ECCI-UCR **II-2020, Lunes 19 de Octubre**
CI-0120 ARQUITECTURA DE COMPUTADORAS **Prof. Ileana Alpízar A.**
Entrega final: 19 de Noviembre

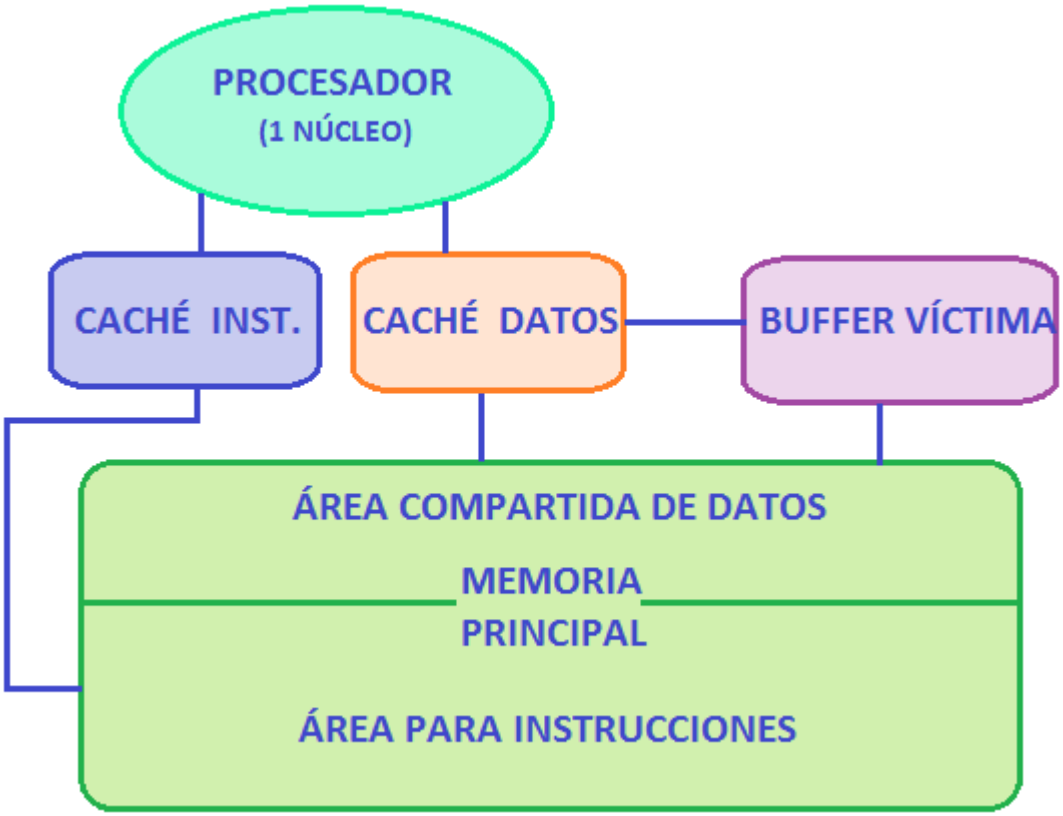
PROYECTO PROGRAMADO (valor 32%)

Simulación de un procesador RISC-V para enteros de un solo núcleo

I. Su trabajo consiste en:

Diseñar y programar la **simulación de un procesador de un núcleo, con un subconjunto de las características y funcionalidades de un procesador RISC-V.** El procesador simulado debe ser capaz de correr cualquier grupo de hilos[1] pertenecientes a un mismo proceso (por lo tanto comparten un área de memoria), hilos que se hayan escrito usando instrucciones del grupo de instrucciones RISC-V que se indican más adelante.

Su programa debe usar paralelismo, siendo **indispensable** que trabaje con las **herramientas de sincronización** que su lenguaje de alto nivel provea.



