"I think there is a world market for maybe five computers."

Thomas Watson, chairman of IBM, 1943

EECS 314 Computer Architecture

Instructors: Professor Chris Papachristou & Francis G. Wolff
wolff@eecs.cwru.edu
Case Western Reserve University

This presentation uses powerpoint animation: please viewshow

Computer Architecture Trends: Post PC era

- 100 million processors were sold for desktop computers

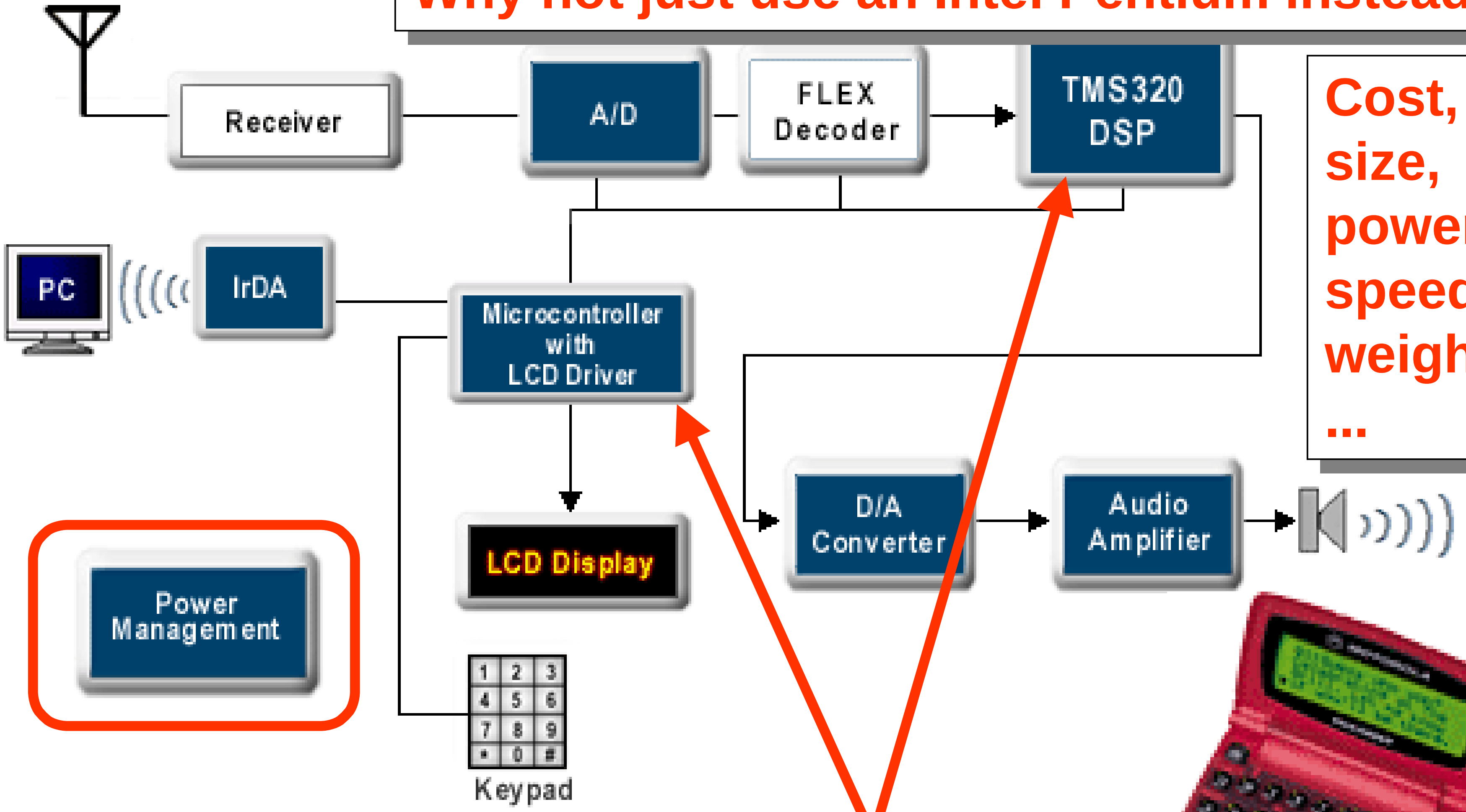
- 3 BILLION processors were sold for embedded systems

It's expected that the average car will be Internet ready and have over $2000 worth of embedded computers

Digital Pager Architecture

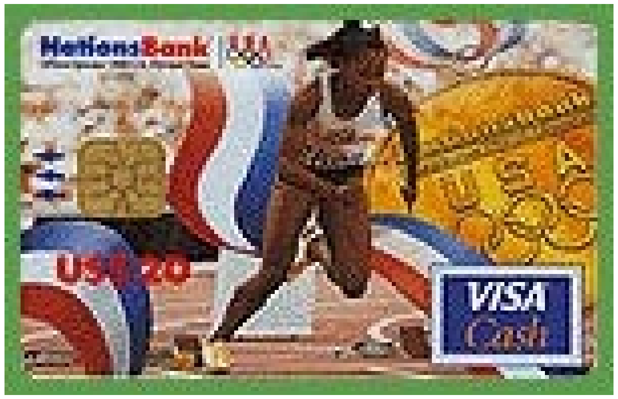
Why not just use an Intel Pentium instead?

Cost, size, power, speed, weight, ...



Two completely differently optimized Instruction Set Architectures

Smart Cards: Hardware/Software Co-Design



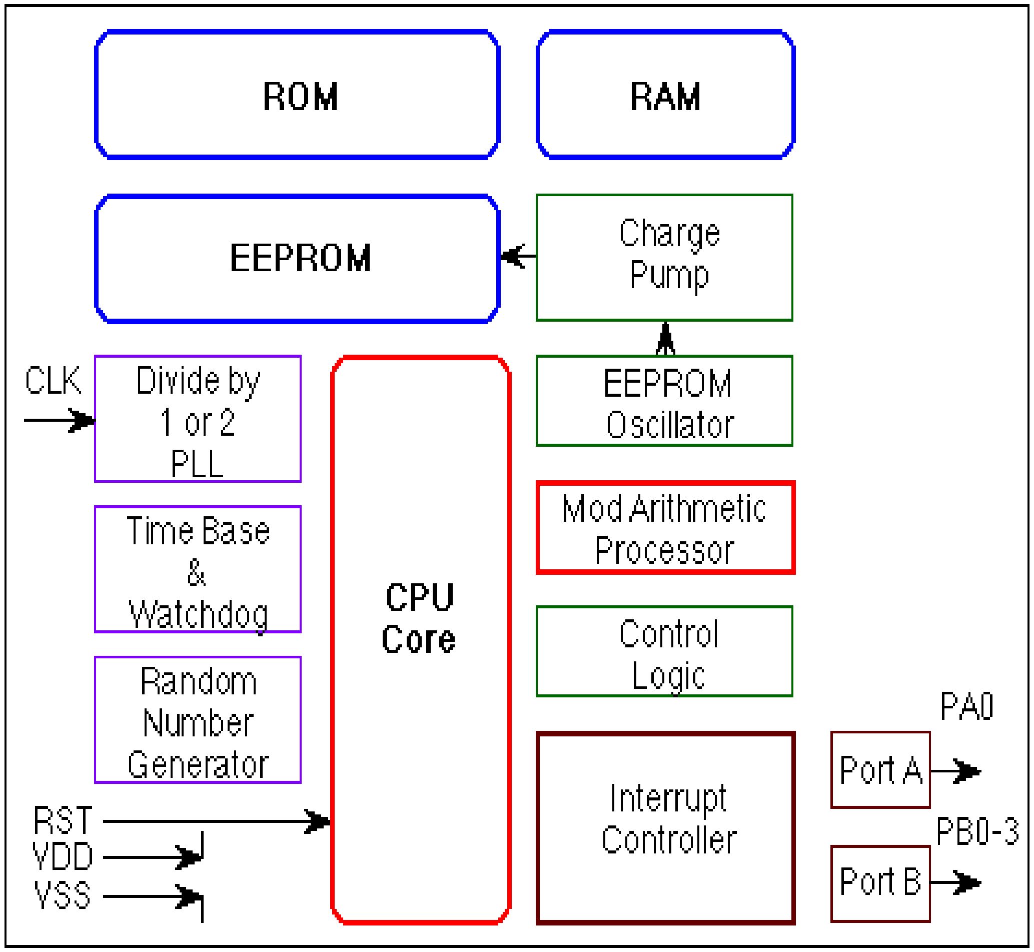
• There are currently 2.8 billion smart cards in use:

• 575 million phone, 36 million financial, 30 million ID cards, …

• Smart cards differ from credit cards in using onboard memory chips and microprocessors or micro-controllers instead of magnetic strips.

