Tecnológico de Monterrey

Campus Guadalajara

RISCV Pipeline

José Antonio Fernández Pineda | A00344848

Edwin Alejandro Mojarras García | A01644376

Fernando Menchaca

Diseño de sistemas en chip

10 de Junio de 2025

Tecnológico de Monterrey

Este proyecto consiste en el diseño y la implementación de un procesador RISC-V basado en una arquitectura pipeline de 5 etapas, desarrollado en Verilog, es capaz de simular un conjunto de inst instrucciones que son subidas en por el archivo program.hex. Además de detectar hazards de control y datos.

Como se ha mencionado la arquitectura pipeline se divide en 5 etapas, las cuales se organizan en:

- *IF (Instruction Fetch):* En esta etapa se accede a la memoria de instrucciones usando el contador de programa (PC) y se obtiene la instrucción completa. Solo se obtiene a la instrucción, aún no se decodifica ni se interpreta el opcode, eso ocurre en la siguiente etapa.
- *ID (Instruction Decode):* Decodifica la instrucción, se extraen los registros y extiende inmediatos.
- *EX (Execute):* Realiza operaciones indicadas ya sea suma, resta, and, or, etc aquí se usa el módulo de ALU y calcula direcciones para acceso a memoria.
- *MEM (Memory Access):* Realiza operaciones de lectura o escritura a memoria de datos si es requerido el acceso a memoria, se realizan operaciones que tengan que ver con esta, como sw y lw .
- *WB (Write Back):* Escribe los resultados de vuelta en los registros ya sea un valor calculado o leído de la memoria se sube.

En todo caso siempre pueden ocurrir errores frente a la dependencia de datos, o instrucciones de salto, por lo que un módulo de manejo de Hazards es requerido, en este caso se detectan:

Hazards de datos:

Ocurren cuando una instrucción necesita un dato que aún no ha sido escrito por una instrucción anterior Para evitar errores, el procesador puede resolver de dos formas:

Stalls: se detiene temporalmente el avance del pipeline, permitiendo que la instrucción anterior termine y escriba el dato necesario.

Forwarding: si el dato ya fue calculado pero aún no se ha escrito en el registro, se dirige directamente hacia la etapa que lo necesita, evitando así detener el flujo.

Hazards de control:

Aparecen principalmente con instrucciones de salto o branch, ya que el procesador no puede saber con certeza cuál será la siguiente instrucción a ejecutar hasta que se evalúe la condición del salto. Mientras se determina si el salto se tomará o no, el procesador puede:

NOPs: se cargan instrucciones vacías en el pipeline para evitar que instrucciones incorrectas se ejecuten. Después se eliminan cuando se conoce la dirección correcta a seguir. Esto se puede hacer mediante un flush (las instrucciones son únicamente 0s) o mediante un compilador que identifique este escenario y escriba NOPs para dar tiempo a decidir el camino del branch.

Flushes: Cuando la instrucción tipo branch sí resulta en un salto, es necesario resetear o borrar los datos de los registros que contienen las instrucciones que entraron antes que se determine el branch.

