Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computadores

Fundamentos de arquitectura de computadores

Tarea 2

Métricas sobre un procesador RISC-V

Estudiante	Carné
Bolaños Barboza Gabriel	2022327511
Rodríguez Rojas Andrés	2019279722
Somarribas Montero Isaac	2020125516
Soto Varela Óscar	2020092336

Profesor:

Luis Alberto Chavarría Zamora

5 de junio de 2025

TEC | Tecnológico de Costa Rica

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computadores
CE 1107 — Fundamentos de Arquitectura de Computadores

Tarea 2
Métricas sobre un procesador RISCV

Fecha de asignación:	Lección 1 semana 14	Fecha de entrega:	Lección 2 semana 15
Grupo:	1 persona	Profesores:	Luis Chavarría Zamora

Para este taller debe usar el siguiente simulador de procesador RISC-V: Ripes [1] programando en ensamblador RISCV realice lo siguiente (para la investigación, discuta sobre la cantidad de accesos a memoria, saltos y operaciones con registros y relaciónelo contra teoría referenciada en IEEE):

1. Investigue, compruebe y documente internamente con un programa los accesos a memoria de un sistema (15 %). Muestre el código final en ensamblador y sus comentarios con las cuantificaciones de accesos a memoria. Trabaje con múltiples (miles) accesos a memoria.

2. Investigue, compruebe y documente internamente con un programa los saltos dentro de un programa (15 %). Muestre el código final en ensamblador y sus comentarios con las cuantificaciones de saltos. Trabaje con múltiples (miles) saltos.

3. Investigue, compruebe y documente internamente con un programa el uso de operaciones con registros (10 %). Muestre el código final en ensamblador y sus comentarios con la cantidad de operaciones realizadas entre registros. Trabaje con múltiples (miles) de operaciones.

Para cada programa realizado, extraiga la información de ejecución (estadísticas o **Execution info** dada por el software) usando los siguientes procesadores dentro de la configuración de Ripes:

1. Pipeline de 5 etapas sin hazard (25 %).

2. Pipeline de 5 etapas con hazard (25 %).

Para los procesadores donde no exista hardware que solucione riesgos, soluciónelo mediante software (transformación de código). Discuta en una tabla los tres aspectos de funcionamiento del sistema según los benchmarks recibidos. Relacione esta información con la teoría.

1. Entregables

1. Documento en PDF, desarrollado en LaTeX donde resuma con estructura de introducción, desarrollo y conclusión el documento (10 % de formato).

Si tienen dudas puede escribir al profesor al correo electrónico de Luis Chavarría. La entrega es por el TEC-Digital, por cada minuto de entrega tardía se rebaja un punto.

[1] Está disponible para Linux, Windows y MAC, adicionalmente hay ejemplos de las instrucciones en `Source Code`

I. Introducción

El presente trabajo tiene como objetivo analizar el comportamiento y desempeño de distintos tipos de instrucciones en un procesador basado en la arquitectura RISC-V, utilizando métricas de rendimiento recopiladas en el entorno simulado Ripes. Se considerarán específicamente tres categorías clave de instrucciones: accesos a memoria, saltos o bifurcaciones del flujo de control ("branches"), y operaciones aritméticas sobre registros. Para cada categoría se diseñó y ejecutó un programa representativo, observando su desempeño tanto en un pipeline de 5 etapas sin presencia explícita de "hazards", como en otro que sí los incluye.

A partir de estos escenarios se recopilan métricas como el número de ciclos de reloj, instrucciones retiradas, CPI (ciclos por instrucción), IPC (instrucciones por ciclo), y frecuencia de reloj, con el fin de contrastar los efectos de los "hazards" sobre la eficiencia del procesador.

II. Desarrollo

Los accesos a memoria y las operaciones sobre registros resultan fundamentales a la hora de llevar a cabo funciones vitales dentro de un procesador, del mismo modo se considera la importancia de los saltos o cambios de "branches" a la hora de llevar a cabo una solución implementada utilizando un ISA como RISC-V.

II-A. Accesos a memoria (`sw` y `lw`)

En la arquitectura RISC-V, los accesos a memoria son fundamentales y se realizan exclusivamente mediante instrucciones de carga ("load") y almacenamiento ("store"). Instrucciones como `lw` (load word) y `sw` (store word) permiten transferir datos entre la memoria y los registros, utilizando un formato con direccionamiento inmediato, lo cual favorece una implementación más eficiente del hardware [2].

