*Nota: Si hay algún error sepan disculpar, estuve trabajando en base a la teoría de arquitectura de computadoras del 2023, videos de Genaro Camele y conocimiento propio, cualquier apoyo a mejorar este documento bienvenido es, nos vemos en el parcial del 14/11. Gracias por leer <3*

Inicio: 20/10/2023

Fin: 11/11/2023

**Manual para el 2do Parcial de Arquitectura de Computadoras 2023**

Hecho por: San Rojas de **La Fuente**

Índices

[1\_ INTRODUCCIÓN A MIPS 1](#_Toc150596332)

[2\_ SET DE INSTRUCCIONES 4](#_Toc150596333)

[3\_ ATASCOS Y CONFIGURACIONES 7](#_Toc150596334)

[4\_ PUNTO FLOTANTE 16](#_Toc150596335)

[5\_ USO DE REGISTROS (Convenciones) 17](#_Toc150596336)

[6\_ SUBRUTINAS 18](#_Toc150596337)

[7\_ ENTRADA Y SALIDA 21](#_Toc150596338)

[8\_ PARCIAL 24](#_Toc150596339)

# 1\_ INTRODUCCIÓN A MIPS

**RISC (*Reduced Instruction Set Computer*)**

El hardware se organiza de una manera nueva llamada Segmentación de Cauce (Pipeline), quiere decir que se puede realizar más de una operación al mismo tiempo (¡PARALELISMO! WOW!!)

Las instrucciones se organizan en FASES de manera que esto sea posible (el paralelismo)

Explota el *paralelismo* entre las fases de distintas instrucciones

**Ejecución secuencial**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **INSTRUCCIÓN 1** | **->** | **INSTRUCCIÓN 2** | **->** | **INSTRUCCIÓN 3** |

**Ejecución segmentada**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **INSTRUCCIÓN 1** | **IF** | **ID** | **EX** | **MEM** | **WB** |  |  |
| **INSTRUCCIÓN 2** |  | **IF** | **ID** | **EX** | **MEM** | **WB** |  |
| **INSTRUCCIÓN 3** |  |  | **IF** | **ID** | **EX** | **MEM** | **WB** |

Las instrucciones se organizan en fases

**[IF]**

**Instruction FETCH**

Se accede a memoria por la instrucción

Se incrementa el PC (Program Counter o antes conocido como Instruction Pointer)

**[ID]**

**Instruction DECODING**

Se decodifica la instrucción

Se accede al banco de registros por los operandos. Se ATASCA si NO están disponibles

Se calcula el valor del operando inmediato

Si es un salto, se calcula el destino y si se toma o no (Requiere acceder el banco de registro)

**[EX]**

**EXECUTION**

Si es una instrucción de cómputo, se ejecuta en la ALU

Si es un acceso a memoria, se calcula la dirección efectiva

Si es un salto, se realiza (Se modifica el registro PC)

**[ME]**

**MEMORY ACCESS**

Si es un acceso a memoria, se lee/escribe el dato

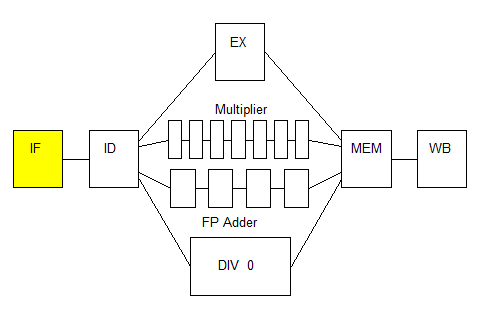
**[WB]**

**WRITE BACK**

Se almacena el resultado (si lo hay) en los registros

*Nota: no todas las etapas duran lo mismo.*

Por ende, no todas pueden manejarse en paralelo (¿No que si se podía?)



**En la etapa de EX tenemos lo siguiente**

Suma: No tarda mucho

Multiplicación: Tarda un poco

División: Tarda mucho

# 2\_ SET DE INSTRUCCIONES

*Aclaración: DES = Destino, FTE = Fuente, (I) = Índice, Ídem = Igual*

Las **INSTRUCCIONES** se dividen en *6 tipos*

Transferencia de Datos

- lb DES, FTE(I) (Copia en DES, un byte [8 BITS] desde la dirección FTE+I, CON SIGNO)

- lbu DES, FTE(I) (Ídem a “lb” pero SIN SIGNO)

- sb FTE, DES(I) (Guarda los 8 bits menos significativos de FTE a DES+I)

- lh DES, FTE(I) (Copia en DES, un half-word [16 BITS], desde la dirección FTE+I, CON SIGNO)

- lhu DES, FTE(I) (Ídem a “lh” pero SIN SIGNO)

- sh FTE, DES(I) (Ídem a "sb" pero con 16 bits)

- lw DES, FTE(I) (Copia en DES, un word [32 BITS], desde la dirección FTE+I, CON SIGNO)

- lwu DES, FTE(I) (Ídem a “lw” pero SIN SIGNO)

- sw FTE, DES(I) (Ídem "sb" pero con 32 bits)

- ld DES, FTE(I) (Copia en DES, un double word [64 BITS], desde la dirección FTE+I)

