

Universidad de Guadalajara  
Centro Universitario de Ciencias Exactas e Ingenierías  
Arquitectura de Computadoras 2025B  
30 de noviembre de 2025

# Procesador MIPS

Actividad en equipo

López Arce Delgado, Jorge Ernesto  
*Asesor*

Basulto Silva, Edgar Andrés  
*Académico*

Bugarin Salvatierra, Juan José  
*Académico*

Lavadores Morgado, María Fernanda  
*Académico*

Sección: D12                      NRC: 227925  
Licenciatura en Ingeniería Informática

El presente proyecto final se centra en el análisis y la implementación de un procesador MIPS básico mediante un pipeline de 5 etapas, así como en el desarrollo de un decodificador de instrucciones escrito en Python. El objetivo principal es comprender cómo un programa escrito en lenguaje ensamblador puede ejecutarse directamente como lógica digital mediante un diseño basado en Verilog. Para ello, se estudian conceptos relacionados con la arquitectura de computadoras, el conjunto de instrucciones MIPS32, la codificación binaria de instrucciones y el funcionamiento interno de un procesador simplificado.

El pipeline MIPS se divide en las etapas clásicas: *Instruction Fetch (IF)*, *Instruction Decode (ID)*, *Execute (EX)*, *Memory Access (MEM)* y *Write Back (WB)*. Cada una cumple un rol específico dentro del ciclo de ejecución y, en conjunto, permiten que el procesador logre una ejecución más eficiente mediante la superposición de instrucciones. Para llevar a cabo esta práctica, también se revisan componentes fundamentales como la Unidad de Control, la ALU, las memorias de instrucciones y datos, el Banco de Registros y los buffers entre etapas.

Por otro lado, el decodificador en Python tiene la función de traducir el código ensamblador escrito por el usuario a su representación binaria exacta en formato MIPS32, respetando la estructura de las instrucciones tipo R, I y J. Esta traducción es necesaria para precargar la memoria de instrucciones del procesador implementado en Verilog y así ejecutar correctamente el programa ensamblador diseñado.

Esta práctica combina tanto teoría como implementación real, reforzando conceptos de arquitectura, codificación de instrucciones, diseño RTL y simulación digital. Además, permite observar de manera directa la relación entre software de bajo nivel y hardware digital, así como el nivel de detalle necesario para que ambos trabajen de manera coherente.

# Objetivo General

Implementar un pipeline MIPS de 5 etapas en Verilog junto con un decodificador en Python capaz de traducir instrucciones ensamblador al formato binario MIPS32, con el fin de demostrar el funcionamiento completo de un programa dentro del procesador diseñado.

## Objetivos Particulares

- Implementar un decodificador en Python que valide, procese y convierta instrucciones MIPS a su representación binaria de 32 bits.
- Diseñar y conectar los módulos principales del procesador: PC, memorias, ALU, Banco de Registros, Unidad de Control, ALU Control y buffers de pipeline.
- Implementar las etapas IF, ID, EX, MEM y WB respetando el flujo de datos del pipeline MIPS.
- Crear un programa en ensamblador que utilice instrucciones aritméticas, lógicas, de comparación, acceso a memoria y saltos.
- Realizar simulaciones del procesador en ModelSim y verificar la ejecución del programa.
- Generar y documentar un flujo completo desde la escritura del programa ensamblador hasta su ejecución en el procesador implementado.

## GUI - Decodificador

Para la construcción del decodificador se analizaron los formatos estándar MIPS32:

- **Tipo R:** opcode = 0, campos rs, rt, rd, shamt y funct.
- **Tipo I:** opcode, rs, rt e inmediato de 16 bits.
- **Tipo J:** opcode y dirección de 26 bits.

El decodificador fue diseñado para:

1. **Leer instrucciones desde archivo o caja de texto.**
2. **Verificar la sintaxis de cada instrucción**, detectando errores como registros inexistentes o cantidad incorrecta de argumentos.
3. **Identificar el tipo de instrucción** (R, I o J).
4. **Generar el binario de 32 bits**, concatenando cada campo según el formato.
5. **Guardar el binario en un archivo compatible** con la memoria de instrucciones del pipeline.

