Plataformas de soporte computacional: arquitecturas avanzadas, sesión 1

Diego R. Llanos, Belén Palop
Departamento de Informática
Universidad de Valladolid
{diego,b.palop}@infor.uva.es

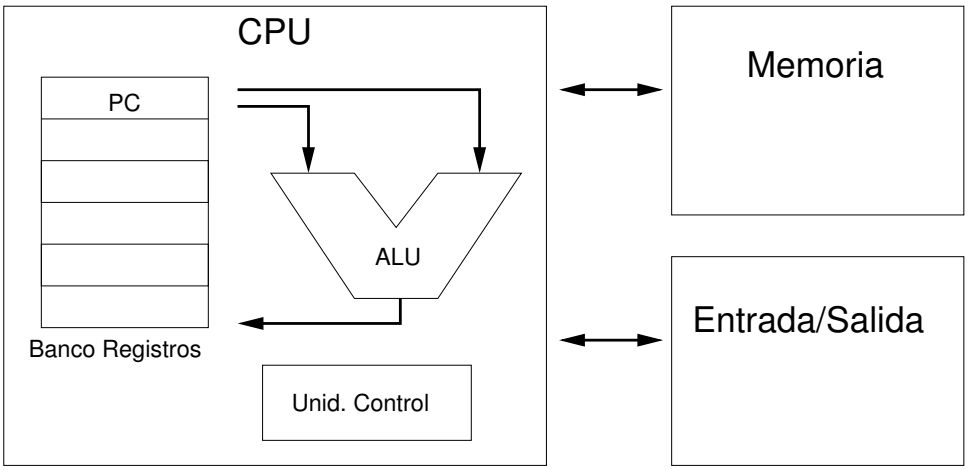
Índice

1. Arquitectura de Von Neumann, ciclo de instrucción

En este apartado se hará un breve repaso a la arquitectura Von Neumann y al concepto de ciclo de instrucción. Para ello se hará uso de un ejemplo de implementación uniciclo basado en la arquitectura MIPS de 32 bits.

Arquitectura Von Neumann

- Arquitectura basada en una serie de unidades funcionales:
  - CPU: encargada de la ejecución de instrucciones. Consta de un banco de registros, una unidad aritmético-lógica (ALU) y una unidad de control, encargada del secuenciamiento de operaciones.
  - Memoria: encargada del almacenamiento de instrucciones y datos.
  - Entrada/salida: interacción con el usuario o con otros equipos.



Ciclo de instrucción

- Fundamento de la ejecución de instrucciones.

  Fases

  1. Fase de búsqueda:
     - Se trae de memoria la instrucción a la que apunta el PC.
     - Se deja al PC apuntando a la siguiente instrucción.
  2. Fase de ejecución: se ejecuta la instrucción.

- Tipos de instrucciones:
  - De transferencia de datos entre registros y memoria
  - Operaciones aritméticas y lógicas (números enteros o reales)
  - Saltos condicionales e incondicionales en el código

Implementaciones de la arquitectura Von Neumann

- Dos filosofías de diseño:
  - Pocas instrucciones de tamaño fijo (RISC)
    - Ejemplos: MIPS, SPARC, PowerPC ($\simeq$ 50 instrucciones)
  - Muchas instrucciones de tamaño variable (CISC)
    - Ejemplo: Intel IA-32 ($\geq$ 300 instrucciones)
- Soluciones de compromiso entre los siguientes factores:
  - Instrucciones de tamaño fijo: simplicidad de la fase de búsqueda (pero dificultades en su diseño y ampliación).
  - Pocas instrucciones: rapidez en su ejecución (pero trabajo añadido al compilador).

2. Arquitectura MIPS

Arquitectura MIPS

- Desarrollada por John Hennessy, de Stanford, entre 1981 y 1985.
- Arquitectura RISC, palabra de instrucción de tamaño fijo, desarrollado para aprovechar al máximo las técnicas de *pipelining*

- Presente en routers, equipos empotrados (TiVO), y consolas como la Play Station, PSP, y PS2.
- ISA con versiones de 32 y 64 bits, además de extensiones para aplicaciones específicas (compresión, instrucciones SIMD para coma flotante, etc)
- Veremos brevemente el ISA de 32 bits, para utilizarlo como caso de ejemplo para las realizaciones uniciclo y multiciclo.
- Definición MIPS: Un "word" o "palabra" equivale a 4 bytes.

