**Arquitectura de computadoras: Resumen 2do Parcial**

**Unidad 0.4: Arquitectura de una computadora (microarquitectura)**

Microarquitectura está integrada por:

* Registros inaccesibles
* Componentes internos
* Conexiones internas

Es lo que le da soporte a una arquitectura.

NO es la ISA (arquitectura de programación de una computadora)

Microarquitectura RISC-V (real es RISC-V2p):

Esta microarquitectura está integrada por componentes de distinto tipo: secuenciales y combinacionales. De estos últimos se pueden mencionar sumadores, multiplexores, compuertas lógicas como AND, NOR, negadores, etc, todos conectados entre sí por líneas de control y buses.

Entre los componentes internos más importantes de esta microarquitectura se encuentran:

1. **Registro Latch**: circuito secuencial con clock que funciona por flanco ascendente. Posee una señal de reset que “limpia” el valor inicial del circuito cuando recién se enciende. Posee también una señal de clock enable (CE), la que cuando está en 0 indica que el circuito no debe actualizar su valor interno con el valor que le llega por la entrada IN. En cambio, cuando está en 1 y a la vez se produce un flanco ascendente, deberá tomar el valor que posea en su entrada, almacenarlo internamente y dejarlo disponible en su salida OUT.
2. **Memoria de instrucciones**: circuito secuencial con clock que funciona por flanco ascendente. Contiene almacenadas todas las instrucciones a ejecutar por la CPU. Internamente cada instrucción se encuentra en un latch, y todos estos latch están conectados a un multiplexor el cual habilitará la salida de uno u otro dependiendo del valor de selección que le ingrese a través del “addr” (address, que indica la dirección de memoria a habilitar). El valor de salida del multiplexor se guardará en otro latch, cuya salida a su vez se habilitará cuando se produzca un flanco ascendente y la señal de CE (clock enable) esté en 1. La memoria de instrucciones también cuenta con una señal de reset.
3. **Contador de ciclo**: circuito secuencial con clock y señal de reset. Se comporta como un contador sincrónico de 2 bits en este caso (es decir, módulo 4). Genera una y otra vez la secuencia 00, 01, 10, 11. Dicha secuencia se utiliza para determinar la etapa o fase de ejecución de una instrucción, las cuales son:
   1. **Fetch** (búsqueda de instrucción – 00)
   2. **Decode/Execute** (se decodifica la instrucción, es decir, se ve qué es lo que se tiene que ejecutar, y se ejecuta – 01)
   3. **Memoria** (acceso a memoria para leer o escribir datos – 10)
   4. **WriteBack** (fase en la que se actualizan los registros RD y el PC – 11)
4. **Decoder o decodificador**: se utiliza para decodificar la instrucción a ejecutar. Es un circuito combinacional que tendrá tantas salidas como tipo de instrucciones a ejecutar existan (en RISC-V habrá una salida para LUI, ADD, etc). Según qué instrucción se deba ejecutar se pondrá en 1 únicamente la salida correspondiente a esa instrucción y las demás estarán en 0.
5. **Program Counter (PC)**: El PC es en sí un registro latch, es decir, un circuito secuencial. La salida del PC (pcout) ingresa en dos sumadores, uno que suma fijo PC + 4 y el otro que suma fijo un valor inmediato(PC + IMM) codificado en la instrucción actual. Esos dos valores se encuentran realimentados a la entrada del PC mediante un MUX que selecciona PC+IMM en el caso que sea una instrucción de Salto condicional efectivo (branch) o salto incondicional (JAL). Si NO es un salto entonces el siguiente valor del PC va a ser PC+4. En el caso que sea un salto relativo a registro (JALR) donde el valor del PC proviene de la suma del contenido de un registro más un offset, entonces su valor se calcula en la ALU cuando se ejecuta una instrucción JALR. En el caso de que sea PC+4 lo que tiene que ingresar, recordemos que es +4 porque cada instrucción ocupa 32 bits en memoria (o sea 4 bytes), y como se direcciona al byte, es decir, cada dirección de memoria apunta a un byte, por eso se le suma 4.
6. **Register File**: es el que contiene los 32 registros accesibles al programador de la arquitectura RISC-V. Cada registro estará conectado en su entrada a la salida de un demultiplexor, el cual tendrá una señal de selección rd para indicar cuál de esos 32 registros se actualizará con el data que ingrese por un datain, cuando el ce esté en 1. Dicho datain contiene entonces el valor nuevo a almacenar en alguno de los registros, valor que puede provenir de la ALU, la memoria, el PC+4 o PC + IMM. El demultiplexor así seleccionará alguno de los 32, o mejor dicho de 31 registros, ya que el registro x0 siempre tiene su valor en cero y no cambia. Por otra parte, todos los registros se encuentran conectados a su salida a dos multiplexores, los cuales se utilizan para seleccionar dos de ellos para utilizar como operandos.
7. **IMM (inmediatos)**: los inmediatos son aquellos valores “directos” que se utilizan en alguna operación sin pasar o almacenarse previamente en uno de los registros vistos. En RISC-V existen distintas formas de “acomodar” los bits de los valores inmediatos (tipo I, tipo U, tipo J, etc), y la forma a utilizar dependerá de qué instrucción se vaya a ejecutar. Tal es así que el decodificador que se utiliza para decodificar la instrucción se conecta a un encoder a la salida. La salida del encoder a su vez se conecta a la señal de selección de un multiplexor, el cual indica qué formato se debe tomar para “acomodar” los bits de dicho inmediato en función de la instrucción decodificada, teniendo en cuenta además que siempre son 32 bits donde se propaga el signo.
8. **Memoria de datos**: es aquella en donde se almacenan variables o donde se hallan mapeados los dispositivos de E/S. Es un circuito secuencial que posee un memio\_ce. La salida de la ALU va a tener la dirección a la que se quiere acceder (ya sea para leer, o sea LOAD, como para escribir, STORE), y dicha dirección está formada por un registro base más un offset.

