Tutorial Paso a Paso:
Integración de Unidad FP16 en CV32E40X

Proyecto 2 - Arquitectura de Computadoras I
Instituto Tecnológico de Costa Rica

Índice

1. Introducción

Este documento es una guía paso a paso para integrar una unidad de punto flotante FP16 en el procesador RISC-V CV32E40X.

Cada paso incluye:

- El comando exacto que debes ejecutar
- La salida esperada en tu terminal
- Notas sobre qué hace ese comando

Requisitos previos:

- Sistema operativo: Ubuntu 22.04 o superior
- Acceso a internet
- Aproximadamente 2GB de espacio libre

2. Parte 1: Instalación del Toolchain RISC-V

2.1. Paso 1.1: Instalar Node.js y npm

Comando a ejecutar

```
sudo apt update
sudo apt install nodejs npm -y
```

Salida esperada en pantalla

```
Reading package lists... Done
Building dependency tree... Done
The following NEW packages will be installed:
  nodejs npm
...
Setting up nodejs (12.22.9~dfsg-1ubuntu3.6) ...
Setting up npm (8.5.1~ds-1) ...
```

2.2. Paso 1.2: Instalar xpm (xPack Package Manager)

Comando a ejecutar

```
npm install --global xpm@latest
```

Salida esperada en pantalla

```
added 234 packages, and audited 235 packages in 15s
found 0 vulnerabilities
```

2.3. Paso 1.3: Instalar el toolchain RISC-V

Comando a ejecutar

```
xpm install --global @xpack-dev-tools/riscv-none-elf-gcc@latest
```

Salida esperada en pantalla

```
Installing @xpack-dev-tools/riscv-none-elf-gcc@14.2.0-3...
Downloading...
Extracting...
Installing globally in ~/.local/xPacks/...
'@xpack-dev-tools/riscv-none-elf-gcc@14.2.0-3' installed
```

Nota importante

Este proceso descarga aproximadamente 400MB y puede tomar varios minutos dependiendo de tu conexión a internet.

2.4. Paso 1.4: Agregar el toolchain al PATH

Comando a ejecutar

```
echo 'export PATH=$PATH:~/.local/xPacks/@xpack-dev-tools/riscv-none-
    elf-gcc/14.2.0-3.1/.content/bin' >> ~/.bashrc
```

Salida esperada en pantalla

```
(No produce salida visible)
```

2.5. Paso 1.5: Recargar la configuración de bash

Comando a ejecutar

```
source ~/.bashrc
```

Salida esperada en pantalla

```
(No produce salida visible)
```

2.6. Paso 1.6: Verificar la instalación

Comando a ejecutar

```
riscv-none-elf-gcc --version
```

Salida esperada en pantalla

```
riscv-none-elf-gcc (xPack GNU RISC-V Embedded GCC x86_64) 14.2.0
Copyright (C) 2024 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE.
```

Nota importante

Si ves esta salida, **¡el toolchain está correctamente instalado!** Si obtienes un error como `command not found`, revisa los pasos anteriores.

2.7. Paso 1.7: Crear aliases para compatibilidad

Comando a ejecutar

```
echo 'alias riscv32-unknown-elf-gcc=riscv-none-elf-gcc' >> ~/.bashrc
echo 'alias riscv32-unknown-elf-objcopy=riscv-none-elf-objcopy' >> ~/.
    bashrc
echo 'alias riscv32-unknown-elf-objdump=riscv-none-elf-objdump' >> ~/.
    bashrc
echo 'alias riscv32-unknown-elf-as=riscv-none-elf-as' >> ~/.bashrc
echo 'alias riscv32-unknown-elf-ld=riscv-none-elf-ld' >> ~/.bashrc
source ~/.bashrc
```

Salida esperada en pantalla

```
(No produce salida visible)
```

3. Parte 2: Clonar el Proyecto CV32E40X

3.1. Paso 2.1: Instalar Git (si no lo tienes)

Comando a ejecutar

```
sudo apt install git -y
```

Salida esperada en pantalla

```
Reading package lists... Done
git is already the newest version (1:2.34.1-1ubuntu1.11).
0 upgraded, 0 newly installed, 0 to remove
```

3.2. Paso 2.2: Crear directorio de trabajo

Comando a ejecutar

```
mkdir -p ~/riscv-projects
cd ~/riscv-projects
pwd
```

Salida esperada en pantalla

```
/home/tu_usuario/riscv-projects
```

Nota importante

Reemplaza `tu_usuario` con tu nombre de usuario real en Ubuntu.

3.3. Paso 2.3: Clonar el repositorio core-v-verif

Comando a ejecutar

```
git clone --recursive https://github.com/openhwgroup/core-v-verif.git
```

Salida esperada en pantalla

```
Cloning into 'core-v-verif'...
remote: Enumerating objects: 25341, done.
remote: Counting objects: 100% (1245/1245), done.
remote: Compressing objects: 100% (456/456), done.
remote: Total 25341 (delta 823), reused 1102 (delta 745)
Receiving objects: 100% (25341/25341), 15.23 MiB | 5.12 MiB/s, done.
Resolving deltas: 100% (16789/16789), done.
Submodule 'core-v-cores/cv32e40p' (https://github.com/...)
...
Submodule path 'vendor/riscv/riscv-isa-sim': checked out 'abc123'
```

Nota importante

Este proceso descarga aproximadamente 500MB y puede tomar 5-10 minutos. Verás muchas líneas de submódulos siendo clonados.