Each chip can hold 100k times the information contained on a standard magnetic-stripe card.

Smart Cards: Computer Architecture



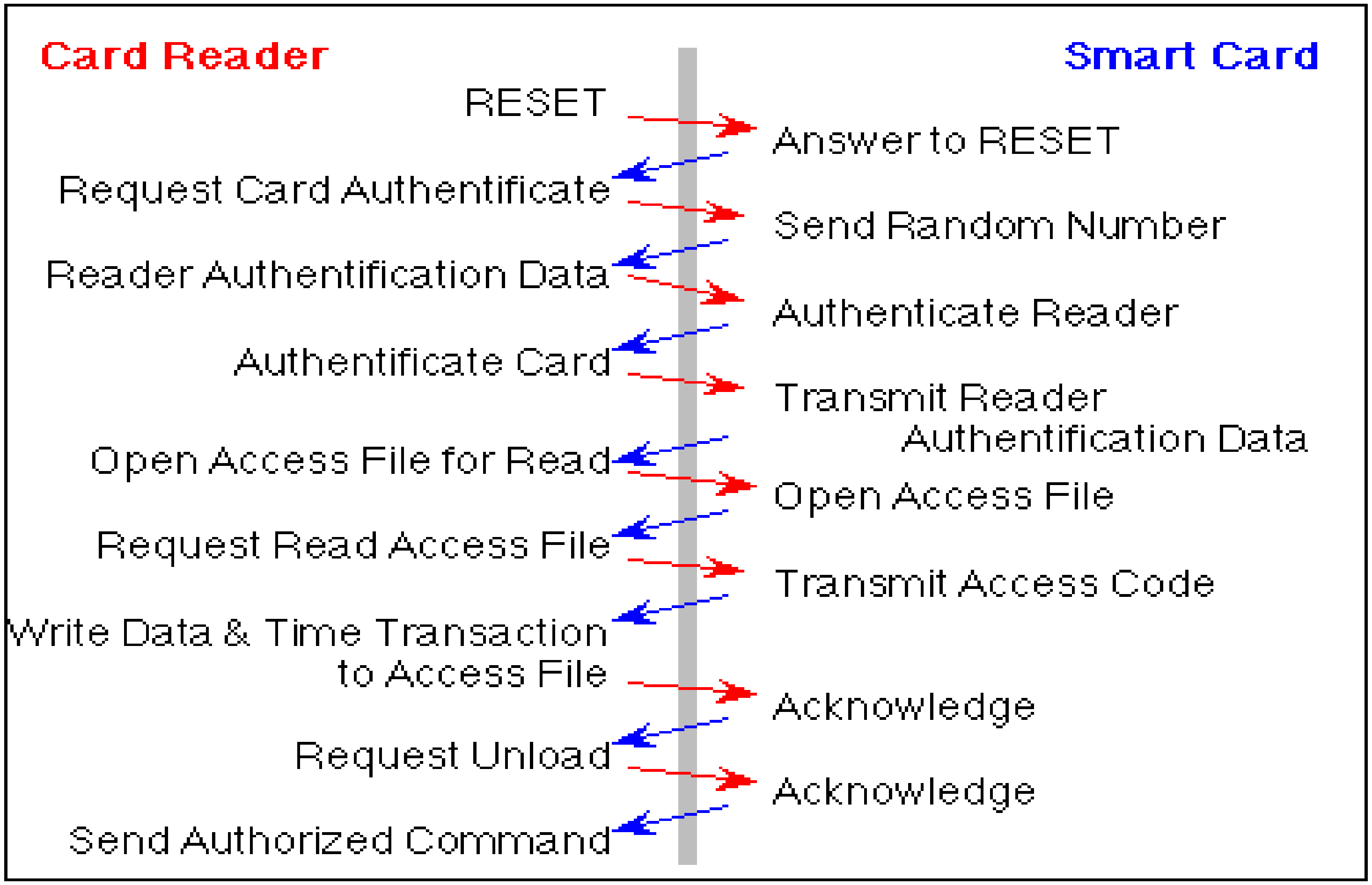
Smart cards have embedded within them a processor and often a crypto-graphically enhanced co-processor.

Features:

- Accelerated Software Cryptography
- Java Card
- Windows for Smart Cards
- Code Compression
- Secure Memory Spaces

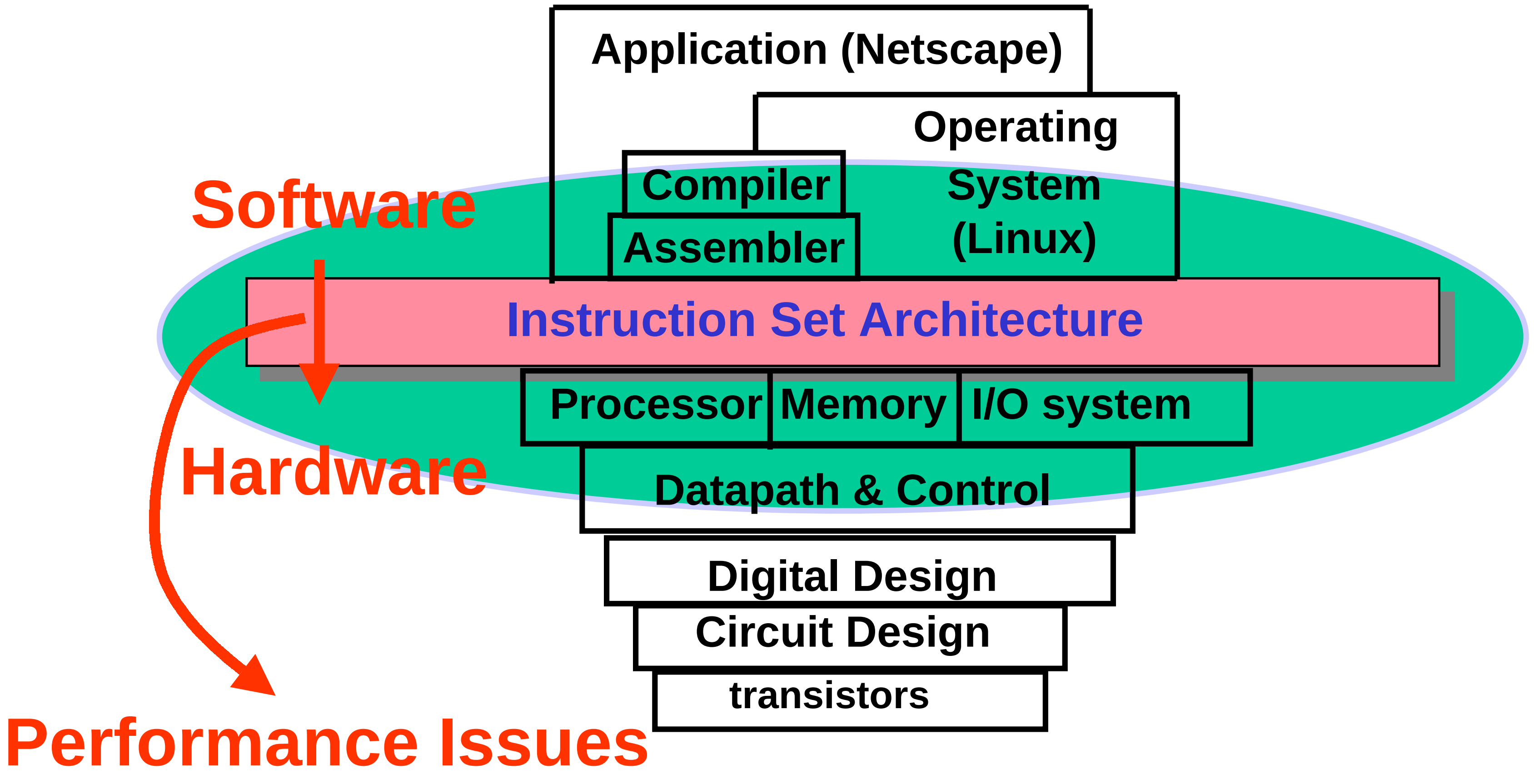
Smart Cards: Hardware/Software Co-Design