Se añadió también una opción automática para manejar instrucciones predeterminadas cuando no pertenecen a ningún formato estándar, permitiendo mayor flexibilidad en el flujo.

```

30 ventana = tk.Tk()
31 ventana.title("ASM -> MIPS32 (convertidor) - Versión completada")
32 ventana.geometry("780x640")
33
34 tipo_seleccionado = tk.StringVar(value="AUTO")
35 vista_bytes = tk.BooleanVar(value=False)
36
37 tk.Label(ventana, text="Tipo de instrucción (selección para validación, o dejar
preferias:).").pack(anchor="w", padx=8, pady=(8,8))
38 menu_tipos = tk.OptionMenu(ventana, tipo_seleccionado, "AUTO", "R", "I", "J")
39 menu_tipos.pack(anchor="w", padx=8)
40
41 tk.Checkbutton(ventana, text="Mostrar por bytes (en salida)", variable=vista_byt
padx=8, pady=(8,8))
42
43 tk.Label(ventana, text="Instrucciones (ASM):").pack(anchor="w", padx=8)
44 frame_entrada = tk.Frame(ventana)
45 frame_entrada.pack(padx=8)
46
47 scroll_in = tk.Scrollbar(frame_entrada)
48 scroll_in.pack(side="right", fill="y")
49 entrada_texto = tk.Text(frame_entrada, height=10, width=98, yscrollcommand=scroll_in.set)
50 entrada_texto.pack(side="left", fill="both")
51 scroll_in.config(command=entrada_texto.yview)
52
53 frame_botones = tk.Frame(ventana)
54 frame_botones.pack(pady=8)
55 tk.Button(frame_botones, text="Cargar archivo", command=cargar_archivo).grid(row=0, column=0)
56 tk.Button(frame_botones, text="Convertir", command=convertir).grid(row=0, column=1)
57 tk.Button(frame_botones, text="Guardar resultado", command=guardar_archivo).grid(row=0, column=2)
58 tk.Button(frame_botones, text="Limpiar", command=limpiar).grid(row=0, column=3)
59
60 def reg_a_binario(reg: str) -> str:
61     if not isinstance(reg, str):
62         raise ValueError(f"Registro inválido: {reg}")
63     r = reg.strip().lower()
64     if r in REG_ALIAS:
65         num = REG_ALIAS[r]
66     else:
67         m = re.fullmatch(r"$(\d{1,2})", reg.strip())
68         if not m:
69             raise ValueError(f"Formato de registro inválido: '{reg}' (use $n o a)
70         num = int(m.group(1))
71     if not (0 <= num <= 31):
72         raise ValueError(f"Registro fuera de rango: '{reg}' (debe ser 0..31).")
73     return format(num, '05b')
74
75 def int_a_bin_signed(value: int, bits: int) -> str:
76     min_val = - (1 << (bits - 1))
77     max_val = (1 << (bits - 1)) - 1
78     if not (min_val <= value <= max_val):
79         raise ValueError(f"Valor con signo fuera de rango para {bits} bits: {val
80     if value < 0:
81         value = (1 << bits) + value
82     return format(value & ((1 << bits) - 1), f'0{bits}b')
83
84 def uint_a_bin(value: int, bits: int) -> str:
85     if value < 0 or value >= (1 << bits):
86         raise ValueError(f"Valor sin signo fuera de rango para {bits} bits: {val
87     return format(value, f'0{bits}b')
88
89 def parse_immediate(token: str) -> int:
90     token = token.strip()

```

```

1 import tkinter as tk
2 from tkinter import messagebox, filedialog
3 import os
4 import re
5 from typing import Optional, Tuple, List, Dict
6
7 instrucciones = {
8     "ADD": {"opcode": "000000", "funct": "100000"},
9     "SUB": {"opcode": "000000", "funct": "100010"},
10    "AND": {"opcode": "000000", "funct": "100100"},
11    "OR": {"opcode": "000000", "funct": "100101"},
12    "SLT": {"opcode": "000000", "funct": "101010"},
13    "NOP": {"opcode": "000000", "funct": "000000"},
14 }
15
16 instrucciones_I = {
17     "ADDI": {"opcode": "001000", "signed_imm": True},
18     "ANDI": {"opcode": "001100", "signed_imm": False},
19     "ORI": {"opcode": "001101", "signed_imm": False},
20     "XORI": {"opcode": "001110", "signed_imm": False},
21     "SLTI": {"opcode": "001010", "signed_imm": True},
22     "BEQ": {"opcode": "000100", "signed_imm": True, "branch": True},
23     "BNE": {"opcode": "000101", "signed_imm": True, "branch": True},
24     "BGTZ": {"opcode": "000111", "signed_imm": True, "special_rs_only": True},
25     "LW": {"opcode": "100011", "signed_imm": True},
26     "SW": {"opcode": "101011", "signed_imm": True},
27 }
28
29 instrucciones_J = {
30     "J": {"opcode": "000010"},
31 }

```

Tipo de instrucción (selección para validación, o dejar R/I/I según preferias):

AUTO

☒ Mostrar por bytes (en salida)

Instrucciones (ASM):

ADD \$t0, \$t1, \$t2  
SUB \$s0, \$s1, \$s2  
AND \$t3, \$t4, \$t5  
OR \$a0, \$a1, \$a2  
SLT \$v0, \$v1, \$a0  
ADDI \$t0, \$t1, 10  
ANDI \$s0, \$s1, 0xFF  
ORI \$t2, \$t3, 5  
XORI \$s2, \$s3, 12  
SLTI \$a1, \$a2, -3

Cargar archivo

Convertir

Guardar resultado

Limpiar

Resultado (binario):

00000001  
00101010  
01000000  
00100000  
00000010  
00110010  
10000000  
00100010  
00000001  
10001101  
01011000  
00100100  
00000000

# Desarrollo Pipeline MIPS

## PC\_Procesador.v

Este módulo implementa el Programa Contador, el cual es un registro encargado de almacenar la dirección de la instrucción que está en ejecución. Su valor se actualiza en cada flanco de reloj con la dirección de la siguiente instrucción a ejecutar, permitiendo el flujo secuencial del programa.

