**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 tipos 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. Dependiendo el tipo de instrucción (**R o I**) la salida de la ALU se selecciona tomando los 3 bits de **func3**. 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 1.000.000 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

Tenemos formas de clasificar los procesadores en función de cómo ejecutan las instrucciones:

* **ILP - Instruction Level Parallelism:** La técnica de Pipelining explota el **paralelismo a nivel de instrucciones**. En nuestro ejemplo con 5 etapas luego de una latencia de 4 ciclos de clock buscamos obtener un CPI=1. Es importante que el ancho de cada ciclo de reloj tenga como duración el peor caso en las fases. Por ejemplo, si la memoria de datos tiene un tiempo de acceso de 10ns, el ancho de pulso de clock debe tener como mínimo 10ns.
* **SISD vs SIMD:** Cuando el procesador trabaja con **una sola instrucción sobre un dato en particular**, es **SISD** (Single-Instruction Single Data). Ciertos procesadores pueden utilizar una **instrucción para trabajar en más de un dato**. Ej: a[x] = b[x]+c[x] para x desde 0 a 9. Un procesador **SIMD** (Single Instruction Multiple Data) realiza la operación de suma pero sobre un conjunto de datos en paralelo (vector).
* **Scalar vs Superscalar:** Cuando el procesador trabaja con **una única instrucción sobre un único dato es Scalar**. Los procesadores con pipelining que buscan el CPI=1 son Scalar. Si el procesador ejecuta **múltiples instrucciones pero cada una sobre un único dato es superscalar**. O sea en cada ciclo puede decodificar más de una instrucción (hypher-threading). Pero para esto tenemos que entender los procesadores con más de una unidad de ejecución…los cuales ejecutan las instrucciones fuera de orden y poseen dentro de la ALU más de una unidad de cada componente interno, por ej, màs de un sumador, más de un divisor, etc.

**Unidad 1: Memoria**

**Conceptos fundamentales**

Bloque funcional cuyo objetivo es almacenar todo tipo de información (datos e instrucciones).

Al diseñar y/o utilizar memorias existen características deseables:

● Capacidad de almacenamiento.

● Mayor velocidad de acceso.

● Menor costo.

● Mayor seguridad.

En arquitecturas de tipo Von Neumann, los datos e instrucciones compartian el mismo espacio de almacenamiento.

En arquitecturas de tipo Harvard existe un espacio para memoria de instrucciones y otro para memoria de datos.

**Organizacion jerarquica de la memoria**:

- Los **registros** tiene una capacidad muy limitada de almacenamiento. Por ejemplo el register file de una microarquitectura RISC-V tiene 32 registros. Son muy rapidos. Costo de CPU en funcion de la superficie de silicio que ocupan

- **Memoria cache** es intermediaria, de mayor capacidad de almacenamiento que los registros, aunque en lineas generales es de tamaño reducido. Velocidad de acceso alta, pero menor que la de los registros. No forma parte de la CPU, ni de la arquitectura de la computadora

- **Memoria principal**: mayor capacidad que las anteriores, más lenta, menor costo, nivel de seguridad elevado

- **Memoria secundaria**: no esta conectada directamente a los buses de la CPU, sino a travès de una interfaz, de ahi su nombre de secundaria. capacidad muy grande, lentas, costo bajo en relación a la capacidad, y la seguridad depende de la tecnología (si son magneticos, solidos)

En una arquitectura Harvard, la memoria de instrucciones es de solo lectura, mientras que la memoria de datos es de lecura/escritura.

**Diccionario de términos**:

* **Tiempo de acceso**: tiempo que se da entre que la CPU inicia una transferencia de datos hasta que la misma se completa.
* **Tiempo de ciclo**: es el tiempo de acceso más el tiempo necesario que se debe esperar hasta que se realice la próxima lectura.
* **Palabra de memoria**: unidad de N bits que puede ser consultada o grabada en memoria.
* **Palabra de CPU**: cantidad de bits con que puede operar la CPU en forma simultánea.
* **Costo**: relación dinero/capacidad de almacenamiento
* **Capacidad de almacenamiento**: tamaño de información que puede ser almacenada en un dispositivo de memoria específico. Puede ser expresada en diferentes unidades.
* **Wait state**: retardo mediante pulsos de clock producido por la CPU al esperar que un dispositivo de almacenamiento termine su tarea.

**Clasificación de memorias**

Las memorias pueden ser clasificadas de acuerdo a diversos criterios:

* Según su función:
  + ROM: memorias de solo lectura
  + RWM: memorias de lectura/escritura
  + RMM: memorias de lectura/escritura pero mayormente de lectura
  + NVRWM: memorias de lectura/escritura no volátiles
* Según la vida útil de los datos que se almacenan en ella:
  + Volátiles: pierden los datos cuando se interrumpe o corta el suministro eléctrico
  + No volátiles: no pierden los datos al cortarse el suministro eléctrico
* Según tecnología de fabricación:
  + Óptica
  + Magnética
  + Semiconductora:
    - Unipolar (MOS)
      * Estáticas
      * Dinámicas
    - CMOS
      * Estática
* Según forma de acceso a los datos:
  + Accesible por dirección: se requiere conocer la posición física del dato
    - Acceso aleatorio: tiempo de acceso no es en función de la posición de memoria (ej: RAM, memoria principal, etc)
    - Acceso secuencial: tiempo de acceso es en función de la posición de memoria
  + Acceso por contenido: se utiliza una parte del dato para acceder al resto (ej: memoria caché)

**La memoria principal está compuesta por:**

● Una parte de memoria secuencial.

● Una gran fracción destinada a RWM.

● Una porción de ROM.

● Y una cantidad escasa o nula de RMM (dependiendo del uso que se le dé a la computadora).

**Utilización de la memoria**

**ROM**

● Microprograma (CPU)

● Biblioteca de subrutinas (BIOS)

● Programas (BIOS)

● Tablas

● Programa completo en sistemas embebidos

**RMM**

● Parámetros que no se modifican a menudo

**RWM**

● Variables, estructuras, etc.

