Instituto Tecnológico de Buenos Aires

Departamento de Sistemas Digitales y Datos

Ingeniería Electrónica

22.15 - Electrónica V

Trabajo Práctico N°2

Procesador ERV24

Grupo N°2

CORCOS, Manuel
mcorcos@itba.edu.ar
60131

DE VINCENTI, Olivia
odevincenti@itba.edu.ar
60354

HEIR, Alejandro Nahuel
aheir@itba.edu.ar
62496

OH, Victor Christian
voh@itba.edu.ar
56679

SBRUZZI, Juan Francisco
jsbruzzi@itba.edu.ar
62517

Profesores

Mg. Ing. RODRIGUEZ, Andrés Carlos

Ing. WUNDES, Pablo

Junio - 2024

Resumen

Este trabajo describe la creación de un procesador RISC-V, ERV24, en una FPGA. El procesador incluye una arquitectura de memoria dividida, un flujo de procesamiento de cinco etapas y una predicción básica de saltos. Además, se implementaron periféricos esenciales como un puerto de entrada/salida de 8 bits y una salida de video VGA.

Índice

Índice de figuras

Lista de códigos

1. Introducción

En este trabajo práctico se desarrolla la implementación de un procesador RISC-V en una FPGA, específicamente basado en el conjunto de instrucciones RV32I. El procesador, denominado ERV24, cuenta con una arquitectura Harvard, separando la memoria de programa de la memoria de datos. Se implementó un pipeline de cinco etapas y un predictor de saltos del tipo Backwards Taken Forward Not Taken (BTFNT). Además, se incorporaron periféricos como un GPIO de 8 bits y una salida VGA con espacio de color RGB 3:3:2. También se implementaron CSRs sin privilegios como Cycle y Time.

El documento detalla la arquitectura del procesador, describiendo cada etapa del pipeline, la decodificación de instrucciones, y la predicción de saltos. Asimismo, se abordan las instrucciones de ALU, memoria, saltos y operaciones específicas como LUI y AUIPC. Se incluye un análisis del uso de periféricos y la interacción con la memoria mediante instrucciones de LOAD y STORE en tamaños de 1, 2 o 4 bytes.

Además, se presentan ejemplos de funcionamiento que ilustran la integración de periféricos, como la generación de gráficos en una pantalla VGA y la manipulación de puertos GPIO. Estos ejemplos sirven para demostrar la capacidad del procesador ERV24 para ejecutar programas complejos y manejar distintos tipos de operaciones y periféricos.

En resumen, este trabajo ofrece una visión detallada de la implementación y funcionamiento de un procesador RISC-V en FPGA, destacando su arquitectura, manejo de instrucciones y utilización de periféricos.

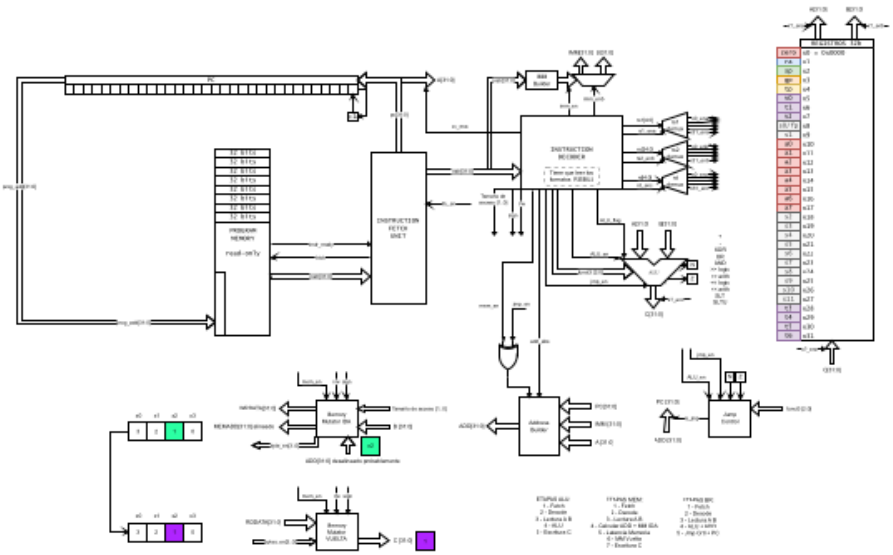

Figura 1.1: Vista previa de la organización de la CPU (drawio).

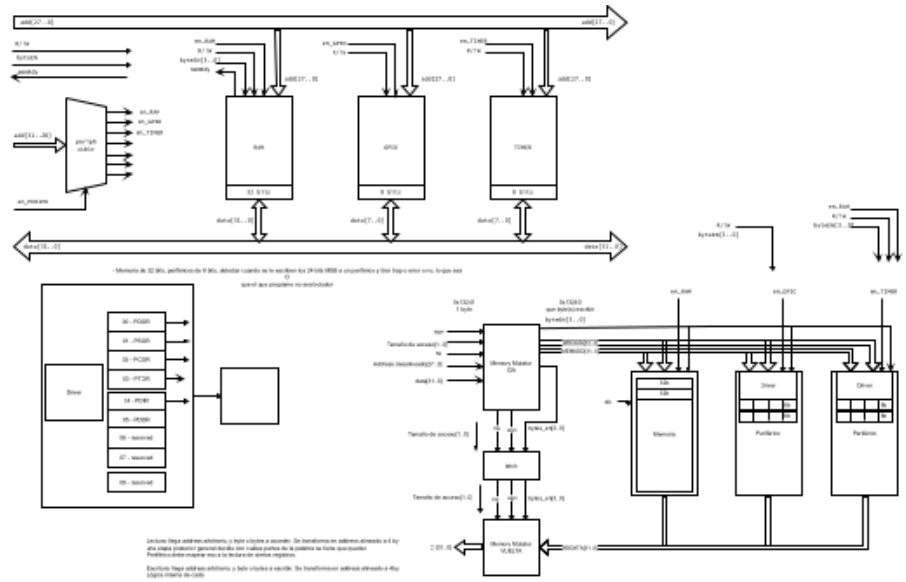

Figura 1.2: Vista previa de la organización de periféricos (drawio).

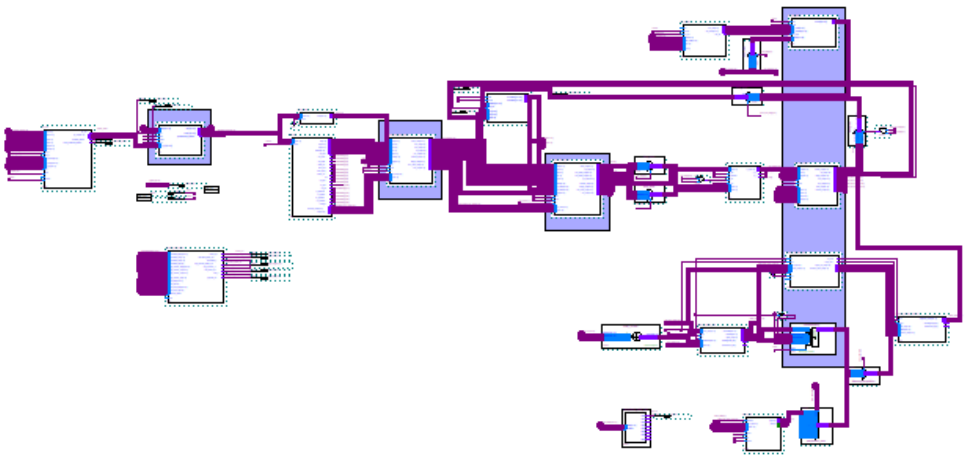

Figura 1.3: Vista previa del procesador en Quartus.

2. Arquitectura

Se implementó un esquema de 5 etapas para el data path del microprocesador, definidas como se puede apreciar en la figura 2.1.

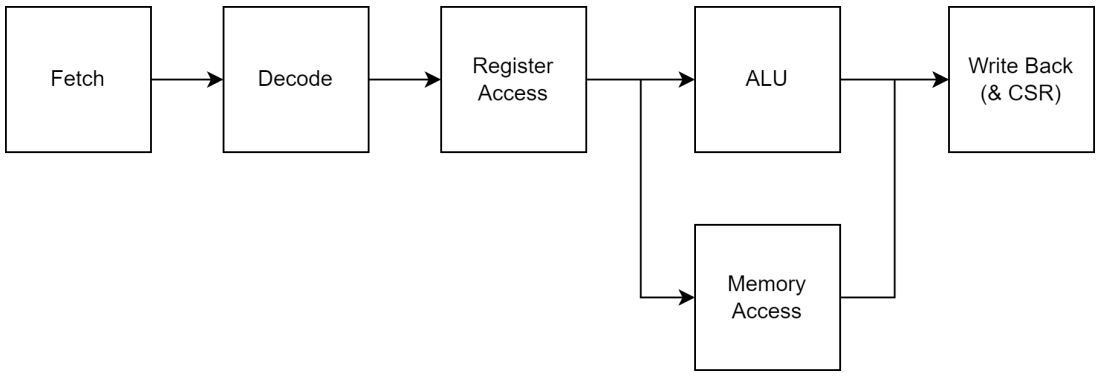

Figura 2.1: Esquema de etapas del pipeline.

2.1. IFU

2.1.1. Predicción de saltos

Se implementó la predicción de saltos mediante un bloque adicional dentro de la IFU. Este se ocupa de hacer una decodificación rápida de la instrucción para detectar si se trata de un salto incondicional (`JAL`) o un branch. Ante un JAL, inmediatamente se altera el PC actual evitando desperdiciar ciclos de clock adicionales. Ante un branch entra en juego el predictor de saltos, que conociendo el signo del dato inmediato decide si tomar el salto o no. Mediante la señal `predicted_taken` se informa al pipeline (y eventualmente al bloque `jmp_ctrl`) que la instrucción se trata de un salto que fue predicho.

Se implementó un predictor estático Backwards Taken Forward Not Taken, de manera que se toma la decisión en base únicamente al bit de signo del dato inmediato.