Arquitectura MIPS

Nombre	Ejemplo	Comentarios
32 registros	`$s0..$s7, $t0..$t9, $gp, $fp, $zero, $sp, $ra, $at, Hi, Lo`	Almacenamiento de datos. En MIPS, para hacer operaciones aritméticas los datos tienen que estar almacenados en registros. El registro `$zero` siempre vale cero. El registro `$at` está reservado por el ensamblador para manejar constantes grandes. Los registros `Hi` y `Lo` almacenan el resultado de la multiplicación y división.

Arquitectura MIPS

Nombre	Ejemplo	Comentarios
2^{30} memory words	Memory[0], Memory[4]...	Sólo accesible a través de transferencia de datos. MIPS utiliza direcciones de bytes, por lo que las direcciones de word son múltiplo de 4. La memoria almacena estructuras de datos y los contextos de las llamadas a subrutinas.

Arquitectura MIPS: instrucciones aritméticas

- Operaciones aritméticas: tres registros como operandos, el primero es siempre el destino de la operación.
- Ejemplos: `add`, `addu`, `addi`, `sub`, `subu`, `mfc0`, `mult`, `multu`, `div`, `divu`, `mfhi`, `mflo`.

Instrucción	Ejemplo	Significado	Comentarios
suma	`add $s1, $s2, $s3`	`$s1=$s2+$s3`	Tres operandos, detecta overflow
resta	`sub $s1, $s2, $s3`	`$s1=$s2-$s3`	Tres operandos, detecta overflow
suma inmediata	`addi $s1, $s2, 100`	`$s1=$s2+100`	Detecta overflow
suma sin signo.	`addu $s1, $s2, 100`	`$s1=$s2+$s3`	Tres operandos, no detecta overflow
resta sin signo.	`subu $s1, $s2, 100`	`$s1=$s2-$s3`	Tres operandos, no detecta overflow
multiplicación	`mul $s1, $s2`	`(Hi,Lo)=$s1`×`$s2`	Producto 64 bits en (Hi,Lo)
división	`div $s1, $s2`	`Lo=$s1 / $s2` `Hi=$s1 mod $s2`	Lo = división entera Hi = resto
Mover desde Hi	`mfhi $s1`	`$s1=Hi`	Recupera valor de Hi
Mover desde Lo	`mflo $s1`	`$s1=Lo`	Recupera valor de Lo

Arquitectura MIPS: instrucciones lógicas

Instrucción	Ejemplo	Significado	Comentarios
and	`and $s1, $s2, $s3`	`$s1=$s2&+$s3`	Tres operandos
or	`or $s1, $s2, $s3`	`$s1=$s2|$s3`	Tres operandos
and inmediato	`andi $s1, $s2, 100`	`$s1=$s2&100`	Tres operandos
or inmediato	`ori $s1, $s2, 100`	`$s1=$s2|100`	Tres operandos
desplazamiento lógico izquierda	`sll $s1, $s2, 10`	`$s1=$s2<<10`	Despl. constante
desplazamiento lógico derecha	`srl $s1, $s2, 10`	`$s1=$s2>>10`	Despl. constante

Arquitectura MIPS: instrucciones de transferencia

- Transferencia de datos: direccionamiento por base y desplazamiento (el desplazamiento es opcional).
- El primer operando no siempre es el destino de la operación.

Instrucción	Ejemplo	Significado	Comentarios
load word	`lw $s1, 100($s2)`	`$s1=Memory[$s2+100]`	Word de memoria a registro
store word	`sw $s1, 100($s2)`	`Memory[$s2+100]=$s1`	Word de registro a memoria
load byte	`lb $s1, 100($s2)`	`$s1=Memory[$s2+100]`	Byte de memoria a registro
store byte	`sb $s1, 100($s2)`	`Memory[$s2+100]=$s1`	Byte de registro a memoria
load upper immediate	`lui $s1, 100`	`$s1`=100*2^{16}	Carga sobre los 16 bits sup.

