Arquitectura de Computadoras 2

Alumno: Santiago Vietto

Docente: Agustín Laprovitta

Institución: UCC

Año: 2023

Abstracciones informáticas y tecnología

Introducción

Clases de computadoras

Computadoras personales: se caracterizan por tener buenas prestaciones a bajo costo, y permiten ejecutar programas de terceros o software estándar.

Servidores: estos se construyen con la misma tecnología que las computadoras personales, pero están basados en la red, y permiten un alto rendimiento y una mayor ampliación de su capacidad de entrada y salida.

Supercomputadoras: estas disponen de muchos procesadores, y se utilizan para cálculos científicos e ingeniería de alta calidad.

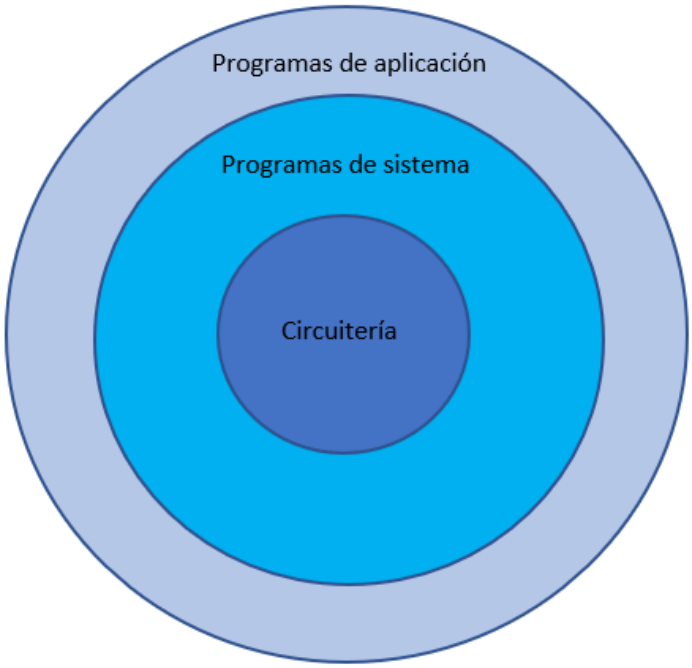
Computadoras embebidas: se refieren a los microprocesadores que se encuentran en diversos artefactos, y se diseñan para ejecutar un conjunto de aplicaciones relacionadas que están integradas al hardware.

Rendimiento

_ A continuación, mencionamos componentes de hardware y software, y como es que estos afectan a las prestaciones:

- Algoritmo: determina el número de sentencias de alto nivel, y el número de operaciones de E/S, que se ejecutarán.
- Lenguaje de programación, compilador y arquitectura: determinan el número de instrucciones máquina que se ejecutarán por cada sentencia de alto nivel.
- Procesador y sistema de memoria: determinan que tan rápido se pueden ejecutar las instrucciones.
- Sistema de E/S (hardware y sistema operativo): determina que tan rápido se pueden ejecutar las operaciones de E/S.

Bajo los programas



Programas de aplicación: o aplicaciones de software, son las que se escriben en lenguaje de alto nivel.

Programas de sistema: o software de sistemas, se dividen en sistemas operativos, donde se manejan las operaciones básicas de E/S, asignación de espacio de almacenamiento y memoria, entre otras.

Circuitería: o hardware, compuesto por el procesador, la memoria y los controladores de E/S.

Niveles de lenguaje

Lenguaje de alto nivel: está compuesto por palabras y notaciones algebraicas que el compilador puede traducir. Estos proporcionan productividad y portabilidad.

Lenguaje de bajo nivel: o Assembly, es la representación simbólica de las instrucciones de la máquina.

Lenguaje máquina: es la representación binaria de las instrucciones.

Bajo la cubierta

_ Los cinco componentes clásicos de un computador son:

- Componentes de entrada y salida: como dispositivos de interfaz de usuario (pantalla, teclado, ratón), dispositivos de almacenamiento (disco duro, CD/DVD, flash), y adaptadores de red (para comunicarse con otras computadoras).
- Memoria
- Camino de datos (data path)
- Control

_ El data path y el control están combinados, formando lo que se conocen como procesador. Tanto el hardware como el software están estructurados en niveles jerárquicos, donde cada uno oculta detalles al nivel superior.

Prestaciones/performance

Tiempo de respuesta y rendimiento

Tiempo de respuesta: es el tiempo total requerido por un computador para completar una tarea, incluidos los accesos a disco y memoria, las operaciones de E/S, sobrecarga del sistema operativo, etc. También se lo conoce como “tiempo de ejecución” ya que hace referencia al tiempo entre el inicio y el final de una tarea.

Rendimiento: o productividad, se define como el número de tareas que se completan, o el trabajo total realizado, por unidad de tiempo. También se conoce como “ancho de banda”. Por ejemplo, tareas o transacciones por hora. Para calcular el rendimiento relativo hacemos, decimos que X es n veces más rápido que Y, haciendo (el tiempo de ejecución de Y es mayor que el de X:

$$\text{Performanc e}_X / \text{Performanc e}_Y = \text{Execution time}_Y / \text{Execution time}_X = n$$

_ Siempre cuando hablemos de aumento de velocidad o mejora de rendimiento, trabajamos sobre el tiempo de ejecución, sin haber hecho la mejora, respecto del tiempo de ejecución con la mejora. Cualquier número mayor a 1 indica un aumento de velocidad.

Mediación de las performance

Tiempo de transcurrido: o tiempo total de respuesta, se refiere al tiempo total que tarda una tarea en completarse, incluyendo accesos o procesamientos de E/S, sobrecarga del sistema operativo, tiempo de inactividad y operaciones. Este determina el rendimiento del sistema.

Tiempo CPU: es el tiempo que la CPU dedica a ejecutar una tarea concreta, y no incluye el tiempo perdido en las actividades de E/S. Se divide en:

- Tiempo CPU del usuario: el tiempo consumido por el programa.
- Tiempo CPU del sistema: el tiempo consumido por el sistema operativo.

Ciclos de clock: determinan el momento en que tiene lugar los sucesos en el hardware.

Periodo de clock: es la duración de un ciclo de reloj, y se mide en segundos (s).

Frecuencia de clock: es el inverso al periodo de reloj, y se mide en hercios (Hz).

_ Las siguientes ecuaciones constituyen la performance del CPU:

$$\text{Tiempo de ejecución de CPU para un programa} = \text{Ciclos de reloj de la CPU para el programa} \times \text{Tiempo del ciclo del reloj}$$

$$\text{Tiempo de ejecución de CPU para un programa} = \frac{\text{Ciclos de reloj de la CPU para el programa}}{\text{Frecuencia de reloj}}$$

_ Esta fórmula nos indica que se puede mejorar la performance reduciendo la longitud o el número de ciclos de clock requeridos por un programa. Añadiendo también una mejora por el aumento de la frecuencia de clock.

Ciclos de clock por instrucción (CPI): es el número medio de ciclos de reloj que una instrucción necesita para ejecutarse.

$$\text{Ciclos de reloj de CPU} = \text{Instrucciones de un programa} \times \text{Medida de ciclos por instrucción}$$

_ El CPI proporciona una manera de comparar dos realizaciones diferentes de la misma arquitectura del repertorio de instrucciones, ya que el número de instrucciones requeridas por un programa será el mismo.

$$\text{Tiempo de ejecución} = \text{Número de instrucciones} \times \text{CPI} \times \text{Tiempo de ciclo}$$

$$\text{Tiempo de ejecución} = \frac{\text{Numero de instrucciones} \times \text{CPI}}{\text{Frecuencia de reloj}}$$

_ Estas dos fórmulas anteriores son útiles porque distinguen los tres factores claves que influyen en la performance.

Resumen del desempeño

_ A continuación, detallamos de que depende el rendimiento en los siguientes componentes de hardware o software:

- Algoritmo: afecta y determina el número de instrucciones del programa fuente ejecutadas. El algoritmo puede también afectar al CPI, favoreciendo instrucciones más lentas o más rápidas. Por ejemplo, si el algoritmo utiliza más operaciones en punto flotante, tenderá a tener un mayor CPI.
- Lenguaje de programación: afecta al número de instrucciones, ya que las sentencias del lenguaje son traducidas a instrucciones del procesador, lo cual determina el número de instrucciones. Las características del lenguaje también pueden afectar al CPI. Por ejemplo, un lenguaje con soporte para datos abstractos (como java), requerirá llamadas indirectas, las cuales utilizaran instrucciones con un CPI mayor.
- Compilador: la eficiencia del compilador afecta tanto al número de instrucciones como al promedio de los ciclos por instrucción, ya que el compilador determina la traducción de las instrucciones del lenguaje fuente a instrucciones del computador. El papel del compilador puede ser muy complejo y afecta al CPI de formas complejas.
- Arquitectura del repertorio de instrucciones: afecta a los tres aspectos de las prestaciones de la CPU, ya que afecta a las instrucciones necesarias para una función, al coste en ciclos de cada instrucción, y a la frecuencia del reloj del procesador.

Muro de la potencia

_ La tecnología dominante en la fabricación de circuitos integrados es la tecnología “CMOS”. En CMOS, la fuente principal de disipación de potencia es la potencia dinámica, es decir, la potencia consumida en las transacciones.

$$\text{Potencia} = \text{Carga capacitiva} \times \text{Voltaje}^2 \times \text{Frecuencia de conmutación}$$

Tipos de arquitectura

_ Tenemos dos tipos de arquitecturas:

Complex Instruction Set-Computer (CISC): es utilizado en Intel.

Reduced Instruction Set-Computer (RISC): es utilizado en ARM y en el resto del mundo tecnológico.

	CISC	RISC
Sistema de instrucción	El sistema de instrucciones del ordenador es rico, y hay instrucciones especiales para completar funciones específicas.	Las instrucciones se utilizan eficientemente. Es necesario implementar diferentes funciones combinando instrucciones.
Operación de memoria	Hay muchas instrucciones de funcionamiento en la memoria de la máquina, por lo que la operación es directa.	El control se simplifica debido a la restricción en la operación de la memoria.
Procedimiento	La programación del lenguaje ensamblador es relativamente simple, el cálculo científico y el funcionamiento complejo del diseño de la comunidad de programación es relativamente fácil, de alta eficiencia.	Los programas del lenguaje ensamblador generalmente necesitan un gran espacio de memoria, y es difícil diseñar los programas para realizar funciones especiales.
Interrupción	Una máquina responde a una interrupción despues de ejecutar una instrucción.	La máquina puede responder a las interrupciones donde se ejecuta la instrucción.
CPU	La CPU contiene una variedad de unidades de circuito, por lo que tiene potentes funciones, áreas grandes y alto consumo de energía.	La CPU contiene menos circuitos de unidades, lo que resulta en un área pequeña y bajo consumo de energía.
Período de diseño	El microprocesador tiene una estructura compleja y un largo periodo de diseño.	El microprocesador es simple en estructura, compacto en diseño, corto en el ciclo de diseño, y fácil de adoptar la última tecnología.
Usuario	El microprocesador es complejo en estructura y potente en función, por lo que es fácil realizar las funciones especiales.	El microprocesador tiene una estructura simple, instrucciones regulares, rendimiento fácil de entender, fácil de aprender y usar.

Falacias y errores habituales

_ Falacias:

- Las instrucciones más potentes implican prestaciones mayores. Esto es falso porque requieren menos instrucciones. Pero las instrucciones complejas son difíciles de implementar.
- Es necesario escribir en lenguaje Assembly para obtener prestaciones más altas. Esto es falso, pero los compiladores modernos son mejores para manejar

procesadores modernos. Cuantas más líneas de código, más errores y menos productividad.
- La importancia de la compatibilidad binaria comercial implica que los repertorios de instrucciones exitosos no cambian. Pero sí acumulan más instrucciones.

_ Errores habituales:

- Olvidar que las direcciones secuenciales de las palabras en máquinas con direccionamiento byte no se diferencian en uno.
- Usar un puntero a una variable automática fuera de la definición de su procedimiento.

Direccionamiento y Lógica de Decodificación de Memorias

Memoria

_ Ahora trabajamos con memorias en general, es decir, ya no se cumple la regla de que la palabra de memoria es de 8 bits como en LEGv8, por ende ahora podemos trabajar con distintos tamaños de palabras.

_ La memoria está organizada como si fuera un arreglo, donde cada palabra puede tener una X cantidad de bits. Despues vamos a tener una cantidad de palabras que va a estar dada por cuantas direcciones tenemos, es decir, la cantidad de palabras que podemos direccionar son la cantidad de palabras que tenemos en la memoria. Podemos dejar direcciones de memoria sin llenar con palabras, pero esos espacios de dirección seguirán estando disponibles.

Mapa de memoria

_ En el mapa de la memoria lo que se hace es indicar que tamaño tienen los chip de memoria que están conectados dentro de la memoria de ese sistema. También sirve para aclarar direcciones y el tamaño de la palabra. Como podemos observar en el grafico tenemos las direcciones, en donde si tenemos 2N palabras de memoria, vamos a tener N bits para la dirección, ya que con N bits podemos obtener 2N combinaciones de direcciones posibles. Podemos observar las direcciones expresadas en binario y en hexadecimal. En el caso de este gráfico, las direcciones más bajas están en la parte más baja del gráfico, y las direcciones más altas están en la parte más alta del gráfico, aun asi esto es indistinto. Lo que tenemos guardado como dato dentro de la memoria es lo que vamos a conocer como contenido de la memoria. Y finalmente lo que tenemos es una capacidad total de la memoria, que indican cuantos bits permite o tiene guardados, por ende la cantidad total es cantidad de palabras X ancho de palabra, es decir, 2N * X, en donde X es la cantidad de bits que tiene cada palabra. Por lo general la capacidad total se expresa en bits, y si es en bytes simplemente se lo divide en 8.

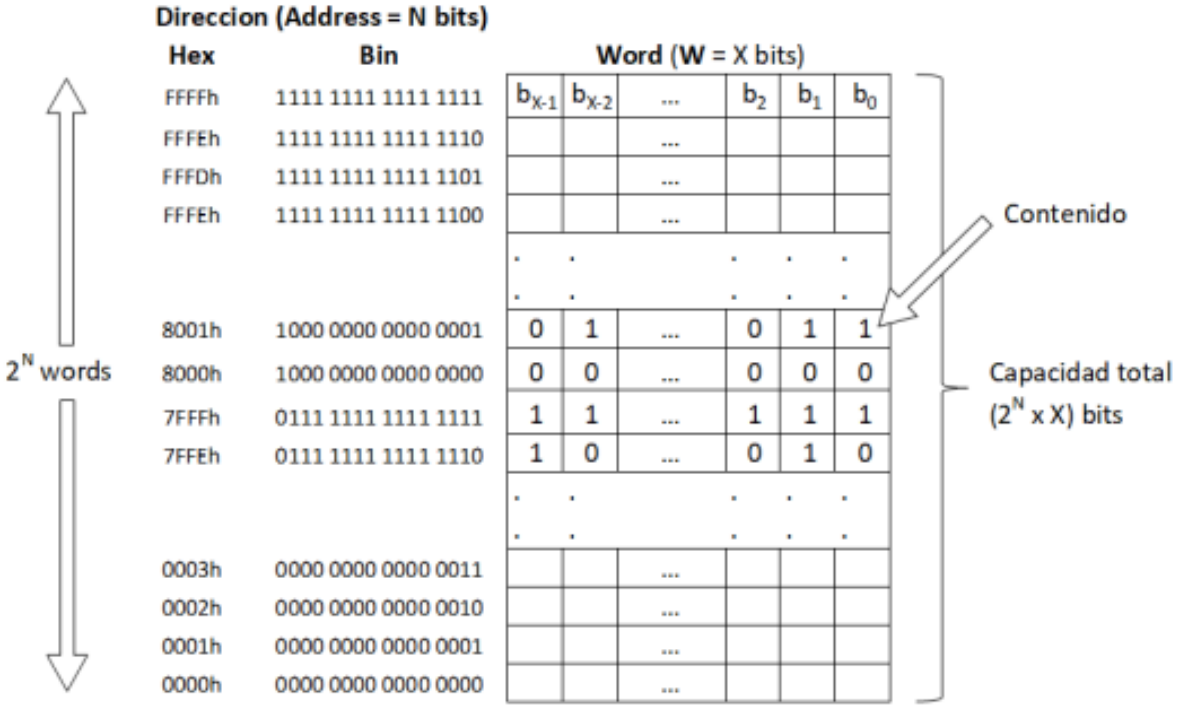


Tabla hexadecimal - binario:

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Tabla de capacidad de direccionamiento y unidades de almacenamiento de información

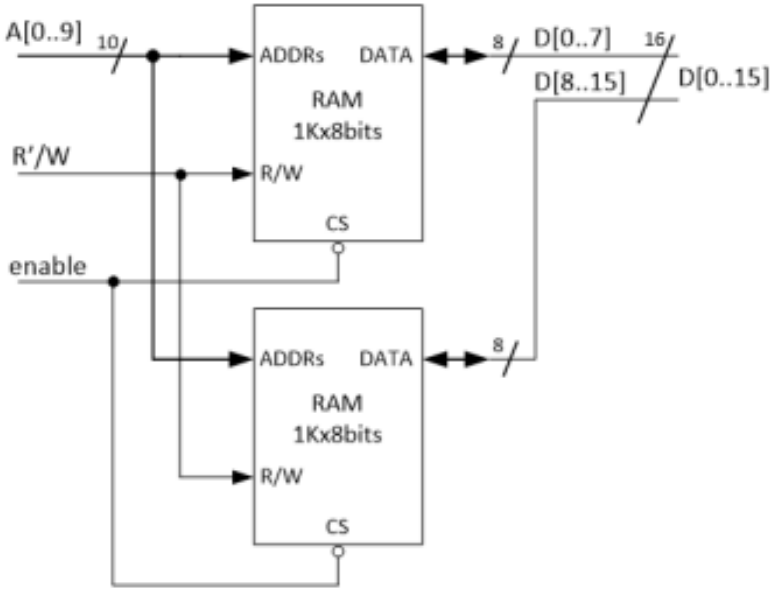
_ La siguiente tabla nos ayudara a calcular las capacidades de direccionamiento y unidades de almacenamiento. Todas las unidades son múltiplos de 2N ya que estamos trabajando con números en base 2.

N bits (2^N)		Capacidad (en words)
2^1	=	**2** words
2^2	=	**4** words
2^3	=	**8** words
2^4	=	**16** words
2^5	=	**32** words
2^6	=	**64** words
2^7	=	**128** words
2^8	=	**256** words
2^9	=	**512** words

N bits (2^N)		Capacidad (en words)		Simbolo [Prefijo]
2^{10}	=	1024 words	=	**1Kw [Kilo]**
2^{20}	=	1024 Kw	=	**1Mw [Mega]**
2^{30}	=	1024 Mw	=	**1Gw [Giga]**
2^{40}	=	1024 Gw	=	**1Tw [Tera]**
2^{50}	=	1024 Tw	=	**1Pw [Peta]**

Conexiones

Conexión en paralelo: esta conexión se utiliza para aumentar el ancho de una palabra. En el caso que vemos a continuación, tenemos 2 chips de 1K de RAM por 8 bits, pero necesitamos 1 K de RAM pero con un ancho de palabra de 16 bits, y esto se obtiene con la conexión que vemos en el diagrama, en donde en un chip se guarda la parte baja de la palabra y en el otro chip, la parte alta de la palabra. En el primer chip tenemos los datos de los bits del 0 al 7, y en el segundo chip vamos a tener los bits del 9 al 15. Entonces leyendo la salida de esos dos chips, podemos armar una palabra de 16 bits



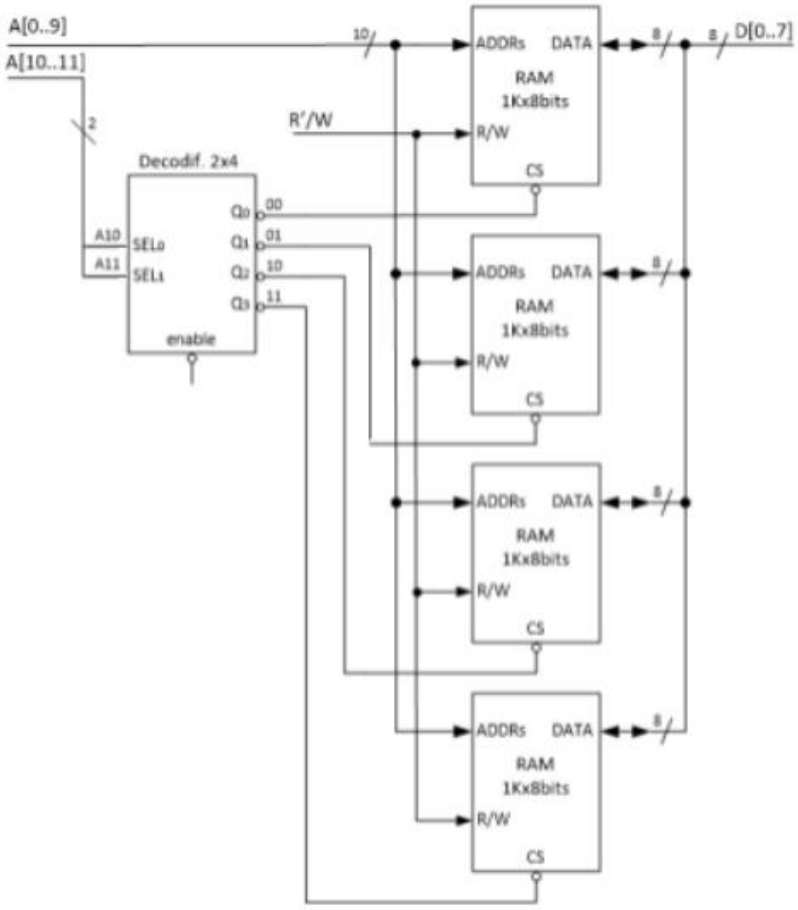
_ Lo importante a tener en cuenta a la hora de hacer una conexión en paralelo, es que como primera medida nos tenemos que asegurar de sacar las dos partes de la palabra de los dos chips, porque si direccionamos mal uno respecto del otro, podemos traer la mitad del dato y la mitad de otro dato que no tiene nada que ver. Entonces, es fundamental que la dirección sea la misma para ambos casos, y lo podemos ver en la entrada de ADDRs que entra a ambos chips. Entonces, nos posicionamos en el mismo elemento en los dos chips de memoria.

_ Y por otra parte, es importante que ambos chips estén habilitados, ya que si no habilitamos un chip, a la salida no podremos ver nada de ese chip y veremos una parte de la palabra del chip que este habilitado. Entonces, es fundamental que los chips select (CS) de los dos chips de memoria, estén conectados entre sí para que se habiliten juntos.

_ Finalmente tenemos los datos de Read/Write, ya que obviamente en estos chips no podemos leer ni escribir a la vez, sino que podemos hacer una o la otra, es por eso que usan una misma señal de control, en donde con R' en 0 habilitaríamos la lectura y con 1 habilitaríamos W, es decir, la escritura en memoria. Direccionamos siempre de la misma forma R/W, pero en un caso leemos y en el otro escribimos. Las doble flechas

indican que los datos pueden entrar o salir de la memoria dependiendo de si estoy haciendo una acceso de lectura o escritura.

