretan como comparaciones entre 0 y 255.
* Reescalado: con entradas de distinto tamaño, el resultado coincide con el tamaño de A y mantiene valores binarios.

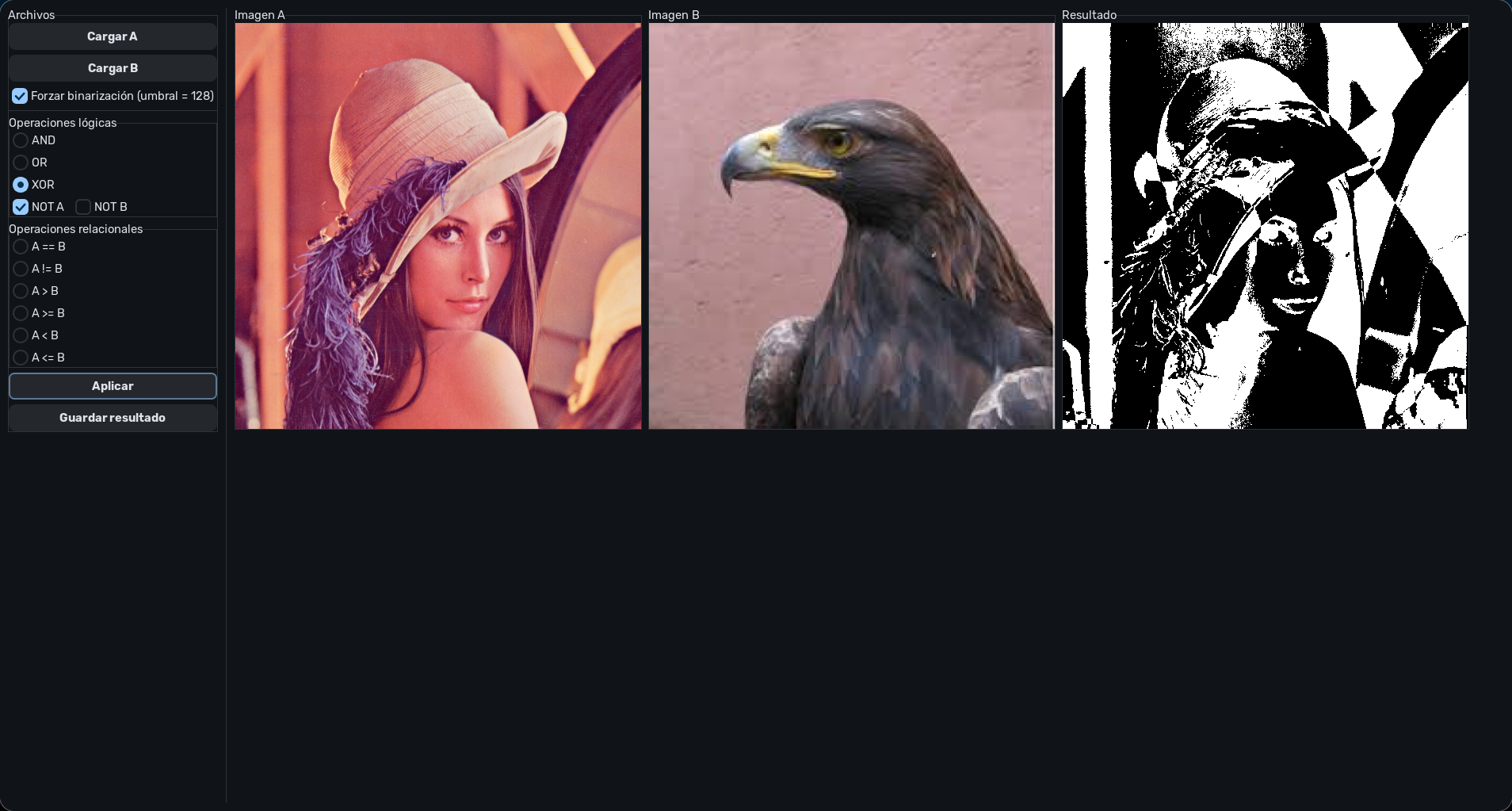