2.2. Decoder

El decoder interpreta las instrucciones que se encuentran en la memoria y determina las señales de control necesarias para ejecutarlas. Se implementaron todas las instrucciones del set RV32I, exceptuando las instrucciones de FENCE, ECALL y EBREAK.
A partir de los 5 MSBs del opcode se clasifica la instrucción entrante en una de las siguientes categorías:

- ALU
- LUI o AIUPC
- Memoria
- Saltos
- FENCE o SYSTEM

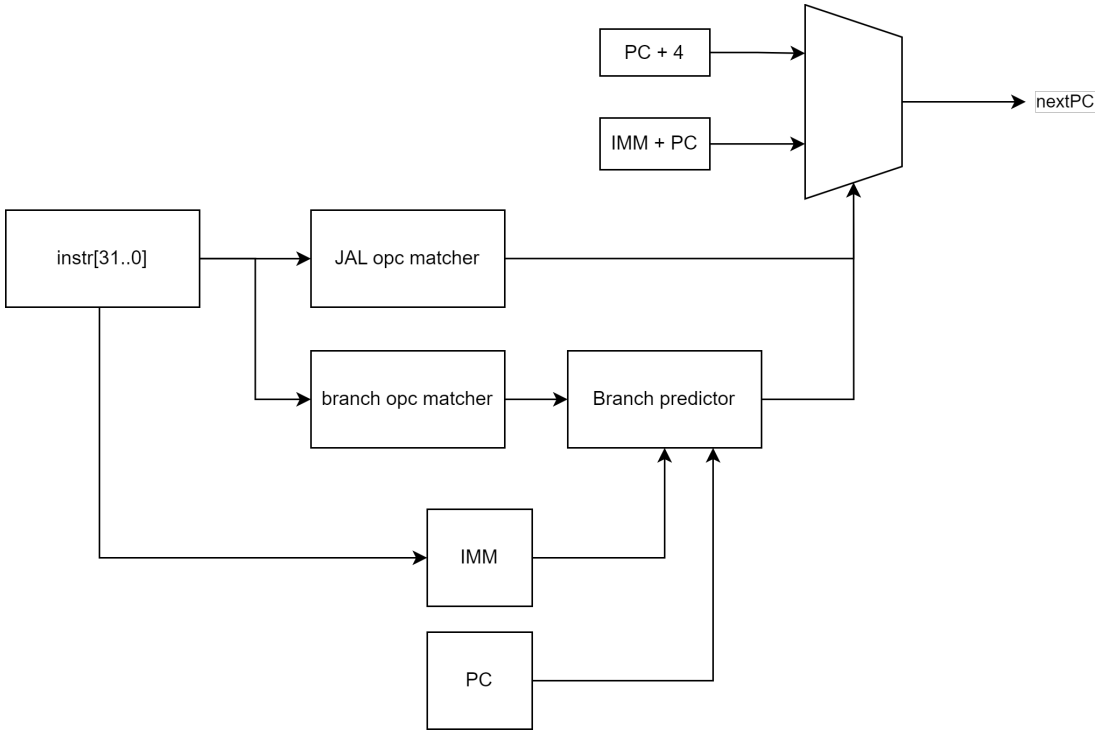

Figura 2.2: Diagrama del predictor de saltos.

Se cableó de manera fija los parámetros `rd`, `rs1`, `rs2` y `funct3` ya que siguen un esquema estructurado y se utilizan en la mayoría de las instrucciones. Según la categoría se utilizará el dato mediante señales de habilitación.

2.2.1. Instrucciones de ALU

En las operaciones de ALU se realizan cálculos numéricos, Requieren habilitar la ALU, la escritura en `rd` y la lectura de `rs1` y `rs2` o el valor `imm`. Se consideran las siguientes señales de control:

- `ALU_en`: Habilita el uso de la ALU.
- `rd_enc`: Habilita la escritura del bus C en el registro `rd`.
- `rs1_ena`: Habilita la lectura del registro `rs1` al bus A.
- `rs2_enb`: Habilita la lectura del registro `rs2` al bus B, cuando la instrucción es tipo R; y la la lectura del valor `imm` al bus B, cuando la instrucción es tipo I.
- `imm_en`: Habilita el resultado del valor `imm`, cuando la instrucción es tipo I.
- `ALU_flag`: Dado que las instrucciones SRLI y SRAI tienen el mismo opcode, al igual que ADD y SUB, se agregó una flag para indicar a la ALU cuál de las dos operaciones se debe realizar.

2.2.2. Instrucciones LUI o AIUPC

Estas instrucciones se utilizan para generar constantes, en el caso de LUI se realiza $r_d = 0+imm$ mientras que en AIUPC $rd = PC + imm$. Se consideran las siguientes señales de control:

- `ALU_en`: Habilita el uso de la ALU.
- `rd_enc`: Habilita la escritura del bus C en el registro `rd`.
- `rs1_ena`: Habilita la lectura del registro `x0` al bus A, cuando la instrucción es LUI; y habilita la lectura del PC al bus A cuando la instrucción en AIUPC.
- `rs2_enb`: Habilita la lectura del valor `imm` al bus B.
- `imm_en`: Habilita el resultado del valor `imm`.

2.2.3. Instrucciones de Memoria

Las instrucciones LOAD y STORE permiten el manejo de memoria y periféricos. La ISA considera el acceso de a 8, 16 o 32 bits que pueden ser signados y el address se calcula como $add = rs1 + imm$. Se consideran las siguientes señales de control:

- `mem_en`: Habilita el acceso a memoria.
- `rs1_ena`: Habilita la lectura del registro `rs1` al bus A, que se utilizará para calcular el address.
- `imm_en`: Habilita el resultado del valor `imm` al bus IMM, que se utilizará para calcular el address.
- `rd_enc`: Habilita la escritura del bus C en el registro `rd`, cuando la instrucción es LOAD.
- `rs2_enb`: Habilita la lectura del valor `rs2` al bus B, cuando la instrucción es STORE.
- `rw`: Indica si se realizará una lectura o una escritura. Será 1 si la instrucción es LOAD, 0 si es STORE.
- `unsign`: Indica si se debe realizar extensión de signo. Será 1 si la instrucción es LHU, LBU, SHU o SBU.
- `access_size`: Indica si el acceso a memoria será de a 8, 16 o 32 bits.

2.3. Instrucciones de Saltos

La instrucciones de saltos engloban los saltos condicionales e incondicionales, es decir, JAL, JALR y BRANCH. El address se calcula como $add = PC+imm$ en JAL y los BRANCH, mientras que se usa $add = rs1 + imm$ en JALR. Se consideran las siguientes señales de control:

- `is_jmp`: Indica que la instrucción es una de salto.
- `is_jal`: Indica que la instrucción es un salto absoluto incondicional (JAL).
- `is_jal`: Indica que la instrucción es un salto relativo incondicional (JALR).

- `is_branch`: Indica que la instrucción es un salto condicional (BRANCH).
- `imm_en`: Habilita el resultado del valor `imm` al bus IMM, que se utilizará para calcular el address.
- `rs1_ena`: Habilita la lectura del registro `rs1` al bus A, que se utilizará para calcular el address si la instrucción es JALR y para comparar si es BRANCH.
- `rd_enc`: Habilita la escritura del bus C en el registro `rd`, si la instrucción es JAL o JALR.
- `rs2_enb`: Habilita la lectura del valor `rs2` al bus B, si la instrucción es BRANCH.
- `ALU_en`: Habilita el uso de la ALU si es BRANCH.

2.3.1. Instrucciones de FENCE o SYSTEM

De esta categoría, sólo están implementadas las instrucciones de CSR, aunque se proveen flags que permiten identificar las demás instrucciones de este caso. Se consideran las siguientes señales de control:

- `is_fence`: Indica que la instrucción es FENCE.
- `is_system`: Indica que la instrucción es SYSTEM.
- `rd_enc`: Habilita la escritura del bus C en el registro `rd`, si la instrucción es de CSR.
- `rs1_ena`: Habilita la lectura del registro `rs1` al bus A, que se utilizará escribir si la instrucción de CSR es relativa.
- `imm_en`: Habilita el resultado del valor `imm` al bus IMM, que se utilizará escribir si la instrucción de CSR es inmediata.
- `csr`: Address del registro CSR a acceder.

Además se implementó la flag `is_invalid` que se prende al recibirse una instrucción inválida.

El primer latch del pipeline es `decode_imm_stage_latch`, que propaga los flags del decoder y el dato del `imm_builder`, que construye el dato inmediato en simultáneo con la decodificación.

2.4. Lectura de operandos

El segundo latch del pipeline es `op_read_stage_latch`, que propaga los flags de la etapa anterior y recibe la salida del banco de registros (`rs1` y `rs2`). Durante este ciclo de clock entonces se estarán leyendo los datos necesarios para operar.

2.5. ALU, acceso a memoria, y CSRs

2.5.1. ALU

Se implementó el módulo de la ALU en VHDL. Las entradas son los bits de selección de operación, los buses de 32-bits de los operandos, una señal de indicación si se debe calcular un salto, y una señal de habilitación. Las salidas del módulo son el bus de 32-bits del resultado de la operación seleccionada y señales de indicación si el resultado es cero y si el resultado es negativo.

Las operaciones implementadas son las que indica el set de instrucciones:

- Suma
- Resta
- Shift izquierda
- Shift derecha lógico (no conserva el signo)
- Shift derecha aritmético (conserva el signo)
- Comparación lógica AND
- Comparación lógica OR
- Comparación lógica XOR
- Comparación aritmética signada (SLT) (devuelve 1 si la comparación signada de A $<$ B es cierta. Devuelve 0 en caso contrario)
- Comparación aritmética no signada (SLTU) (devuelve 1 si la comparación no signada de A $<$ B es cierta. Devuelve 0 en caso contrario)

El tercer latch del pipeline en este camino es `alu_stage_latch`, que propaga los flags de la etapa anterior y recibe la salida de la ALU.

2.5.2. Acceso a memoria y periféricos

En el caso de un acceso a memoria o periférico, se tiene un camino alternativo a la ALU donde se pasa por el `Memory Mutator (ida)` para generar la dirección a leer. En estos casos el tercer latch encontrado es `memory_mutator_latch`, triggereado en simultáneo con la lectura de la memoria.

El acceso a memoria o periféricos se logra por igual mediante instrucciones de `LOAD` o `STORE`, mapeando cada dispositivo a una región de memoria distinta, por lo que la selección de uno u otro depende directamente del address que se envíe. Todos los dispositivos comparten las líneas de control (excepto el enable individual de cada uno) y buses de data (31:0) y address (31:0).