Conexión en serie: esta conexión se utiliza para aumentar la capacidad de direccionamiento, es decir, cuantas palabras tenemos en nuestro sistema de memoria.



_ Como vemos en el gráfico, ahora ya no tenemos una palabra guardada en varios chips, sino que en cada chip vamos a tener una palabra del ancho completo de lo que es la salida. Pero, tenemos que encontrar en cual de esos chips tenemos la palabra que queremos direccionar, para luego poder buscar esos datos. Para resolver esto, por lo general, lo que se usa en este tipo de conexiones, son los decodificadores. El decodificador del grafico es activo por bajo, es decir, su salidas siempre van a estar en 1, salvo la salida activa cuyo valor será 0, y cuando una salida se active y se ponga en 0, se va a habilitar el correspondiente chip select que esté conectado a esa salida, y asi podremos habilitar o no un chip de memoria.

_ En este ejemplo, vamos a necesitar distinta cantidad de bits para direccionar cada uno de los chips de memoria, que para direccionar la totalidad. Como vemos en el grafico cada chip es de 1K palabras, y para direccionar 1K usamos 10 bits, entonces vamos a necesitar address del 0 al 9, pero si necesitamos formar 4K o más, los bits de diferencia que no usamos en el address para direccionar cada uno de los chips, son los que vamos a usar para elegir en que chip esta guardado o vamos a guardar la

información. En este caso, esos bits de diferencia son los más significativos, y entran como entrada de selección al decodificador, en donde si ambos están en 0, lo que se selecciona es la salida Q0 y por lo tanto se va a seleccionar el primer chip, y asi sucesivamente. Podemos observar, que la dirección más chica que se puede generar son 12 ceros y para esto se activaría el primer chip, y la dirección más grande con 12 unos, activaría el cuarto chip.

Sistema jerárquico de memoria - Caché

Introducción

Principio de localidad

_ El principio de localidad establece que los programas acceden a una parte relativamente pequeña del espacio de direcciones en un determinado instante. Existen dos tipos de localidad:

Localidad temporal: si se accede a la dirección de memoria de un dato, pronto se accederá de nuevo a esa dirección. Por ejemplo, las instrucciones en un bucle, variables de inducción, etc.

Localidad espacial: si se accede a la dirección de memoria de un dato, pronto se accederá a las direcciones de memoria que se encuentran próximas a ella. Por ejemplo, acceso a instrucciones secuenciales, matriz de datos, etc.

Jerarquía de la memoria

_ La jerarquía de la memoria aprovecha el principio de localidad, implementando la memoria de un computador. Esta jerarquía está formada por varios niveles de memoria con diferentes tiempos de accesos y capacidades. En ella se usan tres tecnologías:

- Memoria dinámica de acceso aleatorio (DRAM): la memoria principal es implementada con esta. Tiempo de acceso típico es 0,5 - 2,5 ns.
- Memoria estática de acceso aleatorio (SRAM): a esta la utilizan los niveles más cercanos al procesador. Tiempo de acceso típico es 50 - 70 ns.
- Discos magnéticos: para implementar el mayor y más lento nivel de la jerarquía se usan estos discos. Tiempo de acceso típico es 5000000 - 20000000 ns.

_ La memoria más rápida está cerca del procesador y las más lentas están debajo de ella. Lo mismo ocurre con los precios.

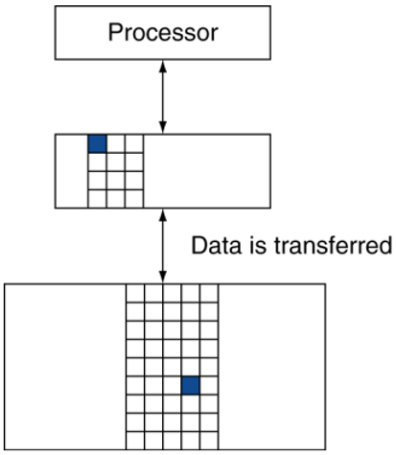
Bloque o línea: es la unidad mínima de información.

Frecuencia de aciertos: es la fracción de accesos a memoria cuyos datos se encuentran en el nivel superior. Y se suele usar como medida de performance de la jerarquía de memoria.

Frecuencia de fallos: es la fracción de los accesos a memoria cuyos datos no se encuentran en el nivel superior.

Tiempo de acierto: es el tiempo necesario para acceder al nivel superior de la jerarquía de memoria, que incluye el tiempo requerido para determinar si el acceso es un fallo o acierto.

Penalización del fallo: es el tiempo para reemplazar un bloque en el nivel superior por el correspondiente bloque del nivel inferior.

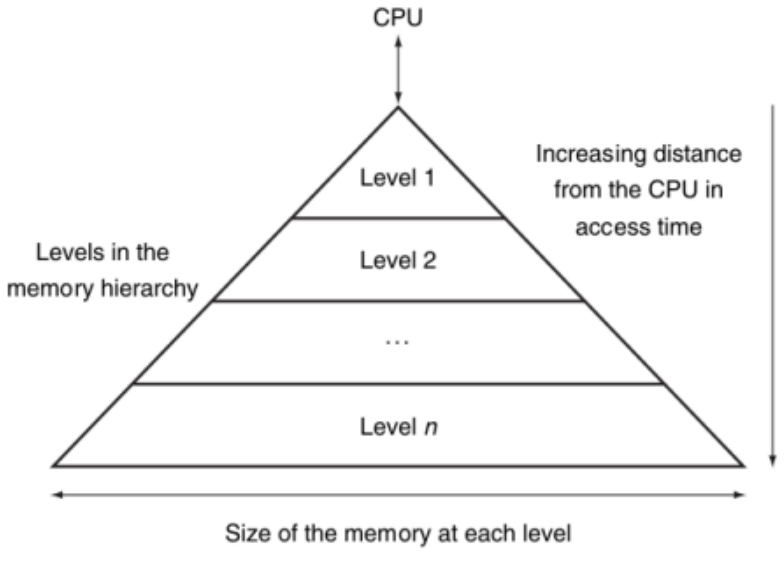


Estructura

_ Como mencionamos anteriormente, tenemos una estructura jerárquica, con diferentes niveles. Siempre la que intenta acceder a la memoria es la CPU.

_ La estructura es de forma piramidal porque nos muestra el tamaño de cada uno de los niveles, entonces en el nivel más bajo y que está más lejos de la CPU, tenemos una memoria mucho más grande, ya que, a medida que nos acercamos a la CPU la memoria es cada vez más pequeña. Esto se debe a que, cada vez es más costosa.

_ También, mientras más lejos estemos de la CPU más tiempo demoramos en acceder a la misma, y esto tiene que ver también con la tecnología involucrada en cada una de las memorias. La memoria del nivel 1 es super rápida y cada bit nos va a costar muchos más de lo que nos va a costar en una memoria de nivel n. Y podemos tener n niveles, ya que dependiendo del sistema computacional podemos necesitar más o menos memoria.



_ Siempre los acceso a memoria son lo más lento que hay en el procesador, y es por eso que hay diferentes técnicas y uso de jerarquía de memoria, como para tratar de acelerar el proceso porque esta parte demora mucho tiempo.

Caché

Direccion de memoria principal

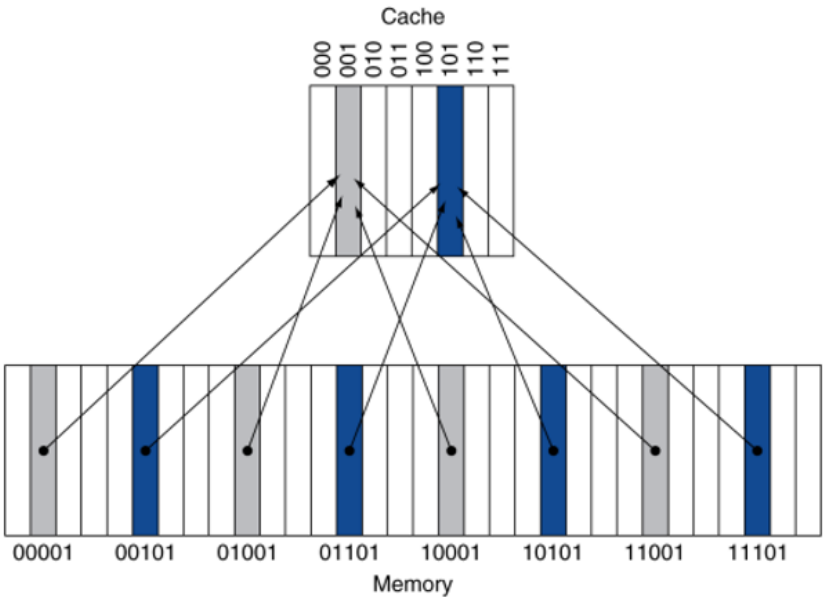
_ En nuestro caso esta dirección de memoria principal tiene 64 bits. Cuando queremos acceder a una cache de mapeo directo, para saber dónde se va a guardar ese dato en la cache, vamos a tener que analizar la dirección de la memoria principal. Entonces, la dirección de 64 bits se divide en 4 grupos:

- Offset de byte: en los bits menos significativos vamos a tener los offset de byte, y lo llamamos asi porque asumimos que siempre estamos en nuestro micro donde el ancho de la memoria es de 8 bits, entonces los bits de offset de byte son los que van a indicar cuantas posiciones de memoria vamos a traer a la cache, cuando traemos una palabra de la memoria principal.
- Offset word: con este identificamos entre las distintas palabras que están guardadas en una línea de cache.
- Índex: nos indica en que línea de cache estamos, es decir, cual es la línea a la que vamos a acceder.
- Tag: está formado con los bits sobrantes, es decir, restamos los 64 bits a la cantidad de bits del índex, el offset word y el offset byte.

_ En cache no guardamos direcciones sino que guardamos contenido de la memoria principal. Únicamente guardamos una parte de la dirección que son los bits más significativos que corresponden al tag, pero despues, lo que va como dato (dato), es lo que sacamos de la memoria.

Caché de correspondencia directa

_ La memoria cache es el nivel de la jerarquía de memoria más cercano a la CPU. En inglés, Direct Mapped Cache, es una organización de caché en la que cada posición de la memoria principal se corresponde con una única posición de caché. Ya que cada posición de la caché puede alojar el contenido de varias posiciones diferentes de memoria principal, es necesario añadir un conjunto de etiquetas a la caché.



Etiquetas de caché

_ Las etiquetas o tag, contienen la información de la dirección que se necesita para identificar si una palabra de la caché se corresponde con la palabra solicitada. De esta manera podremos saber en qué bloque en particular o ubicación de la caché, está almacenado tal dato proveniente de la memoria principal.

Bit de validación

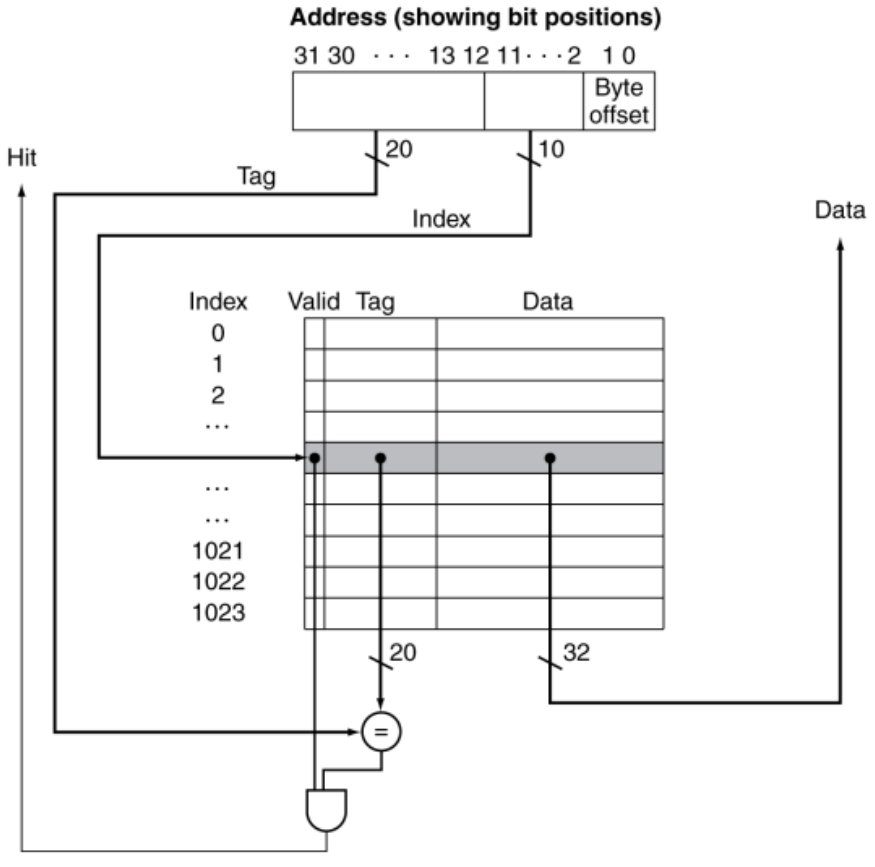
_ Entonces, para saber si hay datos o no en una ubicación de la caché, tenemos el bit de validez, que indica si una entrada contiene una dirección con información válida o no. Inicialmente el bit está en 0:

- Si el bit de validación vale 1, significa que hay un dato presente.
- Si el bit de validación vale 0, significa que hay un dato presente.

Acceso a la caché

_ En una caché de correspondencia directa solo existe un lugar donde alojar los elementos recientemente solicitados, y de ahí que solo exista una única opción para decidir qué reemplazar. Una dirección solicitada se divide en:

- Valid: bit que se pone en 1 a medida que cargamos un dato.
- Tag: etiqueta que se usa para compararla con el valor de la etiqueta almacenada en la caché. Cargamos la etiqueta de la palabra a escribir en cache.
- Un índice de la caché que se utiliza para seleccionar el bloque de datos.
- Data: en esta parte iría el contenido de la memoria principal al que accedimos cuando colocamos cada una de las address. Como no sabemos que hay dentro da la memoria principal, lo expresa como M[num_direccion], y asi representamos que es el contenido de la memoria direccionado por esa dirección.



_ El índice de un bloque de caché, junto con el contenido de la etiqueta para este bloque, determinan con precisión la dirección de memoria de la palabra almacenada en el bloque de caché.

_ Los bloques más grandes aprovechan la localidad espacial para reducir la frecuencia de fallos. Al aumentar el tamaño del bloque, la frecuencia de fallos se reduce. Pero si el bloque se hace demasiado grande, la frecuencia de fallos puede aumentar.

_ La penalización por fallo viene determinada por el tiempo necesario para llevar el bloque desde el siguiente nivel inferior de la jerarquía hasta la caché. Este tiempo tiene dos componentes que son:

- La latencia de la primera palabra.
- Tiempo de transferencia del resto del bloque.

Manejo de los fallos de la caché

Acierto de cache (hit): se produce, por ejemplo, cuando estamos buscando la palabra con el número 25 desde la memoria principal, entramos en la caché y dicha palabra direccionada con su número 25 existe y es válida, produciéndose de esta manera el hit. Nosotros tratamos de organizar la cache de forma tal que tengamos la mayor cantidad de aciertos posibles.

Fallo de caché (miss): es una solicitud de acceso a datos de la caché, que no puede ser atendida debido a que los datos no están presentes en dicha caché. La unidad de control debe detectar y procesar un fallo trayendo los datos solicitados desde la memoria principal. Entonces, se produce por ejemplo cuando entramos a la caché buscando la palabra con el número 25, dicha palabra no existe por alguna razón, y por ende, la cache da una fallo. Esto implica que la cache no puede devolver el dato, por lo que tiene que ir a una memoria de mayor jerarquía (un nivel más bajo pero con más capacidad de memoria) en donde seguro esta ese dato, y lo trae a la cache para que se lo pueda usar.

_ En el caso de un hit o acierto en la caché, la CPU continúa normalmente. Pero, en el caso de un fallo de caché, tenemos:

- Se detiene el pipeline de la CPU.
- Obtenemos el bloque del siguiente nivel de jerarquía.
- Error de caché de instrucciones.
- Falta de caché de datos.

_ El manejo de los fallos de caché se realiza con una unidad de control del procesador y con un controlador independiente que pone en marcha el acceso de memoria y actualiza el contenido de la caché.

Manejo de las escrituras

Escritura directa (Write-Through): es la técnica en la cual las escrituras siempre actualizan tanto la caché como la memoria principal, asegurando que los datos siempre son coherentes en ambas memorias.

Buffer de escritura: es una cola que almacena temporalmente datos mientras que los datos esperan o que sean escritos en memoria principal. Cuando finaliza dicha escritura, la entrada en el buffer de escritura se libera. Si el buffer se llena cuando el procesador ejecuta una escritura, el procesador debe pararse hasta que exista una entrada libre en el buffer de escritura.

Escritura retardada (Write-Back): es la técnica que maneja las escrituras, primero actualizando solo los valores del bloque de la caché, y luego guardando el bloque en el nivel inferior de la jerarquía, cuando este mismo sea reemplazado.

Evolución y mejora de las prestaciones de la caché

_ Hay dos técnicas para mejorar las prestaciones de la caché:

- La primera se centra en reducir la frecuencia de fallos, disminuyendo la probabilidad de que los dos bloques de memoria distintas compitan por la misma posición de caché.
- La segunda reduce la penalización por fallo añadiendo un nivel más a la jerarquía. Esta técnica se denomina caché multinivel.

_ Existe una amplia variedad de técnicas para realizar el emplazamiento (mecanismo a través del cual se le notifica a las partes demandadas de que hay un proceso en su contra) de bloques, a continuación tenemos:

Correspondencia directa: en donde un bloque puede ser emplazado en una única posición.

One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Completamente asociativa: donde un bloque puede ser emplazado con cualquier posición, porque dicho bloque puede estar asociado a cualquier entrada de la caché. Este seria un caso de asociativa de 8 vías. En este caso, no tenemos ningún índex, por lo que todos los datos que se guardan en la cache, en una sola línea.

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

Asociativa por conjunto: esta es una forma intermedia entre las dos anteriores, en donde existe un numero establecido de posiciones donde cada bloque puede ser emplazado. Tenemos asociativa por conjunto de 2 vías, en donde tenemos 8 líneas pero distribuidas en dos vías, por eso el índex es del 0 al 3. Y tenemos asociativa por conjunto de 4 vías, en donde tenemos 8 líneas pero distribuidas en 4 vías. En este tipo, por cada línea de cache hay otro tag, otro data y otro bit de verificación.

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Elección del bloque a reemplazar

_ Cuando se produce un fallo o miss en una caché de correspondencia directa, el bloque solicitado solo puede guardarse en una única posición, y por ello, el bloque que ocupe esa posición debe ser reemplazado.

- En una caché asociativa tenemos la posibilidad de elegir donde guardar el bloque solicitado, ya que podemos elegir que bloque sustituir.
- En una caché asociativa por conjunto, se puede elegir entre los bloques del conjunto seleccionado.

Política del menos recientemente usado (LRU): esta es una política de reemplazo que normalmente se utiliza, en donde el bloque reemplazado es el que no ha sido utilizado por el periodo más largo.

_ El control de la caché debe especificar las señales que se van a activar en un paso dado y el siguiente paso de secuencia.

- Máquinas de estados finitos: el método de control multi-paso más extendido se basa en estas máquinas, que consisten en un conjunto de estados e indicaciones de cómo se producen los cambios de estado.
- Función de estado siguiente: las indicaciones se definen con esta función, que obtiene el nuevo estado a partir del estado actual y las entradas.

Problema de coherencia de caché

_ Podemos decir que un sistema de memoria es coherente si cualquier lectura de una dato devuelve el valor más recientemente modificado de ese mismo dato. Esta definición contiene dos aspectos diferentes del comportamiento del sistema de memoria, críticos para una escritura correcta:

- Coherencia: que define que valores deben obtenerse con una lectura.
- Consistencia: que determina cuando un valor que se escribe será devuelto en la lectura.

_ Los multiprocesadores con coherencia de caché proporcionan:

- Migración: un dato puede moverse en una caché local y utilizarse ahí de una forma transparente. La migración reduce la latencia de acceso a los datos compartidos que están almacenados en forma remota.
- Replicación: cuando los datos compartidos se leen simultáneamente por parte de varios procesadores, se hace una copia del dato en las caches locales. La replicación reduce la latencia de acceso y la dispara en la lectura de un dato compartido.

Conceptos

Palabra direccionable directamente en memoria: esto significa que no necesitamos saber de cuantos bits tiene es esa palabra, ya que simplemente sabemos que una palabra en la memoria es equivalente a una palabra en caché, y que ocupa una sola dirección. Debido a esto, no vamos a tener bits de offset de byte o de alineamiento en memoria, porque por cada posición de memoria, tenemos una palabra.

AMAT (Average Memory Access Time): una de las cosas importantes sobre una memoria caché, es el tiempo promedio de acceso que uno tiene en la memoria cuando uno pone la caché a funcionar en el nivel 1 del micro. Entonces, hay un tiempo de acceso a la memoria principal, pero tenemos un tiempo de acceso a memoria en general. En este tiempo de acceso promedio, influye el porcentaje de fallos respecto de los hits que tenemos dada la distribución de caché. Pero esto dependerá del problema.

Stall de memoria: son los accesos a memoria sobre la totalidad del programa, es decir, porcentaje de instrucciones que acceden a la memoria dentro del programa.

Sistema de E/S y excepciones

Sistemas de entrada y salida

_ El sistema de E/S es el sistema que nos permite extraer datos de la realidad. El procesador tiene un gran BUS (de dato y address), que se denomina BUS del sistema. Mediante el address se indica quien es el responsable de poner el dato en el BUS, y cuando ya está en el BUS, el procesador adquiere dicha información.

_ Hay dos tipos de mapeo o direccionamiento, que dependen del tipo de arquitectura:

Standar I/O: es decir, estándar E/S, que es el más utilizado por Intel o arquitecturas convencionales, en donde hay un mapa para las entradas y salidas, y otro aparte para la memoria. Para saber a cuál se accede hay que tener en cuenta el valor de la memoria I/O. En este tipo de mapeo tenemos un sistema para cada uno por lo cual no se pierde espacio direccionable.

Memory-Mapped I/O: es el más usado en la actualidad, y en este caso, el mapa de entrada y salida esta mapeado en la memoria, y se usan las mismas instrucciones para acceder a dichos registros que los usados en memoria. No necesita instrucciones especiales ya que se usa el mismo assembler para acceder a memoria como a los registros E/S.

