eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
GASTEIZKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE VITORIA-GASTEIZ

1 Hardware, Interrupciones, DMA y programa en memoria

Introducción a los Sistemas Operativos, 2022-2023

Pablo González Nalda

Depto. de Lenguajes y Sistemas Informáticos
EU de Ingeniería de Vitoria-Gasteiz,
UPV/EHU

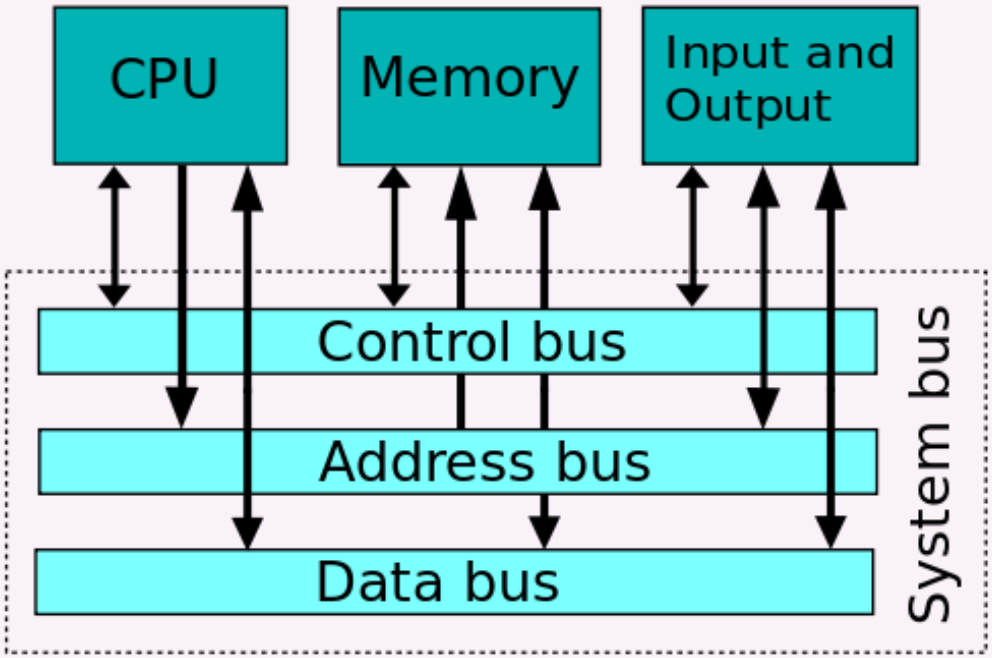
25 de enero de 2023

Contenidos de la presentación

Contenidos

Arquitectura Von Neumann

La *Arquitectura Von Neumann* es la habitual en los ordenadores. El procesador puede acceder a la memoria directamente con una instrucción en ensamblador/lenguaje máquina: `http://www.c-jump.com/CIS77/ASM/Addressing/lecture.html`



De W Nowicki - Trabajo propio, based on a diagram which seems to in turn be based on page 36 of The Essentials of Computer Organization and Architecture By Linda Null, Julia Lobur, https://books.google.com/books?id=f83XxoBC_8MC&pg=PA36, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=15258936

Multiprocesador y Multicomputador

UMA: todos los procesadores pueden acceder a toda la memoria (Uniform Memory Access), por ejemplo los núcleos de una CPU de escritorio o de móvil.

NUMA: cada procesador tiene acceso exclusivo a una parte de la memoria (por ejemplo, en las tarjetas gráficas).

Multicomputador: cada procesador tiene su propia memoria y sólo se puede comunicar con otro procesador mediante paso de mensajes por una red de alta velocidad (E/S por MPI).

