El repertorio de instrucciones de los procesadores MIPS: registros básicos y codificación

Escuela Técnica Superior de Ingeniería Informática
Universidad de La Laguna

Marzo 1, 2023

Esquema de la lección

Historia de las implementaciones (1/2)

- Cuando la memoria era poca y cara, era importante que los programas fueran tan pequeños como fuera posible. Dado que había que escatimar hasta el último byte, las máquinas tenían un número variable de bytes para las instrucciones. Para hacernos una idea, el Intel 8086 tenían solamente 64KB de memoria. Es decir, 65536 bytes, y sus instrucciones ocupaban entre 1 y 5 bytes.
- Otra aproximación fue eliminar los registros, dado que se pensaba que los compiladores no los podrían gestionar eficientemente. Eso hizo que compañías como Hewlet Packard se basaran en un modelo de pila de ejecución. Los operandos se introducían en la pila desde memoria y viceversa. Las operaciones guardaban sus resultados en la pila.

Historia de las implementaciones (2/2)

- En los años sesenta los sistemas operativos se escribían en lenguaje ensamblador, lo que era una gran tarea. El UNIX fue escrito originalmente en ensamblador pero en 1973, la versión 4 fue reescrita en C. Esto llevó a que se quisieran realizar máquinas con arquitectura de alto nivel. Es decir, que las instrucciones de hardware fueran lo más parecida posibles a los lenguajes de programación.
- Esta filosofía de diseño fue sustituida en los años ochenta por RISC, que significa Computador de repertorio de instrucciones reducido (Reduced Instruction Set Computer)

¿Porqué aprender ensamblador? (Hardware)

- Al contrario que los lenguajes de alto nivel que son independientes de la máquina, el ensambaldor está estrechamente relacionado con la arquitectura de la misma. Por tanto, conocer el ensamblador nos permite comprender cómo funciona un procesador a bajo nivel.
- Tan importante es que se considera como uno de los niveles de descripción de la máquina. Es necesario conocer tanto el formato de las instrucciones a bajo nivel (cuantos bits tiene cada instrucción y cómo se organizan), así como los modos de direccionamiento (cómo acceder a los datos) y las instrucciones de comparación de condiciones y de salto que nos permiten implementar las estructuras de control.

¿Qué ensamblador enseñar

- Si queremos programar una máquina concreta debemos aprender a programar el ensamblador de la misma. Si es un PC, debemos aprender el ensamblador del 8086, pero si es queremos programar una GPU Nvidia a bajo nivel tenemos que aprender el Parallel Thread Execution ISA (PTX).
- Si lo que queremos es aprender los conceptos básicos de ensamblador es mejor elegir uno que sea lo más regular posible. Por ejemplo que tenga todas las instrucciones del mismo tamaño y que no tenga demasiadas instrucciones. Además, debemos pedirle que tenga un entorno de desarrollo lo más amigable posible. Debido a todas estas razones hemos elegido el ensamblador MIPS para su explicación en esta asignatura.

Características básicas de los procesadores MIPS

- La familia de procesadores MIPS (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages responde a un diseño de principios de los años 80. Fueron utilizados en productos de NEC, Nintend, Silicon Graphics, Sony, etcétera.
- El diseño del procesador MIPS tiene una serie de características básicas que afectan al repertorio de instrucciones:
  - El juego de instrucciones es reducido. Se trata de un procesador RISC (**R**educed **I**nstruction **S**et **C**omputer).
  - El número de operandos por instrucción es el mismo dentro de la misma clase (regularidad en el formato de las instrucciones).
  - Es un máquina de carga - almacenamiento, esto es, todos los datos con los que hay que calcular deben pasar por los registros del procesador.
  - Dispone de un gran número de registros.
  - El tamaño de palabra, en la versión de MIPS que vamos a estudiar, es 32 bits.

Juego o repertorio de instrucciones - definición

- El juego de instrucciones de un procesador consiste en la lista de instrucciones que el procesador puede ejecutar, así como como los tipos de datos que puede manipular, los registros y modos de direccionamiento que permite y algunas otras características que ser irán viendo en la asignatura.