- sd FTE, DES(I) (Ídem "sb" pero con 64 bits)

- l.d DES, FTE(I) (Copia en DES, un punto flotante [64 BITS], desde la dirección FTE+I)

- s.d FTE, DES(I) (Ídem "sb" pero con punto flotante)

- mov.d DES, FTE (Copia el valor del registro FTE al registro DES)

- mtc1 FTE, DES (Copia 64 bits del registro ENTERO FTE al registro FLOTANTE DES) -> INT FTE to FLOAT DES

- mfc1 FTE, DES (Copia 64 bits del registro FLOTANTE FTE al registro ENTERO DES) -> FLOAT FTE to INT DES

- cvt.d.l DES, FTE (Convierte a un FLOTANTE el valor ENTERO de FTE hacia DES) -> INT FTE into FLOAT to DES

- cvt.l.d DES, FTE (Convierte a un ENTERO el valor FLOTANTE de FTE hacia DES) -> FLOAT FTE into INT to DES

Aclaración: DES = Destino, OP1 = Operando 1, OP2 = Operando 2, INM = Inmediato, (±) = Positivo o Negativo

Instrucciones Aritméticas

**Suma**

- dadd DES, OP1, OP2 (Suma OP1 con OP2 y lo deja en DES, valores CON SIGNO) des = (±) op1 + (±) op2

- daddi DES, OP1, INM (Suma OP1 con Valor Inmediato y lo deja en DES, valores CON SIGNO) des = (±) op1 + (±) Inmediato

- daddu DES, OP1, OP2 (Suma OP1 con OP2 y lo deja en DES, valores SIN SIGNO) des = op1 + op2

- daddui DES, OP1, INM (Suma OP1 con Valor Inmediato y lo deja en DES, valores SIN SIGNO) des = op1 + Inmediato

- add.d DES, OP1, OP2 (Suma OP1 con OP2 y lo deja en DES, PUNTO FLOTANTE) des = op1 + op2 (donde op es un número con coma)

**Resta**

- dsub DES, OP1, OP2 (Resta OP2 con OP1 y lo deja en DES, valores CON SIGNO) des = (±) op2 - (±) op1

- dsubu DES, OP1, OP2 (Resta OP2 con OP1 y lo deja en DES, valores SIN SIGNO) des = op2 - op1

- sub.d DES, OP1, OP2 (Resta OP2 con OP1 y lo deja en DES, PUNTO FLOTANTE) des = op2 - op1 (donde op es un número con coma)

**Multiplicación**

- dmul DES, OP1, OP2 (Multiplica OP1 con OP2 y lo deja en DES, valores CON SIGNO) des = (±) op1 \* (±) op2

- dmulu DES, OP1, OP2 (Multiplica OP1 con OP2 y lo deja en DES, valores SIN SIGNO) des = op1 \* op2

- mul.d DES, OP1, OP2 (Multiplica OP1 con OP2 y lo deja en DES, PUNTO FLOTANTE) des = op1 \* op2 (donde op es un número con coma)

**División**

- ddiv DES, OP1, OP2 (Divide OP1 con OP2 y lo deja en DES, valores CON SIGNO) des = (±) op1 / (±) op2

- ddivu DES, OP1, OP2 (Divide OP1 con OP2 y lo deja en DES, valores SIN SIGNO) des = op1 / op2

- div.d DES, OP1, OP2 (Divide OP1 con OP2 y lo deja en DES, PUNTO FLOTANTE) des = op1 / op2 (donde op es un número con coma)

**Comparación (Son las nuevas condiciones de salto)**

- slt DES, OP1, OP2 (Compara OP1 con OP2 solo si OP1 < OP2 -> DES = 1, valores CON SIGNO) si (±) op1 menor (±) op2 entonces des = 1

- slti DES, OP1, INM (Compara OP1 con Valor Inmediato solo si OP1 < INM -> DES = 1, valores CON SIGNO) si (±) op1 menor (±) Inmediato entonces des = 1

- c.lt.d OP1, OP2 (Compara OP1 con OP2, solo si OP1 < OP2 -> flag FP = 1, PUNTO FLOTANTE) si OP1 menor OP2 entonces FP = 1 (donde op son números con coma)

- c.le.d OP1, OP2 (Ídem, solo si OP1 <= OP2 -> flag FP = 1, PUNTO FLOTANTE) si OP1 menor o igual OP2 entonces FP = 1 (donde op son números con coma)

- c.eq.d OP1, OP2 (Ídem, solo si OP1 = OP2 -> flag FP = 1, PUNTO FLOTANTE) si OP1 igual OP2 entonces FP = 1 (donde op son números con coma)

Instrucciones Lógicas

- and DES, OP1, OP2 (Realiza un AND entre OP1 y OP2 bit a bit y lo deja en DES) des = nbits AND nbits

- andi DES, OP1, INM (Ídem, pero con un valor inmediato) des = nbits AND nbits inmediatos

- or DES, OP1, OP2 (Realiza un OR entre OP1 y OP2 bit a bit y lo deja en DES)

- ori (Ídem a "or" pero con un valor inmediato)

- xor DES, OP1, OP2 (Realiza un XOR entre OP1 y OP2 bit a bit y lo deja en DES)

