ARQUITECTURA E INGENIERÍA DE COMPUTADORES

Tema 1.3

Miquel Gómez Corral

Ing. Informática ETSINF, UPV

Tema 1.3

Diseño de los juegos de instrucción

GENERALIDADES

Las ISA son la interfaz entre los programas y la ruta de datos (lo de la segmentación).

Caso Frecuente/Caso raro: Se debe **optimizar el caso frecuente** (*lo que se hace más veces*), centrarse en él para mejorarlo. Por otro lado, el **caso raro** no lo puedes descuidar, pero con que **esté correcto** ta bien.

Regularidad/Ortogonalidad: Siempre que tenga sentido, las operaciones, modos de direccionamiento y tipos de datos deben ser independientes → *Simplifica la generación de código, sobre todo si la decisión se toma en dos fases de la compilación distintas.*

- **Regu**: Se dice cuando el **juego de instrucciones NO sorprende al usuario** (*si hay una de + habrá de -*).
- **Orto**: Que, si tengo varios modos de direccionamiento o de tipo de datos, si elijo uno con otro (combinaciones), funciona sin que tenga que mirar si cuadra o no. (*Ortogonal help compilación sea rápida*).

Ofrecer primitivas y no soluciones: **Evitar incluir soluciones concretas** que den soporte directo a construcciones de alto nivel, pero Solo funcionarán con un lenguaje, al ser muy específicas.

*Lo que hay que hacer es **dar operaciones sencillas y rápidas**, ya se apañará el compilador para usarlas bien, no te metas en el alto nivel, pa eso ya está el compi que hace código muy optimizado.*

Principio "uno o todo": O hay **una sola forma** de hacer una determinada cosa, o **todas las formas** son posibles. El objetivo es Simplificar el coste del cálculo de cada alternativa. *Según la arquitectura es una u otra.*

- **Ejemplo**: Con las condiciones de salto, si solo tienes ">, =" solo hay Una forma de generar todas las condiciones combinandolas, pero si tienes ">, >=, <=, =, ¡=" o sea, Todas las formas de generar condiciones.

Incorporar instrucciones con constante (*El modo inmediato*): Si tengo **cosas que sé que no van a variar** (*el número 5*), pa que las voy a guardar en memoria y traérmelo sí sé que es un valor sin más → *addi, ori, li…*

JUEGO DE INSTRUCCIONES RISC-V: *rv64imfd*

Se hace después de mucho, por lo que está bien hecho. Oculta los detalles de implementación y debe verse como un interfaz software para una gran variedad de implementaciones. *Se usa en una gran cantidad de cosas y sitios.*

Conjunto básico y extensiones

Conjunto básico: Conjunto **mínimo**, presente en cualquier procesador RISC-V y Soporte **aritmética** (suma/resta) de enteros y **operaciones lógica**. *Se puede usar en multinúcleos, procesadores sin segmentar…*

El RISC-V es modular, eso quiere decir que, tengo un set básico y puedo "añadir" más según donde lo vayas a usar. Luego le dice al compilador qué instrucciones tiene disponibles y este se adapta para ser lo más optimo posible.

Extensiones: Algunas añaden multiplicación y división de enteros (M), opresiones atómicas (A), operaciones en coma flotante (simple y doble, F y D), instrucciones comprimidas en memoria (16 bits, C) … NOSOTRO: **rv64imfd**

Mirar en el otro PDF las instrucciones del RISK-V

TIPOS DE JUEGOS DE INSTRUCCIONES

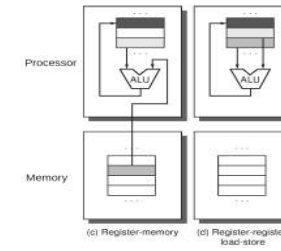
Condicionantes

El juego de instrucciones **condiciona el proceso de compilación** de los programas: *Según como y donde están/acceden a los operandos, cuantos registros hay, de que manera se accede a los datos y de que tipos, como se controlan el flujo de ejecución (saltos).*

No existe una única respuesta a cada una de estas preguntas → existen distintos tipos de juegos de instrucciones.

Paradigma actual: Memoria direccionable y banco de **registros de propósito general**. Se almacenan los operandos en la CPU y los datos en la memoria *(se opera sobre estos datos y se genera un resultado)*. En cada instrucción los operandos deben nombrarse explícitamente. Todo esto permite ***Compilación eficiente***.