**Evidencia de salida**

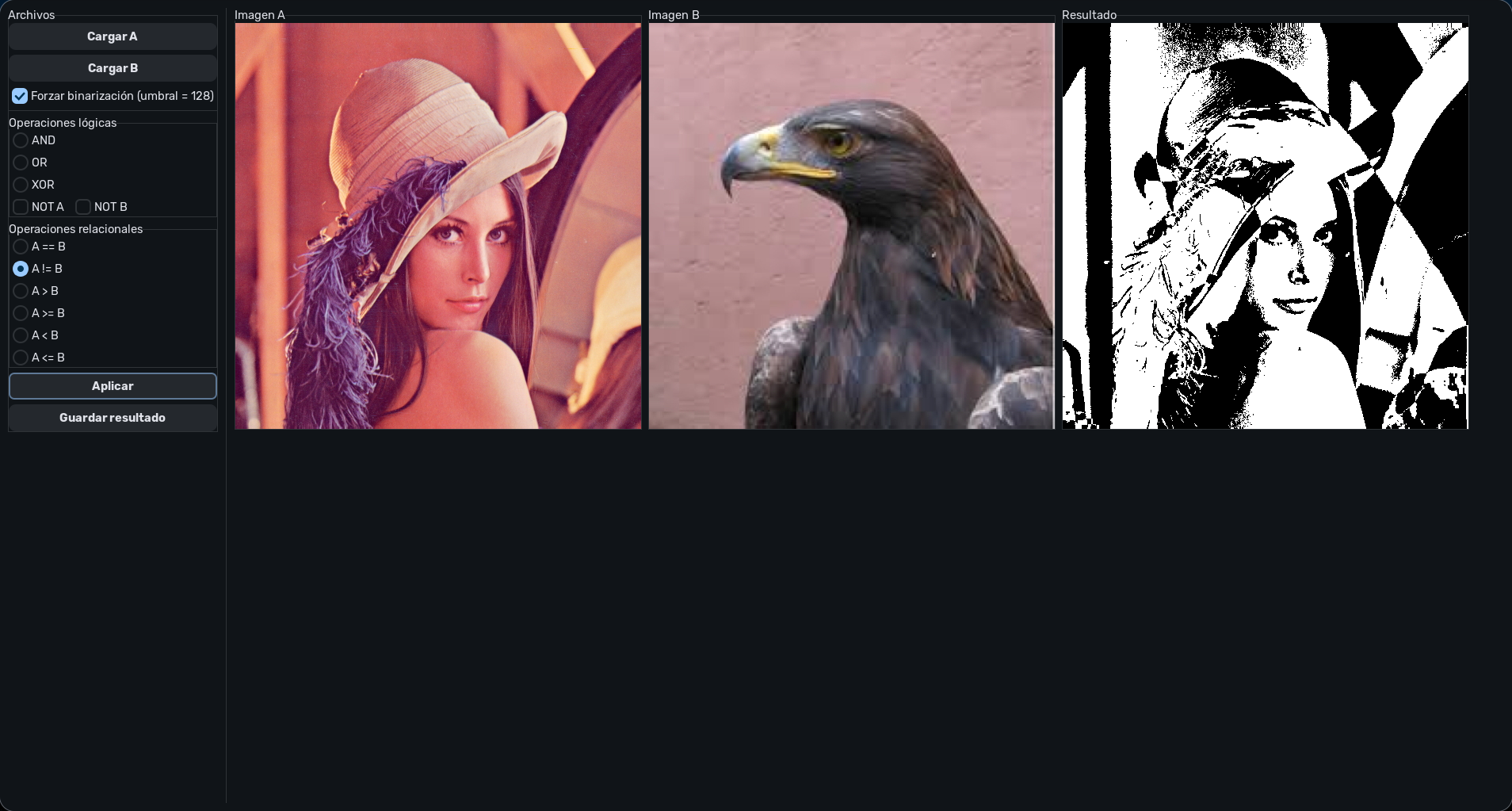
1) Carga: al seleccionar imágenes A y B se muestran en sus visores.  
2) Configuración: se elige operador lógico o relacional; en lógicas se puede activar NOT A / NOT B.