Juego de instrucciones - Lenguaje máquina y ensamblador

- Para el procesador cada instrucción es un código en binario. El conjunto de todos ellos es lo que se conoce como Lenguaje Máquina. Cada código binario junto a otras entradas de la unidad de control activa una secuencia de señales de control. Cada señal de control hace actuar a un componente del procesador (por ejemplo, volcar el contenido de un registro en un bus interno). Cuando se termina la secuencia de activación de señales de control asociada a una instrucción podemos decir que la instrucción se ha ejecutado.
- El lenguaje ensamblador por el contrario está dirigido al programador. Aunque en los principios los programadores tenían que desarrollar sus programas en lenguaje máquina con código en binario, la aparición del primer ensamblador en 1947 cambió el panorama. En la mayoría de las ocasiones cada instrucción del ensamblador de un procesador tiene su contrapartida en una instrucción del repertorio, pero esto no siempre es así.

El pipeline MIPS (I)

¿Qué es un pipeline?. Ejemplo de "See Mips Run": sirviendo fish and chips

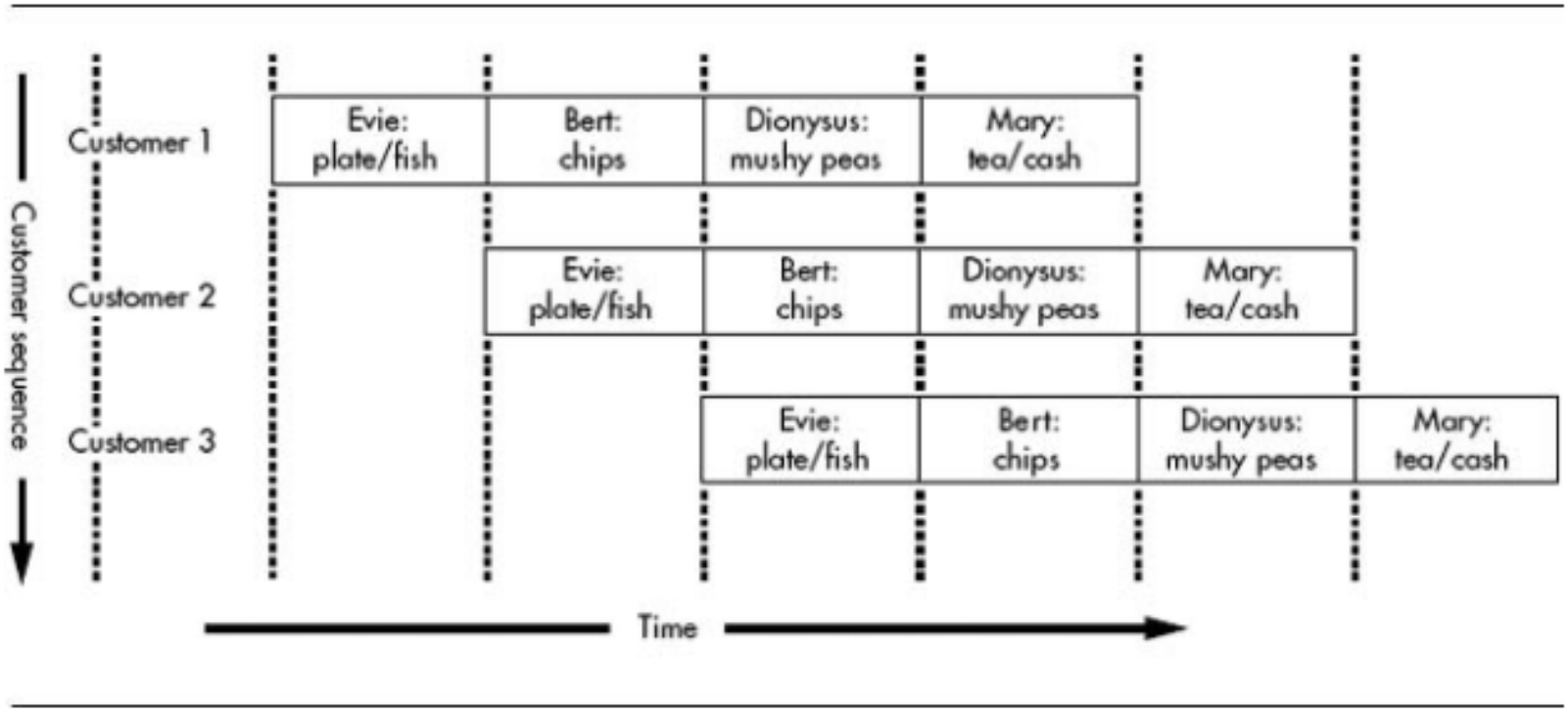

FIGURE 1.1 Evie's fish and chip shop pipeline.

Figura: Un pipeline de fish and chips

El pipeline MIPS (II)

El pipeline de MIPS tiene cinco etapas. Este pipeline determina las características del juego de instrucciones.