Arquitectura MIPS: algunas instrucciones de saltos

- Saltos o transferencia de control: actúan sobre el contador de programa.
- Saltos condicionales e incondicionales. Algunos ejemplos:

Instrucción	Ejemplo	Significado	Comentarios
branch on equal	`beq $s1, $s2, 25`	if `($s1==$s2)` goto PC=PC+4+100	Relativo al PC
branch on not equal	`bne $s1, $s2, 25`	if `($s1!=$s2)` goto PC=PC+4+100	Relativo al PC
set on less than	`slt $s1, $s2, $s3`	if `($s2<$s3) $s1=1` else `$s1=0`	(instrucción aritmética)
set on less than imm.	`slti $s1, $s2, 100`	if `($s2<100) $s1=1` else `$s1=0`	Complemento a dos
set on less than imm. unsigned	`sltiu $s1, $s2, 100`	if `($s2<100) $s1=1` else `$s1=0`	Binario natural
jump	`j 2500`	PC=2500*4	Absoluto
jump register	`j $ra`	PC=$ra	Absoluto
jump and link	`jal 2500`	$ra=PC; PC=2500*4	Absoluto

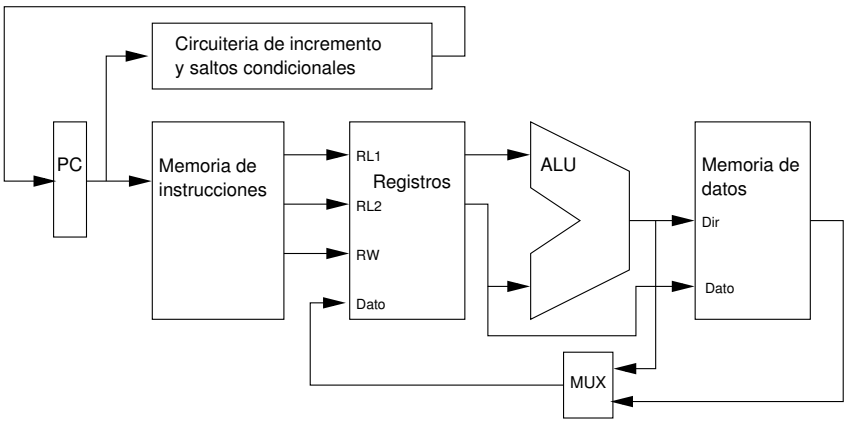
Arquitectura MIPS: Formatos de instrucción

Tipo	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Observaciones
Tipo R	op	rs	rt	rd	shamt	funct	Formato de instrucción aritmético
Tipo I	op	rs	rt	address/immediate			Formato transferencias, saltos e inmed.
Tipo J	op	dirección destino del salto					Formato instrucción `j` y `jal`

- Instrucciones de ancho fijo: 32 bits.
- Todas las instrucciones son de uno de los tres tipos (R, I, J).
- El tipo viene determinado por los seis bits de más peso. Por ejemplo, las instrucciones de tipo R tienen los primeros seis bits a 0, y los últimos seis bits (campo `funct`) codifica la operación aritmética.

3. Implementación uniciclo de la arquitectura MIPS

Arquitectura MIPS: implementación uniciclo, esquema



Uso de estas unidades:

Instrucción	Mem. Instr.	Lectura Reg.	ALU ALU	Mem. Datos	Escritura Reg.	Total
`j dir`	X					1 etapa
`beq $t0, $t1, dir`	X	X	X			3 etapas
`add $t0, $t1, $t2`	X	X	X		X	4 etapas
`sw $t0, ($t1)`	X	X	X	X		4 etapas
`lw $t0, ($t1)`	X	X	X	X	X	5 etapas