## MEM\_WB\_Procesador.v

Este es el registro de *pipeline* que se encuentra entre la etapa de Memoria y la etapa de Escritura en Registros. Su principal responsabilidad es almacenar temporalmente el dato leído de la memoria o el resultado de la ALU, junto con las señales de control, para que la escritura final se realice en el siguiente ciclo.

## MemInstrucciones\_Procesador.v

Este módulo simula la Memoria de Instrucciones del procesador. Se encarga de leer el archivo instrucciones punto txt y entregar la instrucción de 32 bits correspondiente a la dirección de memoria solicitada por el Programa Contador.

## MUX\_5bit\_Procesador.v

Este es un multiplexor de selección de dos entradas de cinco bits. En el contexto del *datapath*, se utiliza específicamente para seleccionar cuál de los campos de la instrucción, Rt o Rd, debe usarse como la dirección del registro destino para la operación de escritura.

## MUX\_Procesador.v

Este módulo implementa un multiplexor de 32 bits. Es un componente fundamental en la ruta de datos que permite elegir una de dos fuentes de datos, como el resultado de la ALU o el dato leído de la memoria, antes de que el valor se escriba en el Banco de Registros.

## PROCESADOR.v

Este módulo representa el diseño de alto nivel de todo el procesador MIPS. Se encarga de interconectar todos los subcomponentes como la Unidad de Control, el Banco de Registros, la ALU, y los registros de *pipeline*, orquestando el flujo de información para la ejecución completa de las instrucciones.

## Shift\_Left\_2\_Procesador.v

Este módulo realiza un desplazamiento lógico de dos posiciones a la izquierda sobre una entrada de 32 bits. Esta operación es crucial en la arquitectura MIPS para calcular las direcciones de salto (*branch* o *jump*) al multiplicar el inmediato o la porción de dirección por cuatro.

## PROCESADOR\_TB.v

Este archivo es el banco de pruebas principal utilizado para validar la funcionalidad del módulo PROCESADOR. Define el reloj y el *reset* inicial, y establece la duración total de la simulación para verificar el comportamiento del diseño bajo condiciones controladas.

## ram\_sync\_Procesador.v

Este módulo modela la Memoria de Datos síncrona. Su función es almacenar y proporcionar datos al procesador. Los datos se escriben en la memoria solo en el flanco positivo del reloj cuando la señal MemWrite está activa.

## Sign-Extend\_Procesador.v

Este componente realiza la extensión de signo de un valor de 16 bits (el inmediato) a un valor de 32 bits. Esto es esencial para las instrucciones de tipo I que utilizan constantes, asegurando que el valor conserve su signo al ser utilizado en operaciones aritméticas de 32 bits.

## Control\_Procesador.v

La Unidad de Control es el componente lógico que interpreta el código de operación de la instrucción y genera todas las señales de control binarias necesarias para dirigir las operaciones en el resto del *datapath*, como la activación de la escritura en registros o la lectura de memoria.

## ADD.v

Este es un módulo combinacional simple que implementa un sumador de 32 bits. Se utiliza en varias partes del *datapath*, siendo la más común la de calcular el valor del PC más 4 para la siguiente instrucción secuencial.

## ALU\_Control\_Procesador.v

Este módulo genera el código de operación específico de tres bits para la Unidad Lógico Aritmética. Utiliza la señal ALUOp de la Unidad de Control y el campo Funct de las instrucciones tipo R para seleccionar la función exacta que la ALU debe realizar.

## ALU\_Procesador.v

La Unidad Lógico Aritmética es el bloque que ejecuta todas las operaciones de datos, incluyendo sumas, restas, operaciones lógicas AND u OR y la función Set Less Than. Recibe un código de selección para determinar la función a realizar sobre sus dos operandos.

## AND.v

Este módulo implementa una puerta lógica AND de un solo bit. En el diseño del procesador, es crucial para combinar señales de control, típicamente la señal

Branch con la señal Zero de la ALU, para decidir si se debe tomar un salto condicional.

## BR\_Procesador.v

Este módulo simula el Banco de Registros, el cual almacena los 32 registros de propósito general. Permite la lectura asincrónica de dos registros a la vez y la escritura sincrónica en el registro destino especificado.