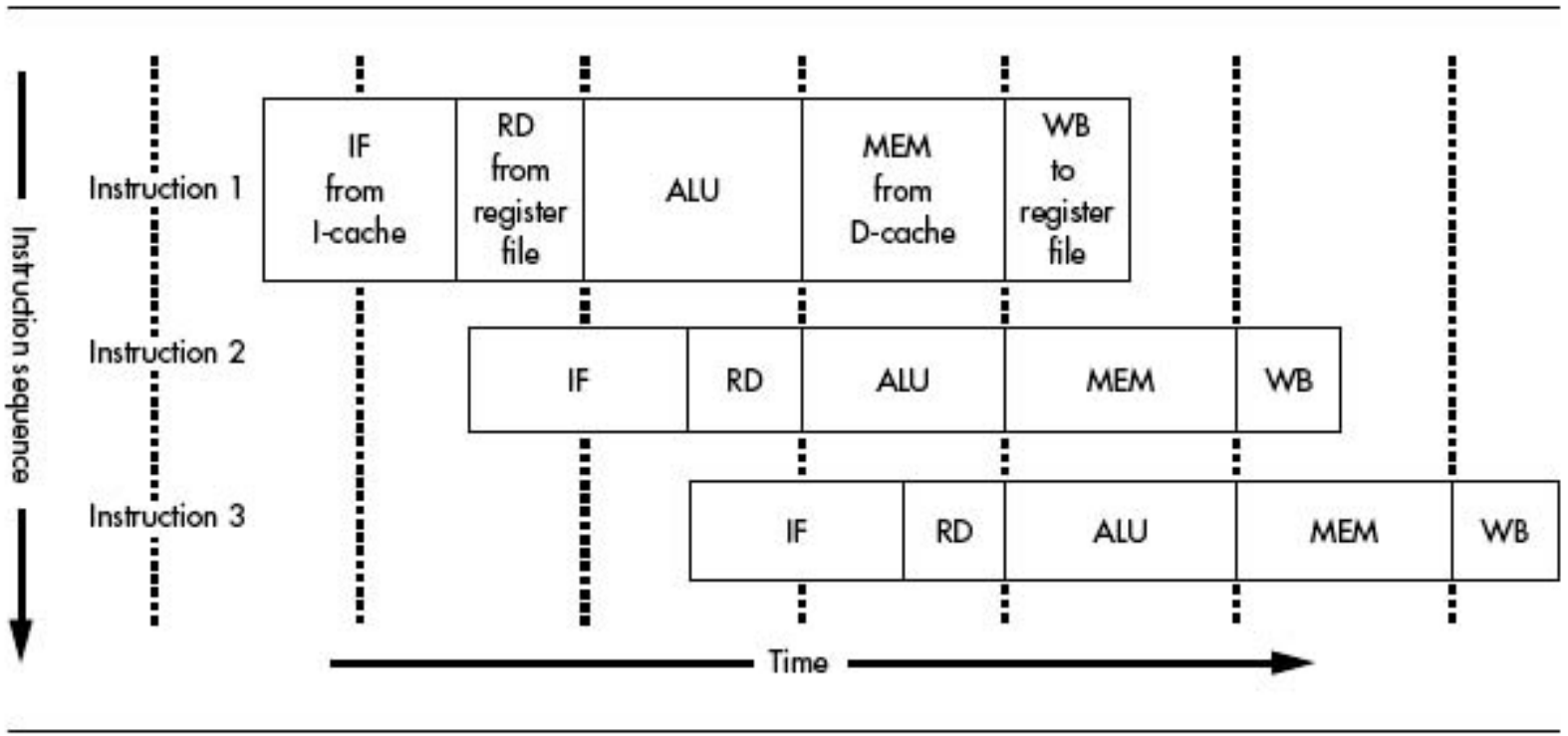

FIGURE 1.2 MIPS five-stage pipeline.

Figura: Pipeline básico de MIPS (asumimos caché de instrucciones y de

El pipeline MIPS (III)

Etapas del pipeline MIPS

- **IF o Instruction Fetch**. Toma la siguiente instrucción del caché de instrucciones.
- **ID o Instruction Decoding**. Decodifica la instrucción, lee los registros cuyos números están codificados como fuentes en la instrucción.
- **ALU o Aritmethic Logic Unit Step**. Realiza operaciones aritméticas y lógicas con los operandos utilizando la circuitería de la CPU.
- **MEM o Read/Write Memory Variables**. Leer o escribir en la caché de datos. Operaciones de **carga / almacenamiento**.
- **WB o Write back**. Escribe el resultado de la operación en el registro establecido.

Ventajas del pipeline

- Se reduce el tiempo dedicado a procesar un conjunto de instrucciones.
- Pero sobre todo se reduce el tiempo entre instrucciones procesadas: aumenta el throughput

10ns	10ns	10ns	10ns	10ns	10ns	10ns
IF	ID	ALU	MEM	WB		
	IF	ID	ALU	MEM	WB	
		IF	ID	ALU	MEM	WB

Si las instrucciones tuvieran que esperar unas por otras, cada una tardaría aproximadamente 40ns y el conjunto 120ns, y el tiempo entre instrucciones sería 40ns.
Así, el conjunto tarda 70ns, y el tiempo entre instrucciones procesadas es 10ns.

Registros del pipeline

- Cada etapa no puede volcar directamente la información en la entrada de la siguiente etapa. Si lo hicieran, se sobreescribiría la informaci'on con la que trabaja la siguiente etapa.

10ns	10ns	10ns	10ns	10ns	10ns	10ns
IF	ID	ALU	MEM	WB		
	IF	ID	ALU	MEM	WB	