En el caso de que queramos leer un dato de memoria (LW, LH, LB, LHU, LBU), el addr como se dijo es el valor que ingresa desde la salida de la ALU con la posición de memoria a la que se quiere acceder. El valor que esté en esa dirección de memoria quedará disponible a la salida dataout. Y según de qué tipo de lectura se trate, el valor de dataout pasará por un propagador de signo. Por ej, si es LW (load Word), ya son 32 bits así que no se hace nada. Pero en el caso de LH (load half), son solo 16 bits, y para formar un valor de 32 bits se necesita propagar el signo, para lo cual se toma el valor del bit más significativo de esos 16 y se rellenan con ese valor los bits faltantes. En el caso de LHU y LBU, como es una lectura unsigned, solo se propagarán ceros.

En el caso de que se quiera almacenar un valor en memoria (SW, SH, SB), se utiliza la línea de write para indicar a la memoria que almacene el valor que ingresa por datain en la dirección apuntada por address. El valor que ingresa por datain sólo puede provenir del register file (seleccionado por rs2). Al igual que los loads, la dirección solo puede provenir de la ALU.

En el caso de write, representamos la escritura con un cero en la entrada write. Generalmente las memorias indican con cero la escritura y con uno la lectura por cuestiones de ruido eléctrico. Otros modelos (RISC-V2p) utilizan una línea de write por cada byte a escribir, permitiendo fácilmente implementar SW, SH y SB.

1. **ALU**: es enteramente combinacional. Cuenta con dos entradas que hacen referencia a los dos operandos a utilizar (a y b). En el caso de a puede ser cero o provenir de S1, y en el caso de b puede ser un inmediato o provenir de S2. Estos dos operandos luego se conectan en paralelo a todos los circuitos internos combinatorios que realizan al mismo tiempo las operaciones disponibles para ellos (suma, resta, AND bit a bit, OR bit a bit, XOR bit a bit, desplazamientos), y luego se seleccionará el resultado correspondiente para colocar a la salida. La ALU también contiene el CCR (code condition register) que es un como módulo interno separado para comparar a y b e indicar si son iguales, si a es menor a b, si a es mayor a b, etc (es decir, todos los condicionales conocidos en RISC-V).

Este procesador utiliza una secuencia de 4 ciclos por instrucción.

**● 00 Fetch:** se produce un pulso en el **ce** de la memoria de instrucciones para que la misma acceda a la instrucción. La dirección proviene del PC.

**● 01 Decode/Execute:** la instrucción se decodifica, por ende toda lógica que dependa del decoder se acomoda a la instrucción actual. Se buscan los operandos (sea inmediato o registros en el register file). La ALU es enteramente combinacional por ende resuelve todas las operaciones en paralelo.