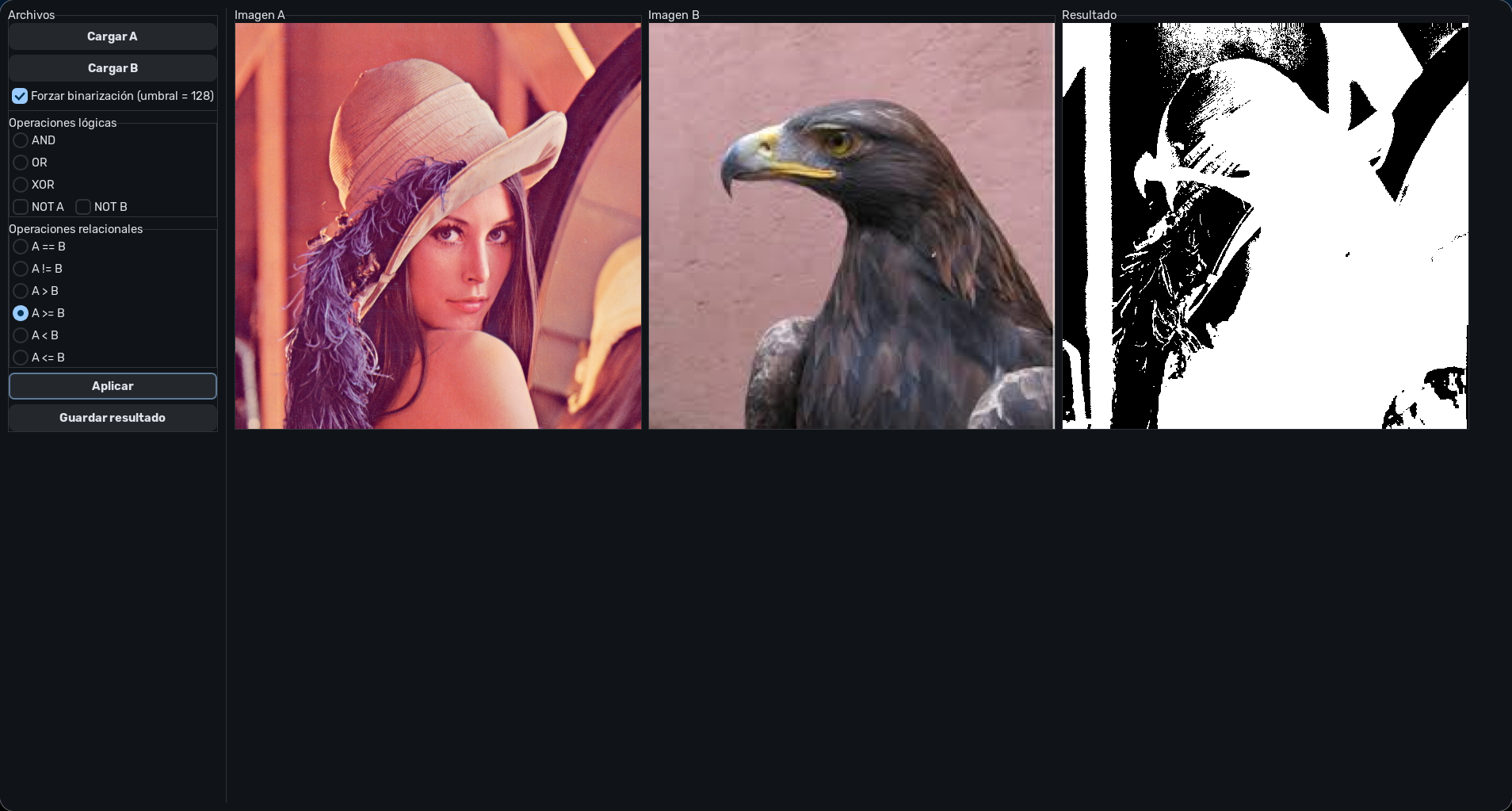












3) Aplicación: al presionar “Aplicar” se calcula la máscara y se despliega en Resultado.  
4) Guardado: “Guardar resultado” abre diálogo y escribe PNG.