● Programas en ejecución

● Copia de porciones de ROM (shadow)

● Buffers de E/S

**● Estáticas / Dinámicas (SRAM / DRAM)**

Las RWM (Read Write Memory) son memorias de escritura/lectura, conocidas (no muy adecuadamente) como memorias RAM

Las NVRWM con las memorias de lectura/escritura no volatiles, ya que aunque se corte el suministro electrico, permanecen conectadas a una bateria que les continua proveyendo energia

En cuanto a la clasificacion de las memorias por tecnologia, tenemos que las memorias opticas y magneticas ocuparon un lugar muy importante, pero hoy en dia estan siendo cada vez mas reemplazadas por las memorias fabricadas con semiconductores.

Estas ultimas estan hechas con transistores y son de dos tipos: unipolares, de transistores de una misma polaridad (canal N o canal P); CMOS, de transistores de ambas polaridades.

BIOS -> Basic Input Output System

**Memorias SRAM y DRAM**

**SRAM - Static RAM**

Se usan en memoria principal y cache, aunque principalmente en cache.

Internamente estan compuestas por 6 transistores, que son los que se necesitan para formar un Flip Flop tipo D para 1 bit.

Internamente posee transistores complementarios, es decir, transistores de canal N y canal P.

- Almacenan cada bit en un circuito secuencial (Flip-Flop).

● Requieren varios transistores para almacenar cada bit.

● Retienen el valor almacenado mientras se mantenga la alimentación eléctrica.

● Relativamente rápidas tanto en la lectura como en la escritura.

**DRAM - Dynamic RAM**

Tienen mayor densidad (mayor cantidad de bits de almacenamiento por superficie de silicio)

Se usan en memorias principales

Solo se necesita un transistor para almacenar un bit, lo que implica un menor costo, pero tambien tiene una gran desventaja, que es que este tipo de memoria requiere un procedimiento que se llama refresco. Esto consiste en la lecrura sistematica y ciclica de sus datos para poder mantener los datos validos en memoria. Cada celda de memoria esta compuesta por un transistor + un capacitor para mantener la energia.

Un capacitor es basicamente un dispositivo que consiste en dos placas de elementos conductores y un elemento aislante entre ellas. Si conectamos una fuente de energia a un capacitor, en principio el capacitor va a estar totalmente descargado. Va a empezar una circulacion de corriente que va a seguir mientras se va cargando el capacitor. Una vez que el capacitor esta completamente cargado, la corriente deja de circular. Una de las propiedades, es que una vez que se le quita la fuente de alimentacion, este aun conserva su carga durante un periodo de tiempo que va a estar en funcion a las perdidas que pueda llegar a tener el material aislante. A lo largo de ese tiempo, el capacitor se va a descargar, y para mantener la carga que se le dio antes con la fuente de alimentacion, deberemos cargarlo nuevamente.

Las memorias DRAM tienen la desventaja de que sus capacitores pierden carga electrica, razon por la cual existe la necesidad de refresco. Para generar dicho refresco, se debe realizar una lectura de los datos en memoria, pero las lecturas son destructivas, es decir, nos devuelven el dato, pero tambien descargan el capacitor. Como contrapartida, simultaneamente un circuito regenera el dato en cada lectura, y hay que realizarlo en un periodo no mayor a 2ms, para no perder los datos.

El refresco consume tiempo. frente a esto, el diseño matricial nos permite realizar lecturas de una celda de memoria, lo que nos refresca toda la fila. Luego tenemos la regeneracion en rafaga. Y finalmente la regeneracion distribuida, que implica que cada circuito integrado, independientemente regenera la memoria alternando con accesos normales, y eso libera a la CPU de la tarea de refresco. Antes la CPU debia detener su proceso en curso, leyendo datos de la memoria (pero ignorandolos), para realizar el refresco, pero ese procedimiento con el tiempo se fue embebiendo en el propio chip de memoria para evitar que la CPU tuviera que llevar a cabo esta tarea.

* No utilizan circuitos secuenciales para almacenar un bit,

sino capacitores.

● Solamente requieren de un transistor para almacenar un bit.

● Se descargan aún cuando están alimentadas eléctricamente.

● Requieren circuitos de refresco.

● Se emplean en memorias de alta capacidad de almacenamiento donde se amortiza el costo de los CI adicionales.

**Memorias de solo lectura**

Son memorias que solamente pueden ser leídas, pero no grabadas por el usuario. La

grabación o programación de la memoria la realiza el fabricante a pedido del cliente.

**VENTAJAS**

**● En general son memorias de**

**acceso rápido**

**● No se pueden modificar**

**DESVENTAJAS**

**● Inconvenientes en el costo de fabricación**

○ La matriz de datos (memoria) es costosa. Para poder amortizarla se deben fabricar en gran cantidad.

**● Problemas de tiempo de fabricación**

○ El proceso habitualmente se terceriza y es realizado en el exterior.

**● No se pueden cometer errores**

○ Si se tienen en cuenta las desventajas anteriores, se puede observar fácilmente que un error tiene un costo en tiempo y dinero.

**Memorias de sólo lectura programable (PROM, memoria ROM “mejorada”)**

PROM: Memoria de sólo lectura cuyos datos son almacenados por el usuario.

● Su modificación posterior dependerá de la tecnología utilizada

● Se podría clasificar como RMM

● Dentro de las memorias PROM podemos encontrar

**PROM OTP**

**EPROM**

**EEPROM (E²PROM)**

**PROM OTP (One Time Programmable)**

● Pueden ser grabadas una única vez: el método de grabación es destructivo.

● Todos los bits que componen la memoria antes de ser grabada se encuentran en un

valor inicial conocido (por ejemplo en 0).

● Estos bits se encuentran conectados por un fusible que puede ser quemado con un

programador especial.

● Con la aplicación de un nivel de tensión y corriente especificados es posible quemar dicho fusible, programando un 1 en una posición particular.

● No puede revertirse: la escritura destruye un contacto físico (quema fusibles).

**EPROM (Erasable Programmable Read Only Memory)**

● Memorias de programación no destructiva.

● No se programan destruyendo fusibles.