**● 10 Memoria:** si la instrucción produce acceso a memoria (lw,sw,etc) en este ciclo la salida de la ALU posee la dirección, por ende se accede a memoria.

**● 11 Write Back:** Se actualiza el PC y se escribe en registro apuntado por **rd** en el caso que corresponda a la instrucción. Al actualizar el PC, el ciclo comienza de nuevo accediendo a la nueva instrucción.

Resumido:

Fetch → Memoria de instrucción

Decode → Decoder, IMM y RegFile

Execute → ALU

Mem → Memoria de datos

WriteBack → PC y RegFile RD

**Unidad 0.5: Eficiencia y Pipelining**

**Eficiencia**

Para saber qué tan eficiente es una computadora para cierta tarea, existen suites de prueba que sirven para ejecutar de manera exhaustiva (o abusiva) determinados algoritmos y programas, según el uso al que se le piense dar a esa computadora, para tener una idea de cómo se comporta la misma y su arquitectura bajo estrés. Dichas suites son conocidas como benchmark suites, las cuales no son gratuitas, y tienen como ventajas que se pueden ejecutar en diferentes sistemas para poder realizar comparativas entre uno y otro de los resultados, además de que tratan de ejecutar pruebas cercanas al uso real que se les piense dar a las computadoras.

Whetstone y Dhrystone, por otro lado, son benchmark clásicos en donde se ejecutan ciertas instrucciones que en porcentaje son representativas de la carga promedio de un sistema. No son real-world.

Si conocemos la arquitectura de una computadora podemos también realizar mediciones específicas para medir su eficiencia.

Una de estas mediciones es el **CPI o Clocks por Instrucción**, es decir, la cantidad promedio de pulsos de reloj que lleva ejecutar una instrucción promedio. En el caso de RISC-V, los pulsos de clock promedio por instrucción serán 4 clocks, excepto en LOAD que serán 5 clocks.

Una forma de calcular el CPI, es teniendo la cantidad de ciclos de cada tipo de instrucción junto con el porcentaje que integra dentro de un programa cada tipo de instrucción, realizar un promedio ponderado de los mismos:

EJ:

**CPI** = %Loads \* CiclosLoad + %Stores \* CiclosStores + %Registros \* CiclosRegistros + %Saltos \* CiclosSaltos

**CPI** = 0,2\*4 + 0,1\*5 + 0,45\*4 + 0,25\*3 = 3,85

Recordar que los porcentajes hay que pasarlos a su probabilidad (ej: 20% sería un 0,2)

Otra medición para medir la eficiencia de una computadora son las **millones de instrucciones por segundo o MIPS**. La misma se calcula con el CPI y con el dato de la cantidad de pulsos de clock que ejecuta la CPU en 1 segundo (que sería la frecuencia, que se mide en Mhz, teniendo en cuenta que 1000000 Hz es 1 Mhz).

**MIPS = frecuencia (en Mhz) / CPI**

Finalmente, otra medición que se utiliza según el programa que se esté ejecutando, en especial aquellos en los que se recorren elementos de un vector por ejemplo, son los **clocks en función de la cantidad de elementos.**

En estos casos, se nos da como dato la cantidad de clocks de cada instrucción, y se llega a una fórmula o ecuación que nos da la cantidad de clocks en función de cantidad de elementos ( Clocks(e) ). Para esto tener en cuenta que hay instrucciones en RISC-V que son pseudo-instrucciones que pueden estar integradas por más de una instrucción y en esos casos si hay x cantidad de clocks por instrucción sería x clocks por la cantidad de instrucciones de esa pseudo-instrucción. También hay que tener en cuenta los branchs y saltos.

Ej del ppt:

Clocks(e) = 36 clocks + (e-1) \* 24 -> Clocks(e) = 36 clocks + 24e-24

En ese ejemplo, el e-1 sería elementos – 1 porque el primer elemento del vector ya se cargó antes del loop, así que lo del loop contaría para los elementos desde el segundo en adelante.

