

# Proyecto 3: TinuC, un Microcontrolador basado en RISC-V

Sergio Gil León

Lucas Broch Monfort

Alejandro Bataller Sastre

Alejandro López Azorín

Jose Luis Martínez Cerdá

Mohamed Yassir Aboulmakarim Lemoudden



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA

Integración de Sistemas Digitales

Universitat Politècnica de València

Enero 2025

# Introducción

En este proyecto de prácticas se ha implementado un microcontrolador sencillo basado en la arquitectura *open-source* RISC-V32I. Para ello se ha hecho uso de los conocimientos aprendidos en el conjunto de asignaturas orientadas al diseño y verificación de hardware con Verilog. Todo el código al que se hace referencia puede ser encontrado en el siguiente repositorio de Github: <https://github.com/sgilleo/risc-v>.

Todo el hardware descrito en este trabajo, empezando por las unidades funcionales de la ALU y memorias ha sido verificado con su respectivo testbench para comprobar el funcionamiento de manera exhaustiva. A continuación se ha diseñado una CPU monociclo capaz de utilizar todos estos elementos para ejecutar instrucciones de la ISA de RISC-V. Por último se ha mejorado el diseño para segmentar cada una de las etapas de las instrucciones y se ha incluido todo en un top de la jerarquía que incluye varios puertos de entrada y salida con los que interactuar con el microcontrolador.

## Unidades Funcionales

*Diseño: Sergio Gil, Lucas Broch, Mohamed Yassir, Alejandro Lopez.*

*Verificación: Sergio Gil, Jose Luis Martinez.*

Para que el procesador pueda ejecutar instrucciones necesita una serie de bloques como la ALU (Arithmetic Logic Unit), los bancos de registros o las memorias de datos e instrucciones. Todos ellos han sido implementados en ficheros separados y tienen su respectivo testbench que comprueba las funcionalidades básicas de cada uno.

Las memorias son de **escritura síncrona** y **lectura asíncrona**, y **se ha mantenido** este diseño a lo largo de todo el proyecto, es decir, la CPU segmentada sigue haciendo uso de estas memorias asíncronas por simplicidad y reutilización de código en medida de lo posible.

## Programas en ensamblador

*Diseño: Sergio Gil.*

El ensamblador de RISC-V es como su nombre indica un ensamblador con un juego de instrucciones reducido, por lo que no podemos utilizar métodos de direccionamiento muy complejos, como por ejemplo el direccionamiento indexado que nos era de mucha utilidad en el Motorola 68000 de la asignatura SMICRO. En este proyecto nos tendremos que conformar con escribir un código algo más enrevesado ganando simpleza en el diseño del hardware.

Los programas de Fibonacci se encuentran en la carpeta *assembly/* como *fibonacci.asm* y *fibonacci\_segmentado.asm*. Los programas de ordenamiento están en la misma carpeta como *selection\_sort.asm* y *selection\_sort\_segmentado.asm*. Por último en el mismo directorio se pueden encontrar los volcados en código máquina ensamblados con el simulador de kvakil: <https://venus.kvakil.me/> bajo el mismo nombre que los archivos de ensamblador pero con la extensión .hex.

Los programas para el core segmentado son los mismos que para el core monociclo pero con una serie de instrucciones NOP (No Operation) estratégicamente añadidas para evitar riesgos por dependencias de memoria y registros.

En cuanto al programa de **Fibonacci** genera los primeros N términos (valor N almacenado en el registro x2) y los almacena en la memoria de datos a partir de la dirección 0x000.

En cuanto al programa de **ordenamiento por selección** necesitamos la manera de saber cuándo se acaban los datos que queremos ordenar. De la misma manera que en SMICRO y en muchos lenguajes de ensamblador se utiliza un valor de cero para terminar una cadena de texto con caracteres ASCII, en nuestro programa hemos optado por marcar el final de los datos con el número 0, con el inconveniente de que este valor no puede ser ordenado.

Por último, este programa tiene una primera fase donde carga valores arbitrarios en las primeras posiciones de memoria, y una segunda fase donde comienza el ordenamiento de los datos de menor a mayor hasta que todos están ordenados.