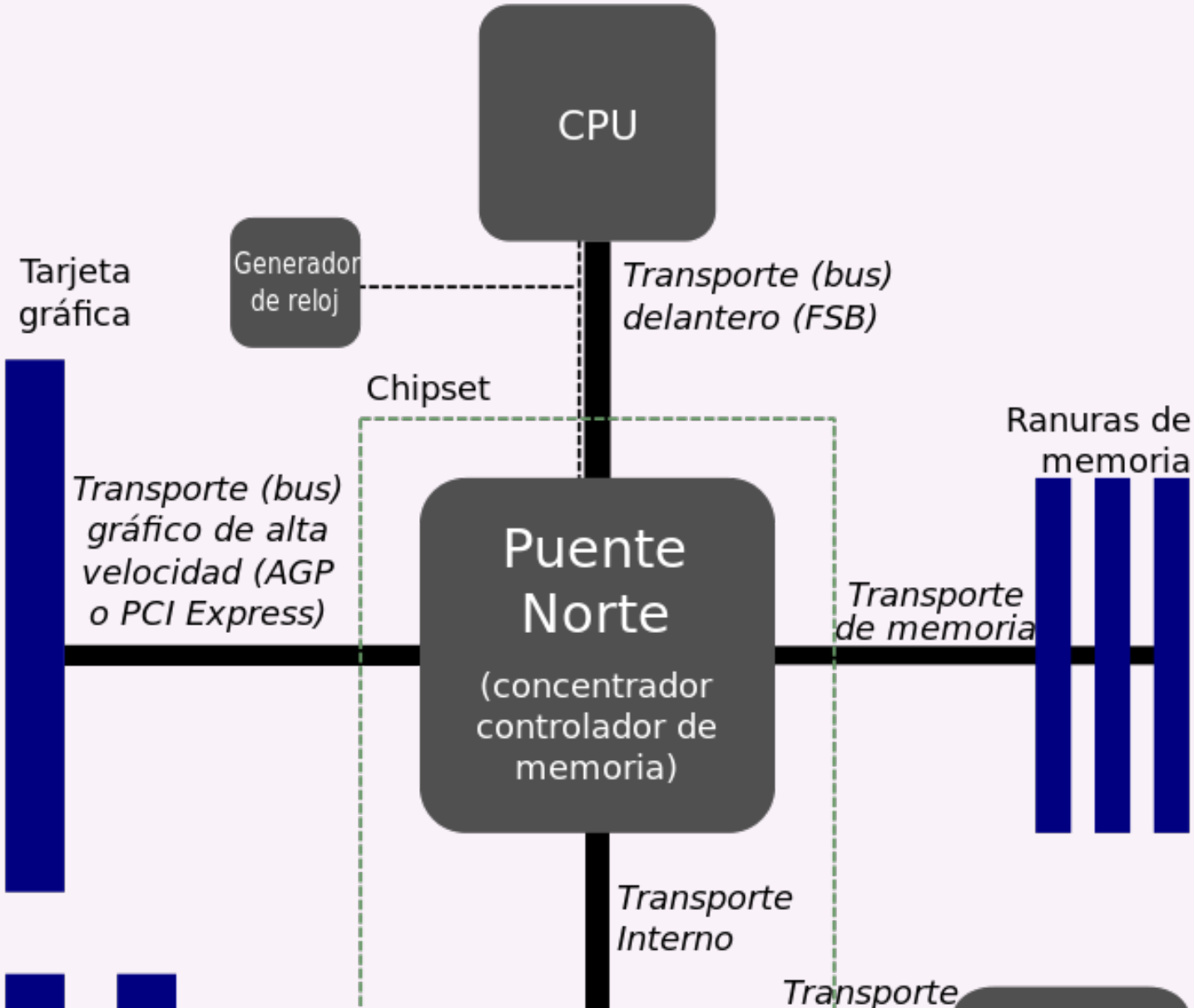
Arquitectura de la placa base

La arquitectura de la placa base se compone de dos *puentes*. El *Northbridge* gestiona el acceso por parte de la CPU a la RAM y al PCI-E (tarjeta gráfica) [1].

