# CS 423
# Computer Architecture
# Spring 2012

# Lecture 01: Introduction

Ozcan Ozturk

http://www.cs.bilkent.edu.tr/~ozturk/cs423/

[Adapted from *Computer Organization and Design*,
Patterson & Hennessy, © 2005, UCB]

# Course Administration

- Instructor: Ozcan Ozturk ozturk@cs.bilkent
  EA421
  Office Hrs: T 08:40-10:30
- TA: Ismail Akturk
  Office Hrs: posted on the course web page
- URL: http://www.cs.bilkent.edu.tr/~ozturk/cs423/
- Text: Required: *Computer Org and Design*, 4th Edition, Patterson and Hennessy ©2006
- Slides: pdf on the course web page after lecture

# Grading Information

- Grade determinates
  - Midterm Exam ~25%
    - March ??? , Location: TBD
  - Final Exam ~35%
    - May ???, Location TBD
  - Homeworks (3-5) ~30%
    - Due at the beginning of class (or, if its code to be submitted electronically, by 17:00 on the due date). No late assignments will be accepted.
  - Class participation & pop quizzes ~10%
- Let me know about midterm exam conflicts ASAP
- Grades will be posted on the course homepage
  - Must submit email request for change of grade after discussions with the TA (Homeworks/Quizzes) or instructor (Exams)

# Course Content

- Content
  - Principles of computer architecture: CPU datapath and control unit design (single-issue pipelined, superscalar, VLIW), memory hierarchies and design, I/O organization and design, advanced processor design – chip multiprocessors
- Course goals
  - To learn the organizational paradigms that determine the capabilities and performance of computer systems. To understand the interactions between the computer's architecture and its software so that future software designers (compiler writers, operating system designers, database programmers, …) can achieve the best cost-performance trade-offs and so that future architects understand the effects of their design choices on software applications.
- Course prerequisites
  - CS 224 Computer Organization

# What You Should Know – CS223 and CS224

- Basic logic design & machine organization
  - logical minimization, FSMs, component design
  - processor, memory, I/O
- Create, assemble, run, debug programs in an assembly language
  - MIPS preferred
- Create, compile, and run C (C++, Java) programs
- Create, organize, and edit files and run programs on Unix/Linux

# Introduction

- This course is all about how computers work
- But what do we mean by a computer?
  - Different types: desktop, servers, embedded devices
  - Different uses: automobiles, graphics, finance, genomics…
  - Different manufacturers: Intel, Apple, IBM, Microsoft, Sun…
  - Different underlying technologies and different costs!
- Analogy: Consider a course on “automotive vehicles”
  - Many similarities from vehicle to vehicle (e.g., wheels)
  - Huge differences from vehicle to vehicle (e.g., gas vs. electric)
- Best way to learn:
  - Focus on a specific instance and learn how it works
  - While learning general principles and historical perspectives

# Why learn this stuff?

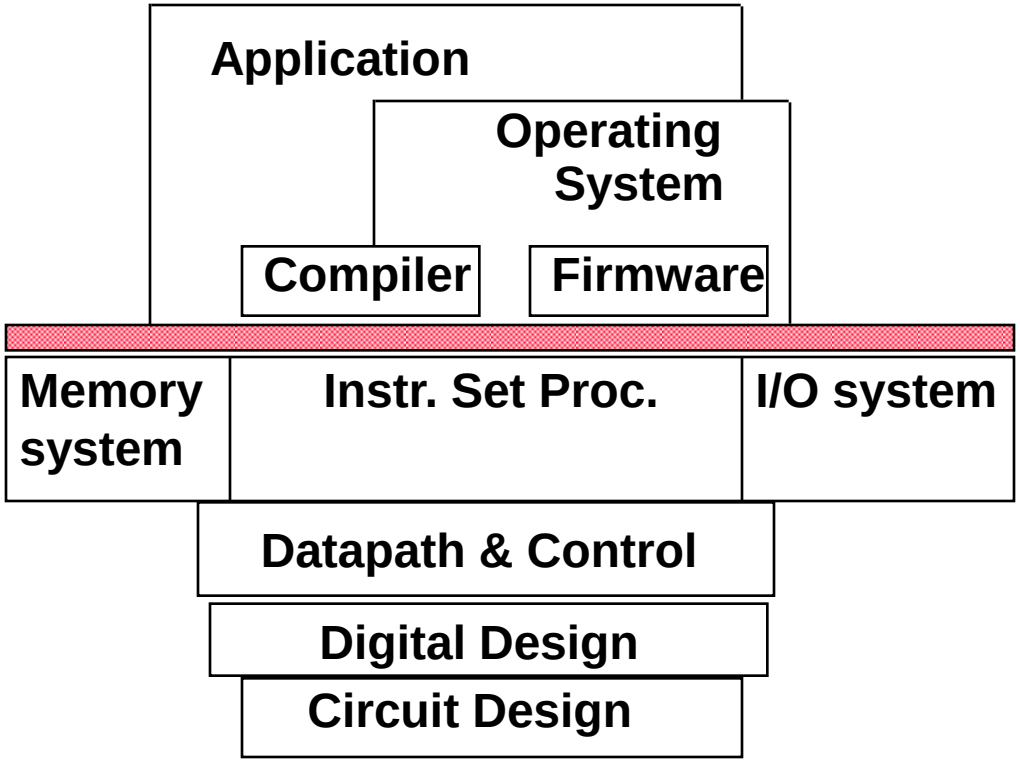
- You want to call yourself a “computer engineer”
- You want to build software people use (need performance)
- You need to make a purchasing decision or offer “expert” advice
- Both Hardware and Software affect performance:
  - Algorithm determines number of source-level statements
  - Language/Compiler/Architecture determine machine instructions
- Processor/Memory determine how fast instructions are executed
- Assessing and Understanding Performance