Hay arquitecturas que permiten acceder a memoria para conseguir operandos mientras que otros te exigen llevar esos operandos a la CPU (registros) y ya luego operar.



Tipos de juegos de instrucciones: *Principalmente 2 tipos*

IA (*Intel Architecture*): Propia de los procesadores de Intel y compatibles (AMD)

RISC: Propia de los procesadores RISC-V, MIPS, ARM, SPARC, POWER

	Intel Architecture	RISC
Modelo prog.	Registro-memoria (*R-M*)	Load/Store (*L/S*)
Registros	Pocos, long. variable	Muchos, long. fija
Formato instr.	Variable	Fijo (32 bits)

Modelos L/S y R-M

La elección afecta al tiempo de ejecución: $Tej = I \times CPI \times T$ ES MÁS FÁCIL DE SEGMENTAR Y DECODIFICAR L/S que R-M. *Ya que las instrucciones de L/S tienen una complejidad muy parecida entre ellas es casi siempre igual. Sin embargo, R-M tiene instrucciones muy muy sencillas y otras muy muy complejas, entonces no se equiparan tan bien.*

RISC Modelo L/S

1. Las **instrucciones de cálculo** operan sólo con registros y tienen tres operandos (*unas pocas cuatro*).
2. Hay **instrucciones especializadas load y store** que transportan los datos entre los registros y la memoria.
3. La **cantidad de trabajo** de las instrucciones del juego es **parecida** (*o un cálculo o un acceso a la memoria*).
4. Ejecución:
   a. Un programa tiene más instrucciones para hacer el mismo trabajo: **I ↑**
   b. Formato más sencillo, más fácil decodificar: **T ↓**
   c. Cantidad de trabajo por instrucción homogéneo: **CPI ↓**

IA Modelo R-M

1. Las **instrucciones de cálculo** pueden operar con registros o con memoria y tienen dos operandos.
2. Instrucción MOV para transportar datos: R ⇔ M y R ⇔ R
3. La **cantidad de trabajo** por instrucción es muy diversa.
4. Ejecución:
   a. Un programa tiene menos instrucciones para hacer el mismo trabajo: **I ↓**
   b. Formato variable, más difícil decodificar: **T ↑**
   c. cantidad de trabajo por instrucción muy diferente: **CPI ↑**

Intel 32 bits	Equivalente en RISC-V
add EAX,EBX	add t0, t0, t1
add EAX,a	lw t1, a add t0, t0, t1

add a,EAX	lw t1, a
	add t1, t1, t0
	sw t1, a

Registros y tipos de operando

RISC-V

- Los registros son **numerosos** **y tienen todos** **igual tamaño**.
- **Cada instrucción** de cálculo opera con el contenido completo de los registros → *Conversión entre tipos si las load cargan datos de tamaño inferior al tamaño del registro (extensión de zeros o de signo).*
- Algunas instrucciones operan con medio registro (addw, subw, ...). *En el Embebed con registros de 16 o así*
- Se pueden hacer **accesos** solo a **Bytes**, **Half** o **Word** enteras… Esto permite mucha **flexibilidad**.

IA

- Pocos **registros y adaptados** a los tipos de datos disponibles cada instrucción aritmética tiene un código de operación para cada tipo → *En x86-64 hay cuatro versiones de add para operandos de 8, 16, 32 y 64 bits*
- En esta se añaden tantas operaciones aritméticas como tipos, en el RISC hay una o dos genéricas para todos, aquí se han de especificar y eso complica las cosas mucho:

Cambio de longitud: Es más fácil cambiar la longitud de palabra en RISC-V: *Ya que en Risc, solo tienes instrucciones genéricas, si quiero cambiar en IA tengo que poner un montón nuevas.*

Registros y tipos de operando

Registros IA-32: Son un poco raros, cada uno se supone que es para una cosa diferente (aun que se les puede dar cualquier uso). Se pueden tratar de diferentes formar: *4 registros de 16 bits o 8 registros de 8 bits*.