An example of the software handshaking protocol is shown below



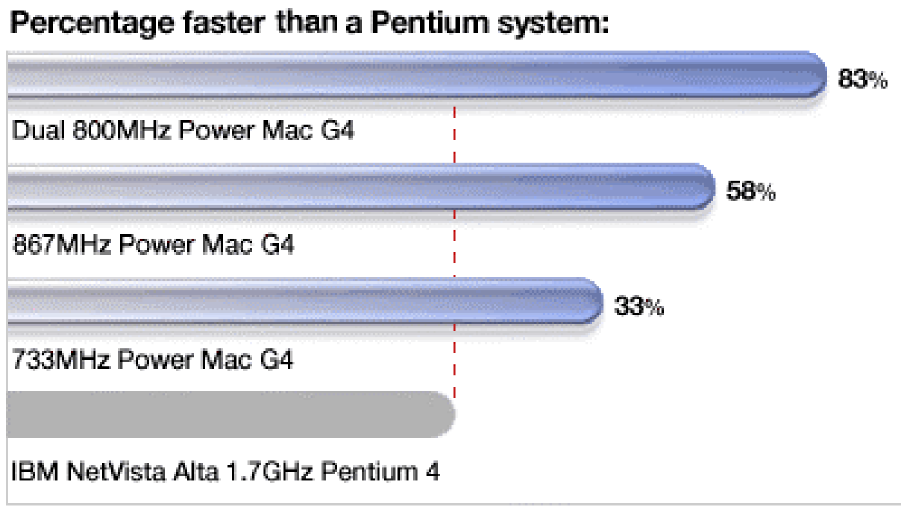
Design Abstractions

- Coordination of many *levels of abstraction*



Speed
Power
Size

“The Megahertz Myth.”


Percentage faster than a Pentium system:
83%
Dual 800MHz Power Mac G4
58%
867MHz Power Mac G4
33%
733MHz Power Mac G4
IBM NetVista Alta 1.7GHz Pentium 4

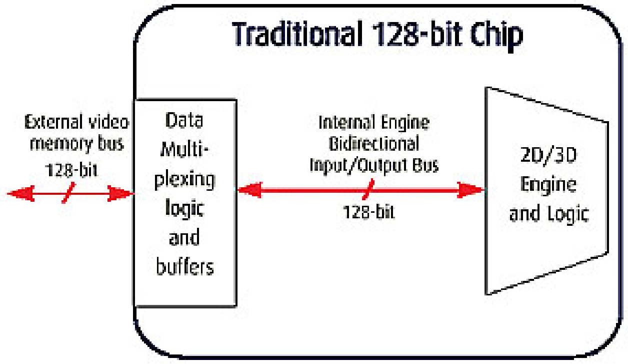
Why the clock speed of a computer isn’t an accurate way to compare system performance.

Overall system design and processor-architecture differences affect real-world application performance, otherwise you might be fooled by what Jon terms “The Megahertz Myth.”

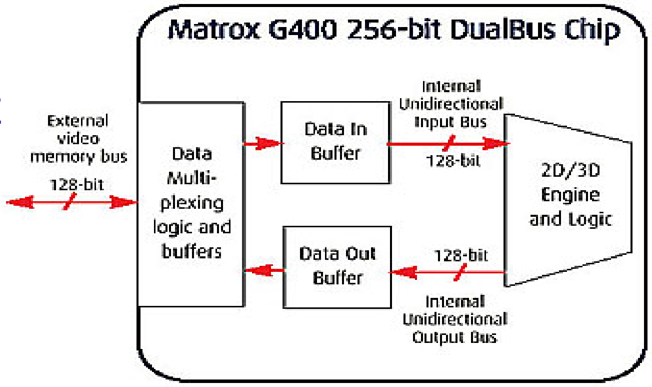
--Jon Rubinstein, Apple Senior VP of Hardware

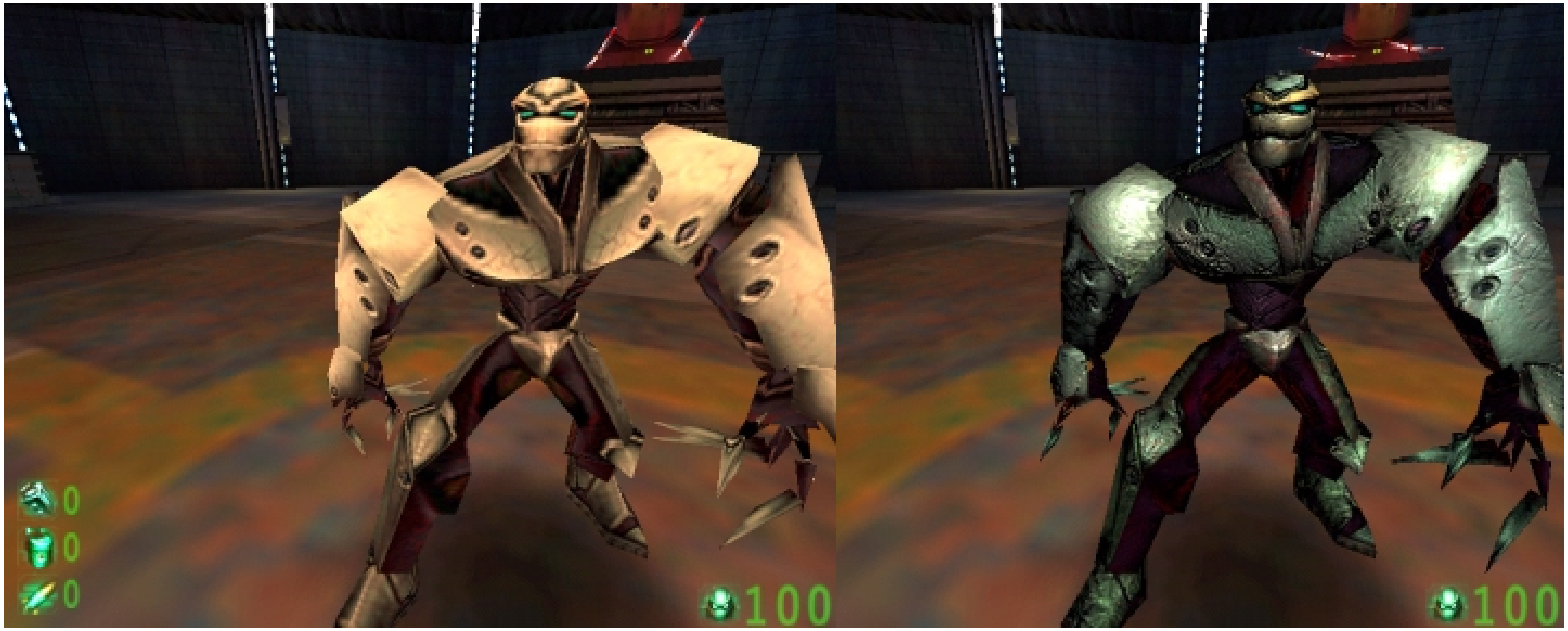
High Performance: Video Graphic Architectures

DirectX is a set of components, developed to provide Windows-based programs with high-performance, real-time access to available hardware on current computer systems.


Traditional 128-bit Chip
External video memory bus
128-bit
Data Multi-plexing logic and buffers
Internal Engine Bidirectional Input/Output Bus
128-bit
2D/3D Engine and Logic

DirectX enables any 3D hardware with Environment Mapped Bump Mapping (EMBM) improves the visual realism of 3D rendered scenes.