Dado que RISC-V especifica que los accesos a memoria son de a byte (las direcciones de memoria hacen referencia a regiones de 8 bits), y dado que tanto la RAM como los periféricos se implementaron como registros de 32 bits, para lograr accesos 8, 16 o 32 bits, entre los periféricos per se y el procesador se intercalan los módulos `memory_mutator_IDA` y `memory_mutator_VUELTA`, que se encargan de hacer el mapeo/conversión de direcionamientos de 8 bits con accesos de 8, 16 o 32 bits (como viene de las instrucciones), hacia un formato universal para RAM y demás periféricos, que consiste en un address de 32 bits y una línea de *byte enable (3:0)* que indica cuál o cuáles de todos los bytes de una palabra se pretenden acceder. Luego, cada dispositivo responde adecuadamente a la petición que le llega, ya sea modificando sus registros internos o volcando alguno/s de ellos sobre el bus de data de salida. En la figura 2.3 se presenta una esquematización de este flujo.

Además de encargarse de la adaptación de address y tamaño de acceso, los `memory_mutator` se encargan de aplicar extensión de signo a datos leídos (el `VUELTA` en particular).

La razón de tener tener uno un módulo de *ida* y otro de *vuelta* es dar soporte al pipeline cuando los accesos a periféricos son lentos y debe esperarse más de un ciclo a que tomen efecto las acciones. Si bien en la implementación realizada estas demoras no ocurren, pues tanto la

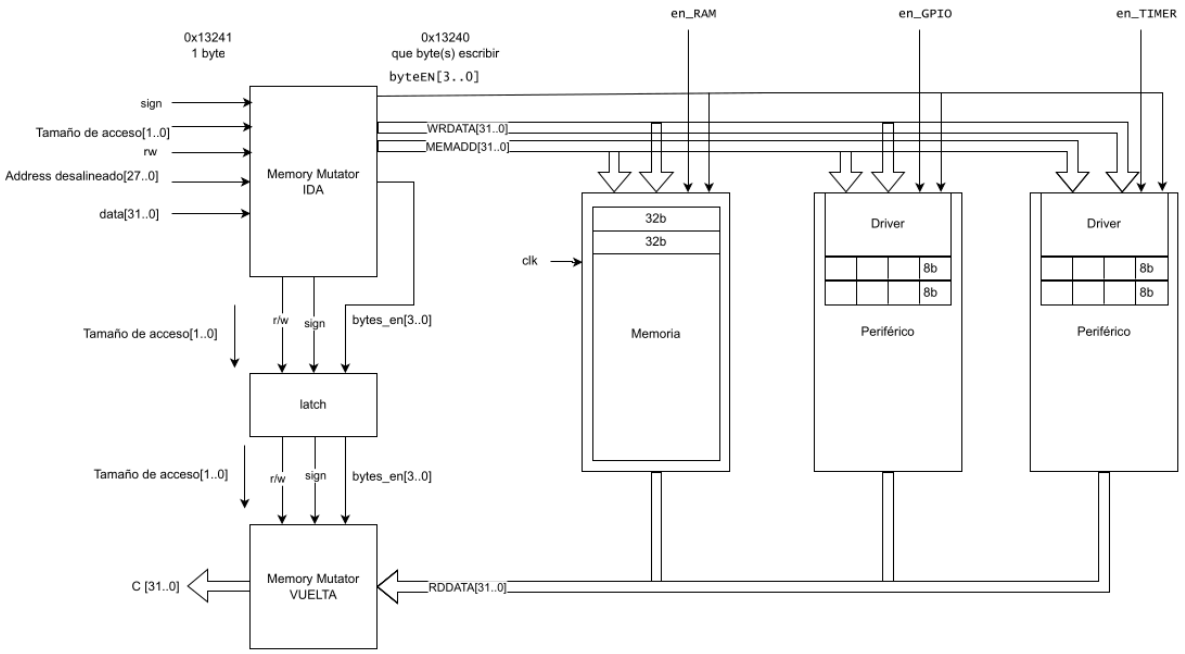

Figura 2.3: Diagrama-planificación del acceso a memoria y periféricos.

RAM como los periféricos se realizaron con registros internos (SRAM), se adoptó igualmente este diseño proyectando la inclusión de dispositivos más lentos.

En las figuras 2.4, 2.5 y 2.6 se presentan las implementaciones de los *memory mutator* y su correspondiente latch intermedio en Quartus.

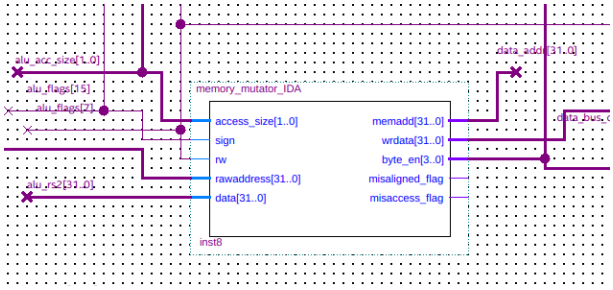

Figura 2.4: `memory_mutator_IDA` en Quartus.

2.6. Escritura de resultados

Los resultados de las operaciones pueden ser almacenados en dos lugares, a partir de la etapa `alu_stage_latch`:

- En el banco de registros, en el caso de la mayoría de las instrucciones será a partir de un resultado de la ALU, pero también puede deberse a una lectura de memoria (o periférico mapeado a memoria) o CSR.
- En el banco de CSR, siendo el dato almacenado `rs1` o `imm`.

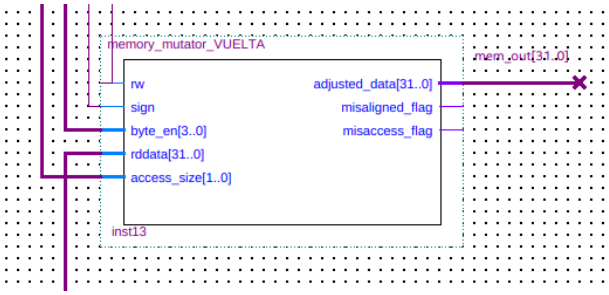

Figura 2.5: `memory_mutator_VUELTA` en Quartus.

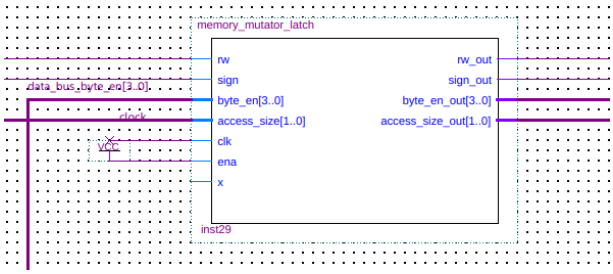

Figura 2.6: `memory_mutator_latch` en Quartus.

La selección de cada caso se realiza mediante los flags que acompañan la instrucción, en particular entran en juego `rs1_ena`, `mem_en`, `rw` (de memoria), e `is_system`. Ya que la lectura del CSR debe poder escribirse en el banco de registros, esta se triggerea en simultáneo a la etapa `alu_stage_latch`.

2.7. Control de saltos

El control de saltos consiste en un trabajo conjunto de la IFU, el predictor de saltos, y el bloque `jmp_ctrl` que de cierta manera maneja la IFU.

Este bloque tiene dos tareas principales:

- Ante un `JALR`, recibe el resultado del acceso a registros (en particular `rs1`) para formar el nuevo PC como `rs1 + imm` e indicarlo a la IFU.
- Ante un branch, tomando los flags de la ALU y conociendo si fue predicho en la IFU o no, define si se debe:
  - Restaurar el PC al valor `PC + 4`, si se predice como tomado un branch que resultó que no se debía tomar.
  - Hacer el salto pasando el PC al valor `PC + imm`, si se predijo que no se tomaría un salto que finalmente fue tomado.

Ante una predicción errónea, el bloque `jmp_ctrl` indica a la IFU el PC correcto, y el pipeline tiene la inteligencia para descartar los efectos de cualquier instrucción incorrecta que haya sido procesada gracias a una predicción errónea.

3. Periféricos y extras

3.1. GPIO

Se implementó un GPIO simple de 8 bits, inspirado en el microcontrolador K64F de NXP. Cuenta con los registros `PDOR` (Port Data Output Register, permite escribir 0 o 1 en cada pin), `PSOR` (Port Set Output Register, permite poner a 1 cada pin), `PCOR` (Port Clear Output Register, permite poner a 0 cada pin), `PTOR` (Port Toggle Output Register, permite invertir el estado de cada pin), `PDIR` (Port Data Input Register, permite leer el estado del pin), y `PDDR` (Port Data Direction Register, permite definir si cada pin es una entrada o una salida).

En el *Pin Planner* de Quartus, se asignó los 8 bits de GPIO a los 8 LEDs de la placa DE0-Nano.

3.2. Módulo VGA y graficador

Se implementó un periférico para enviar gráficos a un monitor VGA. Este periférico consiste de un módulo que genera las señales de sincronización para el protocolo VGA de la figura 3.1, un módulo generador de datos y un conversor digital analógico para generar las tensiones requeridas por el protocolo VGA.

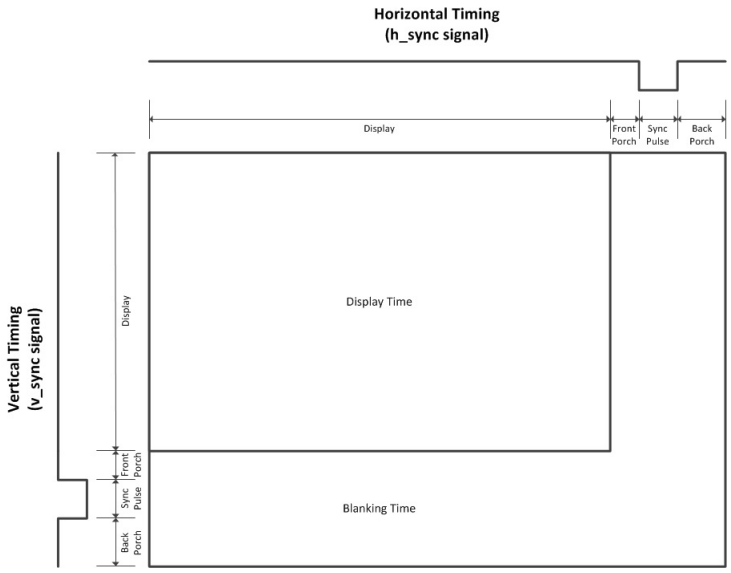

Figura 3.1: Diagrama de tiempos de VGA

3.2.1. Sincronizador VGA

Se configuró el generador para cumplir con los tiempos de pulsos necesarios en las señales de sincronización para obtener una resolución de 640 × 480px a 75 Hz de refresco, como se observa en la figura 3.1. La información fue obtenida de http://tinyvga.com/vga-timing/640x480@75Hz. Además genera posición en la pantalla del píxel a evaluar para determinar el color a la salida y avisa si se encuentra dentro del cuadro visible o no.

