Tema 04: Arquitectura del Set de Instrucciones (ISA)

Arquitectura de Computadoras

Ing. Nicolás Majorel Padilla (npadilla@herrera.unt.edu.ar)

http://microprocesadores.unt.edu.ar/arqcom/

¿Cuáles temas veremos?

- Primera parte: **Generalidades**
  - Introducción. Conceptos generales.
  - ***P**erformance, **P**rice & **P**ower*.
  - Conceptos de **P**aralelismo y ***P**ipelining*.
- Segunda parte: **Procesadores**
  - ISA: qué es, cómo se diseña.
  - Diseños de procesadores para *performance*.
- Tercera parte: **Subsistema de Memoria**
  - Memorias Caché.
  - Memoria Virtual.
- Cuarta parte: **Sistema completo**
  - Subsistemas de Entrada/Salida y de Comunicaciones.
  - Consumo de energía.
- Quinta parte: **Nuevos paradigmas**
  - Arquitecturas específicas.
  - Seguridad.

Temas que veremos

- Contexto y evolución histórica de los ISA.
- Importancia de un ISA.
- Parámetros de Diseño de un ISA.
  - Diferentes alternativas.
- Métricas de Diseño de un ISA.
- Principios de Diseño de un ISA.

Lectura recomendada

- Computer Organization and Design, RISC-V Edition (2da ed, 2021)
  - Sección 2.1: *Introduction*
  - Sección 2.24: *Historical Perspective and Further Reading*
  - Sección 2.5: *Representing Instructions in the Computer*
  - Sección 2.10: *RISC-V Addressing for Wide Immediates and Addresses*
  - Sección 2.19: Real Stuff: *x86 Instructions*
  - Sección 2.20: Real Stuff: *The Rest of the RISC-V Instruction Set*
  - Sección 2.22: *Fallacies and Pitfalls*
  - Sección 2.23: *Concluding Remarks*
- Guía Práctica de RISC-V (1ra ed)
  - Capítulo 1: *¿Por qué RISC-V?*
- Para los más curiosos:
  - Sección 2.12: *Translating and Starting a Program*
  - Sección 2.17: Real Stuff: *ARMv7 (32-bit) Instructions*
  - Sección 2.18: Real Stuff: *ARMv8 (64-bit) Instructions*

Evolución Histórica de ISAs

- Prehistoria (1960): instrucciones simples.
  - Procesadores con pocos transistores y memorias muy caras.
- 1970-1980's: era CISC (*Complex Instruction Set Computer*).
  - Procesadores cada vez con más transistores, impulsados por la Ley de Moore.
  - Instrucciones cada vez más poderosas, que buscaban "cerrar la brecha" con los lenguajes de alto nivel.
  - Memoria era cara, así que había que optimizar su uso.
  - x86: Intel y todos sus clones.

Evolución Histórica de ISAs

- 1990's-**actualidad**: era RISC (*Reduced Instruction Set Computer*).
  - Vuelta a instrucciones simples, motivada por **Performance**.
    - *"It was discovered that RISC reads a few more instructions (approximately 30% more) but reads them so much faster (around 5 times faster) that the net result is that RISC is 3 to 4 times faster than CISC."*
    - MIPS, SPARC, PowerPC, ARM, DEC Alpha, etc.
    - La simplicidad favorecía diseños en **Pipelining**, que permitieron aumentar la frecuencia de reloj.
    - La memoria se fue haciendo progresivamente más barata, pero no más rápida.
- Conviven con CISC, que *mutaron para sobrevivir*.
  - Mantienen compatibilidad, pero internamente traducen las instrucciones (complejas) en micro-operaciones que se ejecutan como RISC.
  - El éxito comercial puede obligar a soluciones de ingeniería extremadamente complejas.

Evolución Histórica de ISAs

- En la actualidad, hay **dos ISAs dominantes**:
  - **x86_64**, en PCs, Servidores y Supercomputadoras.
    - Porcentaje del mercado: 80% estimado (2025)
      - ¡Era entre 95-99% hace 5 años!
    - Diseñado originalmente en 2003 por… AMD.
    - Principales fabricantes: Intel y AMD.
  - **ARM** en sistemas embebidos, celulares, etc.
    - Porcentaje del mercado: 95-99%.
    - Principales fabricantes: Qualcomm, Samsung, Apple.
    - Pero en realidad, hay varios ISAs distintos:
      - Thumb, Thumb-2, ARMv7, ARMv8, ARMv9…
      - Usados en distintos tipos de sistemas embebidos.
- Estos ISAs son propietarios.
  - Lo que implica que para usarlos, hay que pagar licencias ($$$).

Evolución Histórica de ISAs

- *¿Por qué Intel o AMD no pueden vender chips para celulares?*
- O, *¿por qué a ARM le costó tanto meterse en el negocio de desktops / servers / supercomputadoras?*

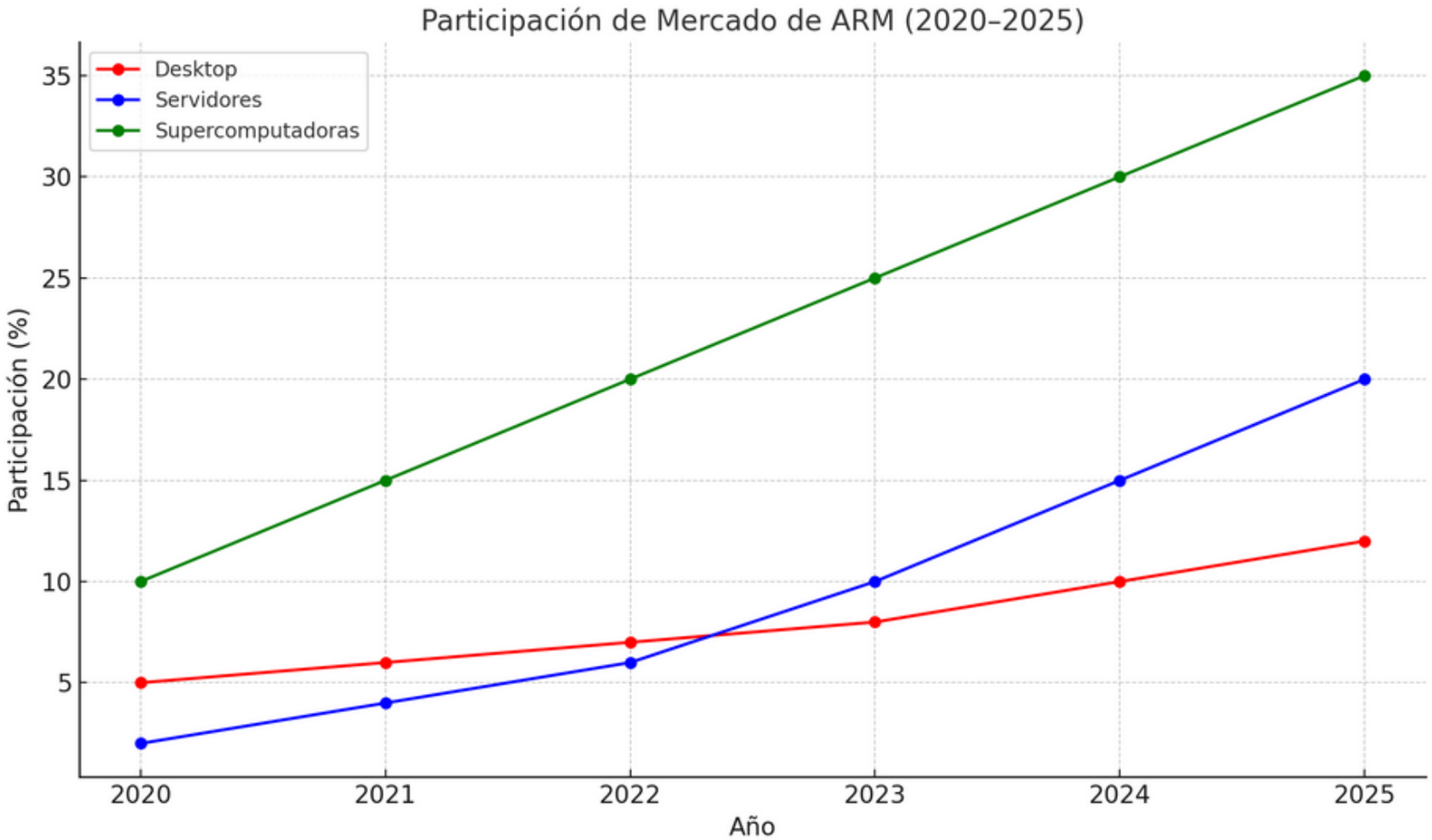
- Porque el **ISA es la interfaz más importante en un sistema de computadoras**.
  - Es donde el software se encuentra con el hardware.
    - "Es donde el software le dice al hardware qué hacer, y el hardware le promete al software cómo se verá el mundo."
  - El ISA define un "ecosistema" para las computadoras.
    - *"Novel hardware is great, but all the performance and features in the world won't do you much good if commonly used software doesn't run on it."*