En procesadores segmentados ("pipelined"), estas instrucciones afectan principalmente la etapa `MEM` del pipeline. Una gestión adecuada de accesos a memoria es clave para evitar "hazards" y mantener el rendimiento, ya que cualquier latencia o mal diseño puede provocar interrupciones en el flujo de instrucciones. Así, la correcta implementación de los accesos a memoria es crucial tanto para la funcionalidad como para la eficiencia del procesador [1].

II-A1. Programa: Accesos a memoria: A continuación (figura 1) se detalla el código fuente de un programa que escribe sobre 1000 direcciones de memoria (organizadas en un arreglo) y les asigna un valor numérico entero entre 0 y 999 inclusive. Posteriormente procede a leer cada dirección de memoria y a cargar su valor o contenido en un registro específico del sistema.

```
# PROGRAM THAT STORES AND LOADS AN INTEGER FROM 0 TO 999 IN 1000 MEMMORY REGISTERS.

.data

# Array to save 1000 integers in memory directions
array:  .word 4000

    .text
    .globl main

main:
    # t0 = 0: First value to store in array.
    li t0, 0

    # t1 saves the direction of the first position of the array.
    la t1, array

    # t2 = 1000. t2 is a mark number to manage the loop stop condition.
    li t2, 1000

storeLoop:
    # Stop condition. Stops when t0 equals t2 (1000 iterations).
    bge t0, t2, resett0

    # Calculates the memmory direction offset.
    slli t3, t0, 2

    # Determinates t4 as the first position of array plus offset.
    add t4, t1, t3

    # Saves t0 in array[t0]
    sw t0, 0(t4)

    # Modifies t0 to be t0+1 (t0++)
    addi t0, t0, 1

    # Restarts loop
    j storeLoop

resett0:
    # Resets t0 to 0 to reuse as index in the load loop.
    li t0, 0

loadLoop:
    # Stop condition. Stops when t0 equals t2 (1000 iterations).
    bge t0, t2, end

    # Calculates the memmory direction offset.
    slli t3, t0, 2

    # Determinates t4 as the position of array[t0].
    add t4, t1, t3

    # Loads the value at array[t0] into t5.
    lw t5, 0(t4)

    # Modifies t0 to be t0+1 (t0++)
    addi t0, t0, 1

    # Restarts loop
    j loadLoop

end:
    nop
```

Figura 1: Programa que ejemplifica el acceso a memoria.

II-A2. Información de ejecución: Pipeline de 5 etapas sin "hazard": Para la ejecución sin "hazard" se recopilaron las métricas de la tabla I.

Información de ejecución	
Ciclos	8016
Instrucciones retiradas	6008
CPI	$1,33$
IPC	$0,75$
Frecuencia de reloj	$3,38\ khz$

Cuadro I: Información de ejecución para el programa que ejemplifica los accesos a memoria en su versión sin "hazard".

II-A3. Información de ejecución: Pipeline de 5 etapas con "hazard": Para la ejecución con "hazard" se recopilaron las métricas de la tabla II.

Información de ejecución	
Ciclos	16016
Instrucciones retiradas	12008
CPI	$1,33$
IPC	$0,75$
Frecuencia de reloj	$495,48\ khz$

Cuadro II: Información de ejecución para el programa que ejemplifica los accesos a memoria en su versión con “hazard”.

II-A4. Análisis de métricas obtenidas: La comparación entre las dos versiones del programa muestra que la introducción de “hazards” duplica tanto la cantidad de instrucciones ejecutadas como los ciclos necesarios para completar la tarea. Aunque el CPI (ciclos por instrucción) se mantiene constante en ambas versiones, esto puede deberse a que los “hazards” hayan sido tratados mediante una reorganización del código que evita incidencias explícitas, a costa de mayor longitud y complejidad en la ejecución.

La versión sin “hazards” resulta claramente más eficiente, ejecutando menos instrucciones en menos tiempo, con un comportamiento más limpio del pipeline, lo que evidencia la importancia de minimizar dependencias entre instrucciones y manejar cuidadosamente el acceso a memoria y el uso de registros [2].