II. Descripción del sistema a simular

1. Descripción general del procesador

Un procesador RISC-V solo para operaciones de enteros, que pueda ejecutar hilos de un mismo proceso (a los que de ahora en adelante llamaremos **"hilillos").** Procesador de un solo núcleo, el cual tiene una caché L1 de instrucciones, una caché L1 de Datos, un buffer víctima y

[1] Siempre y cuando estos puedan ser almacenados en la memoria simulada para el procesador. Más abajo se indican las dimensiones de dicha memoria.

la memoria principal en la que se distingue un área para memoria compartida de datos, y un área para instrucciones (no se trabajará con área de memoria local para datos).

Por simplicidad, no se pide que este núcleo sea *pipelineado* en su ejecución de instrucciones, por lo que los registros necesarios de simular son un **PC**, un **IR**, los **32 registros** de propósito general para enteros **(x0, x1, ... x31)** y el registro **RL** para la coherencia de caché.

La asignación de CPU la hace el "sistema operativo" [2] con el algoritmo *Round Robin.*

2. Descripción de la jerarquía de memoria

- La **caché de instrucciones** es de mapeo directo, con capacidad para **8 bloques de dos palabras cada uno** (cada bloque entonces tendrá dos instrucciones, cada una compuesta por 4 enteros, según se explica más adelante)

- La **caché de datos** tiene capacidad para **4 bloques de 2 palabras** (cada palabra contiene un dato, el cual será un entero). La **estrategia para acierto (hit) de escritura** es **ñWRITE BACKò** (escribe solo en caché, y la actualización a memoria se hace **cuando** se va a **reemplazar el bloque**). La estrategia para fallo de escritura será **ñ WRITE ALLOCATEò**, es decir se sube el bloque a caché (Se trae de memoria principal- , pero si está en el buffer víctima, se debe traer de ahí, sin afectar el hecho de que este bloque se debe escribir a memoria). La estrategia de asignación de bloquesde esta caché es **asociativa por conjuntos de 2 vías**, y en cada conjunto se trabaja con **LRU como estrategia de reemplazo**.

- El **buffer víctima** (o buffer de escritura para víctimas) tiene capacidad para **8 bloques de 2 palabras.** Su **función es la de liberar al núcleo de la espera de la escritura a memoria de un bloque modificado** de la caché de datos L1 cuando va a ser reemplazado (bloque víctima). Para ello el bloque se debe copiar al buffer víctima, y es este buffer el que se encargará de que se copie a memoria. En el núcleo por mientras, se continúa con la solución del fallo de caché, es decir, se procede a subir el bloque que se necesita.

  En este buffer, se puede copiar **un bloque víctima "M"**, solo si hay espacio disponible, es decir, si existe al menos un espacio (de los 8 que posee el buffer) en el que no hay un bloque en proceso de escritura a memoria**,** o en espera de que inicie dicha escritura. **Si no hay espacio disponible, el núcleo debe esperar** a que haya espacio para almacenar el bloque "M" que desea reemplazar.

  Si el bloque que se va a copiar al buffer ya se encuentra ahí, producto de un reemplazo anterior, entonces, **si este bloque no se ha comenzado a copiar a memoria**, la nueva versión debe ser escrita sobre esta vieja versión del bloque (**merging buffer**). Si ya se comenzó a escribir a memoria, entonces se debe colocar en otra posición del buffer.

  El orden de escritura a memoria de los bloques del buffer víctima es **FIFO.**

  Notar que para resolver un fallo en caché L1 de datos (sobre un bloque víctima "M" o no), el bloque necesitado podría estar en el buffer víctima, de donde debe copiarse a la

[2] Al no solicitarse la simulación también de un sistema operativo, su programa deberá ofrecer toda la funcionalidad necesaria que haría el S.O.

caché. Si se está escribiendo a memoria, se espera a que se escriba, y luego se debe leer de memoria, pues el buffer víctima libera (o pone como disponible) inmediatamente ese espacio.

Para el buffer víctima hay un bus y puerto exclusivos para acceso a memoria.