## CPU Monociclo

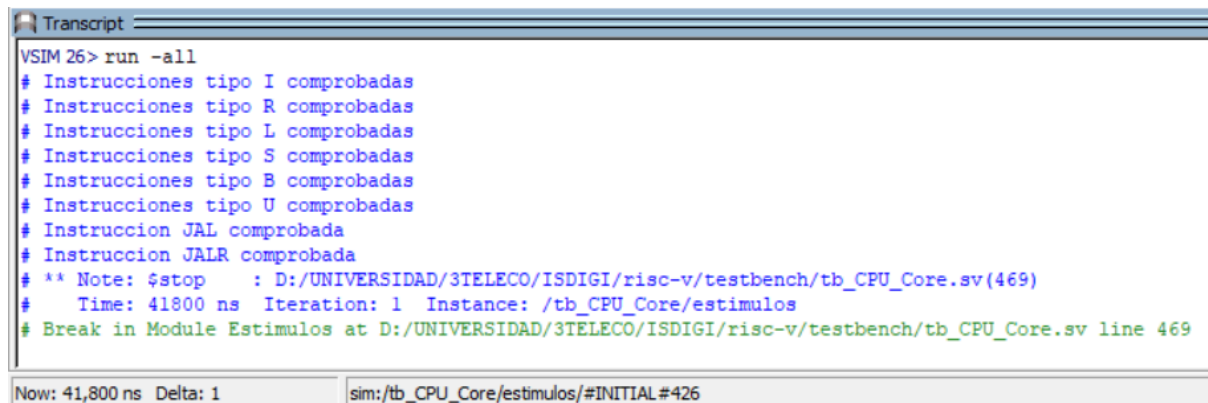
*Diseño: Sergio Gil, Alejandro Lopez, Lucas Broch, Mohamed Yassir, Jose Luis Martinez.*

*Verificación: Sergio Gil, Lucas Broch, Alejandro Bataller.*

La implementación del core monociclo se ha realizado en un solo fichero llamado *CPU\_Core.sv* realizando las instanciaciones pertinentes y manteniendo las memorias fuera.

Además se ha realizado una pequeña modificación en el path para permitir la ejecución de instrucciones adicionales como JAL, JALR, AUIPC, LUI, BGE, BLT o BEQ, entre otras.

El procesador funciona correctamente y ha sido verificado exhaustivamente mediante un testbench estructurado que aleatoriza cada tipo de instrucción y comprueba que su ejecución sea exitosa:



```
Transcript
VSIM 26> run -all
# Instrucciones tipo I comprobadas
# Instrucciones tipo R comprobadas
# Instrucciones tipo L comprobadas
# Instrucciones tipo S comprobadas
# Instrucciones tipo B comprobadas
# Instrucciones tipo U comprobadas
# Instruccion JAL comprobada
# Instruccion JALR comprobada
# ** Note: $stop      : D:/UNIVERSIDAD/3TELECO/ISDIGI/risc-v/testbench/tb_CPU_Core.sv(469)
#   Time: 41800 ns  Iteration: 1  Instance: /tb_CPU_Core/estimulos
# Break in Module Estimulos at D:/UNIVERSIDAD/3TELECO/ISDIGI/risc-v/testbench/tb_CPU_Core.sv line 469

Now: 41,800 ns  Delta: 1          sim:/tb_CPU_Core/estimulos/#INITIAL#426
```

Covergroups									
Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_instances	Get_ins	
[-] /tb_CPU_Core/esti...		100.00%							
+ TYPE RCover		100.00%	100	100.00...	<div><div></div></div>	✓	auto(0)		
+ TYPE ICover		100.00%	100	100.00...	<div><div></div></div>	✓	auto(0)		
+ TYPE LCover		100.00%	100	100.00...	<div><div></div></div>	✓	auto(0)		
+ TYPE SCover		100.00%	100	100.00...	<div><div></div></div>	✓	auto(0)		
+ TYPE BCover		100.00%	100	100.00...	<div><div></div></div>	✓	auto(0)		
+ TYPE UCover		100.00%	100	100.00...	<div><div></div></div>	✓	auto(0)		
+ TYPE JALCove...		100.00%	100	100.00...	<div><div></div></div>	✓	auto(0)		
+ TYPE JALRCov...		100.00%	100	100.00...	<div><div></div></div>	✓	auto(0)		

Además por supuesto se han cargado los programas de Fibonacci y ordenamiento de números conectando el core a las memorias de instrucciones, datos y a un reloj externo de 50MHz.

En el visualizador de memorias de Questa Sim podemos acceder a la RAM para ver los contenidos y comprobar que los programas de la CPU funcionan bien:

Memory Data - /tb_Fibonacci/ram/memoria									
000000e7	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
000000db	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
000000cf	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
000000c3	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
000000b7	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
000000ab	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000009f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000093	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000087	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000007b	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000006f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000063	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000057	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000004b	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000003f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000033	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000027	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000001b	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000000f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	00000022	00000015	0000000d	00000008
00000003	00000002	00000001	00000001	00000000					
0000001b									

En el testbench *tb\_Fibonacci.sv* podemos comprobar cómo efectivamente obtenemos los 10 primeros valores de la serie de Fibonacci (en formato hexadecimal), tal y como estaba programado el procesador.

