# Guía para la instalación del procesador RI5CY o **CV32E40P**

**Cabe resaltar que esta guía solo fue testeada y corroborada para ubuntu 24, por lo tanto, no nos hacemos responsables de problemas o cualquier otro inconveniente por usar otra distribución que no sea la mencionada.**

## Primeramente, instalaremos paquetes necesarios:

```
sudo apt update
sudo apt install -y git build-essential make cmake python3 python3-venv python3-pip \
  verilator nodejs npm
```

## Seguidamente vamos a instalar Toolchain Risc – V , el cual es una suit de herramientas que convierte el código normal en C a código máquina ejecutable por un procesador Risc-V

```
# Instala el toolchain newlib en ~/.local/xPacks/...
npx -y xpm@latest install --global @xpack-dev-tools/riscv-none-elf-gcc@latest

# Detecta su carpeta real y agrégalo al PATH de esta terminal
export GCCDIR=$(ls -d "$HOME/.local/xPacks/@xpack-dev-tools/riscv-none-elf-gcc"/*/.content |
head -n1)
export PATH="$GCCDIR/bin:$PATH"

# Verifica
riscv-none-elf-gcc –version
```

## En consecuencia, vamos a clonar y preparar el entorno del core:

```
git clone https://github.com/openhwgroup/core-v-verif.git
cd core-v-verif/bin
python3 -m venv ../.venv
source ../.venv/bin/activate
pip install -r requirements.txt
```

## Ahora vamos a configurar variables de entorno del proyecto:

```
cd ../cv32e40p/sim/core

cat > env.sh <<'EOF'
# Ruta del toolchain xPack (ajusta si tu versión cambia)
export CV_SW_TOOLCHAIN="$HOME/.local/xPacks/@xpack-dev-tools/riscv-none-elf-gcc/14.2.0-3.1/.content"
export CV_SW_PREFIX="riscv-none-elf-"
export CV_SW_MARCH="rv32imc_zicsr"
EOF

source ./env.sh

# Sanidad
which ${CV_SW_PREFIX}gcc
${CV_SW_PREFIX}gcc --version | head -n1
```

## Por último, copilamos y ejecutamos.

```
make -j"$(nproc)"
```

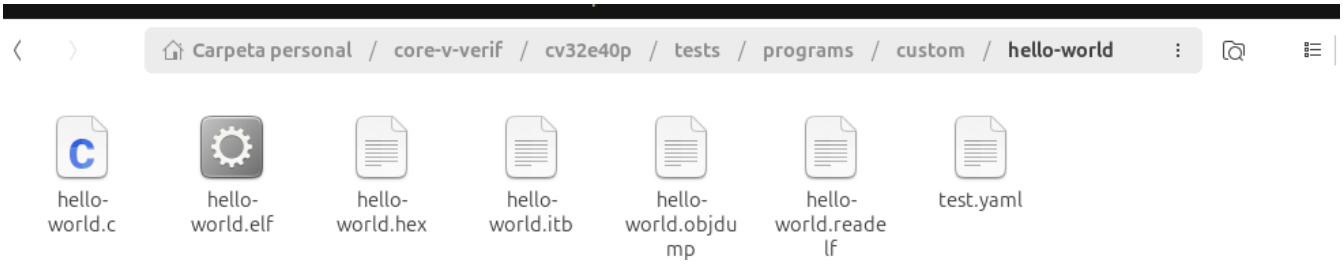
## Si todo se ejecutó de manera correcta deberías ver algo así en pantalla:

```
HELLO WORLD!!!
This is the OpenHW Group CV32E40P CORE-V processor core.
...
XLEN is 32-bits
Supported Instructions Extensions: MIC
...
EXIT SUCCESS
```

**Si cierras la consola y quiere volver a ejecutar el CORE deberás digitar el siguiente comando:**

cd ~/core-v-verif/cv32e40p/sim/core && source ./env.sh && make clean && make -j"$(nproc)"

Si deseas hacer un testbench y tienes tu propio código en C, puedes venir a esta ruta que veras en la siguiente imagen:




En el archivo hello-word.c pegaras tu testbench despues del mensaje de licencias y copyright, ejecutaras con el comando anteriormente mencionado y podrás ver el funcionamiento del procesador.

