2. DISEÑO DEL REPERTORIO DE INSTRUCCIONES MÁQUINA

- 2.1. Tipos de juegos de instrucciones
- 2.2. Compiladores, repertorio de instrucciones y rendimiento
- 2.3. Computadores de juego reducido de instrucciones: RISC

2.1. Tipos de juegos de instrucciones

- **Característica** más **definitoria** de un juego de instrucciones:
  - Modo de **almacenamiento interno** de la UCP
- **Modos básicos** de almacenamiento interno (por orden de antigüedad de aparición):
  - **Acumulador**. Ejs.: EDSAC, PDP-8, MC6809
  - **Pila**. Ejs.: B5500, HP 300070, JVM (Java Virtual Machine)
  - **Registros**. (Ejs. en pág. 4)

Número de operandos explícitos en instrucciones aritmético-lógicas binarias

- **Pila**: **0**. Los operandos se toman de la pila y el resultado se coloca en la pila
- **Acumulador**: **1** (de memoria). El otro operando fuente está en el acumulador que sirve también como destino
- **Registros**: **3** (**ó 2** si el operando destino coincide con uno de los operandos fuente)

Procesadores basados en registros: lugar de almacenamiento de los operandos

- Tres posibilidades según el **número máximo de operandos de memoria** en las instrucciones aritmético-lógicas binarias:
  - Procesadores **registro-registro, *load-store* o de carga-almacenamiento**: **0**. Ejs.: ARM, MIPS, PowerPC
    - ✓ Sólo las instrucciones *load* y *store* pueden mover datos entre memoria y registros
    - ✓ Hoy día casi todos los procesadores son de este tipo
  - Procesadores **registro-memoria**: **1**. Ej.: IBM 360/370
  - Procesadores **memoria-memoria: 2 ó 3**. Hoy en día no se fabrican. Ej.: VAX

Código correspondiente a `D:=A*B-C` para distintos tipos de procesador

Pila	Acumulador	Load-store con 2 operandos explícitos	Registro-memoria con 3 op. explic.
PUSH A PUSH B MPY PUSH C SUB POP D (Notación postfija: ***AB*C***+)	LOAD A *; Ac ← A* MPY B SUB C *; Ac ← Ac - C* STORE D	LOAD R1,A LOAD R2,B MPY R1,R2 LOAD R3,C SUB R1,R3 STORE D,R1 (En este ejemplo, primer operando explícito = operando destino)	LOAD R1,A MPY R4,R1,B SUB R4,R4,C STORE D,R4 (En este ejemplo, primer operando explícito = operando destino)

Juegos de instrucciones orientados a acumulador y a pilas: ventajas e inconvenientes

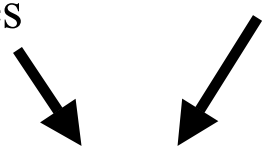
	Ventajas	Inconvenientes
Acumulador	❖**Hardware sencillo** ❖Instrucciones cortas	❖**Mayor tráfico memoria-UCP** que en los de pilas o registros (mínima capacidad de almacenamiento en la UCP)
Pilas	❖La mejor **densidad de código** (dado que las instrucciones son muy cortas) ❖Las expresiones aritméticas se compilan con máxima facilidad (a partir de la notación postfija)	❖Tráfico memoria-UCP alto (sólo hay dos registros rápidos) ❖El compilador tiene más difícil la generación de código eficiente (ya que el acceso a la pila **no** es **aleatorio**)

Juegos de instrucciones orientados a registros: ventajas e inconvenientes

	Ventajas	Inconvenientes
En general	❖**Menor tráfico memoria-UCP => Se acelera el programa** (los registros son más rápidos que la memoria) ❖El **compilador** tiene más **facilidad** para **generar código eficiente**. Sobretodo porque: ▪ Puede **ubicar variables en registros** => Se reduce más el tráfico memoria-UCP	❖**Peor densidad de código** que los de pilas o acumulador (más operandos explícitos)
Del tipo registro-registro	❖Codificación simple de las instrucciones ❖Instrucciones de **igual longitud y tiempos** de **ejecución** similares	❖Se necesitan **más instrucciones** para un mismo algoritmo
Del tipo registro-memoria	❖Ahorro de instrucciones *load* y *store* => mejor densidad de código ❖Formato de instrucciones sencillo, pero menos => implementación y compilación más complejas	❖Más variación en los tiempos de ejecución

2.2. Compiladores, repertorio de instrucciones y rendimiento

- Históricamente, la importancia de los compiladores ha sido creciente
- Actualmente, casi todas las instrucciones máquina que ejecutan los computadores son obtenidas por compiladores
- Una compilación eficiente repercute muy positivamente en el TUCP (básicamente, reduciendo IE)