		IF	ID	ALU	MEM	WB
		ID vs ALU				

- La solución pasa por introducir registros intermedios inter-etapas que guarden la información que est'a procesando la siguiente etapa: IF/ID, ID/ALU, ALU/MEM, MEM/WB

Data hazards

Observen el siguiente par de instrucciones:

```
add $t1, $t2, $t3
add $a0, $t1, $t4
```

IF(add)	ID(t2,t3)	ALU	MEM	WB (t1)		
	IF (add)	ID (t1,t4) X	ALU	MEM	WB (a0)	

En la etapa marcada con una X (ID) se precisa t1 actualizado, pero este registro aún no ha sido calculado por la instrucción anterior. Esto es lo que se conoce como un data hazard.

Soluciones a los data hazards

- Por software: insertar instrucciones **NOP** entre las dos instrucciones para dar tiempo a que se calcule el registro.
- Por hardware: detectar la dependencia y por hardware no permitir que la instrucción salga de la etapa primera hasta que se den las condiciones. Esto se denomina **stall**
- **Data forwarding**. En el ejemplo anterior, el registro ALU/MEM ya contiene el valor correcto del registro t1, puede así usarse directamente como entrada de la alu.

Data hazard en la carga

Observen el siguiente par de instrucciones:

```
lw $t1, 100($s0)
add $a0, $t1, $t2
```

- Vemos el data hazard como antes, pero ahora el data fowarding no funciona porque el valor no estará en el registro MEM/WB cuando hace falta

IF(lw)	ID	ALU	MEM	WB (t1)	
	IF (add)	ID (t1,t2)	ALU X	MEM	WB (a0)

- Por lo tanto en el caso de las instrucciones de carga se necesita hacer el stall o introducir nop. Si se introduce un stall es un ciclo adicional para las instrucciones de carga.