Métodos de operación

_ Los métodos de operación se refieren a aquellos mediante el cual con el software se puede controlar el hardware. Las interfaces son los registros, por lo que mediante el software hacemos un acceso a E/S, y esa lectura o escritura tiene una acción directa respecto al hardware. Tenemos tres métodos de operación:

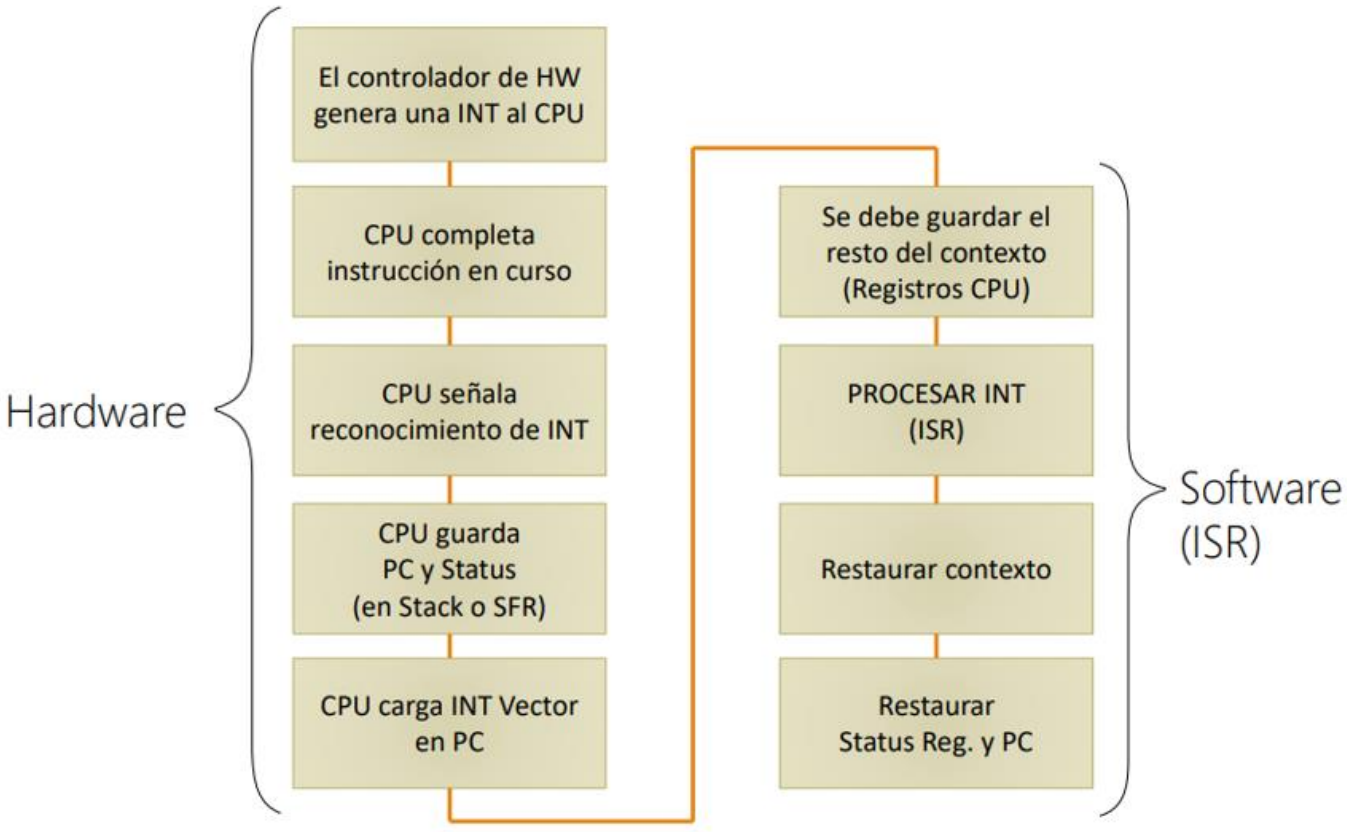
Direct Memory Access (DMA): permite que ciertos dispositivos periféricos accedan directamente a la memoria principal del sistema sin la intervención del procesador. En lugar de que el procesador realice las transferencias de datos entre los dispositivos periféricos y la memoria, el DMA permite que un controlador de DMA se encargue de estas transferencias de manera independiente, liberando al procesador de esta tarea y permitiéndole realizar otras operaciones.

Polling-driven: o E/S programada, se utiliza para controlar y manejar dispositivos de E/S (entrada/salida) mediante la verificación continua de su estado. En lugar de utilizar interrupciones para notificar eventos, el método polling-driven implica que el programa principal consulte periódicamente el estado del dispositivo para determinar si se ha producido alguna acción o cambio relevante. El manejo se realiza mediante el uso de instrucciones E/S por código de programa. Una de las principales desventajas es que se desperdician muchos ciclos de instrucción en revisar el estado del módulo de E/S (Polling).

Interrupt-driven: o interrupción, es un recurso físico que sale de los módulos periféricos, y que luego entra a la CPU. A esa señal se la denomina Interrupt-request (Int). Entonces, el procesador interrumpe la ejecución normal y secuencial de un código cuando la señal esta activa. Al finalizar cada ciclo de instrucción al CPU verifica automáticamente si hay Int pendientes. Se hace un Polling por hardware, en donde si hay un Int, el CPU salta a una posición de memoria especifica llamada vector de interrupciones.

- Vector de interrupciones: contiene el código (o su referencia) con los procedimientos necesarios para dar servicio a dicha interrupción. Este código se denomina Interrupt Service Routine.
- Interrupt Service Routine (ISR): es un bloque de código especializado que se ejecuta en respuesta a una interrupción generada por un dispositivo o evento externo en un sistema informático. Si hay un sistema operativo, este mismo es el dueño de la ISR, pero si no lo hay, el dueño es el usuario.
  - Dirección fija (fixed interrupt): esta dirección está establecida en la lógica de la CPU, y no puede ser modificada (esto es correcto si la cantidad de periféricos es pequeña). La CPU puede contener la dirección real, o contener una instrucción de salto a la dirección real de la ISR si no hay suficiente espacio reservado.
  - Direccion vectorizada (vectored interrupt): cuando los módulos de E/S son más grandes se utiliza esta dirección. El periférico provee la dirección al CPU por medio del bus de datos. Aquí se necesita una señal más para implementar una interrupción, que viaja desde el procesador hacia los periféricos, y se denomina Interrupt Acknowledge (Int Ack).
    Muy utilizado en sistemas con múltiples periféricos conectados por un bus.

_ Las interrupciones se procesan de la siguiente manera:



Tipos de interrupciones

_ Podemos encontrar dos tipos de interrupciones:

Enmascarables: son aquellas que se pueden desactivar o bloquear temporalmente mediante el uso de una máscara o un mecanismo de habilitación/deshabilitación. Esto significa que el sistema puede decidir ignorar o posponer temporalmente el procesamiento de estas interrupciones en determinadas situaciones. Entonces, estas pueden ser deshabilitadas, a pesar de que el evento, que genera la interrupción, se dé. Esto se puede solucionar mediante un bit de control que causa que el procesador ignore una solicitud de interrupción.

No enmascarables: son aquellas que no se pueden desactivar o bloquear temporalmente. Estas interrupciones son consideradas críticas y requieren una respuesta inmediata por parte del sistema. No se les permite ser ignoradas o pospuestas, ya que su ocurrencia indica eventos importantes o condiciones excepcionales que deben ser atendidos de inmediato. Están asociadas a cuestiones de funcionamiento del procesador o son recursos auxiliares que el procesador necesita para funcionar.

Métodos de arbitraje

_ Cuando hay solicitudes de varias interrupciones, es necesario utilizar métodos de arbitraje, que se usan dependiendo la situacion. Algunos de ellos son:

Software polling: técnica utilizada en programación para verificar continuamente el estado de un dispositivo o una condición en lugar de esperar una interrupción o una notificación activa. En lugar de utilizar interrupciones o eventos para detectar cambios en el estado de un dispositivo, el software polling utiliza un bucle de código que verifica de manera repetitiva si se ha producido algún cambio. El bucle de encuesta típicamente se ejecuta de forma continua, revisando el estado deseado o esperando una condición específica. Se utiliza cuando estamos usando una solución de vector único en el procesador. La prioridad es establecida en la ISR, según el orden de búsqueda.

Arbitro de prioridades: su función principal es determinar el orden en el que se deben ejecutar las instrucciones en función de su prioridad relativa. Las instrucciones pueden ser reordenadas y ejecutadas en un orden diferente al que aparecen en el programa original. Esto se hace para maximizar la utilización de los recursos del procesador y mejorar el rendimiento. Se lo conoce como controlador de interrupciones. Los periféricos hacen la petición de Int Req al controlador, y este último al CPU. La prioridad puede ser fija o configurable o en forma de cola (FIFO).

Conexión en cadena: en este caso, los periféricos están dispuestos en el sistema en forma de cadena. El primer dispositivo está conectado directamente al segundo, el segundo al tercero, y así sucesivamente, hasta llegar al último dispositivo de la cadena. Cada dispositivo tiene un puerto de entrada y un puerto de salida, a través de los cuales se transmiten los datos. Cuando se envía un mensaje o una señal a través de la

cadena, este se transmite secuencialmente de un dispositivo al siguiente, pasando por todos los dispositivos de la cadena hasta llegar al destino final. Los periféricos tienen cuatro señales por interrupción y se le agrega un Int Req de entrada ya que poseía uno de salida. Y por otro lado se agrega un Int Ack de salida ya que poseía uno de entrada.

Arbitraje de bus: proceso mediante el cual se determina qué dispositivo tiene acceso al bus de datos en un sistema de computadora o en una red. El bus de datos es un canal de comunicación compartido utilizado para la transferencia de datos entre diferentes componentes del sistema. Cuando varios dispositivos desean acceder al bus de datos al mismo tiempo, es necesario establecer un mecanismo de arbitraje para evitar conflictos y garantizar un acceso ordenado y justo al bus. El arbitraje de bus se encarga de resolver estos conflictos y determinar cuál de los dispositivos tiene el derecho de acceso al bus en un determinado momento. El periférico debe primero obtener la sesión del bus para luego requerir una interrupción.

Implementación de la ISA + Ensamblado y desensamblado de LEGv8

Introducción a LEGv8

ARMv4: Complete single-cycle processor

_ LEGv8 es un sub set de las instrucciones de ARMv8 y con algunas ligeras diferencias. Pero la gran diferencia con el micro procesador ARMv8, es que los registros de ARMv4 son de 32 bits, y los de ARMv8 son de 64 bits. Por otro lado, ya no vamos a tener 16 registros sino que vamos a tener 32 registros para trabajar. Es importante tener en cuenta que en este caso, el tamaño de instrucción no cambia. Antes teníamos registros de 32 bits e instrucciones de 32 bits, y si bien ahora los registros van a duplicar su tamaño y cantidad, las instrucciones van a seguir siendo de 32 bits.

El procesador

_ El compilador y la arquitectura de repertorio de instrucciones, determinan el número de instrucciones requeridas por un cierto programa. Nosotros analizamos el micro LegV8, que está formado por las siguientes instrucciones:

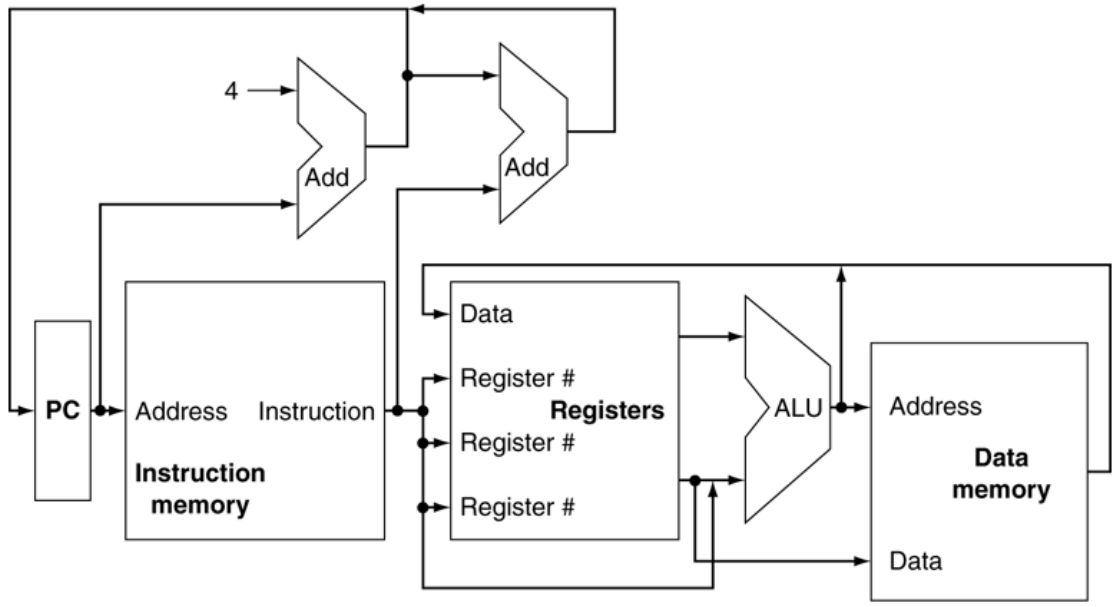
- Referencia a memoria: LDUR y STUR.
- Aritmético-Lógicas: ADD, SUB, OR, AND y SLT.
- Control: CBZ.

_ Para que se ejecute una instrucción, se realiza lo siguiente:

- Se envía el program counter (PC) a la memoria de instrucción que contiene el código, y se carga la instrucción desde esa memoria.
- Luego, se lee uno o dos registros. Para instrucciones de carga se lee un registro, y en el resto de los casos, dos registros.

- Luego, el accionar depende de cada caso.

_ A continuación, vemos la implementación básica anterior, en el micro:



_ En la práctica las líneas de datos no pueden conectarse directamente; debe añadirse un elemento que seleccione entre múltiples orígenes y dirija una de estas fuentes al destino. Esta selección se realiza mediante un multiplexor, que es el que se encarga de la selección de una de las entradas según la configuración de sus líneas de control.

Diseño lógico

_ Las unidades funcionales constan de dos tipos de elementos lógicos; Los que operan con datos y los que contienen estados.

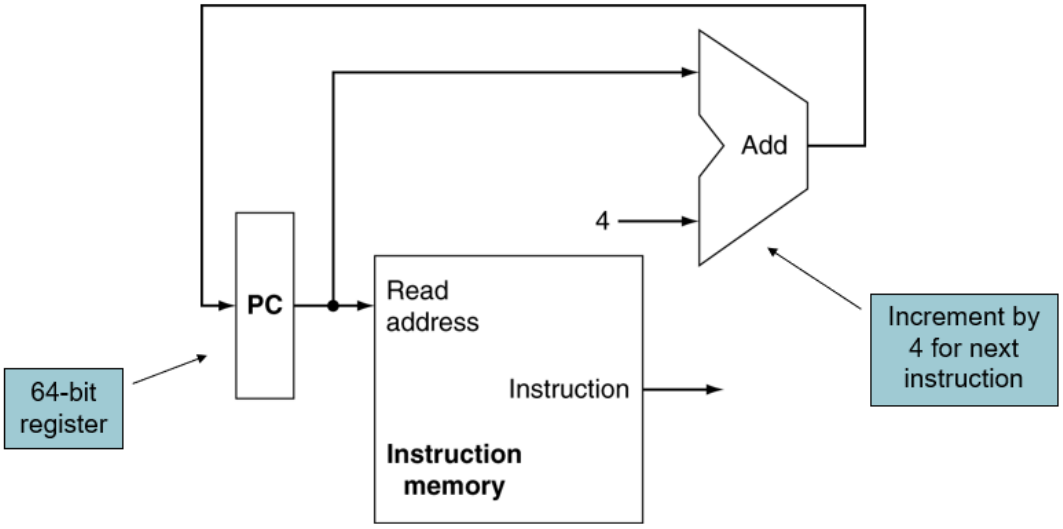
- Elementos combinacionales: son aquellos que operan con datos, lo que significa que sus salidas dependen únicamente de los valores actuales de las entradas. Estos son la compuerta AND, el ADDER, el multiplexor y la ALU.
- Elementos de estado o secuenciales: se si apaga la máquina, se puede reiniciar cargando dicho elemento con lo que contenía antes de ser apagado. Estos guardan información, y usan una señal de clock determinada para cuando deben actualizar el valor guardado.

Construcción de un DataPath

_ Algunos de los elementos necesarios son:

- Una unidad de memoria donde almacenar y suministrar las instrucciones a partir de una dirección.
- Un programa counter (PC), que se utiliza para almacenar la dirección de la instrucción actual.
- Y por último, un sumador para incrementar el PC, con el fin de que este apunte a la dirección de la instrucción siguiente.

Fetch Instruction: proceso mediante el cual se recupera una instrucción de memoria y se almacena en un registro dentro del procesador para su posterior ejecución. A continuación, vemos el diagrama básico del fetch de una instrucción, y llamamos asi, al procedimiento de apuntar a una dirección de la memoria de instrucciones mediante PC (Program Counter), y extraer de la memoria esa instrucción direccional. Entonces lo que sale de la memoria (flecha Instruction), van a ser 32 bits:



_ Al igual que en el micro procesador ARMv4, la memoria de instrucciones está organizada de a bytes, recordando que 1 byte son 8 bits. Entonces, si queremos formar una instrucción de 32 bits, con palabras de 8 bits, lo que vamos a necesitar son 4 palabras de memoria. Es por eso que cuando leemos una instrucción, con el dato de PC que nos dice donde estaba la instrucción que leímos, le sumamos 4, es decir, que avanzamos 4 posiciones en la memoria. Pero al avanzar esas 4 posiciones en la memoria, solo pasamos a la instrucción siguiente. Entonces, en el diagrama anterior lo único que tenemos es el hardware que necesitamos para registrar la instrucción que vamos a leer y despues calcular como llegar a la instrucción siguiente para ejecutarla.

_ Notamos que el PC también es un registro, aunque no esté dentro del Register File, y también es de 64 bits, por lo tanto, la memoria de instrucciones va a tener el doble de capacidad, porque tenemos el doble de capacidad de direccionamiento.

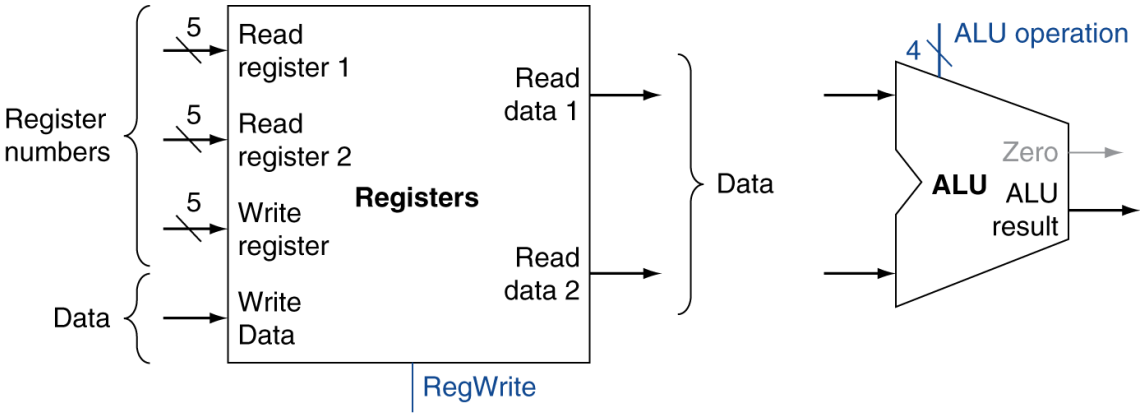
Instrucciones

_ Analizamos los tipos de instrucciones que tenemos en el sub set de LEGv8:

Banco de registros: es una colección de registros donde cualquier registro puede leerse o escribirse especificando su número.

Instrucciones R: el tipo más básico de instrucciones que tenemos son las tipo R. Estas son operaciones entre registros. El hardware que vamos a necesitar para poder ejecutar estas instrucciones, además del recurso de la memoria de instrucciones y el PC, son aquellos que nos permitan acceder al bloque de registros.

_ Podemos observar en el gráfico que el bloque de registro tiene 3 entradas de address, en donde 2 se usan para direccionar los dos registros que podemos leer y 1 es para direccionar el registro que vamos a escribir como resultado. La operación entre registros toma dos valores, hace alguna operación en la ALU (Unidad Aritmética Lógica) y luego escribe el resultado de la operación en un tercer registro, que puede ser alguno de los anteriores o no. Las salidas en Read data 1 y en Read data 2 van a ser de 64 bits, y por otro lado, al tener 32 registros, vamos a direccionar con 5 bits, ya que con 2^5 tenemos 32 combinaciones posibles de números binarios, que son los que representan el número del registro al que estamos accediendo. El registro 31, es un registro que siempre tiene el valor 0, es por eso cuando operamos con registros, que denominamos X y no R, el X31 o XZR responde al registro 31 que siempre tiene el valor 0. A continuación, vemos que el banco de registros y la ALU son los dos elementos necesarios para la implementación de instrucciones de tipo R (R-format):



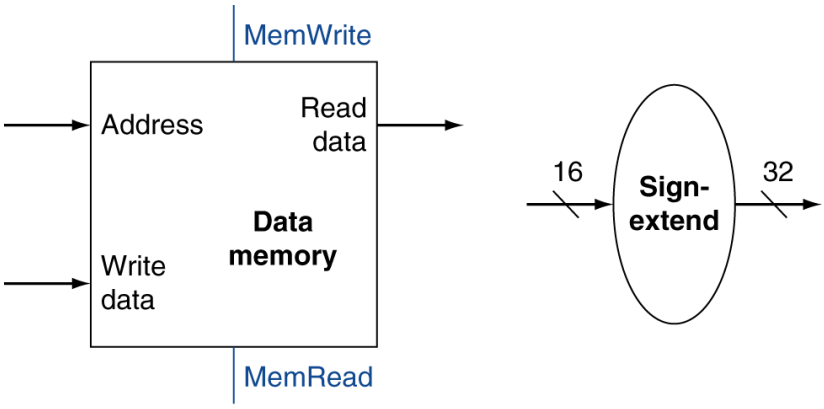
a. Registers

b. ALU

Instrucciones de acceso a memoria (load/store): estas instrucciones pueden ser

- Load: estas permiten cargar en un registro un dato que previamente sacamos de memoria.
- Store: estas se usan cuando tomamos el valor de un registro y lo guardamos en la memoria.