3.4. Paso 2.4: Verificar la estructura del proyecto

Comando a ejecutar

```
cd core-v-verif
ls -la
```

Salida esperada en pantalla

```
total 152
drwxrwxr-x 17 usuario usuario  4096 nov 10 09:00 .
drwxrwxr-x  3 usuario usuario  4096 nov 10 08:55 ..
drwxrwxr-x  5 usuario usuario  4096 nov 10 09:00 bin
drwxrwxr-x  4 usuario usuario  4096 nov 10 09:00 core-v-cores
drwxrwxr-x 10 usuario usuario  4096 nov 10 09:00 cv32e40p
drwxrwxr-x 10 usuario usuario  4096 nov 10 09:00 cv32e40s
drwxrwxr-x 11 usuario usuario  4096 nov 10 09:00 cv32e40x
drwxrwxr-x  8 usuario usuario  4096 nov 10 09:00 .git
-rw-rw-r--  1 usuario usuario  2814 nov 10 09:00 LICENSE.md
-rw-rw-r--  1 usuario usuario  4126 nov 10 09:00 README.md
...
```

3.5. Paso 2.5: Navegar al directorio CV32E40X

Comando a ejecutar

```
cd cv32e40x
ls -la
```

Salida esperada en pantalla

```
total 48
drwxrwxr-x 11 usuario usuario 4096 nov 10 09:00 .
drwxrwxr-x 17 usuario usuario 4096 nov 10 09:00 ..
drwxrwxr-x  2 usuario usuario 4096 nov 10 09:00 bsp
drwxrwxr-x  3 usuario usuario 4096 nov 10 09:00 docs
drwxrwxr-x  4 usuario usuario 4096 nov 10 09:00 env
-rw-rw-r--  1 usuario usuario  601 nov 10 09:00 README.md
drwxrwxr-x  2 usuario usuario 4096 nov 10 09:00 regress
drwxrwxr-x  2 usuario usuario 4096 nov 10 09:00 rtl
drwxrwxr-x  5 usuario usuario 4096 nov 10 09:00 sim
drwxrwxr-x  5 usuario usuario 4096 nov 10 09:00 tb
drwxrwxr-x  8 usuario usuario 4096 nov 10 09:00 tests
drwxrwxr-x  6 usuario usuario 4096 nov 10 09:00 vendor_lib
```

4. Parte 3: Copiar tu Unidad FP16

4.1. Paso 3.1: Crear directorio para unidades custom

Comando a ejecutar

```
mkdir -p ~/riscv-projects/core-v-verif/cv32e40x/rtl/custom_units
```

Salida esperada en pantalla

```
(No produce salida visible)
```

4.2. Paso 3.2: Copiar archivos de tu unidad FP16

Advertencia

IMPORTANTE: Reemplaza `/ruta/a/tu/ComaFlotante` con la ruta real donde tienes tu proyecto de punto flotante.

Comando a ejecutar

```
cp /ruta/a/tu/ComaFlotante/*.v ~/riscv-projects/core-v-verif/cv32e40x/
   rtl/custom_units/
```

Salida esperada en pantalla

```
(No produce salida visible si todo va bien)
```

Ejemplo con ruta real:

Comando a ejecutar

```
cp /home/milagro/Escritorio/ComaFlotante/*.v ~/riscv-projects/core-v-
   verif/cv32e40x/rtl/custom_units/
```

4.3. Paso 3.3: Verificar archivos copiados

Comando a ejecutar

```
ls ~/riscv-projects/core-v-verif/cv32e40x/rtl/custom_units/
```

Salida esperada en pantalla

```
control.v            fp16_add_lane.v     fp16_mul2lanes.v
diferencial5bits.v   fp16_classify.v     fp16_mul_lane.v
especial_cases.v     fp16_expacc.v       fp16_normround.v
fp16_add2lanes.v     fp16_mantmul.v      fp16_reciprocal_dual.v
fp16_reciprocal.v    fp16_special_mul.v  fp16_unpack.v
Manager.v            mux_2x10bits.v      shift_left_right.v
mux_ExpM.v           shift_regis_right.v sumador_Alubig.v
sumador_Alusmall.v
```

Nota importante

Debes ver todos tus archivos .v listados aquí. Si no ves ningún archivo, verifica la ruta de origen.

5. Parte 4: Crear Programa de Prueba

5.1. Paso 4.1: Navegar al directorio de tests

Comando a ejecutar

```
cd ~/riscv-projects/core-v-verif/cv32e40x/tests/custom
pwd
```

Salida esperada en pantalla

```
/home/usuario/riscv-projects/core-v-verif/cv32e40x/tests/custom
```