Evolución Histórica de ISAs

- El ISA es un “contrato implícito” desde hace 60 años:
  - El hardware puede cambiar radicalmente, pero el código que se ejecutaba en la máquina de ayer **tiene que ejecutarse exactamente igual en la de mañana**, eventualmente más rápido.
  - Aunque cambie, el hardware siempre se muestra igual para el software, “hablando el mismo idioma”.
- Es el fundamento base del concepto de **compatibilidad**.
  - *¿Recuerdan las fuerzas en constante movimiento del Tema 01?*
- Romper este contrato prácticamente implica generar un nuevo mercado.
  - Eventualmente para soluciones específicas, no generales.
  - Lo cual lleva a una paradoja: “*No one uses a new piece of hardware because it’s not supported by software, and it’s not supported by software because no one’s using it.*”

Evolución Histórica de ISAs

- Tendencia actual: ARM está ingresando fuertemente en el otro mercado.
  - De la mano de Apple, con sus procesadores M.
  - De la mano de Amazon y NVIDIA en servidores.
  - De la mano de Fugaku, que fue n° 1 del top500 en 2021.



Evolución Histórica de ISAs

- Tendencia actual: hay un tercer ISA creciendo su participación en el mercado (de ARM).
- **RISC-V**.
  - Es el más nuevo (2014), y empezó en 2010 como un proyecto en UC Berkeley, USA.
  - No es un producto. Es (o pretende ser) un estándar, como USB o HTTP.
  - Es **gratuito** y **abierto**. https://github.com/riscv/
  - Se está acercando rápidamente a los valores de performance y consumo de energía de los chips de ARM, con un costo mucho menor.
  - No es manejado por un empresa, sino por una Fundación.
    - Entre sus miembros están Intel, Google, Nvidia, Western Digital, Qualcomm, Samsung, NXP, IBM, Siemens, Huawei, Seagate, Oracle, Sony, y **otros 4200 más** entre empresas y universidades de más de 70 países.

Evolución Histórica de ISAs

- Tendencia actual: hay otro mercado, con otro dominador casi absoluto.
  - GPUs y procesadores para IA.
- **NVIDIA**.
  - De la mano de CUDA (*Compute Unified Device Architecture*), una plataforma de computación paralela (y modelo de programación) que permite utilizar la GPU para tareas de propósito general.
  - Técnicamente, CUDA no es un ISA.
    - Tienen un ISA virtual, denominado PTX (*Parallel Thread Execution*).
    - Y un ISA real, denominado SASS (*Streaming Assembler*), oculto bajo muchos candados.
  - Pero brinda la misma capa de abstracción, con los mismos resultados, sobre un hardware muy diferente.
    - CUDA define un “ecosistema”.
  - Al igual que x86 y ARM, es propietario.

Características de un ISA

- El ISA es una **abstracción** (Gran Idea).
- Permite separar el desarrollo del software del desarrollo del hardware.
  - No hace falta saber cuántos transistores hay, o si hay un pipeline.
- Un ISA es independiente de su implementación.
  - Un mismo ISA puede ser ejecutado en procesadores pequeños para sistemas embebidos o de alta performance para servidores.
- El ISA es conocido como ***el Modelo del Programador de la Máquina.***
- Es la información que se requiere para escribir programas para un determinado procesador.
- Uds. ya vieron un ISA y una implementación del mismo.

Repaso – Componentes de un ISA

1. Celdas de almacenamiento

- Registros de propósito general y especial del CPU.
- Muchas celdas de propósito general de igual tamaño en Memoria.
- Almacenamiento relacionado con los dispositivos de I/O.

2. Formatos de las instrucciones

- Tamaño y significado de los diferentes campos de las instrucciones.
- Modos de direccionamiento soportados.

3. El conjunto (Set) de instrucciones

- Repertorio completo de las operaciones de la máquina.
- Cuatro tipos: movimiento de datos, de procesamiento, control de flujo y misceláneas.
- Usa celdas de almacenamiento, formatos y resultados del ciclo de la instrucción.

4. Naturaleza del ciclo de la instrucción

- Cosas que se hacen independientemente de la instrucción en cuestión.
- Manejo de interrupciones y excepciones
- Convenciones para manejo de la pila.

Repaso de Performance

$$\text{CPU Time} = \frac{\text{Instrucciones}}{\text{Programa}} \times \frac{\text{Ciclos de reloj}}{\text{Instruccion}} \times \frac{\text{Segundos}}{\text{Ciclo de reloj}}$$

- En la cantidad de instrucciones (CI) influyen:
  - El lenguaje de programación, el algoritmo utilizado, el compilador utilizado y el **ISA de la máquina**.
- En el CPI influyen:
  - La **implementación del ISA** y la **organización del hardware**.
  - En menor medida, el compilador.
- En la duración del ciclo (T) o en la frecuencia (f) influyen:
  - La **organización del hardware** y la tecnología.

Impacto de un buen diseño de ISA

- El ISA influye directamente en CI e indirectamente en CPI y en T.
  - Un buen diseño de ISA es simple y ordenado.
- Performance se mejora con Paralelismo y Pipelining.
  - Un buen diseño de ISA tiene en cuenta estos factores de antemano.