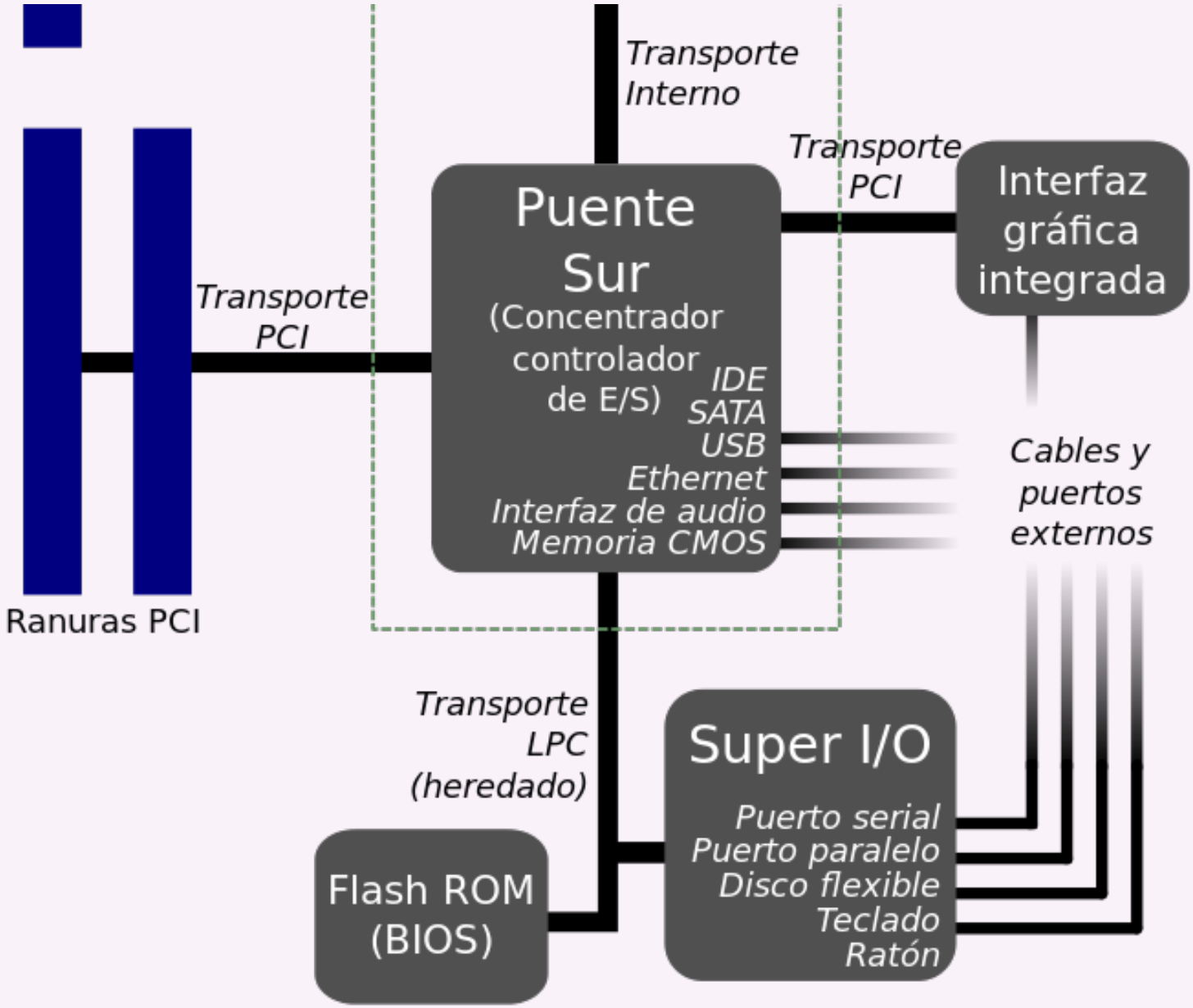
El *Southbridge* gestiona el acceso por parte de la CPU a los PCI y el restos de dispositivos de E/S [2].

[1] https://en.wikipedia.org/wiki/Northbridge_(computing)
[2] https://en.wikipedia.org/wiki/Southbridge_(computing)

Arquitectura de la placa base: *puente norte*



Arquitectura de la placa base *puente sur*



Contenidos

Hardware

Documento de Dr. Manmohan Sharma:

Computer Organization and Architecture

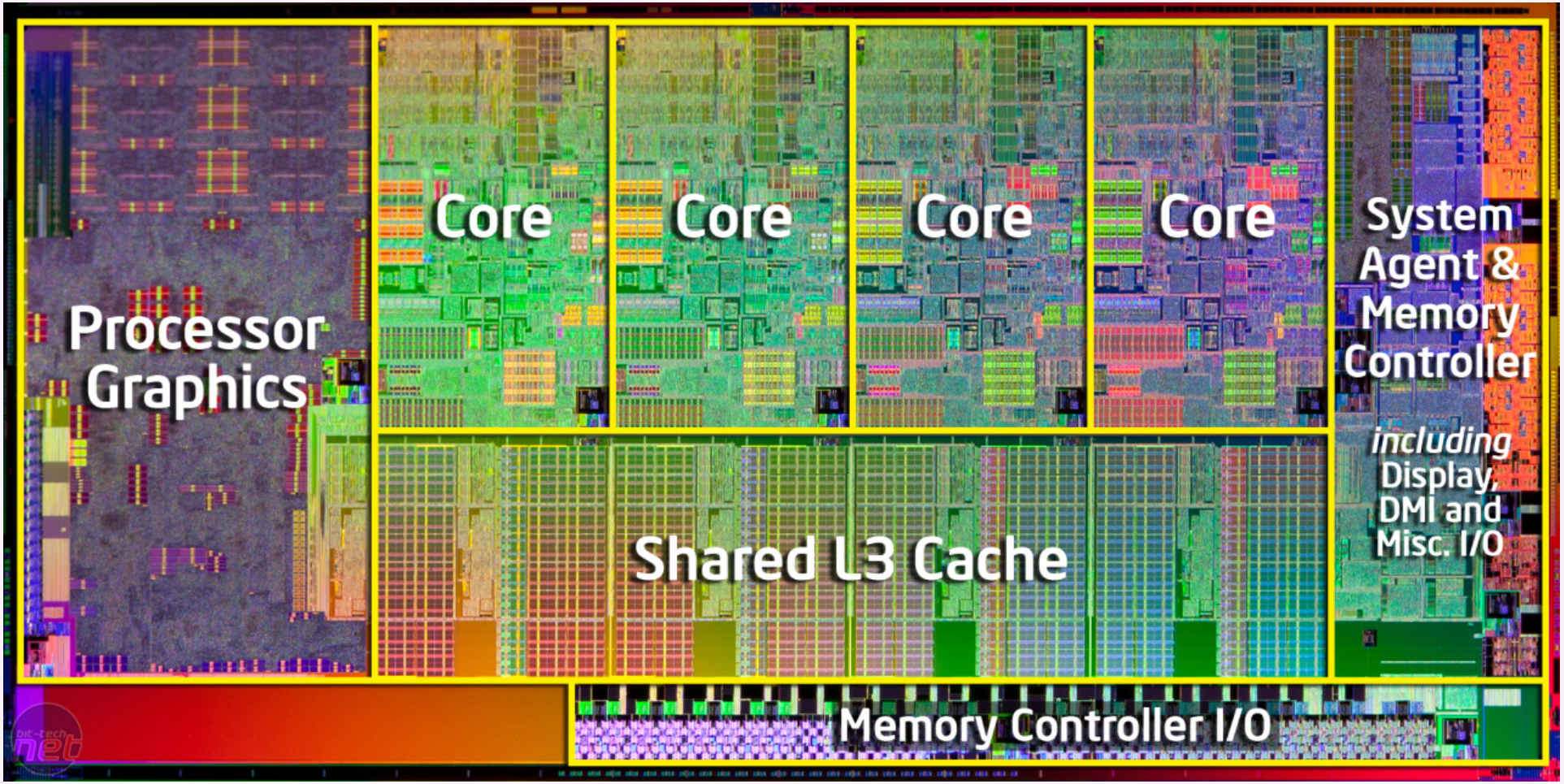
Puertas lógicas, *flip-flops* y registros