- xori DES, OP1, OP2 (Ídem a "xor" pero con un valor inmediato)

Instrucciones de desplazamiento de bits

- dsll DES, FTE, N (Desplaza a la izquierda N veces de FTE y lo deja en DES)

- dsllv DES, FTE, OP2 (Desplaza a la izquierda OP2 veces de FTE y lo deja en DES)

- dsrl DES, FTE, N (Desplaza a la derecha N veces de FTE y lo deja en DES)

- dsrlv DES, FTE, OP2 (Desplaza a la derecha OP2 veces de FTE y lo deja en DES)

- dsra DES, FTE, N (Ídem “dsrl” pero mantiene el signo del valor desplazado)

- dsrav DES, FTE, N (Ídem “dsrlv” pero mantiene el signo del valor desplazado)

Instrucciones de Transferencia de Control

- j OffN (Salta a la dirección rotulada offN, offN es una ETIQUETA)

- jal OffN (Salta a la dirección rotulada offN, offN es una ETIQUETA y copia en R31 la dirección de retorno)

- jr DES (Salta a la dirección contenida en DES)

- beq OP1, OP2, offN (Si OP1 = OP2, salta a la dirección rotulada offN, offN es una ETIQUETA)

- bne OP1, OP2, offN (Si OP1 ≠ OP2, salta a la dirección rotulada offN, offN es una ETIQUETA)

- beqz OP1, offN (Si OP1 = 0, salta a la dirección rotulada offN, offN es una ETIQUETA)

- bnez OP1, offN (Si OP1 ≠ 0, salta a la dirección rotulada offN, offN es una ETIQUETA)

- bc1f offN (Salta a la dirección rotulada offN si flag FP=0 (false) en PUNTO FLOTANTE)

- bc1t offN (Salta a la dirección rotulada offN si flag FP=1 (true) en PUNTO FLOTANTE)

Instrucciones de Control

- nop (Operación nula)

- halt (Detiene el simulador)

# 3\_ ATASCOS Y CONFIGURACIONES

*Los ATASCOS se pueden dar en situaciones que impiden a la siguiente instrucción que se ejecute en el ciclo que le corresponde*

Los ATASCOS se dividen en 3

**Estructurales**

Provocados por conflicto de recursos

**Dependencia de control**

Se producen al esperar la decisión de otra instrucción anterior (Por ejemplo: si se realiza un salto o no)

**Dependencia de datos**

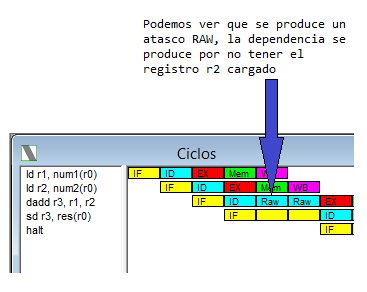
**RAW:** Read After Write

**WAR:** Write After Read

**WAW:** Write After Write

RAW

Se produce cuando se está leyendo un dato que todavía está en la etapa de escritura, es un atasco de **DEPENDENCIA DE DATOS**, al igual que **WAR** y **WAW**



Analizando el siguiente código obtenemos lo siguiente:

.data

num1: .*word* 10

num2: .*word* 15

res: .*word* 0

.code

ld r1, num1(r0) *; CARGO en r1 lo que vale num1(r0), donde r0 me indica el índice*

ld r2, num2(r0) *; CARGO en r2 lo que vale num2(r0), donde r0 me indica el índice*

dadd r3, r1, r2 *; SUMO lo que vale r1 con r2 y lo dejo en r3*

sd r3, res(r0) *; GUARDO lo que hay en r3 y lo dejo en res(r0), donde r0 nuevamente, me indica el índice*

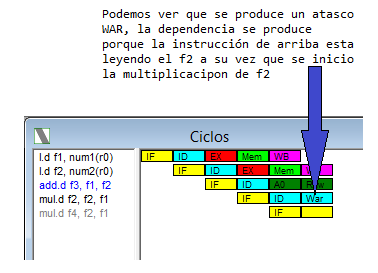
halt *; Instrucción que finaliza el programa*

Ahora bien, pasemos a la explicación paso por paso de porque se produce un **RAW** en la suma:

Cuando yo quiero sumar 2 registros, en esos registros ya debe estar cargado el dato a sumar, en **r1** yo ya tengo el dato a sumar, **r1 = 10**, pero en r2 **todavía** no tengo el 15, como aparece en la imagen, luego de que se termine la etapa WB puedo realizar la instrucción de suma, ahora ¿Qué pasa en la siguiente instrucción? “sd r3, res (r0)”, como r3 **todavía** no tiene el resultado de la suma, se vuelve a producir otro atasco RAW (teóricamente lo solucionaríamos con la configuración **FORWARDING**)

WAR

Se produce cuando se está **ESCRIBIENDO** un dato que se está **LEYENDO** por otra instrucción, es posible que este atasco se produzca mayormente en **PUNTO FLOTANTE**



Analizando el siguiente código obtenemos lo siguiente:

.data

Num1: .*word* 8.5

Num2: .*word* 2.5

Res: .*word* 0

.code

l.d f1, num1 (r0)*; CARGO en f1 lo que vale Num1 (r0)*

l.d f2, num2 (r0)*; CARGO en f2 lo que vale Num2 (r0)*

add.d f3, f1, f2*; SUMO lo que vale f1 con f2 y lo dejo en f3 en PUNTO FLOTANTE*

mul.d f2, f2, f1*; MULTIPLICO lo que vale f2 con f1 y lo dejo en f2 en PUNTO FLOTANTE*

mul.d f4, f2, f1*; MULTIPLICO lo que vale f2 con f1 y lo dejo en f4 en PUNTO FLOTANTE*

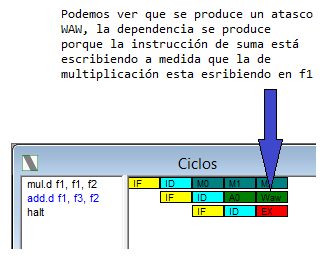
s.d f3, res (r0)*; GUARDO lo que hay en f3 y lo dejo en Res (r0)*

halt*; Instrucción que finaliza el programa*

El WAR (Write After Read) se produce cuando la instrucción que **ESTA LEYENDO** un dato, a su vez está siendo **MODIFICADO**, por lo tanto, la instrucción para por un momento su ejecución con este atasco y permite que la instrucción anterior pueda ejecutarse sin problemas.

WAW

Se produce cuando se está **ESCRIBIENDO** un dato que se está **ESCRIBIENDO**, lo que produce un atasco WAW (Write After Write)



Analizando el siguiente código obtenemos lo siguiente:

.data

.code

mul.d f1, f1, f2*; MULTIPLICO f1 con f2 y lo dejo en f1 (PUNTO FLOTANTE)*

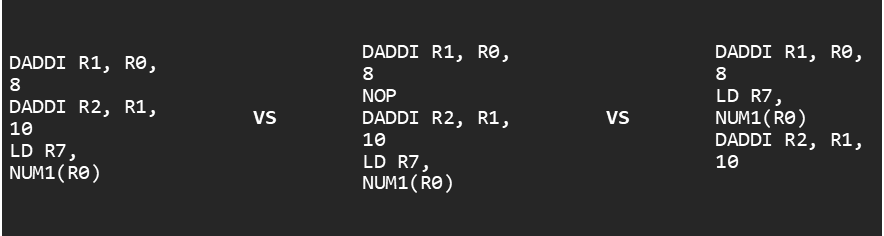
add.d f1, f3, f2*; SUMO f3 con f2 y lo dejo en f1 (PUNTO FLOTANTE)*

halt*; Instrucción de fin*

Tanto la instrucción “mul.d” y “add.d” tienen más o menos ciclos (la multiplicación se tarda más que la suma), en este caso, la suma tendría que terminar **ANTES** que la multiplicación, entonces, se sobrescribe el valor de f1 con el de la **suma** y se lo pasaría a la **multiplicación,** PERO, en este caso se produce un atasco **WAW** lo que permite que se ejecute *primero* la multiplicación y así ejecutar en orden correcto las instrucciones

**¿Cómo solucionar ATASCOS?**

**Por software**



**Con NOPs Ordenando sentencias**

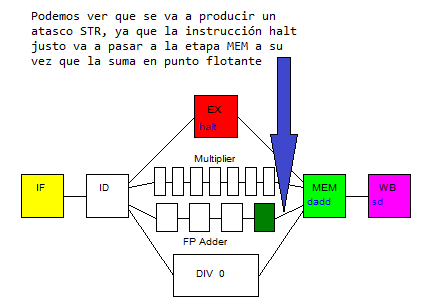
**Por hardware**

Si ya tenemos los valores necesarios, podemos “adelantarlos” con la opción de Forwarding.

**Atascos – Estructurales**

Los atascos **estructurales** son provocados por conflictos por los recursos

En MIPS, sucede cuando dos instrucciones intentan acceder a la etapa **MEM** simultáneamente



Cuando dos instrucciones van a parar a la misma etapa, se produce un atasco de tipo STR, lo que permite que una instrucción se ejecute y que **LUEGO** pase la siguiente instrucción

Vemos que donde está la flecha apunta a la etapa EX del cauce “FP Adder” que sería la suma en punto flotante, cuando termina de sumar, también termina de ejecutar el “halt”, y se produce un atasco STR para evitar que ambas se encuentren en la misma etapa

**Atascos – Dependencia de control**

Tenemos dos tipos de saltos

**Incondicionales:** Salta siempre

**Condicionales:** Salta dependiendo de que se cumpla una condición

**Configuraciones**

**Forwarding**

**BTB** (Branch Target Buffer)

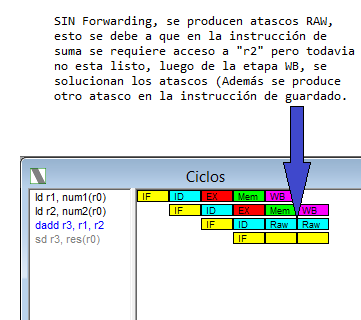
**Delay Slot**

Forwarding

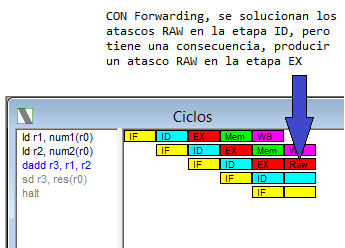
Ahora tendremos dos buffers entre antes y despues de la etapa MEM, se almacenan los valores para que los puedan usar en las próximas instrucciones

De esta manera no hace falta esperar a las etapas **MEM** y **WB** para usar los valores, a esta configuración se le llama Forwarding, además con esta opción habilitada, podremos postergar la “necesidad” de los operandos.