Parámetros de Diseño de un ISA

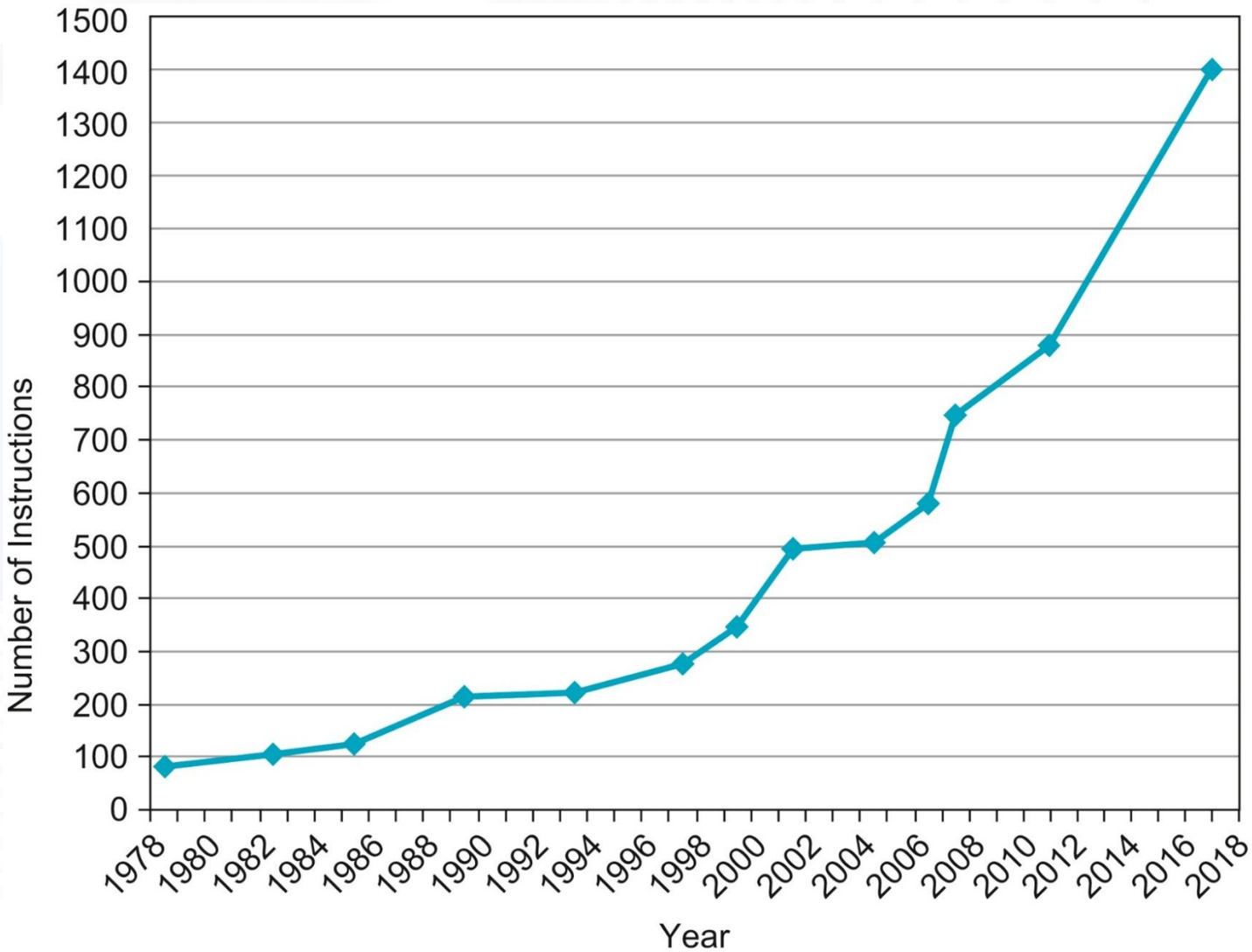
1. Modularidad.
2. Dimensiones.
3. Tipos de datos a soportar.
4. Cantidad de operandos.
5. Accesos a memoria.
6. Celdas de almacenamiento.
7. Cantidad de instrucciones.
8. Modos de direccionamiento.
9. Ortogonalidad y Regularidad.
10. Tipo de Formatos de Instrucciones.
11. Manejo de constantes.
12. Evaluación de condiciones de saltos
13. Tipos de saltos.
14. Distancia de los saltos
15. Manejo de la pila

1. Modularidad

- Define la forma en la que un ISA crece con el tiempo.
  - *¿Por qué debería crecer con el tiempo?*
- Un ISA puede ser **incremental**:
  - Con el tiempo se van agregando instrucciones
  - El nuevo hardware debe soportar lo nuevo además de lo anterior.
  - Fundamental para mantener compatibilidad.
- O puede ser **modular**:
  - Se define un núcleo base, con muchas extensiones (ampliaciones) opcionales.
  - El diseñador de hardware elige cuáles de las extensiones desea implementar y cuáles no.
    - Permite una gran adaptabilidad a diferentes necesidades.

1. Modularidad

- Ejemplo de ISA incremental: **x86_64**
  - Comienza en 1974 (con el 8080) y en 1978 (con el 8086).
  - Desde entonces, cada generación agrega nuevas instrucciones.
    - Agregando nuevas funcionalidades, pero manteniendo compatibilidad con todas las anteriores.



1. Modularidad

- Ejemplo de ISA modular: **RISC-V**
  - **RV32I** es el núcleo base, obligatorio, para manejo de enteros (40 instr.).
  - Extensiones estándar: certeza que no tienen conflictos con otras extensiones.
    - **M**: multiplicación y división (8 instrucciones).
    - **A**: instrucciones atómicas en memoria (11 instr.). Usadas por Sistemas Operativos.
    - **F**: Punto Flotante (precisión simple, 32 bits) (26 instrucciones).
    - **D**: Punto Flotante (precisión doble, 64 bits) (26 instrucciones).
    - **Zicsr**: para manejo de registros de estado y de control (6 instrucciones).
    - **Zifencei**: para sincronización (1 instrucción).
    - **G**: combinación general I+M+A+F+D+Zicsr+Zifencei (118 instr. en total)
    - **C**: agrega versiones comprimidas en 16 bits de RV32I (40 instrucciones).
  - Además, tiene un montón de extensiones no estandarizadas.
  - Y además, permite crear extensiones personalizadas.

2. Dimensiones

- Define el tamaño de las dos variables más importantes de un ISA:
  - **El ancho de sus registros de propósito general** (GPRs).
    - De manera indirecta define el ancho de los buses que harán las transferencias entre registros.
  - **El tamaño de sus direcciones de memoria**.
    - De manera indirecta define el espacio de direccionamiento del ISA.
      - Con direcciones de memoria de n bits se tiene un espacio de direccionamiento de 2^n.
    - *"Solamente hay un error que se puede cometer en diseño de computadoras que es difícil de reponerse: no tener suficientes bits para el manejo y direccionamiento de la memoria"* (Gordon Bell, 1976).

2. Dimensiones

- Actualmente, la mayoría de los microprocesadores de mediana y alta performance son de 64 bits.
  - *¿Cuál es su espacio de direccionamiento?*
- La mayoría de microprocesadores de baja performance o para sistemas embebidos son de 32 bits.
- ISAs más pequeños son difíciles de encontrar hoy en día, pero aún existen.
- *¿Hay ISAs de 128 bits?*

2. Dimensiones

- En el ISA **x86**:
  - El procesador 8080 (1974) era de 8 bits.
  - El 8086 (1978), de 16 bits.
  - El 80286 (1982) extiende las direcciones de memoria a 24 bits.
  - El 80386 (1985), de 32 bits.
  - AMD64 (2003), de 64 bits.
- En **RISC-V**:
  - El formato de nombre es: RV[###][abc…xyz]
    - [###] puede ser {32,64,128}.
  - Así como hay un RV32I hay un RV64I, que tiene exactamente las mismas instrucciones.
  - El estándar ya define un RV128I, pensando en centros de datos futuros con almacenamiento masivo.

3. Tipos de datos

- Define a cuáles tipos de datos dará soporte el ISA (y a cuáles no).
  - Qué es lo que el hardware entenderá de forma nativa.

- Dar soporte a algunos tipos de datos tendrá implicancias en las implementaciones.
  - Por ejemplo, para dar soporte a números de punto flotante suele ser necesario agregar más registros.