Matrox G400 256-bit DualBus Chip
External video memory bus
128-bit
Data Multi-plexing logic and buffers
Data In Buffer
Data Out Buffer
Internal Unidirectional Input Bus
128-bit
128-bit
Internal Unidirectional Output Bus
2D/3D Engine and Logic



Instruction Set Architecture

A very important abstraction: Instruction Set Architecture

- **interface between hardware and low-level software**
- **standardizes instructions, machine language bit patterns, ...**
- **advantage:** ***different implementations of the same architectu***
- **disadvantage:** ***sometimes prevents using new innovations***

Modern instruction set architectures:

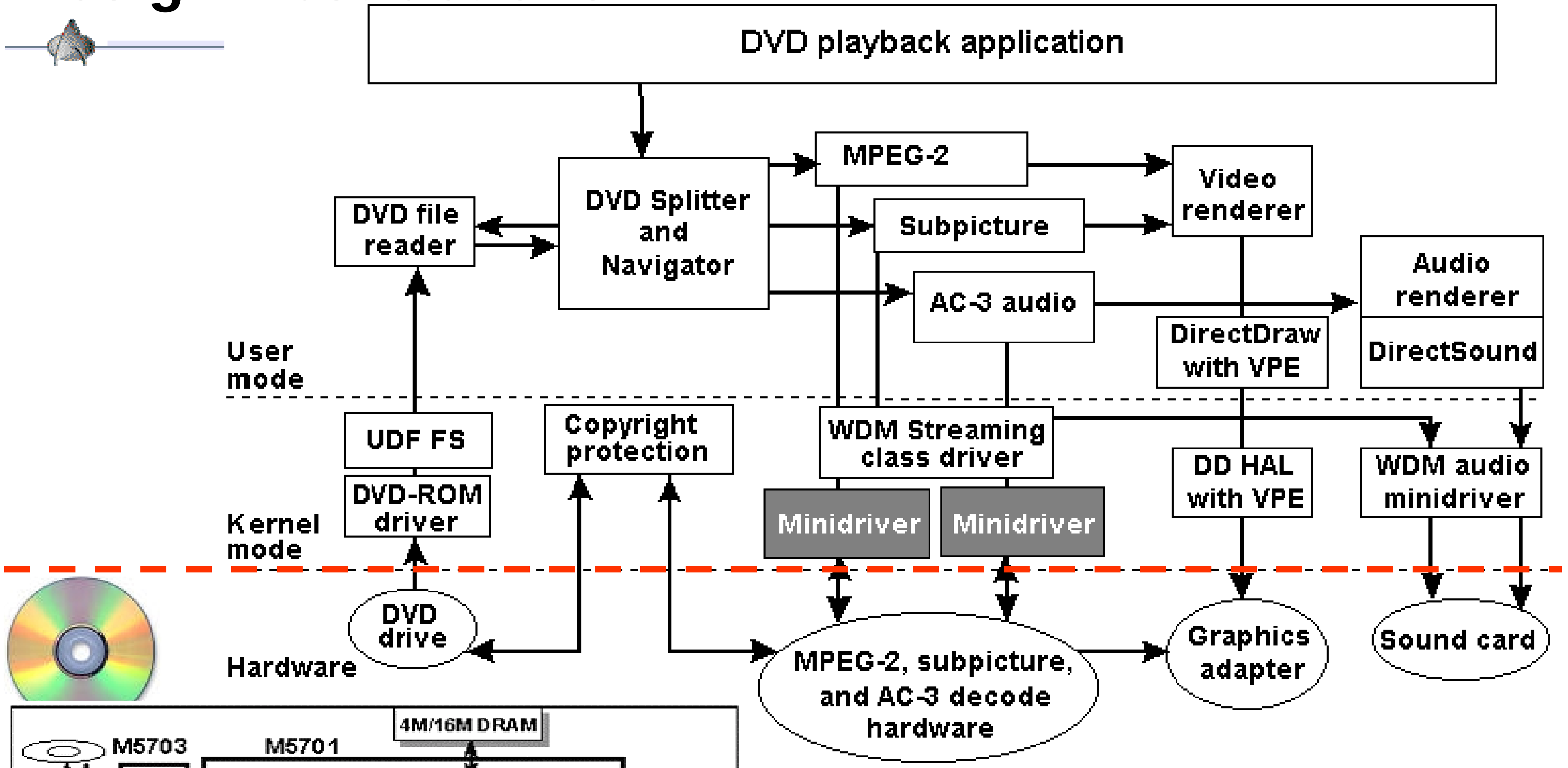
PowerPC, DEC Alpha, MIPS, SPARC, HP, 80x86/Pentium/K6*

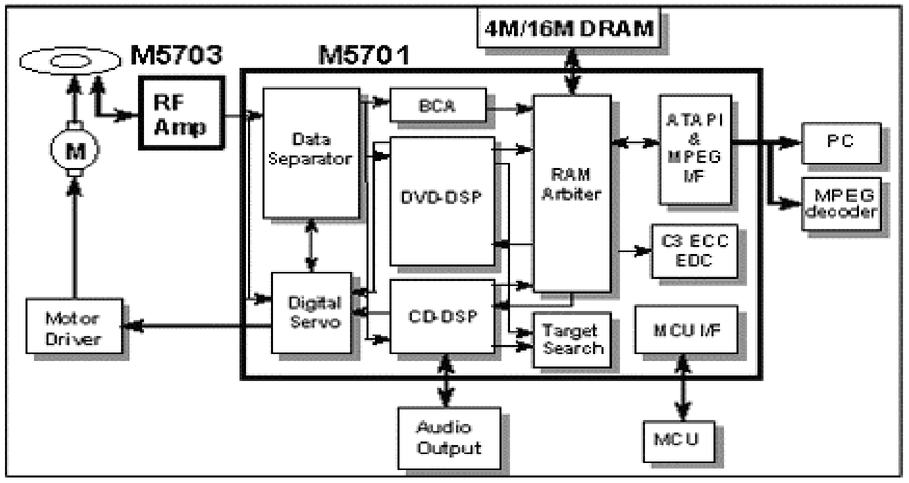
True or False: Binary compatibility is important?

Yes (Microsoft/Intel alliance) No - (Unix, Linux, C++, Java)

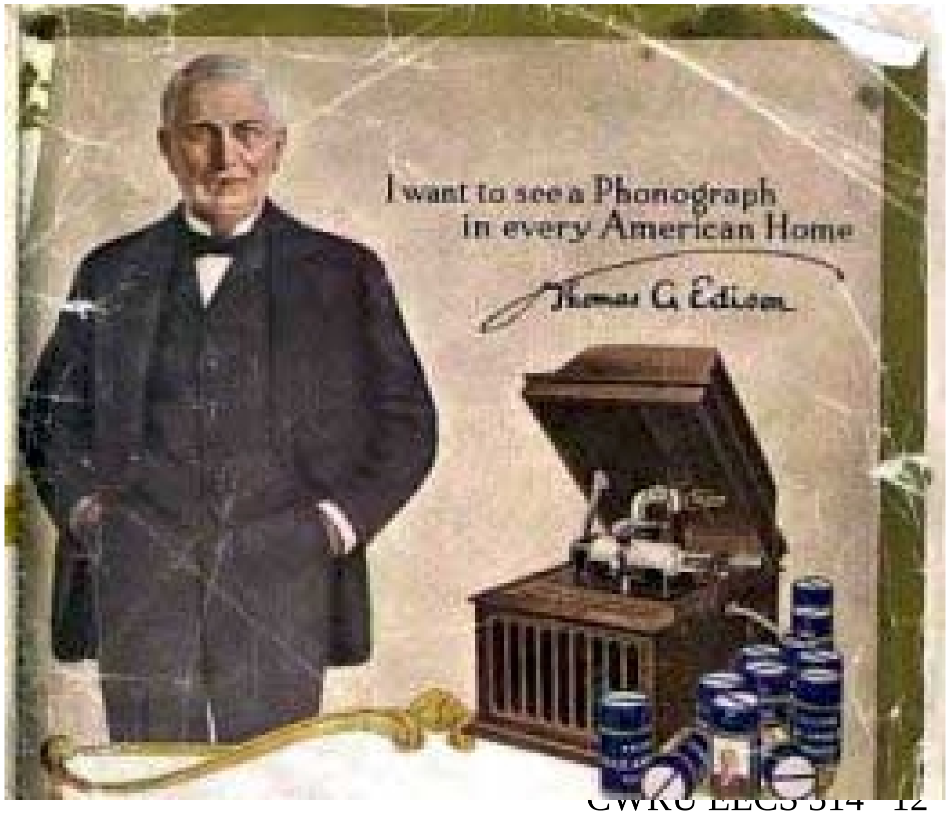
Yes - Sales, Marketing No - Speed, Engineers, Programmer