● En su lugar poseen *transistores de puerta flotante* (FAMOS: Floating Gate Avalanch MOS).

● Los transistores permiten retener o no cargas eléctricas (1/0) por un largo período de tiempo.

● Son sensible a la radiación ultravioleta: pueden ser borrados con ella.

● Poseían una “ventana” para dejar ingresar luz al circuito para el borrado.

● La operación de borrado afectaba toda la memoria.

● Debía removerse del CI para el borrado.

● Se requería un dispositivo especial para el grabado.

● Tiempo de borrado variaba entre 5 a 10 minutos..

● La vida útil de los transistores quedaba limitaba a la cantidad ciclos de escritura.

**EEPROM (Electrically alterable EPROM o Electrically Erasable ROM)**

Ventajas:

● Poseen el mismo tipo de transistores que las EPROM con una pequeña modificación:

se posibilita el borrado a nivel de bit

● El borrado se realiza con electricidad, no con luz ultravioleta

● Incluyen en el CI lógica y hardware para escritura y borrado (no hace falta quitar la placa de memoria para escritura y borrado, como pasaba con las EPROM)

● Se pueden leer, escribir y borrar

● Admiten operaciones a nivel de bloque o a nivel de bit

Desventajas:

● El tiempo de acceso de lectura es mucho menor que el tiempo de escritura (Tacc **lectura** <<< Tacc **escritura**). Los bits deben ser borrados antes de grabarse.

● Al igual que las EPROM, las escrituras reducen la vida útil haciéndolas limitadas. En este caso la vida útil de las EEPROM es mayor que la de las EPROM.

● Estas memorias son más costosas que las EPROM: Requieren 2 transistores por bit (contra 1 de las EPROM), además de la lógica agregada para el borrado.

● La densidad de información por área es menor que en las EPROM.

● Se emplean para almacenar información que no será modificada regularmente.

**Memorias FLASH**

El objetivo al desarrollar las memorias FLASH era fabricar una memoria que sea no volátil, que pueda borrarse de forma electrónica sin extraerla del circuito pero con densidades y costos más cercanos a las EPROM y con velocidades de acceso parecidas a las de las EEPROM.

● Solución de compromiso entre las memorias EPROM y EEPROM (podrían catalogarse como RMM o NVRWM)

● Utilizan un tipo de transistor denominado ETOX (de funcionamiento similar al FAMOS)

● Permiten el borrado a nivel de bloque logrando así mayor densidad de bits que las EEPROM

● NOR FLASH: Cada CI que conforma un bit se asemeja a una compuerta NOR.

○ Densidad baja, mayor velocidad de lectura , menor velocidad de escritura y borrado

● NAND FLASH: Cada CI que conforma un bit se asemeja a una compuerta NAND.

○ Densidad media, menor velocidad de lectura, mayor velocidad de escritura y borrado

**Memoria Principal**

**● Tecnología semiconductora**

**● Acceso por dirección**

○ Principalmente de acceso aleatorio

**○** Pequeña parte de acceso secuencial (STACK)

**● Compuesta por**

**○** Lectura escritura (RW)

**○** Sólo lectura (RO)

**● Almacena datos y programas**

**● Forma parte de la arquitectura**

**de programación de la máquina**

(accesible al programador a través de instrucciones)

○ Debe estar disponible en cantidad suficiente para el programador

**● Transferencias**

**○** Hacia la CPU se realiza por palabra

**○** Hacia otra jerarquía de memoria se realiza por bloques (conjunto de palabras) controlada por hardware

La memoria principal almacena palabras. Una palabra se puede definir como la cantidad de bits que pueden ser manejados simultáneamente. La memoria principal, además, puede ser considerada como un **vector de registros**, al que accedemos indicando el subíndice (dirección) de cada posición dentro de ella.

La cantidad de líneas del bus de direcciones indica que la cantidad de “registros” o posiciones que se pueden direccionar son 2^n. El bus de control indica si el acceso es para realizar lectura o escritura. Y el bus de datos es para transferir datos hacia y desde la memoria principal. La cantidad de líneas del bus de datos indica el tamaño de palabra de memoria. Ej: un bus de datos de 16 bits indica que una palabra de memoria tiene 16 bits (para los casos de Load Word, aunque sabemos que se direcciona al byte).

Ver ppt desde pág **72 a 79** para temas **Bus de direcciones Multiplexado** y **calculo cant de circuitos integrados**

**Memoria Caché**

● La memoria caché es una memoria relativamente pequeña pero muy rápida.

● Su propósito es acelerar las operaciones CPU <-> Memoria.

● Lógicamente se ubica entre CPU y la memoria principal.

● Físicamente puede estar en el mismo chip de la CPU o en un módulo separado (y

ambos si es multinivel).

○ El ancho de banda es muy superior cuando se encuentra en el mismo CI (on-chip)

La CPU sigue accediendo sólo a la memoria principal. La memoria caché opera en forma

transparente a las operaciones de la CPU.

***Patrones comunes y predecibles en el acceso a memoria:***

**Principio de localidad temporal:** cuando se accede auna instrucción o dato es altamenteprobable que se la vuelva a accederen el futuro cercano (ejemplo típico: bucles)

**Principio de localidad espacial:** cuando se accede auna instrucción o dato es altamenteprobable que las instrucciones odatos cercanos sean accedidos en elfuturo cercano (ejemplo típico: elementos de un vector)

MMU -> Memory Manager Unit (es el administrador de la memoria cache)

La memoria cache se carga a medida que se requiere. Observe que se carga un bloque entero

**¡Debemos almacenar *dirección y contenido*!**

**La ETIQUETA se guarda en cada línea. El OFFSET no se guarda, se utiliza para seleccionar la palabra requerida.**

**Cada línea de MC contendrá una etiqueta y el contenido de las posiciones de memoria cuyas direcciones comienzan con ETIQUETA.**

**Bits de control:** *(Los fija y mantiene la MMU):*

* **Validez**: indica si el contenido de la línea debe tenerse en cuenta en los accesos a MC.
* **Suciedad (Dirty)**: indica si el contenido de la línea se modificó luego de su carga, y debe copiarse en MP antes de ser reemplazada.