3. Tipos de datos

- Ejemplos básicos:
  - Manipulación de bits.
  - Enteros y caracteres:
    - 8-bit (byte), 16-bit (*halfword*), 32-bit (*word*) , 64-bit (*double-word*).
    - Números con signo y sin signo.
    - Los números con signo siempre en complemento a 2.
- Punto flotante, según el estándar IEEE 754:
  - Precisión simple (*float*, 32-bit), doble (*double*, 64-bit) y quádruple (*quadword*, 128-bit).

3. Tipos de datos

- Ejemplos adicionales de **x86**:
  - Provee soporte para datos BCD.
  - Provee soporte para cadenas de caracteres.
    - La instrucción movs copia una cadena de caracteres desde una dirección origen a una destino.
- Ejemplos en procesadores para IA:
  - Están surgiendo muchas variantes de tipos de datos novedosos:
    - INT8, FP16, FP8, **FP4**, bfloat16, TensorFloat32, etc.
    - Difieren entre ellos en precisión y rango.
    - Más importante, ocupación de memoria y ancho de banda.

4. Cantidad de operandos

- Define cuántos operandos manejará el ISA.
- Clasificación basada en las operaciones aritméticas diádicas (que tienen dos operandos y un resultado).
  - C = A + B
- Definen indirectamente cómo y dónde se ubican los operandos, y cómo estos operandos son especificados por la instrucción.
- Un ISA puede ser de 0, 1, 2 o 3 operandos.
  - ISAs de más de 3 operandos hubo, pero no prosperaron.

4. Cantidad de operandos

- **ISA de 0 operandos**:
  - Usa la pila como almacenamiento temporal (ToS ← ToS op SoS).
  - Instrucciones muy compactas.
  - Suele necesitar muchas instrucciones.
  - PUSH A
    PUSH B
    ADD
    POP C
  - Ejemplos actuales: JVM, Java Bytecode.

4. Cantidad de operandos

- **ISA de 1 operando:**
  - Usa un registro de manera implícita, llamado “Acumulador”, que actúa como fuente y como destino (Acc ← Acc op Op1).
  - LOAD A
    ADD B
    STORE C
  - Ejemplo: microcontrolador 8051 y derivados.

4. Cantidad de operandos

- **ISA de 2 operandos:**
  - Permite usar registros o memoria de manera indistinta, como fuente o como destino.
  - Reutiliza uno de los operandos fuente como destino (Op1 ← Op1 op Op2).
  - MOV r1, A
    ADD r1, B
    MOV r1, C
  - Ejemplo actual: **x86_64**.

4. Cantidad de operandos

- **ISA de 3 operandos:**
  - Usa registros como operandos fuente y destino (Op3 ← Op1 op Op2).
  - Necesita previamente cargar los registros
    - lw x1, (A)
      lw x2, (B)
      add x3, x1, x2
      sw x3, (C)
  - Ejemplos actuales: **RISC-V**, ARMv8, ARMv9.

5. Accesos a memoria

- Definir cuántos (y cuáles) de los operandos acceden a memoria.
- Podría permitir que **todos** sus operandos accedan a memoria:
  - ADD A, B, C M(A) ← M(B) + M(C)
    - Instrucciones largas.
      - *¿Cuántos bits aproximadamente?*
    - Programas con pocas instrucciones (bajo CI).
    - Muchos accesos a memoria por instrucción, generan un cuello de botella.
      - *¿Cuántos accesos aproximadamente?*
    - Al ser instrucciones complejas, tienen alto CPI.
    - Ejemplo: VAX.

5. Accesos a memoria

- Podría hacer que **ninguno** de sus operandos acceda a memoria, y trabajen únicamente con registros:
  - ADD r1, r2, r3 R(r1) ← R(r2) + R(r3)
  - Instrucciones cortas.
    - *¿Cuántos bits aproximadamente?*
  - Necesitan más instrucciones para un mismo programa, porque hay que cargar previamente los registros (aumenta CI)
  - *¿Cuántos accesos a memoria por instrucción aproximadamente?*
  - Al ser instrucciones simples, suelen tener bajo CPI.
  - Ejemplo: RISC-V, ARM.

5. Accesos a memoria

- Podría hacer que **alguno** de sus operandos acceda a memoria:
  - ADD r1, A R(r1) ← R(r1) + M(A)
  - ADD A, r1 M(A) ← R(r1) + M(A)
  - En general, permiten que sólo uno de los operandos acceda a memoria.
  - En general, son ISAs de dos operandos, por lo que uno de los operandos fuente se sobreescribe.
  - Instrucciones de longitud y duración diferente, según el operando.
    - Complica la decodificación.
  - Suelen ser las más versátiles, y producen que los programas sean de menor CI.
  - Ejemplo: x86.

5. Accesos a memoria

- *¿Cuál de las alternativas creen que es mejor? ¿En base a qué criterio?*

- Posibles métricas:
  - Tamaño de las Instrucciones.
  - Cantidad de Instrucciones.
  - Tamaño del programa.
  - Cantidad total de accesos a memoria.
  - Tasa de accesos a memoria por instrucción.
  - *¿Cuál es mejor?*

5. Accesos a memoria

- Actualmente, hay un ganador absoluto: los ISA basados en registros.
  - Porque **los registros son mucho más rápidos que los accesos a memoria**.
  - Además porque el compilador puede optimizar su uso, con operaciones en distinto orden y/o en paralelo si tuviese los recursos (Gran Idea).
- **Arquitectura de tipo Load-Store:**
  - Únicamente las instrucciones de tipo Load y de tipo Store pueden acceder a Memoria.
  - Ideal para implementar en un pipeline (Gran Idea).
- En el fondo es una cuestión tecnológica.
  - *¿Qué pasaría si la memoria principal se vuelve tan rápida como los registros?*

5. Accesos a memoria

- Hay tres detalles adicionales a definir.
- **Cantidad mínima de memoria direccionable**.
  - En casi todos los ISA, la memoria es direccionable de a 1 byte.
- ***Endianness***: organización en memoria de datos en múltiples bytes.
  - *Big endian*: byte más significativo en la dirección más baja de memoria.
    - Más fácil de “leer” para los humanos.
  - *Little endian*: byte menos significativo en la dirección más baja de memoria.
    - Ejemplo: x86_64, RISC-V.
- **Alineamiento de la memoria**: cómo acceder a un dato de 32 bits que empieza en la dirección 0x03.
  - Se puede permitir, pero con penalización a la performance (x86).
  - Se puede prohibir, y lanzar una excepción en ese caso para que se corrija por software (RISC-V)

6. Celdas de almacenamiento

- Definir cantidad y uso de las celdas de almacenamiento.
- Normalmente, se define la cantidad de registros de propósito general (GPRs) que soportará el ISA.
  - Atención, no se define el ancho de esos registros.
    - Eso ya está definido en la dimensión del ISA.
  - Tener más registros casi siempre es más ventajoso.
  - Tener más registros es más costoso.
  - Tener más registros puede complicar los formatos de instrucción.
- Ejemplos:
  - RISC-V posee 32 GPRs (31 siendo muy correctos).
  - x86 posee 16 registros, de los cuales 8 son GPRs.
    - Recién en 2023 ampliaron a 32 registros.

6. Celdas de almacenamiento

- Además de los GPRs, se deben definir registros de propósito específico, si los hubiere.
  - PC es el más común.
  - Acumulador, Stack Pointer, entre otros.
  - Registros de punto flotante.
- Deben existir registros de estado y de control.
  - Seguramente varios por cada modo de privilegio.
- Pueden especificarse registros adicionales para monitoreo de variables performance.
- Debe especificarse también el mapa de memoria:
  - Bloques de memoria reservados, para el programa, para los datos estáticos y dinámicos (*heap*), para el *stack*.
  - Bloques para dispositivos de entrada/salida.