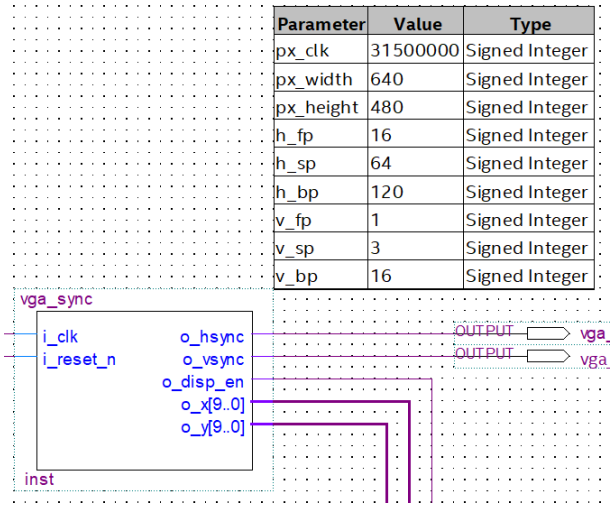

Figura 3.2: Bloque generador de señales de sincronización de VGA

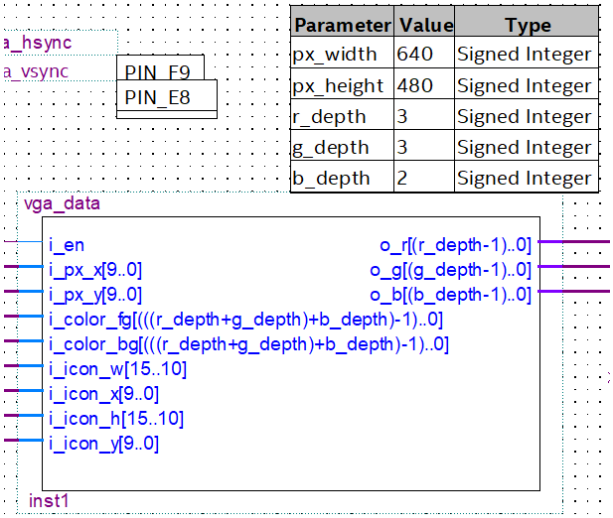

Figura 3.3: Bloque generador de gráficos

3.2.2. Generador de gráficos

El bloque de la figura 3.3 dibuja un rectángulo sobre la pantalla. Se puede configurar el ancho, alto, color, posición vertical y posición horizontal del rectángulo y el color de fondo de la pantalla.

Los colores se encuentran en el espacio RGB 3:3:2, y por lo tanto la entrada de colores del generador de imágenes es de 8 bits que pueden ser configurados en software.

Para alinear y comprimir la cantidad de conexiones, se utilizaron 6 bits para el ancho del rectángulo y 10 bits para la posición horizontal, compartiendo un registro de 16 bits. De la misma manera está estructurada la altura y la posición vertical del rectángulo.

La lógica del módulo es que si el píxel a dibujar pertenece al rectángulo, el color a la salida del módulo corresponderá al color del rectángulo, y cuando no pertenece al rectángulo, el color a la salida será el color de fondo.

3.2.3. Conversor digital analógico

El conversor digital analógico de la figura 3.4 se construyó para convertir la señal digital de color a la salida a las señales analógicas necesarias para el protocolo VGA. Como la intensidad de cada color RGB está determinada por una tensión entre $0 - 0.7V$, y la placa recibe 3 entradas digitales por cada color, habrá que convertir una entrada digital de $0 - 7$ a las tensiones correspondientes. Esto se logró combinando resistencias de forma que su combinación entre pines a 3.3V o GND genere los 8 niveles tensión del espacio de color de la señal VGA. Los resistores utilizados son los que se observan en la figura 3.5, y la resistencia de 75Ω corresponde a la impedancia de entrada del monitor. Como HSYNC y VSYNC son señales digitales no es necesaria este acondicionamiento.

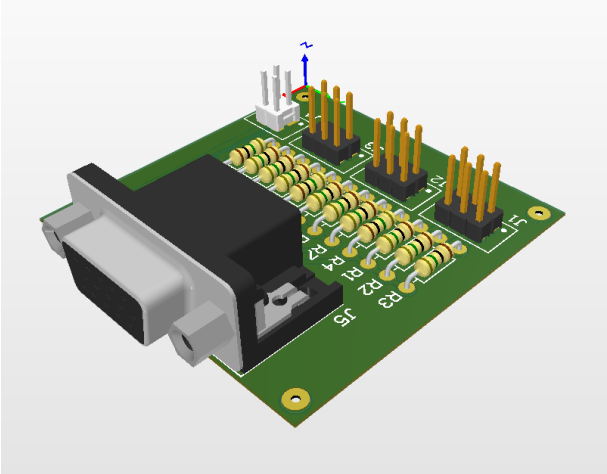

Figura 3.4: Conversor digital analógico para VGA

Como el espacio de color de salida de la FPGA es de 8 bits, el bit B0 se configuró en copia al bit B1, o a GND para respetar el blanqueo.

Se encontró que la tensión no alcanza la simulada requerida. Esto puede ser por un problema de adaptación de impedancia o de corrientes de salida de la propia FPGA, por lo cual en otra revisión de esta placa puede mejorarse este aspecto utilizando drivers externos para alcanzar la corriente necesaria.

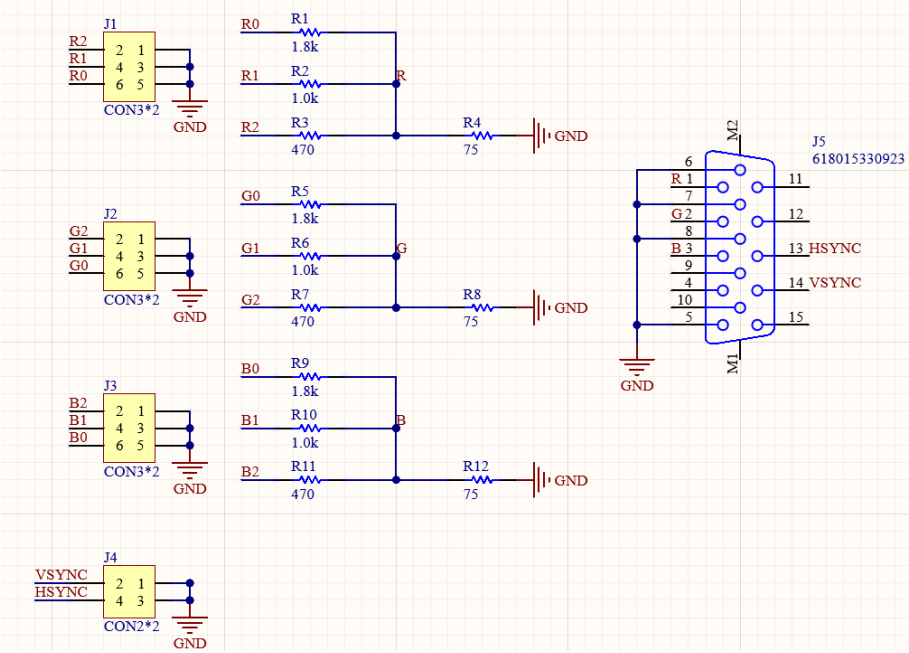

Figura 3.5: Esquemático del conversor

3.2.4. Adaptación como periférico

Para acceder a las funcionalidades de VGA descriptas como un periférico más, el módulo `vga_peripheral_adapter` es responsable de convertir los interacciones con los buses de periférico en las señales de VGA requeridas. En la figura 3.6 se observa su bloque en Quartus.

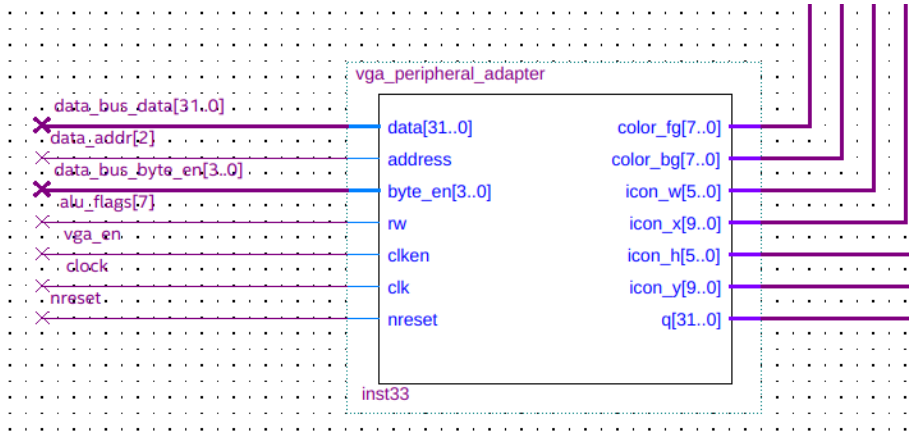

Figura 3.6: Bloque-módulo `vga_peripheral_adapter`.

3.2.5. En funcionamiento

En la figura 3.7 se muestra la conexión física entre la placa DE0-Nano y el DAC de VGA, y en la figura 3.8 se observa en funcionamiento el sistema de gráficos detallado anteriormente, en donde se destaca el barrido horizontal de pixeles como un contador ascendente.

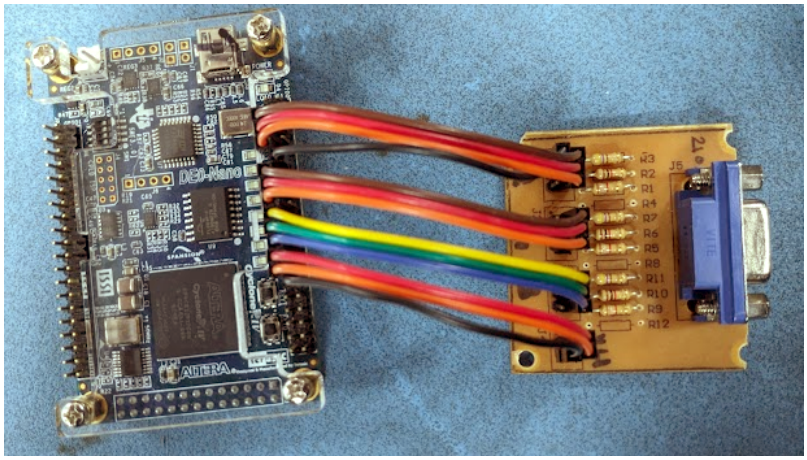

Figura 3.7: DE0-Nano conectado a DAC de VGA.