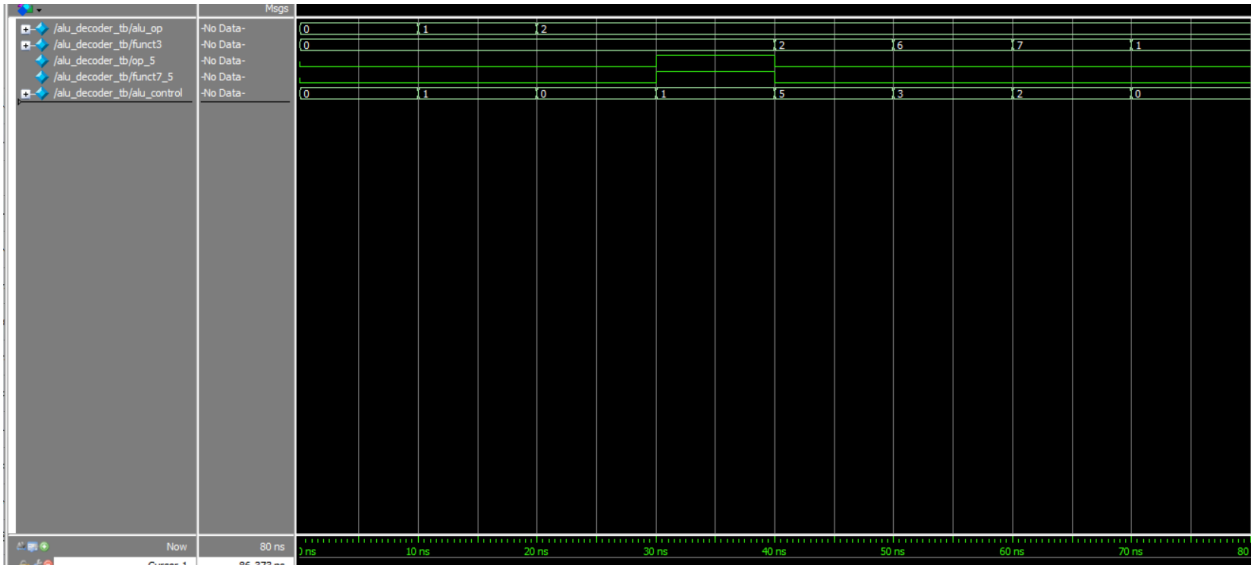
Este manejo de hazards asegura que las instrucciones se ejecuten en el orden y con los datos correctos para un buen funcionamiento del programa, manteniendo los resultados que daría un single cycle con la ventaja de un throughput más alto que es característico de la arquitectura *pipelined.*

Capturas de simulación

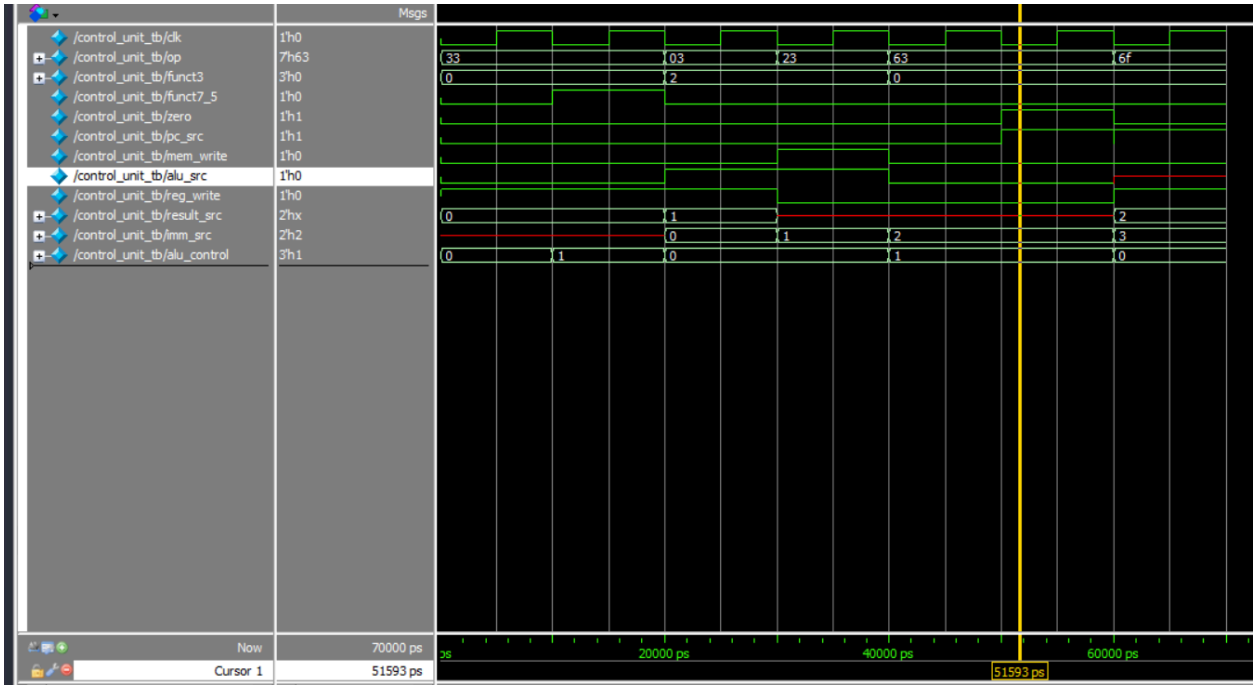
A continuación se muestran capturas de los testbenches de cada módulo por separado y al final la implementación de un módulo Top donde se muestra todo el funcionamiento del procesador, se muestra en la parte superior el nombre de cada archivo correspondiente a la imagen.