**Bloques de código importantes**

Fragmentos clave del zip que conectan la UI con el procesamiento binario. Se muestran recortes representativos (sin cambiar la lógica).

**1) Controlador: enrutamiento de operación según radio-buttons (AppController::on\_apply)**

void AppController::on\_apply(){  
 model\_->set\_force\_binarize(view\_.chk\_binarize().get\_active());  
 std::string err;  
 // Determine op  
 if(view\_.rb\_and().get\_active() || view\_.rb\_or().get\_active() || view\_.rb\_xor().get\_active()){  
 LogicalOp op = LogicalOp::AND;  
 if(view\_.rb\_or().get\_active()) op = LogicalOp::OR;  
 if(view\_.rb\_xor().get\_active()) op = LogicalOp::XOR;  
 bool not\_a = view\_.chk\_not\_a().get\_active();  
 bool not\_b = view\_.chk\_not\_b().get\_active();  
 if(!model\_->apply\_logical(op, not\_a, not\_b, err)) { show\_error(err); return; }  
 } else {  
 RelOp op = RelOp::EQ;  
 if(view\_.rb\_ne().get\_active()) op = RelOp::NE;  
 else if(view\_.rb\_gt().get\_active()) op = RelOp::GT;  
 else if(view\_.rb\_ge().get\_active()) op = RelOp::GE;  
 else if(view\_.rb\_lt().get\_active()) op = RelOp::LT;  
 else if(view\_.rb\_le().get\_active()) op = RelOp::LE;  
 if(!model\_->apply\_relational(op, err)) { show\_error(err); return; }  
 }  
 view\_.refresh\_images();  
}

**2) Modelo: preparación y ejecución de operación lógica (ImageModel::apply\_logical)**

bool ImageModel::apply\_logical(LogicalOp op, bool not\_a, bool not\_b, std::string& err) {  
 if(!pix\_a\_ || !pix\_b\_) { err = "Carga A y B primero."; return false; }  
 auto A = force\_binarize\_ ? binarize\_128(pix\_a\_) : ensure\_gray8(pix\_a\_);  
 auto Btmp = force\_binarize\_ ? binarize\_128(pix\_b\_) : ensure\_gray8(pix\_b\_);  
 auto B = resize\_to\_match(A, Btmp);  
 if(!A || !B) { err = "Error preparando imágenes."; return false; }  
 pix\_result\_ = BinaryOps::logical(A, B, op, not\_a, not\_b);  
 return static\_cast<bool>(pix\_result\_);  
}

**3) Modelo: ejecución de comparación relacional (ImageModel::apply\_relational)**

bool ImageModel::apply\_relational(RelOp op, std::string& err) {  
 if(!pix\_a\_ || !pix\_b\_) { err = "Carga A y B primero."; return false; }  
 auto A = force\_binarize\_ ? binarize\_128(pix\_a\_) : ensure\_gray8(pix\_a\_);  
 auto Btmp = force\_binarize\_ ? binarize\_128(pix\_b\_) : ensure\_gray8(pix\_b\_);  
 auto B = resize\_to\_match(A, Btmp);  
 if(!A || !B) { err = "Error preparando imágenes."; return false; }  
 pix\_result\_ = BinaryOps::relational(A, B, op);  
 return static\_cast<bool>(pix\_result\_);  
}