# Course Structure

- Design focused class
  - Various homework assignments throughout the semester
  - Simulation of architecture alternatives using SimpleScalar

- Lectures:
  - http://www.cs.bilkent.edu.tr/~ozturk/cs423/syllabus.htm

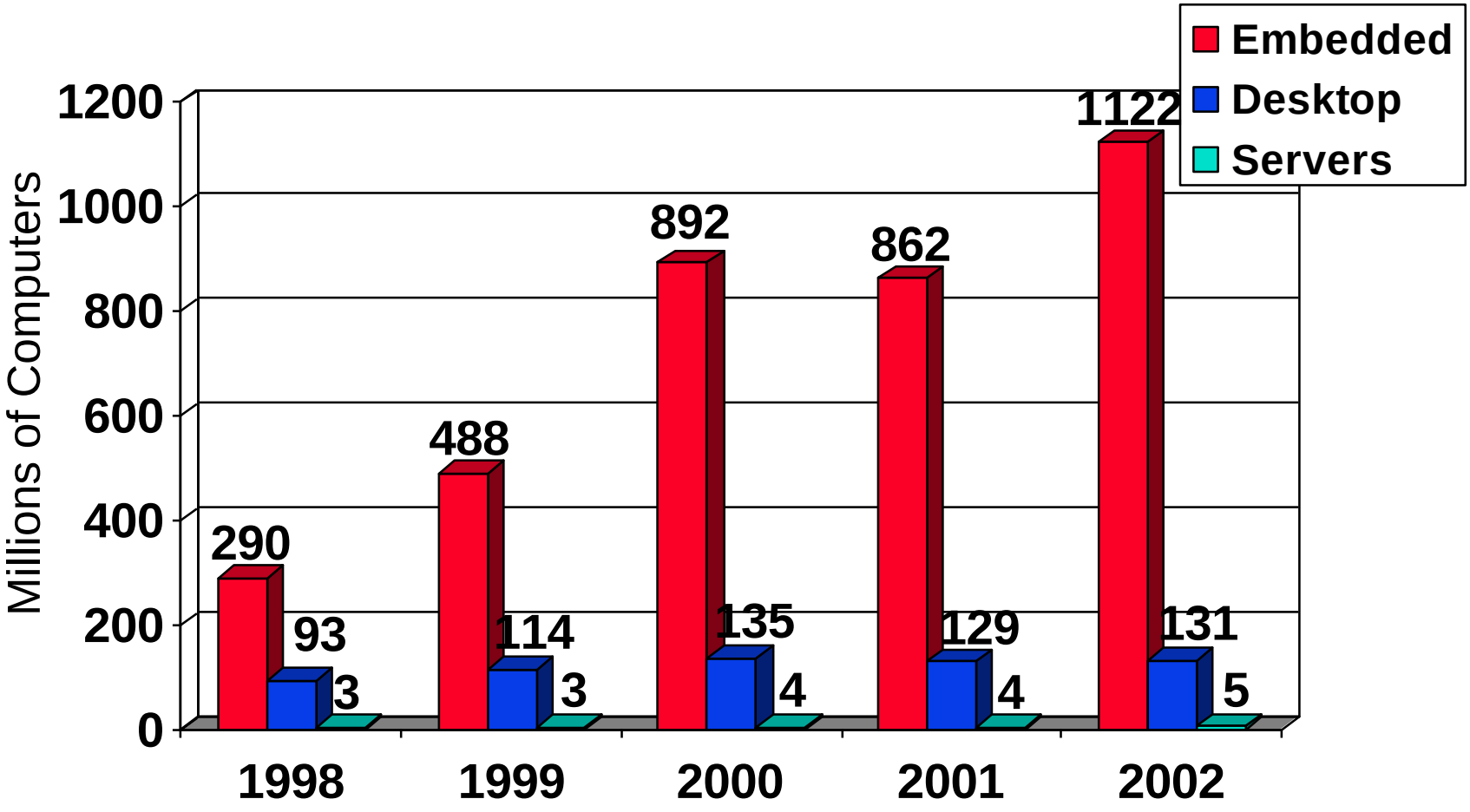
# How Do the Pieces Fit Together?