Branch Hazard

- En una instrucción de salto puede que no sea la siguiente instrucción la que tenga que entrar en el pipeline. Es lo que se conoce como branch hazard.
- Una forma de resolverlo es incluir hardware específico para determinar el salto correcto en la etapa de ID y entonces introducir en el pipeline la instrucción correcta.
- Esto obliga a introducir después de la instrucción de salto un **NOP** o una instrucción que no tenga incidencia en las instrucciones posteriores del salto.

El conjunto de los registros (I)

- Recordemos que un registro es un elemento de almacenamiento básico para el procesador. Es una memoria más "rápida" que la memoria principal. Se suelen utilizar para diversos fines:
  - Controlar el funcionamiento del procesador reflejando características de su estado. Por ejemplo, el registro Program Counter (PC) que indica **la dirección en la memoria de la siguiente instrucción a ser ejecutada**.
  - Operandos implícitos de las instrucciones.
  - Operandos explícitos de las instrucciones.
- En MIPS el tamaño de cada registro es 32 bits.
- MIPS dispone de un total de 32 registros de uso ordinario y 32 registros para punto flotante.

El conjunto de los registros (II)

Número	Nombre	Descripción
0	$zero	Valor constante cero
1	$at	Temporal reservado para el ensamblador. Se utiliza al traducir las pseudoinstrucciones.
2 – 3	$v0-$v1	Suelen usarse para devolver el resultado de las funciones.
4 . . . 7	$a0 . . . $a3	Suelen usarse como argumentos para las subrutinas. No debe esperarse que se preserve el contenido tras la llamada.
8 . . . 15	$t0 . . . $t7	Registros temporales. No debe esperarse que se preserven cuando se llama a una subrutina.
16 . . . 23	$s0 . . . $s7	Se denominan valores salvados. El programador de una subrutina debería "salvar" su valor antes de usarlos, para "restaurarlos" antes de la finalización.
24 . . . 25	$t8 y $t9	Continuación de los registros temporales.
26 – 27	$k0 y $k1	Reservados para el kernel (manejo de las interrupciones).
28	$gp	Puntero global. Apunta a la mitad del bloque de 64KB en el segmento de datos estáticos.
29	$sp	Puntero de pila (stack pointer). Ojo: no se actualiza automáticamente.
30	$s8 o $fp	Puede usarse como valor salvado. También se usa en al forma denominada "frame pointer" que permite acceder de forma indexada a diferentes posiciones de la pila.
31	$ra	Dirección de retorno utilizadas por algunas instrucciones de construcción de subrutinas.

Cuadro: Registros MIPS (no se incluyen los 32 registros para punto flotante)

Las operaciones de carga y salvado (I)

- La transferencia de un dato de la memoria al registro se denomina **cargar (load) el registro)**.
- La transferencia de un dato desde un registro a la memoria se denomina **salvar (store) el registro**.
- Formato de las instrucciones de carga y salvado en MIPS:
  - Nombre de la operación: carga **lw**, salvado **sw**.
  - Registro a cargar o salvar.
  - Constante.
  - Un registro para acceder a la memoria. La dirección de la memoria a cargar o salvar, se obtiene sumando la constante al contenido del registro de 32 bits.

Las operaciones de carga y salvado (II)

- Ejemplo: sea la operación $g = h + A[8]$

Ensamblador MIPS:

```
1  lw $t0,32($s3) # Carga
       desde la memoria a
       registro
2  add $s1, $s2, $t0 # Suma
       de los registros y el
       resultado a otro.
```

- Ejemplo: sea la operación $A[12] = h + A[8]$

Ensamblador MIPS:

```
3  lw $t0,32($s3) # Carga
       desde la memoria a
       registro
4  add $t0, $s2, $t0 # Suma
       de los registros y el
       resultado a otro.
5  sw $t0,48($s3)
```

Codificación de las instrucciones

- Cada instrucción se codifica en una palabra de 32 bits.
- Cualquier instrucción del repertorio se codificará en alguno de los siguientes formatos:
  - **Formato R:** Formato para instrucciones de cálculo aritmético como **add** o **sub**.
  - **Formato I:** Utilizado para operaciones que requieren de un valor constante como la transferencia de datos en **lw** o **sw**.
  - **Formato J:** Instrucciones de salto incondicional.