Memory Data - /tb_Sorting/ram/memoria - Default									
000000e7	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
000000db	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
000000cf	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
000000c3	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
000000b7	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
000000ab	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000009f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000093	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000087	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000007b	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000006f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000063	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000057	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000004b	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000003f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000033	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000027	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000001b	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000000f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	00000000	00000009	00000008	00000007
00000003	00000004	00000003	00000002	00000001					

En el testbench *tb\_Sorting.sv* podemos ver que después de escribir valores del 1 al 9 en orden arbitrario, el programa los ordena hasta que se encuentra con el 0, que marca el final de los datos.

Si realizamos una compilación completa en Quartus podemos ver las características de nuestro diseño, como las especificaciones de área o la frecuencia máxima de operación:

Flow Summary		Slow 1200mV 85C Model Fmax Summary			
<<Filter>>		<<Filter>>			
Flow Status	Successful - Thu Jan 16 15:16:49 2025				
Quartus Prime Version	23.1std.1 Build 993 05/14/2024 SC Lite Edition				
Revision Name	TinuC				
Top-level Entity Name	CPU_Core				
Family	Cyclone IV E				
Device	EP4CE115F29C7				
Timing Models	Final				
Total logic elements	2,879 / 114,480 ( 3 % )				
Total registers	1024				
Total pins	120 / 529 ( 23 % )				
Total virtual pins	0				
Total memory bits	0 / 3,981,312 ( 0 % )				
Embedded Multiplier 9-bit elements	0 / 532 ( 0 % )				
Total PLLs	0 / 4 ( 0 % )				
		Fmax	Restricted Fmax	Clock Name	
1		64.41 MHz	64.41 MHz	CLK	

## CPU Segmentada

**Diseño:** Sergio Gil.

**Verificación:** Sergio Gil, Lucas Broch, Alejandro Bataller, Alejandro Lopez.

A continuación se va a segmentar la CPU en 5 etapas: IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory), WB (Write Back). La descripción del hardware se ha realizado en el fichero *CPU\_Core\_Pipelined.sv*, y se le han implementado todas las instrucciones adicionales: BEQ, BGE, BLT, AUIPC, LUI, JAL, JALR, etc...

El correcto funcionamiento de la CPU segmentada puede ser comprobado mediante el testbench estructurado *tb\_segmentado.sv*.

El resumen de la compilación en Quartus nos muestra las siguientes especificaciones de área y velocidad máxima. Es importante mencionar que tal y como sabíamos hemos aumentado la frecuencia máxima de operación ya que segmentando el procesador hemos reducido la longitud del path crítico, que en el caso de la CPU monociclo era el path de las instrucciones de lectura de memoria.

Flow Summary		Slow 1200mV 85C Model Fmax Summary			
<<Filter>>		<<Filter>>			
Flow Status	Successful - Thu Jan 16 15:25:07 2025				
Quartus Prime Version	23.1std.1 Build 993 05/14/2024 SC Lite Edition				
Revision Name	TinuC				
Top-level Entity Name	CPU_Core_Pipelined				
Family	Cyclone IV E				
Device	EP4CE115F29C7				
Timing Models	Final				
Total logic elements	2,721 / 114,480 ( 2 % )				
Total registers	1483				
Total pins	120 / 529 ( 23 % )				
Total virtual pins	0				
Total memory bits	0 / 3,981,312 ( 0 % )				
Embedded Multiplier 9-bit elements	0 / 532 ( 0 % )				
Total PLLs	0 / 4 ( 0 % )				
		Fmax	Restricted Fmax	Clock Name	
1		103.96 MHz	103.96 MHz	CLK	

Cabe destacar que entre nuestra CPU monociclo y segmentada **no hay mejora en el rendimiento** en términos de instrucciones ejecutadas por ciclo (IPC): en ambos núcleos el rendimiento del procesador sigue siendo de **1 instrucción por ciclo de reloj**.