Arquitectura MIPS: implementación uniciclo, detalles

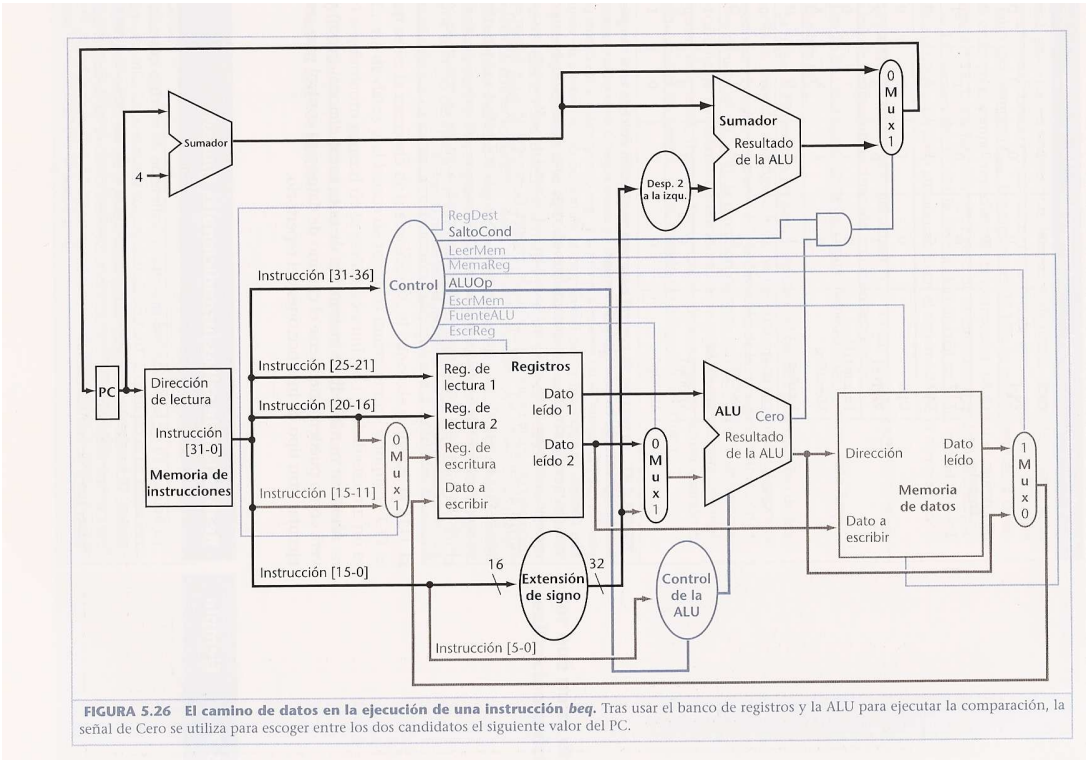


FIGURA 5.26 El camino de datos en la ejecución de una instrucción *beq*. Tras usar el banco de registros y la ALU para ejecutar la comparación, la señal de Cero se utiliza para escoger entre los dos candidatos el siguiente valor del PC.

Por qué no es rentable la implementación uniciclo

- Supongamos que los accesos a memoria de instrucciones y datos consumen 2 ns cada uno, a la ALU 2 ns y a los registros 1 ns.
- Los tiempos de cada instrucción quedarían así:

Instrucción	Mem. Instr.	Lectura Reg.	ALU ALU	Mem. Datos	Escritura Reg.	Total
`j dir`	2					2 ns
`beq $t0, $t1, dir`	2	1	2			5 ns
`add $t0, $t1, $t2`	2	1	2		1	6 ns
`sw $t0, ($t1)`	2	1	2	2		7 ns
`lw $t0, ($t1)`	2	1	2	2	1	8 ns

¿De qué tamaño elegimos el ciclo de reloj?

- Tamaño variable: complicado; coste adicional fijo.
- Tamaño fijo: necesariamente, el de la instrucción más larga.

Implementación uniciclo: ejemplo de coste

- Sean los tiempos de ejecución de la tabla anterior.
- Supongamos que los *loads* suponen el 24 % de las instrucciones ejecutadas.
- Supongamos que los *stores* suponen el 12 % de las instrucciones ejecutadas.
- Supongamos que los saltos condicionales suponen el 18 % de las instrucciones ejecutadas, y los incondicionales el 2 %.
- Supongamos que las instrucciones aritméticas y lógicas suponen el 44 % de las instrucciones ejecutadas.

Pregunta:
¿Cuál es el tiempo medio de ejecución de una instrucción con un ciclo de tamaño variable, y qué relación guarda con una realización con ciclo de tamaño fijo?

- La comparación no es muy realista, porque no cuenta los costes de implementación del ciclo variable.
- Sin embargo, nos da una visión del margen de mejora.
- El tiempo medio de la implementación variable será:

$$T_V = 8 \times 24\,\% + 7 \times 12\,\% + 6 \times 44\,\% + 5 \times 18\,\% + 2 \times 2\,\% = 6{,}3\text{ns}$$