SIN FORWARDING



CON FORWARDING



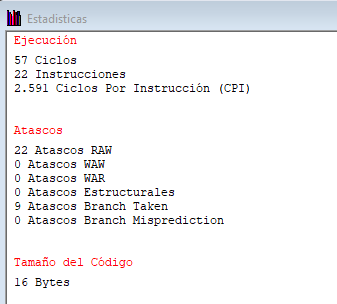
Branch Target Buffer

Al igual que con los atascos de dependencia de datos, tenemos diferentes técnicas para evitar los atascos por saltos

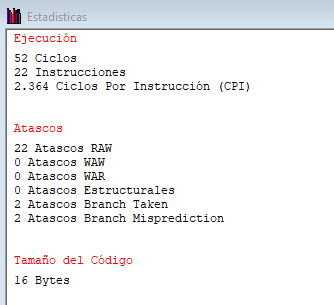
Consiste en tener un flag que indica si debe saltar incondicionalmente o no dependiendo de qué hizo antes (es decir, predice)

Cada vez que ese flag/buffer se actualiza cuenta como un *atasco de salto*, cada vez que **erramos** en la predicción se produce un atasco

SIN BRANCH TARGET BUFFER



CON BRANCH TARGET BUFFER

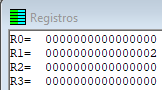


BTB, nos ayuda a reducir potencialmente los atascos de tipo Branch Taken, que se producen cuando una instrucción entra al cauce pero se **pierde**, entonces, se podria decir que, sin BTB se van a producir muchos atascos Branch Taken porque cada que se ejecuta una instrucción de salto, siempre se va a perder la instrucción que le sigue. Con la configuración de BTB habilitada, nos va a permitir tener un “buffer”, o flag, que se actualiza cada vez que entra al salto y cada que sale, produciendo únicamente como se ve en la imagen, 2 atascos de tipo Branch Taken y **Branch Misprediction**, este atasco significa que el buffer/flag se equivocó, y va a suceder siempre 2 veces, cuando entre la primera vez y al salir, la primera vez se equivoca porque pregunta si debe saltar, y la flag esta seteada en “no saltar”, entonces corrige ese error actualizando la flag a “si saltar” e itera hasta que se **no** se cumpla la condición de salto, en este caso vuelve a equivocarse y corrige nuevamente el error seteando la flag a “no saltes” y sale del bucle, volviendo a lo de antes, con BTB activado, se van a producir menos atascos de saltos

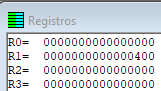
Delay Slot

Consiste en ejecutar **SIEMPRE** la siguiente instrucción a un salto, el problema de esta opción es cuando tenemos un salto y luego una instrucción “halt”, lo que produce es que se ejecute el “halt” antes que el salto y termine el programa, usarlo sabiamente

SIN DELAY SLOT



CON DELAY SLOT



Yo tengo el siguiente código muy simple:

.data

.code

daddi r1, r0, 1

daddi r2, r0, 10

loop: daddi r2, r2, -1

bnez r2, loop

dadd r1, r1, r1

halt

El algoritmo es el siguiente:

R1 = 1

R2 = 10

Un for de 10 iteraciones

Pregunta si r2 ≠ 0, si se cumple vuelve a iterar

R1 = R1 + R1

Termina

Sin delay slot, ejecuta 1 sola vez la suma cuando termina de iterar

Con delay slot, ejecuta por cada bucle la instrucción de suma, de ahí es que cambia el resultado **sin** delay slot de R1 = 2 a **con** delay slot de R1 = 400

Y si yo elimino la instrucción de suma, se ejecutaria el **halt** y terminaria el programa antes de iniciar el bucle y yo **no** quiero que suceda eso, por eso hay que usar con cuidado la configuración **delay slot**

# 4\_ PUNTO FLOTANTE

MIPS utiliza IEEE 754 para números en punto flotante (en el caso de los registros R, utiliza BSS o CA2 para números con signo)

Contamos con 32 registros: Desde el F0 (siempre vale 0) hasta el F31 (F0 – F31) igual que con los registros de tipo R (R0 – R31)

Para declarar un tipo flotante se utiliza el “.double” (antes usábamos .word y asi)

Ejemplo:

.data

A: .double 8.10

.code

...