Figura 3.8: Captura de imagen generada

Se verificó las señales de entrada del módulo de VGA con el *Signal Tap Logic Analyzer*. Referencias: `https://www.youtube.com/watch?v=vsEXs5insJI`

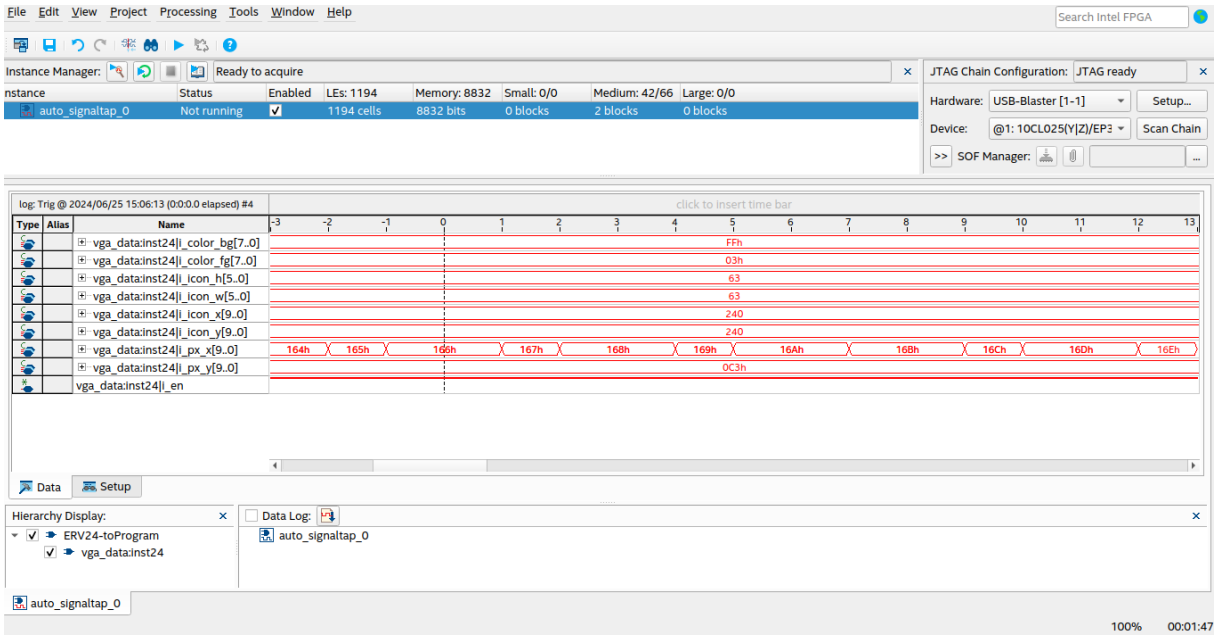

Figura 3.9: Captura de pantalla del *Signal Tap Logic Analyzer* para VGA

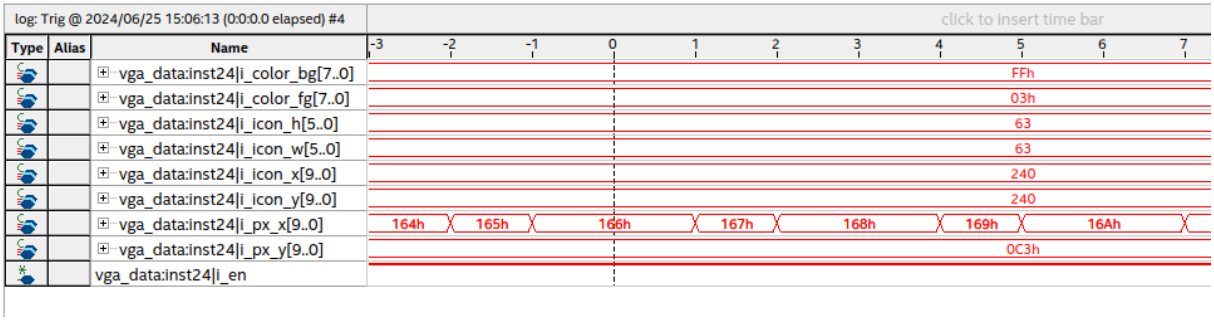

Figura 3.10: Captura de pantalla del *Signal Tap Logic Analyzer* para VGA, detalle.

4. Generación de código ejecutable

4.1. Herramientas online

Con el objetivo de generar código ejecutable por el procesador, sin escribir instrucción por instrucción a nivel bit, se disponen de variadas herramientas online que permiten convertir instrucciones en formato opcode a su equivalente hexadecimal para cargar en el archivo `.hex` de la memoria de programa del procesador.

Una herramienta de utilidad en un comienzo, en las etapas de diseño de la arquitectura principal, es el conversor `https://luplab.gitlab.io/rvcodecjs/`, que permite ingresar un instrucción en formato opcode y obtener su equivalente hexadecimal, o viceversa. Se observa su interfaz en la imagen 4.1.

Figura 4.1: Herramienta online de LupLab para convertir opcode a hex.

Otra herramienta destacable es `https://riscvasm.lucasteske.dev/`, un compilador online de assembly para RISC-V, que permite escribir código assembly, vincularlo con un script de linker (para la gestión de zonas de memorias), generar el hex de salidad y visualizar un disassembly del código. En la imagen 4.2 se observa su interfaz.

4.2. GNU toolchain for RISC-V

Para no depender de herramientas online y, principalmente, poder generar código de manera más universal y modular, se dispone del toolchain de GNU específico para RISC-V en sus varias extensiones, para compilar código C/C++. Su repositorio principal que incluye los pasos para instalarlo es `https://github.com/riscv-collab/riscv-gnu-toolchain`.

Para forzar que el toolchain solo soporte la extensión base del procesador, RV32I, se debe indicar la configuración detallada debajo al momento de la instalación, lo cual evita generar instrucciones no soportadas como aquellas para otras extensiones y las versiones comprimidas (16 bits) de las mismas.

```
./configure --prefix=/opt/riscv --with-arch=rv32i --with-abi=ilp32
```

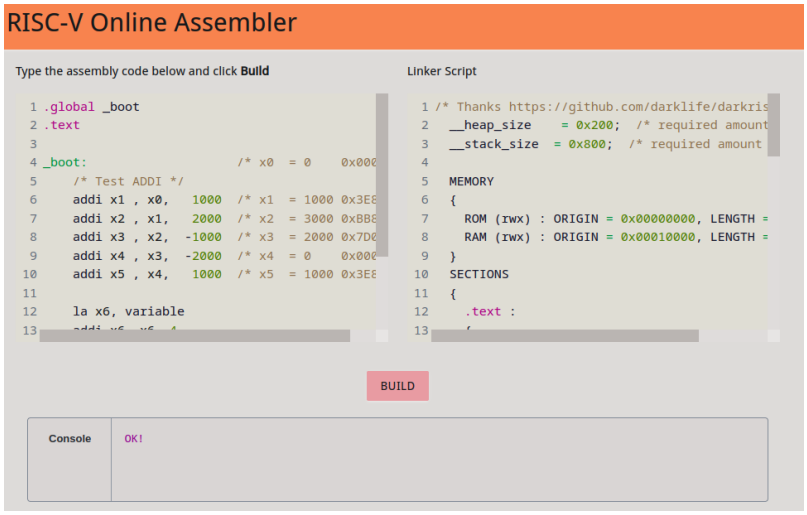

Figura 4.2: Compilador online de assembly de RISC-V.

En los apéndices B y C se adjuntan templates de proyectos en assembly y C, basados en el toolchain de RISC-V, respectivamente.

5. Ejemplos de funcionamiento

5.1. Fibonacci + GPIO

Se probó el siguiente programa, en el que se tiene un bucle de muchas repeticiones luego del cual se calcula el siguiente número de la serie de Fibonacci y se asigna al puerto GPIO de 8 bits, para luego repitir el proceso.

Sobre la FPGA física se verificó el funcionamiento adecuadamente. Para testearlo sobre Quartus se redujo la cantidad de bucles (usados solo para permitir apreciar el cálculo en la vida real) a 10.

A continuación se detalla el código sobre el cual se trabajó:

```
1  00000000 <_boot>:
2     0: 400002b7            lui t0,0x40000
3     4: 0ff00313            li  t1,255
4     8: 006282a3            sb  t1,5(t0) # 40000005 <GPIO_BASE+0x3ffc0005>
5     c: 00028023            sb  zero,0(t0)
6    10: 00100e13            li  t3,1
7    14: 00018eb7            lui t4,0x18
8    18: 6a0e8e93            addi  t4,t4,1696 # 186a0 <DELAY>
9    1c: 00100f13            li  t5,1
10   20: 00100513            li  a0,1
11
12 00000024 <loop>:
13   24: 001e0e13            addi  t3,t3,1
14   28: ffde1ee3            bne t3,t4,24 <loop>
15   2c: 00a28023            sb  a0,0(t0)
16   30: 00af05b3            add a1,t5,a0
17   34: 00050f13            mv  t5,a0
18   38: 00058513            mv  a0,a1
19   3c: 00100e13            li  t3,1
20   40: fe5ff06f            j 24 <loop>
```

Código 1: Assembly del ejemplo Fibonacci + GPIO

Las características notables de este programa son:

- Hay dependencias directas de instrucciones sucesivas (como por ejemplo las de las líneas 17 y 18 o las 13 y 14), por lo que deberá actuar el pipeline.
- Hay saltos incondicionales y branches.
- Se trabaja con la memoria para el GPIO.

La parte inicial del programa es la configuración, se carga la dirección de los GPIO como salidas (líneas 3 y 4), y se carga la cantidad de bucles que se utilizan para hacer el delay (líneas 7, 8).

Se puede apreciar el funcionamiento simulado del microprocesador en las siguientes capturas, donde se destacaron como señales el pin de habilitación de cada etapa del pipeline, el puerto GPIO de 8 bits, y las señales de control del acceso y escritura a registros.