**4) Modelo: conversión a gris (luma) y réplica R=G=B (ensure\_gray8)**

RefPtr<Gdk::Pixbuf> ImageModel::ensure\_gray8(const RefPtr<Gdk::Pixbuf>& src) {  
 if(!src) return RefPtr<Gdk::Pixbuf>();  
 if(src->get\_n\_channels() == 1 && !src->get\_has\_alpha()) {  
 return src;  
 }  
 // convert RGB(A) to grayscale (luma)  
 auto gray = Gdk::Pixbuf::create(Gdk::COLORSPACE\_RGB, false, 8, src->get\_width(), src->get\_height());  
 src->copy\_area(0,0,src->get\_width(), src->get\_height(), gray, 0,0);  
 // now gray is still RGB; convert to 1 channel buffer  
 int w = gray->get\_width(), h = gray->get\_height();  
 int rs = gray->get\_rowstride();  
 int nchan = gray->get\_n\_channels(); // 3  
 auto out = Gdk::Pixbuf::create(Gdk::COLORSPACE\_RGB, false, 8, w, h);  
 // create final mono buffer  
 auto mono = Gdk::Pixbuf::create(Gdk::COLORSPACE\_RGB, false, 8, w, h); // we'll pack into 1 channel at the end by copying data  
 // We cannot create 1-channel Pixbuf directly, so we'll synthesize by duplicating gray into R=G=B and then later treat as gray.  
 // Simpler: allocate raw buffer and create\_from\_data (keep data alive)  
 std::vector<unsigned char> buf(w\*h);  
 for(int y=0; y<h; ++y) {  
 const unsigned char\* p = gray->get\_pixels() + y\*rs;  
 for(int x=0; x<w; ++x) {  
 unsigned char r = p[x\*nchan+0];  
 unsigned char g = p[x\*nchan+1];  
 unsigned char b = p[x\*nchan+2];  
 unsigned char lum = static\_cast<unsigned char>(0.299\*r + 0.587\*g + 0.114\*b);  
 buf[y\*w+x] = lum;  
 }  
 }  
 // Create 1-channel Pixbuf using create\_from\_data with colorspace RGB=false? Pixbuf doesn't support 1 channel directly.  
 // Workaround: keep grayscale as RGB by replicating channel (R=G=B). Operations will read R only.  
 auto rgb = Gdk::Pixbuf::create(Gdk::COLORSPACE\_RGB, false, 8, w, h);  
 int rs2 = rgb->get\_rowstride();  
 int nc2 = rgb->get\_n\_channels(); // 3  
 for(int y=0; y<h; ++y){  
 unsigned char\* p2 = rgb->get\_pixels() + y\*rs2;  
 for(int x=0; x<w; ++x){  
 unsigned char v = buf[y\*w+x];  
 p2[x\*nc2+0] = v;  
 p2[x\*nc2+1] = v;  
 p2[x\*nc2+2] = v;  
 }  
 }  
 return rgb;  
}

**5) Modelo: binarización por umbral fijo 128 (binarize\_128)**

RefPtr<Gdk::Pixbuf> ImageModel::binarize\_128(const RefPtr<Gdk::Pixbuf>& src) {  
 if(!src) return RefPtr<Gdk::Pixbuf>();  
 auto g = ensure\_gray8(src);  
 int w = g->get\_width(), h = g->get\_height();  
 auto out = Gdk::Pixbuf::create(Gdk::COLORSPACE\_RGB, false, 8, w, h);  
 int rs\_in = g->get\_rowstride();  
 int rs\_out = out->get\_rowstride();  
 int nc = g->get\_n\_channels(); // 3  
 int nco= out->get\_n\_channels(); // 3  
 for(int y=0;y<h;++y){  
 const unsigned char\* pi = g->get\_pixels() + y\*rs\_in;  
 unsigned char\* po = out->get\_pixels() + y\*rs\_out;  
 for(int x=0;x<w;++x){  
 unsigned char lum = pi[x\*nc+0];  
 unsigned char v = (lum >= 128) ? 255 : 0;  
 po[x\*nco+0]=po[x\*nco+1]=po[x\*nco+2]=v;  
 }  
 }  
 return out;  
}

**6) Modelo: reescalado de B al tamaño de A con vecino más cercano (resize\_to\_match)**

RefPtr<Gdk::Pixbuf> ImageModel::resize\_to\_match(const RefPtr<Gdk::Pixbuf>& a, const RefPtr<Gdk::Pixbuf>& b) {  
 if(!a || !b) return RefPtr<Gdk::Pixbuf>();  
 if(a->get\_width()==b->get\_width() && a->get\_height()==b->get\_height()) return b;  
 return b->scale\_simple(a->get\_width(), a->get\_height(), Gdk::INTERP\_NEAREST);  
}