**Pipelining**

Antes dijimos que existe una secuencia de clocks donde cada clock hace referencia a una etapa o fase de ejecución de una instrucción. Si bien en un principio mencionamos 4, en RISC-V podemos plantear este modelo didáctico de 5 pulsos de reloj, con la indicación resumida de qué componentes se ven involucrados en cada uno:

Fetch (IF) → Memoria de instrucción

Decode (ID) → Decoder, IMM y RegFile

Execute (EX) → ALU

Mem (MEM) → Memoria de datos

WriteBack (WB) → PC y RegFile RD

Como primera idea, pensaríamos que cada instrucción se resuelve una detrás de la otra, es decir, se llevan a cabo los 5 ciclos de una instrucción, dicha instrucción termina, y luego comienza otra que tendrá sus siguientes 5 ciclos, y así sucesivamente.

Pero como en cada ciclo de clock intervienen componentes diferentes entre sí, surge la idea del pipelining que es reutilizar dichos componentes entre ciclos de instrucciones distintas. O dicho de otra forma, que no haya que esperar a que una instrucción finalice todos los ciclos para comenzar la siguiente. Esto claramente nace con el objetivo de aumentar la eficiencia de la computadora.

De esa forma, si tenemos un conjunto de instrucciones y supongamos que empieza a ejecutarse la primera, el primer ciclo corresponde a IF. Pero cuando esta primera instrucción pasa a la fase de ID, la instrucción siguiente puede comenzar a ejecutarse en la fase de IF, y así sucesivamente.

Esto es posible gracias a que en medio de cada etapa y de cada grupo de componentes de cada etapa habrá registros intermedios que irán almacenando los valores de cada fase.

De esa manera, se llegaría a un CPI = 1, luego de exactamente los primeros 5 pulsos de reloj, luego de los cuales todos los componentes se encontrarían en uso y a partir de allí en cada nuevo pulso de reloj se terminaría de ejecutar una instrucción (ver pag 245 ppt).

Para poder aplicar pipelining es importante tener en cuenta que todas las instrucciones deben tener cantidad de ciclos de clock uniformes. Es decir, si hay instrucciones que tal vez no necesitan acceder a memoria, entonces carecerían del ciclo MEM. Pero para que todas tengan la misma cantidad de ciclos, se agrega entonces un NOP (stall o demora), para mantener la uniformidad.

El pipelining sin embargo puede verse en problemas, de los cuales se mencionan tres categorías:

* **Problemas estructurales**: se resuelven con hardware dedicado para evitar repetir etapas (como el caso del uso de sumadores especiales para el PC en lugar de reutilizar el de la ALU)
* **Problemas con los operandos** (datos): se da cuando una instrucción aún no terminó de ejecutarse (no llegó a la fase de WB), y la siguiente instrucción necesita de operandos que dependen de valores provenientes de la primera instrucción (esto se conoce como Read Before Write o Read After Write). Esto hay varias maneras de solucionarlo:
  + Agregando NOPs sucesivos, es decir, retrasando dicha segunda instrucción hasta que se dé el WB de la primera, perdiendo el CPI = 1
  + Agregando más hardware para que al mismo tiempo que se escribe un valor, el mismo pueda leerse
  + Cambiar el orden de las operaciones, en la medida de lo posible
* **Problemas en la ejecución**: estos se relacionan más que nada con los saltos (branchs) dentro de la ejecución de un programa. Se da en los casos en que se llega a una instrucción de un salto y el valor contra el cual se analiza la condición aún se desconoce. Una forma de solucionar esto (que es la utilizada hoy en día) es mediante un predictor de saltos. *“si supieras efectivamente si tenés que saltar o no entonces no fallaría ningún salto. El tema es que demorar la ejecución hasta que efectivamente sepas el valor del registro en cuestión implica que vas a perder ciclos. Ahora...en un salto...puede pasar que dependiendo del valor de dicho registro se tenga que saltar o no....o sea ..si te la jugas por saltar....tenés un 50% de chance de haber hecho lo correcto. Entonces elegis hacer algo siempre.  Algunas estructuras se repiten mucho...por ejemplo en un FOR vas a ejecutar algo y el salto va a ser hacia atrás....pensa que ese.salto va a salta siempre excepto en el último caso donde sale del FOR...por ende el predictor de salto va a notar que es un salto hacia atrás y va a jugársela por saltar.....y va a tener una chance mejor que el 50%.”*
  + En el caso de que el predictor de saltos ejecute algo incorrecto, se descartan aquellas instrucciones que no debían ejecutarse, pero se pierde el CPI = 1