A grandes rasgos en la simulación de la figura 5.1 se puede apreciar el funcionamiento adecuado del procesador. Efectivamente el GPIO está siguiendo la serie de Fibonacci, con lo que los store en memoria están efectivamente comunicándose con el periférico de GPIO. Se puede destacar en la línea de `pipe_jmp_en` la presencia de 9 pulsos que indican la activación del bloque de control de saltos, que revisa que la predicción haya sido tomada o no según corresponda. Se tienen 9

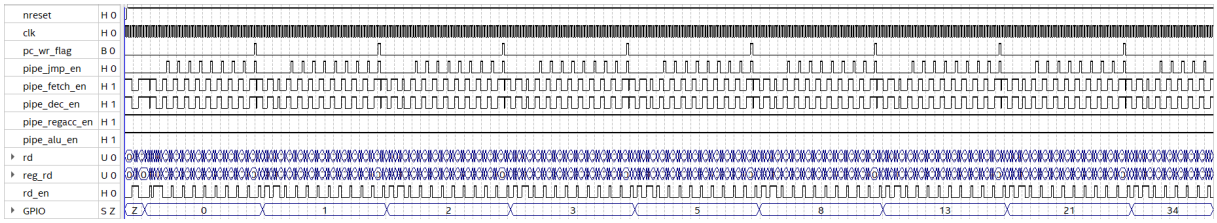

Figura 5.1: Captura de simulación del programa.

pulsos debido a que en la línea 19 se carga con 1 el acumulador. En cada uno de los pulsos la línea de escritura del PC no se acciona, a excepción de justamente el último ya que no se debe tomar el salto hacia atrás.

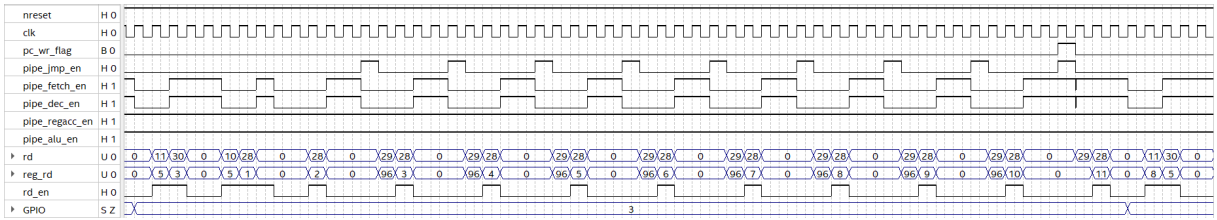

Figura 5.2: Detalle de la simulación del programa

Con mayor detalle, en la figura 5.2 se resalta como el pipeline frena las etapas de fetch y decoding para resolver las dependencias de registros entre instrucciones. Además se puede notar el registro en el que se almacenan los resultados cuando la señal `rd_en` se acciona, y el valor almacenado en `reg_rd`. De esta manera, se ve como el acumulador va sumando de a una unidad hasta llegar a 10, momento en el que se toma el branch, y luego se almacenan valores intermedios para el cálculo de la serie.

5.2. VGA + GPIO

```
1  .globl _boot
2  .section .text
3
4  .equ GPIO_BASE, 0x40000
5  .equ GPIO_OFF_PDOR, 0x0
6  .equ GPIO_OFF_PSOR, 0x1
7  .equ GPIO_OFF_PCOR, 0x2
8  .equ GPIO_OFF_PTOR, 0x3
9  .equ GPIO_OFF_PDIR, 0x4
10 .equ GPIO_OFF_PDDR, 0x5
11
12 .equ VGA_BASE, 0x50000
13 .equ VGA_OFF_C_FG, 0x0
14 .equ VGA_OFF_C_BG, 0x1
15 .equ VGA_OFF_I_WX, 0x2
16 .equ VGA_OFF_I_HY, 0x4
17
18 _boot:
19     lui     t0, GPIO_BASE
20
21     addi    t1, zero, 0xFF
```

```
22      addi    t2, zero, 0x55
23      sb      t1, GPIO_OFF_PDDR(t0)
24      sb      t2, GPIO_OFF_PDOR(t0)
25
26      lui     t2, VGA_BASE
27
28      li      t3, 0xFCF0FFF0
29      sw      t3, VGA_OFF_C_FG(t2)
30
31      li      t3, 0xFCF0
32      sh      t3, VGA_OFF_I_HY(t2)
33
34      lb      t4, VGA_OFF_C_FG(t2)
35      sb      t4, GPIO_OFF_PDOR(t0)
36
37  loop:   j   loop
38
39  .section .data
```

Código 2: Assembly del ejemplo VGA + GPIO

```
1
2   start.elf:     file format elf32-littleriscv
3
4
5   Disassembly of section .text:
6
7   00000000 <_boot>:
8      0: 400002b7            lui t0,0x40000
9      4: 0ff00313            li  t1,255
10     8: 05500393            li  t2,85
11     c: 006282a3            sb  t1,5(t0) # 40000005 <VGA_BASE+0x3ffb0005>
12    10: 00728023            sb  t2,0(t0)
13    14: 500003b7            lui t2,0x50000
14    18: fcf10e37            lui t3,0xfcf10
15    1c: ff0e0e13            addi  t3,t3,-16 # fcf0fff0 <VGA_BASE+0xfcebfff0>
16    20: 01c3a023            sw  t3,0(t2) # 50000000 <VGA_BASE+0x4ffb0000>
17    24: 00010e37            lui t3,0x10
18    28: cf0e0e13            addi  t3,t3,-784 # fcf0 <__stack_size+0xf4f0>
19    2c: 01c39223            sh  t3,4(t2)
20    30: 00038e83            lb  t4,0(t2)
21    34: 01d28023            sb  t4,0(t0)
22
23  00000038 <loop>:
24    38: 0000006f            j 38 <loop>
```

Código 3: Disassembly del ejemplo VGA + GPIO

Apéndice A Simulaciones varias

Se adjuntan simulaciones en Quartus de algunos de los bloques principales del procesador, que no fueron incluídos en el cuerpo del informe.

A.1 IFU

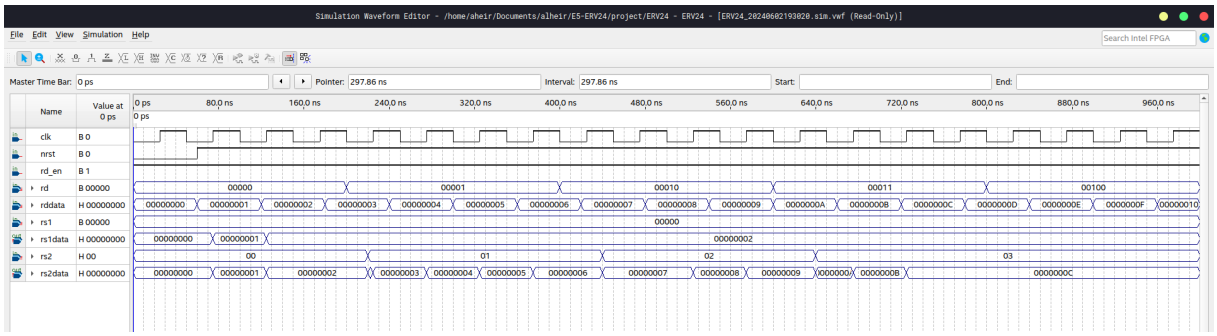

Figura A.1: Simulación de operación de la IFU.

A.2 Banco de registros

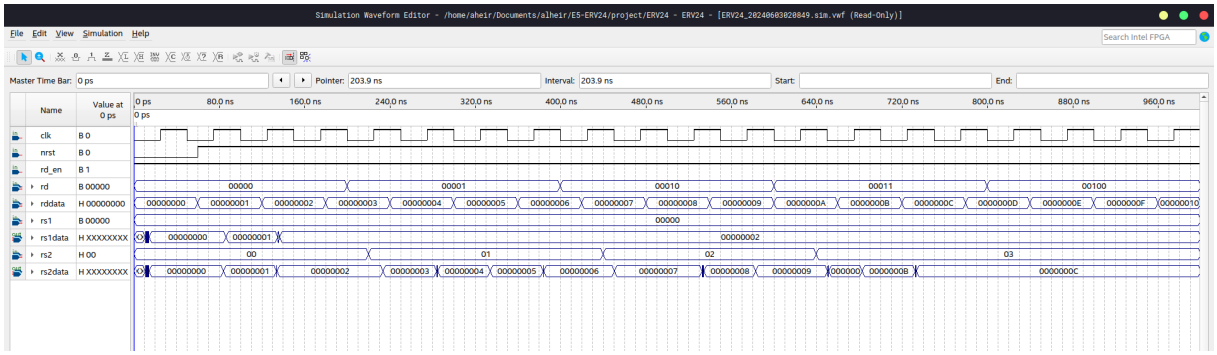

Figura A.2: Simulación de accesos a banco de registros.

A.3 Timings de RAM

Se realizó simulación de tiempos de respuesta de la IP de RAM, sin latch de salida, ya comunicándose con el `memory_mutator_IDA`.

1. Se escribe un byte con 0x08 en dirección 2.
2. Se lee un byte en dirección 2.
3. Sale palabra de memoria
4. Sale byte con 0x08 por mutator_VUELTA

- Unos $0.4ns$ entre que sale algo de memoria y se sale procesado en el mutator $(4-3)$.

- Unos $0.8ns$ entre que se lee algo y sale por memoria $(3-2)$.
- Unos $1.3ns$ entre que se lee algo y sale procesado en el mutator $(4-2)$.

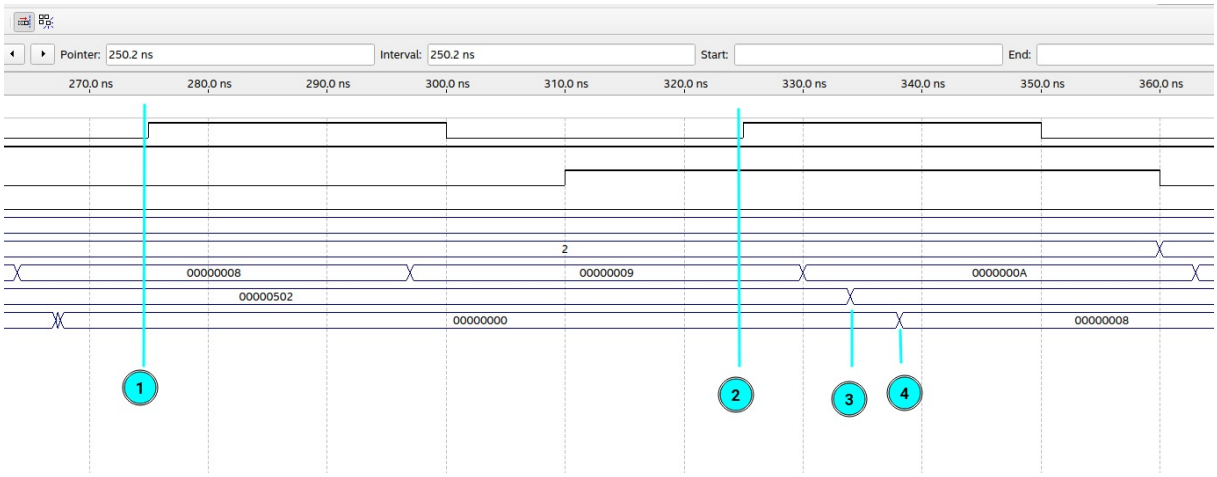

Figura A.3: Simulación de tiempos de acceso de RAM.