**Si se te dificulta hacerlo por el explorador de archivos, podrías usar estos comandos:**

cd ~/core-v-verif/cv32e40p/tests/programs/custom/hello-world
cp hello-world.c hello-world.c.bak

**Y para editar el programa:**

nano hello-world.c

# Guía para ejecutar modificación a la arquitectura RI5CY para que ejecute de forma nativa la función XNOR.

Primeramente vas a ejecutar el test bench que se mostrara a continuación:

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

/*
  Demo: Forzar fallo (Illegal Instruction) en CV32E40P con XNOR (Zbb)
  - En PC/Visual Studio: NO hay __riscv -> usa C puro (~(a^b)) y corre bien.
  - En RISC-V (tu core): SÍ hay __riscv -> emite XNOR (Zbb) y el core NO la soporta,
    por lo que TRAPea por instrucción ilegal (justo lo que quieres demostrar).
*/

#define USE_RISCV_XNOR 1   // siempre ASM en build para tu core


static inline uint32_t xnor32_u(uint32_t a, uint32_t b) {
#if USE_RISCV_XNOR
   uint32_t r;
   // R-type: opcode=0x33 (OP), funct3=0x4 (XOR-class), funct7=0x20 -> XNOR (Zbb)
   // .insn r 51, 4, %0, %1, %2, 32
   __asm__ volatile (
      "mv t0, %1\n\t"
      "mv t1, %2\n\t"
      ".word ((0x20 << 25) | (0x5 << 20) | (0x6 << 15) | (4 << 12) | (0x7 << 7) | 0x33)\n\t"
      "mv %0, t2\n\t"
      : "=r"(r) : "r"(a), "r"(b));
   return r;  // <-- En CV32E40P: instrucción ilegal (no Zbb)
#else
   return ~(a ^ b); // versión portable (PC/VS)
#endif
}

int main(void) {
   uint32_t a = 0xF0F0A5A5u;
   uint32_t b = 0x0FF00FF0u;

#if USE_RISCV_XNOR
   puts("RISC-V build: intentando ejecutar XNOR (Zbb) -> deberia TRAPear en CV32E40P");
#else
   puts("Build no-RISC-V: usando version en C (no asm)");
#endif
```

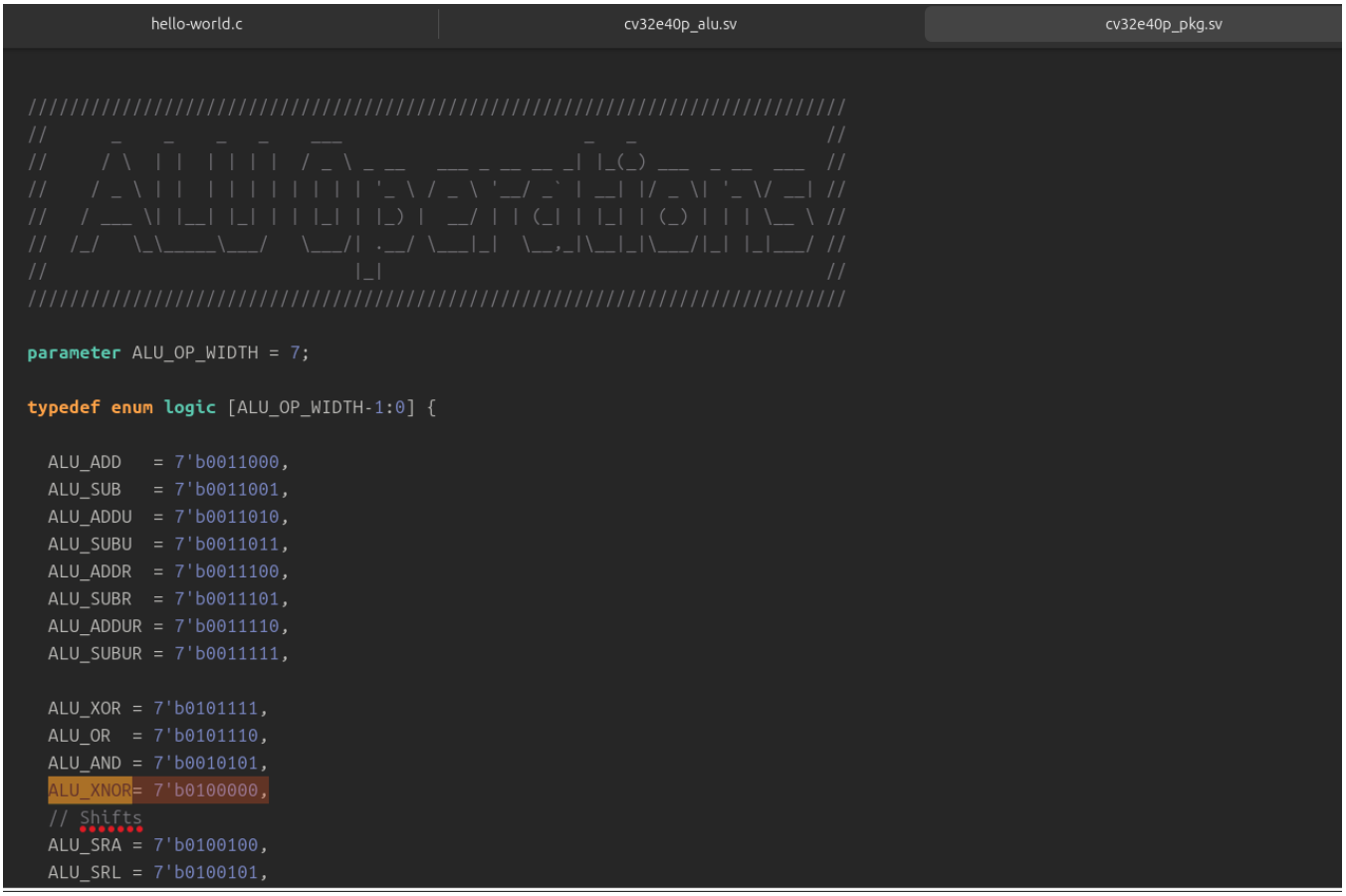
```
  // Referencia en C, por si se imprime en PC:
  uint32_t r_sw = ~(a ^ b);
  printf("refC=0x%08" PRIX32 "\n", r_sw);