_ Para esto lo que usamos es la memoria de datos. Si bien la memoria es una sola, por cuestiones didácticas separamos por un lado la memoria de instrucciones y por otro la memoria de datos. Pero ambas memorias tienen la misma organización, es decir, que la memoria de datos esta ordenada con palabras de bytes (8 bits). Entonces, debido a que con las instrucciones nos movemos de a 4 bits, al llegar a esta etapa, como nos tenemos que mover de datos, ahora son de a 8 bits. Entonces, las dos unidades necesarias para la implementación de cargas y almacenamientos, además del banco de registros y la ALU, son la unidad de memoria de datos y la unidad de extensión de signo:

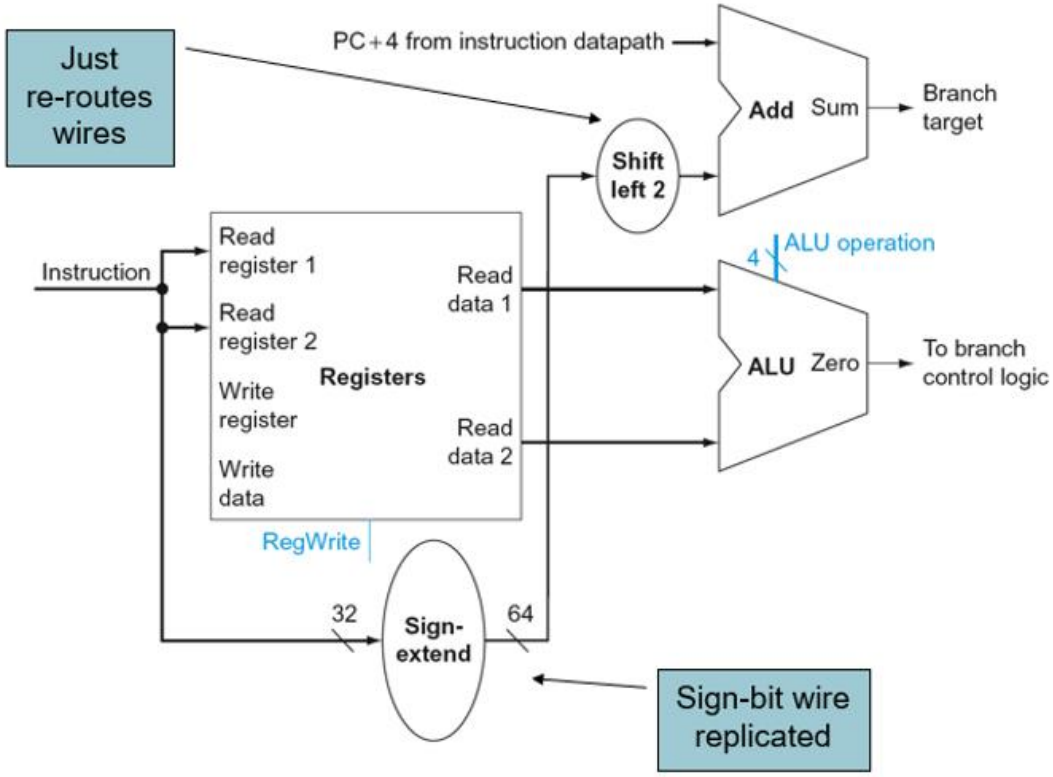


a. Data memory unit b. Sign extension unit

_ Entonces, para acceder a la memoria de datos, los tipos de instrucciones que tenemos están conformados por un registro y un valor inmediato (offset). Por ende, para calcular la dirección a la que vamos a acceder, tomamos el valor de ese registro, le sumamos el offset, y es por eso que aparece en esta operación el Sign-extend que lo que hace es tomar un numero inmediato y le extiende el signo para mantener la validez del bit de signo al extender el valor de bits más bajos a bits más altos. Por ejemplo, si tenemos un valor de 8 bits, y queremos extenderlo a 16 bits, se tomará el bit más significativo (el bit de signo) y se repetirá en los 8 bits adicionales. Esto asegura que el valor extendido tenga el mismo signo que el valor original.

_ Lo que entra a Sign-extend son los 32 bits de la instrucción, luego se toma el valor real del inmediato y se extiende a 64 bits porque en la memoria de datos tenemos que operar con un registro que es de 64 bits, entonces a partir de ahí podemos sumar 64 con 64, y asi obtener la dirección de acceso a memoria.

Instrucciones de salto (branch): estas nos permiten que, por ejemplo, al venir ejecutando un programa que arranco en la instrucción 0, en donde el PC está en 0, se ejecuta dicha instrucción, al PC se le suma 4 y pasamos a la siguiente instrucción, ejecutamos esta última y asi vamos avanzando de a una instrucción. Salvo que nos encontremos con un branch o una instrucción de salto, y en ese caso el PC va a tener que cambiar, y nos va a llevar a lo que en el código vemos como un label, pero en realidad lo que el micro hace es calcular a donde esta ese label para poder saltar hasta ahí. El camino de datos, para un salto condicional, utiliza la ALU para evaluar la condición del salto, y un sumador aparte para calcular la dirección destino del salto como la suma del PC incrementado y los 16 bits de menor peso de la instrucción con el signo extendido (el desplazamiento de salto) y desplazado dos bits a la izquierda. Particularmente en el micro controlador que estamos viendo, lo que tiene implementado es un tipo de salto que se llama compare branch si es 0.



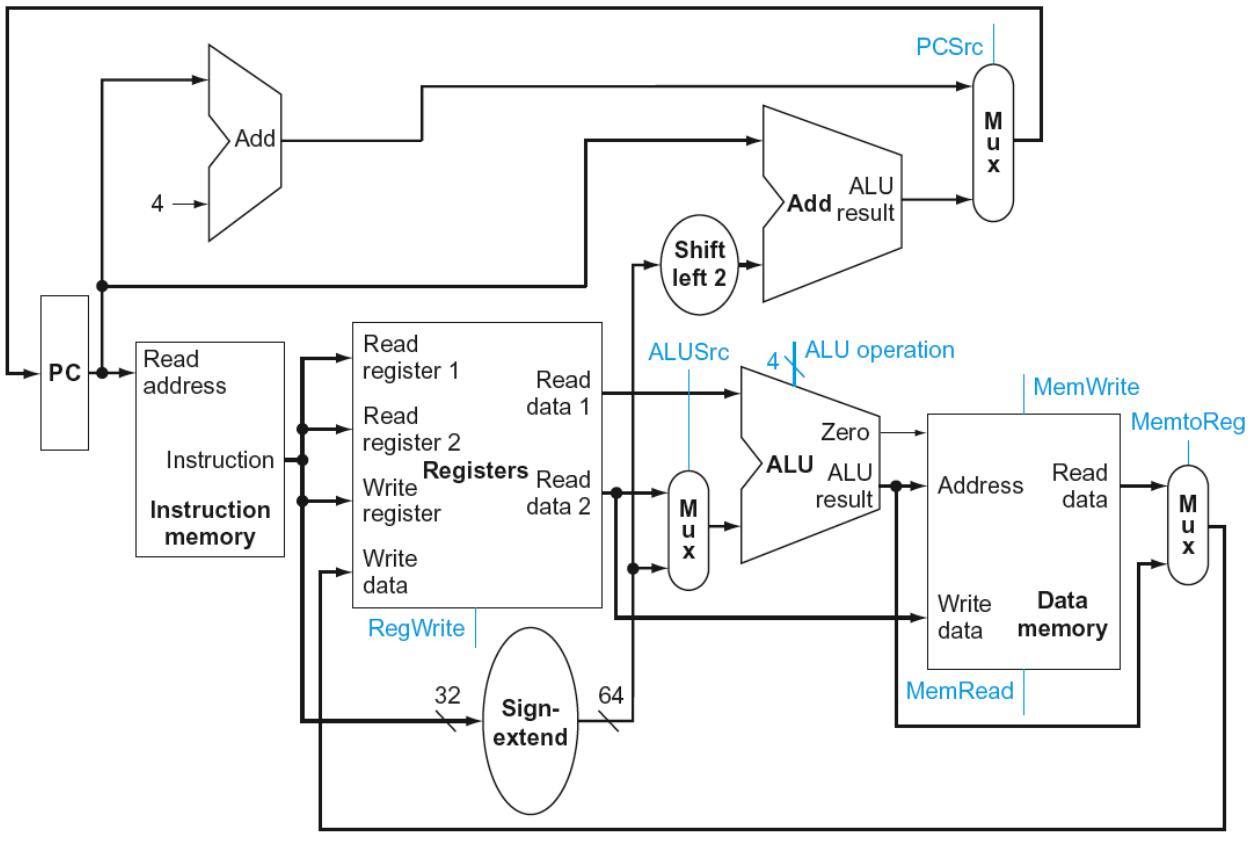
_ El funcionamiento de este salto es el siguiente:

- Toma un registro, que es el que va a comparar.
- Ese registro pasa a través de la ALU, en una operación que se llama pass input B. La ALU en realidad no hace ninguna operación sino que simplemente ese dato pasa a la salida y ya es el resultado, pero lo que se hace es evaluar si ese valor que sacamos del registro es 0. La ALU tiene dos salidas, una es la salida del resultado de la operación que hayamos hecho, y la otra es la flag o bandera de 0.
- Entonces, hasta ahora sacamos el registro del bloque de registros, pasamos por la ALU, sin operar nada, sino que simplemente se verifica si ese registro está en

0 o no. En donde si está en 0, queremos que el salto se ejecute, caso contrario no. Esto sería por un lado el control de la ejecución o no del salto.

- Por otro lado, tenemos que calcular hacia donde vamos a saltar. En este caso, lo que se pasa como dato en la instrucción, no es otra instrucción, ya que los saltos son relativos. Lo que pasamos en la instrucción es cuantas instrucciones deberíamos saltar en memoria, en donde si es para adelante, es un numero positivo y si es para atrás será un numero negativo, pero siempre como numero de instrucción.
- Ese número de instrucciones que pasamos, que por lo pronto es un inmediato, que viene dentro del campo de instrucción, pasa por el bloque sign-extend (que funciona distinto para cada tipo de instrucción) y a ese número de instrucción que pasamos como dato se lo extiende a 64 bits.
- Despues hacemos un shift left 2, que se utiliza para multiplicar por 4, y se multiplica por 4 porque le pasamos el número de instrucciones pero el salto es referido al contador del programa, y este último apunta a la memoria, y en la memoria cada instrucción ocupa 4 posiciones. Es por eso que al multiplicar por 4 al número de instrucciones, los convertimos en posiciones de memoria. Por esto a la salida de shift left 2 obtenemos cuantas posiciones de memoria, hacia adelante o hacia atrás, son las que queremos saltar.
- Este último valor es el que se suma al PC, es decir, el resultado final es lo que efectivamente realimenta la entrada del PC, y nos dice cuál va a ser la siguiente instrucción despues de ejecutar el salto.
- Obviamente todo esto pasa si la bandera de si solo si dio 0, caso contrario simplemente pasamos a la instrucción siguiente.

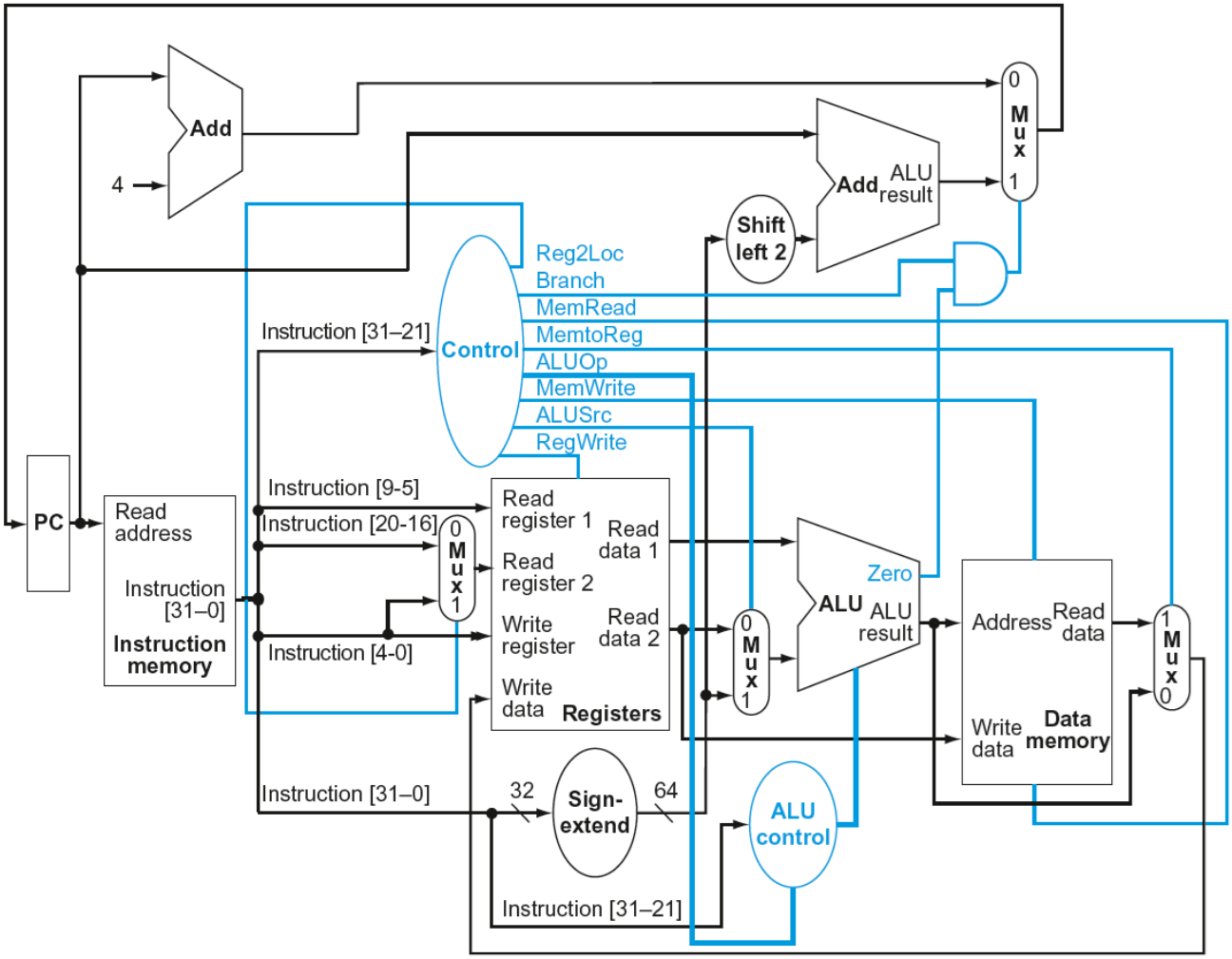
_ Finalmente, si combinamos los elementos de los diferentes tipos de instrucciones, formamos el siguiente micro:



Datapath con la unidad de control

_ El siguiente diagrama es muy similar al de ARMv4, pero de lo que sería LEGv8. Esta discriminado en negro la parte del datapath, y lo que está en azul es la parte de señales de control. Para generar la señal de control, a ese bloque entran 11 bits, y en la salida tenemos señales de control. Las salidas de la unidad de control son similares, pero la operación de la ALU se determina en una unidad separada (ALU control). El ALU control esta por fuera del control, en donde tienen la salida ALUOp de 2 bits que entra al ALU control, y recién en Control se determina cual es la operación que va a realizar la ALU.

_ Otras cosas a aclarar, es que la única instrucción de salto implementada es CBZ. Por otro lado, el bloque Sign-extend recibe la instrucción completa. La entrada a Read register 1 no está multiplexada. Y recordamos que tenemos 32 registros de 64 bits.



ALU control

_ El siguiente cuadro posee un listado de las instrucciones LEGv8 que efectivamente se pueden ejecutar en el datapath anterior. Por eso las que están implementadas en el hardware anterior son:

- LDUR: es un load hacia un registro, de una palabra de memoria de 64 bits.
- STUR: guarda los 64 bits de un registro en memoria.
- CBZ: cuyo valor es 0.
- R-type: de estas solo vamos a tener las funciones add, subtract, AND, ORR y NOR.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	pass input b
1100	NOR

_ A continuación, tenemos cuales son las señales de control para que la ALU ejecuta una u otra operación:

- Load/Store: siempre la ALU va a ser una suma (add), porque la ALU se utiliza para calcular dirección de acceso a memoria y lo que va a calcular es el contenido del registro que tiene la dirección con el offset que pasamos como inmediato.
- CBZ: en este caso será pass input b en donde simplemente toma el registro y lo envía a la salida del resultado para evaluarlo si es 0 o no (subtract).
- R-type: acá se va hacer una u otra operación dependiendo de la instrucción que estemos por ejecutar.

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	0111
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		ORR	100101	OR	0001

Setting of the control lines

_ Podemos observar cómo están configuradas las señales de control ante la ejecución de cada una de estas instrucciones:

Instruction	Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	0	0	0	1	0	0	0	1	0
LDUR	X	1	1	1	1	0	0	0	0
STUR	1	1	X	0	0	1	0	0	0
CBZ	1	0	X	0	0	0	1	0	1

_ Podemos observar que en algunos casos hay X que significan la condición no importa. Por ejemplo, LDUR, en Reg2Loc, tiene una X que significa que como Reg2Loc no usa ese registro para nada, da lo mismo si este multiplexor deja pasar lo que está en 0 o en 1, porque la salida de ese registro nunca se va a leer. Pero en todos los otros casos si es importante que estén configurados en 0 para seleccionar esa entrada, o en 1. Y siempre lo que sea uso de recursos como escribir un registro o escribir o leer en memoria, entre otras, también tienen que estar indicados para no usar un recurso que no es necesario y para no escribir algo erróneo. Entonces, lo que tenemos acá es como configuramos cada uno de los casos para que se ejecute correctamente la instrucción y para que no se produzca ningún error o nada que no corresponda.

Operaciones aritméticas

_ A nivel de código o instrucciones, lo más básico que vamos a ver son las operaciones aritméticas, donde tenemos por ejemplo tres operandos, en donde el primer operando en este caso “a” es el operando destino y los otros dos son la fuente. En a se va a guardar b+c.

```
ADD a, b, c     // a gets b + c
```

_ A continuación vemos el siguiente código en C:

```
f = (g + h) - (i + j);     // g es X20, h es X21, i es X22, j es X23
```

_ Su ejecución en LEGv8 seria:

```
ADD X0, X20, X21     // X0 = g + h
ADD X1, X22, X23     // X1 = i + j
SUB X24, X0, X1      // f es X24, en donde X4 = X0 - X1
```

_ Nunca podemos operar con la memoria directamente, cada acceso a memoria es muy lento, y es por eso que siempre se trata de operar entre registros.

Campos de instrucciones

_ Todas las instrucciones de cualquier tipo son de 32 bits, y lo que va a cambiar son como están ordenados los distintos campos de esa instrucción.

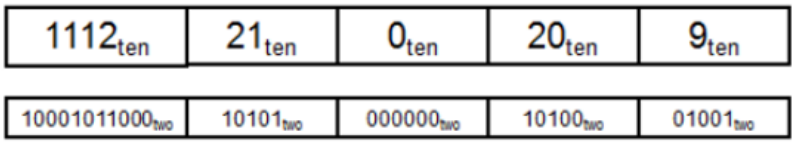
Instrucciones R: tenemos los siguientes campos:

- Opcode (operation code): tenemos los 11 bits más significativos, nos dice que instrucción es, y es la que se usa para configurar todas las señales de control.
- Rm: los siguientes 5 bits corresponden a la dirección del segundo registro de operandos. Son 5 bits ya que podríamos estar usando cualquier registro entre el 0 y el 31
- Shamt (shift amount): estos 6 bits se utilizan para las instrucciones de shifteo, y son 6 bits porque podemos movernos 64 lugares para un lado o para el otro en memoria, ya que vamos a estar trabajando sobre un registro, y asi shifteamos un registro completo. (00000 for now)
- Rn: los siguientes 5 bits son el primer registro operando.
- Rd: finalmente los 5 bits menos significativos corresponden al registro destino que es donde guardamos el resultado de la operación.

_ A modo de ejemplo si queremos ejecutar la operación:

ADD X9, X20, X21

_ Podemos ver como quedaría ensamblada la instrucción en binario o en hexadecimal:



$$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{two} = 8B150289_{16}$$

Instrucciones de acceso a memoria (Load/Store): cuando trabajamos con la memoria esta está organizada de a bytes, y en la mayoría de los casos si no se dice lo contrario, estamos trabajando con elementos de 64 bits. A continuación, vemos un ejemplo en C en donde queremos guardar en el elemento 12 del arreglo A, h más el valor del elemento 8 del arreglo A:

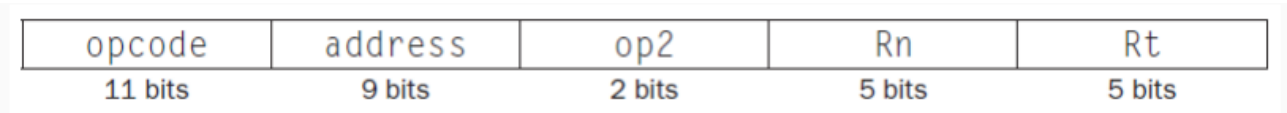
```
A[12] = h + A[8];      // h es X21, base address de A que será X22
```

_ Su ejecución en LEGv8 seria:

```
LDUR X9, [X22,#64]     // Index 8 requires offset of 64
ADD X9, X21, X9
STUR X9, [X22,#96]
```

_ Como no podemos trabajar contra un valor de memoria, lo primero que hacemos para poder ejecutar esta suma, es buscar el elemento 8 del arreglo A, traerlo a un registro para asi luego poder operar contra h, y a esto lo hacemos con LDUR. El registro destino en este caso es X9 y en las llaves tenemos el valor del registro X22 con el offset 64. Esto es asi, porque para calcular la dirección a la que vamos a acceder en la memoria en este caso, va a ser el contenido de X22 sumado al número 64. Sabemos que X22 tiene la dirección base de A, o sea tiene la dirección del elemento 0 y queremos ir al elemento 8, es por eso que se le suma 64, porque estamos hablando de posiciones de memoria, en donde si la memoria está organizada de a bytes y queremos acceder a palabras de 64 bits, cada palabra ocupa 8 posiciones de memoria. Entonces para llegar al elemento 8, tenemos que hacer 8 por las 8 posiciones de memoria que ocupa cada uno de ellos y de ahí sale el 64. Una vez calculada la dirección del elemento 8, con LDUR lo que hacemos es acceder a la posición de la memoria de X22, y obtenemos el valor de esa posición con los 7 bytes siguientes, siendo 8 en total (gracias a los 64 bits), y los guardamos en el registro X9. Despues de eso hacemos un ADD con X21 para sumar h más el elemento de A, y finalmente tenemos que guardar ese resultado, en el elemento 12 de A. Para guardar en memoria usamos STUR, en este caso guardamos en X9, accedemos de nuevo a la dirección base de A en X22 pero con un offset de 96 para movernos 12 elementos en el arreglo o 96 posiciones de memoria hasta poder acceder al elemento 12 del arreglo.

Intrusiones D: son todas aquellas que acceden a transferencia de datos



_ En este caso tenemos que:

- Opcode: en este caso, nuevamente es de 11 bits.
- address: tenemos 9 bits para el offset (positivos o negativos) dándonos un límite de los valores que podemos poner (-256 to 255).
- op2: temenos 2 bits “00”.
- Rn: son 5 bits para el registro donde tenemos guardado la dirección base al que vamos a acceder y luego le sumamos el offset.
- Rt: tenemos 5 bits para lugar donde ponemos el valor del destino destination (load) o el registro donde tengo el valor que queremos guardar en memoria (store).

Instrucciones de salto (branch): en este caso si tenemos tantos saltos condicionales como no condicionales.