Formato R

opcode	rs	rt	rd	shamt	funct
31 . . . 26	25 . . . 21	20 . . . 16	15 . . . 11	10 . . . 6	5 . . . 0

- **opcode:** Código de la clase de operación (selector del microprograma).
- **rs:** Primer registro operando.
- **rt:** Segundo registro operando.
- **rd:** Registro destino. En el ensamblador aparece como primer operando de la instrucción.
- **shamt:** Desplazamiento de bits (usado en instrucciones de desplazamiento).
- **funct:** Acompaña al opcode para terminar de definir la operación completa.

Formato R: Ejemplos

- La instrucción **add** (suma) tiene formato R. Su opcode es el 0 y su campo funct es 20. Su definición es $R[rd] = R[rs] + R[rt]$.
- La instrucción **addu** (suma) tiene formato R. Su opcode es el 0 y su campo funct es 21. Su definición es $R[rd] = R[rs] + R[rt]$.
- Como vemos, la definición de add y addu parece la misma, tienen el mismo opcode pero su campo funct difiere. La razón que es add puede provocar una excepción de overflow, mientras que addu no.
- La instrucción **sll** (desplazamiento a la izquierda tiene formato R. Su opcode es el 0 y su funct es 0. Su definición es $R[rd] = R[rs] << shamt$. Como vemos no se usa el campo rt, y si el campo shamt.

Ensamblador MIPS:

```
6   add $t0, $s2, $t1 # rd es $t0, rs es $s2 y rt es $t1
```

Formato I

opcode	rs	rt	immediate
31 . . . 26	25 . . . 21	20 . . . 16	15 . . . 0

- **opcode:** Código de la clase de operación (selector del microprograma).
- **rs:** Operando. Si no se trata de una instrucción de cálculo que devuelve un valor en un registro, éste sería el primer operando de la instrucción en ensamblador.
- **rt:** Actúa como destino en las operaciones de cálculo: en este caso en la instrucción en ensamblador aparece como primer operando de la instrucción.
- **immediate:** Constante de 16 bits.

Formato I: ejemplos (suma inmediata) (I)

- La instrucción **addi** (suma inmediata) tiene formato I. Su definición es: dado **addi $rt, $rs, imm**, $R[rt] = R[rs] + SignExtImm$. Esta instrucción tiene opcode 8. El significado de SignExtImm es **extensión del signo de la parte inmediata**.
- Como la constante tiene 16 bits, bajo la interpretación de complemento a dos, el signo es el bit más significativo, es decir, el que ocupa la posición 15. La suma binaria requiere del mismo número de bits en ambos operandos, así que necesitamos que la constante pase de tener de 16 bits a 32 bits.

Formato I: ejemplos (suma inmediata) (II)

- **Extensión del signo:** Queremos aumentar el número de bits sin cambiar el significado del código bajo la interpretación de complemento a 2. Para lograr esto, es necesario copiar el bit más significativo (bit de signo) tantas veces hasta completar el número de bits requerido. Veamos un par de ejemplos de extensiones de signo:
  - $10010 \rightarrow 1111111111110010$
  - $00010 \rightarrow 0000000000000010$

Formato I: ejemplos (carga, carga superior e inferior inmediata y copia de registros) (I)

- La instrucción de carga **lw** tiene formato I. Su definición es: dado **lw $rt,imm($rs)**, realizar $R[rt] = M[R[rs] + SignExtImm]$. Como vemos, la constante se interpreta en complemento a 2, por lo que el desplazamiento en la memoria respecto a la dirección base, puede ser positivo o negativo.
- La instrucción de carga superior inmediata es **lui** y tiene formato I. Su definición es: $R[rt] = \{imm, 16'b0\}$. Es decir, si hacemos **lui $t0,0xedf0**, cargaremos en los 16 bits superiores de $t0, la constante 0xedf0 y pondremos los 16 bits inferiores a 0.