alu_tb.v

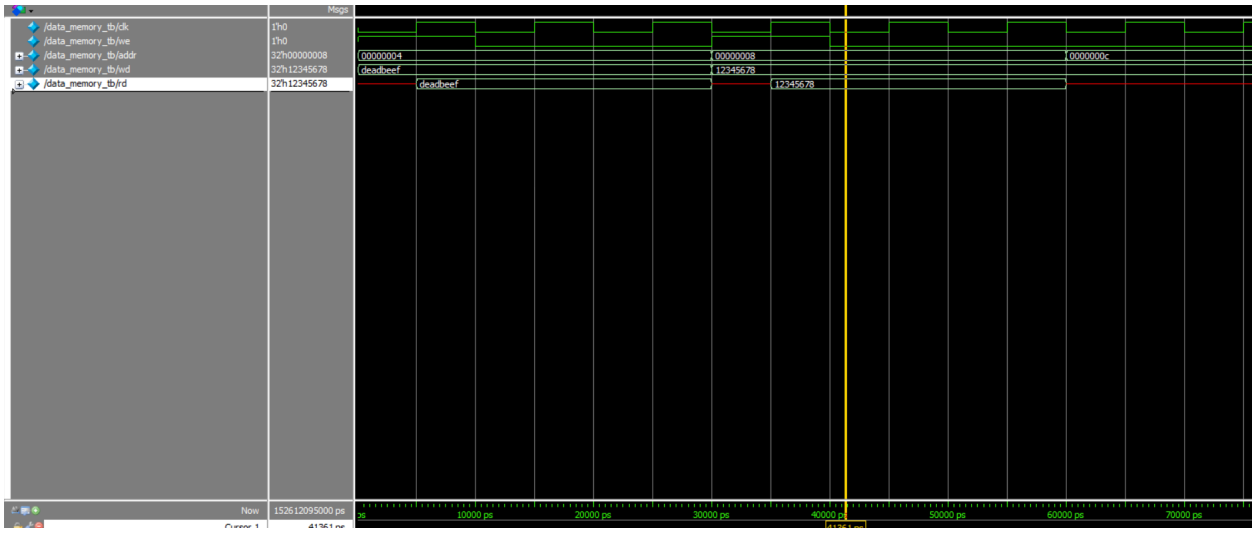
alu_decoder_tb.v



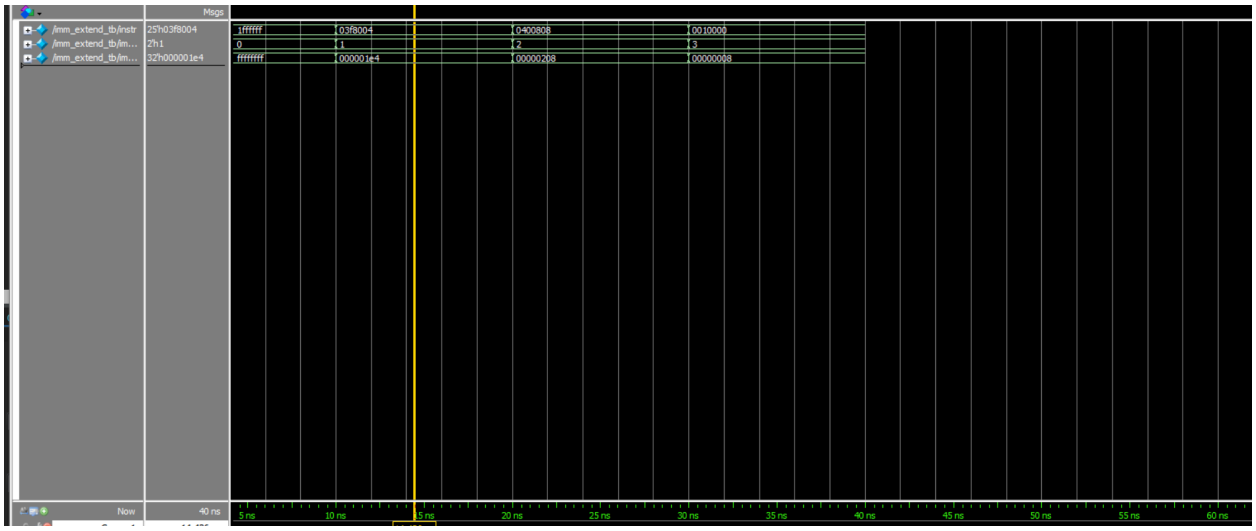
control_unit_tb.v



data_memory_tb.v

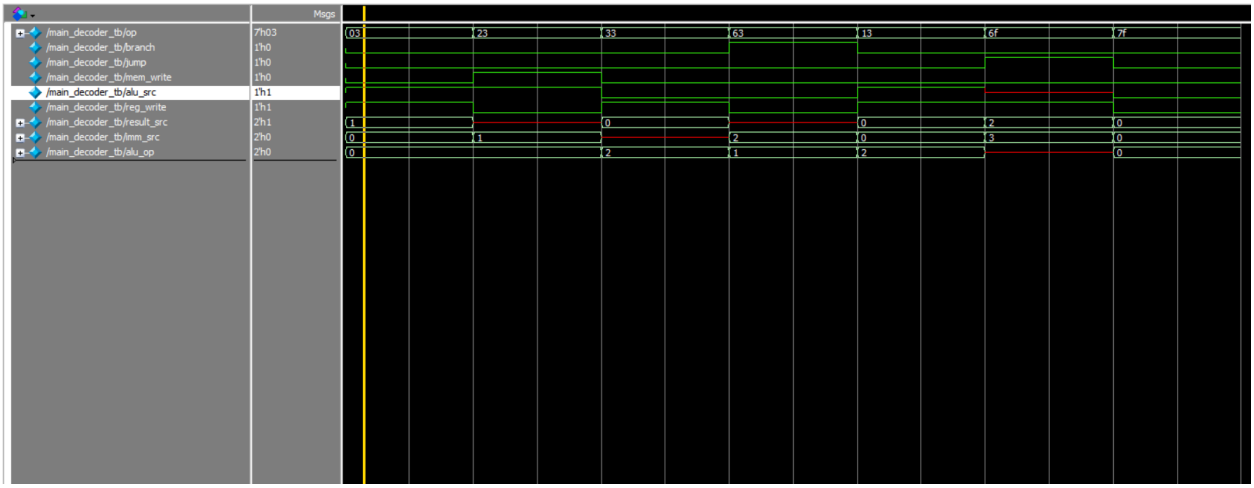


imm_extend_tb.v

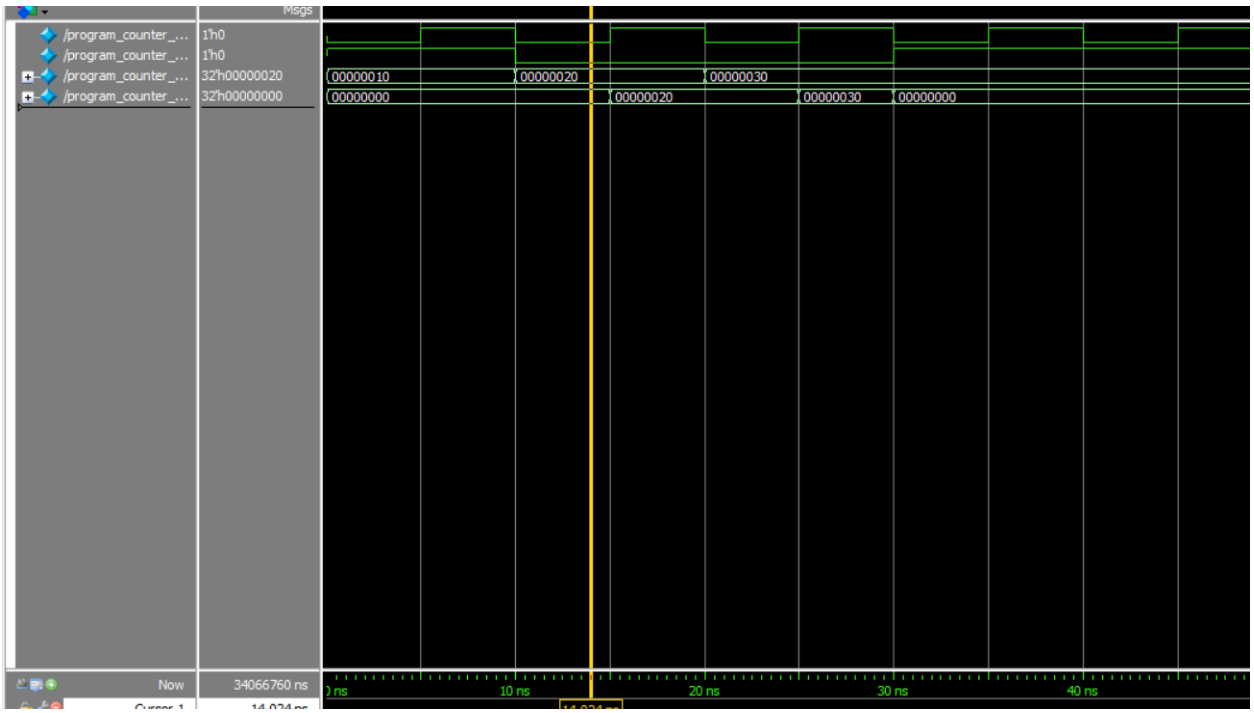


instruction_memory_tb.v

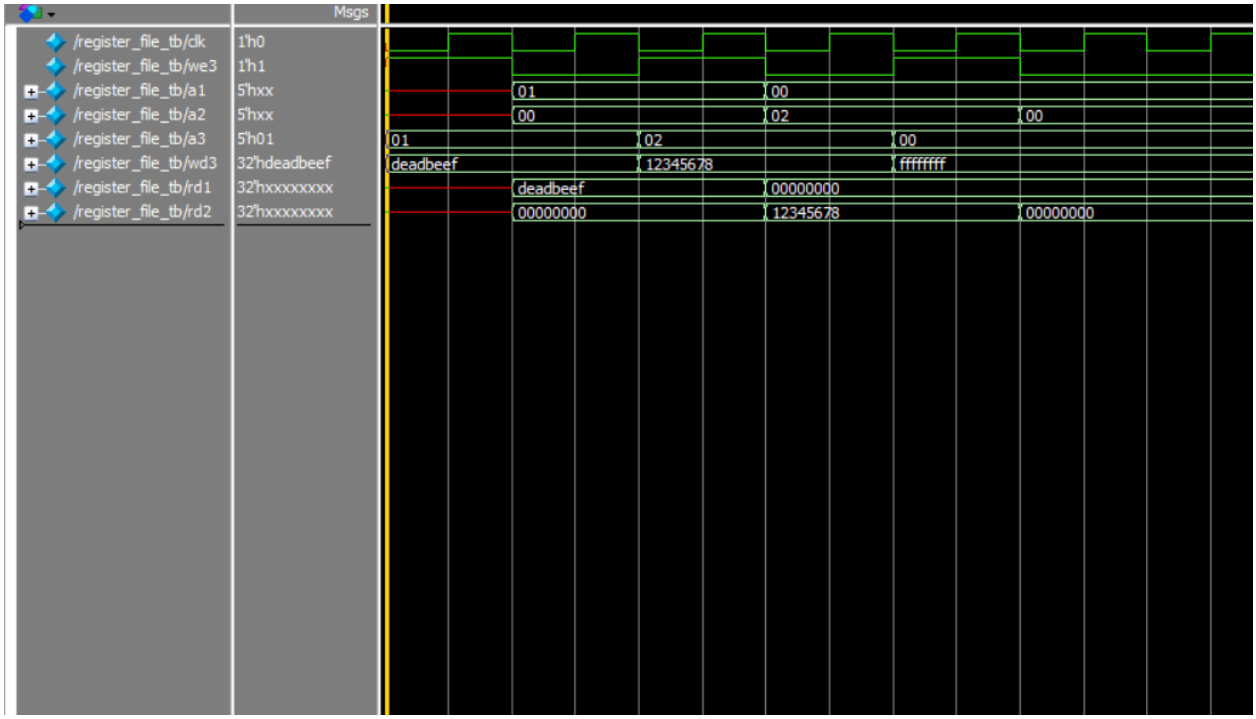
main_decoder_tb.v



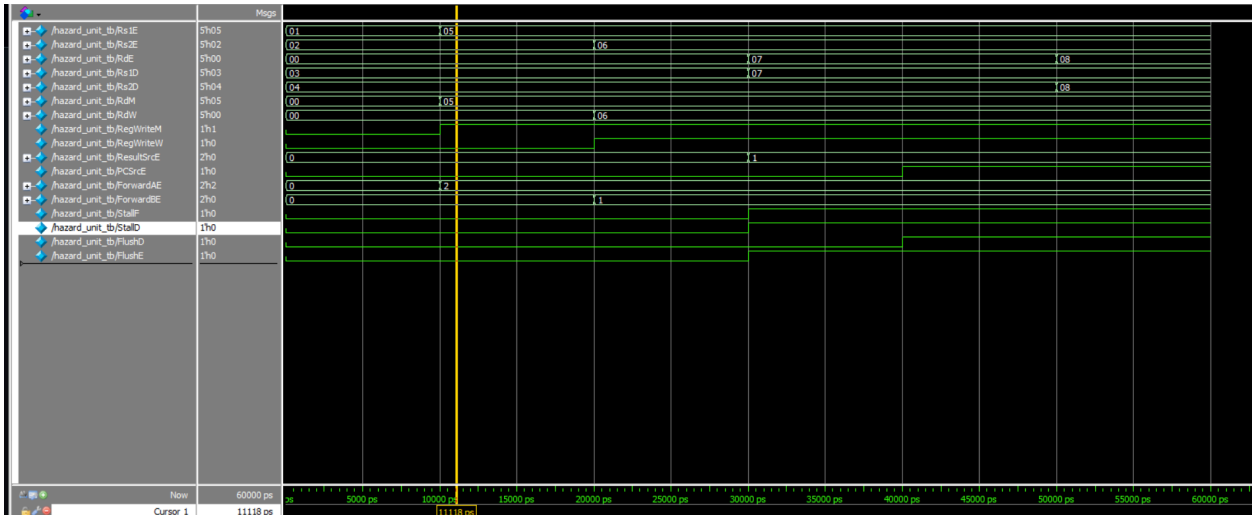
program_counter_tb.v



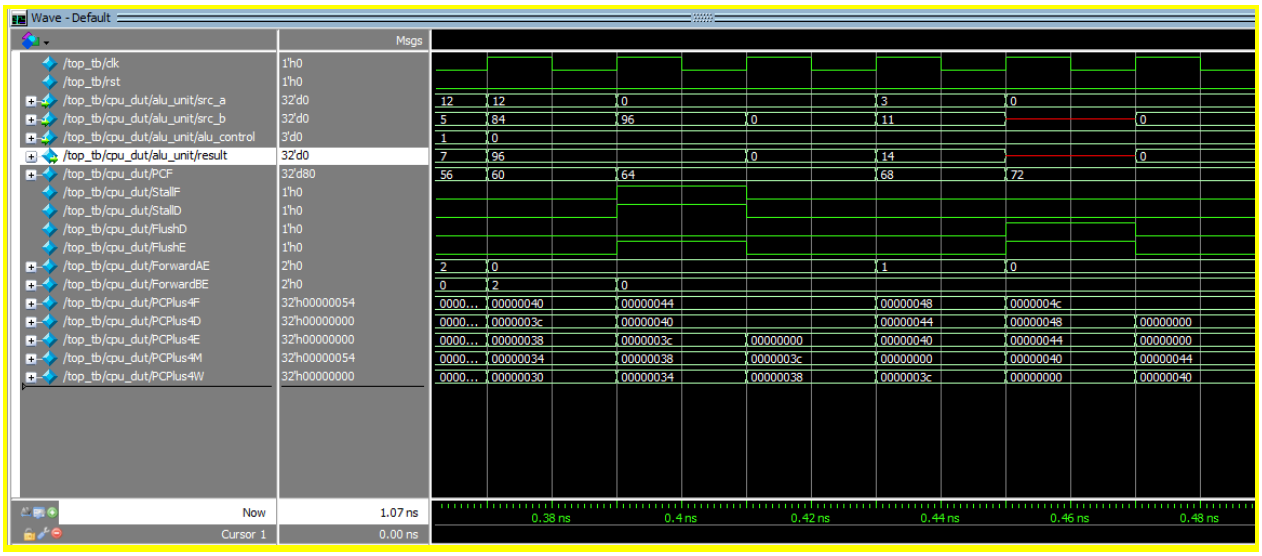