  // Aquí, en el core, se ejecuta la XNOR (y TRAPea):
  uint32_t r = xnor32_u(a, b);
  printf("demo=0x%08" PRIX32 "\n", r);
  puts("Si ves esta linea en el core, NO trapo (algo no cuadra).");
  return 0;
}
```

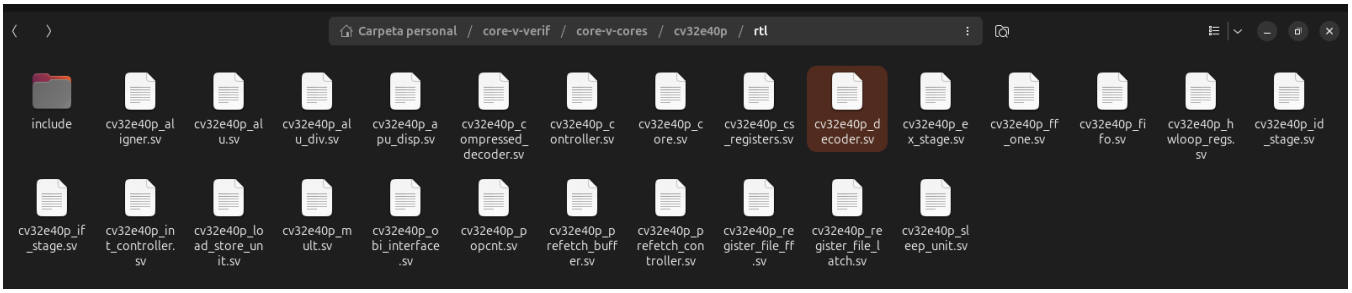
Cuando intentes copilar este código te darás cuenta de que con la arquitectura actual no puede ejecutar este código, por lo cual se procedió a modificar la arquitectura RI5CY para demostrar la factibilidad de agregarle más instrucciones ISA a la arquitectura, por lo cual se darán los pasos necesarios para que puedan modificar la arquitectura.

Primeramente deberán ir a la capeta donde esta ubicada la arquitectura y buscaras la siguiente dirección que veras en la foto y editaras el archivo en block de notas llamado cv32e40p_pkg.sv

Cuando estén dentro del archivo al inicio hay un título grande llamado ALUOPERATION ahí pegaras el siguiente comando ALU_XNOR= 7'b0100000 y guardas el archivo.




Seguidamente modificaremos otro archivo llamado cv32e40p_decoder.sv que estará ubicado en la siguiente dirección:



Dentro del archivo buscaran el titulo llamado ALU donde deberán encontrar este sub índice llamado RV32I ALU operations, cuando ya encuentres este sub índice agrega este comando y guarda el archivo:

{6'b10_0000, 3'b100}: alu_operator_o = ALU_XNOR;  // XNOR (funct7=0100000, funct3=100)

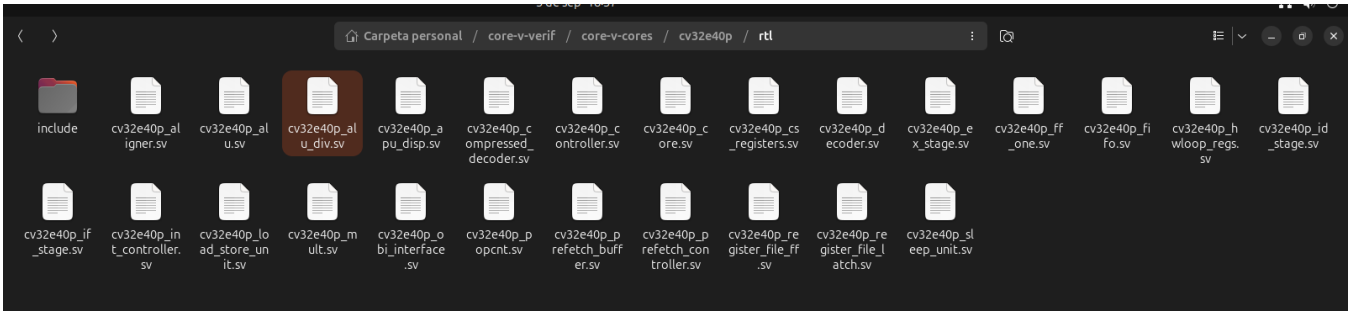
```
// PREFIX 00/01
else begin
  // non bit-manipulation instructions
  regfile_alu_we = 1'b1;
  rega_used_o    = 1'b1;