Instruction Set Architecture

- Coordination of many *levels of abstraction*
- Under a rapidly changing set of forces
- Design, measurement, *and* evaluation

# Where is the Market?



# Instruction Set Architecture (ISA)

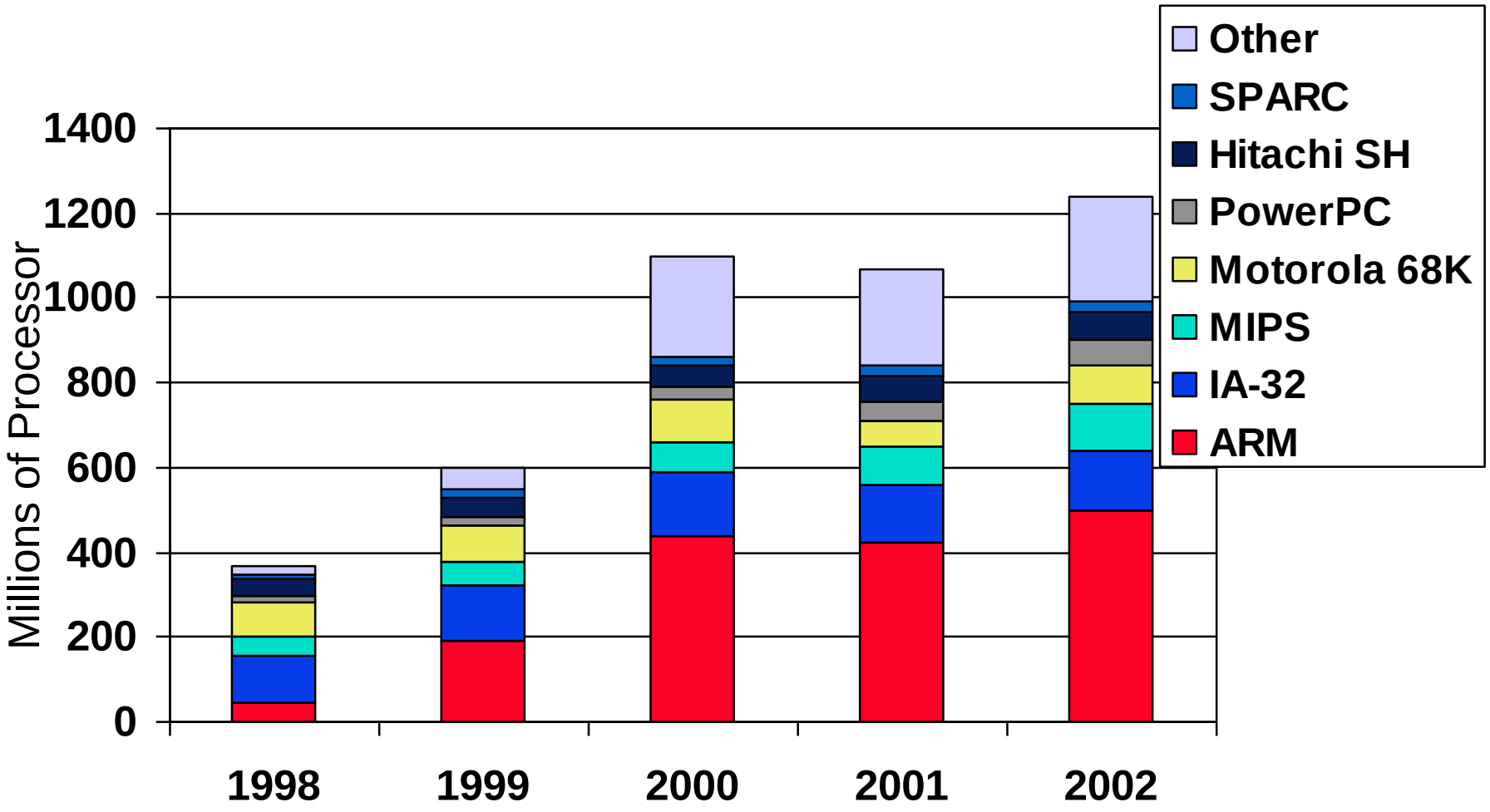
- ISA: An abstract interface between the hardware and the lowest level software of a machine that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on.