Ejemplo de puerta lógica a partir de la electrónica: punto 1.2 del documento

Flip-flop como unidad mínima de memoria: punto 2.2.2

Un registro es la unión de *flip-flops* para contener un byte o más: punto 2.3

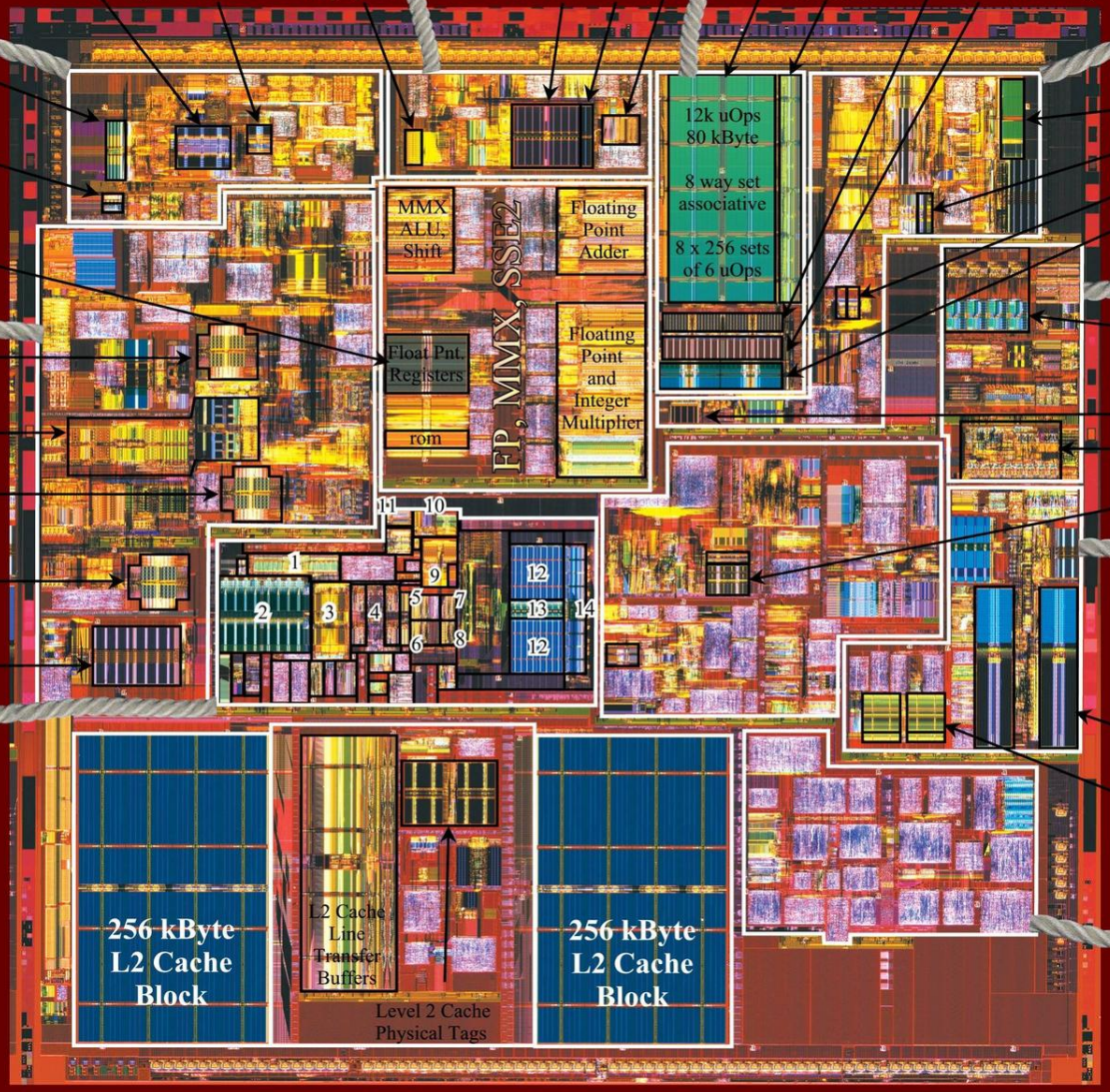
Un registro es una variable con una funcionalidad concreta y un lugar en el procesador.



Las cachés L1 y L2 están en cada núcleo o *core*.

Más datos y la siguiente imagen en
`https://superuser.com/questions/324284/what-is-meant-by-the-terms-cpu-core-die-and-p`

Intel Pentium 4 Northwood



Buffer Allocation & Register Rename

Instruction Queue (for less critical fields of the uOps)

General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)

Floating Point, MMX, SSE2 Renamed Register File 128 entries of 128 bit.

uOp Schedulers

FP Move Scheduler: (8x8 dependency matrix)

Parallel (Matrix) Scheduler for the two double pumped ALU's

General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)

Load / Store uOp Scheduler: (8x8 dependency matrix)

Load / Store Linear Address Collision History Table

Integer Execution Core

(1) uOp Dispatch unit & Replay Buffer Dispatches up to 6 uOps / cycle

(2) Integer Renamed Register File 128 entries of 32 bit + 6 status flags 12 read ports and six write ports

(3) Databus switch & Bypasses to and from the Integer Register File.

(4) Flags, Write Back

(5) Double Pumped ALU 0

(6) Double Pumped ALU 1

(7) Load Address Generator Unit

(8) Store Address Generator Unit

(9) Load Buffer (48 entries)

(10) Store Buffer (24 entries)

Execution Pipeline Start

Instruction Trace Cache

Trace Cache Access, next Address Predict

Trace Cache Branch Prediction Table (BTB), 512 entries.