Formato I: ejemplos (carga, carga superior e inferior inmediata y copia de registros) (II)

- Supongamos que deseamos hacer algo parecido a **lui**, es decir, cargar una constante de 16 bits, pero en la parte menos significativa del registro. Una forma de hacerlo para la constante 0x7df0 sería: **addi $t0, $zer0, 0x7df0**. Obsérvese que esto no sería válido si el bit de signo fuera 1, habría que hacerlo de otra forma.
- Otra utilidad de **addi** puede ser la de copiar registros. Por ejemplo, si queremos pasar el contenido del registro $t1 al $t0, lo podemos conseguir mediante **addi $t0,$t1,0x0**.

Formato I: ejemplos (salto condicional) (I)

- La instrucción de salto condicional **beq** (branch on equal) tiene formato I. Su definición es: dado **beq $rs, $rt, immm**, realizar:

$$if(R[rs] == R[rt])PC = PC + branchaddr$$

donde $branchaddr = \{14immediate[15], immediate, 2b'0\}$ (desplazamos el valor inmediato dos lugares a la izquierda y extendemos su bit de signo hasta completar los 32 bits).

- El desplazamiento a la izquierda supone una multiplicación por 4. Esto se debe a que la dirección de salto debe situarse en la frontera de una palabra siempre, ya que ahí es donde comienza el código de la instrucción.

Formato I: ejemplos (salto condicional) (II)

- Es decir, si ambos registros operando son iguales, tomamos como referencia la dirección de la siguiente instrucción (valor del contador de programa) y nos desplazamos *branchaddr* obtenido como se menciona anteriormente.
- Así pues, tenemos un total de 18 bits para expresar el desplazamiento. Esto supone un rango de complemento a dos que va desde $-2^{15} \times 4 = -2^{17}$ hasta $(2^{15} - 1) \times 4 = 2^{17} - 4$ o lo que es lo mismo 131068 bytes hacia adelante y 131072 bytes hacia atrás, o 32768 instrucciones (palabras de 32 bits).
- Por tanto, el rango del salto condicional hacia adelante o hacia atrás está limitado por los 16 bits destinados a la parte inmediata en el formato I.
- Si se intentase realizar definir un salto más allá de esta cantidad, tendríamos un error en tiempo de compilación.
- En el uso habitual del ensamblador se usan etiquetas para definir los

Formato J

opcode	immediate
31 . . . 26	25 . . . 0

- El formato J se utiliza normalmente para el salto incondicional a una dirección inmediata.
- Hay dos instrucciones básicas de salto que lo utilizan y son:
  - Jump (j). Se trata del salto incondicional a una dirección de memoria.
  - Jump and link (jal). Antes de saltar se guarda en el registro $ra la dirección de la siguiente instrucción a ejecutar. A continuación se realiza el salto. De esta forma, podemos retomar la siguiente instrucción, usando el contenido del registro.

Formato J: Ejemplo (Jump)

- La instrucción **j** permite el salto a una dirección inmediata. Se define del siguiente modo: si tenemos **j immediate**, entonces el efecto es $PC = JumpAddr$, donde $JumpAddr = \{PC[31:28], imm, 2b'0\}$.
- Es decir, se produce un salto a una dirección absoluta compuesta a partir de los 26 bits de la parte inmediatade la instrucción en el modo descrito. Como vemos, los cuatro bits más significativos del PC se mantienen. Esto significa que la memoria queda dividia en segmentos, donde cada segmento no puede ser alcanzado desde otro segmento mediante saltos absolutos. El tamaño de los segmentos es $2^{28} = 256MB$
- El resto de la dirección consiste en la parte inmediata desplazada dos bits a la izquierda (pues la dirección del salto debe apuntar siempre a un múltiplo de cuatro).