- **Memoria principal:** Estará compuesta por 128 bloque. Tendrá una parte dedicada a la memoria compartida de datos, y otra para instrucciones. Los **bloque son de 2 palabras**, y las **direcciones de memoria deben ser alineadas.**

  Del bloque 0 al 47 (48 bloques) está el área para datos, **memoria compartida**.

  Del bloque 48 al 127 (80 bloques) está el **área para instrucciones**.

  En total **128 bloques de 2 palabras cada uno**.

0		1		2		3		4		5		6		7	
0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
8		9		10		11		12		13		14		15	
64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	124
16		17		18		19		20		21		22		23	
128	132	136	140	144	148	152	156	160	164	168	172	176	180	184	188
24		25		26		27		28		29		30		31	
192	196	200	204	208	212	216	220	224	228	232	236	240	244	248	252
32		33		34		35		36		37		38		39	
256	260	264	268	272	276	280	284	288	292	296	300	304	308	312	316
40		41		42		43		44		45		46		47	
320	324	328	332	336	340	344	348	352	356	360	364	368	372	376	380
48		49		50		51		52		53		54		55	
384	388	392	396	400	404	408	412	416	420	424	428	432	436	440	444
56		57		58		59		60		61		62		63	
448	452	456	460	464	468	472	476	480	484	488	492	496	500	504	508
64		65		66		67		68		69		70		71	
512	516	520	524	528	532	536	540	544	548	552	556	560	564	568	572
72		73		74		75		76		77		78		79	
576	580	584	588	592	596	600	604	608	612	616	620	624	628	632	636
80		81		82		83		84		85		86		87	
640	644	648	652	656	660	664	668	672	676	680	684	688	692	696	700
88		89		90		91		92		93		94		95	
704	708	712	716	720	724	728	732	736	740	744	748	752	756	760	764
96		97		98		99		100		101		102		103	
768	772	776	780	784	788	792	796	800	804	808	812	816	820	824	828
104		105		106		107		108		109		110		111	
832	836	840	844	848	852	856	860	864	868	872	876	880	884	888	892
112		113		114		115		116		117		118		119	
896	900	904	908	912	916	920	924	928	932	936	940	944	948	952	956
120		121		122		123		124		125		126		127	
960	964	968	972	976	980	984	988	992	996	1000	1004	1008	1012	1016	1020

Figura que representa la memoria principal.
Se indica el número de cada bloque y la dirección de cada palabra.
Direcciones alineadas para palabras de 4 bytes.

3. Ejecución de hilos de un mismo proceso (hilillos)

La **asignación de la CPU** entre se hará utilizando un algoritmo de planificación por turno rotatorio **(round robin**) con un **quantum de X** ciclos de reloj para cada ñhililloò Para efectos de la simulación, el valor del quantum será dado como un dato inicial por el usuario. Esto significa que cada vez que se saca de cpu un hilillo RISC-V, se debe **guardar su contexto**, es decir, el valor actual de su **PC y el de sus 32 registros**. Para ello no **use la memoria principal** que es parte del procesador simulado, si no una estructura de datos independiente. De igual manera, cuando le vuelve a tocar tiempo de cpu, este contexto se debe copiar a las estructuras del procesador. (**Context switch**)

Cuando un hilillo está ñgastandoò su **quantum**, pueden darse 2 situaciones que afectan el control: una, que termine por completo su ejecución; y la otra es que acabe su quantum pero

que aún necesite más tiempo de procesador, en este caso el hilillo pasará al final de la cola que maneja el sistema operativo a esperar por su próximo turno de tiempo de procesador. Note que también puede ocurrir que el hilillo finalice cuando termina su quantum. **Al finalizar el quantum** de un hilillo ya no se leen más instrucciones para este, pero se saca hasta que la instrucción en ejecución finalice, así esté resolviendo fallos de caché.

Al finalizar el quantum de un hilillo ya no se leen más instrucciones para este, pero se saca del procesador hasta que ya no quede ninguna de sus instrucciones en el pipeline.