**7) Núcleo: AND/OR/XOR + NOT opcionales, salida 0/255 (BinaryOps::logical)**

Glib::RefPtr<Gdk::Pixbuf> logical(const RefPtr<Gdk::Pixbuf>& A, const RefPtr<Gdk::Pixbuf>& B,  
 LogicalOp op, bool not\_a, bool not\_b) {  
 auto out = make\_out\_like(A);  
 int w = A->get\_width(), h = A->get\_height();  
 int rsa = A->get\_rowstride(), rsb = B->get\_rowstride(), rso = out->get\_rowstride();  
 int nca = A->get\_n\_channels(), ncb=B->get\_n\_channels(), nco=out->get\_n\_channels();  
  
 for(int y=0;y<h;++y){  
 const unsigned char\* pa = A->get\_pixels() + y\*rsa;  
 const unsigned char\* pb = B->get\_pixels() + y\*rsb;  
 unsigned char\* po = out->get\_pixels() + y\*rso;  
 for(int x=0;x<w;++x){  
 unsigned char va = pa[x\*nca+0];  
 unsigned char vb = pb[x\*ncb+0];  
 bool ba = va >= 128;  
 bool bb = vb >= 128;  
 if(not\_a) ba = !ba;  
 if(not\_b) bb = !bb;  
 bool br = false;  
 switch(op){  
 case LogicalOp::AND: br = (ba && bb); break;  
 case LogicalOp::OR: br = (ba || bb); break;  
 case LogicalOp::XOR: br = (ba ^ bb); break;  
 }  
 unsigned char vr = br ? 255 : 0;  
 po[x\*nco+0]=po[x\*nco+1]=po[x\*nco+2]=vr;  
 }  
 }  
 return out;  
}

**8) Núcleo: comparaciones relacionales → máscara binaria (BinaryOps::relational)**

Glib::RefPtr<Gdk::Pixbuf> relational(const RefPtr<Gdk::Pixbuf>& A, const RefPtr<Gdk::Pixbuf>& B, RelOp op) {  
 auto out = make\_out\_like(A);  
 int w = A->get\_width(), h = A->get\_height();  
 int rsa = A->get\_rowstride(), rsb = B->get\_rowstride(), rso = out->get\_rowstride();  
 int nca = A->get\_n\_channels(), ncb=B->get\_n\_channels(), nco=out->get\_n\_channels();  
  
 for(int y=0;y<h;++y){  
 const unsigned char\* pa = A->get\_pixels() + y\*rsa;  
 const unsigned char\* pb = B->get\_pixels() + y\*rsb;  
 unsigned char\* po = out->get\_pixels() + y\*rso;  
 for(int x=0;x<w;++x){  
 unsigned char va = pa[x\*nca+0];  
 unsigned char vb = pb[x\*ncb+0];  
 // compare raw intensities (post-binarización si se activó antes)  
 bool cond=false;  
 switch(op){  
 case RelOp::EQ: cond = (va == vb); break;  
 case RelOp::NE: cond = (va != vb); break;  
 case RelOp::GT: cond = (va > vb); break;  
 case RelOp::GE: cond = (va >= vb); break;  
 case RelOp::LT: cond = (va < vb); break;  
 case RelOp::LE: cond = (va <= vb); break;

**Conclusión**

La práctica integra una interfaz funcional para procesar dos imágenes y ejecutar operadores lógicos y relacionales con salida binaria. El enfoque MVC separa responsabilidades: el Controlador decide la operación desde la UI, el Modelo garantiza entradas compatibles (gris/binarización/reescalado) y BinaryOps realiza el cálculo píxel a píxel. El resultado es una herramienta clara para generar máscaras (segmentación básica, comparación de regiones, detección de diferencias) manteniendo consistencia binaria (0/255), requisito central para operaciones lógicas/relacionales.

**Referencias**

* R. C. Gonzalez, R. E. Woods, Digital Image Processing, Pearson.
* Documentación de GTKmm 3 (gtkmm-3.0) y Gdk::Pixbuf (carga y reescalado).
* CMake + pkg-config para enlace de librerías en Linux.