  if (~instr_rdata_i[28]) regb_used_o = 1'b1;

  unique case ({instr_rdata_i[30:25], instr_rdata_i[14:12]})
    // RV32I ALU operations
    {6'b00_0000, 3'b000}: alu_operator_o = ALU_ADD;   // Add
    {6'b10_0000, 3'b000}: alu_operator_o = ALU_SUB;   // Sub
    {6'b00_0000, 3'b010}: alu_operator_o = ALU_SLTS;  // Set Lower Than
    {6'b00_0000, 3'b011}: alu_operator_o = ALU_SLTU;  // Set Lower Than Unsigned
    {6'b00_0000, 3'b100}: alu_operator_o = ALU_XOR;   // Xor
    {6'b00_0000, 3'b110}: alu_operator_o = ALU_OR;    // Or
    {6'b00_0000, 3'b111}: alu_operator_o = ALU_AND;   // And
    {6'b00_0000, 3'b001}: alu_operator_o = ALU_SLL;   // Shift Left Logical
    {6'b00_0000, 3'b101}: alu_operator_o = ALU_SRL;   // Shift Right Logical
    {6'b10_0000, 3'b101}: alu_operator_o = ALU_SRA;   // Shift Right Arithmetic
    {6'b10_0000, 3'b100}: alu_operator_o = ALU_XNOR;  // XNOR (funct7=0100000, funct3=100)


    // supported RV32M instructions
    {6'b00_0001, 3'b000}: begin // mul
      alu_en          = 1'b0;
      mult_int_en     = 1'b1;
```

Por ultimo modificaremos el archivo llamado cv32e40p_alu.sv que se encontrará en la siguiente dirección:



A este archivo agregaremos el siguiente comando:
ALU_XNOR: result_o = operand_a_i ~^ operand_b_i;

```
////////////////////////////////////////////////////////
//   ____                 _ _     __  __              //
//  |  _ \ ___  ___ _   _| | |_  |  \/  |_   ___  __  //
//  | |_) / _ \/ __| | | | | __| | |\/| | | | \ \/ /  //
//  |  _ <  __/\__ \ |_| | | |_  | |  | | |_| |>  <   //
//  |_| \_\___||___/\__,_|_|\__| |_|  |_|\__,_/_/\_\  //
//                                                    //
////////////////////////////////////////////////////////

always_comb begin
  result_o = '0;

  unique case (operator_i)
    // Standard Operations
    ALU_AND: result_o = operand_a_i & operand_b_i;
    ALU_OR:  result_o = operand_a_i | operand_b_i;
    ALU_XOR: result_o = operand_a_i ^ operand_b_i;
    ALU_XNOR: result_o = operand_a_i ~^ operand_b_i;
```

Con estas modificaciones podremos ejecutar el programa tal como se especifica arriba en este documento. La salida esperada es:

```
        "+firmware=../../tests/programs/custom/hello-world/hello-world.hex" \
       | tee simulation_results/hello-world/0/test_program/hello-world.log
  scopesDump:
    SCOPE 0x792854fffa00: TOP.tb_top_verilator
    SCOPE 0x792854fffa38: TOP.tb_top_verilator.cv32e40p_tb_wrapper_i.ram_i
    SCOPE 0x792854fffa70: TOP.tb_top_verilator.cv32e40p_tb_wrapper_i.ram_i.dp_ram_i
       DPI-EXPORT 0x5e37625aaa20: read_byte
       DPI-EXPORT 0x5e37625aaa30: write_byte

[tb_top_verilator] finished dumping memory
RISC-V build: intentando ejecutar XNOR (Zbb) -> deberia TRAPear en CV32E40P
refC=0x00FF55AA
demo=0x00FF55AA
```