4. Sincronización ofrecida por procesador

La sincronización entre los hilos (hilillos) RISC-V que ejecuta este procesador **se realiza solo mediante semáforos binarios** (mutex o locks) los cuales permiten a solo un hilillo como máximo el ingreso a una sección crítica (exclusión mutua). El procesador solo tiene dos primitivas de sincronización el **Load reserved (lr) y el Store conditional (sc), por lo que los hilillos que ejecute, realizan su sincronización utilizando solo dichas instrucciones.**

III. Detalles para la simulación y del funcionamiento del sistema simulado

5. Formato Decimal

Los hilillos RISC-V que ejecutará el procesador ya vendrán **codificados en "lenguaje de máquina"** de acuerdo con los formatos RISC-V. Cada uno será un archivo de texto con una instrucción por línea, pero NO en código binario, sino decimal, a razón de 4 enteros separados por un espacio en blanco por instrucción (ver más adelante).

Todas las estructuras de datos que se utilizarán para simular cachés, buffer víctima, memoria de datos, memoria de instrucciones (en la que se copiará cada uno de los hilillos de los archivos de texto), TLB´s (el equivalente al PCB de los procesos pero para hilos), memoria para instrucciones, memoria para datos, PC, IR, RL, Regs x1 a x31; albergarán entradas que representarán valores expresados en formato decimal, por lo que lo más simple es que sean **estructuras de datos de enteros**.

En particular, **se pide (no es una recomendación) que la estructura para memoria principal sea un vector de enteros**. **Se recomienda que se maneje aparte la memoria de datos compartidos y la memoria de instrucciones**, por lo que en ese caso serían 2 vectores de enteros.
Si se definen dos estructuras para memoria, debe hacerse respetando totalmente las direcciones que se indicaron en la tabla anterior:

- Ambas estructuras de datos deben ser solo **vectores simples de enteros**. Las direcciones mostradas arriba deben respetarse. Las instrucciones inician en la dirección 384, aún cuando sea un vector aparte del de datos.
- Notar que una instrucción, la cual se debe **simular de 4 bytes** para efectos de su dirección de memoria, está compuesta por 4 enteros (ver más abajo el formato de codificación).
- Cada **dato entero del área de datos también se debe simular que ocupa 4 bytes**, por lo que las direcciones de memoria de las entradas del vector de datos corresponderían a direcciones de memoria multiplicadas por 4, mientras que las direcciones de las instrucciones almacenadas en el vector de instrucciones correponderían solo al indice del vector + 384.

- En resumen, el vector de memoria de **datos** con **96 entradas**, una para cada palabra. Y el de memoria de **instrucciones**, que deberá poder albergar 160 instrucciones (palabras), y como cada instrucción está en formato decimal expresada con 4 enteros, necesitará 160*4= **640 entradas**.

Se utilicen o no dos vectores para memoria, notar que al tener 4 enteros por instrucción, será muy simple simular sus direcciones de memoria múltiplo de 4 ya que ***cada inicio*** de instrucción estaría en un subíndice del vector que es "múltiplo de 4".

6. Inicialización de cachés, buffer víctima y memoria principal

Por la lógica de los hilillos de prueba final, es necesario que la **memoria de datos** se inicialice con **el valor 1** en cada una de sus palabras. Esto es para simular que todos los candados fueron puestos por el ñhilillo principalò

Por claridad y para evitar errores, debe **inicializarse las cachés con ceros en cada entrada** y con un número de bloque igual a **-1** en cada posición y estado inválido. La **memoria de instrucciones debe inicializarse** con 0´s para poder notar los cambios con claridad. El buffer víctima debería tener todos sus espacios disponibles, con 0´s en las palabras de cada bloque, y con -1 como identificación para cada bloque.

7. Operaciones aritméticas

La manera de simular las operaciones aritméticas será simplemente calculando la operación **con instrucciones del lenguaje de alto nivel** elegido para la simulación.