Return Stacks (2x16 entries)

Trace Cache next IP's (2x)

Miscellaneous Tag Data

Instruction Decoder

Up to 4 decoded uOps/cycle out. (from max. one x86 instr/cycle) Instructions with more than four are handled by Micro Sequencer

Trace Cache LRU bits

Raw Instruction Bytes in

Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

Instruction Fetch from L2 cache and Branch Prediction

Front End Branch Prediction Tables (BTB), shared, 4096 entries in total

Instruction TLB's 2x64 entry, fully associative for 4k and 4M pages. In: Virtual address [31:12] Out: Physical address [35:12] + 2 page level bits

Front Side Bus Interface, 400..800 MHz

(11) ROB Reorder Buffer 3x42 entries

(12) 8 kByte Level 1 Data cache four way set associative. 1R/1W

(13) Summed Address Index decode and Way Predict

(14) Cache Line Read / Write Transferbuffers and 256 bit wide bus to and from L2 cache

April 19, 2003 www.chip-architect.com

Contenidos

Códigos de instrucción y operandos

Punto 4.1: el procesador (CPU, un núclo o *core* de la CPU) lee un byte o más de la RAM (en la posición almacenada por el registro Contador de Programa o *PC*) y lo guarda en el registro IR (registro de instrucción).

La Unidad de Control (CU) de la CPU interpreta el contenido del IR como una instrucción de Lenguaje Máquina, partiendo en el código de operación y operandos (por ejemplo, código de suma y sumandos). La CU ejecuta el microcódigo para controlar registros y sistemas de la CPU.

Los operandos se encuentran en registros de datos del procesador o en la RAM.

Registros del procesador

Punto 4.2: Registros.

1. registros de datos
2. registros de direcciones
3. palabra de estado del procesador
4. contador de programa PC
5. puntero de la pila SP

Fases de la ejecución de una instrucción

Punto 5.2: Ejecución de una instrucción:

1. Obtener la instrucción (uno o varios bytes)
2. Decodificar la instrucción (microcódigo)
3. Obtener operandos (calcular indirecciones)
4. Realizar la operación (ALU o unidad aritmético-lógica, FPU o unidad de coma flotante)
5. Guardar resultado en registro, memoria o E/S.

Pipelining

Figura 14.1:

En vez de la ejecución secuencial de las fases de una instrucción (*fetch*,*decoding*, *execution*), podemos aprovechar a cargar las unidades funcionales del procesador para acelerar la ejecución de las instrucciones, como una cadena de montaje de una fábrica.

¿Qué hacemos si hay un salto condicional, cómo seguimos si todavía no sabemos si hay que saltar, qué instrucción empezamos a ejecutar?

¿Qué ocurre si hay una interrupción?

Ejemplo de un programa de CPU

Ejemplo de programa en ensamblador y su transcripción a Lenguaje Máquina (en hexadecimal) ejecutándose en una CPU:

`Animated Working of 8085 Microprocessor with addition program`

Programa en C

De un programa en C a Lenguaje Máquina.

```
gcc -S -masm=intel -fverbose-asm hola.c
```

Verboso: abundante, copioso y prolijo en palabras.

```
1 int main() {
    int i,f=1;
    for (i=1;i<5;i++)
4     f=f*i;
  }
```

Fichero 1: `as.c`

Programa en Ensamblador


```
1  .LFB0:
       push    rbp #
       .cfi_def_cfa_offset 16
4      .cfi_offset 6, -16
       mov rbp, rsp    #,
       .cfi_def_cfa_register 6
7  # as.c:2:      int i,f=1;
       mov DWORD PTR -4[rbp], 1    # f,
   # as.c:3:      for (i=1; i<5;i++)
10     mov DWORD PTR -8[rbp], 1    # i,
       jmp .L2 #
   .L3: # as.c:4:          f=f*i;
13     mov eax, DWORD PTR -4[rbp]  # tmp90, f
       imul    eax, DWORD PTR -8[rbp]  # tmp89, i
       mov DWORD PTR -4[rbp], eax  # f, tmp89
16 # as.c:3:      for (i=1; i<5;i++)
       add DWORD PTR -8[rbp], 1    # i,
   .L2: # as.c:3:      for (i=1; i<5;i++)
19     cmp DWORD PTR -8[rbp], 4    # i,
       jle .L3 #,
       mov eax, 0  # _5,
22 # as.c:5: }
       pop rbp #
       ret
```