- CBZ: en este caso nos fijamos en el valor del registro, en donde si es 0, entonces hacemos el salto, calculando hacia donde saltar.
  CBZ register, L1
  if (register == 0) branch to instruction labeled L1;

- CBNZ: funciona de manera opuesta a CBZ, es decir, verifica si el valor del registro no es 0
  CBNZ register, L1
  if (register != 0) branch to instruction labeled L1;

- Branch unconditonally: que significa que cuando el código llega a ese salto, directamente ese salto se ejecuta, ya que no hay ninguna condición.
  B L1

Instrucciones I: estas instrucciones lo que hacen es operar entre un registro y un inmediato, y guardan el resultado en un registro, por ejemplo podemos hacer un:

ADD X1, X2, #8

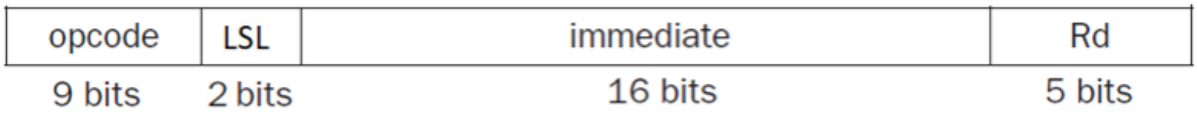
_ De esta forma podemos poner un número que no está en un registro, sino que es un inmediato. Son muy similares a las tipo R, en donde la estructura es:

_ En este caso tenemos que:

- Opcode: es de 10 bits para aumentar el campo del inmediato.
- Immediate: tenemos hasta 12 bits para poner un valor inmediato.
- Rn: 5 bits para el registro donde tenemos guardado la dirección base al que vamos a acceder y luego le sumamos el offset.
- Rd: finalmente los 5 bits menos significativos corresponden al registro destino que es donde guardamos el resultado de la operación.

Instrucciones IW: estas nos permiten cargar un valor en un registro, e indicarle en que parte del registro vamos a poner ese valor. Tenemos dos tipos de instrucciones:

- MOVZ: toma un valor inmediato que le pasamos, que como vemos en este caso es de 16 bits, lo pone en alguna parte del registro, y al resto de los bits los pone en 0.
- MOVK: toma los 16 bits que pasamos en el campo de la instrucción, los pone en alguna parte, pero no modifica el resto de los valores del registro.



_ En este caso tenemos que:

- Opcode: es de 9 bits.
- LSL: de 2 bits.
- Immediate: que es de 16 bits.
- Rd: finalmente los 5 bits menos significativos corresponden al registro destino que es donde guardamos el resultado de la operación.

_ Entonces si hacemos un MOVZ al destino X2, ese X2 se va a poner todo en 0 y el valor que le pasamos como inmediato. Ahora si hicimos un MOVK, lo primero que hace es leer el valor del registro X2, para no modificarlo, y solo pisar esos 15 bits que queremos escribir.

_ En LSL tenemos 2 bits que están indicados como LSL que es un shift left. En esos dos bits vamos a codificar cuantos bits queremos movernos hacia la izquierda con ese valor inmediato que pusimos. Por ejemplo, ponemos el número 17 en X2 y aplicamos LSL 0, nos va a poner el número 17 en la parte más baja del registro, pero si ahora queremos poner el número 17 en X2 pero con LSL 16, lo que va a hacer es ponernos ese número pero lo va a shiftear 16 lugares a la izquierda, entonces lo coloca entre los bits 16 y 32 de ese registro.

Bit 22	Bit 21	LSL
0	0	0
0	1	16
1	0	32
1	1	48

_ Repetimos entonces, si hicimos un MOVZ, esos 16 bits quedaron con valores y el resto en 0. Si hicimos un MOVK, hicimos una máscara guardando lo que había en ese registro, y simplemente se pisaron esos bits entre el 16 y el 32. También podemos hacer un LSL de 32 o de 48, en donde es como si al registro de 64 bits lo partimos en 4 y le indicamos en cual de esos cuartos de ese registro queremos cargar el valor inmediato.

_ A modo de ejemplo vemos distintas ejecuciones de MOV:

- Visualizamos el contenido de X9:

110100101	01	0000 0000 1111 1111	01001

- Luego ejecutamos lo siguiente y vemos como queda X9, con todos 0s y entre los bits 16 y 32, tiene cargado el valor 255:
  MOVZ X9, 255, LSL 16

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 1111 1111	0000 0000 0000 0000

- Luego de ejecutar lo anterior ahora queremos ejecutar:
  MOVK X9, 255, LSL 0

111100101	00	0000 0000 1111 1111	01001

- Vemos como resultado en X9:

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 1111 1111	0000 0000 1111 1111

Flags

_ Tenemos otros tipos de saltos o branchs, que dependen de otras condiciones. Pero además, tenemos que saber que hay una serie de instrucciones que terminan en S que hace referencia a setear flag. En el micro controlador tenemos una serie de bits dentro de un registro que nos dan cierta información sobre el resultado de una operación:

- Negative (N): por ejemplo, si hacemos una operación que setea flags y el resultado da negativo, se activa la bandera de negativo y setea el valor de la misma en 1, indicándonos que el resultado de esa operación fue negativo.
- Zero (Z): de igual manera que lo anterior, hacemos una operación que setea flags, en donde dependiendo el resultado, se setea el valor de la flag en 1, si el resultado que estableció el código de condición fue 0, caso contrario será 0.
- Carry (C): estas dos banderas son similares. El carry sirve por ejemplo cuando hacemos una suma más grande de lo que podemos guardar, y básicamente indica que el resultado no puede entrar en el registro destino, o también cuando tenemos que pedir un 1. El Carry funciona para los números sin signo.
- Overflow (V): este nos indica que hubo un cambio en el valor del resultado que nos cambió el signo, entonces si nuestro resultado debería ser positivo, y al hacer la operación de suma nos dio negativo o al revés. El Overflow funciona para los números con signo.

Signed and Unsigned numbers	
Instruction	CC Test
Branch on minus (B.MI)	N= 1
Branch on plus (B.PL)	N= 0
Branch on overflow set (B.VS)	V= 1
Branch on overflow clear (B.VC)	V= 0

_ A continuación vemos algunos casos de Overflow:

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

_ El seteo de banderas nos va a servir para entender como era la relación entre los registros con los que estuvimos esperando.

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	~(Z=0 & N=V)	B.LS	~(Z=0 & C=1)
>	B.GT	(Z=0 & N=V)	B.HI	(Z=0 & C=1)
≥	B.GE	N=V	B.HS	C=1

_ Otros saltos que debemos mencionar son:

Branch with Link (BL): básicamente lo que hace es un salto no condicional pero que además guarda en el registro X30, el valor de la siguiente instrucción. Esto nos sirve por ejemplo para saltar a una función o un condicional, y para luego poder volver a la ejecución del código normal.

Branch with Register (BR): este es el único branch que nos permite saltar a cualquier parte del código. Saltamos al valor que está contenido en cualquier registro mediante el PC.

_ Entonces, por ejemplo, si queremos implementar una función, en nuestro código en el main, ponemos un BL, nos redirige a otra parte del código en donde se ejecuta lo que queremos, y al final de esa parte del código colocamos un BR a X30 y volvemos a la parte original del código. El problema de esto es que siempre se guarda en X30.

Procesador con pipelines

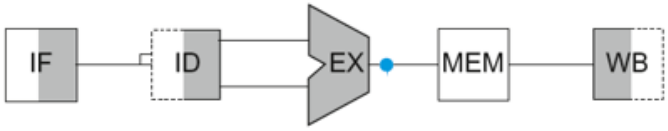
Pipelining

Concepto

_ El pipelining es una técnica de implementación que consiste en solapar la ejecución de múltiples instrucciones. De esta manera, dividimos el micro LegV8 en cinco etapas:

- IF: instruction fetch from memory.
- ID: instruction decode and register read.
- EX: execute operation or calculate address.
- MEM: access memory operand.
- WB: write result back to register.

_ Cuando se ejecutan las cinco etapas, el tiempo de cada una es igual a la que tarda más tiempo en ejecutar una es igual a la que tarda más tiempo en ejecutarse.



_ El pipelining mejora las prestaciones incrementando la productividad de las instrucciones, en lugar de disminuir el tiempo de ejecución de cada instrucciones individuales.

Etapas del pipeline

_ Básicamente lo que hicimos al momento de agregar el pipeline, fue tomar el micro LEGv8, y lo dividimos en distintas partes, permitiendo que haya cosas que se ejecuten en simultaneo. Siempre lo que nosotros queremos es que se ejecuten más instrucciones en menos tiempo, y esto en principio se lograría con el pipeline. A continuación, explicamos cada etapa:

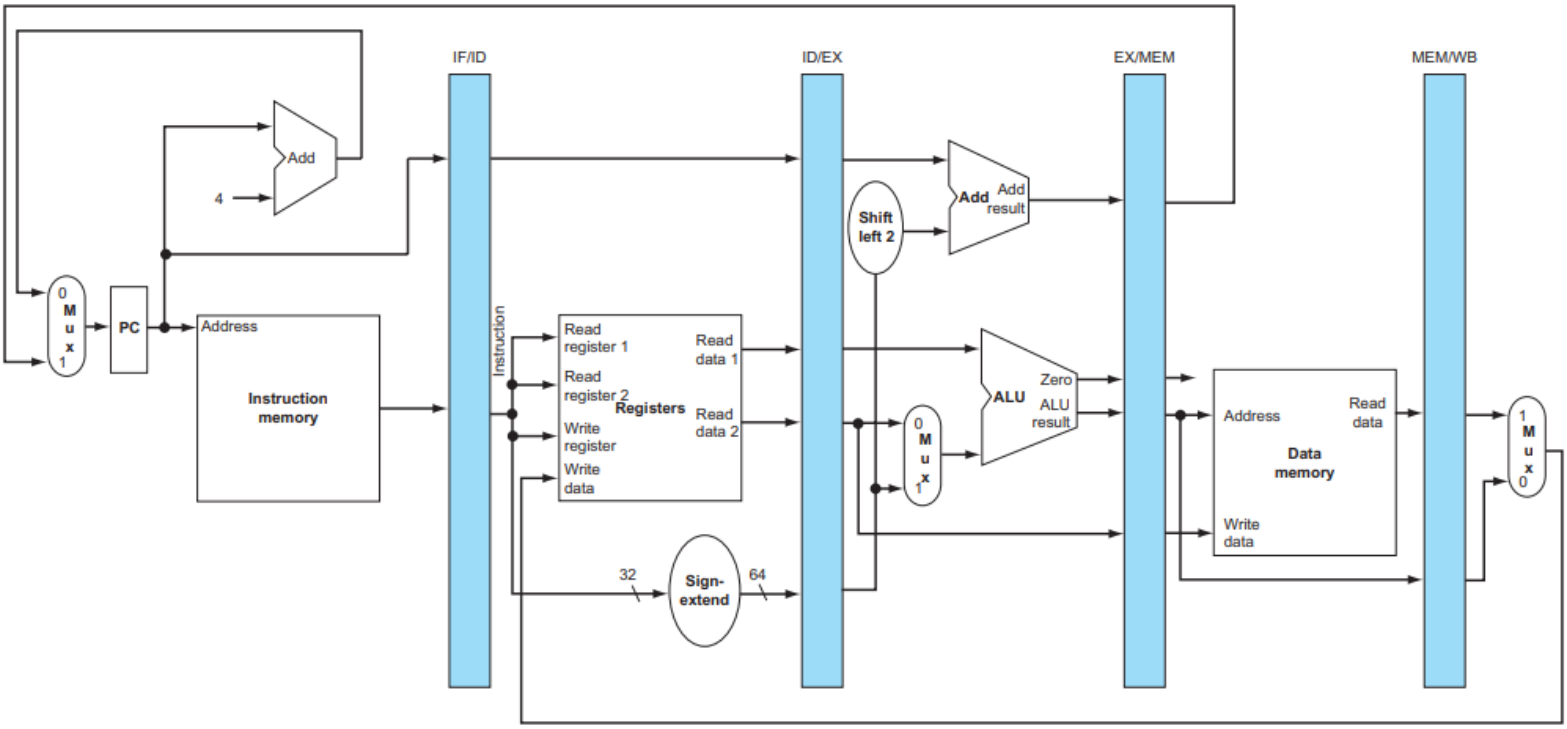
Etapa de fetch: el fetch de una instrucción es cuando vamos a la memoria de instrucciones, direccionamos con el valor que tengamos en el program counter, y sacamos esa instrucción correspondiente de la Instruction memori.

Etapa de decode: lo que hacemos es preparar cuales son los valores con los que vamos a trabajar. Ya sea que tengamos una operación con registros, entonces tendremos que leer los registros, o si tenemos que trabajar con un inmediato, por lo tanto tendremos que extenderle el signo.

Etapa de execute: en esta etapa ya se calculan cosas. Entonces, tenemos a la ALU trabajando por un lado, y por otro el sumador que usamos para calcular la dirección de salto. En este micro solo tenemos implementado el LDUR, STUR, ADD, SUB, AND, OR, y el CB si es 0, entonces a ese sumador lo usamos para calcular la dirección a la que vamos a saltar en el caso de un CB.

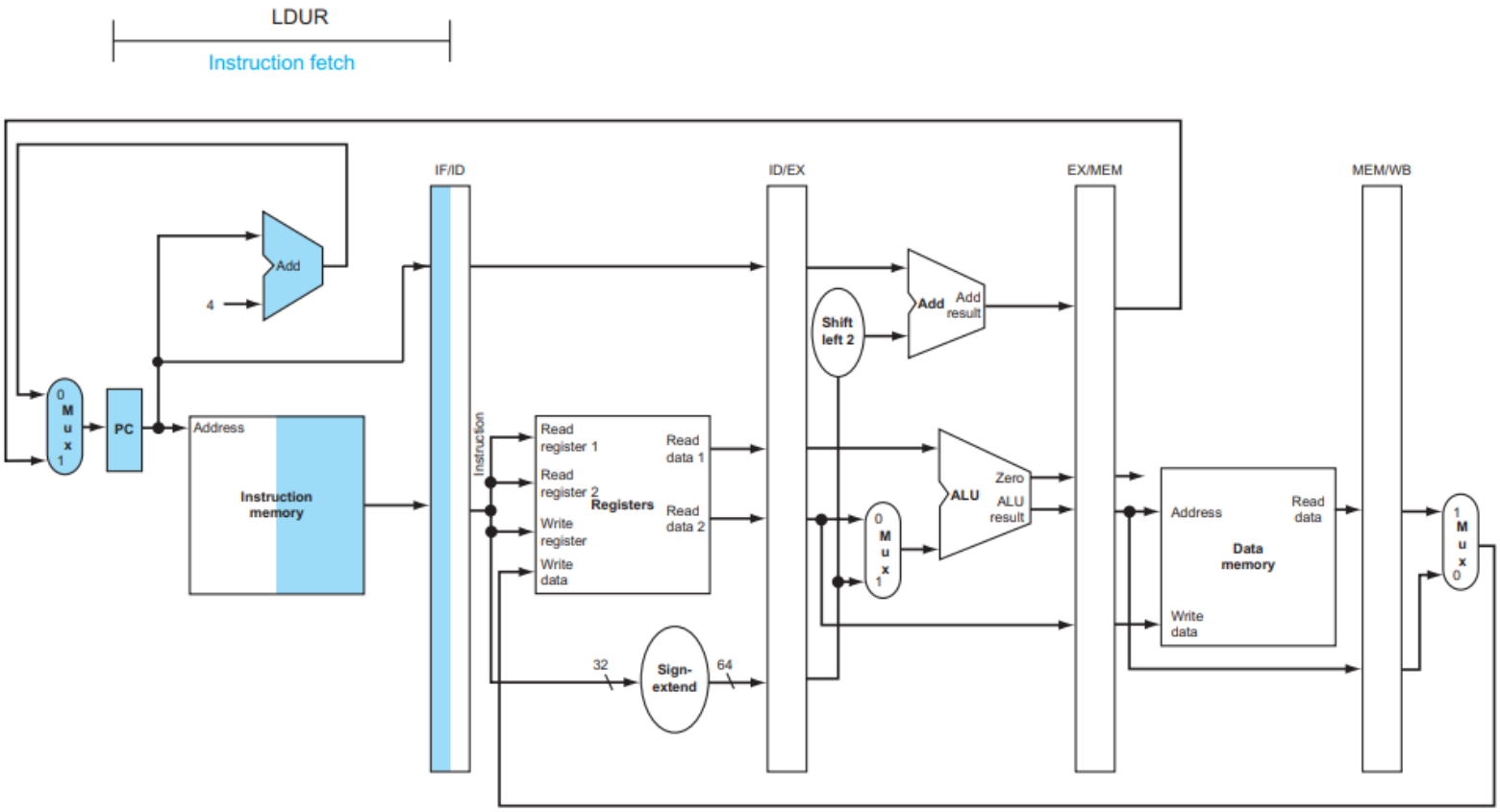
Etapa de memori: tenemos por un lado la actualización del program counter, como podemos ver, despues de haber pasado el registro EX/MEM, el dato se reinyecta en el program counter en el caso de un salto, si no hay un salto, en cuanto se lo lee, en el ADD de la primera etapa, se le suman 4 para que quede listo esperando el siguiente ciclo de clock para leer la siguiente instrucción. Por último, en esta etapa, como tenemos la memoria, todos los accesos a memoria se hacen aquí.

Etapa de write back: cualquier dato, ya sea que haya salido como resultado de la ALU o de la memoria, se van a reinyectar en los registros. Entonces, es el momento en el que efectivamente, escribimos un resultado en el registro.

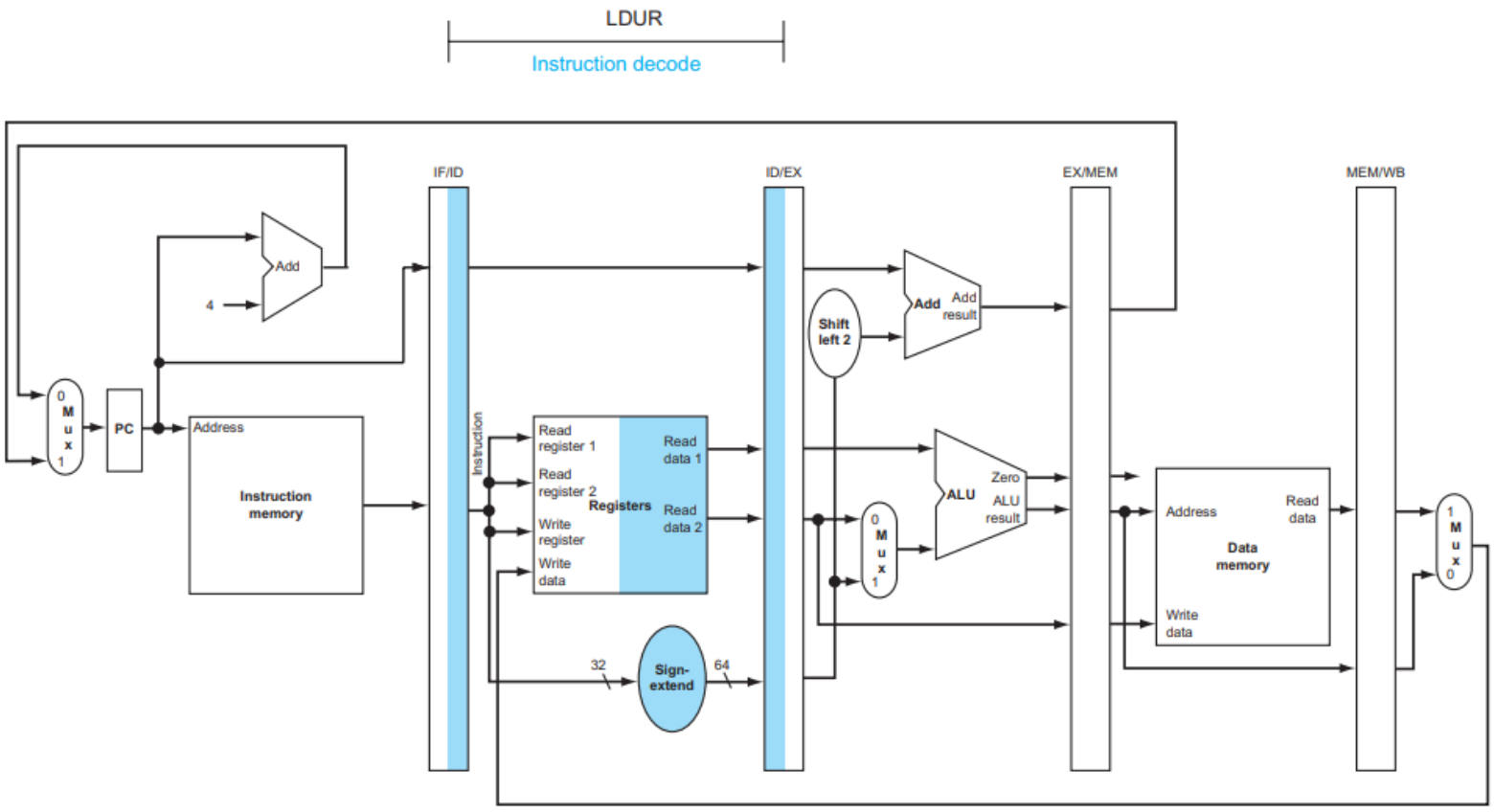


Analizamos la secuencia: cuando ejecutamos una instrucción, en el caso del procesador de un ciclo, todo el procesador está involucrado en la ejecución de esa instrucción.

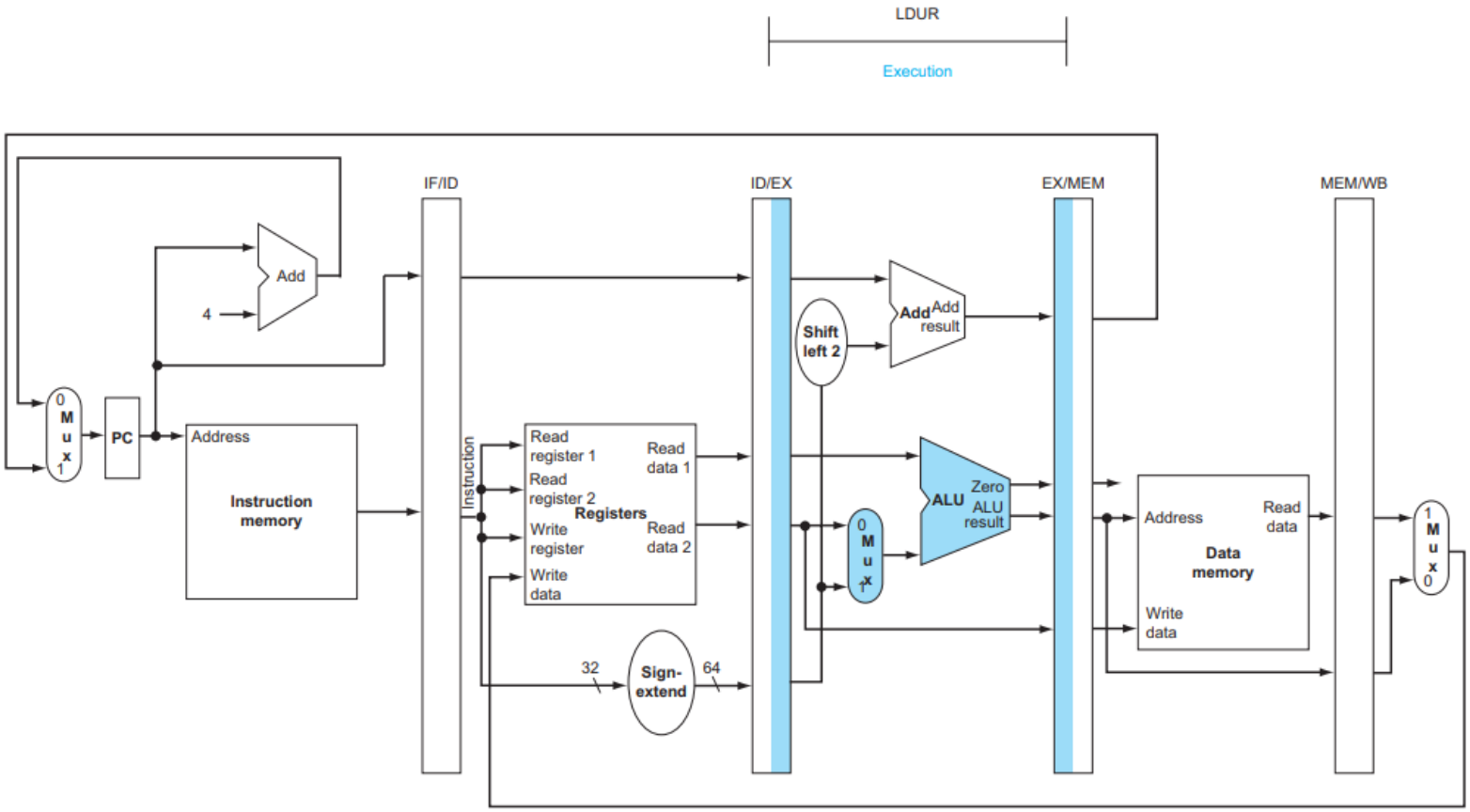
_ Entonces, según lo pintado de azul, vemos que primero viene el PC, identificamos una instrucción LDUR, y se empezaron a activar todos los componentes que participan en la ejecución de esa instrucción, hasta que termine. Pero en el caso del micro con pipeline, desde un ciclo de clock hasta el siguiente, solo vamos a estar como entre registros, entonces siguiendo el grafico, el PC apunta a la dirección 0 de address, donde tiene cargado un LDUR, entonces al venir un flanco de clock, se leyó la memoria de instrucciones, en donde la memoria saca un dato (por eso está de celeste), y ese dato se queda en la entrada del registro IF/ID, y ahí queda, porque hasta que no venga un flanco positivo de clock, esa información permanece ahí, y el resto del micro no está haciendo nada.