La memoria caché contiene una **copia** de una **porción** de memoria principal.

1. CPU solicita acceso a una dirección de memoria

2. El administrador de la memoria caché (MMU) verifica si la dirección existe en la caché

3. Si está presente (HIT, éxito) se lee desde caché

4. Si no está presente en la caché (MISS, falla), se lee el bloque correspondiente de MP (*prefetch*) y se actualiza la MC

**Características de la memoria caché**

**Construcción:** memoria estática de alta velocidad (SRAM)

**Direccionamiento**: por contenido (CAM: *content addressable memory*).

No existen direcciones de caché, sino que se almacena la dirección y el contenido.

Utiliza comparadores para verificar si una dirección de MP está presente en MC.

**Política de carga y escritura:** en qué momento se actualiza la memoria principal

cuando se modifica la caché. Cómo se realiza la lectura de MP para completar la MC.

**Organización:** Debe determinarse la existencia de una palabra en caché de forma

extremadamente rápida (se analizan a continuación tres formas).

**Tamaño de la memoria y del bloque:** impactan en la performance por lo que se

busca un equilibrio

**Costo:** la memoria caché se encarece al aumentar de tamaño por su propia construcción

y por los comparadores que requiere para operar.

34

En cuanto a los tipos de organización/asignación en MC, tenemos:

* **Asociativa**
* **Mapeo directo**
* **Asociativa por conjunto**

Ver detalle de cada una en ppt y apuntes

**Comparativa: Ventajas y desventajas**

**Asociativa:**

● Buena performance, pero también diseño más complejo.

● Comparadores de mayor tamaño

● Obliga a implementar políticas de reemplazo

● Se utiliza en el TLB con unas pocas docenas de entradas (muy pequeña)

**Mapeo directo:**

● Diseño simple: menor tamaño de la memoria de etiquetas y comparadores más reducidos

● No requiere búsqueda asociativa.

● Se suele usar para caché de instrucciones.

● Riesgo de thrashing

● Peor performance

**Asociativa por conjuntos:**

● Combina simpleza de Directa con eficiencia de Asociativa.

● Cuanto más grande el conjunto (n- vías) mayor la performance pero también la complejidad y costo.

● Obliga a implementar políticas de reemplazo

● Mejor performance

**Ver Más info (con graficos) en ppt de Memoria cache y apuntes propios (la mayoría de los puntos importantes ya están en los apuntes)**

**Unidad 2: Entrada/Salida – Debugging - Comunicaciones**

**Unidad de Entrada/Salida (apuntes de los videos)**

Dicha unidad incluye tanto las interfaces de E/S (que se conectan con la CPU a través de los buses) como los dispositivos de E/S que son aquellos que conectan elementos como los periféricos y almacenamiento secundario (disco rigido, etc) a las interfaces de E/S

**Interfaces** las hay **genéricas** (que pueden permitir conectar distintos dispositivos de E/S por ej el caso de un puerto serie o un puerto USB), y las hay **dedicadas** (que conectan un tipo de dispositivo en particular, ej un puerto HDMI que es para conectar dispositivos relacionados a videos)

**Dispositivos** los hay **dedicados a carácter** y dispositivos **orientados a bloque**

Los dedicados a carácter son los dispositivos que transmiten los datos hacia los periféricos o reciben los datos de a un carácter (por ej, valores ASCII, de 8 bits). Son dispositivos más antiguos que están dejando de existir en la industria. La tecnología actual trabaja por bloques, transmitiendo y recibiendo los datos por grupos de bytes.

Las interfaces de E/S son necesarias como intermediarias entre los dispositivos de E/S y la CPU por distintos motivos:

* Adecuar las velocidades
* La transferencia de datos hacia la CPU sea uniforme sin importar el dispositivo.
* Permitir transferencias desde y hacia dispositivos asincrónicos (ej, teclados, que cuando uno escribe en ellos no lo hace regido por un mecanismo de clock que es como los datos viajan en el interior de la CPU).

Las interfaces a su vez cumplen funciones varias, siendo una de ellas el almacenamiento temporal de los datos, el cual menciono porque este concepto hace referencia a los buffer. Es como si se acumularan en un buffer los datos y la interfaz los va transmitiendo en un flujo ordenado hacia la CPU (pensar en el concepto de un embudo).

Funciones:

* Comunicación con CPU / buses (adaptador)
* Comunicación con el dispositivo de E/S (controlador)
* Control y temporización
* Almacenamiento temporal de datos
* Detección / Corrección de errores

Existen estructuras de **3 buses** y **4 buses** en cuanto a lo que es el mapeo de E/S

La de **3 buses** implica que en memoria principal habrá un espacio dedicado a direccionamiento para las interfaces de E/S, que recibirá las direcciones especificadas por la CPU y luego remitirá a la interfaz de E/S correspondiente.

La de **4 buses** en cambio implica que habrá un bus específico para direccionamiento de E/S que saldrá de la CPU y se conectará directamente con la unidad de E/S, sin pasar por la memoria principal.

Tener en cuenta que aunque estemos hablando de 4 buses en realidad son 4 buses lógicos, pues físicamente siguen siendo 3 buses pero mediante el bus de control la CPU indica si se trata de una lectura/escritura para MP o para E/S. O sea, el 4to bus se implementa *realmente* multiplexando el bus de direcciones con una línea adicional de bus de control: M/IO.

En el caso de RISC-V se utiliza la estructura de 3 buses, entre otras cosas porque permite utilizar las mismas instrucciones para MP que para E/S.

**Tres buses: (Memory mapped IO)**

1. - Se usan las mismas instrucciones para E/S que para memoria.
2. - Se pueden efectuar operaciones aritméticas y lógicas en los datos de E/S.
3. - Requiere lógica de decodificación más compleja.
4. - Las direcciones de E/S son del mismo tamaño que las direcciones de memoria.
5. - Reduce la memoria disponible.