II-B. Saltos y cambios de “branch” (`b` y `j`)

En la figura 2 se detalla el código fuente de un programa que realiza cambios de “branch” o saltos de manera reiterada hasta que sus contadores respectivos alcancen el valor en que el programa debe terminar.

```
# PROGRAM THAT EXECUTES TWO LOOPS, ADDERLOOP ADDS AN UNIT TO t2 UTIL t2 REACHES t1
VALUE, COMPARATOR LOOP ADDS AN UNIT TO t1 UNTIL t1 REACHES t0 VALUE.

main:
    # Stop mark for comparatorLoop.
    li t0, 1000

    # Stop mark for adderLoop.
    li t1, 0

    # Jumps to comparatorLoop.
    j comparatorLoop

adderLoop:
    # Stop condition. Stops when t2 equals t1.
    bge t2, t1, comparatorLoop

    # Adds an unit to t2.
    addi t2, t2, 1

    # Restarts loop
    j adderLoop

comparatorLoop:

    # Stop condition. Stops when t1 equals t0 (1000 iterations).
    bge t1, t0, end

    # Adds an unit to t1.
    addi t1, t1, 1

    # Counter for aderLoop.
    li t2, 0

    # Jumps to adderLoop
    j adderLoop


end:
    nop
```

Figura 2: Programa que demuestra los saltos y cambios de “branches”.

II-B1. Programa: Saltos y cambios de “branch”:

II-B2. Información de ejecución: Pipeline de 5 etapas sin “hazard”: Para la ejecución sin “hazard” se recopilaron las métricas de la tabla III.

Información de ejecución	
Ciclos	117780
Instrucciones retiradas	70581
CPI	$1,67$
IPC	$0,599$
Frecuencia de reloj	$1,74\ khz$

Cuadro III: Información de ejecución para el programa que ejemplifica los saltos y cambios de "branch" en su versión sin "hazard".

II-B3. Información de ejecución: Pipeline de 5 etapas con "hazard": Para la ejecución con "hazard" se recopilaron las métricas de la tabla IV.

Información de ejecución	
Ciclos	2511513
Instrucciones retiradas	1506505
CPI	$1,67$
IPC	$0,6$
Frecuencia de reloj	$313,43\ khz$

Cuadro IV: Información de ejecución para el programa que ejemplifica los saltos y cambios de "branch" en su versión con "hazard".

II-B4. Análisis de las métricas obtenidas: El análisis de estos resultados revela un impacto significativo de los "hazards" en un programa centrado en operaciones de salto y cambios de "branch"; en la versión sin "hazards", el programa ejecuta alrededor de 70000 instrucciones a lo largo de 117000 ciclos, con un CPI moderadamente alto $(1,67)$. Lo anteriormente descrito sugiere que, incluso en ausencia de conflictos explícitos, los saltos y bifurcaciones introducen penalizaciones en el pipeline, principalmente debido a interrupciones del flujo secuencial y predicciones de "branch" que requieren de un manejo interno específico.

En la versión con "hazards", tanto los ciclos como las instrucciones se multiplican notablemente con respecto a su versión sin "hazards", esta diferencia indica que la resolución de "hazards" asociados a los "branches" exige una gran cantidad de instrucciones adicionales, verificaciones o sincronizaciones que eviten resultados erróneos al modificar el flujo de control. A pesar de ello, el CPI se mantiene en $1,67$ y el IPC varía levemente, lo que refuerza la idea de que la eficiencia por instrucción no se ha visto afectada.

II-C. Operaciones con registros (`add`, `sub`, `mul` y `div`)

El código de la figura 3 detalla el comportamiento de un programa que realiza cuatro posibles operaciones sobre registros, llevando a cabo un total de 1000 operaciones sobre los mismos y almacenando el resultado de la operación realizada en cada iteración en un registro específico reservado para dicho propósito.

```
# PROGRAM THAT OPERATES t1 DEPENDING ON ITS OWN VALUE AND SOME MILESTONES. FROM t3
TO t4 ADDS a0 TO t1, FROM t4 TO t5 SUBTRACTS a0 FROM t1, FROM t5 TO t6 MULTIPLIES t1
BY a1 AND IF t1 IS GREATER THAN t6 THEN DIVIDES t1, BY a1. RESULTS ARE STORED IN t2.

main:
    # Stop mark for mainLoop.
    li t0, 1000

    # Counter for mainLoop.
    li t1, 0

    # Milestone for adderBlock.
    li t3, 0

    # Milestone for subtractorBlock.
    li t4, 250

    # Milestone for multiplierBlock.
    li t5, 500

    # Milestone for dividerBlock.
    li t6, 750

    # Value to operate on addition/subtraction.
    li a0, 1

    # Value to operate on multiplication/divition.
    li a1, 2

    # Jumps to comparatorLoop.
    j mainLoop

adderBlock:
    # Adds a0 to t1 and stores it in t2.
    add t2, t1, a0

    # Jumps back to mainloop
    j mainLoop

subtractorBlock:
    # Subtracts a0 from t1 and stores it in t2.
    sub t2, t1, a0

    # Jumps to comparatorLoop.
    j mainLoop

multiplierBlock:
    # Multiplies t1 by a1 and stores it in t2.
    mul t2, t1, a1

    # Jumps to comparatorLoop.
    j mainLoop

dividerBlock:
    # Divides t1 by a1 and stores it in t2.
    div t2, t1, a1

    # Jumps to comparatorLoop.
    j mainLoop

mainLoop:
    # Stop condition. Stops when t1 equals t0 (1000 iterations).
    bge t1, t0, end

    # Adds an unit to t1.
    addi t1, t1, 1

    # Integer to store operations results.
    li t2, 0

    # Jumps to dividerBlock if t1 reaches dividerBlock milestone (t6).
    bge t1, t6, dividerBlock

    # Jumps to multiplierBlock if t1 reaches multiplierBlock milestone (t5).
    bge t1, t5, multiplierBlock

    # Jumps to subtractorBlock if t1 reaches subtractorBlock milestone (t4).
    bge t1, t4, subtractorBlock

    # Jumps to adderBlock if t1 reaches adderBlock milestone (t3).
    bge t1, t3, adderBlock

end:
    nop
```