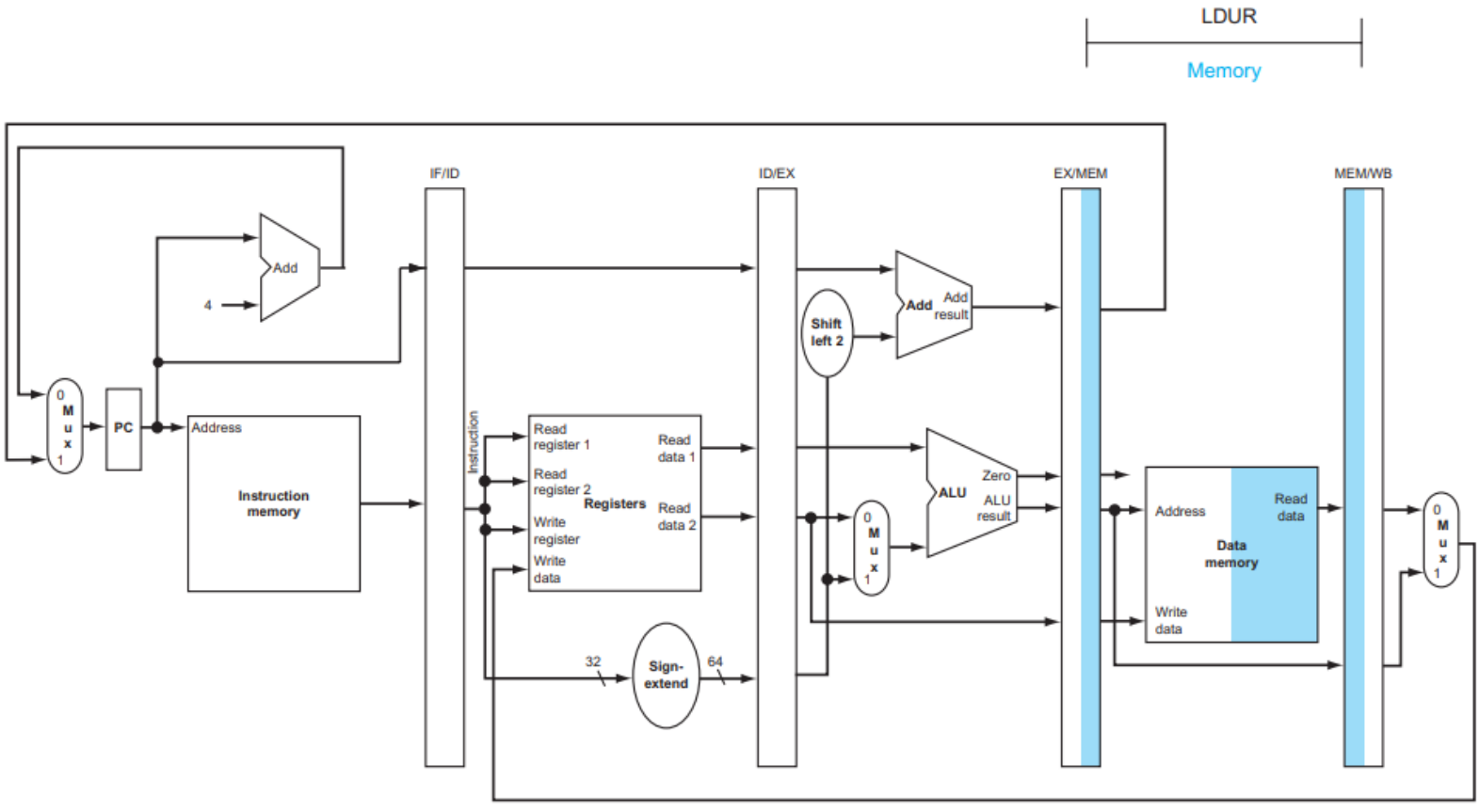
_ Cuando venga un flanco de clock siguiente, ahora sí la instrucción se copia al otro lado del registro IF/ID, y se sigue ejecutando el LDUR, en la etapa de decode. El resto de la Instruction memori queda vacía y ya no se ejecuta nada. Entonces, se hace la decodificación, y ya sea todo lo que sacamos del registro como los resultados del sign-extend, quedan en la entrada del registro ID/EX esperando a ser registrado, hasta que venga el nuevo flanco de clock.



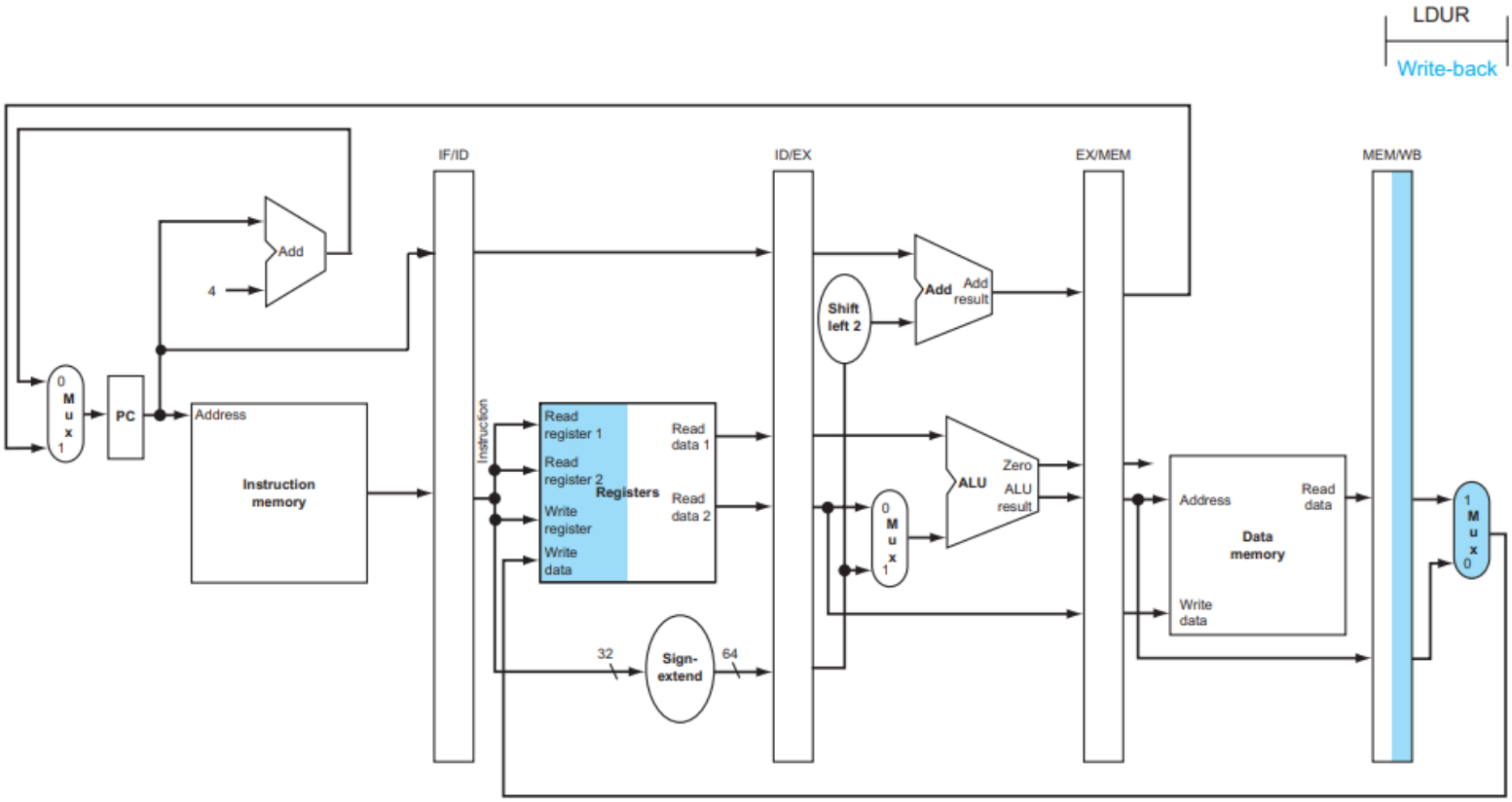
_ Al venir el nuevo flanco de clock, pasamos a la etapa de ejecución. Al ser un LDUR no va a usar nada de la parte de saltos, ya que simplemente usara la ALU para calcular la dirección de acceso a memoria.



_ Luego, cuando lleguemos a la etapa de memori, LDUR va a leer la memoria, quedando el resultado en la entrada del registro MEM/WB esperando ser registrado.

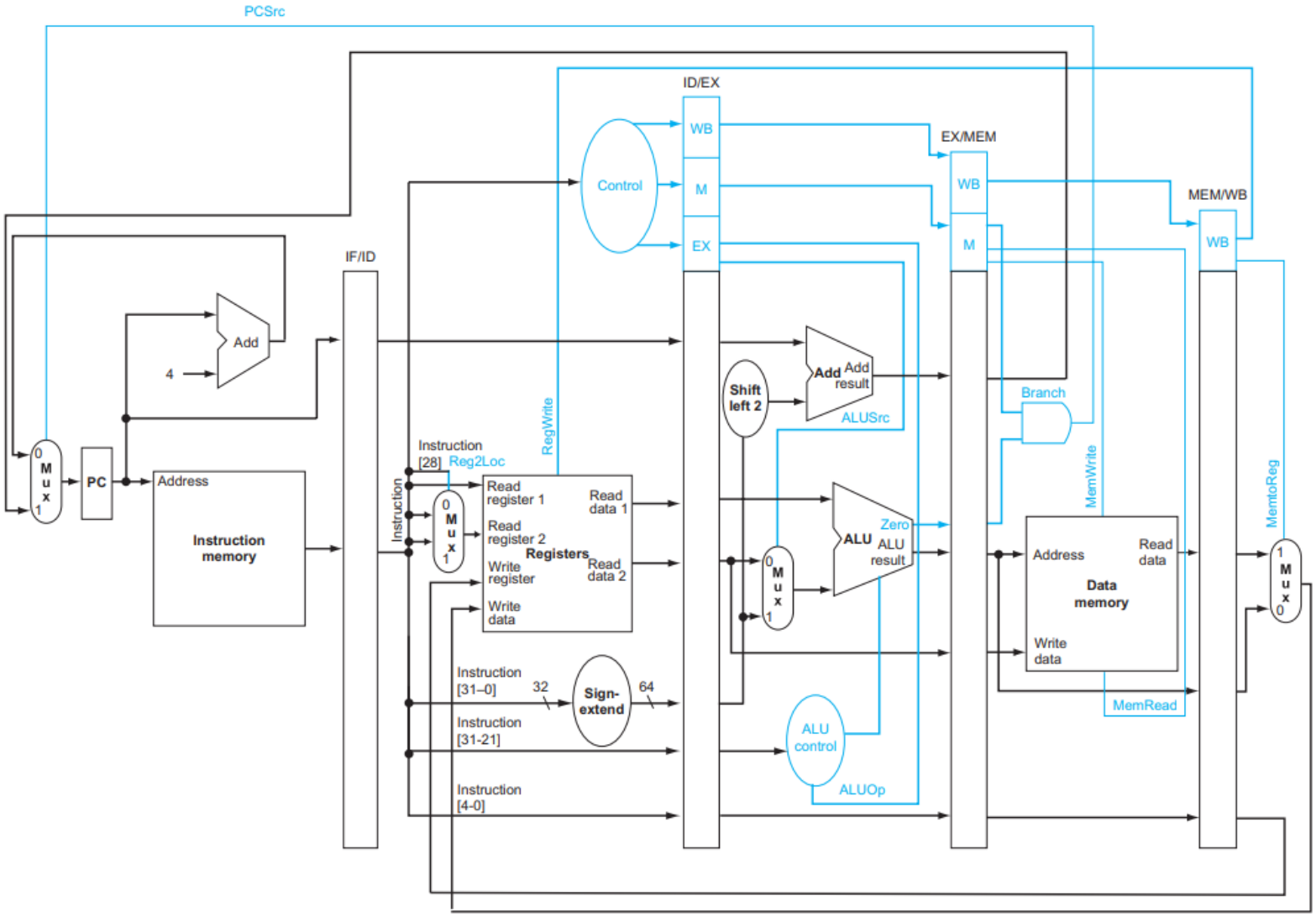


_ Finalmente, ese valor se va a ir y se va a guardar en el registro. El registro está pintado a la mitad, porque se asume que en la primer parte del tiempo, el resultado se escribe en el registro, y entonces luego lo podemos leer en cualquier momento. Es decir, la escritura del registro es síncrona, ya que apenas tengamos el resultado en la etapa de write back, vamos al registro y lo escribimos en él. Por otro lado, la lectura del registro es asíncrona, es decir, que hasta que no venga el próximo flanco de clock y queremos sacar un dato, el valor que vamos a registrar es el valor actualizado del registro. Entonces, si hay una instrucción en la etapa de write back y una instrucción en la etapa de decode, y la instrucción que está en write back escribe un dato que va a usar la instrucción que está en decode, no hay problema, porque la instrucción se va a escribir antes de que se quiera leer, y asi no se produce ninguna colisión.



Datapath con pipeline y control

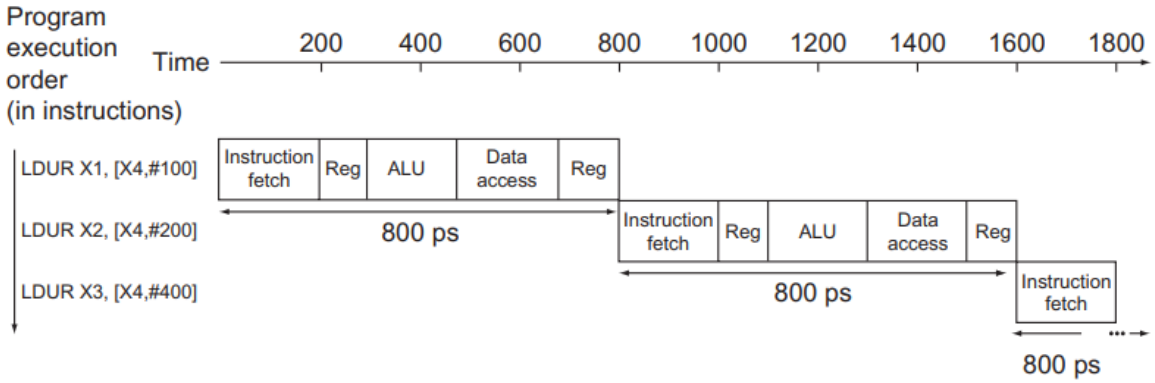
_ Es importante saber que el pipeline se extiende a las señales de control, ya que no solo necesitamos que se vayan registrando la parte de los datos, sino que también necesitamos registrar la parte de control, porque como la instrucción se va a ir moviendo entre etapas, queremos que las señales de control, que indican que es lo que se tiene que hacer, están asociadas a la instrucción que estamos ejecutando. Por ejemplo, a la vez que registramos el registro 2 mediante un LDUR, al llegar al ID/EX, además de llegar lo que estaba registrado tiene que venir la señal de control de la ALUSrc, entonces esa señal también se tiene que registrar para que llegue a tiempo con los datos.



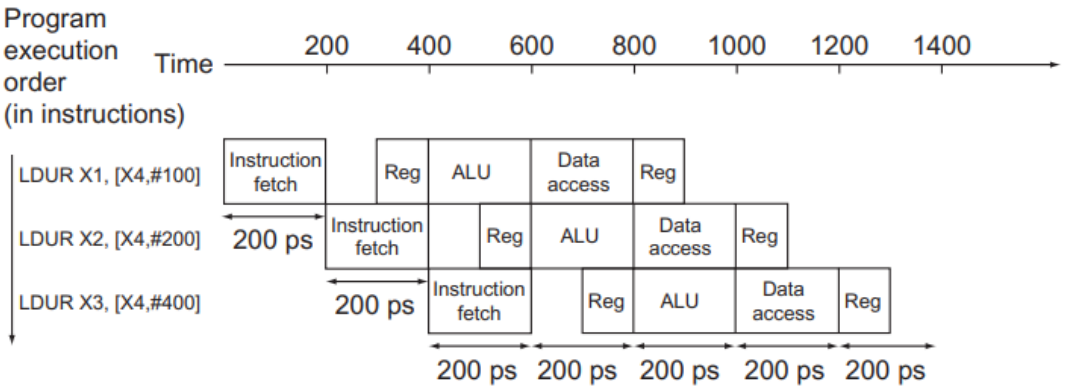
Diferencia micro sin pipeline y con pipeline

_ Vamos a determinar las diferencias que aparecen en los tiempos de ejecución cuando estamos en un micro sin pipeline o en otro con pipeline. A continuación, tenemos un programa que ejecuta tres instrucciones en orden (tres loads), y lo hace sobre un micro procesador que no tiene pipeline, de un solo ciclo. Despues se muestra como seria la ejecución ahora en el tiempo, de esas mismas tres instrucciones en el micro con pipeline.

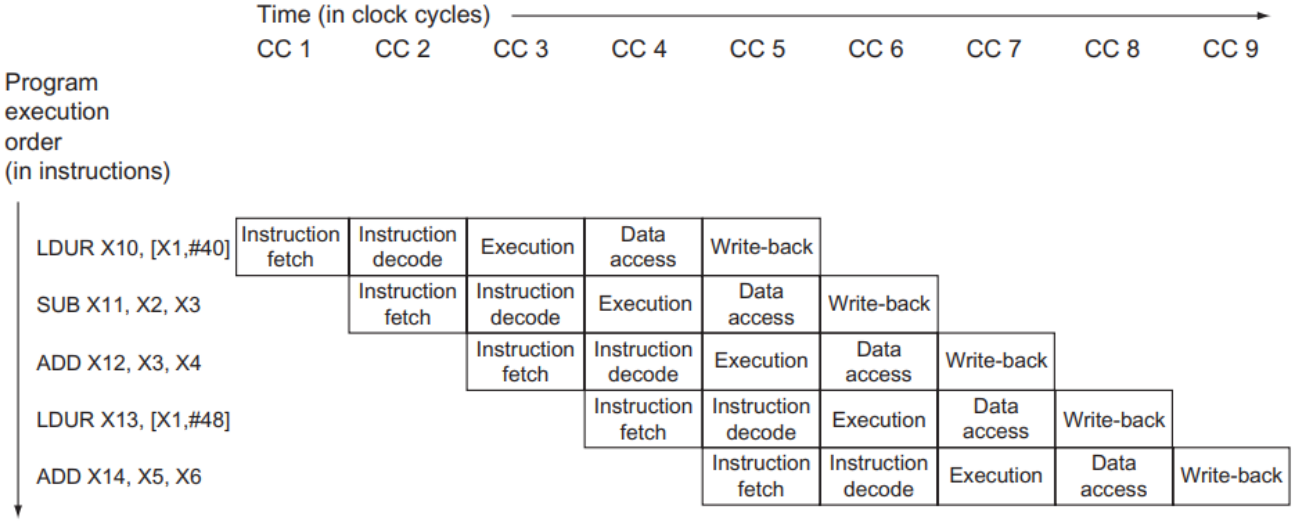
Ejecución sin pipeline: podemos ver que la instrucción en el micro sin pipeline, tiene que hacer todas las etapas en cada instrucción. Como el tiempo de ciclo es uno solo y es transversal a todo el procesador, siempre el periodo del clock va a ser igual a la instrucción que tome mayor tiempo o la que tenga mayor latencia, entonces la que demore más en ejecutarse, y la que demora más es la LDUR que usa todas las etapas del micro, por ende esta marca el tiempo del ciclo. Entonces, en total, en este micro sin pipeline, ejecutar estas tres instrucciones LDUR, nos llevó 2400 pico segundo.