7. Cantidad de instrucciones

- Definir cuántas instrucciones debe incluir el ISA.
- Vimos que no hay un número máximo, con los ISA incrementales.
  - *¿Habrá un número mínimo?*
  - *¿Habrá un número óptimo?*
- Lo importante es que el ISA sea **completo**.
  - Que pueda hacer todas las operaciones que se requieran.
  - Se probó que con una sola instrucción alcanza ☺
    - SUBLEQ (*Subtract and Branch if Less than or Equal to zero*).
  - **movfuscator**, compilador de C para x86 que corre DOOM, utilizando únicamente instrucciones **mov**.

7. Cantidad de instrucciones

- La cantidad en sí no es crítica, sino que sea **eficiente**:
  - Hacer más rápido el caso más común (Gran Idea).
  - Las instrucciones más usadas deben ser las más rápidas.
- Entonces, si una instrucción es lenta…
  - …y obliga a que las demás también sean lentas...
  - …no incluirla.
  - **Principio de Diseño: no perjudicar a la mayoría por culpa de unas pocas.**
- *¿Y si esa instrucción es necesaria para que el set sea completo?*
  - Hacer una **pseudoinstrucción** que la reemplace.
  - Y que el ensamblador la expanda.

7. Cantidad de instrucciones

- Lo mejor es **medir**, en varios programas, cuáles son realmente las instrucciones más usadas, siempre considerando frecuencia dinámica.

Rango	Modos de direccionamiento	Promedio
1	Load	22%
2	Conditional Branch	20%
3	Compare	16%
4	Store	12%
5	Add	8%
6	And	6%
7	Sub	5%
8	Move Reg. to Reg.	4%
9	Call	1%
10	Return	1%
	Total	96%

- ¡Las 10 instrucciones más usadas cubren el 96% de los casos!
- Y además, ¡son de las más simples!

8. Modos de direccionamiento

- Definir cuáles modos de direccionamiento tendrá el ISA.

- Son el soporte que brinda el ISA para una forma útil y eficiente de determinar dónde se encuentra un operando.
- Diferentes modos de direccionamiento resuelven distintos problemas de los lenguajes de alto nivel.
  - Algunas variables se conocen en tiempo de compilación.
  - Otras se conocen recién en tiempo de ejecución (p.ej: punteros)
  - Puede ser necesario calcular las direcciones (p.ej: componentes de una tabla o vector, o estructura o registro)
  - Es posible almacenar valores constantes en la misma instrucción, o adyacente, sin usar memoria.

8. Modos de direccionamiento

- Por ejemplo, un acceso a A[i]:
  - A es un puntero a la dirección base del vector, es constante.
  - i es el índice variable.
    - No puede superar el tamaño del vector.
  - Para poder acceder a A[i] se necesita una “aritmética de direcciones”*:*
    - Dirección base de A[] + i * tamaño de los elementos de A[]
- Una dereferencia a un puntero (*ptr = 10):
  - La dirección del puntero ptr seguramente esté contenida en un registro.
  - Se actualiza el contenido de la dirección de memoria apuntada por un registro.
    - M[R[x]] ← 10
- Un acceso al campo de un objeto (alumno.dni):
  - Alumno es la dirección base del objeto, dentro del espacio de memoria del *heap*.
  - Dni es un desplazamiento de valor fijo dentro de la estructura del objeto.

8. Modos de direccionamiento

- **Implícito.**
  - El operando no se menciona. El hardware sabe donde está por la naturaleza de la instrucción.
  - RET (x86): toma la dirección de retorno de una subrutina del tope de la pila.
  - CLC (x86): borra el bit de *carry* del registro de estado EFLAGS.
- **Inmediato**.
  - El operando es una constante que forma parte de la instrucción.
  - addi t0, t0, 1 (RISC-V)
  - add eax, #6765 (x86)
- **Entre registros**.
  - Los operandos están en registros del procesador.
  - add t0, t1, t2 (RISC-V)
  - mov eax, ebx (x86)

8. Modos de direccionamiento

- **Registro indirecto**.
  - El operando es un registro que contiene la dirección de memoria donde está el dato.
  - lw t0, 0(t2) (RISC-V)
  - mov eax, [ebx] (x86)
- **Directo (o Absoluto).**
  - El operando es la dirección de memoria donde está el dato.
  - add eax, [#6765] (x86)
  - LDR R1, [0x3F8] (ARM)
- **Indirecto (o Indirecto en Memoria)**.
  - El operando es una dirección de memoria, la cual contiene a su vez la dirección de memoria del dato final.
  - MOVL @(R1), R2 (VAX) R[R2] ← M[M[R[R1]]]

8. Modos de direccionamiento

- **Indexado (Base + Desplazamiento)**.
  - Un operando es un registro que se usa como base y otro operando es una constante que se usa como desplazamiento.
  - La dirección de memoria se calcula como la suma de los dos operandos.
  - lw t0, 8(sp) (RISC-V) R[t0] ← M[R[sp] + 8]
  - mov ebx, [edi + 45] (x86)
- **Indexado (Base + Índice)**.
  - Similar al anterior, pero el segundo operando es otro registro en vez de una constante.
  - mov eax, [rbx + rcx] (x86)
  - LDR R0, [R1, R2] (ARM)

8. Modos de direccionamiento

- **Escalado.**
  - Similar al indexado, pero donde el índice es multiplicado por un factor de escala.
  - El factor de escala puede ser 2, 4 u 8, y representa el tamaño del tipo del dato.
  - mov eax, [rbx + rcx * 8] (x86)
  - LDR R0, [R1, R2, LSL #3] (ARM)
  - Los ISA de ARM implementan el factor de escala como un registro de desplazamiento ubicado en una de las entradas de la ALU (*Barrel Shifter)*.
- **Indexado Escalado.**
  - Similar al escalado, pero además con un desplazamiento.
  - Lo más completo para recorrer cualquier tipo de estructura.
  - mov eax, [rbx + rcx * 8 + 0x20] (x86)

8. Modos de direccionamiento

- **Post-incrementado.**
  - Se accede a la dirección apuntada por un registro, y luego se incrementa automáticamente el valor del registro.
  - Se usa en combinación con otros modos.
  - LDR R0, [R1], #4] (ARM)
  - PUSH R0 (ARM)
- **Pre-incrementado (o Pre-decrementado).**
  - Similar al anterior, pero modificando primero el valor del registro.
  - STR R0, [R1], #-4]! (ARM)
- **Relativo (o Relativo a PC).**
  - La dirección destino de un salto se calcula sumando un desplazamiento al valor actual de PC.
  - jne 0x0A (x86)
  - beq t0, t1, fin (RISC-V)

8. Modos de direccionamiento

- Entre todas esas alternativas, *¿cuáles conviene elegir?*
- Igual que antes, lo mejor es **medir** su uso en un conjunto de programas:

Modos de direccionamiento	Promedio	Mínimo	Máximo
Indexado con Desplazamiento	42%	32%	55%
Inmediato	33%	17%	43%
Registro Indirecto	13%	3%	24%
Escalado	7%	0%	16%
Memoria Indirecta	3%	1%	6%
Otros	2%	0%	3%

- ¡El 88% de las veces se usan solamente 3 modos de direccionamiento!

9. Ortogonalidad y Regularidad

- Definir si cualquier instrucción puede utilizar cualquier modo de direccionamiento.