A.4 *Memory mutators*

En las figuras A.4 y A.5 se muestran simulaciones realizadas para los *memory mutators*, resaltando distintas situaciones de acceso de escritura/lectura.

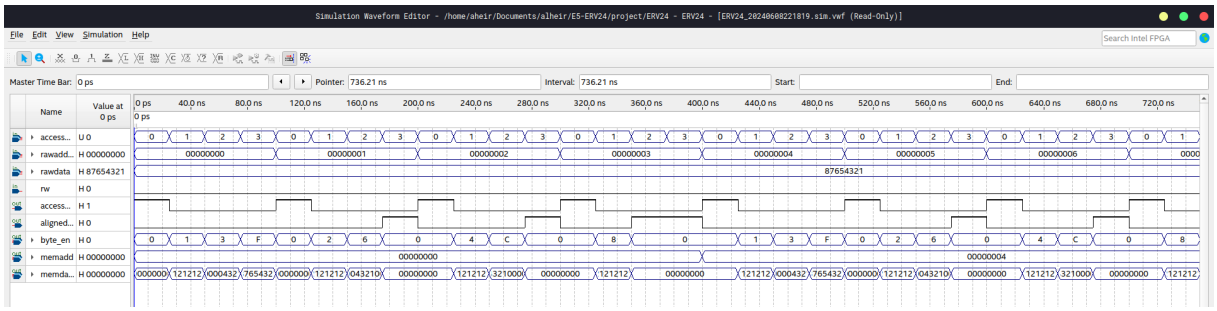

Figura A.4: Simulación de `memory_mutator_IDA`.

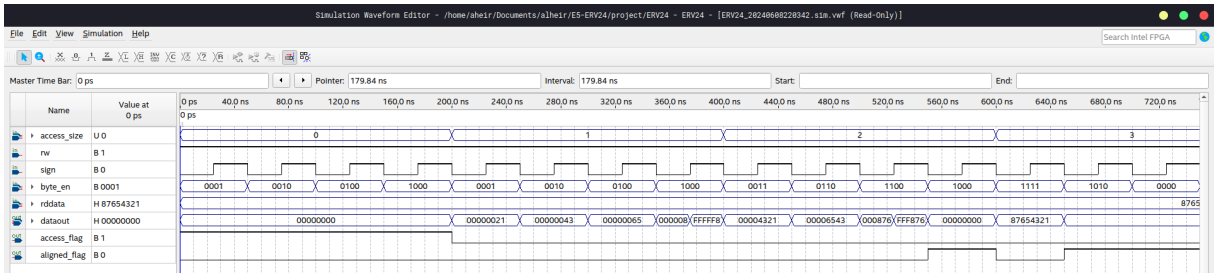

Figura A.5: Simulación de `memory_mutator_VUELTA`.

Apéndice B Template proyecto en assembly

El proyecto base en assembly consiste en 3 archivos:

- `start.S`, código principal
- `loader.ld`, script del linker. Se debe adaptar según implementación y capacidades de memoria.
- `Makefile`

Referencias:

- `https://github.com/darklife/darkriscv`
- `https://riscvasm.lucasteske.dev/`

```
.globl _boot
.section .text

_boot:
    lui     t0, 0xFF

    # ...

loop:   j   loop

.section .data
```

Código 4: `start.S` del template de assembly

```
/* Thanks https://github.com/darklife/darkriscv */
__heap_size     = 0x200;  /* required amount of heap */
__stack_size    = 0x800;  /* required amount of stack */

MEMORY
{
    ROM (rwx) : ORIGIN = 0x00000000, LENGTH = 0x02000
    RAM (rwx) : ORIGIN = 0x00010000, LENGTH = 0x02000
}
SECTIONS
{
    .text :
    {
        *(.text)
        *(.rodata*)
    } > ROM

    .data :
    {

        *(.data)
        *(.bss)
        *(.sbss)
        *(COMMON)
    } > RAM

    .heap :
    {
        . = ALIGN(4);
        PROVIDE ( end = . );
        _sheap = .;
        . = . + __heap_size;
```

```
        . = ALIGN(4);
        _eheap = .;
    } >RAM

    .stack :
    {
        . = ALIGN(4);
        _estack = .;
        . = . + __stack_size;
        . = ALIGN(4);
        _sstack = .;
    } >RAM
}
```

Código 5: `loader.ld` del template de assembly

```
# Nombre del archivo de ensamblaje
ASM_FILE = start.S
# Nombre del archivo objeto
OBJ_FILE = start.o
# Nombre del archivo ELF
ELF_FILE = start.elf
# Nombre del archivo binario
BIN_FILE = start.bin
# Nombre del archivo hex
HEX_FILE = start.hex
# Nombre del archivo de desensamblado
DISASM_FILE = start.dis
# Archivo de script de vinculacion
LINKER_SCRIPT = loader.ld

# Comandos de compilacion
AS = riscv32-unknown-elf-as
LD = riscv32-unknown-elf-ld
OBJCOPY = riscv32-unknown-elf-objcopy
OBJDUMP = riscv32-unknown-elf-objdump
HEXDUMP = hexdump

# Opciones de compilacion y vinculacion
ASFLAGS =
LDFLAGS = -T $(LINKER_SCRIPT) -m elf32lriscv -nostdlib --no-relax
OBJCOPYFLAGS = -O binary
OBJDUMPFLAGS = -d

# Regla por defecto
all: $(HEX_FILE) $(DISASM_FILE)

# Regla para generar el archivo objeto
$(OBJ_FILE): $(ASM_FILE)
  $(AS) $(ASFLAGS) $< -o $@

# Regla para generar el archivo ELF
$(ELF_FILE): $(OBJ_FILE)
  $(LD) $(LDFLAGS) $< -o $@

# Regla para generar el archivo binario
$(BIN_FILE): $(ELF_FILE)
  $(OBJCOPY) $(OBJCOPYFLAGS) $< $@

# Regla para generar el archivo hex
$(HEX_FILE): $(BIN_FILE)
  $(HEXDUMP) -e '"%08x\n"' $< > $@

# Regla para generar el archivo de desensamblado
$(DISASM_FILE): $(ELF_FILE)
  $(OBJDUMP) $(OBJDUMPFLAGS) $< > $@
```

```
# Limpiar archivos generados
clean:
  rm -f $(OBJ_FILE) $(ELF_FILE) $(BIN_FILE) $(HEX_FILE) $(DISASM_FILE)

.PHONY: all clean
```

Código 6: `Makefile` del template de assembly

Apéndice C Template proyecto en C

Disclaimer:
Si bien el proyecto en C compila y crea hex válido, no se logró adaptar correctamente el script del linker a las características de memoria del procesador, por lo que la ubicación en memoria de variables y demás termina siendo incorrecta.

El proyecto base en C consiste en 6 archivos:

- `main.c`, código principal
- `startup.c`, código de arranque, con inicializaciones en assembly para configuración de stack, heap, inicialización de variables globales, entre otros.
- `linker.ld`, script del linker para la ubicación en memoria de las *cosas*.
- `Makefile`
- `utils.c`, `utils.h`, implementación de `memset` y `memcpy`.

Referencias:

- `https://github.com/five-embeddev/riscv-scratchpad/tree/master/baremetal-startup-c`
- `https://github.com/darklife/darkriscv`

```c
#include <stdint.h>

#define GPIO_BASE 0x40000000 // Base offset for GPIO peripheral
#define GPIO_OFF_PDOR 0x0
#define GPIO_OFF_PSOR 0x1
#define GPIO_OFF_PCOR 0x2
#define GPIO_OFF_PTOR 0x3
#define GPIO_OFF_PDIR 0x4
#define GPIO_OFF_PDDR 0x5

#define GPIO_PDOR (*(volatile uint8_t *)(GPIO_BASE + GPIO_OFF_PDOR))
#define GPIO_PDDR (*(volatile uint8_t *)(GPIO_BASE + GPIO_OFF_PDDR))

int global;

int main()
{
    GPIO_PDDR = 0xFF; // Configurar GPIO como salida
    GPIO_PDOR = 0x00; // Todo el GPIO en 0

    int localint;

    float localfloat = 1.4142;

    while (1)
    {
        __asm__ volatile("nop");
    }

    return 0;
}
```

Código 7: `main.c` del template de C

```c
#include <stdint.h>
#include <string.h>
#include "utils.h"

typedef void (*function_t)(void);

extern uint8_t metal_segment_bss_target_start;
extern uint8_t metal_segment_bss_target_end;
extern const uint8_t metal_segment_data_source_start;
extern uint8_t metal_segment_data_target_start;
extern uint8_t metal_segment_data_target_end;

extern void _enter(void) __attribute__((naked, section(".text.metal.init.enter")
    ));

extern void _start(void) __attribute__((noreturn));
void _Exit(int exit_code) __attribute__((noreturn, noinline));

extern int main(void);

void _enter(void)
{
    __asm__ volatile(
        ".option push;"
        ".option norelax;"
        "la gp, __global_pointer$;"
        "la sp, _sp;"
        ".option pop;"
        "j _start;"
    );
}

void _start(void)
{
    // Initialize .data section
    memcpy(&metal_segment_data_target_start, &metal_segment_data_source_start,
           &metal_segment_data_target_end - &metal_segment_data_target_start);

    // Zero .bss section
    memset(&metal_segment_bss_target_start, 0,
           &metal_segment_bss_target_end - &metal_segment_bss_target_start);

    // Call main
    int ret = main();

    _Exit(ret);
}

void _Exit(int exit_code)
{
    while (1)
    {
        // Infinite loop to hang the program
    }
}
```