5.2. Paso 4.2: Crear archivo de prueba FP16

Comando a ejecutar

```
nano fp16_test.c
```

Nota importante

Se abrirá el editor nano. Copia el siguiente código EXACTAMENTE como está:

```
/*
 * Test FP16 Custom Instructions
 */

#include <stdint.h>

#define DEBUG_OUT (*((volatile uint32_t*)0x10000000))

// FP16.ADD rd, rs1, rs2
#define FP16_ADD(rd, rs1, rs2) \
    __asm__ volatile ( \
        ".insn r 0x2B, 0x0, 0x00, %0, %1, %2" \
        : "=r"(rd) \
        : "r"(rs1), "r"(rs2) \
    )

// FP16.MUL rd, rs1, rs2
#define FP16_MUL(rd, rs1, rs2) \
    __asm__ volatile ( \
        ".insn r 0x2B, 0x1, 0x00, %0, %1, %2" \
        : "=r"(rd) \
        : "r"(rs1), "r"(rs2) \
    )

// FP16.RCP rd, rs1, rs2
#define FP16_RCP(rd, rs1, rs2) \
    __asm__ volatile ( \
        ".insn r 0x2B, 0x2, 0x00, %0, %1, %2" \
        : "=r"(rd) \
        : "r"(rs1), "r"(rs2) \
    )

void test_pass() {
    DEBUG_OUT = 0x12345678;
    while(1);
}

int main() {
    uint32_t a, b, result;

    DEBUG_OUT = 0xFEED0001;

    // Test 1: FP16 ADD
    a = 0x3C003C00;
    b = 0x3C003C00;

    DEBUG_OUT = 0x0A000000;
    DEBUG_OUT = a;
    DEBUG_OUT = b;

    FP16_ADD(result, a, b);

    DEBUG_OUT = result;
```

```
    // Test 2: FP16 MUL
    a = 0x40004000;
    b = 0x3C003C00;

    DEBUG_OUT = 0x0D000000;
    DEBUG_OUT = a;
    DEBUG_OUT = b;

    FP16_MUL(result, a, b);

    DEBUG_OUT = result;

    // Test 3: FP16 RCP
    a = 0x40004000;
    b = 0x00000000;

    DEBUG_OUT = 0x0C000000;
    DEBUG_OUT = a;

    FP16_RCP(result, a, b);

    DEBUG_OUT = result;

    DEBUG_OUT = 0xFEED0002;

    test_pass();

    return 0;
}
```

Nota importante

Una vez copiado todo el código:

- Presiona `Ctrl+O` (guardar)
- Presiona `Enter` (confirmar nombre)
- Presiona `Ctrl+X` (salir)

5.3. Paso 4.3: Compilar el programa

Comando a ejecutar

```
riscv32-unknown-elf-gcc -march=rv32imc -mabi=ilp32 -O0 -g -nostdlib -
    nostartfiles -T /dev/null -Wl,--entry=main -o fp16_test.elf
    fp16_test.c
```

Salida esperada en pantalla

```
/home/usuario/.local/xPacks/@xpack-dev-tools/riscv-none-elf-gcc/14.2.0-3.1/.
    content/bin/../lib/gcc/riscv-none-elf/14.2.0/../../../../riscv-none-elf/
    bin/ld: warning: fp16_test.elf has a LOAD segment with RWX permissions
```

Nota importante

Esta advertencia es **normal** y no afecta la funcionalidad. Si ves esta advertencia, significa que **compiló correctamente**.

5.4. Paso 4.4: Convertir a formato hexadecimal

Comando a ejecutar

```
riscv32-unknown-elf-objcopy -O verilog fp16_test.elf fp16_test.hex
```

Salida esperada en pantalla

```
(No produce salida visible si todo va bien)
```

5.5. Paso 4.5: Verificar archivos generados

Comando a ejecutar

```
ls -lh fp16_test.*
```

Salida esperada en pantalla

```
-rw-rw-r-- 1 usuario usuario 1.5K nov 10 12:11 fp16_test.c
-rwxrwxr-x 1 usuario usuario 6.8K nov 10 12:11 fp16_test.elf
-rwxrwxr-x 1 usuario usuario  881 nov 10 12:15 fp16_test.hex
```

Nota importante

Si ves estos tres archivos, **¡la compilación fue exitosa!**

6. Parte 5: Verificar Instrucciones Custom

6.1. Paso 5.1: Ver el código desensamblado

Comando a ejecutar

```
riscv-none-elf-objdump -d fp16_test.elf | grep "\.insn"
```

Salida esperada en pantalla

```
  6a:   00e787ab                .insn   4, 0x00e787ab
  b6:   00e797ab                .insn   4, 0x00e797ab
  f0:   00e7a7ab                .insn   4, 0x00e7a7ab
```

Nota importante

Estas tres líneas son tus instrucciones custom FP16:

- `0x00e787ab` = FP16.ADD
- `0x00e797ab` = FP16.MUL
- `0x00e7a7ab` = FP16.RCP

6.2. Paso 5.2: Ver el desensamblado completo (opcional)

Comando a ejecutar

```
riscv-none-elf-objdump -d fp16_test.elf > fp16_test.asm
head -50 fp16_test.asm
```

Salida esperada en pantalla

```
fp16_test.elf:     file format elf32-littleriscv

Disassembly of section .text:

00000000 <test_pass>:
   0:   1141                    addi    sp,sp,-16
   2:   c606                    sw      ra,12(sp)
   4:   c422                    sw      s0,8(sp)
   6:   0800                    addi    s0,sp,16
   8:   100007b7                lui     a5,0x10000
   c:   12345737                lui     a4,0x12345
  10:   67870713                addi    a4,a4,1656
  14:   c398                    sw      a4,0(a5)
  16:   a001                    j       16 <test_pass+0x16>

00000018 <main>:
  18:   1101                    addi    sp,sp,-32
...
```

7. Resumen de lo Completado

Estado del Proyecto

Al completar todos estos pasos, has logrado:

1. Instalar y configurar el toolchain RISC-V
2. Clonar el proyecto CV32E40X completo
3. Copiar tu unidad FP16 al proyecto
4. Crear un programa de prueba con instrucciones custom
5. Compilar exitosamente el programa
6. Generar archivos .hex para simulación
7. Verificar que las instrucciones custom se generaron correctamente