- Un ISA tiene **ortogonalidad** cuando sus operandos pueden utilizar cualquiera de sus modos de direccionamiento.
  - El caso extremo fue la computadora VAX.
- Actualmente, la ortogonalidad es limitada.
  - Un operando puede usar algunos modos de direccionamiento. Otro operando, otros modos.
  - Algunas instrucciones de x86 funcionan solamente con algunos registros.
    - ¡Pesadilla para el compilador! Lo ideal es que todos los registros sean iguales.

9. Ortogonalidad y Regularidad

- Un ISA tiene **regularidad** si sus instrucciones mantienen una coherencia predecible.
  - Si hay una instrucción de suma, se espera que haya una de resta.
  - Si hay un desplazamiento a la izquierda, se espera que haya uno a la derecha.
  - Si las instrucciones de tipo load usan un modo de direccionamiento, se espera que las instrucciones de tipo store usen el mismo.
- La regularidad:
  - Se obtiene mediante formatos de instrucción uniformes, con campos en los mismos lugares**.**
  - Simplifica la decodificación.
  - Permite ganar tiempo,
    - Al saber dónde están algunos campos, puedo ir haciendo cosas de manera simultánea.
- **Principio de Diseño: la simplicidad favorece la regularidad.**

10. Formatos de instrucciones

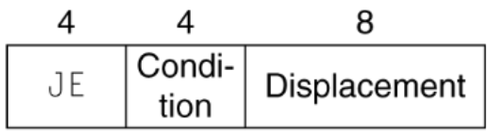
- Definir cómo será la longitud de los formatos de las instrucciones.
- Dos alternativas: fija o variable.
- Cuando los formatos son de **longitud fija**, todos tienen la misma longitud.
  - Idealmente, con campos similares.
  - Facilitan la decodificación.
  - Al ser más simples y con mayor regularidad, suelen tener mejor performance.
  - Tienen una menor posibilidad de ampliación.
    - En algún momento se acabarán los bits disponibles.
  - Ejemplo: RISC-V.

10. Formatos de instrucciones

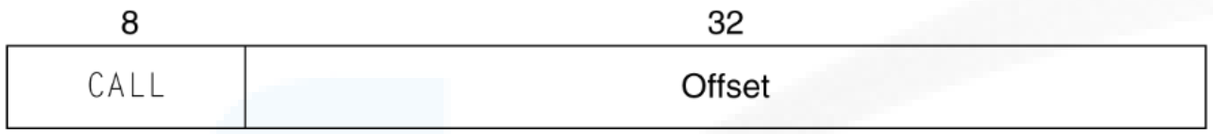
- Los formatos de **longitud variable** dependen de la cantidad de operandos y de sus modos de direccionamiento.
  - Buscan como objetivo minimizar el tamaño del programa en memoria.
  - Permiten una mejor densidad de código.
    - Necesitan menos instrucciones para hacer una tarea.
  - No tienen limitaciones para su ampliación.
  - Ejemplo: x86

10. Formatos de instrucciones

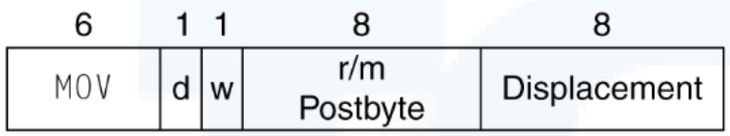
a. JE EIP + displacement



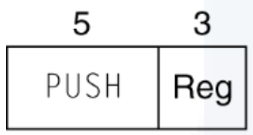
b. CALL



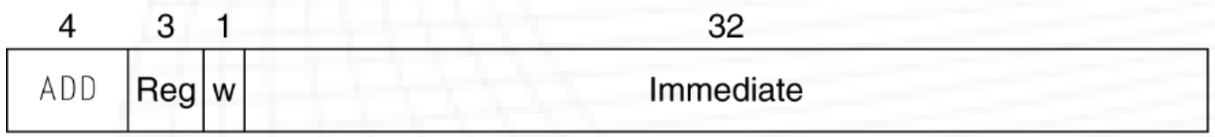
c. MOV EBX, [EDI + 45]



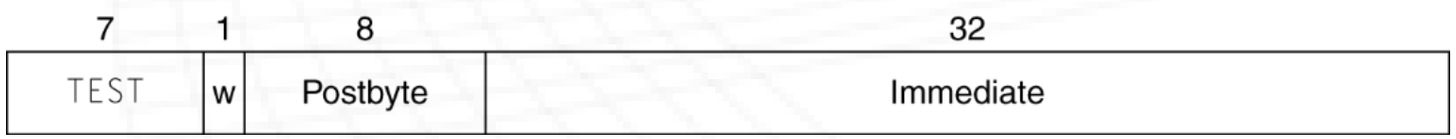
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- Tiene instrucciones con longitudes desde 1 byte hasta 15 bytes.
- Nótese los distintos tamaños de opcode.
- La decodificación es muy compleja y costosa.

10. Formatos de instrucciones

- *¿Cuál de las dos alternativas es más conveniente?*
- Actualmente, está de moda que los ISA posean formatos con **dos longitudes fijas**.
  - Por ejemplo, instrucciones de 32 y de 16 bits únicamente.
  - Intenta obtener las ventajas de ambos esquemas.
  - Ejemplo: RISC-V con su extensión C.

- Los formatos de longitud fija suelen usar campos adicionales de la instrucción como **extensión del opcode**.
  - Los campos función de RISC-V.
  - En estos casos, el opcode identifica grupos de instrucciones.
  - Son una manera de sobrellevar su limitación a la posibilidad de ampliación.

11. Manejo de constantes

- Definir tamaños de las constantes y cómo se las manejará.
- Las constantes pueden ser usadas de diversas maneras.
  - Como operandos:
    - addi sp, sp, 4 R[sp] ← R[sp] + 4
  - Como desplazamientos:
    - lw t0, 16 (sp) R[t0] ← M[R[sp] + 16]
  - Como direcciones de memoria:
    - movabs rax, [0x1122334455667788] R[rax] ← M[0x1122334455667788]
- *¿Qué tamaño convendría asignarles?*
  - Una vez más, la respuesta pasa por **medir** su uso.

11. Manejo de constantes

- Para operandos, el 60% de las veces alcanza con 8 bits, y el 80% alcanza con 16 bits.
- Para desplazamientos, sólo el 1% supera los 16 bits.
- La constante más utilizada (por lejos) es el valor cero.
- Manejando una constante de entre 12-16 bits dentro de algún campo de un formato de instrucción suele ser suficiente.
  - Formato de instrucción Tipo I en RISC-V.
- Es una buena idea que el valor cero esté disponible siempre.
  - En un registro, cableado, para que sea más rápido.
  - Hacer más rápido el caso más común.
  - Se pierde un registro, pero se gana en eficiencia.

11. Manejo de constantes

- Sin embargo, el ISA igual debe dar soporte a constantes más largas.
  - En un formato de longitud variable es simple, se agregan campos.
  - En un formato de longitud fija, se podría pensar en instrucciones más largas.
    - En contra de un Principio de Diseño ya visto
      - No perjudicar a la mayoría por culpa de unas pocas.
    - Se soluciona utilizando combinaciones de instrucciones.
      - Formato Tipo U en RISC-V.
      - Aumenta CI.
- En general, definir el tamaño de las constantes genera un compromiso de diseño.
- **Principio de diseño: un buen diseño requiere compromisos.**

12. Condiciones para saltos

- Definir cómo se evaluarán las condiciones para los saltos condicionales.
- La alternativa más común es utilizar un registro de estado que almacene los **códigos de condición** (*flags*).
- Instrucciones aritméticas setean estos códigos, las instrucciones de salto los evalúan.
- Los códigos de condición más comunes son O (overflow), C (carry), N (negative), Z (Zero), pero puede haber más.
- Ejemplo: x86 posee un registro EFLAGS, una instrucción cmp (compara dos operandos) y una instrucción je (*jump if equal*, salta si Z=1).