## IF\_ID\_Procesador.v

Este es el registro de *pipeline* que transfiere la instrucción obtenida en la etapa de Fetch y el valor de PC más 4 a la siguiente etapa de decodificación en el próximo ciclo de reloj.

## EX\_MEM\_Procesador.v

Este registro de *pipeline* conecta la etapa de Ejecución con la etapa de Memoria. Transfiere el resultado de la ALU, el dato a ser escrito en la memoria de datos, y las señales de control de lectura y escritura de memoria.

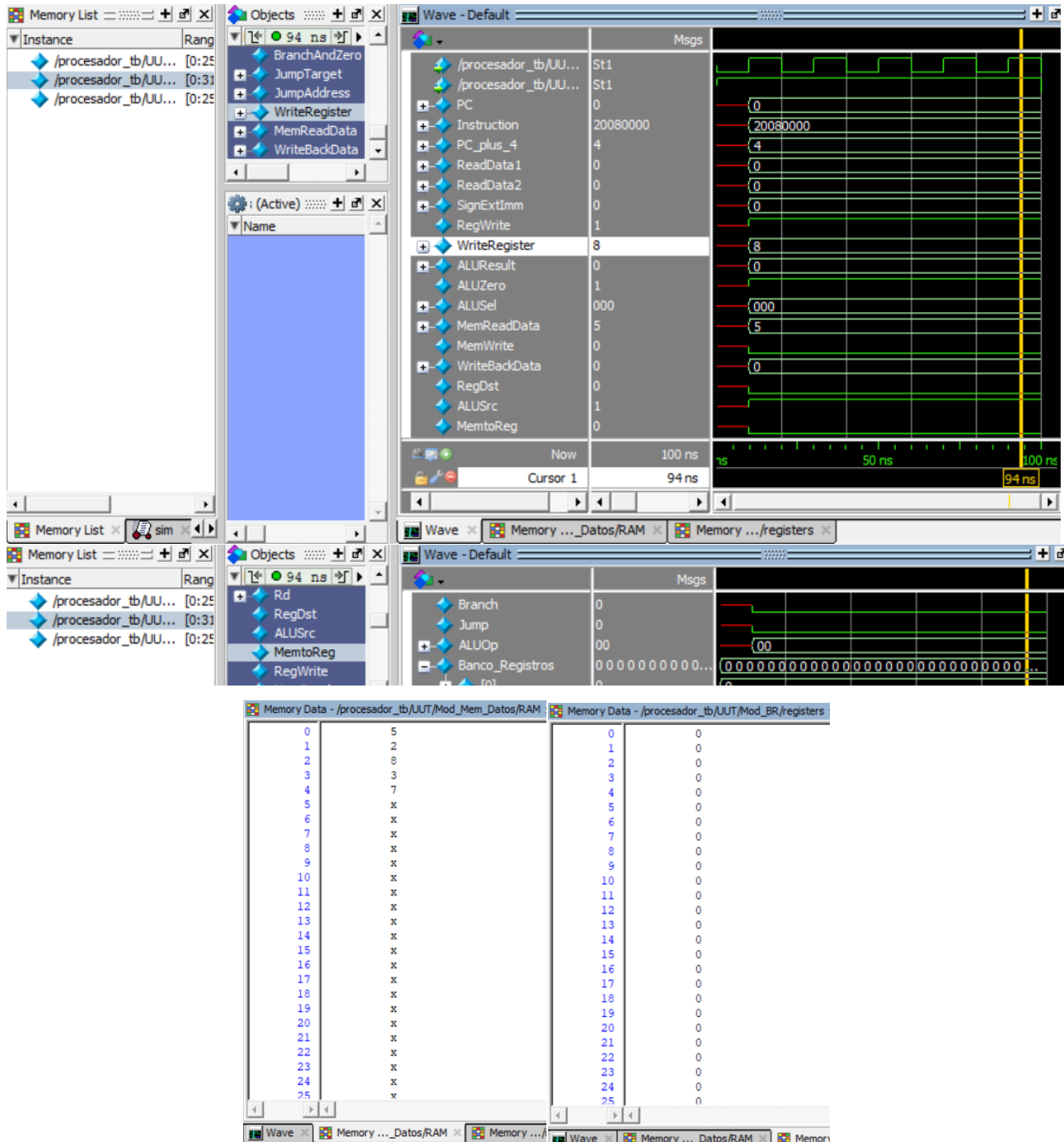
## ID\_EX\_Procesador.v

Este registro de *pipeline* se ubica entre la etapa de Decodificación y la etapa de Ejecución. Su función es transferir los operandos de los registros, el inmediato extendido y las señales de control hacia la Unidad Lógico Aritmética para la ejecución.