8. Quantum, "context switch" y fin de hilillo

Cuando un hilillo está ñgastandoò su **quantum**, pueden darse 2 situaciones que afectan el control: una, que termine por completo su ejecución; y la otra es que acabe su quantum pero que aún necesite más tiempo de procesador, en este caso el hilillo pasará al final de la cola que maneja el *sistema operativo* a esperar por su próximo turno de tiempo de procesador. Note que también puede ocurrir que el hilillo finalice cuando termina su quantum. **Al finalizar el quantum** de un hilillo ya no se leen más instrucciones para este, pero se saca del procesador hasta que la instrucción en ejecución finalice, así esté resolviendo fallos de caché.

- **Fin de quantum para un hilillo**

Antes de asignar el siguiente hilillo de la cola al núcleo al finalizar el quantum del hilo que ejecutaba, **se salva el contexto de este hilo**, de manera tal que cuando le corresponda tiempo de CPU, se copien los valores de su contexto al núcleo. El contexto debe tener como mínimo el valor del PC (dirección de la siguiente instrucción a ejecutar de ese hilo) y el valor de los 32 registros.

Por simplificación, no se utilizará la memoria principal del procesador para almacenar ahí los contextos de cada hilillo, se usará una estructura de datos aparte.

- **Fin de hilillo**

Una vez que finaliza la ejecución de un hilillo, no se deben perder los últimos valores de los registros, ya que, junto con otros datos de la ejecución del jhilillos, se deben desplegar en pantalla al final de la simulación.

9. Manejo del tiempo en el procesador

- **Reloj del procesador**

Cada ciclo que pase de manera lógica en el núcleo y en el buffer víctima, deberá ser contabilizado y el avance al siguiente se debe hacer de manera sincronizada.
Para ello, tomar en cuenta que:

- **la finalización de una instrucción implica un ciclo de reloj**, es decir, si se lee una instrucción de la caché de instrucciones sin fallo de caché, y si se usa caché de datos, tampoco hay fallo, esa instrucción tardaría un ciclo de reloj nada más (como ocurriría cuando hay pipeline, y cada ciclo de reloj finaliza una instrucción si no hay algún retraso)
- **los accesos a memoria, tanto por parte de caché de instrucciones, caché de datos y buffer de escritura también implican gasto de tiempo.** Ciclos que se deberán contabilizar uno a uno.
- **Debe sincronizarse el reloj del núcleo y el del buffer víctima** (o manejar un único reloj para los 2). No es así como ocurre en la realidad, pero como el núcleo es simulado de manera independiente casi independiente del buffer, si no se tiene un "mismo reloj", se comportarían de manera muy irreal, ya que por ejemplo, mientras en el núcleo pasan 4 ciclos, para el buffer ya pasaron 50. Entonces lo que importa es que tanto en el núcleo como en el buffer, se cambie de ciclo de reloj al "mismo tiempo", aunque la longitud de ciclo de reloj no sea la misma cada vez. Esto se logra utilizando herramientas de sincronización para los hilos que ud. programe. Sugerencia (investigue "barreras")

- **Ciclos de retraso por accesos a memoria**

Para contabilizar los ciclos de retraso cuando se produce un fallo en alguna caché, se supondrá lo siguiente (aunque esto no se debe simular, tan solo se explica para comprender el tiempo que se tarda):

Como los bloques son de 2 palabras, y para subir un bloque a memoria (o para escribir un bloque a memoria en el caso del buffer víctima) se hace palabra por palabra, el total de ciclos de reloj que se tarda es:
2 * (lo que se tarda en leer o escribir un palabra de o en memoria)=
= 2 * (transmisión de dirección de mem. en el bus +latencia de la memoria para buscar la palabra + transmisión en el bus a caché de la palabra solicitada en el bus) =

2 * (1 + 10 +1) = **24 ciclos de reloj**

Notar que no se pide simular los buses, tan solo se trabajará simulando lo que resulta si se utilizaran y **contabilizando el tiempo que tardan**.