Registros RISC-V (*rv64imfd*): 32 registros de 64 bits : x0 . . . x31. Tiene una manera de usarse a seguir, pero se pueden usar como quieras

Código fuente	x86-64	RISC-V

byte a, b, c;
...
c = a + b;
	a db ...
	...
	MOV AL, a
	add AL, b
	MOV c, AL
		a: byte ...
		...
		lb t0, a
		lb t1, b
		add t2, t0, t1
		sb t2, c

Codificación de instrucciones

Estrategias de codificación: Las instrucciones se almacenan en la memoria según un formato, que indica la operación a realizar (*código de operación*) y los operandos. Formato fijo vs. variable:

- **Fijo**: Todas las instrucciones se codifican utilizando el **mismo número de bits**.
  - **Facilita** la búsqueda de instrucciones y su decodificación. C*omo siempre son iguales no te complicas*
  - A veces, **derrocha bits en el formato**, ya que no todas las instrucciones requieren el mismo espacio.
- **Variable**: El **número de bits** requerido para codificar la instrucción **varía** según el tipo de instrucción.
  - **Complica** la búsqueda de instrucciones y su decodificación
  - **Optimiza espacio ocupado** por las instrucciones, y, por lo tanto, por los programas. *Se adapta a cada instrucción sin usar memoria innecesaria, le da a cada una lo que le hace falta.*

*Si tus instrucciones son **más o menos igual** te renta que sea **fijo** xq derrochas, pero no mucho. Ahora, si tienes una **instrucción enorme**, como pilles el fijo te jode el resto de las instrucciones, tendrías que ir al **variable**.*

Nº de bits del formato

El **número de bits destinado** al formato impone un límite al espacio destinado a cada uno de los campos, el cual limita el nº de variantes de este: nº de ***instrucciones*** (*códigos de operación*), nº de ***registros***, ***espacio de memoria*** direccionable…

Hay elegir cierta cantidad de bits y luego las instrucciones se tendrán que adaptar a eso. Si gastas 20 de 24 bits en la instrucción y 2 registros, a lo mejor solo tienes 4 bits para el inmediato.

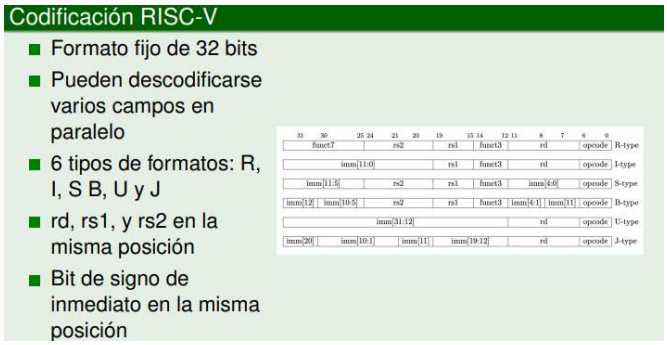
Formato único: La correspondencia entre los bits del formato y los campos es **siempre la misma**.

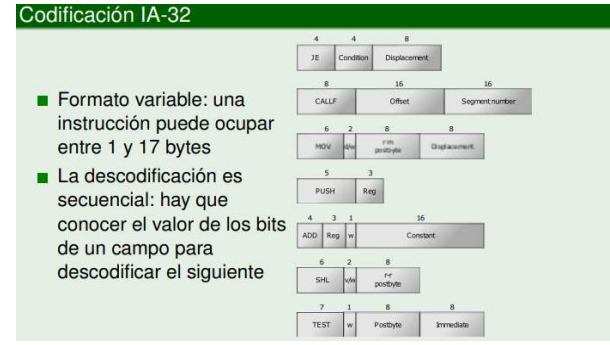
- **Facilita la decodificación** de la instrucción.
- **Derrocha bits en el formato**, ya que no todas las instrucciones requieren todos los campos previstos.

Múltiples formatos: **Cada formato puede tener campos distintos** y se relacionan estos y los bits del formato.

- Permite **ajustar mejor los bits ocupados por la instrucción** y los campos requeridos

Estrategias de codificación





Lo de los **inmediatos barajados** es para optimizar la implementación (*en el B se obvia el 0 xq las direcciones están x2*)

Las de **IA** son muy **variadas** y cada una mide lo que le hace falta. En el **RISC todas miden 32 bits**, pero en cada tipo se usan siempre 7 bits de operaciones + otros, si te sobran bits los inviertes en otro campo para que sea más mejor.