Ejemplos de instrucciones

*Nota: Recuerden que las posiciones siempre son enteras, yo no puedo acceder a la posición 3,5, súper ilegal eso.*

- Carga L.D F1, NUM1(R0) (Load.Double de F1 en NUM1 (0))

- Suma      ADD.D F1, F2, F3 (Add.Double de F2 + F3 en F1)

- Resta SUB.D F1, F2, F3 (Sub.Double de F3 - F2 en F1)

- Multiplicación    MUL.D F1, F2, F3 (Mul.Double de F2 \* F3 en F1)

- División      DIV.D F1, F2, F3 (Div.Double de F2 / F3 en F1)

- Almacenamiento    S.D F1, RES(R0) (Store.Double de F1 en RES (0))

Ejemplo de un código en PUNTO FLOTANTE:

RES = (A+B)\*C (¿Cómo implementamos esto en código?)

.data

A: .double 5.5

B: .double 8.6

C: .double 10.0

RES: .double 0.0

.code

L.D F1, A(R0)

L.D F2, B(R0)

L.D F3, C(R0)

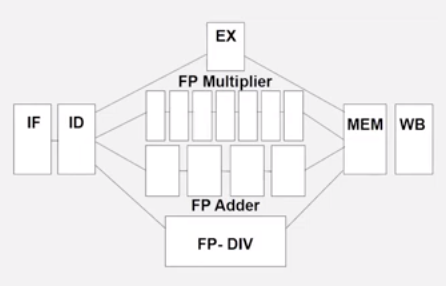
ADD.D F1, F1, F2

MUL.D F1, F1, F3

S.D F1, RES(R0)

HALT

Recordar que **no todas las etapas duran lo mismo**



**Etapa EX = 1 ciclo**

**Sumar en Pto. F. = 4 ciclos**

**Multiplicar en Pto. F. = 7 ciclos**

**Dividir en Pto. F. = 24 ciclos**

# 5\_ USO DE REGISTROS (Convenciones)

R0 ($zero) = Siempre tiene el valor 0 y no se puede cambiar

R30 ($fp) = Frame Pointer – Puntero de pila. Debe ser salvado

R31 ($ra) = RETURN ADDRESS – Dirección de retorno de subrutina. Deben ser salvados

R2 – R3 ($v0 - $v1) = Valores de retorno de la subrutina llamada

R4 – R7 ($a0 - $a3) = Argumentos pasados a la subrutina llamada

R8 – R15 ($t0 – $t7) y R24 – R25 ($t8 - $t9) = Variables locales

R16 – R23 ($s0 – $s7) = Variables locales. Deben ser salvados

R29 ($sp) = Stack Pointer – Puntero al tope de la pila. Deben ser salvados

# 6\_ SUBRUTINAS

Las subrutinas pueden ya no se invocan con **CALL**, ahora utilizaremos la instrucción **JAL**

jal etiqueta

Y para volver **RET**, que ahora se llama **JR** ($ra = return address en el r31)

jr $ra

Ejemplo de una subrutina común y corriente:

.data

NUM1: .*word* 5

NUM2: .*word* 8

RES: .*word* 0

.code

ld $a0, NUM1 (R0)

ld $a1, NUM2 (R0)

jal SUMAR*; Llamo la subrutina*

sd $v0, RES (R0)

halt

SUMAR: DADD $v0, $a0, $a1

jr $ra*; Retorna a la próxima instrucción*

Ejemplo de subrutina que suma 2 números pasados por pila dentro de una subrutina y lo guarda en una variable:

.data

A: .*word* 10

B: .*word* 20

resul: .*word* 0

.code

daddi $sp, $0, 0x400 *; $sp = 0x400 (EL tope de la pila)*

ld $t0, A($0)

ld $t1, B($0)

daddi $sp, $sp, -8 *; Hago un push*

sd $t0, 0($sp) *; Guardo en la pila el $t0*

daddi $sp, $sp, -8 *; Hago otro push*

sd $t1, 0($sp) *; Guardo en la pila el $t1*

jal subr *; jal = CALL*

sd $v0, resul($0) *; resul = $v0*

halt *; Finalizar el programa*

subr:   ld $t0, 8($sp) *; Cargo el valor A en t0*

    ld $t1, 0($sp) *; Cargo el valor B en t1*

    daddi $sp, $sp, 16 *; Posiciono el SP a 0x400*

    dadd $v0, $t0, $t1 *; $v0 = $t0 + $t1*

    jr $ra *; jr = RET*

Preservación de los registros

Las subrutinas deben garantizar el guardado de los registros que correspondan

De esta manera, una subrutina podrá llamar a otra sabiendo que esta no modificara el valor de estos registros

Para poder mantener esa garantía, es necesario guardar los registros en la **pila**, pero **MIPS no tiene pila**.

Pero existe un registro que por **convención** todas las subrutinas usaran como puntero al tope de la pila.

Usemos el registro $sp

Subrutinas anidadas

Llamar una subrutina dentro de otra subrutina es posible, pero antes de eso debemos guardar la dirección de retorno y/o registros que usemos que no queremos perder, ya que una vez que yo llame una subrutina, y se me modifica algo se pierde (Como la dirección de retorno)

También podemos reservar espacio de memoria (siempre de a Nx8) donde N son los registros a guardar, por ejemplo, siempre que usemos más de 1 subrutina, debemos guardar la $ra (1x8), y si tenemos registros que debemos guardar, haremos 1+N registros x 8, y ahí calculamos cuanto espacio debemos restarle a $sp para reservar espacio.