- Es fundamental **diseñar** los **repertorios de instrucciones** de modo que la **compilación** pueda ser **eficiente**

Estructura de los compiladores y principales optimizaciones

- Para **facilitar** su **diseño y verificación** los compiladores se dividen en **fases o pasos**
  - En cada paso el compilador **lee el programa** resultante del paso anterior **y lo transforma** en una representación de más bajo nivel
- Diversos compiladores pueden **tener en común algunas fases** => menor esfuerzo de diseño
- Inconveniente: **problema de ordenación de fase**
  - En una fase el compilador debe tomar una decisión cuyo acierto depende de una fase posterior
    - ✓ Revisar la decisión es imposible por el aumento de complejidad y tiempo de compilación que conllevaría

Fases de los compiladores (1 d 4)

- **Formato para lenguaje**
  - El programa fuente se transforma en una forma intermedia común
  - Esta fase es dependiente del lenguaje de alto nivel e independiente de la máquina
- **Optimizador de alto nivel**
  - Ej. de optimización: integración de procedimiento la llamada a un procedimiento por el cuerpo del procedimiento)
  - Fase algo dependiente del lenguaje y, en su mayor parte, independiente de la máquina

Fases de los compiladores (2 d 4)

- **Optimizador global** (poca dependencia del lenguaje y de la máquina). Incluye:
  - **Optimizaciones locales**
    - ✓ Son las que se aplican a **bloques básicos** =
      - Secuencias de instrucciones sin saltos (salvo quizá la última) ni destinos de salto (salvo quizá la primera)
    - ✓ Ejemplos:
      - Eliminar subexpresiones comunes (la 1ª. vez se calcula y almacena su valor, las sucesivas veces se sustituye la expresión por el valor almacenado)
      - Propagación de constantes (sustituir variable a la que se ha asignado una constante, por su valor)

Fases de los compiladores (3 d 4)

- **Optimizaciones globales**
  - ✓ Extienden las locales a través de los saltos
  - ✓ Ejemplos:
    - Eliminación global de subexpresiones comunes (como la local pero no restringida a bloques básicos)
    - Movimiento de código (saca del bucle el código que calcula el mismo valor en todas las iteraciones)
- **Ubicación de registros**
  - ✓ Ubicar las variables en registros y no en memoria
  - ✓ Contribuye a **hacer útiles otras optimizaciones**
    - Ej.: la eliminación de subexpresiones comunes es útil si el valor que se guarda se almacena en un registro (observa el problema de ordenación de fase)

Fases de los compiladores (4 d 4)

- **Generador de código**
  - Puede incluir o ir seguido de un ensamblador
  - Genera las instrucciones máquina o ensamblador definitivas
  - Realiza optimizaciones dependientes de la máquina. Ejs.:
    - ✓ Reducción de potencia (por ejemplo, sustituir el producto por una constante por sumas y desplazamientos)
    - ✓ Planificación de la segmentación encauzada o *pipeline* (se reordenan las instrucciones para mejorar el rendimiento de la UCP encauzada evitando ciclos de espera)
- Los pasos de optimización son **opcionales**
  - Se pueden saltar cuando se prefiere una compilación rápida

Efecto sobre el rendimiento de diversas optimizaciones del compilador

Optimizaciones realizadas al compilar ciertos programas en Pascal y Fortran	Ganancia del TUCP obtenida respecto a una compilación sin optimizaciones
Integración de procedimiento	1,10
Optimizaciones locales	1,05
Optimizaciones locales y globales	1,14
Opt. locales y ubicación de registros	1,26
Opt. locales y globales y ubicación de registros	1,63
Opt. locales y globales, ubicación de registros e integración de procedimientos	1,81

- La **ubicación de registros** es la optimización que más acelera el código ejecutable y, por tanto, **la más importante**
- Se confirma que contribuye a **hacer útiles otras optimizaciones**

Ubicación de registros

- Se basa en algoritmos de **coloreado de grafos**
  - Color → registro;; nodo → variable
  - Variables con rangos vivos solapados → nodos adyacentes → colores distintos
  - No se conocen algoritmos óptimos => se usan **algoritmos heurísticos**
    - ✓ **Mínimo número de registros** aceptable: **16**
      - Con menos registros es probable que el algoritmo falle (demasiadas variables sin ubicar en registros)

Optimizaciones de compilación y CPI

- Las optimizaciones afectan a las frecuencias de los distintos tipos de instrucciones y, por tanto, al CPI:
  - Aumenta el porcentaje de saltos y de instrucciones aritmético-lógicas
  - Disminuye el porcentaje de cargas y almacenamientos