- **Retrasos buffer víctima**

Como se indicó, el buffer víctima le ahorra tiempo al núcleo al encargarse de las escrituras a memoria, sin embargo:

- el núcleo debe **copiar al buffer cada bloque** que se deba escribir a memoria, el tiempo para esto es de: **4 ciclos de reloj**
- De igual manera, si se va a resolver un fallo de caché de datos con un bloque que está en el buffer víctima, **la copia del bloque a la caché** tarda también **4 ciclos de reloj**.
- Estos tiempos no retrasan al buffer víctima en su trabajo de escrituras a memoria. Pero igual se contabilizan uno a uno en sincronía.

Por otra parte, el **buffer víctima al enviar a memoria un bloque** para su escritura, tiene un retraso de **24 ciclos de reloj**: 2*(1+1+10) envía 2 palabras, para cada una de ellas envía en bus su valor y su dirección, y luego 10 ciclos de latencia de la memoria. Estos ciclos de reloj no provocan retraso al núcleo, solo al buffer, pero deben ser contabilizados uno a uno en sincronía con el núcleo.

- **Manejo de los ciclos de reloj de manera sincronizada**

Si el procesador **debe realizar un acceso a memoria que se atrasa** (cuando la caché no lo resuelve en un ciclo de reloj), entonces la simulación deberá permitir que los ciclos de reloj continúen avanzando para que el buffer víctima pueda continuar trabajando mientras en éste se resuelve la solicitud de memoria. Si el buffer víctima no tiene campo para copiar un bloque "M" que se reemplazará, los ciclos que espera se **deben ir contabilizando uno a uno.** Al inicio de cada ciclo intenta ver si finalizó y si no, pues "avisa" para cambiar de ciclo. **NO es válido averiguar cuánto le falta al buffer para terminar de escribir un bloque y liberar el espacio**.

- **Muy importante**: ninguno de estos ciclos de retraso **se toman en cuenta para el quantum.**
- Note que entonces **no se tiene una longitud de ciclo de reloj fija**, es totalmente irregular y manejada por su programa.

10. Instrucciones de RISC-V que se implementarán y estructura de los hilillos de prueba

- **Instrucciones**

En el proyecto se implementarán las siguientes instrucciones RISC-V, **Tomado de RV32I Base Instruction Set, de RV32M Standard Extension y de RV32A Standard Extension (risc-spec-v2.2)**

0´s en este color se agregaron para completar 4 enteros por instrucción. Note que el inmediato ñnò siempre va en la cuarta posición.

A las instrucciones lr y sc se les llama **primitivas para sincronización**. Instrucciones como estas se utilizan para hacer las rutinas de sincronización de los sistemas operativos. Esas rutinas de sincronización del S.O. son "llamadas" por las herramientas de sincronización de los lenguajes de programación de alto nivel usadas por el programador cuando desea realizar algún tipo de sincronización. Lo que ocurre es que se da una interrupción para que la rutina del S.O. tome el control.

En el caso del procesador del proyecto y de los hilillos que correrán ahí, cuando se necesita hacer sincronización, se hace directamente (sin llamar a rutinas del sistema operativo) al incluir en el código las instrucciones **lr y el sc implementando candados, o mutex.**

Se define que un valor **0 en el ñlock o mutexò indica que está disponible, y un 1 que está ocupado**

INSTRUCCIÓN			CODIFICACIÓN				
Opera-ción	Operandos	Acción	0-6 y otros bits	7-11	15-19	20-24	Dif. bits
			Cód. Op. (DECIMAL)	rd	rf1	rf2	inmediato
addi	x1, x2, n	x1 ← x2 + n	19	x1	x2	n	
add	x1,x2,x3	x1 ← x2 + x3	71	x1	x2	x3	
sub	x1,x2,x3	x1 ←x2 - x3	83	x1	x2	x3	
mul	x1,x2,x3	x1 ←x2 * x3	72	x1	x2	x3	
div	x1,x2,x3	x1 ←x2 / x3	56	x1	x2	x3	
lw	x1, n(x2)	x1 ← M[n + x2]	5	x1	x2	n	
sw	x1, n(x2)	M[n + x2] ← x1	37	x2		x1	n
beq	x1,x2, etiq	si x1 = x2 PC += inm*4	99	x1		x2	n
bne	x1,x2, etiq	si x1 ≠ x2 PC += inm*4	100	x1		x2	n
lr	x1, x2	x1 ← M[x2] y establece el "link register" RL ← x2	51	x1		x2	0
sc	x1, n(x2)	Si RL = n+x2 entonces M(n + x2) ← x1 y mantien valor de x1 si no No escribe en memoria y x1← 0	52	x2		x1	n
jal	x1,n	x1←PC, PC← PC+n	111	x1	0		n
jalr	x1,x2,n	x1= PC ; PC = x2+n	103	x1	x2	n	
"FIN"		"Finaliza el programa"	999	0	0	0	