Figura 3: Programa que ejemplifica las operaciones con registros.

II-C1. Programa: Operaciones con registros:

II-C2. Información de ejecución: Pipeline de 5 etapas sin "hazard": Para la ejecución sin "hazard" se recopilaron las métricas de la tabla V.

Información de ejecución	
Ciclos	11517
Instrucciones retiradas	7509
CPI	$1,53$
IPC	$0,652$
Frecuencia de reloj	$308,04\ khz$

Cuadro V: Información de ejecución para el programa que ejemplifica las operaciones con registros en su versión sin "hazard".

II-C3. Información de ejecución: Pipeline de 5 etapas con "hazard": Para la ejecución con "hazard" se recopilaron las métricas de la tabla VI.

Información de ejecución	
Ciclos	11516
Instrucciones retiradas	7508
CPI	$1,53$
IPC	$0,652$
Frecuencia de reloj	$1,44\ khz$

Cuadro VI: Información de ejecución para el programa que ejemplifica las operaciones con registros en su versión con "hazard".

II-C4. Análisis de métricas obtenidas: En este caso, el comportamiento del programa que ejecuta operaciones con registros muestra una diferencia prácticamente nula entre su versión con y sin "hazards". Tanto el número de instrucciones retiradas como la cantidad de ciclos son casi idénticos en ambas ejecuciones, y métricas como el CPI $(1,53)$ y el IPC $(0,652)$ se mantienen sin variaciones.

Lo anterior indica que los posibles "hazards" asociados al uso compartido de registros fueron o bien inexistentes o resueltos de forma natural por la estructura del programa o del pipeline.

La única diferencia observable es en la frecuencia de reloj, dado que se trata de un entorno simulado, este valor no implica un cambio real en el rendimiento.

III. Conclusión

A lo largo del análisis realizado en este documento, se observaron diferencias sustanciales en el comportamiento del procesador según el tipo de instrucción y la presencia o ausencia de "hazards". En particular, las instrucciones relacionadas con accesos a memoria y con saltos o cambios de "branch" mostraron una mayor sensibilidad a estos conflictos, generando un aumento notable tanto en la cantidad de ciclos necesarios como en el número total de instrucciones ejecutadas en los escenarios con "hazards", lo que refleja la complejidad inherente al manejo de dependencias y predicciones dentro del pipeline.

Por otro lado, las operaciones aritméticas sobre registros demostraron ser más resilientes frente a la introducción de "hazards", manteniendo métricas prácticamente equivalentes entre ambas versiones del programa. Este resultado sugiere que, al no involucrar accesos a memoria ni cambios de flujo, las dependencias en operaciones entre registros pueden ser minimizadas con facilidad o directamente evitadas mediante técnicas y algoritmos internos del pipeline [2].

IV. Código fuente

En el siguiente repositorio se encuentra el código fuente de los programas implementados para el desarrollo de este trabajo.

Repositorio - Tarea 2: https://github.com/JoseAndres216/FAC_Tarea2

V. Referencias bibliográficas

[1] David Money Harris y Sarah L. Harris. *Digital Design and Computer Architecture*. 2nd. Morgan Kaufmann, 2012. ISBN: 978-0-12-394424-5.

[2] David A. Patterson y John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. 2nd. Morgan Kaufmann, 2020. ISBN: 978-0-12-820331-6.