Design Abstractions: DVD

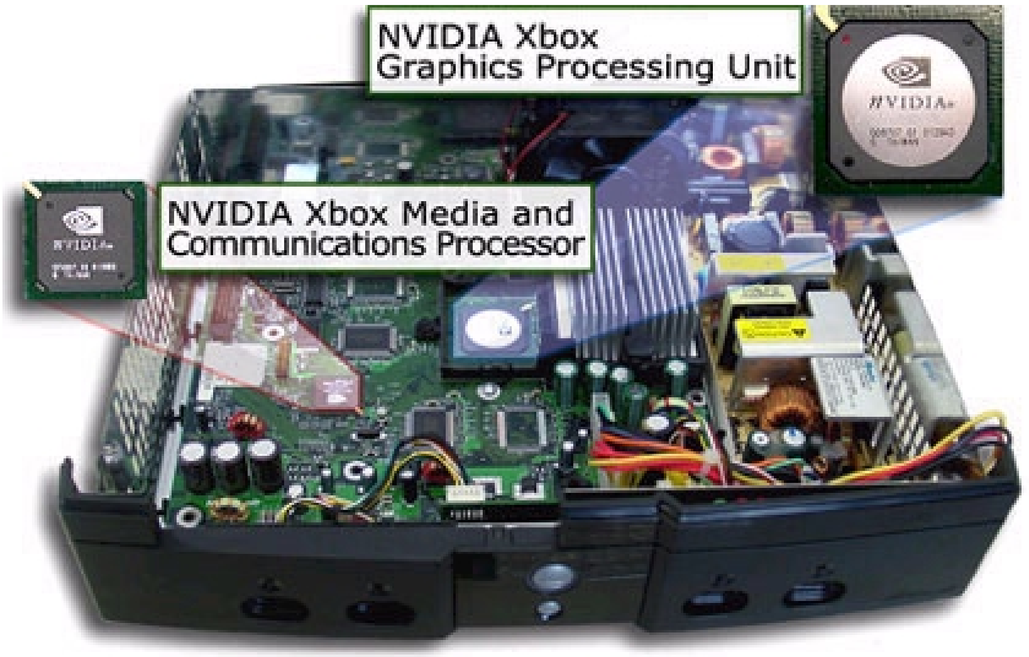


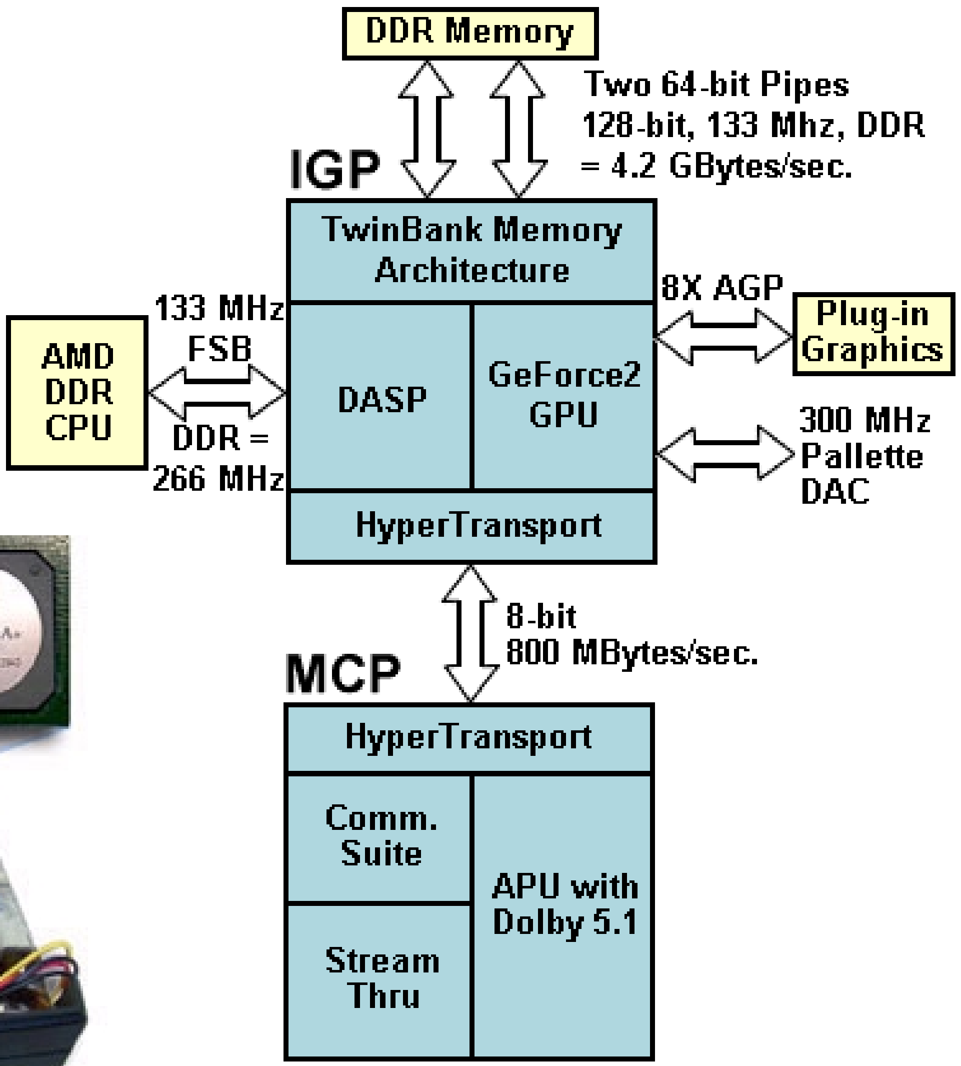


An early DVD version



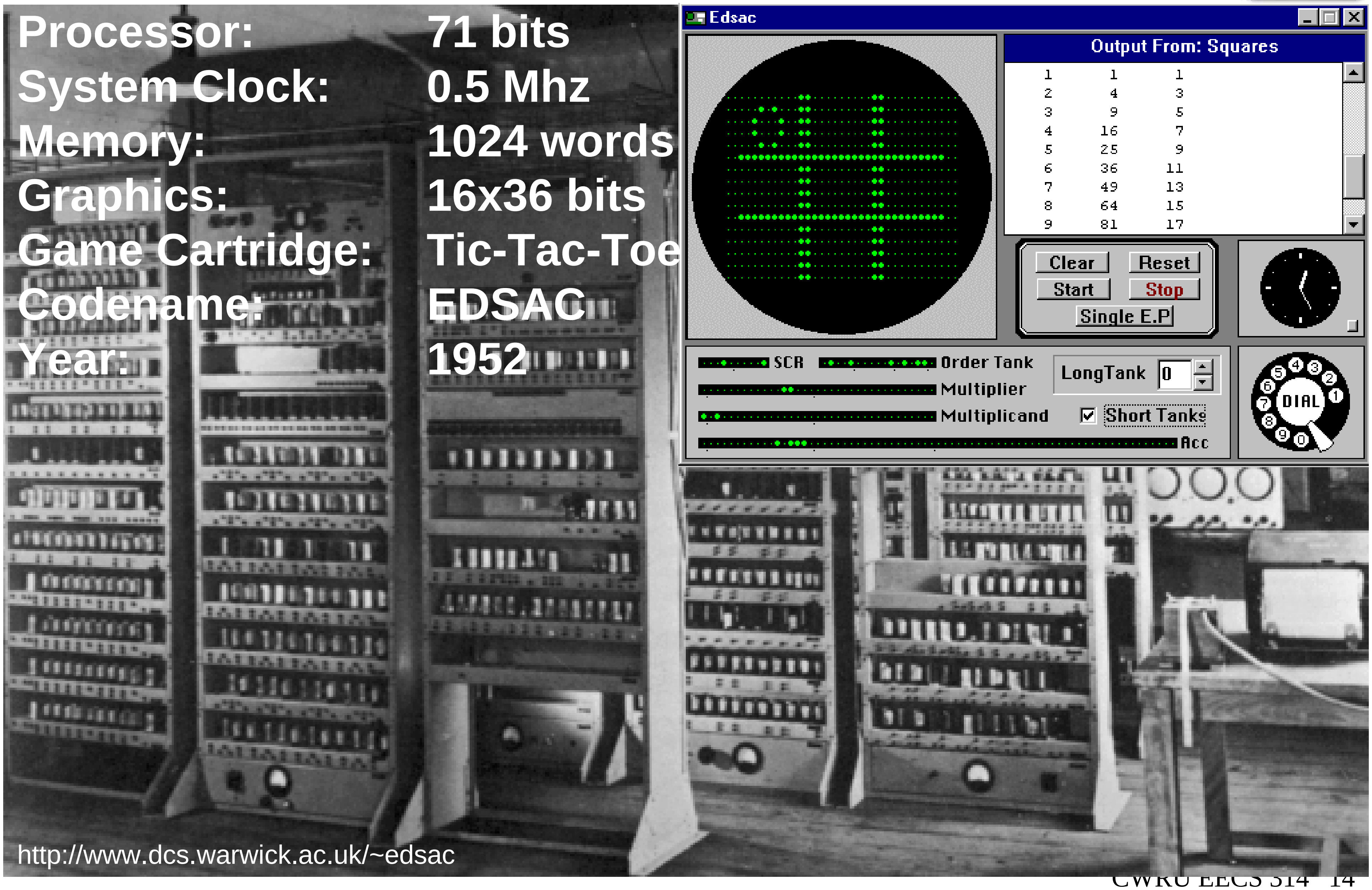
High Performance: Video Graphic Architectures