**Cuatro buses (IO mapped IO)**

1. - Se usan instrucciones especiales (p/e IN, OUT).
2. - No se pueden efectuar operaciones aritméticas y lógicas directo en E/S.
3. - La lógica de decodificación es más simple.
4. - El espacio de direccionamiento de E/S es inferior.
5. - Las direcciones de E/S suelen ser más reducidas que las de memoria.
6. - Toda la memoria direccionable está disponible.

Cada interfaz tiene una **dirección base**. Esta es la identidad de la interfaz, es único por interfaz.

Basándome en el diagrama simplificado de lo que es un interfaz para entender también el concepto de dirección base, vendría a ser algo como esto (ver gráfico de diap 10 ppt E/S):

Tenemos por un lado el bus de direcciones, el de datos y el de control. El bus de direcciones esta conectado a un registro buffer que esta dentro de la interfaz y el de datos esta conectado a otro registro buffer tambien dentro de la interfaz. En ambos casos el registro buffer sirve para almacenar temporalmente en el caso del bus de direcciones la ultima direccion que se encuentra en dicho bus, y en el caso del bus de datos el ultimo dato que sale de la interfaz de E/S o bien el ultimo dato que ingresa.

La dirección que se encuentre almacenada en el registro buffer conectado al bus de direcciones se compara con un comparador contra otro registro interno (**registro de direccionamiento base**) de la interfaz que es el que contiene la famosa dirección base. Dicha dirección base no es otra cosa que una dirección en la que se encuentran representados los bits más significativos que comparten y son comunes a otros registros dentro de esa y solo esa interfaz. Si el comparador comprueba que la dirección que está en el registro buffer en sus bits más significativos coincide con el registro de direccionamiento base entonces eso significa que la CPU ordenó interactuar con esa interfaz de E/S en particular, ya sea para recibir datos de un dispositivo, enviar datos a dicho dispositivo, recibir un estado del dispositivo (ej una impresora que indica que falta papel, o esta sin tinta), o bien para enviar una señal de control al dispositivo. Luego de la comparación, esto se hace a través de un decodificador interno de la interfaz y con otros registros que poseen dicha dirección base en sus bits más significativos pero que varían en los bits menos significativos (ver diagrama para entender mejor). Uno de ellos tendrá la dirección correspondiente que provino del bus de direcciones, enviada por la CPU desde el MAR (memory access register).

**Descripción de la interfaz**

**Registros de datos:** Almacenan datos provenientes del periférico o que se transferirán al mismo.

**Registros de control:** CPU escribe ahí los comandos para el periférico.

**Registro de estado:** Contiene el estado del dispositivo, p/e errores.

**Decodificador:** Determina a qué registro (entrada, salida, estado, control) direccionar lo que se transmite por el bus de datos (lectura o escritura), Utilizando para ello lo recibido del bus de direcciones.

Independientemente si se trata de una estructura de 3 buses o 4 buses, las interfaces siempre se mapean, es decir, para la CPU la comunicación con las interfaces se va a realizar por medio de direccionamientos. Si es una estructura de 3 buses, formara parte de los direccionamientos de memoria, y si es de 4 buses, formara parte de un direccionamiento independiente.

El conjunto de interfaces de entrada/salida (mapeado en memoria o en forma independiente) se ve desde la CPU como un rango de direcciones. Dicho rango puede ser discontinuo.

Cada interfaz es vista desde CPU como un conjunto de registros accesibles al programador, dedicados, que forman parte de la arquitectura.

Cada interfaz tendrá un rango de direcciones asociado, donde cada registro interno accesible tendrá una dirección correspondiente.

La transferencia de datos entre interfaces y CPU puede ser sincrónica (comparten reloj, o señales de clocks) o asincronica (cada interfaz puede operar a su propia velocidad sin importar la velocidad de otras interfaces sobre el mismo bus). Esta ultima es más compleja, por eso la mayoría de las computadoras modernas utilizan la sincrónica.

**Sincrónica**

1. - CPU e interfaz E/S comparten reloj
2. - La temporización interna generalmente difiere

**Asincrónica**

1. - CPU e interfaz E/S tienen temporización independiente
2. - Requiere señales de control para indicar transferencia
3. - Utilización de pulso de habilitación
4. - Handshaking (reconocimiento mutuo) {ACK, timeout, etc.}

**Algunos conceptos**

**Latencia**: tiempo total desde que se inicia la transferencia hasta que concluye.

**IOPs**: Input/Output Operations per Second.

**Throughput**: Similar a ancho de banda, medida de transferencia por unidad de tiempo.

**Ancho de banda**: tasa máxima de transferencia.

Lo ideal es que en lo que son las interfaces de E/S exista la menor latencia posible, la mayor cantidad de IOPs (operaciones de entrada/salida por segundo) y el mayor throughout. Este ultimo esta relacionado con el ancho de banda y debemos pensarlo de esta manera: supongamos la Gral Paz cuya velocidad maxima es de 80 km/h (eso seria el ancho de banda, el maximo teorico de transferencia), pero en la realidad un lunes a la mañana no se alcanza esa velocidad por el flujo de vehiculos que se da a esa hora, la velocidad que se alcanza es menor. Esa velocidad real es lo que vendria a ser el throughput.

**Tipos de administración de entrada/salida:**

1. Lentos

* E/S Programada (Software): Polling
* E/S por Interrupciones (Hardware)

1. Rápidos

* Acceso directo a Memoria (DMA)

**- Polling** (se hace por software, programada):

CPU verifica en forma constante el estado de la interfaz, si por algún motivo la interfaz requiere atención según lo que arroje el registro de control, se atenderá y luego se volverá al polling constante de consultas del estado.

Al ser por software es más flexible, pero también hace que la CPU deba estar continuamente "preguntando" por el estado de las interfaces.

Según el caso, cuando hay más de una interfaz involucrada, se puede decidir cómo se maneja esto. Si se pregunta por el estado de una interfaz, se detecta que se debe atender o ejecutar una subrutina de acuerdo a un estado, una vez concluida dicha "atención" se puede decidir por software si se reiniciara el ciclo consultando por el estado de cada interfaz de nuevo desde el principio, o si se continuara preguntando por el estado de la interfaz siguiente.