Ejemplo de programa

Ejemplo de programa en Ensamblador con Lenguaje Máquina:

```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2


 C000                    ORG    ROM+$0000 BEGIN MONITOR
 C000 8E 00 70  START    LDS    #STACK

                *************************************
                * FUNCTION: INITA - Initialize ACIA
                * INPUT: none
                * OUTPUT: none
                * CALLS: none
                * DESTROYS: acc A

 0013           RESETA  EQU    %00010011
 0011           CTLREG  EQU    %00010001

 C003 86 13     INITA   LDA A  #RESETA   RESET ACIA
 C005 B7 80 04          STA A  ACIA
 C008 86 11             LDA A  #CTLREG   SET 8 BITS AND 2 STOP
 C00A B7 80 04          STA A  ACIA

 C00D 7E C0 F1          JMP    SIGNON    GO TO START OF MONITOR

                *************************************
                * FUNCTION: INCH - Input character
                * INPUT: none
                * OUTPUT: char in acc A
                * DESTROYS: acc A
                * CALLS: none
                * DESCRIPTION: Gets 1 character from terminal

 C010 B6 80 04  INCH    LDA A  ACIA      GET STATUS
 C013 47                ASR A            SHIFT RDRF FLAG INTO CARRY
 C014 24 FA             BCC    INCH      RECIEVE NOT READY
 C016 B6 80 05          LDA A  ACIA+1    GET CHAR
 C019 84 7F             AND A  #$7F      MASK PARITY
```

La *Pila* o *stack* y el ensamblador

Lo habitual es que cada programa tenga su *pila* (punto 7.3), una estructura LIFO (Last In First Out) para almacenar datos en la memoria asignada al programa (punto 7.3.2).

Las variables locales y la dirección de retorno de un subprograma se colocan en la pila, apuntada por el SP, puntero de la pila o *stack pointer*, que es un registro del procesador.

`La pila, Wikipedia`

`Uso de la pila, StackOverflow` (reserva de espacio para variables locales)

`Chiste sobre StackOverflow`

Ensamblador: `Assembly language, Wikipedia`

`Comparing C to machine language, Youtube`

Control del programa

Tabla 8.1 y punto 8.3: el programa puede realizar saltos (cambiar el contador de programa PC) incondicionales (cargando una dirección en el PC o sumándole un desplazamiento positivo o negativo).

Los saltos también pueden ser condicionales, en función del contenido en un registro o en la palabra de estado (bits que guardan el estado del procesador y de la última instrucción), ver la tabla 8.5.

Una subrutina es un subprograma al que entramos con CALL (se guarda en la pila el PC) y volvemos con RET (que recupera el PC de la pila).

Contenidos

Entrada/salida

Punto 11.3: entrada/salida.

1. síncrona: si el sistema del dispositivo de E/S tiene un ritmo fijo de funcionamiento.
2. asíncrona: se debe comprobar el dispositivo de entrada/salida para seguir con la operación.
3. por interrupción: el dispositivo tiene un mecanismo de aviso para requerir que el procesador continúe con la E/S.

DMA, Acceso Directo a Memoria

Punto 11.3: El DMA (Acceso Directo a Memoria) es un sistema que realiza operaciones de movimiento de datos entre RAM y dispositivos *mapeados en memoria*. El DMA es programado por la CPU y la descarga de trabajo.

Las CPUs y el DMA deben compartir el bus del sistema, y por lo tanto el DMA debe solicitar el uso del bus (BRQ, BusReQuest) y la CPU concederlo (BGN, BusGraNt).

La concesión y transferencia se realiza por *ráfagas* (*bursts*).

Un procesador de E/S (punto 11.7) es un DMA programable con un juego de instrucciones propio para hacer operaciones complejas.

E/S por registros

La E/S (I/O) se puede realizar mediante el uso de registros: datos, direcciones, estado y control.

Pueden necesitar instrucciones específicas de ensamblador o estar *mapeados* en memoria (ver `https://en.wikipedia.org/wiki/Memory-mapped_I/O`).

Datos lectura o escritura del dato en movimiento desde/hacia el dispositivo.

Direcciones Ubicación del dato en el dispositivo.

Estado Sólo lectura, con información del estado del dispositivo (disponible/ocupado, error, etc.).

Control Sólo escritura, para indicar que el dato y la dirección están escritos y se debe realizar la operación de E/S.

E/S por Espera Activa (*Polling*)

La E/S por Espera Activa es una forma asíncrona, y consiste en un ciclo que ejecuta el procesador, en el que consulta un registro o posición de memoria hasta que el dispositivo cambia el valor del registro y se puede hacer la E/S.

```
while (lee_estado(disp1) == NO_DISPONIBLE) {
    ; // ninguna operaci'on
}
// se supone que el dispositivo se encuentra disponible
coloca_dato(disp1, dato_por_escribir);
// bit de escritura en el registro de control
realiza_escritura(disp1);
```

https://en.wikipedia.org/wiki/Polling_(computer_science)

E/S por Interrupción

Punto 11.5: la E/S por Interrupción se lleva a cabo mediante una señal hardware que ejecuta un código de forma **asíncrona**.