Código 8: `startup.c` del template de C

```
OUTPUT_ARCH("riscv")
```

```
ENTRY(_enter)

MEMORY
{
    ram (arw!xi) : ORIGIN = 0x00002000, LENGTH = 0x2000
    rom (irx!wa) : ORIGIN = 0x00000000, LENGTH = 0x2000
}

PHDRS
{
    rom PT_LOAD;
    ram_init PT_LOAD;
    tls PT_TLS;
    ram PT_LOAD;
    text PT_LOAD;
}

SECTIONS
{
    __stack_size = DEFINED(__stack_size) ? __stack_size : 0x400;
    PROVIDE(__stack_size = __stack_size);

    __heap_size = DEFINED(__heap_size) ? __heap_size : 0x800;

    PROVIDE(__metal_boot_hart = 0);
    PROVIDE(__metal_chicken_bit = 1);
    PROVIDE(__metal_eccscrub_bit = 0);

    PROVIDE(metal_dtim_0_memory_start = 0x80000000);
    PROVIDE(metal_dtim_0_memory_end = 0x80000000 + 0x4000);

    .init : {
        KEEP (*(.text.metal.init.enter))
        KEEP (*(.text.metal.init.*))
        KEEP (*(SORT_NONE(.init)))
        KEEP (*(.text.libgloss.start))
    } >rom :rom

    .fini : {
        KEEP (*(SORT_NONE(.fini)))
    } >rom :rom

    .preinit_array : ALIGN(8) {
        PROVIDE_HIDDEN (__preinit_array_start = .);
        KEEP (*(.preinit_array))
        PROVIDE_HIDDEN (__preinit_array_end = .);
    } >rom :rom

    .init_array : ALIGN(8) {
        PROVIDE_HIDDEN (__init_array_start = .);
        KEEP (*(SORT_BY_INIT_PRIORITY(.init_array.*) SORT_BY_INIT_PRIORITY(.ctors.*)))
        KEEP (*(.init_array EXCLUDE_FILE (*crtbegin.o *crtbegin?.o *crtend.o *crtend?.o )
     .ctors))
        PROVIDE_HIDDEN (__init_array_end = .);
        PROVIDE_HIDDEN ( metal_constructors_start = .);
        KEEP (*(SORT_BY_INIT_PRIORITY(.metal.init_array.*)));
        KEEP (*(.metal.init_array));
        PROVIDE_HIDDEN ( metal_constructors_end = .);
    } >rom :rom

    .fini_array : ALIGN(8) {
        PROVIDE_HIDDEN (__fini_array_start = .);
        KEEP (*(SORT_BY_INIT_PRIORITY(.fini_array.*) SORT_BY_INIT_PRIORITY(.dtors.*)))
        KEEP (*(.fini_array EXCLUDE_FILE (*crtbegin.o *crtbegin?.o *crtend.o *crtend?.o )
     .dtors))
        PROVIDE_HIDDEN (__fini_array_end = .);
```

```
        PROVIDE_HIDDEN ( metal_destructors_start = .);
        KEEP (*(SORT_BY_INIT_PRIORITY(.metal.fini_array.*)));
        KEEP (*(.metal.fini_array));
        PROVIDE_HIDDEN ( metal_destructors_end = .);
    } >rom :rom

    .ctors : {
        KEEP (*crtbegin.o(.ctors))
        KEEP (*crtbegin?.o(.ctors))
        KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o ) .ctors))
        KEEP (*(SORT(.ctors.*)))
        KEEP (*(.ctors))
        KEEP (*(.metal.ctors .metal.ctors.*))
    } >rom :rom

    .dtors : {
        KEEP (*crtbegin.o(.dtors))
        KEEP (*crtbegin?.o(.dtors))
        KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o ) .dtors))
        KEEP (*(SORT(.dtors.*)))
        KEEP (*(.dtors))
        KEEP (*(.metal.dtors .metal.dtors.*))
    } >rom :rom

    .rodata : {
        *(.rdata)
        *(.rodata .rodata.*)
        *(.gnu.linkonce.r.*)
        . = ALIGN(8);
        *(.srodata.cst16)
        *(.srodata.cst8)
        *(.srodata.cst4)
        *(.srodata.cst2)
        *(.srodata .srodata.*)
    } >rom :rom

    .text : {
        *(.text.unlikely .text.unlikely.*)
        *(.text.startup .text.startup.*)
        *(.text .text.*)
        *(.gnu.linkonce.t.*)
    } >rom :text

    .data : ALIGN(8) {
        *(.data .data.*)
        *(.gnu.linkonce.d.*)
        . = ALIGN(8);
        PROVIDE( __global_pointer$ = . + 0x800 );
        *(.sdata .sdata.* .sdata2.*)
        *(.gnu.linkonce.s.*)
    } >ram AT>rom :ram_init

    .tdata : ALIGN(8) {
        PROVIDE( __tls_base = . );
        *(.tdata .tdata.* .gnu.linkonce.td.*)
    } >ram AT>rom :tls :ram_init

    PROVIDE( __tdata_source = LOADADDR(.tdata) );
    PROVIDE( __tdata_size = SIZEOF(.tdata) );

    PROVIDE( metal_segment_data_source_start = LOADADDR(.data) );
    PROVIDE( metal_segment_data_target_start = ADDR(.data) );
    PROVIDE( metal_segment_data_target_end = ADDR(.tdata) + SIZEOF(.tdata) );

    .tbss : ALIGN(8) {
        *(.tbss .tbss.* .gnu.linkonce.tb.*)
        *(.tcommon .tcommon.*)
```

```
        PROVIDE( __tls_end = . );
    } >ram AT>ram :tls :ram

    PROVIDE( __tbss_size = SIZEOF(.tbss) );
    PROVIDE( __tls_size = __tls_end - __tls_base );

    .tbss_space : ALIGN(8) {
        . = . + __tbss_size;
    } >ram :ram

    .bss (NOLOAD): ALIGN(8) {
        *(.sbss*)
        *(.gnu.linkonce.sb.*)
        *(.bss .bss.*)
        *(.gnu.linkonce.b.*)
        *(COMMON)
    } >ram :ram

    PROVIDE( metal_segment_bss_source_start = LOADADDR(.tbss) );
    PROVIDE( metal_segment_bss_target_start = ADDR(.tbss) );
    PROVIDE( metal_segment_bss_target_end = ADDR(.bss) + SIZEOF(.bss) );

    .stack (NOLOAD) : ALIGN(16) {
        PROVIDE(metal_segment_stack_begin = .);
        . += __stack_size; /* Hart 0 */
        PROVIDE( _sp = . );
        PROVIDE(metal_segment_stack_end = .);
    } >ram :ram

    .heap (NOLOAD) : ALIGN(8) {
        PROVIDE( __end = . );
        PROVIDE( __heap_start = . );
        PROVIDE( metal_segment_heap_target_start = . );
        . = DEFINED(__heap_max) ? MIN( LENGTH(ram) - ( . - ORIGIN(ram)) , 0x10000000) :
    __heap_size;
        PROVIDE( metal_segment_heap_target_end = . );
        PROVIDE( _heap_end = . );
        PROVIDE( __heap_end = . );
    } >ram :ram

    /DISCARD/ : {
        *(.eh_frame .eh_frame.*)
    }
}
```

Código 9: `linker.ld` del template de C

```
# Nombres de los archivos fuente y objeto
SRC_FILES = main.c startup.c utils.c #timer.c
OBJ_FILES = main.o startup.o utils.o #timer.o
ELF_FILE = main.elf
BIN_FILE = main.bin
HEX_FILE = main.hex
ASM_FILES = main.s startup.s utils.s #timer.s
LIST_FILE = main.lst
MAP_FILE = main.map
LINKER_SCRIPT = linker.ld

# Comandos de compilacion
CC = riscv32-unknown-elf-gcc
LD = riscv32-unknown-elf-ld
OBJCOPY = riscv32-unknown-elf-objcopy
OBJDUMP = riscv32-unknown-elf-objdump

LIBGCC_PATH = "/opt/riscv/lib/gcc/riscv32-unknown-elf/13.2.0"

# Opciones de compilacion y vinculacion
```

```
CFLAGS = -march=rv32i -mabi=ilp32 -O0 -g -Wall -ffreestanding -nostartfiles
LDFLAGS = -T $(LINKER_SCRIPT) -m elf32lriscv -Map=$(MAP_FILE) -nostdlib --no-relax -L=$(
    LIBGCC_PATH) -lgcc
OBJCOPYFLAGS = -O binary

# Regla por defecto
all: $(HEX_FILE) $(LIST_FILE) $(MAP_FILE)

# Regla para compilar los archivos .c a .o
%.o: %.c
  $(CC) $(CFLAGS) -c $< -o $@

# Regla para generar los archivos .s a partir de los archivos .c
%.s: %.c
  $(CC) $(CFLAGS) -S $< -o $@

# Regla para enlazar los archivos objeto
$(ELF_FILE): $(OBJ_FILES)
  $(LD) $(OBJ_FILES) -o $@ $(LDFLAGS)

# Regla para generar el archivo binario a partir del ELF
$(BIN_FILE): $(ELF_FILE)
  $(OBJCOPY) $(OBJCOPYFLAGS) $< $@

# Regla para generar el archivo hexadecimal a partir del binario
$(HEX_FILE): $(BIN_FILE)
  hexdump -e '"%08x\n"' $< > $@

# Regla para generar el ensamblado intermedio de los archivos .c
asm: $(ASM_FILES)

# Regla para generar el archivo .lst
$(LIST_FILE): $(ELF_FILE)
  $(OBJDUMP) -S $< > $@

# Regla para generar el archivo .map
$(MAP_FILE): $(ELF_FILE)

# Regla de ayuda
help:
  @echo "Comandos disponibles:"
  @echo "  all:                Compila todo y genera los archivos $(HEX_FILE), $(
    LIST_FILE) y $(MAP_FILE)."
  @echo "  clean:              Limpia todos los archivos generados."
  @echo "  asm:                Genera los archivos ensamblador intermedios para cada
    archivo fuente."

# Limpiar archivos generados
clean:
  rm -f $(OBJ_FILES) $(ELF_FILE) $(BIN_FILE) $(HEX_FILE) $(ASM_FILES) $(LIST_FILE) $(
    MAP_FILE)

.PHONY: all clean help asm
```

Código 10: `Makefile` del template de C

```
#include <stddef.h>

void *memset(void *s, int c, size_t n)
{
    unsigned char *p = s;
    while (n--)
    {
        *p++ = (unsigned char)c;
    }
    return s;
}
```

```c
void *memcpy(void *dest, const void *src, size_t n)
{
    unsigned char *d = dest;
    const unsigned char *s = src;
    while (n--)
    {
        *d++ = *s++;
    }
    return dest;
}
```

Código 11: `utils.c` del template de C

```c
#ifndef UTILS_H
#define UTILS_H

#include <stddef.h>

void *memset(void *s, int c, size_t n);
void *memcpy(void *dest, const void *src, size_t n);

#endif // UTILS_H
```

Código 12: `utils.h` del template de C