# Simulación y Ejecución paso a paso

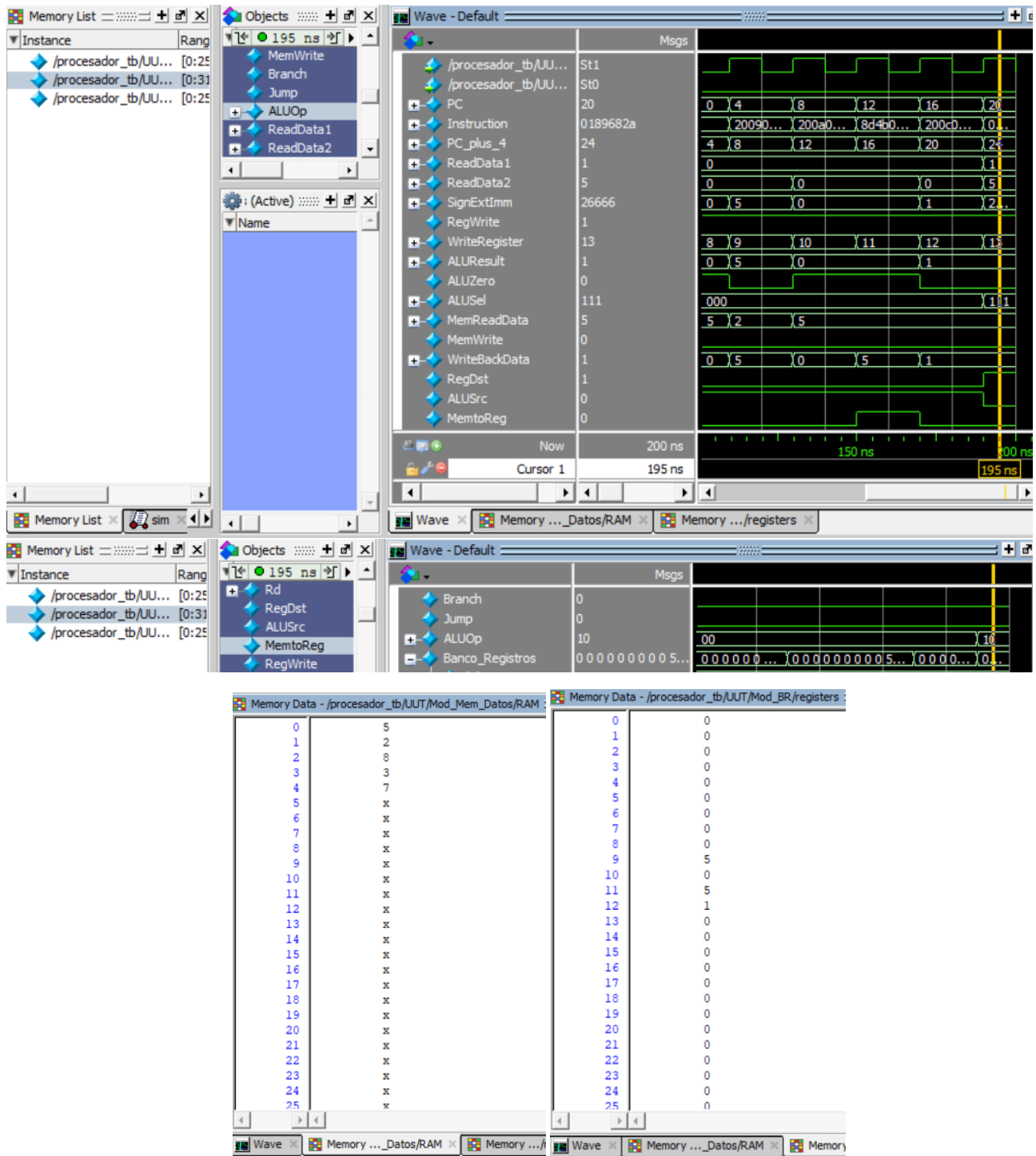
100 ns

El sistema comienza su ejecución al desactivarse la señal de *reset* en este instante. El **Programa Contador** se inicializa a cero, y el procesador comienza la fase de búsqueda de la primera instrucción del programa, que es una operación de inicialización. En este momento, todos los registros del banco se encuentran en cero, ya que aún no se ha completado ninguna fase de escritura o *Write Back*.