**- Interrupciones** (se hace por hardware):

Tiene la gran ventaja de que la CPU no tiene que estar preguntando todo el tiempo por el estado de las interfaces.

Pensemos como si fuera un mozo, que se acerca a la mesa de los comensales cuando estos requieren de su asistencia, para lo cual interrumpe lo que estaba haciendo antes para atender a los clientes.

Ahora la CPU tendrá una señal extra en el bus de control (es decir, hardware) para operar de esta manera. Interrumpe lo que estaba ejecutando, dejando los registros como están, cambiando el PC apuntando a la subrutina encargada de atender la petición de interrupción (ISR), y cuando termina con eso, retoma todo donde lo dejo.

Todas las interrupciones son asincrónicas porque no podemos predecir el momento en que se producirán ni determinarlo nosotros (cuando se oprime una tecla, se conecta un nuevo dispositivo, etc).

Las hay de dos tipos: **IRQ** (interrupciones que se pueden ignorar de momento y atender después) y **NMI** (interrupciones no enmascarables, es decir, que no se pueden ignorar momentáneamente ni postergar). Ej de esta última puede ser cuando se interrumpe el suministro eléctrico y la computadora tiene que decidir y actuar rápidamente para intentar salvaguardar lo último que se estaba trabajando antes de quedarse sin energía por completo.

Ambas son señales que recibirá la CPU por el bus de control.

Tener en cuenta que igual la verificación de que exista alguna IRQ la CPU lo hace justo antes de ejecutarse una instrucción (no durante, ya que en ese caso no interrumpiría dicha ejecución).

¿Ahora bien, como sabe la CPU al momento de recibir una IRQ de que interfaz provino?

Puede ser que **cada interfaz tenga su propia línea de control** para transmitir la IRQ, entonces ahí la CPU sabe claramente cuál fue la interfaz que la origino, pero esto no es tan fácil de implementar.

Otra posibilidad es que exista una **línea compartida por todas las interfaces** para transmitir la IRQ a la CPU, en cuyo caso esta última debe averiguar cuál fue la interfaz que se la envió. Para ello puede solo entonces, cuando recibe la IRQ, hacer una especie de polling hasta saber cuál fue. O bien **comunicación encadenada** (en esta materia no se ve). O bien un **circuito programable de interrupciones** que tiene una única conexión con la CPU y una conexión por interfaz (el famoso PIC). O bien una **interrupción vectorizada**, que implica que en el momento en que la CPU recibe la IRQ, envía una señal para que la interfaz que la origino responda por ej con un vector de datos para identificarla.

**Ver desde diap 18 a 28**

**- DMA** (Acceso directo a memoria):

Es una técnica que va a permitir transferir datos desde memoria principal hacia interfaces y viceversa, evitando así que por ej un dato que viene desde una interfaz que debe guardarse en memoria deba pasar primero por la CPU y de la CPU a memoria.

Para esto habrá una interfaz dedicada a esta tarea. CPU se hace a un costado y no hace de intermediaria.

Es un método muy rápido. Pero esto no trabaja como una interrupción, ya que la CPU no realiza ningún cambio de contexto, le cede los buses al DMA controller.

Programable para mover bloques entre MP y E/S o entre zonas de memoria

Toma el control de los buses cuando CPU se lo concede

Notifica a CPU con una interrupción cuando termina

Para DMA hay dos técnicas:

* **Detención de CPU** (por un periodo muy corto de tiempo mientras se desconecta la CPU de los buses, quedando en un estado de alta impedancia, y el controller ejecuta su función, se transfieren bloques grandes de datos por ráfagas)
* **Robo de ciclos** (el controller utiliza semiciclos de la CPU en los que la misma no está usando los buses para realizar su trabajo, ralentizando un poquito los ciclos de la CPU para ello, y se transmite una o más palabras).

**Ver diap 29 a 35 ppt**

Info extra de **PIC** según ChatGPT:

En el ámbito de interfaces de entrada/salida (E/S), PIC se refiere a "Programmable Interrupt Controller" (Controlador de Interrupciones Programable). Este dispositivo es crucial en sistemas informáticos para gestionar las interrupciones generadas por dispositivos periféricos o eventos dentro del sistema. Su función principal es arbitrar y priorizar las interrupciones provenientes de varios dispositivos conectados al sistema, asegurando que el procesador maneje adecuadamente las interrupciones y responda de manera eficiente a cada una.

El PIC permite al sistema informático atender múltiples dispositivos simultáneamente mediante la gestión de interrupciones, asegurando que cada dispositivo pueda notificar al procesador cuando necesita su atención. Esto es fundamental para la operación eficiente de periféricos como teclados, ratones, unidades de disco, tarjetas de red y otros dispositivos que interactúan con el sistema.

Es importante mencionar que con el avance de la tecnología, los PIC tradicionales han sido reemplazados en muchos sistemas modernos por controladores de interrupciones avanzados como el APIC (Advanced Programmable Interrupt Controller), que ofrece capacidades adicionales y mayor eficiencia en entornos más complejos y con mayores demandas de rendimiento.

**Unidad de Comunicaciones (apuntes de los videos)**

Para comunicaciones en serie, es decir, comunicaciones en las cuales los datos o los bits se transmiten por una linea o cable uno detras del otro, se utiliza el concepto de baudio.

Un **baudio** es una unidad de medida que indica la velocidad de transmision en una comunicacion serie. Representa la **cantidad de bits que se transmiten por segundo**, y se conoce tambien como "baud rate". Es asi que 9600 baudios, serian 9600 bits por segundo.

El **Tbit** es el tiempo dentro de un segundo que corresponde a un bit.

Por ej: Tbit = 1seg / 9600 baudios (o bauds) = 0,000104167 segundos = 104, 167 microsegundos por bit

1 seg = 1.000.000 microsegundos

**UART**

Universal Asynchronous Receiver/Transmitter (UART) es un componente de hardware que facilita la comunicacion en serie en una computadora u otro dispositivo electronico. Es un intermediario entre el procesador y los dispositivos externos que usan una interfaz en serie.