12. Condiciones para saltos

- Sin embargo, el uso de códigos de condición genera dos inconvenientes:
  - No todas las instrucciones modifican los códigos de condición.
    - Hay que asegurar que entre la instrucción que setea el CC y el salto que lo lee no haya instrucciones intermedias, o que no modifiquen los CC si las hubiera.
  - Dificulta su implementación en alta performance.
    - Porque los CC generan dependencias entre las instrucciones.
- Es posible evaluar condiciones de salto sin usar códigos de condición.
  - Instrucciones que comparan registros, y guardan resultado en otro registro.
  - Saltos que realizan la comparación entre registros.
  - Ejemplo: instrucciones slt (*set less than*)y beq (*branch if equal*) en RISC-V.

13. Tipos de saltos

- Definir a qué tipos de saltos dará soporte el ISA.
- Saltos absolutos: reciben la dirección completa del salto.
  - Pueden ser directos, en los que la dirección es parte de la instrucción.
    - J 1000 PC ← 1000
  - O indirectos, en los que la dirección está contenida en un registro.
    - jr ra PC ← R[ra]
    - Muy usados en sentencias switch y para retorno de subrutinas.
- Saltos relativos: en los que la dirección es calculada de manera relativa al PC actual.
- En programas de enteros, el salto condicional por igual/distinto es por lejos el más usado.
  - Hacer más rápido el caso más común.

14. Distancia de los saltos

- Definir el tamaño de las direcciones de destino de las instrucciones de salto.

- La mayoría de los destinos de los saltos son muy cercanos a la propia instrucción de salto.
  - En lazos y estructuras de decisión.
  - Por ello, es muy utilizado el modo de direccionamiento relativo a PC.
  - Con al menos 8 bits para desplazamiento.
    - Permite saltar 128 **instrucciones** hacia adelante o hacia atrás.

14. Distancia de los saltos

- *¿Qué hacemos si queremos saltar más lejos?*
- Se deben soportar instrucciones de salto con rangos más grandes.
  - Por ejemplo, para llamados y retornos a subrutinas.
- *¿Y si es necesario saltar aún más lejos?*
  - Caso poco común, se utiliza una combinación de dos instrucciones (doble salto, o trampolín).
  - Solución similar a casos análogos ya vistos.

15. Manejo de la pila (*stack*)

- Es un método estándar para pasaje de parámetros entre distintos programas.
  - Además de usar los registros, obviamente.
- Es una estructura que se implementa en memoria.
- Usado para guardar el estado de una tarea en caso de excepciones o interrupciones.
- Un ISA puede manejar el *stack* de dos maneras:
  - **Manejo explícito**:
    - Especifican un registro especial: *Stack Pointer* (SP).
    - Especifican instrucciones dedicadas: PUSH y POP.
  - **Manejo implícito**:
    - Utilizan un registro de propósito general (con uso reservado por convención).
    - Realizan manualmente incrementos y decrementos de punteros.

15. Manejo de la pila

- Esta decisión impacta en la manera en la que el ISA maneja las subrutinas.
- Un ISA con manejo explícito del *stack* posee instrucciones específicas:
  - CALL
    - Guarda automáticamente en el *stack* el PC de la siguiente instrucción, los códigos de condición (si tiene) y los registros de estado.
    - Luego salta a la subrutina destino.
    - Los argumentos para la subrutina deberían haber sido pasados antes de la llamada.
  - RETURN
    - Antes de retornar, la subrutina debería restaurar el estado inicial si es que lo modificó.
    - Luego esta instrucción restaura los códigos de condición, los registros de estado y el PC (saltando a la instrucción sigiuente al llamado).

15. Manejo de la pila

- Un ISA con manejo implícito del *stack* no posee instrucciones específicas:
  - Los llamados a subrutinas se realizan mediante saltos que “enlazan” (guardan el PC en un registro de propósito general).
    - jal rd,offset R[rd] = pc+4; pc += offset
  - Los retornos se realizan mediante saltos al valor contenido en un registro.
    - jalr x0, 0(ra)

Métricas para diseño de ISA

- En muchos de los parámetros, vimos la importancia de medir las variables.
  - Una métrica adecuada provee buenos objetivos de diseño.
  - **No se puede mejorar lo que no se puede medir.**
- Además, los procesadores para laptops / desktops / servidores / celulares / sistemas embebidos son esencialmente distintos.
  - Tienen distintos requerimientos de Performance, de Consumo de Energía y de Área (Costo).
  - Inclusive, en un sistema embebido uno conoce exactamente el código a ejecutar, por lo que idealmente podría seleccionar cuáles características de un ISA incluir.

Métricas para diseño de ISA

- **Performance** es la métrica más importante, pero hoy en día también es importante considerar estas otras métricas:
- **Costo:** un ISA pequeño y simple favorece implementaciones pequeñas.
- **Simplicidad:** menos tiempo y costo de implementación, menor documentación, mejor curva de aprendizaje para clientes.
- **Aislamiento de Arquitectura e Implementación:** No favorecer implementaciones particulares en detrimento de implementaciones futuras.
- **Espacio para crecer:** nuevas aplicaciones requieren nuevas instrucciones.
- **Tamaño del programa:** crítico en sistemas embebidos.
- **Facilidad de programar, compilar y linkear:** una buena cantidad de GPR; direccionamiento relativo al PC; duración de las instrucciones.

Justificación de RISC-V

- RISC-V es un ISA que **cumple con todos los criterios de un buen diseño de ISA**.
- Considerando la tecnología actual, y considerando Performance como objetivo principal.
  - Es modular, usamos lo que necesitamos. De 32 o 64 bits.
  - Brinda soporte a los tipos de datos más comunes.
  - De 3 operandos. Arquitectura Load-Store.
  - 32 registros de propósito general, una cantidad adecuada.
  - Pocas instrucciones, simples, las más usadas.
  - Pocos modos de direccionamiento, los más usados.
  - Muy regular.
  - Formatos de instrucción de longitud fija, con extensión comprimida..
  - No usa códigos de condición.
  - Decisiones de compromiso para manejo de constantes, tipos de saltos y distancias de los saltos.