Archivos importantes generados:

- `fp16_test.c` - Código fuente
- `fp16_test.elf` - Ejecutable RISC-V
- `fp16_test.hex` - Memoria en formato Verilog
- `fp16_test.asm` - Desensamblado (opcional)

8. Parte 6: Integración del Hardware FP16

8.1. Introducción

En esta parte vamos a integrar la unidad FP16 en el core CV32E40X modificando el RTL. Los cambios incluyen:

- Agregar opcodes y operadores custom
- Crear decoder para instrucciones FP16
- Integrar la unidad en el EX stage
- Conectar señales de control y datos

8.2. Paso 6.1: Navegar al directorio del core

Comando a ejecutar

```
cd ~/core-v-verif/core-v-cores/cv32e40x/rtl
pwd
```

Salida esperada en pantalla

```
/home/usuario/core-v-verif/core-v-cores/cv32e40x/rtl
```

8.3. Paso 6.2: Hacer backup del package

Comando a ejecutar

```
cp include/cv32e40x_pkg.sv include/cv32e40x_pkg.sv.backup
ls -lh include/cv32e40x_pkg.sv*
```

Salida esperada en pantalla

```
-rw-rw-r-- 1 usuario usuario 43K nov 10 09:55 include/cv32e40x_pkg.sv
-rw-rw-r-- 1 usuario usuario 43K nov 10 15:17 include/cv32e40x_pkg.sv.backup
```

8.4. Paso 6.3: Modificar cv32e40x_pkg.sv - Agregar OPCODE_CUSTOM_1

Comando a ejecutar

```
nano +40 include/cv32e40x_pkg.sv
```

Nota importante

En la línea 51, después de `OPCODE_AMO = 7'h2F`, agregar una coma y una nueva línea:

Listing 1: Agregar en línea 52

```
1                    OPCODE_CUSTOM_1  = 7'h2B
```

Debe quedar:

```
                        OPCODE_LUI       = 7'h37,
                        OPCODE_AMO       = 7'h2F,
                        OPCODE_CUSTOM_1  = 7'h2B
```

Guardar: Ctrl+O, Enter. Salir: Ctrl+X.

8.5. Paso 6.4: Agregar operadores ALU FP16

Comando a ejecutar

```
nano +140 include/cv32e40x_pkg.sv
```

Nota importante

Justo antes de la línea `} alu_opcode_e;`, agregar:

Listing 2: Agregar operadores FP16

```
 // FP16 custom operations
 ALU_FP16_ADD = 6'b101101, // FP16 addition (2 lanes)
 ALU_FP16_MUL = 6'b110110, // FP16 multiplication (2 lanes)
 ALU_FP16_RCP = 6'b111010  // FP16 reciprocal (2 lanes)
} alu_opcode_e;
```

Guardar y salir.

8.6. Paso 6.5: Crear el wrapper FP16

Comando a ejecutar

```
cd ~/core-v-verif/cv32e40x/rtl/custom_units
nano fp16_wrapper.sv
```

Copiar el siguiente código:

Listing 3: fp16_wrapper.sv

```
// fp16_wrapper.sv
module fp16_wrapper
  import cv32e40x_pkg::*;
(
  input  logic        clk,
  input  logic        rst_n,
  input  logic        fp16_en_i,
  input  logic [1:0]  fp16_operator_i,
  input  logic [31:0] operand_a_i,
  input  logic [31:0] operand_b_i,
  output logic [31:0] result_o,
  output logic        illegal_o,
  output logic        op_invalid_o,
  input  logic        valid_i,
  output logic        ready_o,
  output logic        valid_o,
  input  logic        ready_i
);

  logic [31:0] fp16_result;
  logic        fp16_illegal_any;
  logic        fp16_op_invalid_any;
  logic        fp16_mul_illegal_sub;
  logic        fp16_mul_op_invalid;
  logic [1:0]  fp16_rcp_illegal_vec;

  fp16_vec_alu u_fp16_alu (
    .RST            (~rst_n),
    .ENA            (fp16_en_i),
    .a              (operand_a_i),
    .b              (operand_b_i),
    .op_sel         (fp16_operator_i),
    .res            (fp16_result),
    .illegal_any    (fp16_illegal_any),
    .op_invalid_any (fp16_op_invalid_any),
    .mul_illegal_sub(fp16_mul_illegal_sub),
    .mul_op_invalid (fp16_mul_op_invalid),
    .rcp_illegal_vec(fp16_rcp_illegal_vec)
```

```
  );

  assign result_o = fp16_result;
  assign illegal_o = fp16_illegal_any;
  assign op_invalid_o = fp16_op_invalid_any;
  assign ready_o = ready_i;
  assign valid_o = valid_i && fp16_en_i;

endmodule
```

Guardar y salir.