1. Un dispositivo necesita atención, por ejemplo el teclado, ratón, disco, tarjeta de red…y produce una petición de interrupción (IRQ).
2. El dispositivo activa un bit hardware que llega hasta el controlador de interrupciones (CI) (dentro de la CPU).
3. El CI asocia esa activación con un número *n* de interrupción, que servirá para indexar el Vector de Interrupciones (VI). El VI es la tabla de rutinas de código (funciones en C o en ensamblador) que sirve para gestionar las IRQ de todos los controladores hardware.
4. El CI interrumpe la ejecución del programa en curso que ocupa la CPU (guardando los registros de la CPU y el contador de programa o PC) y ejecuta el código de `VI[n]`.
5. La rutina atiende al dispositivo, reparte la información y al terminar restaura el programa que se estaba ejecutando.

`https://en.wikipedia.org/wiki/Interrupt`

Interrupciones originadas en el software

Los errores de software provocan una interrupción software de forma síncrona (*trap*[3]), como salirse de una tabla. El propio programa produce la IRQ con una instrucción específica de lenguaje máquina.

También se denominan *trap* a los errores generados por el hardware, es decir, el hw produce la IRQ. Ejemplos: división por cero, instrucción de código máquina errónea, acceso a memoria incorrecto.

Hay una gran variedad en la nomenclatura y clasificación:

`https://stackoverflow.com/questions/3149175/what-is-the-difference-between-trap-and-interrupt`

[3] *Trap*: `https://en.wikipedia.org/wiki/Trap_(computing)`

Interrupción del reloj

IRQ del reloj: un dispositivo programable[4] [5] realiza una interrupción hardware periódica (cada centésima de segundo, hasta 1 μs) con el objetivo de lanzar operaciones de mantenimiento y gestión del sistema.

Esa IRQ provoca la ejecución de la *Rutina de atención al reloj*, indexada en el Vector de Interrupciones. Cada vez que se produce esa IRQ lo llamamos un *tick de reloj* y el Sistema Operativo realiza operaciones periódicas.

El tiempo en Unix[6] es un entero con signo de 32 bits que cuenta el número de segundos desde 1-1-1970. El último segundo con este sistema, el $2^{31} - 1$ será en 2038.

`https://wiki.osdev.org/Time_And_Date` y `https://lwn.net/Kernel/LDD3/` (capítulo 7)

[4] `https://en.wikipedia.org/wiki/Programmable_interval_timer`
[5] `https://wiki.osdev.org/PIT`
[6] `https://en.wikipedia.org/wiki/Unix_time`

Elementos en memoria: código común y reutilizable

Tendremos en memoria el código dependiente del hardware común a todos los programas, que por ello se puede mantener en memoria y ser reutilizable: es el origen del SO.

- El Vector de Interrupciones (VI)
- Rutinas de atención a la interrupción de reloj
- Rutinas de error hardware y software
- Rutinas Dispersoras (integran diferentes operaciones de una misma IRQ)
- Rutinas de atención a interrupciones de los dispositivos
- Rutinas de E/S

Elementos en memoria: el programa

Y del programa [7] [8] [9]:

- Variables globales (fuera de funciones) y estáticas: *data* (si no están inicializadas, en *bss segment*)
- Constantes: Código (*text segment* en x86) y *data*
- Memoria reservada mediante ubicación dinámica: montículo o *heap*
- Parámetros y variables locales de una función, incluyendo punteros (pila o *stack*)
- Código (próxima instrucción de Lenguaje Máquina apuntada por el registro PC (Contador de Programa))
- Rutinas de librería

[7] https://www.geeksforgeeks.org/memory-layout-of-c-program/

[8] https://www.codeproject.com/Articles/76153/Six-important-NET-concepts-Stack-heap-value-types

[9] https://stackoverflow.com/questions/14588767/where-in-memory-are-my-variables-stored-in-c

¿Más preguntas?

¿Más preguntas?

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
GASTEIZKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE VITORIA-GASTEIZ

1 Hardware, Interrupciones, DMA y programa en memoria

Introducción a los Sistemas Operativos, 2022-2023

Pablo González Nalda

Depto. de Lenguajes y Sistemas Informáticos
EU de Ingeniería de Vitoria-Gasteiz,
UPV/EHU

25 de enero de 2023