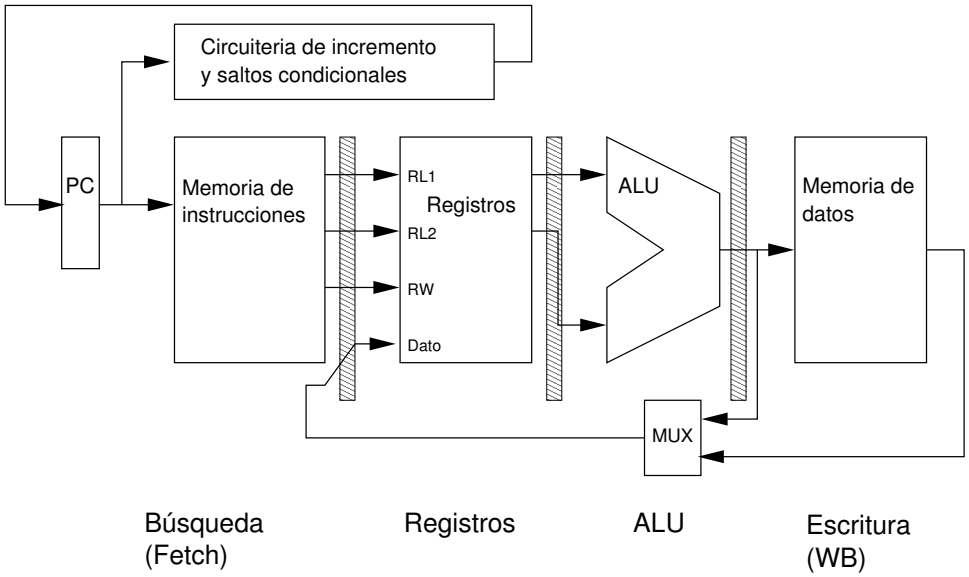
- La implementación con ciclo fijo requiere 8 ns, lo que supone un 27 % más de tiempo en la ejecución de un programa *típico*.
- La cosa empeora para el ciclo fijo si incluimos instrucciones de coma flotante (poco frecuentes pero que pueden consumir 16 ns).

Por eso no se usa la implementación uniciclo.

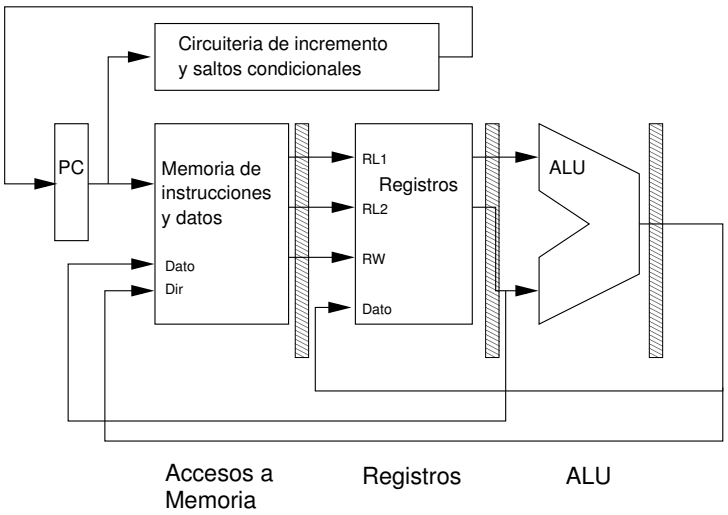
4. Implementación multiciclo de la arquitectura MIPS

Implementación multiciclo

- Objetivo: aprovechar los tiempos muertos que genera la implementación uniciclo.
- Esta implementación divide la ejecución de una instrucción en fases, cada una de ellas de un ciclo de reloj (más corto que antes).
- Requiere *buffers* para almacenar los estados intermedios.



- La implementación multiciclo permite usar una misma memoria para las instrucciones y para los datos (ya que se usan en instantes diferentes).



- Ahora la frecuencia de reloj se ajusta a la duración de la *fase* más lenta (en nuestro ejemplo, 2 ns).