Ejemplo:

Guardo 2 registros de tipo ‘$s’

daddi $sp, $sp, -24 *; Reservo espacio (3x8 = 24)*

sd $ra, 0($sp) *; Apilo la dir. de retorno*

sd $s0, 8($sp) *; Apilo $s0*

sd $s1, 16($sp) *; Apilo $s1*

ld $ra, 0($sp) *; Restauro $ra*

ld $s0, 8($sp) *; Restauro $s0*

ld $s1, 16($sp) *; Restauro $s1*

daddi $sp, $sp, 24 *; Restauro la pila a su tope*

Por eso debemos apilar en $sp todo lo que no queremos perder, principalmente $ra

daddi $sp, $sp, -8

sd $ra, 0($sp)

ld $ra, 0($sp)

daddi $sp, $sp, 8

Para recuperar esos valores guardados y restaurar la pila, debemos aplicar la inversa de lo que hicimos para guardar los registros, es decir, cargamos lo apilado en el registro guardado, y restauramos la pila a su posición original

# 7\_ ENTRADA Y SALIDA

MIPS cuenta con dos “pantallas”

Una **alfanumérica**, donde se puede imprimir texto

Y otra **gráfica**, donde podemos pintar pixeles

Registros Entrada/Salida

Existen dos “registros” (es decir, dos celdas de memoria comunes)

**CONTROL**: Sirve para enviar códigos de operaciones (0x10000)

**DATA**: Sirve para enviar o recibir datos (0x10008)

Como son celdas de memoria se leen y escriben con instrucciones de memoria: LD/L.D/LBU/SD/S.D…

Ejemplo mandando algo a CONTROL, por ejemplo, el valor 1

.data

CONTROL: .*word* 0x10000

.code

ld $s0, CONTROL($zero)

daddi $t1, $zero, 1

sd $t1, 0($s0)

Pantalla – Controles

Tenemos varios tipos de datos en control:

*1* = Imprimir un entero **sin** signo (BSS)

*2* = Imprimir un entero **con** signo (CA2)

*3* = Imprimir un punto flotante (DOUBLE)

*4* = Imprimir un string (cadena de caracteres)

*5* = Pantalla gráfica

*6* = Limpiar la pantalla alfanumérica

*7* = Limpiar la pantalla gráfica

*8* = Leer un número

*9* = Leer un carácter

Pantalla – Alfanumérico

.data

CONTROL: .*word* 0x10000

DATA: .*word* 0x10008

.code

ld $s0, CONTROL($0) *; Cargo control en un registro*

ld $s1, DATA($0) *; Cargo data en un registro*

daddi $t0, $0, -85

sd $t0, 0($s1) *; Guardo en data, el valor de $t0*

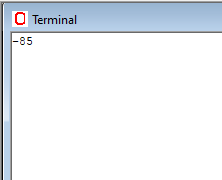
daddi $t0, $0, 2 *; Mando la señal (2), imprimir con signo*

sd $t0, 0($s0) *; Guardo en control, la señal*

daddi $t0, $0, 6 *; Mando la señal (6), limpiar la pantalla*

sd $t0, 0($s0) *; Guardo en control, la señal*

halt



Pantalla – Gráfica

En la pantalla gráfica el registro DATA cuenta con 8 bytes:

Los 3 primeros bytes son los colores R G B y el siguiente byte es la *transparencia*

Luego de esos 4 bytes, tenemos las coordenadas Y,X

Por lo que nos quedaría de esta forma el DATA:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| R | G | B | T | Y | X | - | - |

Ejemplo de la pantalla gráfica:

.data

PIXEL: .*byte* 0, 185, 135, 0, 23, 10, 0, 0

CONTROL: .*word* 0x10000

DATA: .*word* 0x10008

.code

ld $s0, CONTROL($0) *; Cargo control en un registro*

ld $s1, DATA($0) *; Cargo data en un registro*

ld $t0, PIXEL($0) *; Cargo en $t0 el pixel*

sd $t0, 0($s1) *; Guardo en data ese pixel*

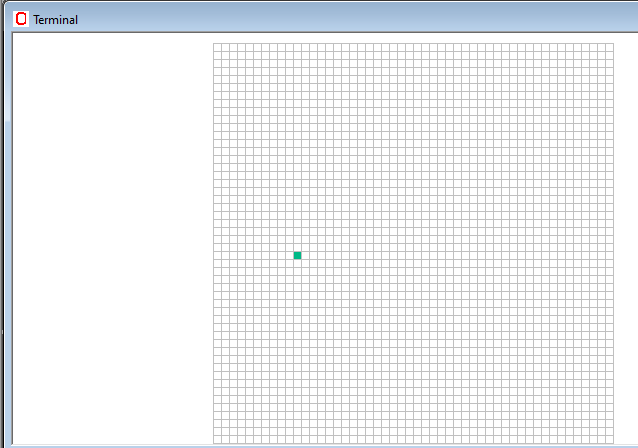
daddi $t0, $0, 5 *; Mando la señal (5), pantalla gráfica*

sd $t0, 0($s0) *; Guardo en control, la señal*

daddi $t0, $0, 7 *; Mando la señal (7), limpiar pantalla gráfica*

sd $t0, 0($s0) *; Guardo en control, la señal*

halt



# 8\_ PARCIAL

Los parciales cuentan con ejercicios parecidos al primer parcial de Arquitectura de Computadoras, el primer ejercicio será leer un código y calcular X tipos de ATASCOS y CPI (Ciclos por Instrucción)