8.7. Paso 6.6: Crear el decoder FP16

Comando a ejecutar

```
cd ~/core-v-verif/core-v-cores/cv32e40x/rtl
nano cv32e40x_fp16_decoder.sv
```

Copiar el siguiente código:

Listing 4: cv32e40x_fp16_decoder.sv

```
module cv32e40x_fp16_decoder
  import cv32e40x_pkg::*;
(
  input logic [31:0] instr_rdata_i,
  output decoder_ctrl_t decoder_ctrl_o
);

  always_comb begin
    decoder_ctrl_o = DECODER_CTRL_ILLEGAL_INSN;
    decoder_ctrl_o.illegal_insn = 1'b0;

    if (instr_rdata_i[6:0] == OPCODE_CUSTOM_1) begin
      if (instr_rdata_i[31:25] == 7'b0000000) begin
        case (instr_rdata_i[14:12])
          3'b000: begin // FP16.ADD
            decoder_ctrl_o.alu_en           = 1'b1;
            decoder_ctrl_o.alu_operator     = ALU_FP16_ADD;
            decoder_ctrl_o.alu_op_a_mux_sel = OP_A_REGA_OR_FWD;
            decoder_ctrl_o.alu_op_b_mux_sel = OP_B_REGB_OR_FWD;
            decoder_ctrl_o.rf_we            = 1'b1;
            decoder_ctrl_o.rf_re[0]         = 1'b1;
            decoder_ctrl_o.rf_re[1]         = 1'b1;
          end
          3'b001: begin // FP16.MUL
            decoder_ctrl_o.alu_en           = 1'b1;
            decoder_ctrl_o.alu_operator     = ALU_FP16_MUL;
            decoder_ctrl_o.alu_op_a_mux_sel = OP_A_REGA_OR_FWD;
            decoder_ctrl_o.alu_op_b_mux_sel = OP_B_REGB_OR_FWD;
            decoder_ctrl_o.rf_we            = 1'b1;
            decoder_ctrl_o.rf_re[0]         = 1'b1;
            decoder_ctrl_o.rf_re[1]         = 1'b1;
          end
          3'b010: begin // FP16.RCP
            decoder_ctrl_o.alu_en           = 1'b1;
            decoder_ctrl_o.alu_operator     = ALU_FP16_RCP;
            decoder_ctrl_o.alu_op_a_mux_sel = OP_A_REGA_OR_FWD;
            decoder_ctrl_o.alu_op_b_mux_sel = OP_B_REGB_OR_FWD;
            decoder_ctrl_o.rf_we            = 1'b1;
            decoder_ctrl_o.rf_re[0]         = 1'b1;
            decoder_ctrl_o.rf_re[1]         = 1'b0;
          end
          default: decoder_ctrl_o = DECODER_CTRL_ILLEGAL_INSN;
        endcase
      end else begin
        decoder_ctrl_o = DECODER_CTRL_ILLEGAL_INSN;
      end
    end else begin
      decoder_ctrl_o = DECODER_CTRL_ILLEGAL_INSN;
    end
  end
endmodule
```

Guardar y salir.

8.8. Paso 6.7: Modificar cv32e40x_decoder.sv

Comando a ejecutar

```
cp cv32e40x_decoder.sv cv32e40x_decoder.sv.backup
nano +105 cv32e40x_decoder.sv
```

Cambio 1: Después de línea 108, agregar:

```
decoder_ctrl_t decoder_fp16_ctrl;
```

Cambio 2: Después del bloque de b_decoder (línea 165), agregar:

```
// FP16 custom decoder
cv32e40x_fp16_decoder fp16_decoder_i
(
  .instr_rdata_i   ( instr_rdata_i      ),
  .decoder_ctrl_o  ( decoder_fp16_ctrl  )
);
```

Cambio 3: En el case (línea 176), agregar antes del default:

```
    !decoder_fp16_ctrl.illegal_insn : decoder_ctrl_mux_subdec = decoder_fp16_ctrl;
```

Guardar y salir.

8.9. Paso 6.8: Modificar cv32e40x_ex_stage.sv

Comando a ejecutar

```
cp cv32e40x_ex_stage.sv cv32e40x_ex_stage.sv.backup
nano +94 cv32e40x_ex_stage.sv
```

Cambio 1: Después de línea 97, agregar:

```
// FP16 signals
logic [31:0]    fp16_result;
logic           fp16_en;
logic           fp16_ready;
logic           fp16_valid;
```

Cambio 2: Buscar `mul_en_gated`, después agregar:

```
assign fp16_en = instr_valid && (id_ex_pipe_i.alu_operator == ALU_FP16_ADD ||
                                 id_ex_pipe_i.alu_operator == ALU_FP16_MUL ||
                                 id_ex_pipe_i.alu_operator == ALU_FP16_RCP);
```

Cambio 3: En el case de rf_wdata_o (línea 152), agregar AL INICIO:

```
    fp16_en             : rf_wdata_o = fp16_result;
```

Cambio 4: Después del bloque DIV, agregar:

```
// FP16 custom unit
fp16_wrapper fp16_wrapper_i
(
  .clk             ( clk                          ),
  .rst_n           ( rst_n                        ),
  .fp16_en_i       ( fp16_en                      ),
  .fp16_operator_i ( id_ex_pipe_i.alu_operator[1:0]),
  .operand_a_i     ( id_ex_pipe_i.alu_operand_a   ),
  .operand_b_i     ( id_ex_pipe_i.alu_operand_b   ),
  .result_o        ( fp16_result                  ),
  .illegal_o       (                              ),
  .op_invalid_o    (                              ),
  .valid_i         ( fp16_en                      ),
  .ready_o         ( fp16_ready                   ),
  .valid_o         ( fp16_valid                   ),
  .ready_i         ( wb_ready_i                   )
);
```

Cambio 5: En `ex_ready_o`, agregar `&& fp16_ready`
Cambio 6: En `ex_valid_o`, agregar:

```
                    (fp16_en && fp16_valid) ||
```

Guardar y salir.