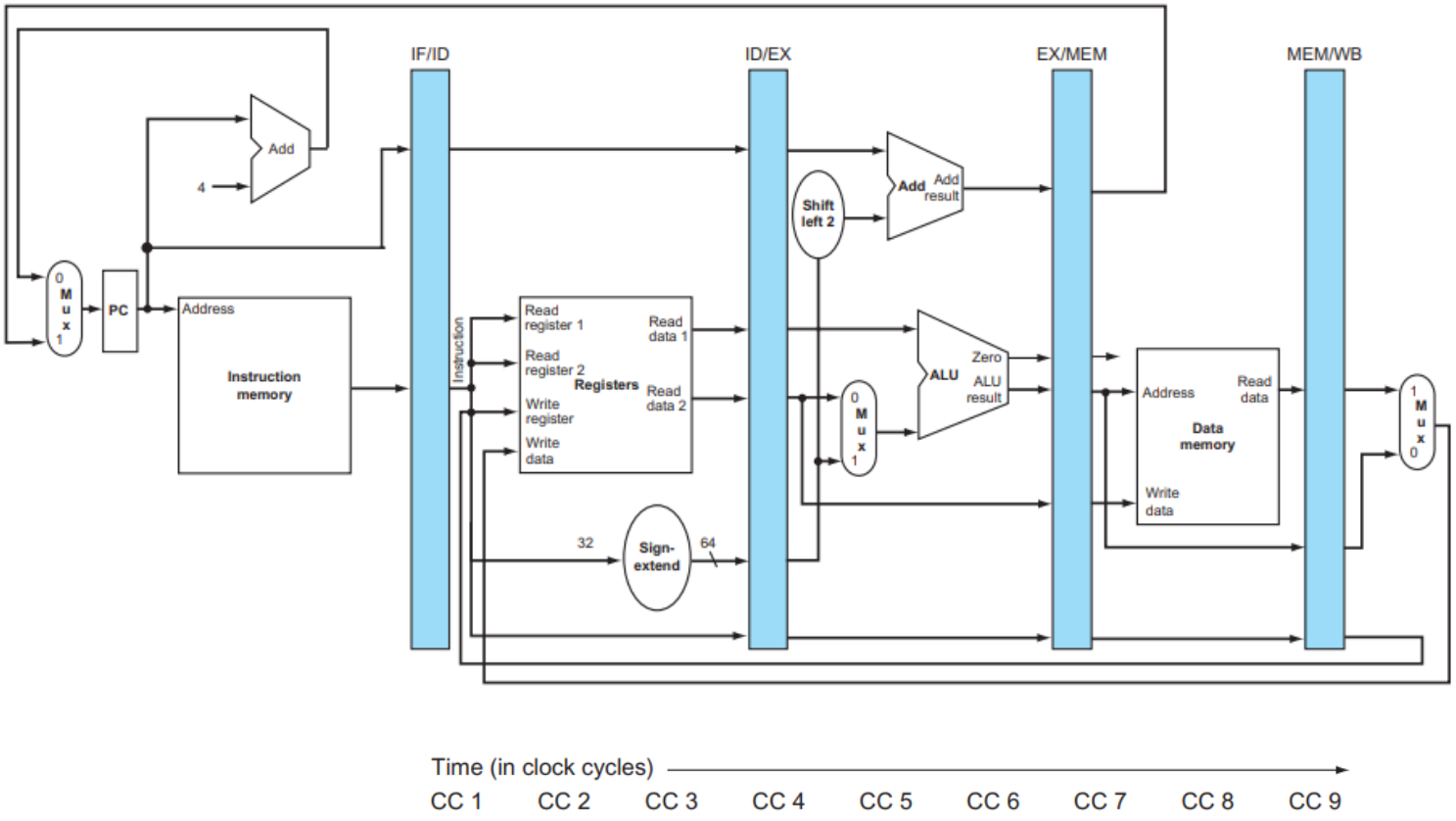
Ejecución con pipeline: ahora, en el micro con pipeline, en una situación ideal sin colisiones, vamos a configurar un clock en un tiempo. El periodo del clock ya no será el tiempo total de resolver la instrucción de mayor latencia, sino que ahora es el tiempo del bloque que lleva la mayor latencia. Cada etapa lleva un cierto tiempo en ejecutarse, pero como ahora el clock afecta a cada etapa por separado, el periodo del clock tiene que respetar cual es la etapa que consume mayor tiempo, y esa es la que va a determinar la máxima frecuencia. Ahora el ciclo de clock es 200 ps, y aun con la pre carga de esas primeras etapas, el tiempo de ejecución es mucho más chico, ya que terminamos a los 1400 ps.

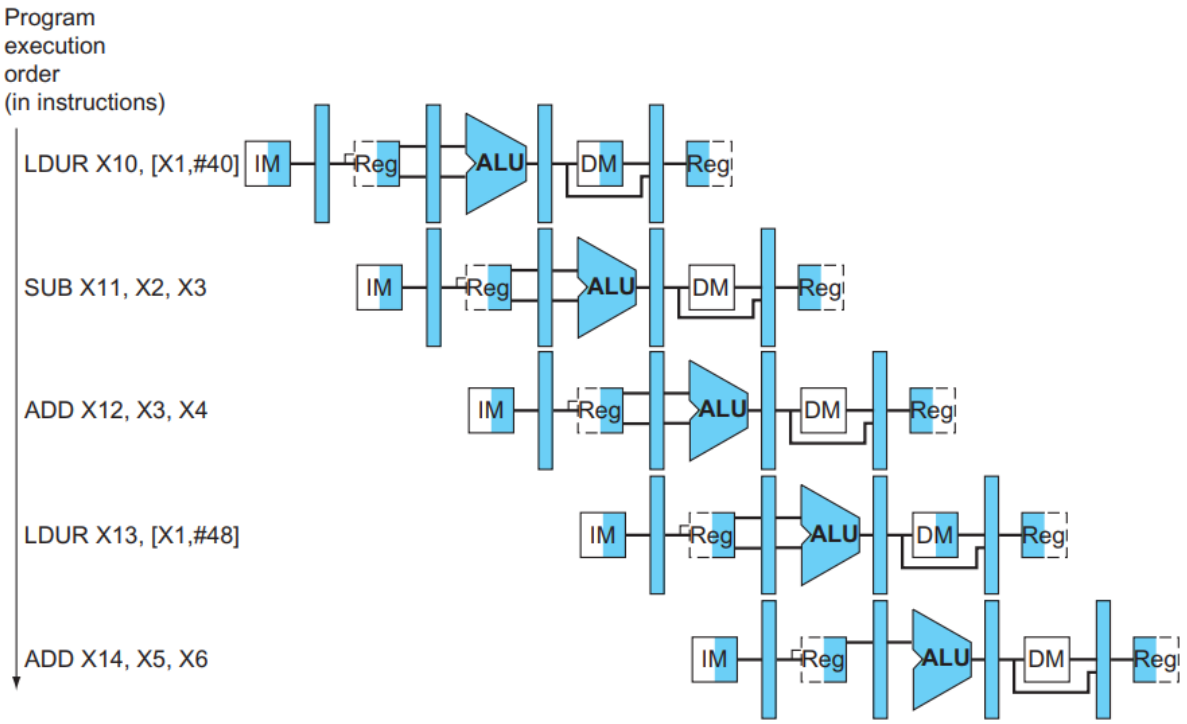


Representaciones del ciclo: a continuación vemos otro ejemplo que utiliza el pipeline, en donde realizamos diferentes representaciones del ciclo de clock:



ADD X14, X5, X6	LDUR X13, [X1,48]	ADD X12, X3, X4	SUB X11, X2, X3	LDUR X10, [X1,40]
Instruction fetch	Instruction decode	Execution	Memory	Write-back





_ En esta última representación podemos ver como sale el dato de la ALU y cómo podemos obtener los valores o datos de la memoria (DM) gracias a esa línea que está por debajo.

Conceptos

Cuello de botella o congestión: seria aquella etapa que demora todo y obliga a que el clock sea más lento.

Ciclo o periodo de clock: si usamos pipeline está definido por la etapa de más larga duración, y si no usamos pipeline está definido por la suma de las etapas de una instrucción.

Latencia o tiempo de ejecución de una instrucción: en el micro de un ciclo el tiempo va a ir cambiando en cada una de las etapas, y con pipeline, va a ser el número de etapas por el periodo de clock.

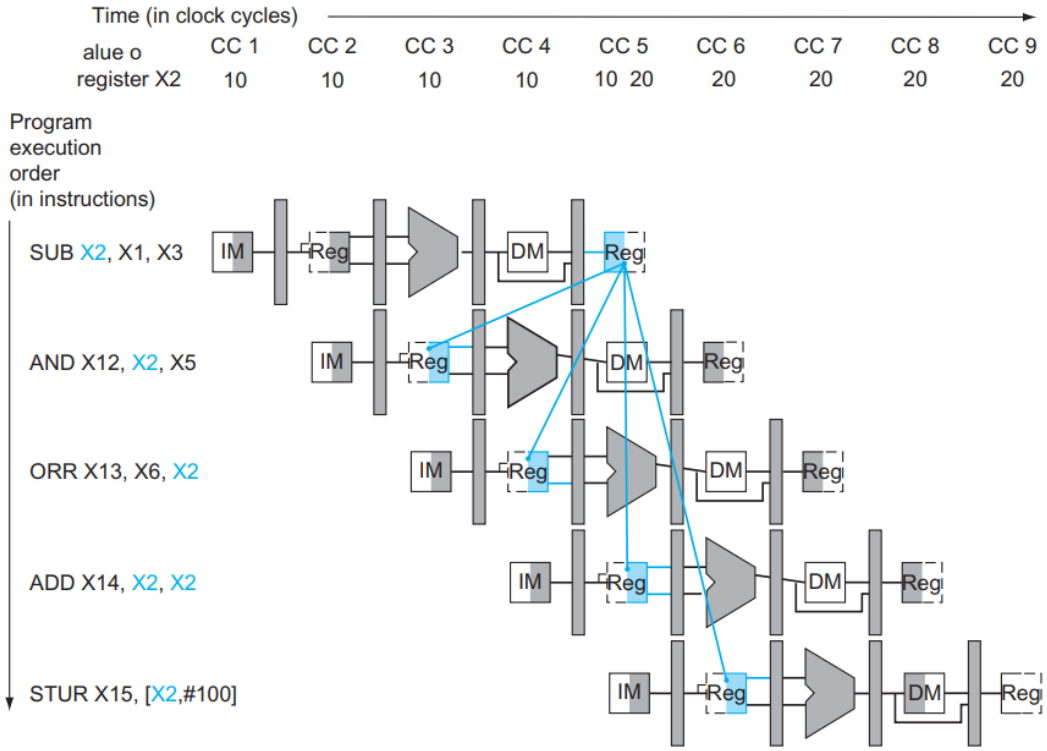
_ Entonces, no hay que confundir periodo de clock con tiempo de ejecución entre instrucciones, ya que una vez que precargamos el pipeline, el tiempo que tenemos entre que se terminó de resolver una instrucción y la siguiente, es un tiempo T. Pero el tiempo de ejecutar una instrucción o la latencia de una instrucción, serian 5T, entonces no es lo mismo el tiempo de ejecutar una instrucción, que es el tiempo de principio a fin, que el tiempo entre instrucciones, en donde por cada ciclo de clock estamos terminando de resolver una instrucción.

Dependencia de datos

_ Al estar en un ciclo, empezamos y terminamos de ejecutar una instrucción antes de empezar con la otra, pero ahora con el pipeline empiezan a surgir problemas con la dependencia de datos, y es que de alguna manera tenemos que asegurarnos de estar usando los datos correctos sin ningún hazard en el medio. Entonces, cuando hablamos de dependencia, hablamos de datos que están relacionados, por ejemplo, instrucciones que están usando el mismo registro. Puede ser que haya una dependencia de datos que no genere hazard, porque puede ser que están demasiado lejos los datos entre sí, puede ser que dos instrucciones estén escribiendo un resultado en un registro, por lo que más allá de que se accede al mismo registro, no hay colisión.

_ A continuación, vemos un ejemplo de un código, y se nos muestra a medida que avanzan los ciclos de clock, que va pasando con los datos. Primero tenemos un SUB, que escribe el resultado en X2, lo siguiente es un AND que usa X2 cuando está en la etapa de decode, en donde su valor va a estar actualizado. En el SUB, el valor a nivel de registro, se va a actualizar en la etapa de write back. En este caso, en el ciclo 5 el SUB va a escribir el valor de X2 en el registro, pero el código siguiente lo necesita recién en el ciclo 3, y acá tenemos un problema. Despues tenemos un ORR, que utiliza X2 y lo va a necesitar de nuevo en la etapa de decode, y nuevamente tenemos un problema, porque esta etapa está en el ciclo 4, y el resultado se escribe en el ciclo 5. Luego tenemos un ADD que usa dos veces el registro X2, o sea tenemos otro problema de dependencia, pero no se generaría en principios un hazard, porque en el clock 5 que es cuando primero se escribe el resultado y despues se lee, no hay conflicto. Si bien hay una dependencia de datos pero no hay problemas. Por último, el STUR que utiliza el X2 para calcular la dirección de acceso a memoria, si bien se hace el Instruction fetch en el

ciclo 5 cuando se está escribiendo el resultado, para cuando STUR llega a decode, el resultado ya está escrito, por lo que no se genera problema.



_ Algunas de las técnicas para solucionar estos problemas de dependencias son:

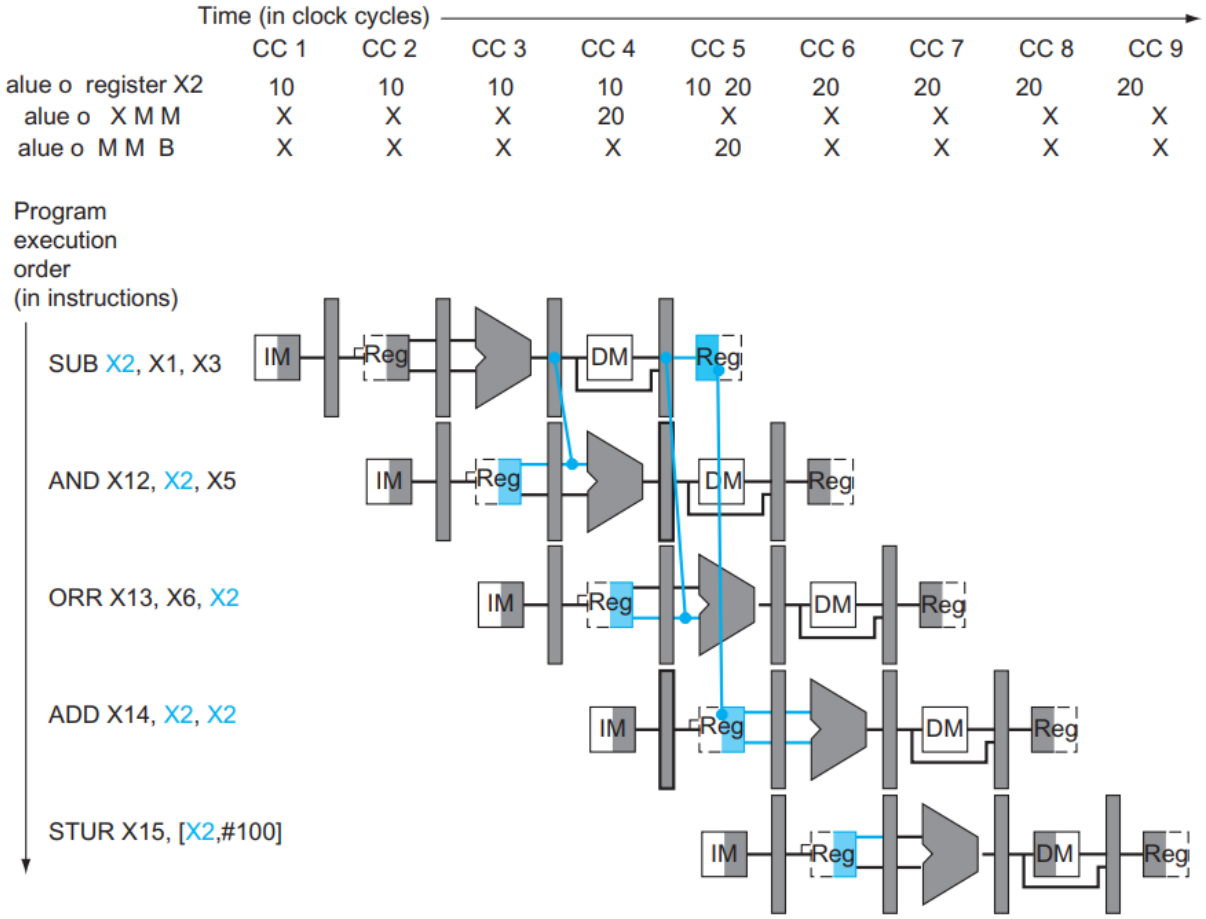
- Utilizar otro registro aplicando register renaming.
- Reacomodar las instrucciones de forma tal que no haya problema.
- Utilizar en el medio instrucciones NOP, de no operaciones, que no hacen nada, y simplemente para dar espacio de tiempo entre la ejecución de una y otra. NOP = XZR, XZR, XZR

_ Y despues tenemos técnicas que son propias del procesador, y estas son:

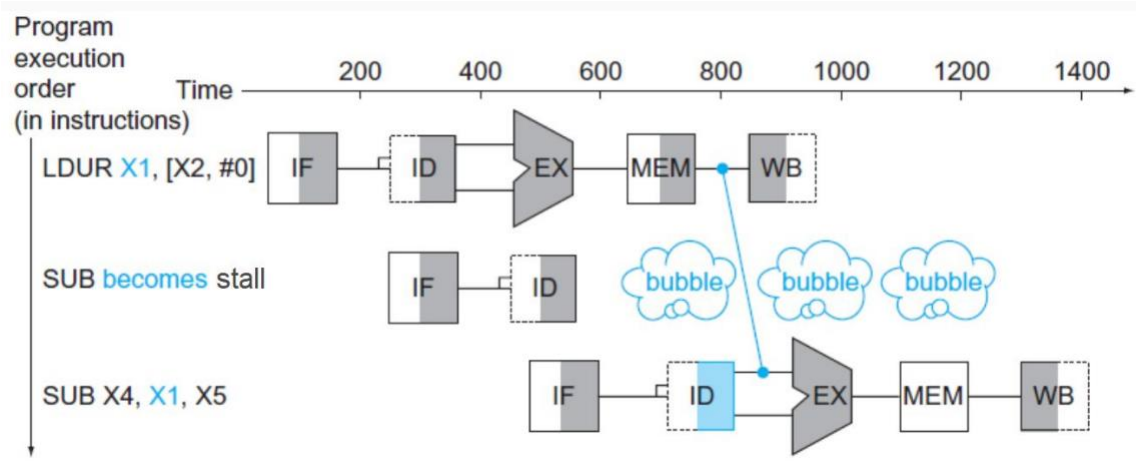
Stall: es muy similar al recurso de usar la instrucción NOP, solamente que al stall automáticamente lo pone el micro, entonces, siguiendo el grafico anterior, se ejecutó el SUB y al querer ejecutar el AND, el micro se da cuenta de que tiene un problema porque necesita el resultado de X2, entonces convierte a esa AND en un stall para que no haga nada, agregando nubecitas, y cuando tenga el resultado, ahora si puede continuar con la ejecución de esa instrucción.

Forwarding: no tenemos infinitos canales de forwarding, por ende uno de los canales que tendríamos seria cuando estamos ejecutando el SUB, y una vez que se realizó la resta tenemos el resultado o el valor que queremos usar, sin haberlo guardado en X2, entonces directamente cuando vamos a hacer el AND, a X2 no lo vamos a sacar del registro, sino que lo vamos a sacar del pipeline. Por ende, hacemos el forwarding, y directamente el valor con el que opera X2 es el resultado del SUB que tomamos de la salida de la ALU. Ahora, en el ciclo siguiente se empezó a ejecutar una ORR, que cuando llega a la etapa de execute, necesita a X2, y como el SUB que escribe el valor en X2 todavía no llego a la etapa de write back, entonces a ese X2 lo vamos a tomar de

nuevo del resultado de la ALU, pero no podemos estando en el ciclo 5 tomar algo que paso en el ciclo 3, y para eso nos sirve el cablecito bajo el DM. El forwarding puede hacerse en cualquiera de las dos entradas a la ALU. En el caso del ADD no hay un forwarding, y en el caso de STUR tampoco ya que se ejecuta despues.



Forwarding stall: cuando se aplica un stall a una instrucción y esta no se puede ejecutar, entonces se transforma en nada, y a esa nada lo representamos con una nube, y al forwarding se lo representa con líneas como en el gráfico anterior. Pero no existe un micro con solo forwarding, ya que o es solo stall o forwarding stall. La unión forwarding stall surge por culpa de las instrucciones load, ya que en este caso no tenemos el resultado o valor de algo en la salida de la etapa de execute, sino que recién lo tenemos despues de haber accedido a la memoria en la etapa de memori en la salida del MEM. Entonces, como se nos presenta en el gráfico, en donde tenemos un load a X1, y la instrucción siguiente utiliza X1, no tenemos otra manera de solucionar el hazard de datos, más que poner un stall en el medio o una instrucción NOP. Entonces, aplicando el stall, cuando llegue al SUB, y se da cuenta que en la etapa de execute no puede usar X1, lo transforma en un stall, y lo que hacemos es representar como si todo se corriera un tiempo y la instrucción hace un fetch un ciclo despues. Entonces el SUB se quiso empezar a ejecutar y cuando llego a la etapa de execute, como había un hazard de datos, se convirtió en stall, y recién en el siguiente ciclo de clock se pudo hacer el execute haciendo un forwarding desde la salida del memori hasta la entrada de execute.



Riesgos del pipeline (hazards)

_ Un hazard es una situacion que se produce cuando la ejecución de una siguiente instrucción, no se puede llevar a cabo en el ciclo siguiente, hasta que finalice la instrucción anterior, con la que tiene conflicto justamente. Esto es un problema que tiene el pipeline en ciertas ocasiones.

_ Tenemos tres tipos de hazards:

Hazard estructural: ocurre cuando una instrucción quiere leer un registro mientras un instrucción anterior esta accediendo a la memoria en la etapa de MEM. Esto es asi porque el hardware no admite una cierta combinación de instrucciones durante el mismo ciclo. Podemos solucionar esto con instrucciones NOP o stalls.

Hazard de datos: ocurre cuando una siguiente instrucción necesita acceder a un registro que se esta trabajando en la instrucción anterior. Entonces, lo que sucede es que el pipeline se debe bloquear debido a que un paso de ejecución debe esperar a que otro paso sea completado. Tenemos algunas soluciones:

- Una solución seria utilizar stalls, en donde las etapas que no se ejecutan, se convierten en burbujas, y luego se ejecutan en el ciclo siguiente.
- Otra solución seria agregar instrucciones NOP, hasta lograr que la etapa de decode de la instrucción dependiente, este alineada con la etapa de write back de la instrucción anterior.
- Forwarding-stall: esta es la principal solución y más eficiente, en donde se predice o anticipa el contenido de un registro, sin que sea necesario esperar a que la instrucción se complete y se escriba en memoria. Aclaramos que lo anterior es forwarding, pero seguimos teniendo stall, ya que en el caso de instrucciones LDUR, no tendremos otra opciones que perder 1 ciclo de clock ya que la instrucción siguiente necesita acceder a la memoria.

Hazard de control: ocurre cuando las instrucciones de salto deben calcular la dirección hacia donde saltara, y esto se hace en la etapa de MEM, por lo tanto, las siguientes instrucciones se harán en flush hasta que coincida la próxima etapa IF con WB. Entendiendo como flush a que se limpia el pipeline, es decir, que a las instrucciones precargadas que están por llegar a la etapa de memori, las levantamos y limpiamos. Cuando hay un salto condicional, y como el procesador cargo instrucciones que al final no se ejecutaron, tuvo que limpiarlas. Entonces, esto surge de la necesidad de tomar una decisión basada en los resultados de una instrucción mientras las otras se están ejecutando. Para solucionar este riesgo tenemos dos posibles formas:

- Forwarding Stall o bloqueo: consta de una instrucción de salto o branch, donde se debe empezar a buscar la instrucción que sigue a un salto justo en el siguiente ciclo de reloj.
- Predicción: se usa una branch prediction, para predecir que saltos se toman y cuáles no.

Latencia: número de etapas en un pipeline o el número de etapas entre dos instrucciones durante la ejecución.