Entre los siguientes ejercicios, podrías encontrarte con “Y si tuviese tal configuración habilitada, ¿cuántos atascos de X tipo se producen?” o “Reordenar las instrucciones de tal manera que no se produzcan atascos de X tipo”

Siguiendo, también puede haber preguntas de tipo “En que etapa se realiza tal acción” o “Explicar brevemente que función cumple X etapa”

*Ctrl + clic para seguir el vínculo de abajo:*

**1\_ INTRODUCCIÓN A MIPS**

Luego, completar instrucciones para realizar una acción mediante Entrada y Salida y/o completar líneas de código como en el primer parcial

Finalmente, el ejercicio que más puntos valen son los que vas a tener que armar un código que resuelva el enunciado dado

Para calcular el CPI, existen 3 maneras:

CPI = Ciclos / Instrucción

CPI\* = Instrucciones + 4 / Instrucciones

CPI Real = Instrucciones + 4 + Atascos / Instrucciones

Ejemplo para calcular el CPI:

.data

.code

daddi r1, r0, 3

loop: dadd r3, r3, r2

daddi r1, r1, -1

bnez r1, loop

halt

Analicemos el siguiente código instrucción a instrucción:

daddi r1, r0, 3 ; Se carga en r1 el valor “3”

loop: dadd r3, r3, r2 ; Guarda en r3 la suma de r3 y r2

daddi r1, r1, -1 ; Decrementa r1 en 1

bnez r1, loop ; Si r1 no es 0, salta a loop

halt ; Finaliza el programa

Vemos que es un for de 3 iteraciones, sin activar ninguna configuración, contemos la cantidad de atascos que hay:



Vemos que, al principio, recién en la etapa WB tengo el valor cargado en r1, pero a medida que se va ejecutando el programa, llego a la instrucción que decrementa el registro, y este no puede porque todavía no lo tiene cargado, así que tenemos un atasco RAW, luego pasamos a preguntar si r1 es distinto de 0, pero esto no lo se porque en la instrucción que decrementa tampoco tengo el resultado (lo vuelvo a obtener en WB) así que se producen 2 atascos RAW. Estos 2 atascos se producen por cada iteración, y como ya tengo en r1 guardado el valor al principio, no vuelve a ocurrir el primer RAW al decrementar el registro.

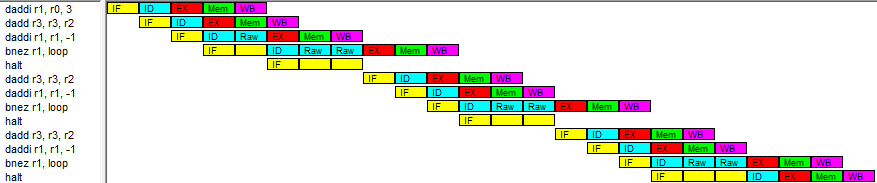


Atascos: 1 RAW al principio, 2 RAW por iteración (x3), entonces nos queda 1 + 6 RAW = 7 RAW

Para calcular los atascos de Salto, tendríamos que armar el pipeline a mano y ver cuando se produce el salto, recordemos que en la etapa ID verifica eso, ósea que mientras está viendo si tiene el registro que me da la condición de salto, la instrucción por siguiente al salto se va a empezar a ejecutar (halt) y luego de saber si va a saltar, corta la instrucción y produce un atasco, esto lo hace al principio, en la segunda iteración, y en la tercera NO, por ende, podríamos decir que la cantidad de atascos Branch Taken va a ser N-1 veces (3-1 = 2).

Para calcular ciclos por instrucción, tomamos la cuenta de “Ciclos = I + 4 + A”

Tenemos que contar cuantas instrucciones hay, ahora sí, dibujando el pipeline:



1 daddi r1, r0, 3

3 daddi r3, r3, r2

3 daddi r1, r1, -1

3 bnez r1, loop

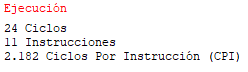
1 halt

11 instrucciones + 4 + Atascos

Cantidad de atascos totales: 7 RAW + 2 Branch Taken = 9 Atascos

11 + 4 + 9 = 24 Ciclos

Ciclos por Instrucción = 24/11 = 2,182 (No hace falta hacer este cálculo)



Como último mensaje, si es que llegaste hasta esta parte, recomiendo muchísimo mirar los dos últimos videos de Genaro Camele que explica un parcial, vuelvo a decir, esta guía está basada en teoría, práctica y videos de él, si a alguno de ustedes les sirve este manual para el parcial, no pido colaboraciones ni nada, simplemente que si el día de mañana ustedes tienen apuntes que quieran compartir conmigo bienvenidos son.

Sin más que decir, éxitos el martes y en el recuperatorio <3.

*Santiago Gabriel Rojas Zapata*