Notar que las instrucciones sombreadas en este color se salen un poco del "patrón" que siguen las demás para su codificación. ¡Tener cuidado con ello!

- **Hilillos**

Cada programa (hilillo) que debe ejecutar el procesador es un **archivo de texto** (.txt) para el que cada instrucción estará compuesta por **4 valores enteros separados entre ellos por un espacio en blanco**, y ocupa una línea (ver siguiente ejemplo)- Esto significa que, aunque debemos simular que ocupa una sola palabra de 4 bytes, en realidad se debe almacenar en una estructura de datos enteros, ocupando 4 de ellos. Por ejemplo, si el hilillo del siguiente ejemplo se guardara al inicio de la parte de instrucciones, y si la memoria de instrucciones se representa con un vector MI:

19	30	0	136	51	2	30	0	100	2	0	-2	19	1	0	1	52	30	1	0	etc

MI[0] = 19, MI[1] = 30, MI[2] = 0, MI[3] = 136, MI[4] = 51, M[4] = 2, ...

Pero el subíndice 0 equivale a la dirección de memoria 384 ya que la primera instrucción inicia en el "byte"384. El subíndice 4 equivale a la dirección de memoria 388 ya que la segunda instrucción estará 4 bytes después de la primera (a pesar de que todas las entradas del vector son enteros de 4 bytes).

Además se debe trabajar con bloques de 2 palabras, por lo que el **bloque 48,** que es el primer bloque de la parte de instrucciones de la memoria del procesador, contiene 2 instrucciones-las primeras 8 entradas de este vector MI:

19	30	0	136	51	2	30	0	100	2	0	-2	19	1	0	1

Ejemplo de un fragmento de código (solo ejemplo para codificación):

En este hilillo todas las instrucciones marcadas en color amarill**o implementan un candado** que está almacenado en memoria en la dirección 136, y como se ve se utilizan lr y sc.

codificado	etiq.	instrucción	
19 30 0 136		**addi**	**x30, x0, 136**
51 2 30 0	**Allá**	**lr**	**x2, x30**
100 2 0 -2		**bne**	**x2, x0, Allá**
19 1 0 1		**addi**	**x1, x0, 1**
52 30 1 0		**sc**	**x1, 0(x30)**
99 1 0 -5		**beq**	**x1, x0, Allá**
19 17 0 176		**addi**	**x17, x0, 176**
19 1 0 1	**mal**	**addi**	**x1, x0, 1**
51 2 17 0	**AHÍ**	**lr**	**x2, x17**
52 17 1 0		**sc**	**x1, 0(x17)**
99 1 0 -4		**beq**	**x1,x0, mal**
100 2 0 -4		**bne**	**x2,x0, AHÍ**
19 3 0 5		**addi**	**x3,x0,5**
999 0 0 0		**FIN**	

El archivo de texto sería:

```
19 30 0 136
51 2 30 0
100 2 0 -2
19 1 0 1
52 30 1 0
99 1 0 -5
19 17 0 176
19 1 0 1
51 2 17 0
52 17 1 0
99 1 0 -4
100 2 0 -4
19 3 0 5
999 0 0 0
```