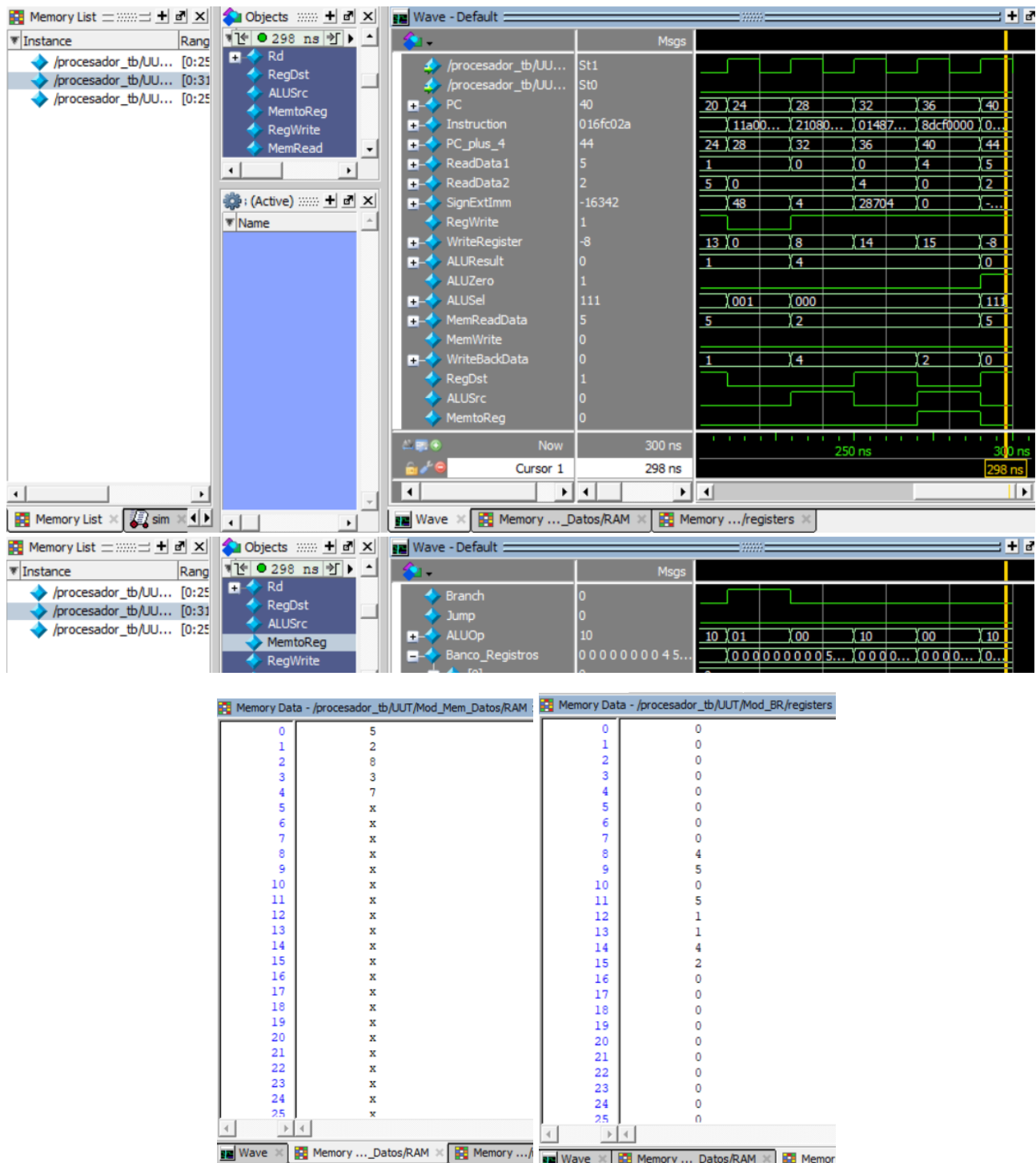
200 ns

En este punto, el procesador ha completado la ejecución de las primeras instrucciones que establecen las variables. El **Programa Contador** se encuentra en la dirección veinte, preparándose para buscar la siguiente instrucción. Los registros clave han sido inicializados correctamente: el registro **t0 de Base es 0**, el registro **t1 de Tamaño N es 5**, el registro **t3 de Máximo Inicial es 7** (cargado de la primera posición del arreglo), y el registro **t4 de Índice i es 1**. La ejecución avanza hacia la lógica del bucle principal.