NVIDIA Xbox Graphics Processing Unit
NVIDIA Xbox Media and Communications Processor

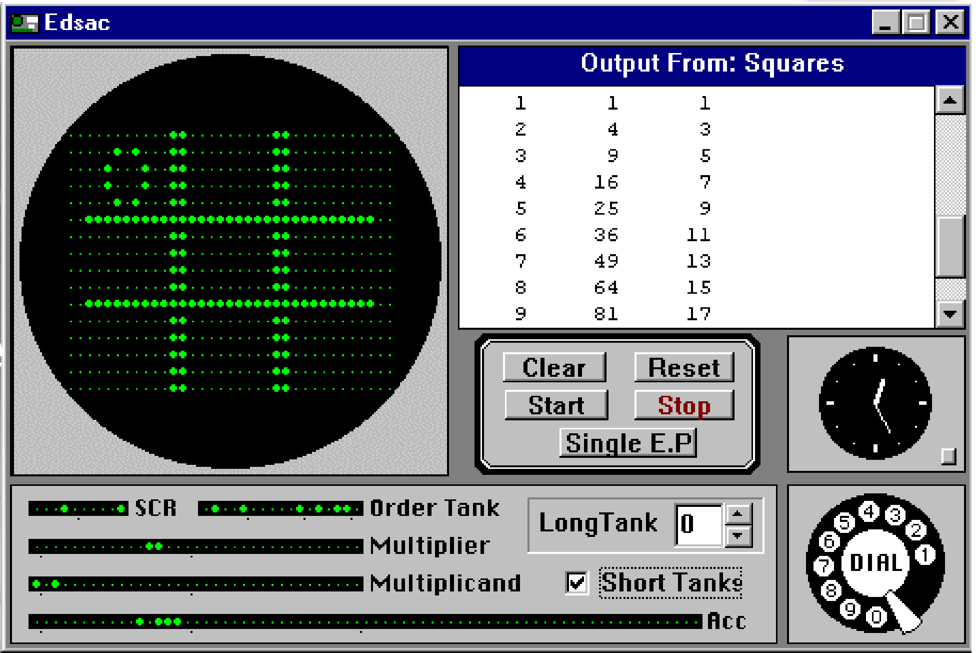


http://www.nvidia.com

An early XBOX version



Processor:	71 bits
System Clock:	0.5 Mhz
Memory:	1024 words
Graphics:	16x36 bits
Game Cartridge:	Tic-Tac-Toe
Codename:	EDSAC
Year:	1952



Output From: Squares

```
  1      1      1
  2      4      3
  3      9      5
  4     16      7
  5     25      9
  6     36     11
  7     49     13
  8     64     15
  9     81     17
```

LongTank 0

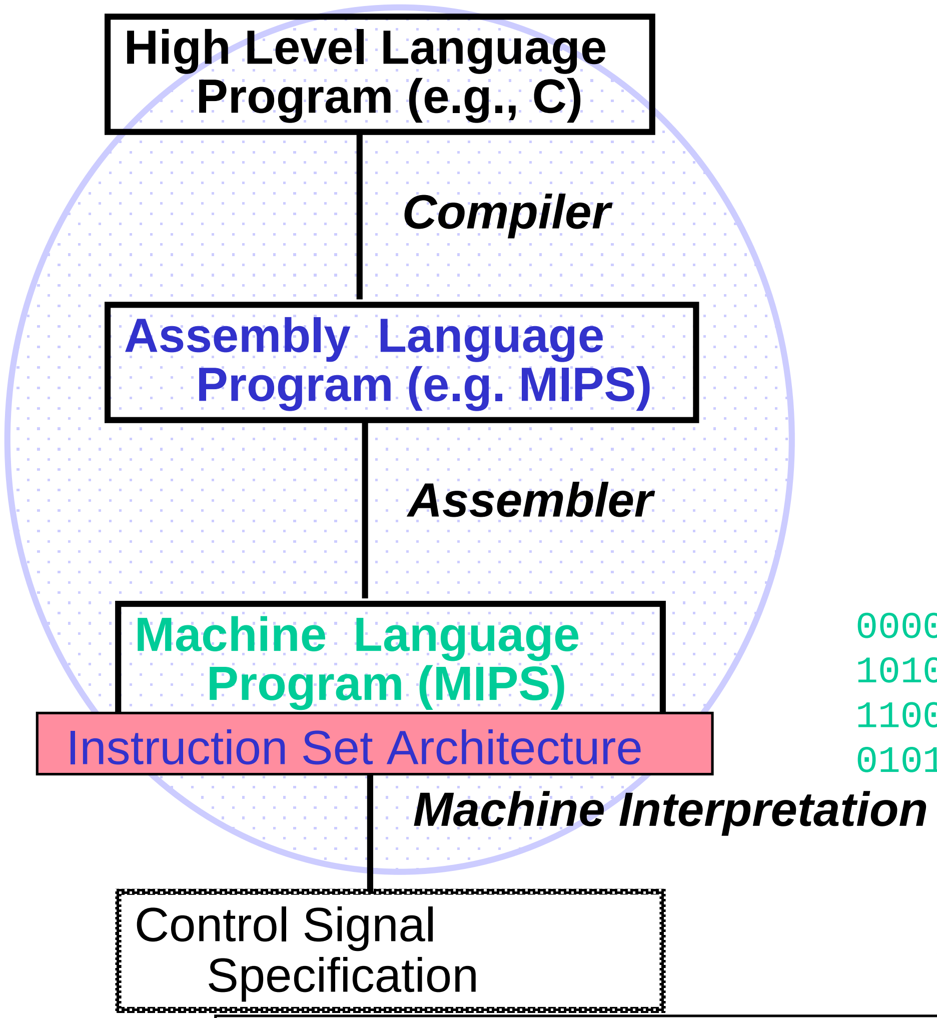
[✓] Short Tanks

http://www.dcs.warwick.ac.uk/~edsac

Brief history

ENIAC:	1940, 18000 Vacuum tubes, 0.1 Mhz, 150 Kwatts, first general purpose computer, 10000 feet^2, 90% down time, Plugboard programming (ROM)
EDSAC:	1949, 3000 Vacuum tubes, 0.5 Mhz, 12 Kwatts, first stored-program computer to operate.
EDVAC:	1956 operational, 1944 designed, 3600 tubes, 10k diodes, 1 Mhz, crashed every 8 hours.
UNIVAC 1:	1951, 1st commerical computer, 5400 tubes, 18k diodes, 2.25 Mhz, 352 feet^2. Sales: 43. Cost $750K.
IBM 701:	1952, total 19 leased, $15000 per month.
UNIVAC 1107:	1960s, Case Institute of Technology, CWRU, http://www.fourmilab.ch/documents/univac/case1107.html