Predicción de saltos

_ Tenemos un problema de hazard de control cuando el micro tiene que tomar una decisión, y justamente tiene que tomar una decisión cuando tiene que ejecutar un salto. Hay algunas cosas que hacer cuando el salto se toma, y otras cosas si no se toma, particularmente ver con que instrucciones vamos a seguir. Con el pipeline vamos precargando cuales son las instrucciones que vienen posteriormente, entonces por esa razón se vuelve más dinámica la ejecución. Ahora, si precargamos incorrectamente, porque tomamos un salto a instrucciones que no vamos a usar, o cuando configuramos que se ejecuten tales instrucciones al tomar un salto, y dicho salto no se toma, entonces ahí tenemos un problema de control, por una mala decisión respecto de ese salto.

_ Existen algunas técnicas para la predicción de saltos, si bien estas no resuelven el problema de hazard de control, algunas son mas eficientes para el manejo de saltos:

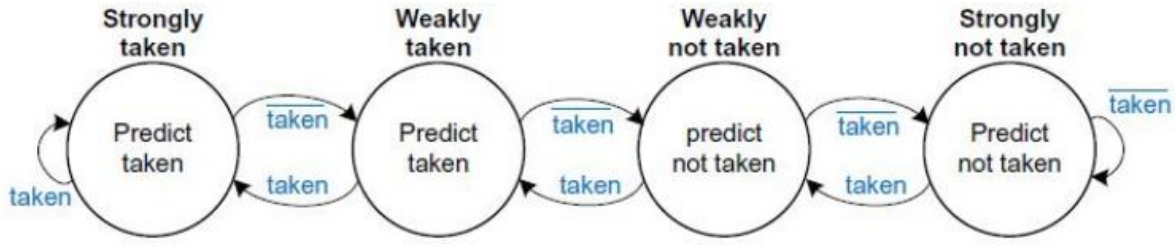
Técnicas estáticas: son las técnicas más básicas, en donde asumimos que el salto siempre se toma o asumimos que el salto siempre no se toma, y trabajamos respecto a eso. En el caso del micro LEGv8, este trabaja con una técnica estática, en donde siempre asume que no va a saltar, es decir, es un Not Taken, con lo cual se siguen ingresando al pipeline las instrucciones siguientes en el orden de ejecución lineal. El procesador siempre toma la misma decisión por defecto.

- Not taken (NT): por defecto se asume que el salto no se va a tomar y se carga la siguiente instrucción, en caso que el salto deba tomarse, se hace flush y se realiza el fetch de la instrucción correcta. La penalidad es de 3 ciclos de clock.
- Taken (T) o Always Taken: por defecto el procesador asume que el salto se toma y carga la instrucción que indica el salto. Esto en un procesador como el

que venimos estudiando tiene una penalidad de 3 clocks siempre (la actualización del PC se realiza en la etapa de memory).

Técnicas dinámicas: básicamente lo que se hace es decidir si saltar o no, considerando como un historial de saltos. Entonces, dependiendo si los anteriores se fueron tomando o no, decidimos que siempre se va a tomar o no, entonces esto es dinámico para cada salto que aparezca en el código. El procesador decide si saltar o no considerando el historial de saltos.

- Predictor de dos bits: básicamente posee dos bits para codificar si el salto se tiene que tomar o no. Si tuviéramos un predictor de un solo bit, podríamos determinar de que si es 0, podemos predecir que el salto se toma, y si es 1 que no se toma, o viceversa. Pero al tener dos bits, nos provee de cuatro combinaciones posibles (00, 01, 10 , 11), por lo tanto, nos permite generar como una pequeña máquina de estados que representa estas combinaciones. Entonces, en lugar de un taken o not taken, vamos a dividir a estos en dos para asi obtener un taken débil y un taken fuerte, un not taken débil y un not taken fuerte, y lo que se hace es dejar que el predictor, a medida que se va ejecutando un código, vaya solo eligiendo en qué estado le conviene estar para este tipo de salto. El estado en el que le conviene estar al predictor, lo va a saber en base a las decisiones que tomo anteriormente. Este tipo de predictor es útil en los ciclos for.



_ Analizando el grafico tenemos que:

1. Empezamos en el estado 0 que fuertemente predice que el salto se tiene que tomar (predict taken). Al llegar a una instrucción de salto, el salto se toma, y quedamos en el mismo estado. Si el salto no se toma cambiamos de estado.
2. Cuando el salto no se toma, aun asi cambiamos a un estado que sigue prediciendo taken, pero de forma débil, en donde si viene una instrucción de salto y este se toma, de nuevo se predice el salto y se vuelve al estado de fuertemente predice el salto. Para salir de estos estados, necesitamos que dos veces no se tome el salto.
3. Cuando no se toma el salto, en dos oportunidades seguidas, pasamos al estado que débilmente no predice el salto, en donde si el salto se toma, regresamos al estado que débilmente predice el salto, pero si nuevamente no se toma el salto pasa al estado siguiente.
4. Entonces, si el salto no se toma por tercera vez consecutiva, llegamos al estado que fuertemente no predice el salto. Si el salto no se toma, permanecemos en

este estado, pero si el salto se toma vuelve al estado anterior. Si el salto se toma tres veces consecutivas volvemos al estado que fuertemente predice el salto.

_ Esta estrategia consiste es fijarse en la dirección de las instrucciones para ver si se trata de un salto que fue tomado la última vez que la instrucción se ejecutó, y en ese caso comenzar a buscar la siguiente instrucción en el mismo sitio que se hizo la vez anterior. Una manera de realizar esta estrategia es mediante la utilización de un:

- Búfer de predicción de salto: esto es una memoria pequeña indexada con la parte menos significativa de la dirección de la instrucción de salto. Esta memoria contiene un bit en cada entrada que dice si el salto ha sido recientemente tomado o no. Además, el búfer puede ser implementado con dos bits.

Técnicas de mejora de rendimiento

Introducción

_ Para ejecutar ciertas instrucciones en el procesador de un solo ciclo, tenemos que iniciar un ciclo de clock, y dejar que ese ciclo dure todo el tiempo requerido hasta que se complete la ejecución de la instrucción más lenta o de mayor latencia, y esta es la que nos determinaba cual era la latencia del procesador de un solo ciclo. Como vimos, la ejecución de las instrucciones está dividida en 5 etapas, pero pueden ser más. Cuando pasamos de un micro sin pipeline a uno con pipeline, no tocamos el hardware ni las unidades funcionales, pero lo que hacemos entre una etapa y la otra es poner registros, que no son más que un flip-flop D, que al llegar un ciclo de clock, se copia lo que tiene en la entrada del registro a la salida del mismo. Pero nosotros a eso no lo consideramos, ya que trabajamos con cuestiones ideales, en donde sabemos que cada etapa tiene distintos tiempos de ejecución, entonces ajustamos el ciclo de clock del micro con pipeline a la que es la etapa más lenta, en donde esta etapa si o si se debe terminar de ejecutar y es la que va a marcar el ciclo de clock, por ende la ejecución de una única instrucción nos lleva 5 veces ese tiempo. Si tuviésemos 6 etapas de pipeline, seria 6 veces el tiempo de la etapa más lenta. Pero es importante saber que por cada partición que hacemos, estamos poniendo una etapa más de registro, y esa etapa de registro, si bien tiene latencias de pico segundos, a la larga, termina siendo un tiempo a considerar.

Retardo de propagación a través de la lógica combinacional: esto es, ni más ni menos, que la latencia del procesador de un ciclo. El retardo de la propagación es lo que tarda la instrucción desde que se puso el PC apuntando a la instrucción que queríamos hacer fetch, la sacamos y siguió todo su flujo hasta que se terminó de ejecutar.

Paralelismo a nivel de instrucciones (FLP)

_ Existen dos estrategias para incrementar la cantidad de potencial del FLP.

1)_ Consiste en aumentar la profundidad del pipeline para solapar la ejecución de más instrucciones.

2)_ Consiste en replicar los componentes externos del computador para poder ejecutar múltiples instrucciones dentro de cada etapa de segmentación. A esta técnica se la conoce como “multiple issue”.

Multiple issue

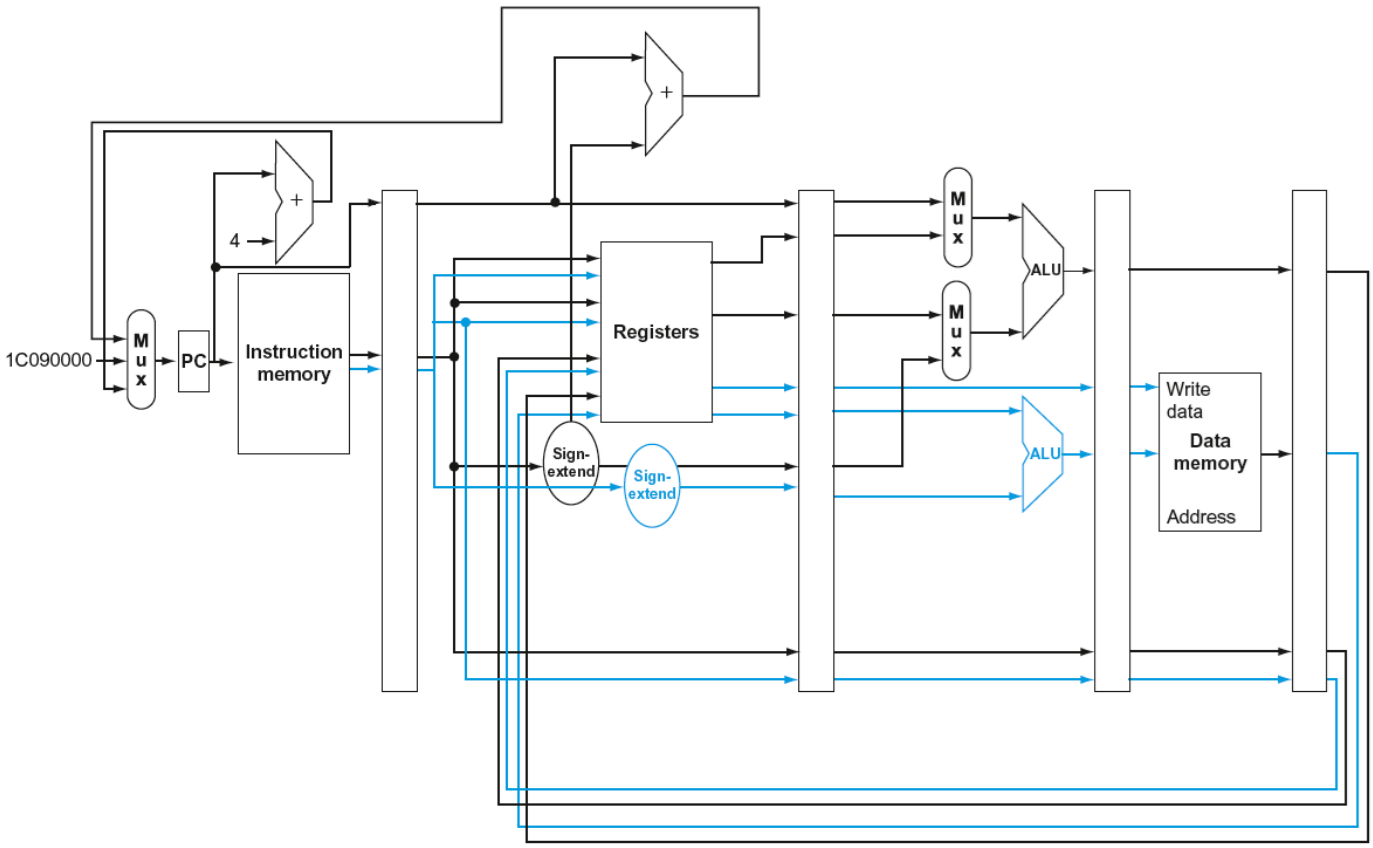
_ Si bien en el micro con pipeline tenemos 5 instrucciones, que se ejecutan en simultaneo, pero en distintas etapas. En el caso de los procesadores múltiple issue, estos permiten que varias instrucciones se empiecen a ejecutar al mismo tiempo, en donde todo el ciclo de ejecución, y paso por las etapas, se da de manera conjunta entre todas las N instrucciones. Hay dos formas de implementar un procesador de ejecución multiple:

Static Multiple Issue

_ Este es un procesador que puede ejecutar varias instrucciones en simultáneo, pero estas instrucciones deben ser "empaquetadas" por el compilador (Issue Packet). Hablamos de empaquetadas porque, a cada conjunto de instrucciones que vamos a ejecutar de manera conjunta, las vamos a llamar Issue Packet, es decir, el paquete de instrucciones que entra a ejecutarse. Se lo llama estático porque el micro, aparte de tener la posibilidad de ejecutar varias instrucciones a la vez, no puede saber que está ejecutando, ya que no puede decidir a quién ejecutar primero, por lo que solo puede ejecutar estáticamente lo que nosotros le pasemos como código. Entonces toda la responsabilidad de armar los paquetes es del compilador, o en nuestro caso en particular, nuestra.

- En algunos casos, se restringe qué tipo de instrucciones pueden ejecutarse en simultáneo.
- La mayoría de los procesadores relegan la responsabilidad de manejar ciertos hazard de datos y control al compilador. Ya sea para prevenir los hazards o para reducirlos.
- Esto se realiza en tiempo de compilación.

LEGv8 Two-Issue processor: en este caso, vemos un hardware two issue, más simple, que permite ejecutar dos instrucciones a la vez. A continuación, lo visualizamos en el gráfico, en donde la base es muy similar al de un micro con pipeline normal de un issue, pero ahora viendo que se duplican las direcciones que entran al registro y los dos datos que entran en ese registro. Se duplica el sign-extend, ya que en este caso, un issue solo puede hacer operaciones con la ALU de tipo aritméticas-lógicas y de saltos, y al otro issue se lo deja exclusivamente para instrucciones que tengan que ver con accesos a memoria.



_ Algo a tener en cuenta es que:

- En cada issue packet, debe haber una instrucción tipo R o branch, y una instrucción de acceso a memoria.
- Ejecutar dos instrucciones por ciclo, requiere hacer fetch y decode de instrucciones de 64 bits (el PC se incrementa de a 8).
- Si uno de los issue no puede utilizarse, se lo debe acompañar con un NOP.

_ Vamos a ordenar las instrucciones en el pipe de la siguiente manera:

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Loop unrolling: es una técnica aplicada por el compilador para los procesadores multiple issue, con el objetivo de conseguir un mayor rendimiento en los lazos. Consiste en realizar múltiples copias del cuerpo del lazo. Despues del desarrollo, se dispone de más ILP (Instruction-Level Parallelism) para poder solapar la ejecución de instrucciones que pertenecen a diferentes iteraciones.

Dynamic Multiple Issue

_ Esto se realiza durante la ejecución, y son conocidos como super escalares. En estos procesadores, las instrucciones se lanzan a ejecutar en orden y en un determinado ciclo de reloj. El procesador decide si se puede ejecutar una o más instrucciones nuevas o ninguna.

Análisis de las dependencias

_ Como vimos anteriormente, las dependencias son un propiedad del programa. El hecho de que esta dependencia de datos se detecte como hazard y si genera o no un stall, es propiedad de la organización del pipeline. Entonces, la dependencia de datos implica:

- La posibilidad de un hazard.
- El orden en que se debe calcular el resultado.
- Un límite superior en cuánto se puede explotar el paralelismo.

_ Una dependencia de datos puede de alguna manera solucionarse:

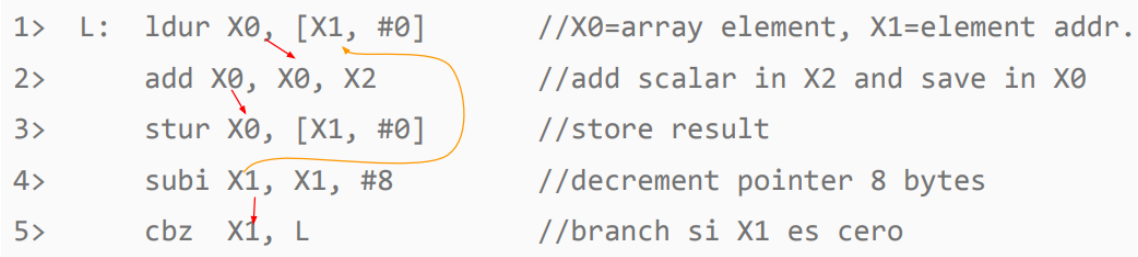
- Manteniendo la dependencia pero evitando el hazard con las técnicas anteriores.
- Eliminando la dependencia al modificar el código.

_ Nosotros en ningún momento estamos hablando de dependencia de datos de memoria. Los datos pueden fluir entre instrucciones mediante memoria o registros. Pero nosotros siempre trabajamos a nivel de registro, pero también podemos tener un problema si hay dos instrucciones que están accediendo a la misma posición de memoria.

- Dependencia entre registros: relativamente sencillo de detectar, es transparente verlo desde las instrucciones.
- Dependencia de memoria: son difícil de detectar y la razón por la que no las tenemos en cuenta. Dos instrucciones pueden referirse a la misma posición pero parecer diferente, por ejemplo: {STUR x0, [x4,#100]; LDUR x1, [x6,#20]}. Además, la misma instrucción ejecutada en distintas partes del código puede apuntar a posiciones distintas.

Dependencia real de datos

_ Estas básicamente con las mismas que las que mencionamos anteriormente, en donde la instrucción i es dependiente en datos con la instrucción j cuando la instrucción i produce un resultado que debe ser utilizado por la instrucción j. En el siguiente ejemplo, LDUR escribe en X0 y luego el ADD lo usa, el ADD escribe el resultado en X0 y el STUR lo usa, y finalmente el SUBI escribe en X1 y el CBZ lo usa para decidir si salta o no. Las dependencias condicionales de datos que vemos son de X1 de la instrucción 4 con la instrucción 1, 3 y 4 cuando el salto se tome en una segunda vuelta.

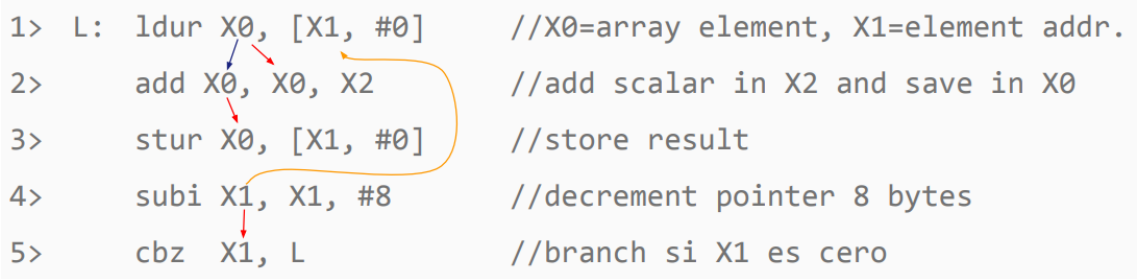


Dependencias de nombre

_ Estas se producen cuando dos instrucciones usan el mismo registro o posición de memoria, pero no hay flujo real de datos. También se les llama:

Dependencia de salida: ocurre cuando las instrucciones i y j escriben la misma posición de memoria o registro. Si la instrucción i precede a la instrucción j en el orden del programa, se debe preservar el orden original para asegurar que el valor final se corresponda con el valor de j. Simplemente, el resultado se tiene que escribir en el registro que tiene que ser y no en otro porque hicimos un intercambio de las instrucciones y modificamos la dependencia de salida.

_ Podemos observar en este ejemplo, que LDUR escribe en X0 y ADD también, por ende hay una dependencia de salida. Suponiendo que queremos empaquetar las instrucciones 1 y 2, y ejecutarlas juntas, al ser una de acceso a memoria y la otra aritmética, se pueden ejecutar en two issue, pero al hacer el write back, las dos instrucciones van a competir por cuál de las dos va a escribir el resultado, en ese registro en particular, pero no podemos asegurar cual se las dos va a escribir en X0 finalmente. Entonces, esto genera un problema cuando armamos un issue packet.



_ Una técnica que podemos usar para eliminar este tipo de dependencia es:

- Register renaming: dado que las dependencias de nombre no son dependencias reales. Las instrucciones se pueden ejecutar simultáneamente o en otro orden, y si el nombre (del registro o de la posición de memoria) se cambia para no generar conflictos.

Dependencia de datos condicional

_ Estas son dependencias de datos que no sabemos si se van a dar porque hay un salto condicional en el medio. Existiría una dependencia si el salto se toma o existiría una dependencia si el salto no se toma, por ende, en rigor no estamos seguro si la dependencia se va a dar o no.

Hazard de datos

_ Debido a lo anterior, ahora podemos definir al problema de hazard de datos como la dependencia de datos o de nombre entre instrucciones que se ejecutan lo suficientemente cerca en tiempo, de forma en que al superponerse en la ejecución se modifique el orden de acceso a los operandos involucrados en la dependencia.

_ Entonces, ahora aparece dos tipos de hazard de datos:

RAW: cuando j intenta leer un dato antes de que i lo escriba, j obtiene el valor desactualizado (dependencia real de datos). Estos son los que tenemos siempre y vimos anteriormente. En los siguiente ejemplos, en ambos escribimos X0 y luego usamos el valor del mismo en la siguiente instrucción:

```
add X0, X1, X2          |       add X0, X1, X2
add X3, X4, X0          |       stur X3, [X0, #0]
```

WAW: cuando j intenta escribir un operando antes de que lo escriba i. Las escrituras se terminan realizando en el orden incorrecto, dejando como valor final el de i, en lugar de j (dependencia de salida). En este ejemplo, ambos tienen registro destino. En el caso del LDUR, si fuese un STUR, el hazard seria RAW porque ya que no estamos escribiendo en un registro sino que STUR necesitaría del valor de X0 de ADD.

```
add X0, X1, X2          |       add X0, X1, X2
add X0, X3, X4          |       ldur X0, [X3, #0]
```

Dependencia condicional (de control)

_ Estas dependencias tienen que ver con, a quien podemos agrupar con quien. Una dependencia condicional determina el ordenamiento de una instrucción cualquiera respecto a una de salto. De forma en que dicha instrucción se ejecuta en el correcto orden de programa y sólo cuando debe ser.

Restricciones impuestas por la dependencia condicional: dado el siguiente código de ejemplo, no podemos agrupar SUB con CBNZ ya que X2 en la instrucción 1 depende en datos de la instrucción 2. Además, siempre que haya un salto, debemos ver si es correcto que se ejecute tal instrucción con tal salto, como en los casos de la instrucción 2 con la 3, en donde no sabemos cuándo se toma el CB, también la instrucción 2 con la 4, en la que nos llevamos en el medio la instrucción 3, y también no es conveniente empaquetar aquellas instrucciones que están en secciones diferentes como la 3 y la 4 que está dentro de la función L1.

```
1>        sub x2,x3,x4
2>        cbnz x2,L1
3>        ldur x1, [x2, #0]
4>   L1:  add x3, x2, x5
```

- Una instrucción que tiene una dependencia condicional sobre un salto no puede moverse antes que el salto (o agruparse al mismo) de forma en que su ejecución no esté controlada por el salto.
- Una instrucción que no tiene una dependencia condicional sobre un salto no puede moverse después del salto de forma en que su ejecución esté controlada por el salto.