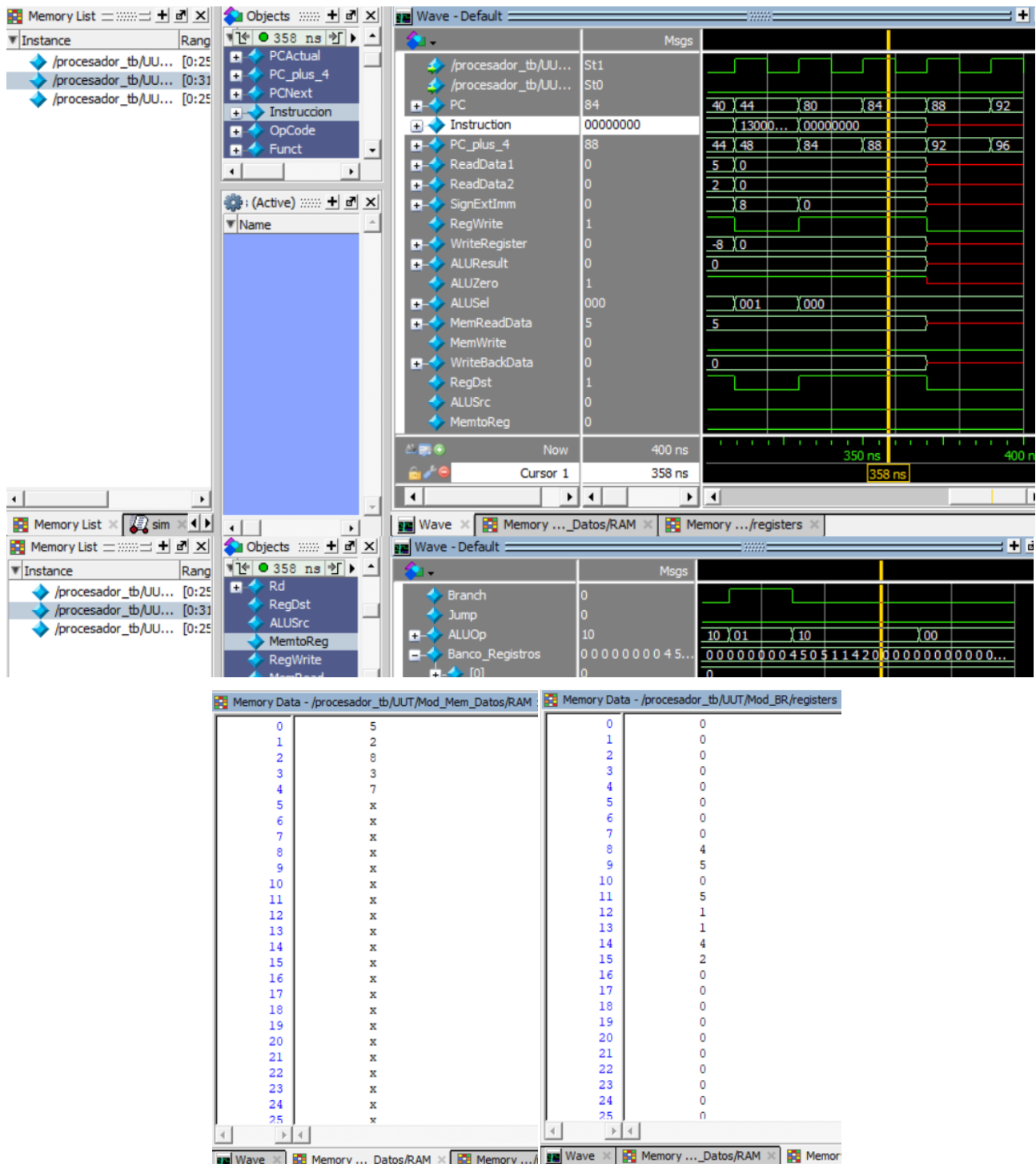
300 ns

El procesador se encuentra ejecutando la primera iteración del bucle, habiendo calculado la dirección del elemento A [1] y cargado su valor. El registro **t6** ahora contiene la dirección **4**, y el registro **t7** almacena el valor **5**, el contenido de la primera posición a comparar. La **condición de salida del bucle** ( $i < N$ ) se ha evaluado como verdadera. El **Programa Contador** está en la dirección cuarenta, en la instrucción de salto condicional **beq**, confirmando que el bucle debe continuar su ejecución.



400 ns

El procesador ha finalizado el ciclo de comparación para el índice  $i = 1$ . El **Máximo Actual (registro t3)** se mantiene en **7**, dado que el valor leído (5) no fue mayor. El índice **t4** se ha incrementado a **2** para la próxima iteración, y el offset para la siguiente posición del arreglo A [2] se ha calculado en **t6 = 8**. El **Programa Contador** apunta a la dirección treinta y dos, donde se procederá a la carga del siguiente elemento, preparando la segunda comparación del bucle.



# Conclusiones

El trabajo final implicó de un proceso bastante extenuante, agotador y de mucho estrés, a pesar de las ayudas de mis compañeros de equipo, la carga laboral combinada con las tareas de las demás materias, provocó que si el procedimiento de la fabricación de este documento, así como de cada uno de los códigos pertinentes, fuera el triple de complicado. A pesar de todo eso, quiero dejar como evidencia mi progreso y aprendizaje a lo largo del curso de Arquitectura de Computadoras, en el que tuvimos una gran carga de información y procesos que yo por lo menos no tenía tan bien formulados, ni sedimentados, lo que me hizo disfrutar de cada pequeño éxito, progreso y aprendizaje que lograba. Desde aprender los tipos de instrucciones R, I, J, los diferentes apartados, componentes de un sistema operativo, así como el aprendizaje de nuevos lenguajes de programación, en los que estaba poco relacionada. Espero que estos conocimientos adquiridos hayan sido suficientes para cumplir con las expectativas del trabajo final.