(Se compiló el programa TeX)	Millones instrucciones ejecutadas		Porcentajes instrucciones ejecutadas	
Tipos de instrucciones	Sin optim.	Con optim.	Sin optim.	Con optim.
Saltos-bifurcaciones	13	12	12%	14%
Instrucciones ALU	50	41	46%	49%
Load y store	45	30	42%	36%
Total	108	83	100%	100%

Características de los juegos de instrucciones que facilitan una compilación eficiente (1 d 3)

- Regularidad
  - Las características del juego de instrucciones deben ser **ortogonales**, es decir, **independientes** entre sí
    - ✓ Los registros de propósito general deben ser igualmente utilizables por todas las instrucciones
      - Lo contrario dificultaría la ubicación de registros
    - ✓ Todas las operaciones deben admitir todos los tipos de datos y todos los modos de direccionamiento

Características de los juegos de instrucciones que facilitan una compilación eficiente (2 d 3)

- **Reducir el número de alternativas a considerar**
  - ✓ El diseño del compilador es más complicado si hay que analizar muchas formas diferentes de compilar una secuencia de instrucciones para escoger la mejor
  - ✓ El **principio de *uno o todo*** ayuda a conseguirlo
    - El juego de instrucciones debe proporcionar una única forma de hacer algo o bien todas. Ejemplo:
      - O bien sólo se dispone de una instrucción máquina de bifurcación sobre *igual* y una sobre *menor que* o bien se dispone de una instrucción para cada relación posible ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$)
      - Si no se hace así, el compilador tendrá que hacer un análisis más complejo para encontrar la mejor forma de implementar la bifurcación

Características de los juegos de instrucciones que facilitan una compilación eficiente (3 d 3)

- **Proporcionar *primitivas*, no *soluciones***
  - Paradójicamente, proporcionar instrucciones máquina que se corresponden con instrucciones de alto nivel (soluciones) no hace más eficiente la compilación. Esas *soluciones*
    - ✓ **O hacen menos de lo que se necesita** (por ejemplo, soportan determinados bucles pero no otros)
    - ✓ **O hacen más de lo que frecuentemente se necesita**, es decir, la mayoría de las veces no se necesita toda la capacidad de la instrucción usada (ej. una que soporte gran variedad de bucles) => El código máquina resulta más lento de lo necesario
  - La compilación se ha mostrado más efectiva cuando el juego de instrucciones proporciona operaciones sencillas (primitivas): permiten ajustar el código a lo que se necesita

2.3. Computadores de juego reducido de instrucciones: RISC

- RISC = *Reduced Instruction Set Computer*
- Idea original: Patterson y Ditzel en 1980
- Hasta entonces la evolución de los juegos de instrucciones había sido hacia una creciente complejidad
- Patterson y Ditzel denominaron **CISC** (Complex *Instruction Set Computer*) a esos computadores con elevado número de instrucciones máquina, instrucciones máquina potentes, gran variedad formatos, modos de direccionamiento, etc.
- Defendieron que un juego de instrucciones sencillo permitiría conseguir procesadores con **mejor relación coste rendimiento (los RISC)**

Características de los RISC

- Procesadores de tipo **carga-almacenamiento**
- **Pocas** instrucciones, instrucciones de longitud fija y formatos sencillos. No hay instrucciones para operaciones complejas
- Son **encauzados**
  - La sencillez del juego de instrucciones facilita la eficiencia de la UCP encauzada
    - ✓ Se consigue un **CPI próximo a 1**
- Reúnen las características que permiten diseñar **compiladores eficientes**

Ventajas de los RISC

- **Menor tiempo de diseño y verificación**
- La UCP ocupa poca área en el chip =>
  - Queda **más área para otros recursos** que mejoran el rendimiento (caché, banco de registros, soporte de pipeline...)
- **Rapidez**. Debido a
  - Que es encauzado
  - El banco de registros reduce el número de accesos a memoria
  - La compilación es eficiente

Inconvenientes de los RISC y modo de paliarlos

- Se necesitan **más instrucciones máquina** que en un CISC => más gasto de **tiempo** en leer instrucciones
  - Solución: usar una **caché de instrucciones**
- Si algunas de las instrucciones de un CISC representan **operaciones complejas frecuentes** en un programa (p. ej. de aritmética en **punto flotante**) el RISC será más lento al no disponer de esas instrucciones
  - Solución: **ampliar la UCP** para permitir operaciones de punto flotante:
    - ✓ Ya no estaríamos ante un RISC *puro*
    - ✓ El **CPI** será **mayor** pero el **TUCP** será **menor** en los programas con operaciones de punto flotante