Design Abstractions


High Level Language Program (e.g., C)
Compiler
Assembly Language Program (e.g. MIPS)
Assembler
Machine Language Program (MIPS)
Instruction Set Architecture
Machine Interpretation
Control Signal Specification

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0,0($2)
lw   $t1,4($2)
sw   $t1,0($2)
sw   $t0,4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

```
ALUOP[0:3] <= InstReg[9:11] & MASK
```

An abstraction omits unneeded detail, helps us cope with complexity

C Operators/Operands (page 9 of K&R)

- **Declaration** (announces the properties of variables)
- **Definition** (declaration that allocates memory):

```
int r3;  /* declare r3 as a signed integer */
```

- **Arithmetic operators:** +, -, *, /, % (mod), &, |, ^
- **Assignment statements:**

```
Variable = expression;
celsius = 5 * (fahr - 32) / 9;
```

- **Operands:**

  Variables: a, b, c, r0, r1, celsius
  Constants: 0, 1000, -17, 15, 0xf3, 017

Note: we begin at chapter 3 of the text book

Assembly Operators

- C Semantics: register int rd, rs, rt;
  rd = rs <op> rt;

- Syntax: <op> $rd, $rs, $rt

<op>:	Opcode (or operator) by name
$rd:	Destination, operand getting result
$rs:	1st source operand for operation
$rt:	2nd source operand for operation

- Called an (assembly language) Instruction
- Example

```
add $r1,$r2,$r3          # C Lanaguage: r1=r2+r3;
```

Assembly Operators/Instructions

- **MIPS Assembly Syntax is rigid:**

  1 operation, 3 variables

  Why? Keep Hardware simple via regularity

  Note: Unlike C each line of assembly contains

  at most 1 instruction

- **How do following C statement?**

  a = b + c + d - e; /* a = sum of b, c, d minus d */

- **Break into more primitive instructions**

```
add a, b, c        # a = sum of b & c
add a, a, d        # a = sum of b,c,d
sub a, a, e        # a = b+c+d-e
```

- # is a comment terminated by end of the line

Compilation

- Example: compile by hand this C code:

```
f = (g + h) - (i + j);
```

- First sum of g and h. Where put result?

```
add f,g,h      # f contains g+h
```

- Now sum of `i` and `j`. Where put result?
  - Cannot use `f` !
  - Compiler creates temp variable to hold sum: `t1`

```
add t1,i,j     # t1 contains i+j
```

- Finally produce difference

```
sub f,f,t1     # f=(g+h)-(i+j)
```

Compilation Summary

- C statement (5 operands, 3 operators):

```
f = (g + h) - (i + j);
```

- Becomes 3 assembly instructions
  (6 unique operands, 3 operators):

```
add f,g,h     # f contains g+h
add t1,i,j    # t1 contains i+j
sub f,f,t1    # f=(g+h)-(i+j)
```

- Big Idea: compiler translates notation from 1 level of abstraction to lower level
- In general, each line of C produces many assembly instructions
  - –One reason why people program in C vs. Assembly; fewer lines of code
  - –Other reasons? Portability, Optimization

Registers: Performance issue

- Unlike C++, assembly instructions cannot use variables

Why not? Keep Hardware Simple

- Instruction operands are registers:
  - Limited to 32 registers in MIPS ($r0 - $r31)
  - Also, each MIPS register is 32 bits wide
  - The width of the register is called the word size
  - C language “int” is the word size of the register

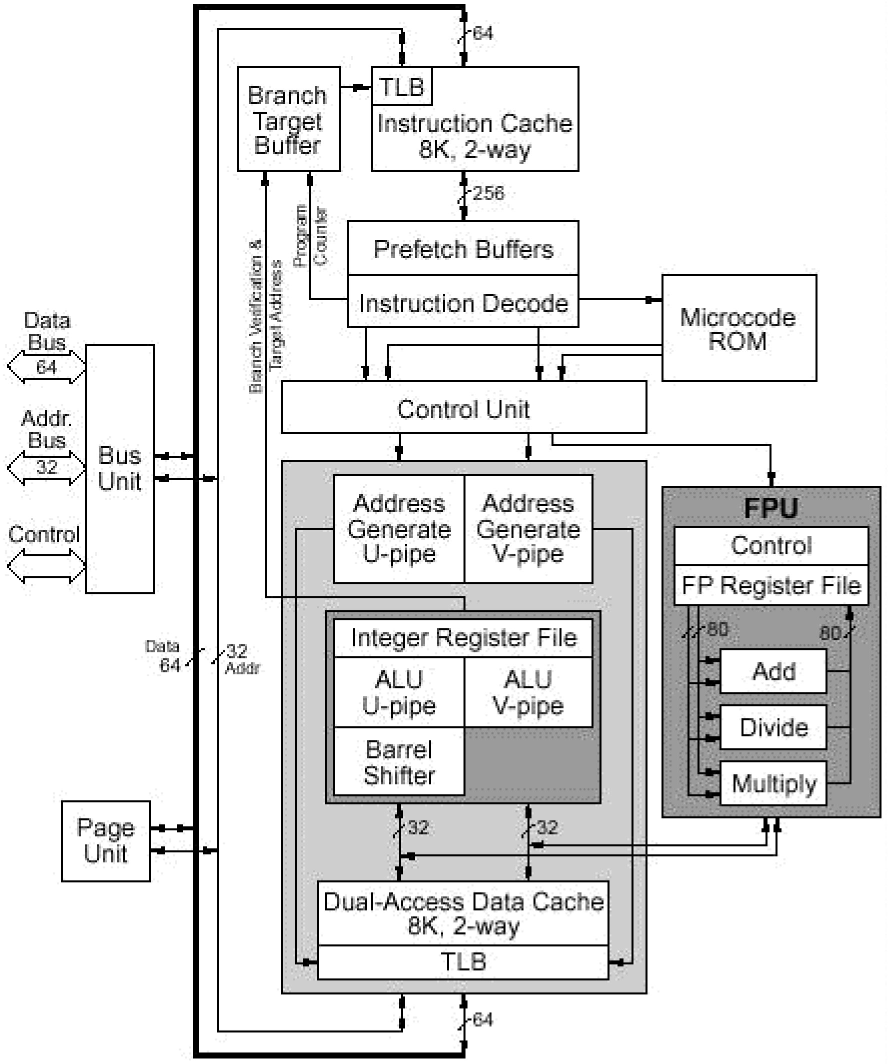
Why 32? Smaller is faster (based on technology)