Además, el RICS, si tiene x campo, SIEMPRE está en la misma posición, las operaciones, los registros destino, el rs1 el rs2, func3, los inmediatos… *El funct 3 sirve para diferenciar entre diferentes tipos de instrucciones (de Branch <, >…)*

Tipos: **Registro** "R", **Inmediatas** "I", **Store** "S", **Branch** (*saltos condicionales*) "B", **Inmediatos grandes** "U", **Jump** "J".

Direccionamiento de Memoria

Esto se refiera a **Como se especifican las direcciones**. Podemos Tener modos de direccionamiento sofisticados o más sencillos.

- Reducción del número de instrucciones de los programas: I ↓
- Hardware más complejo: CPI ↑ y/o T ↑.

Modo	Ejemplo	Significado
Directo a registro	add x1, x2, x3	x1←x2+x3
Inmediato	add x1, x2, 1	x1←
Directo o Absoluto	lw x1,(1000)	x1←Mem[1000]
Registro indirecto	lw x1,(x2)	x1←Mem[x2]
Desplazamiento	lw x1,100(x2)	x1←Mem[100+x2]
Indexado	lw x1,(x2+x3)	x1←Mem[x2+x3]
Indirecto a mem.	lw x1,@(x2)	x1←Mem[Mem[x2]]
Autoincremento	lw x1,(x2)+	x1←Mem[x2] x2←x2+*d*
Autodecremento	lw x1,–(x2)	x2←x2–*d* x1←Mem[x2]
Escalado	lw x1,100(x2)(x3)	x1←Mem[100+x2+x3*d]

IA: Incluye **hasta el modo escalado**. Combina libremente los más complejos, *una instrucción puede tener mucho trabajo*:

RISC-V: Solo tienen **modo desplazamiento** y a partir de ahí **simulas los otros**. Estos otros se usan poco por lo que no es problema: $Rformat: Rd \Leftarrow Rs\ op\ Rt.$ $Iformat: Rd \Leftarrow Rs\ op\ X$

Rango de valores inmediatos: Solución entre disponer de constantes grandes y el número de bits ocupados en el formato. 12 bits típico en RISC-V pero Hay instrucciones para facilitar el trabajo con valores inmediatos más grandes. *NO estás limitado a 12 bits. Puede operar y conseguir en un registro inmediatos MÁS GRANDES:*

- LUI x1, Uvalor : Regs[x1] ← Uvalor << 12 *Parte alta*
- ORI x1,x1, Lvalor : Regs[x1] ← Regs[x1] or Lvalor *Parte baja*

Acceso a la memora con estos inmediatos: Igual que antes, estás limitado a 12 bits, si quieres más tienes que operar y pasar por un registro usando otras instrucciones, después ya puedes usar esa dirección/inmediato.

Control de flujo

3 tipos de salto, Saltos **condicionales** (branch), Saltos **incondicionales** (jump), **Llamadas/retorno** a/de procedimiento (*jal, jalr*). **Se salta mucho***: la estadística es 1/3 para cada uno de los tipos. Entre todos, 5/6 de los saltos saltan.*

Modos de decirle el direccionamiento en el salto

Relativo al PC: El destino **suele estar cerca de la instrucción actual** → las direcciones relativas consumen pocos bits. Los bits para el salto son 13 para condicionales y 21 en incondicionales. *Por lo que renta*

Indirecto a registro: Útil si el **destino del salto es desconocido durante la compilación**. Sentencias que seleccionan una de entre varias alternativas. métodos virtuales en lenguajes orientados a objetos. Paso de funciones como parámetros a otra función Librerías enlazadas dinámicamente.

Instrucciones SIMD

Son operaciones vectoriales hechas por hardware. Lo pasó por encima no parece nada importante la verdad.

Ejercicios

Aceleración CPU: núcleos que tienes ahora / los que tenías antes · mejora de velocidad

Aceleración Memoria: Controladores que tienes ahora / los que tenías antes · mejora de velocidad

Aceleración general: $Sg = \frac{1}{1-F+\frac{F}{S}}$ o con varios componentes: $Sg = \frac{1}{1-\sum Fi+\sum\frac{F}{S}}$