11. Para la ejecución de su simulación

- **Al inicio de la simulación**

  - se le solicitan al usuario los **hilillos** a correr en el procesador. El simulador debe localizar estos hilillos en el mismo directorio en el que se encuentra el simulador

  - Cada hilillo a ejecutar estará escrito en un archivo de texto en donde **cada línea es una instrucción RISC-V ya codificada en lenguaje de máquina de acuerdo a como se indicó.** (Su programa puede también ofrecer la facilidad de localizar el directorio en el que se encuentran dichos archivos)

  - su programa debe leer uno a uno todos los hilillos del directorio indicado y los **debe colocar en la memoria de instrucciones simulada** del procesador guardando el PC para cada uno de ellos en su estructura para su contexto. Inicialice en 0 todos los registros en esta estructura para claridad.

  - Debe solicitar al usuario el valor de "**X" el quantum** (valor mínimo lógico de 10 ciclos)

  - Luego de esto se comienza **asignando un hilillo al núcleo** (copiando su contexto). El hilillo se ejecutará hasta que se acabe el quantum o finalice, lo que ocurra primero.

- **Al finalizar la simulación** debe desplegarse en pantalla
  - El contenido de la memoria compartida de datos (las 96 direcciones y su contenido)
  - el contenido de la caché de datos describiéndose claramente para cada posición el número de bloque, su estado y el contenido de sus dos palabras.
  - el contenido del buffer víctima, y el estado y contenido de cada bloque (no necesariamente todo lo que se debía escribir a memoria se ha hecho cuando finalizan todos los hilillos)
  - Para cada hilillo que corrió:
    * el contenido de los 32 registros del procesador
    * la cantidad de ciclos que tardó su ejecución (solo uso de CPU)
    * el valor del reloj cuando comenzó su ejecución y de cuando finalizó

Ⅳ. Forma de realizar y entregar el proyecto:

- Grupos de trabajo de **3** estudiantes como máximo quienes deben designar a un **líder** de proyecto (responsable de organizar el trabajo, de reportar problemas en el grupo, de informar cómo trabaja el grupo, y de indicar el porcentaje de nota que le debe corresponder a cada integrante del grupo, dependiendo de en cuánto contribuyó al desarrollo del proyecto)

- El programa de simulación: se puede realizar en el **lenguaje de alto nivel que el grupo de trabajo escoja**, pero este lenguaje debe permitir manejar muy bien el paralelismo que se indicó

- El programa debe utilizar **arquitecturas paralelas**, es decir, es **indispensable** que se diseñe utilizando hilos que realmente puedan correr en paralelo y logren no solo aprovechar el paralelismo que permiten los procesadores de al menos doble núcleo sino que también permitan hacer una simulación más ñrealù

- Debe hacerse la **correcta sincronización** entre hilos

- La documentación interna debe dejar muy claros los **algoritmos utilizados para resolver cada problema**.

- Para la entrega se debe enviar:

  - Un "**screenshot**" de la pantalla con los resultados después de correr los hilillos de prueba que se le indicarán más adelante. (Nota importante: si los resultados para estos hilillos son los esperados, no significa que automáticamente su nota es 100. Razones: debe revisarse en la documentación y en el código la lógica utilizada para la solución de los problemas, las estructuras de datos, la eficiencia de la programación, etc.)

  - **El fuente** y el **ejecutable** (si aplica) del programa (debe hacerse el ejecutable, para enviarlo a gmail, cambie la extensión **".exe" por ".eje" o ...**) El código debe venir con una **completa y clara documentación interna.**

  - Un **manual muy claro de instalación si se requiere algo especial**

  - Lista de **problemas no resueltos** al tiempo de entrega e ideas sobre su solución

  - Una **nota** para todos y cada uno de los miembros del grupo de trabajo puesta por el líder de acuerdo con el desempeño del integrante en el desarrollo del proyecto (0 a 100) Con ese valor se calculará la nota de esta parte del proyecto para cada uno de los integrantes del grupo como: **nota obtenida en proyecto por el grupo * porcentaje asignado a ese integrante**.