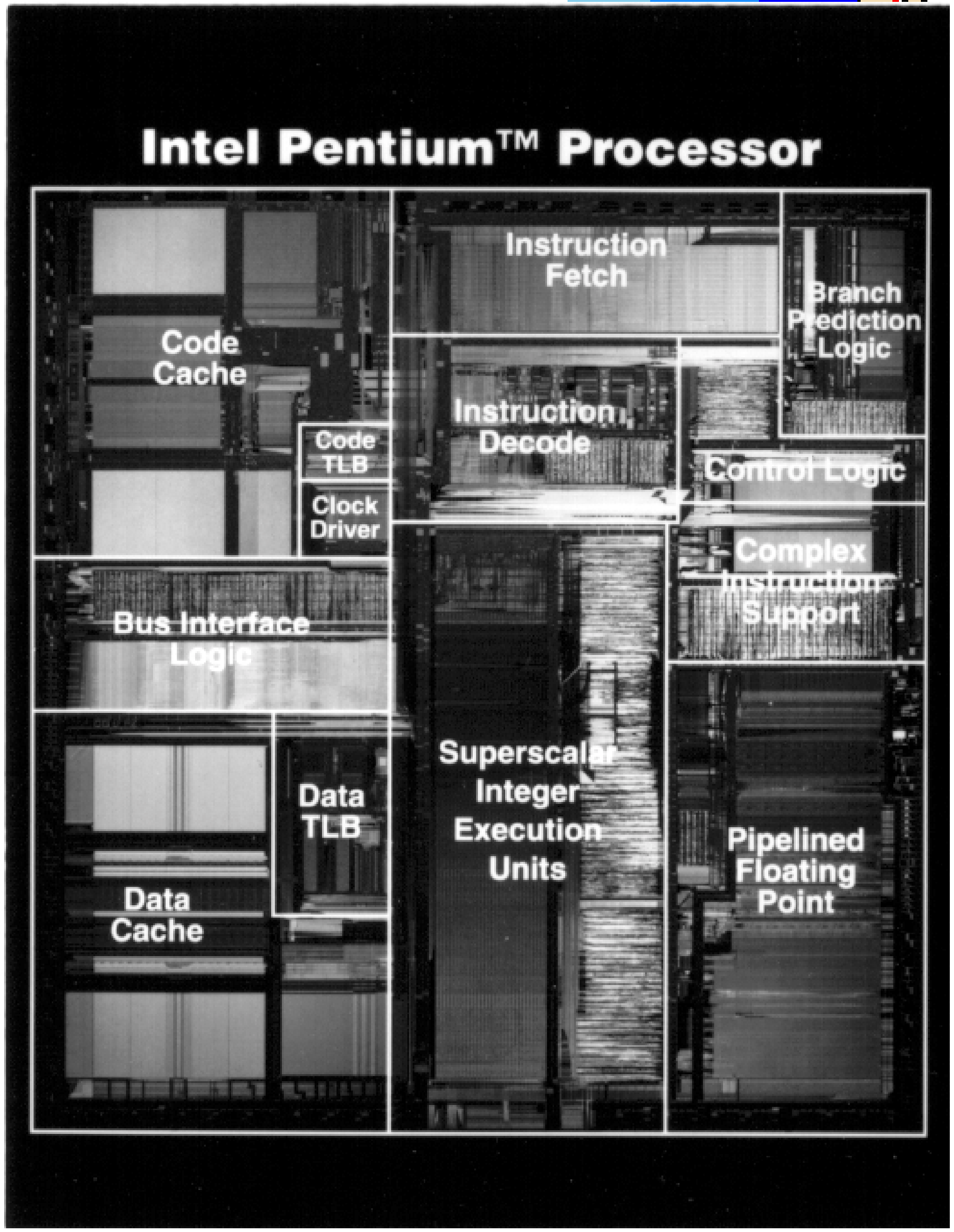
Pentium I: registers example

Process: 0.8-micron 5 Volt BiCMOS

Year: 1993 / 3.1 million transistors

Clock: 60 or 66 MHz





Ref: http://www.laynetworks.com/users/webs/cs12_2.htm

Compilation using Registers

- Compile by hand using registers:

```
f = (g + h) - (i + j);
        # assign registers
        # $r5=“f”,  $r1=“g”,   $r2=“h”
        # $r3=“i”   $r4=“j”
```

- MIPS Instructions:

```
add $r6,$r1,$r2        # $r6 = g+h
add $r7,$r3,$r4        # $r7 = i+j
sub $r5,$r6,$r7        # f=(g+h)-(i+j)
```

Arithmetic operators

• /* default */	register int r0,r1,…,r31;
• /* explicit */	register signed int r0,…;
• add $rd,$rs,$rt	rd = rs + rt;
• sub $rd,$rs,$rt	rd = rs - rt;
• addi $rt,$rs,signed16	rt = rs + 1847;
• subi $rt,$rs,signed16	rt = rs - 1931;

•	register unsigned int r0,r1,…,r31;
• addu $rd,$rs,$rt	rd = rs + rt;
• subu $rd,$rs,$rt	rd = rs - rt;
• addiu $rt,$rs,signed16	rt = rs + 1847;

- What's missing from unsigned? • subiu? Do we need it?
- subiu $rt,$rs,1931? addui $rt,$rs,-1931

Pseudo instructions: move

- Suppose we only had add, sub, addi, subi instructions
  How could we do the following C language operation?

```
register signed int r0=0, r1, r2, r5, r6;
r2 = r6;
```

Note: that some processors (i.e. Mips, Sun Microsystems, sparc) hard code $r0 to zero.

```
addi  $r2,$r6,0               # also: subi  $r2,$r6,0
```

```
# also
add   $r2,$r6,$r0             # also: sub   $r2,$r6,$r0
```

- Pseudo instructions extend the assembly language by substituting with other machine instructions:

```
move  $rd,$rs                 # addi  $r2,$r6,0
```

Pseudo instruction: li, load immediate

- Suppose we only had add, sub, addi, subi instructions
  How could we do the following C language operation?

```
register signed int r0=0, r1, r2, r5, r6;
r2 = 1847;
```

```
addi   $r2,$r0,1847          # mips $r0 is always zero
```

- Pseudo instructions extend the assembly language by substituting with other machine instructions:

```
li $rd,signed16                      # addi  $r2,$r0,signed16
```

bitwise C operators

- /* default */ register int r0,r1,…,r31;
- /* explicit */ register signed int r0,…;
- and $rd,$rs,$rt rd = rs & rt;
- or $rd,$rs,$rt rd = rs | rt;
- xor $rd,$rs,$rt rd = rs ^ rt;
- not $rd,$rs rd = ~rs; /* pseudo instruction */
- andi $rt,$rs,signed16 rt = rs & 1847;
- ori $rt,$rs,signed16 rt = rs | 1931;
- xori $rt,$rs,signed16 rt = rs ^ 1931;
- why is there no "noti"?
- <u>Is the follow correct?</u>
- /* explicit */ register unsigned int r0,r1,…,r31;
- andu $rd,$rs,$rt rd = rs & rt;
- Use "and", Bitwise operators are not arithmetic operators!

Pseudo instructions: clear & not

- review bitwise operators (looking at 1 bit at a time):

not:	0 = ~1;	1 = ~0;		
and:	1 = 1 & 1;	0 = 0 & x;	0 = x & 0; /* conclusive */	
or:	0 = 0 | 0;	1 = 1 | x;	1 = x | 1; /* inclusive */	
xor:	0 = 0 ^ 0;	0 = 1 ^ 1;	1 = 0 ^ 1;	1 = 1 ^ 0;
	also called exclusive or, difference, mod 2 add			

- How is the“clear $rd” pseudo-instruction implemented?

```
clear $rd                        # xori $rd,$rd,$rd
```

- How is the“not rd,rs” pseudo-instruction implemented?

```
not   $rd,$rs                    # xori $rd,$rs,0xffff
```

Pseudo instructions: Multiply and Divide

- **mul $rd, $rs, $rt** # signed pseudo instruction
  - mult $rs, $rt # $(hi:lo)_{64} = rs_{32} * rt_{32}$
  - mflo $rd # rd = lo;
- **mulou $rd, $rs, $rt** # unsigned pseudo instruction
  - multu $rs, $rt # $(hi:lo)_{64} = rs_{32} * rt_{32}$
  - mflo $rd # rd = lo;

- **div $rd, $rs, $rt** # pseudo instruction
  - div $rs, $rt # $(quotient=lo\ rem=hi)_{64} = rs_{32} / rt_{32}$
  - mflo $rd # rd = lo;
- **rem $rd, $rs, $rt** # pseudo instruction
  - div $rs, $rt # $(quotient=lo\ rem=hi)_{64} = rs_{32} / rt_{32}$
  - mfhi $rd # rd = hi;

Shift instructions: sll, srl, sra

- register unsigned int r0,r1,…,r31;
- sll $rd,$rt,const5 rd = rt << 11; /* rd = rt * 2^{11} */
- srl $rd,$rt,const5 rd = rt >> 21; /* rd = rt / 2^{21} */

- register int r0,r1,…,r31;
- sra $rd,$rt,const5 rd = rt >> 4; /* rd = rt / 2^{4} = rt / 16 */

Do we need shift instruction since we have Mul & div?

No, shift left/right can be done with mul/div instructions

example: sra $r2,$r2,4

```
addi   $r1,$r1,16    # 16= 2^4
div    $r2,$r1
```

Performance: shift instructions are faster than mul/div

C Language allows programmer access to shift via >> << ops