También se debe tener en cuenta que esta arquitectura segmentada introduce **4 ciclos de latencia** debido a las etapas adicionales. En las siguientes figuras podemos ver la comparación del tiempo de ejecución del programa de Fibonacci en la CPU monociclo y segmentada. El núcleo monociclo realiza la ejecución en 1570 ns mientras que a la CPU segmentada le lleva 3150 ns. Esto se debe a las instrucciones NOP que hemos añadido al programa para evitar riesgos, lo que lo hace más largo.

```
VSIM 19> run -all
# ** Note: $stop      : D:/UNIVERSIDAD/3TELECO/ISDIGI/risc-v/testbench/tb_Fibonacci.sv(60)
#   Time: 1570 ns   Iteration: 3   Instance: /tb_Fibonacci
# Break in Module tb_Fibonacci at D:/UNIVERSIDAD/3TELECO/ISDIGI/risc-v/testbench/tb_Fibonacci.sv line 60
```

Tiempo de ejecución en la CPU monociclo

```
VSIM 21> run -all
# ** Note: $stop      : D:/UNIVERSIDAD/3TELECO/ISDIGI/risc-v/testbench/tb_Fibonacci_Pipelined.sv(60)
#   Time: 3150 ns   Iteration: 2   Instance: /tb_Fibonacci_Pipelined
# Break in Module tb_Fibonacci_Pipelined at D:/UNIVERSIDAD/3TELECO/ISDIGI/risc-v/testbench/tb_Fibonacci_Pipelined.sv line 60
```

Tiempo de ejecución en la CPU segmentada

## Microcontrolador TinuC

*Jose Luis Martinez, Mohamed Yassir*

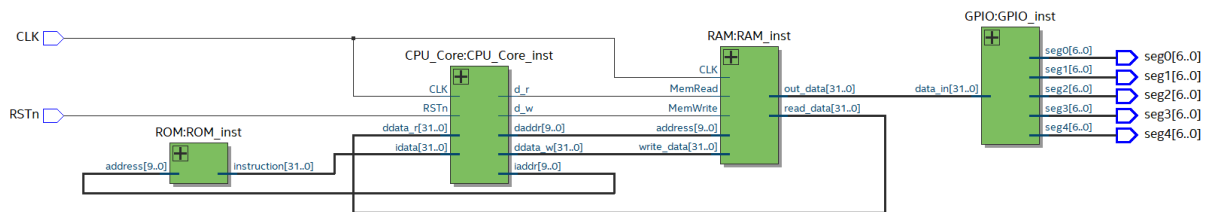
El microcontrolador TinuC es un microcontrolador basado en la arquitectura RISC-V diseñado específicamente para la implementación en FPGA. Su arquitectura optimizada permite una ejecución eficiente de instrucciones, garantizando un uso óptimo de los recursos de hardware disponibles.

El microcontrolador TinuC utiliza un núcleo monociclo en su implementación. Esto implica que todas las instrucciones se ejecutan en un solo ciclo de reloj. Aunque no se utiliza una arquitectura segmentada en este diseño, se lograron resultados funcionales óptimos, manteniendo la simplicidad y minimizando los riesgos asociados con dependencias de datos y control.

La optimización en el uso de recursos ha sido clave en el diseño del microcontrolador TinuC. Se han implementado memorias de escritura síncrona y lectura asíncrona para mejorar la



estabilidad del sistema y reducir los tiempos de acceso.



(Esquema de bloques del TinuC)

A continuación se detallarán algunos aspectos importantes sobre su diseño y funcionamiento:

## Maapeo de memoria para displays de 7 segmentos

Para conectar el microcontrolador con los displays de 7 segmentos, asignamos direcciones específicas en la RAM para controlar los segmentos individuales. Este proceso consiste en mapear direcciones de memoria dentro de la región *out\_data* de la RAM, lo que permite controlar directamente el encendido y apagado de cada segmento desde el software.

Por ejemplo, cada segmento del display se asocia con un bit específico en un registro de la memoria RAM, permitiendo escribir valores binarios que encienden o apagan los segmentos correspondientes. Este método simplifica el control mediante el uso de un mapa de memoria bien definido y accesible desde el procesador.

```
always_comb
begin
case (d0)
4'h0: seg0 = 7'b1000000;
4'h1: seg0 = 7'b1111001;
4'h2: seg0 = 7'b0100100;
4'h3: seg0 = 7'b0110000;
4'h4: seg0 = 7'b0011001;
4'h5: seg0 = 7'b0010010;
4'h6: seg0 = 7'b0000010;
4'h7: seg0 = 7'b1111000;
4'h8: seg0 = 7'b0000000;
4'h9: seg0 = 7'b0010000;
4'hA: seg0 = 7'b0000100;
4'hB: seg0 = 7'b0000011;
4'hC: seg0 = 7'b1000110;
4'hD: seg0 = 7'b0100001;
4'hE: seg0 = 7'b0000110;
4'hF: seg0 = 7'b0001110;
default: seg0 = 7'b1111111;
endcase
end
```