8.10. Verificación de cambios

Comando a ejecutar

```
grep -n "OPCODE_CUSTOM_1" include/cv32e40x_pkg.sv
grep -n "ALU_FP16" include/cv32e40x_pkg.sv
grep -n "decoder_fp16_ctrl" cv32e40x_decoder.sv
grep -n "fp16" cv32e40x_ex_stage.sv | head -10
```

Salida esperada en pantalla

```
52:                              OPCODE_CUSTOM_1   = 7'h2B
148:  ALU_FP16_ADD = 6'b101101,
149:  ALU_FP16_MUL = 6'b110110,
150:  ALU_FP16_RCP = 6'b111010
109:   decoder_ctrl_t decoder_fp16_ctrl;
172:     .decoder_ctrl_o   ( decoder_fp16_ctrl   )
186:       !decoder_fp16_ctrl.illegal_insn : ...
100:   logic [31:0]     fp16_result;
101:   logic            fp16_en;
...
```

Integración RTL Completada

¡Felicidades! Has completado la integración del hardware FP16 en el CV32E40X.
Archivos modificados:

- cv32e40x_pkg.sv
- cv32e40x_decoder.sv
- cv32e40x_ex_stage.sv

Archivos creados:

- cv32e40x_fp16_decoder.sv
- fp16_wrapper.sv

9. Solución de Problemas Comunes

9.1. Error: command not found

Problema:

```
bash: riscv-none-elf-gcc: command not found
```

Solución:

Comando a ejecutar

```
echo $PATH | grep riscv
# Si no aparece nada, ejecutar:
source ~/.bashrc
# Y verificar de nuevo:
riscv-none-elf-gcc --version
```

9.2. Error: No such file or directory al copiar archivos

Problema:

```
cp: cannot stat '/ruta/a/tu/ComaFlotante/*.v':
No such file or directory
```

Solución: Verificar la ruta correcta:

Comando a ejecutar

```
ls /home/$USER/Escritorio/ComaFlotante/
# o buscar el directorio:
find ~ -name "Manager.v" 2>/dev/null
```

9.3. Advertencia: RWX permissions

Problema:

```
warning: fp16_test.elf has a LOAD segment with RWX permissions
```

Solución: Esta advertencia es **normal y puede ignorarse**. No afecta la funcionalidad del programa.

10. Parte 7: Síntesis de la Unidad FP16

10.1. Introducción

En esta parte vamos a sintetizar la unidad FP16 de forma independiente para verificar que compile correctamente y obtener métricas de recursos.

10.2. Paso 7.1: Crear lista de archivos RTL

Comando a ejecutar

```
cd ~/proyecto2_fp16/core-v-verif/cv32e40x
cat > rtl_files_fp16.f << 'EOF'
# FP16 Custom Units
rtl/custom_units/control.v
rtl/custom_units/diferencial5bits.v
rtl/custom_units/especial_cases.v
rtl/custom_units/fp16_add2lanes.v
rtl/custom_units/fp16_add_lane.v
rtl/custom_units/fp16_classify.v
rtl/custom_units/fp16_expacc.v
rtl/custom_units/fp16_mantmul.v
rtl/custom_units/fp16_mul2lanes.v
rtl/custom_units/fp16_mul_lane.v
rtl/custom_units/fp16_normround.v
rtl/custom_units/fp16_reciprocal_dual.v
rtl/custom_units/fp16_reciprocal.v
rtl/custom_units/fp16_special_mul.v
rtl/custom_units/fp16_unpack.v
rtl/custom_units/Manager.sv
rtl/custom_units/mux_2x10bits.v
rtl/custom_units/mux_ExpM.v
rtl/custom_units/shift_left_right.v
rtl/custom_units/shift_regis_right.v
rtl/custom_units/sumador_Alubig.v
rtl/custom_units/sumador_Alusmall.v
EOF
```

Salida esperada en pantalla

(No produce salida visible)

10.3. Paso 7.2: Crear script de síntesis TCL

Comando a ejecutar

```
cat > synthesize_fp16_only.tcl << 'ENDTCL'
# Crear proyecto
create_project -force vivado_fp16_only ./vivado_fp16_only \
    -part xck26-sfvc784-2LV-c

# Leer archivos RTL
set fp [open "rtl_files_fp16.f" r]
set files [split [read $fp] "\n"]
close $fp

foreach file $files {
    set file [string trim $file]
    if {[string length $file] > 0 && ![string match "#*" $file]} {
        if {[file exists $file]} {
            add_files -fileset sources_1 $file
        }
    }
}

# Configurar top module
set_property top fp16_vec_alu [current_fileset]

# Ejecutar sintesis
launch_runs synth_1 -jobs 4
wait_on_run synth_1

# Generar reportes
open_run synth_1
report_utilization -file fp16_utilization.txt
report_timing_summary -file fp16_timing.txt

puts "Sintesis completada!"
puts "Ver reportes en:"
puts "  - fp16_utilization.txt"
puts "  - fp16_timing.txt"

quit
ENDTCL
```

Salida esperada en pantalla

(No produce salida visible)

10.4. Paso 7.3: Ejecutar síntesis

Comando a ejecutar

```
source /tools/Xilinx-2023.2/Vivado/2023.2/settings64.sh
vivado -mode batch -source synthesize_fp16_only.tcl
```

Salida esperada en pantalla

```
****** Vivado v2023.2 (64-bit)
...
Creating project...
Adding sources...
INFO: [Synth 8-6157] synthesizing module 'fp16_vec_alu'
...
Synthesis Optimization Runtime : 00:00:15
Implementation Feasibility Runtime : 00:00:05
...
Synthesis finished with 0 errors, 0 critical warnings
...
Sintesis completada!
Ver reportes en:
  - fp16_utilization.txt
  - fp16_timing.txt
```

Nota importante

El proceso de síntesis puede tomar entre 5-10 minutos dependiendo de tu computadora.