Implementación multiciclo: detalles

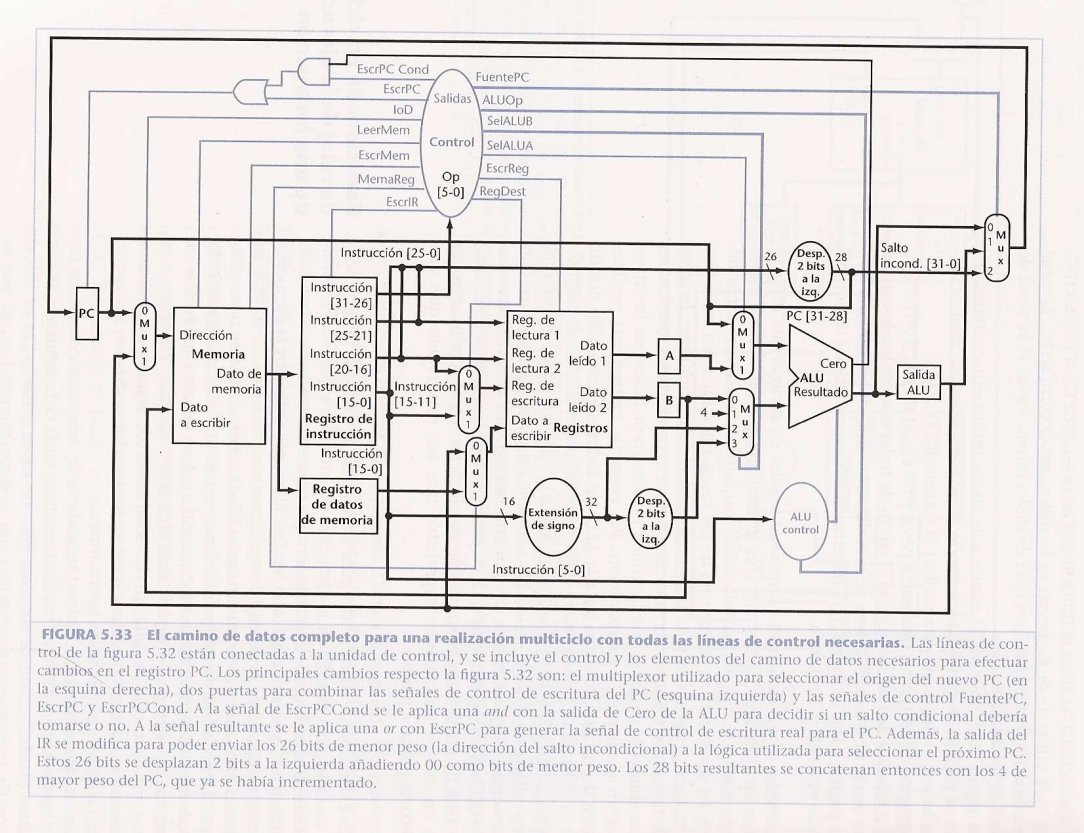


FIGURA 5.33 El camino de datos completo para una realización multiciclo con todas las líneas de control necesarias. Las líneas de control de la figura 5.32 están conectadas a la unidad de control, y se incluye el control y los elementos del camino de datos necesarios para efectuar cambios en el registro PC. Los principales cambios respecto la figura 5.32 son: el multiplexor utilizado para seleccionar el origen del nuevo PC (en la esquina derecha), dos puertas para combinar las señales de control de escritura del PC (esquina izquierda) y las señales de control FuentePC, EscrPC y EscrPCCond. A la señal de EscrPCCond se le aplica una *and* con la salida de Cero de la ALU para decidir si un salto condicional debería tomarse o no. A la señal resultante se le aplica una *or* con EscrPC para generar la señal de control de escritura real para el PC. Además, la salida del IR se modifica para poder enviar los 26 bits de menor peso (la dirección del salto incondicional) a la lógica utilizada para seleccionar el próximo PC. Estos 26 bits se desplazan 2 bits a la izquierda añadiendo 00 como bits de menor peso. Los 28 bits resultantes se concatenan entonces con los 4 de mayor peso del PC, que ya se había incrementado.

Implementación multiciclo: conclusiones

- La implementación multiciclo aprovecha mejor los tiempos variables de las instrucciones, aunque tampoco al 100 %.
- Requiere una *secuenciación* de las operaciones: en lugar de unidades puramente combinacionales, hacen falta unidades de control secuenciales (cableadas o microprogramadas).
- Observación: las unidades funcionales ya no están activas todo el tiempo. Podría lanzarse una instrucción y, mientras avanza su ejecución, ir lanzando la siguiente: *pipelining*
- Problemas: muchos y variados
  - No todas las instrucciones terminan al mismo tiempo.
  - Se producen conflictos (hazards) cuando dos instrucciones necesitan usar la misma unidad funcional.
  - Problemas en los saltos...
- Estos problemas (y sus soluciones) se verán en la próxima sesión.