Lo que hace es convertir los datos paralelos del procesador (esto es, los bits que se reciben en forma paralela a traves de los buses), en una secuencia de bits en serie, es decir, un bit detras del otro en lugar de uno "al lado" del otro en forma paralela, para su transmision, y lo mismo hace cuando recibe los bits externamente en serie y los debe convertir de nuevo en bits paralelos para enviarlos al procesador.

Al ser asincronico, no existe señal de clock o de sincronismo para que cuando el transmisor envia datos sincronicamente el receptor este preparado al mismo tiempo para recibirlos. Esta comunicacion entonces se establece o se lleva a cabo de otra manera, fuera de lo que seria una señal de clock.

En primer lugar, cuando no se estan transmitiendo datos en la linea de transmision (que seria el cable por donde circularian los bits en serie), se encuentra en estado de reposo o "idle", y en este caso se encuentra en 1.

Cuando se comienzan a transmitir los datos, por cada byte se transmite un bit de "start" o inicio para avisar al receptor que se prepare para recibir datos. Dicho bit esta en 0.

Luego vienen los bits de datos, que contienen la informacion en si, los cuales pueden configurarse en 7 u 8 bits.

Finalmente, viene el parity bit (o bit de paridad, el cual es opcional, puede estar o no), y el bit de stop.

El tema es, una vez que se recibe el bit de start o inicio, ¿en que momento sabe el receptor que dicho bit ya concluye y comienzan a recibirse los bits que son propiamente de la informacion?

Para ello, cuando la señal de reposo en 1 pasa a 0 que seria la del bit de start, se inicia un contador que tiene como frecuencia el baud rate (baudios, o cantidad de bits por segundo) multiplicado por 16. es decir, cada bit va a tener un ancho de 16 clock de esa señal.

El contador cuenta hasta 8 para asegurarse de estar en principio en el centro del bit de inicio. Y luego ira contando de a 16 par air ubicandose en el centro de cada bit, sabiendo asi cuándo inician los bits de datos propiamente (los bits de datos seran 7 u 8 segun el caso), hasta llegar al final con el o los dos bit de stop que siempre son 1.

Tener en cuenta que para el bit de paridad:

O -> ODD (impar en ingles, ODD tiene 3 letras, relacionarlo con impar)

E -> EVEN (par en ingles, EVEN tiene 4 letras, relacionarlo con par)

N -> No hay bit de paridad

Ejemplo de formato:

7O2 -> 1 bit de start (siempre esta, aunque implicito) + 7 bits de datos + 1 bit paridad impar + 2 bits stop = 11 bits por caracter

(Ver mas conceptos y calculos IMPORTANTES al respecto en ppt)

bps -> bits de informacion (pura) por segundo. Informacion pura hace referencia al dato en si (no los bits de start, stop, o paridad, que son llamados bits de sincronismo o seguridad)

El cociente entre bauds / bps nos da como resultado la eficiencia del canal

**SPI**

Serial Peripheral Interface, es un protocolo de comunicacion serial para transferir datos entre un master y uno o varios dispositivos esclavos.

El bus cuenta con 4 lineas principales:

- SCLK (serial clock, desde el master a los slaves, para establecer los instantes en que se transmiten los datos),

- MOSI (para transmitir datos desde el master al slave)

- MISO (para transmitir datos desde el slave al master)

- SS (slave select, es una linea por cada slave que este conectado al master)

Todas las lineas son unidireccionales

Ver en ppt explicacion de los pasos en que se da la comunicacion

**I2C**

Inter Integrated Circuit. Es un protocolo de comunicacion serial que permite la transferencia de datos entre dispositivos electronicos usando solo dos cables: una linea de datos (SDA - serial data) y una linea de reloj (SCL - serial clock)

Ver en ppt explicacion de los pasos en que se da la comunicación

**Debugging**

Una **computadora de propósitos generales** contiene los 3 bloques funcionales (**CPU, MEM, E/S**). Generalmente tiene buses deexpansión para E/S (USB, PCI,ETC), puertospara memoria secundaria (sata,M.2, ETC), yzócalos para **expandir la memoria principal** (SDRAM). Suele poseer un **S.O.** que permitecambiar el programa (aplicación) según lanecesidad.

Un **microcontrolador** también contiene los 3 bloques funcionales pero integrados en un **único chip**. Suele tener periféricos de E/S que permiten conectar dispositivos (GPIO, UART, SPI, I2C, ETC). **La memoria principal** **generalmente NO se puede expandir** y la memoria secundaria (FLASH) tampoco. Suelen programarse **Bare-Metal** (sin S.O.) o con un RTOS específico. No suele cambiar la aplicación o el programa (salvo updates).

Un breakpoint es un punto establecido donde la ejecución de un programa se va a detener (break, se va a romper), mientras que un watchpoint es un indicativo que cuando se produce un cambio en una posición de memoria (una variable por ej), se va a levantar una condición que indica justamente ese cambio.

Para insertar un **breakpoint o watchpoint** (variable) se utiliza un **registro donde se almacena una dirección** contra la cual se valida una condición. Estas condiciones dependen de la arquitectura pero en su mayoría todassoportan detectar:

**● Ejecución de instrucción en la dirección**

**● Lectura o Escritura de datos en la dirección**

**La cantidad de estos registros suele ser limitada**. En x86 existen solo 6. En algunas arquitecturas solo 2. En RISC-V depende de la microarquitectura (se recomienda como mínimo 4 hasta 4096). **Estos son conocidos** **como breakpoints HW**. El ESP32c3 soporta 8. El SiFive FE310 soporta solo 2.

Dado que un programa ejecuta desde memoria principal (al menos en x86), los debuggers generan **breakpoints SW**. Esto se logra **reemplazando instrucciones del programa por instrucciones de tipo break**. Estos breaks **detienen la ejecución y el debugger puede entonces introducir la instrucción original**. De esta forma eldebugger se convierte en una especie de intérprete.

**Si el programa ejecuta directamente desde Flash** (con QSPI Flash esto es posible), entonces estos **breakpoints SW son imposibles**.