10.5. Paso 7.4: Ver reporte de utilización de recursos

Comando a ejecutar

```
head -50 fp16_utilization.txt
```

Salida esperada en pantalla

```
1. Slice Logic
--------------

+-------------------------------------+------+-------+-----------+
|              Site Type              | Used | Fixed | Available |
+-------------------------------------+------+-------+-----------+
| Slice LUTs                          | 1109 |     0 |    117120 |
|   LUT as Logic                      | 1109 |     0 |    117120 |
|   LUT as Memory                     |    0 |     0 |     57600 |
| Slice Registers                     |    0 |     0 |    234240 |
|   Register as Flip Flop             |    0 |     0 |    234240 |
|   Register as Latch                 |    0 |     0 |    234240 |
| F7 Muxes                            |    0 |     0 |     58560 |
| F8 Muxes                            |    0 |     0 |     29280 |
+-------------------------------------+------+-------+-----------+

2. DSP
------

+----------+------+-------+-----------+
| Site Type| Used | Fixed | Available |
+----------+------+-------+-----------+
| DSPs     |    2 |     0 |      1248 |
+----------+------+-------+-----------+
...
```

10.6. Paso 7.5: Análisis de recursos

Los resultados de síntesis muestran que la unidad FP16 utiliza:

Recurso	**Usado**	**Disponible**	**Utilización**
LUTs	1,109	117,120	0.95 %
Flip-Flops	0	234,240	0.00 %
DSPs	2	1,248	0.16 %
Block RAM	0	144	0.00 %
CARRY8	92	14,640	0.63 %

Cuadro 1: Recursos utilizados por la unidad FP16 en Kria KV260

Nota importante

La unidad FP16 es **puramente combinacional** (0 Flip-Flops), lo que significa que tiene una latencia de 1 ciclo. Los 2 DSPs se usan para los multiplicadores de mantisa en las operaciones de multiplicación.

11. Parte 8: Simulación y Verificación de la Unidad FP16

11.1. Introducción

En esta parte vamos a crear un testbench para verificar el funcionamiento de la unidad FP16 mediante simulación RTL.

11.2. Paso 8.1: Crear testbench personalizable (Parte 1)

Comando a ejecutar

```
cat > rtl/custom_units/tb_fp16_user.sv << 'EOF'
`timescale 1ns/1ps

module tb_fp16_user;

  logic        RST;
  logic        ENA;
  logic [31:0] a;
  logic [31:0] b;
  logic [1:0]  op_sel;
  logic [31:0] res;
  logic        illegal_any;
  logic        op_invalid_any;
  logic        mul_illegal_sub;
  logic        mul_op_invalid;
  logic [1:0]  rcp_illegal_vec;

  fp16_vec_alu dut (.*);

  task test_fp16(
    input [15:0] a_lo, input [15:0] a_hi,
    input [15:0] b_lo, input [15:0] b_hi,
    input [1:0]  operation,
    input string op_name
  );
    a = {a_hi, a_lo};
    b = {b_hi, b_lo};
    op_sel = operation;
    ENA = 1;
    RST = 0;
    #10;

    $display("========================================");
    $display("Operacion: %s", op_name);
    $display("----------------------------------------");
    $display("Entradas:");
    $display("  a = 0x%h  (Lo: 0x%h, Hi: 0x%h)",
             a, a_lo, a_hi);
    if (operation != 2'b10)
      $display("  b = 0x%h  (Lo: 0x%h, Hi: 0x%h)",
               b, b_lo, b_hi);
    $display("Resultado:");
    $display("  res = 0x%h  (Lo: 0x%h, Hi: 0x%h)",
             res, res[15:0], res[31:16]);
    $display("========================================");
    $display("");
  endtask
```

11.3. Paso 8.1: Crear testbench personalizable (Parte 2)

Comando a ejecutar (continuacion)

```
  initial begin
    $display("");
    $display("========================================");
    $display("   TEST FP16 - Pruebas Personalizadas");
    $display("========================================");
    $display("");

    RST = 1; ENA = 0; #10;
    RST = 0; #10;

    // Prueba 1: ADD
    test_fp16(
      .a_lo(16'h3C00), .a_hi(16'h3C00),  // 1.0
      .b_lo(16'h4000), .b_hi(16'h4000),  // 2.0
      .operation(2'b00), .op_name("ADD: 1.0 + 2.0")
    );

    // Prueba 2: MUL
    test_fp16(
      .a_lo(16'h4000), .a_hi(16'h4200),  // 2.0, 3.0
      .b_lo(16'h4200), .b_hi(16'h4000),  // 3.0, 2.0
      .operation(2'b01),
      .op_name("MUL: Lo:2.0*3.0, Hi:3.0*2.0")
    );

    // Prueba 3: RCP
    test_fp16(
      .a_lo(16'h4000), .a_hi(16'h4400),  // 2.0, 4.0
      .b_lo(16'h0), .b_hi(16'h0),
      .operation(2'b10),
      .op_name("RCP: Lo:1/2.0, Hi:1/4.0")
    );

    $display("");
    $display("========================================");
    $display("      Simulacion Completada");
    $display("========================================");
    $display("");

    $finish;
  end

endmodule
EOF
```

Nota importante

El testbench permite agregar pruebas personalizadas modificando el bloque `initial` antes de la linea `$finish`.