# Bibliografia

(N.d.). Google.com. Retrieved December 1, 2025, from <https://drive.google.com/file/d/1hpbabBDpddJ4zqakToWJK0YPTnkvcZ2M/view>

(N.d.-b). Google.com. Retrieved December 1, 2025, from <https://drive.google.com/file/d/1oB6AaM1SoJeuOxuAFwWltUJDgtCBLBTA/view>

(N.d.-c). Google.com. Retrieved December 1, 2025, from [https://drive.google.com/file/d/1BKjv7EP9p7xnwulgEFC\\_K6Dy8iCXG9Pj/view](https://drive.google.com/file/d/1BKjv7EP9p7xnwulgEFC_K6Dy8iCXG9Pj/view)

(N.d.-d). Googleusercontent.com. Retrieved December 1, 2025, from [https://doc-0o-as-apps-viewer.googleusercontent.com/viewer/secure/pdf/ru0kl89kvjhhodc35ut56hsd38mphjfp/ioppboi8bqj98vgea0m255vhm2nj56uq/1764558075000/drive/09511929245510141497/ACFrOgAUGZUpGXf-uTXoLnKjAZjogVTpjO1285jWffUha26h6ZfjBjVM1jTgJuOh20l02dX2erzM5xWyrow6eqFo7GBaQNb8W8-V5TGP78AQFpVIjVVG1pBbMXT5f4KynXCPKcbd8qSdBJd2nqBAx6YKUfDgcSRTP0XYjqOZAprOMe8QliEYdmveKzPN8SqD9kXvOa3RccFlDgO9mWb3gTu7krP\\_i5GdDT-xx7BFuZnbYTrKTSDUTuFTS4b-qJKSBaZoS\\_xJ0ShT-u?print=true&nonce=9kmoo9pgtrdjk&user=09511929245510141497&hash=9q60pcb1ghh9d36d5mm4nr2lua78pnb](https://doc-0o-as-apps-viewer.googleusercontent.com/viewer/secure/pdf/ru0kl89kvjhhodc35ut56hsd38mphjfp/ioppboi8bqj98vgea0m255vhm2nj56uq/1764558075000/drive/09511929245510141497/ACFrOgAUGZUpGXf-uTXoLnKjAZjogVTpjO1285jWffUha26h6ZfjBjVM1jTgJuOh20l02dX2erzM5xWyrow6eqFo7GBaQNb8W8-V5TGP78AQFpVIjVVG1pBbMXT5f4KynXCPKcbd8qSdBJd2nqBAx6YKUfDgcSRTP0XYjqOZAprOMe8QliEYdmveKzPN8SqD9kXvOa3RccFlDgO9mWb3gTu7krP_i5GdDT-xx7BFuZnbYTrKTSDUTuFTS4b-qJKSBaZoS_xJ0ShT-u?print=true&nonce=9kmoo9pgtrdjk&user=09511929245510141497&hash=9q60pcb1ghh9d36d5mm4nr2lua78pnb)

(N.d.-b). Googleusercontent.com. Retrieved December 1, 2025, from [https://doc-04-as-apps-viewer.googleusercontent.com/viewer/secure/pdf/ru0kl89kvjhhodc35ut56hsd38mphjfp/e5asugml82961rn995lna1b261o38i41/1764558150000/drive/09511929245510141497/ACFrOgD\\_bMi5thvVHu3ZJbv5Xk8JymOyPYlJPxXB2bbSfM\\_Ofey2mi8oUxPcGRLRje9z3lP5mvcd80qGRYw8HU7TwgWeERsYemBKUilR\\_GZmnC79iMaSY5b-725jU8dLpPHHvYzNGLTq5LmzNnvqEaJl9-u-fCvTJgWoaAMsGermheJlPtNCRQOV4KB8TlflWLb5PkyRstjAaxa87v-jJMiCYTeuO2t9xaY3YyYg==?print=true](https://doc-04-as-apps-viewer.googleusercontent.com/viewer/secure/pdf/ru0kl89kvjhhodc35ut56hsd38mphjfp/e5asugml82961rn995lna1b261o38i41/1764558150000/drive/09511929245510141497/ACFrOgD_bMi5thvVHu3ZJbv5Xk8JymOyPYlJPxXB2bbSfM_Ofey2mi8oUxPcGRLRje9z3lP5mvcd80qGRYw8HU7TwgWeERsYemBKUilR_GZmnC79iMaSY5b-725jU8dLpPHHvYzNGLTq5LmzNnvqEaJl9-u-fCvTJgWoaAMsGermheJlPtNCRQOV4KB8TlflWLb5PkyRstjAaxa87v-jJMiCYTeuO2t9xaY3YyYg==?print=true)