Ver diap 17

Según la arquitectura RISC-V, se define como **excepción** a una condición inusual que se produce durante la ejecución de una instrucción. Ejemplo: se ejecuta una instrucción ebreak (breakpoint sw), se accede a una dirección de manera no alineada, etc. Si bien en otras arquitecturas dividir por cero genera una excepción, en RISC-V no (página 44 del manual).

Una **interrupción** es un evento externo asincrónico (que puede o no ser atendido dependiendo de los registros mstatus, mie y mip). Tanto en el caso de una excepción o de una interrupción (atendida), se “atrapa” (**trap**) la misma transfiriendo de manera sincrónica el control de la CPU a una rutina manejadora de trap. A tal fin, existen registros específicos en donde la microarquitectura inserta valores necesarios para identificar el estado:

**● mepc**: Este registro almacena el valor del PC cuando se produjo la excepción o interrupción.

**● mcause**: Este registro indica si se produjo una excepción (bMS=0) o una interrupción (bMS=1). El resto almacena un número de identificación.

**● mtval**: Este registro almacena un valor necesario para resolver el problema. Ej: Load Address Misaligned es un acceso no alineado en una instrucción tipo lw, la dirección accedida (desalineada) se almacena en mtval.

**● mscratch**: Registro auxiliar para valor temporal.

Los registros de uso general NO se almacenan automáticamente. Para retornar del trap se utiliza una instrucción especial **mret** que toma la dirección de mepc.

**Vector de interrupciones**

RISC-V define un registro (**mtvec**) donde se almacena la dirección de comienzo del vector de interrupciones. Los dos bits menos significativos de este registro indica el modo de operación:

● **Modo Directo** : Toda excepción o interrupción es manejada por mtvec[0].

● **Modo Vectorizado**: Las excepciones se manejan por mtvec[0], pero las interrupciones se manejan por mtvec[mcause\*4]. En cada posición del vector se almacena una instrucción. Cada una de esas instrucciones debe saltar al trap correspondiente. Cambiando el bit **mstatus.mie=0** se apagan todas las interrupciones (NO las excepciones). RISC-V soporta niveles de privilegio (usuario, supervisor, maquina). En el caso de interrupciones define 3 tipos de interrupción:

**● Software:** Estas interrupciones son generadas por código que escribe en el controlador programable de interrupciones (CLINT).

**● Timer:** Se generan cuando se produce un evento de timer que requiere atención.

**● Externa:** Se producen como resultado de un dispositivo o periférico que interrumpe.

En el caso de esp32 (esp-idf) esto se define en components/riscv/vectors.S

Apuntes propios de los videos de esto último:

Una excepcion es un evento inusual, no esperado, que se da cuando se esta ejecutando una instruccion, pero el evento se origina a raiz de algo que ocurrio en la propia instruccion.

Una interrupcion, en cambio, es totalmente externo a la ejecucion de un programa, y es asincronico porque no sabemos en que momento llega.

Tanto en el caso de una excepcion como de una interrupcion (atendida) RISC-V atrapa (trap, similar al concepto de catch en Java por ej) la misma para una rutina de la CPU para que maneje la misma, usando ademas unos registros especificos que nos brindan datos respecto de dicha excepcion o interrupcion. NO son los registros de uso cotidiano cuando programamos, pero aun asi son accesibles al programador. Por ej mepc sirve para saber que instruccion fue la que fallo (ya que nos devuelve la direccion de dicha instruccion). mcause nos sirve para identificar la causa del problema (en el bit mas significativo indica con 0 o 1 si fue una excepcion o una interrupcion, y en el resto, en base a una tabla, especifica la causa especifica).

La instruccion que se utiliza para salir de una subrutina trap, que es la que maneja la excepcion o interrupcion, es mret la cual toma el valor que figura en mepc, es decir, la direccion de memoria de la instruccion que provoco el fallo. Como obivamente la ide ano es volver a esa misma instruccion porque nos va a provocar nuevamente la misma excepcion, lo que hacemos es sumar 4 a mepc o sumarle el valor que corresponda para saltar a otra instruccion, saliendo de la subrutina trap.

mtvec es un registro que almacena la direccion de comienzo de un vector. Dicho vector es conocido como vector de interrupciones, y en cada posicion de esa vector va a haber una isntruccion.

Segun el modo dde operacion, se hace un diferente uso de dichas instrucciones, que no son otra cosa que lo que va a hacer el CPU al momento de producirse una excepcion o interrupcion + luego lo que establezca el programador.

Con modo directo, toda excepcion o interrupcion es manejada por mtvec en pos 0 (la isntruccion que esta en esa posicion).

Con modo vectorizado, las excepciones son manejadas por mtvec pos 0, pero las interrupciones seran manejadas por pos mcause (nro de causa) \* 4.

RISC-V soporta tres niveles de privilegio (usuario -> para los programas, supervisor -> para sistema operativo, maquina -> para el manejo directo del hardware). En la materia usariamos el nivel maquina.

En esp32, el vector del que hablabamos antes tiene por ej en la pos 0 un salto a una instruccion con la etiqueta panic\_handler, y en pos 1 a interrupt\_handler.

En el caso de panic\_handler, lo que hace esta en primer lugar es llamar a una macro llamada save\_general\_regs la cual lo que hace es guardar cada uno de los registros (los que usamos siempre) y los almacena en el stack pointer para tener el estado de cada registro y no perderlo antes de hacer el tratamiento de la excepcion. Luego, al final, tiene lo contrario, que es restaurar los valores de los registros con la macro restore\_general\_regs.

En interrupt\_handler pasa lo mismo, y ademas tiene una intruccion que es global\_interrupt\_handler que maneja un vector con instrucciones que manejan cada tipo de interrupcion que se haya producido.

Existen tres tipos de interrupciones: de software (que no las vemos en esta materia), timer (el timer es basicamente un contador, que puede incrementar o decrementar), o externas (producidas por ej por un periferico de E/S).

Las interrupciones de timer lo que hacen es usar un comparador con un valor fijo, y cada vez que el timer o contador llega a ese valor y se verifica en la comparacion, produce una interrupcion. Otra manera de producir interrupciones con timer es cada vez que este contador produce overflow.