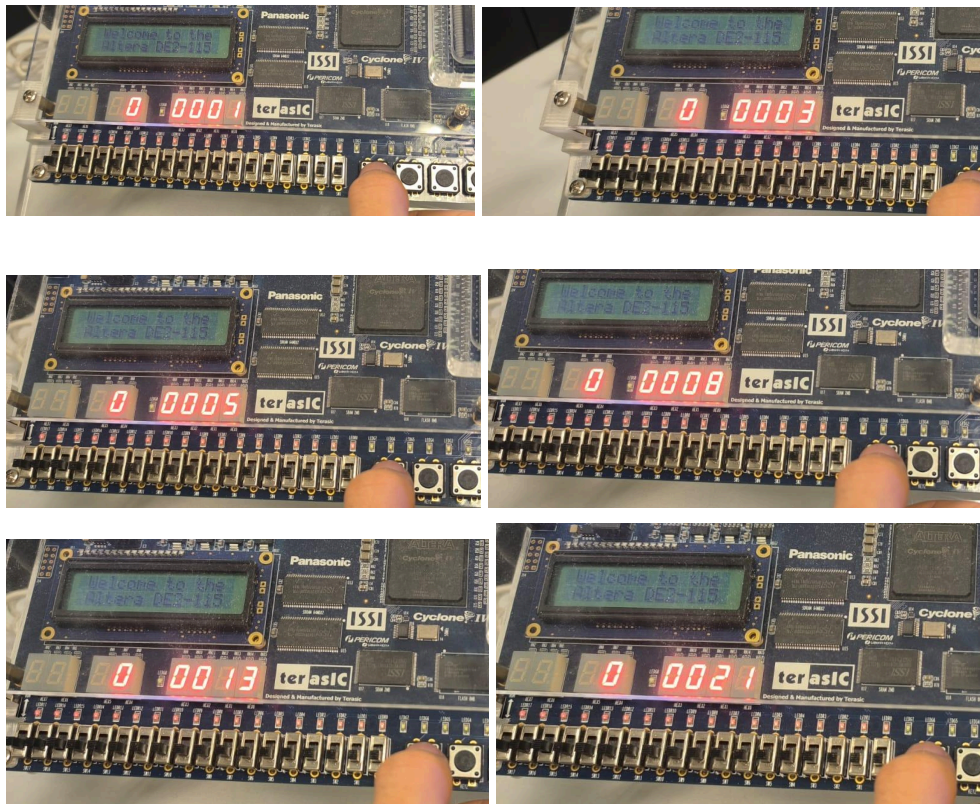
(Ejemplo de la asociación bit-registro)

## Implementación y validación en FPGA

La implementación final ha sido realizada en una FPGA, conectando el microprocesador a periféricos como: el display de 7 segmentos, los pulsadores y los interruptores.

En nuestro caso, hemos implementado el nivel de funcionalidad básico, que sería mostrar la serie de Fibonacci en el display.

Aquí imágenes de ello:



## Conclusiones, Resultados y Observaciones

El proyecto TinuC ha permitido integrar y consolidar conceptos clave del diseño digital y la arquitectura RISC-V. La transición de un diseño monociclo a uno segmentado demuestra la importancia de optimizar recursos y resolver problemas como riesgos de datos y control.

El resultado final del proyecto, validado en FPGA, refleja la aplicación práctica de los conocimientos adquiridos y su potencial en entornos reales.

En lo respectivo a los resultados vamos lo siguiente:

- **Frecuencia máxima:** El modelo monociclo alcanzó una frecuencia máxima de 64.41 MHz, reflejando un diseño eficiente.
- **Recursos utilizados:** Se utilizaron 49,495 LEs (43% de la FPGA Cyclone IV E), 33,792 registros (29.5%) y 37 pines (7% de los disponibles). No se empleó memoria M9K ni multiplicadores embebidos de 9 bits, destacando la eficiencia del uso de los recursos de hardware.



- **Mapeo de periféricos:** Se implementó un controlador de memoria para conectar el microcontrolador con periféricos como GPIO y displays de 7 segmentos, lo que facilitó su interacción con el hardware externo.
- **Rendimiento:** El rendimiento se mantuvo en 1 instrucción por ciclo de reloj, destacando la capacidad de la arquitectura para operar a frecuencias más altas.

El éxito del diseño se reflejó en la capacidad del sistema para ejecutar programas básicos como Fibonacci y ordenamiento de números de manera precisa y eficiente, validando así su funcionalidad y aplicabilidad práctica.