RV32C – La extensión C: Compressed

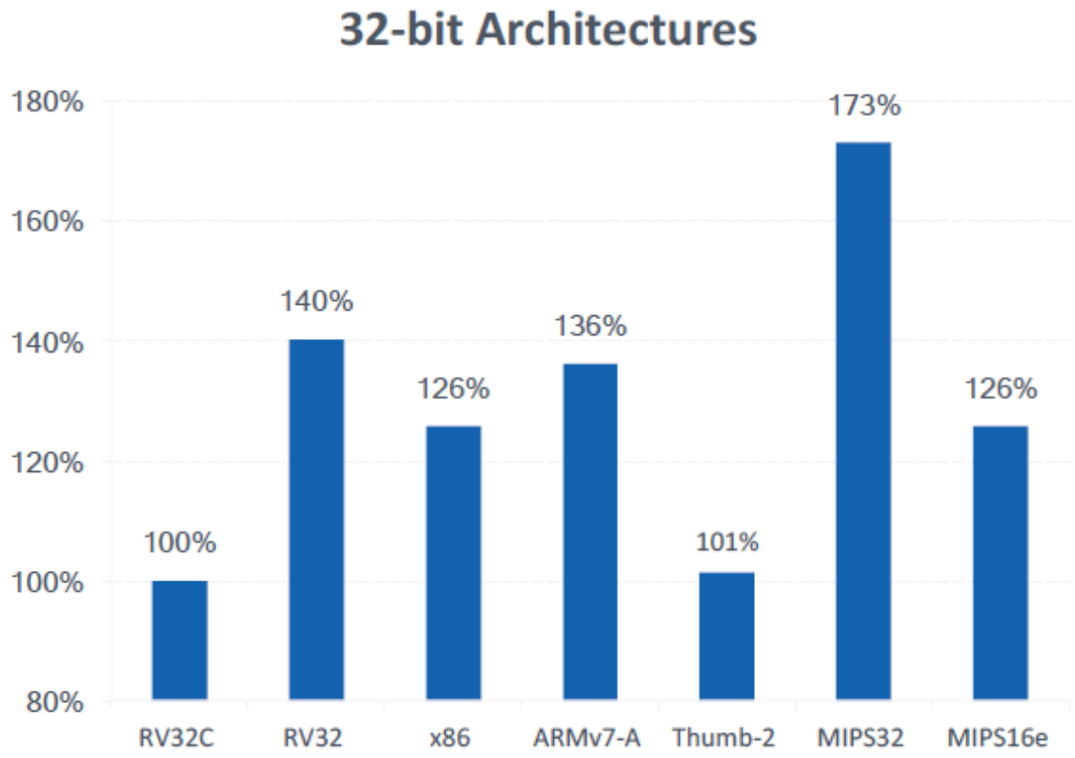
- Todas las instrucciones de RV32I se mapean a una versión equivalente de 16 bits.
  - El mapeo es 1:1, cada instrucción comprimida tiene una equivalente.
  - Acceden a los 10 registros más populares, sobrescriben un operando y usan constantes más pequeñas.
  - Obviamente, define nuevos formatos de instrucción.
- **Idea clave:** el programador escribe las instrucciones “normales”, y el ensamblador (y/o enlazador) se encarga de elegir la versión comprimida cuando sea posible.
- La extensión también es válida para RV64.

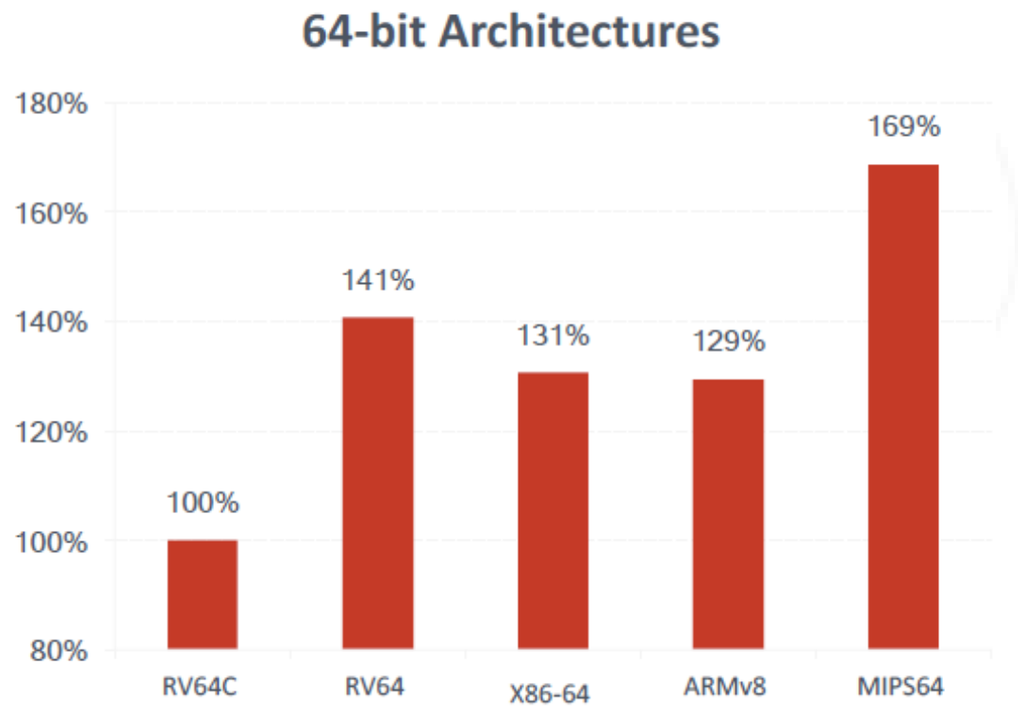
RV32C – La extensión C: Compressed

- Muy usada en Sistemas Embebidos, para reducir costos.
  - Métrica: Tamaño del programa (también conocido como densidad de código).
- Y también en los demás tipos de computadoras.
  - Porque pueden mejorar la performance del sistema de memoria (Caché).
  - Métrica: performance, aunque podría aumentar la cantidad dinámica de instrucciones.
- En el benchmark SPECint2006, al utilizar las extensiones GC, en promedio se buscan 3 bytes por instrucción.
  - ¡Lo que implica que el 50% de las instrucciones puede comprimirse!

RV32C – La extensión C: Compressed

- Comparación de tamaño de código con otros ISA, utilizando SPECint2006, siempre con GCC:





- RISC-V genera el código más pequeño.
  - *¿Por qué no usar siempre esta extensión?*

Instrucciones comprimidas en otros ISAs

- Misma idea: proveer un formato de instrucción alternativo para una misma funcionalidad, quitando bits de alguna de las siguientes maneras:
  - Usando uno de los operandos como fuente y destino.
  - Haciendo que uno de los operandos sea implícito.
  - Limitando la cantidad posible de registros.
  - Limitando la cantidad posible de modos de direccionamiento.
  - Usando constantes más pequeñas.
- Siempre se debe respetar el mapeo 1:1.

Falacias comunes sobre ISAs

- Usar *assembler* mejora la performance
  - Los compiladores modernos son bastante buenos para los procesadores actuales.
  - Más líneas de código, más posibilidad de errores, y menos productividad.
  - Entonces, *¿para qué sirve assembler?*

- No se puede diseñar el hardware sin conocer la interfaz que debe brindar.
- Alguien tiene que diseñar los compiladores.
- Conocer detalles de implementación de un ISA permite entender mejor la performance y escribir mejores algoritmos y programas.
  - Recuerden ejemplo de ventaja competitiva del Tema 02.

Resumen Final

- Dos ISAs dominantes: x86_64 y ARM, cada uno en su mercado.
- El ISA es la interfaz más importante en un sistema de computadoras.
  - Es la base del concepto de compatibilidad.
  - Es el modelo del programador, una abstracción, independiente de las implementaciones.
  - ISA = celdas de memoria + formatos de instrucciones + conjunto completo de instrucciones + particularidades
- Un ISA impacta en las tres variables de la ecuación de performance.

Resumen Final

- Para diseñar un ISA es necesario tomar decisiones sobre 15 parámetros de diseño.
- Importancia de medir para tomar decisiones.
- Performance es la métrica principal, pero no la única.
  - Tener en cuenta otras métricas, según el uso y el tipo de computadora.
- Tener siempre presente los Principios de Diseño.

Principios de Diseño

- La simplicidad favorece la regularidad.
- Hacer más rápido el caso más común.
- No perjudicar a la mayoría por culpa de unas pocas.
- Un buen diseño requiere compromisos.