11.4. Paso 8.2: Crear script de simulación

Comando a ejecutar

```
cat > sim_user.tcl << 'ENDTCL'
create_project -force sim_user ./sim_user \
    -part xck26-sfvc784-2LV-c

# Agregar archivos fuente
set files [list \
  rtl/custom_units/control.v \
  rtl/custom_units/diferencial5bits.v \
  rtl/custom_units/especial_cases.v \
  rtl/custom_units/fp16_add2lanes.v \
  rtl/custom_units/fp16_add_lane.v \
  rtl/custom_units/fp16_classify.v \
  rtl/custom_units/fp16_expacc.v \
  rtl/custom_units/fp16_mantmul.v \
  rtl/custom_units/fp16_mul2lanes.v \
  rtl/custom_units/fp16_mul_lane.v \
  rtl/custom_units/fp16_normround.v \
  rtl/custom_units/fp16_reciprocal_dual.v \
  rtl/custom_units/fp16_reciprocal.v \
  rtl/custom_units/fp16_special_mul.v \
  rtl/custom_units/fp16_unpack.v \
  rtl/custom_units/Manager.sv \
  rtl/custom_units/mux_2x10bits.v \
  rtl/custom_units/mux_ExpM.v \
  rtl/custom_units/shift_left_right.v \
  rtl/custom_units/shift_regis_right.v \
  rtl/custom_units/sumador_Alubig.v \
  rtl/custom_units/sumador_Alusmall.v \
]

add_files -fileset sources_1 $files

# Agregar testbench
add_files -fileset sim_1 rtl/custom_units/tb_fp16_user.sv
set_property top tb_fp16_user [get_filesets sim_1]

# Lanzar simulacion
launch_simulation
run 200ns
quit
ENDTCL
```

11.5. Paso 8.3: Ejecutar simulación

Comando a ejecutar

```
source /tools/Xilinx-2023.2/Vivado/2023.2/settings64.sh
vivado -mode batch -source sim_user.tcl 2>&1 | \
    tee resultados_fp16.log
```

Salida esperada en pantalla

```
****** Vivado v2023.2 (64-bit)
...
================================================
   TEST FP16 - Pruebas Personalizadas
================================================

================================================
Operacion: ADD: 1.0 + 2.0
------------------------------------------------
Entradas:
  a = 0x3c003c00  (Lo: 0x3c00, Hi: 0x3c00)
  b = 0x40004000  (Lo: 0x4000, Hi: 0x4000)
Resultado:
  res = 0x42004200  (Lo: 0x4200, Hi: 0x4200)
================================================

================================================
Operacion: MUL: Lo:2.0*3.0, Hi:3.0*2.0
------------------------------------------------
Entradas:
  a = 0x42004000  (Lo: 0x4000, Hi: 0x4200)
  b = 0x40004200  (Lo: 0x4200, Hi: 0x4000)
Resultado:
  res = 0x46004600  (Lo: 0x4600, Hi: 0x4600)
================================================

================================================
Operacion: RCP: Lo:1/2.0, Hi:1/4.0
------------------------------------------------
Entradas:
  a = 0x44004000  (Lo: 0x4000, Hi: 0x4400)
Resultado:
  res = 0x34003800  (Lo: 0x3800, Hi: 0x3400)
================================================

================================================
      Simulacion Completada
================================================
```

11.6. Paso 8.4: Verificar resultados

Comando a ejecutar

```
grep -A 10 "Operacion:" resultados_fp16.log
```

11.7. Paso 8.5: Validación de resultados

Los resultados de simulación se pueden validar usando la siguiente tabla:

Test	Operación	Entrada	Esperado	Estado
1	ADD	1.0 + 2.0	3.0 (0x4200)	
2	MUL	2.0 × 3.0	6.0 (0x4600)	
3	RCP	1/2.0	0.5 (0x3800)	
4	RCP	1/4.0	0.25 (0x3400)	

Cuadro 2: Validación de resultados de simulación

Verificación Exitosa

Todas las pruebas pasaron correctamente. La unidad FP16 está funcionando según lo esperado.
Cobertura funcional: 100 % de operaciones básicas verificadas.

12. Resumen Final del Proyecto

12.1. Logros Completados

Al finalizar todas las partes del proyecto, se han logrado los siguientes objetivos:

1. Instalación y configuración del toolchain RISC-V
2. Clonación del proyecto CV32E40X
3. Copia e integración de la unidad FP16 custom
4. Creación de programa de prueba con instrucciones custom
5. Compilación exitosa del código de prueba
6. Modificación del ISA con OPCODE_CUSTOM_1
7. Integración del hardware FP16 en el pipeline EX
8. **Síntesis exitosa de la unidad FP16 en FPGA**
9. **Verificación funcional mediante simulación RTL**

12.2. Archivos Generados

Documentación y Reportes:

- `fp16_utilization.txt` - Reporte de utilización de recursos
- `fp16_timing.txt` - Reporte de timing
- `resultados_fp16.log` - Log de simulación completo

Código y Scripts:

- `rtl_files_fp16.f` - Lista de archivos RTL
- `synthesize_fp16_only.tcl` - Script de síntesis
- `sim_user.tcl` - Script de simulación
- `tb_fp16_user.sv` - Testbench personalizable