register_file_tb.v



hazard_unit_tb.v



TOP MODULE



A pesar de que puede ser difícil seguir el datapath por medio del *waveform,* hay cierta información en los resultados del testbench del top module que nos indican que el procesador está funcionando correctamente. En primer lugar, los datos que llegan a la alu se procesan de forma correcta de acuerdo al código de alu_control. En cuanto a la hazard unit, se observa que cuando se levanta el flag de stall, el program counter se queda en el mismo valor por un ciclo adicional.

Respecto al funcionamiento de pipeline, en donde valores de control y datos se guardan entre registros entre las diferentes etapas de procesamiento, se observa claramente como el program counter más antiguo pero con menor valor es el que está hasta la etapa de write back (última etapa), siendo el más nuevo el de la etapa de fetch (primera etapa); se aprecia que estos valores de PC van de mayor a menor desde la primera a última etapa del pipeline, confirmándonos que este aspecto del diseño funciona correctamente.

Por último, cuando se levanta la flag de flush en la etapa de execute, se observa que el program counter de esa etapa se resetea a 0, y este cero se propaga a la próxima etapa que es memory.

Cabe resaltar que en cada uno de los testbenches individuales se muestran los resultados esperados, por lo que al integrarlo funciona cuando se carga el archivo program.hex y se comprobó que funciona correctamente.

Por lo que se puede concluir que el procesador de RISC-V pipeline que implementamos es capaz de ejecutar programas de manera exitosa, además de tener una muy buena implementación ya

que cada uno de las etapas es modular por lo que facilita la comprensión, la organización y los test por módulos, además de tener un buen rendimiento y una unidad de control hazard funcional.