  "... the attributes of a [computing] system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls, the logic design, and the physical implementation."
  – Amdahl, Blaauw, and Brooks, 1964

  - Enables implementations of varying cost and performance to run identical software

- ABI (application binary interface): The user portion of the instruction set plus the operating system interfaces used by application programmers. Defines a standard for binary portability across computers.

# ISA Type Sales




PowerPoint “comic” bar chart with approximate values (see text for correct values)

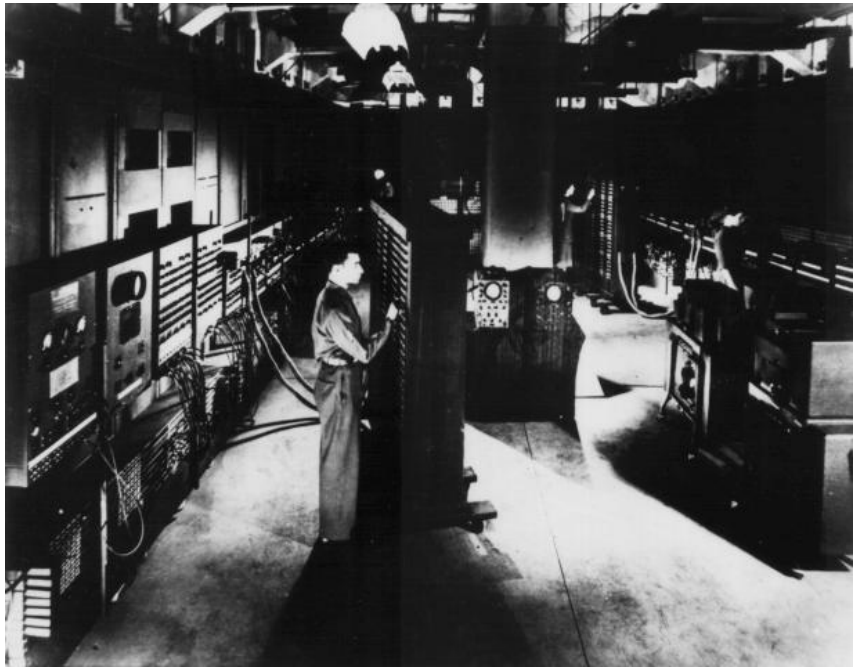
# Moore's Law

- In 1965, Gordon Moore predicted that the number of transistors that can be integrated on a die would double every 18 to 24 months (i.e., grow exponentially with time).

- Amazingly visionary – million transistor/chip barrier was crossed in the 1980's.
  - 2300 transistors, 1 MHz clock (Intel 4004) - 1971
  - 16 Million transistors (Ultra Sparc III)
  - 42 Million transistors, 2 GHz clock (Intel Xeon) – 2001
  - 55 Million transistors, 3 GHz, 130nm technology, 250mm$^2$ die (Intel Pentium 4) - 2004
  - 140 Million transistor (HP PA-8500)

# Historical Perspective

- ENIAC built in World War II was the first general purpose computer
  - Used for computing artillery firing tables
  - 80 feet long by 8.5 feet high and several feet wide
  - Each of the twenty 10 digit registers was 2 feet long
  - Used 18,000 vacuum tubes
  - Performed 1900 additions per second

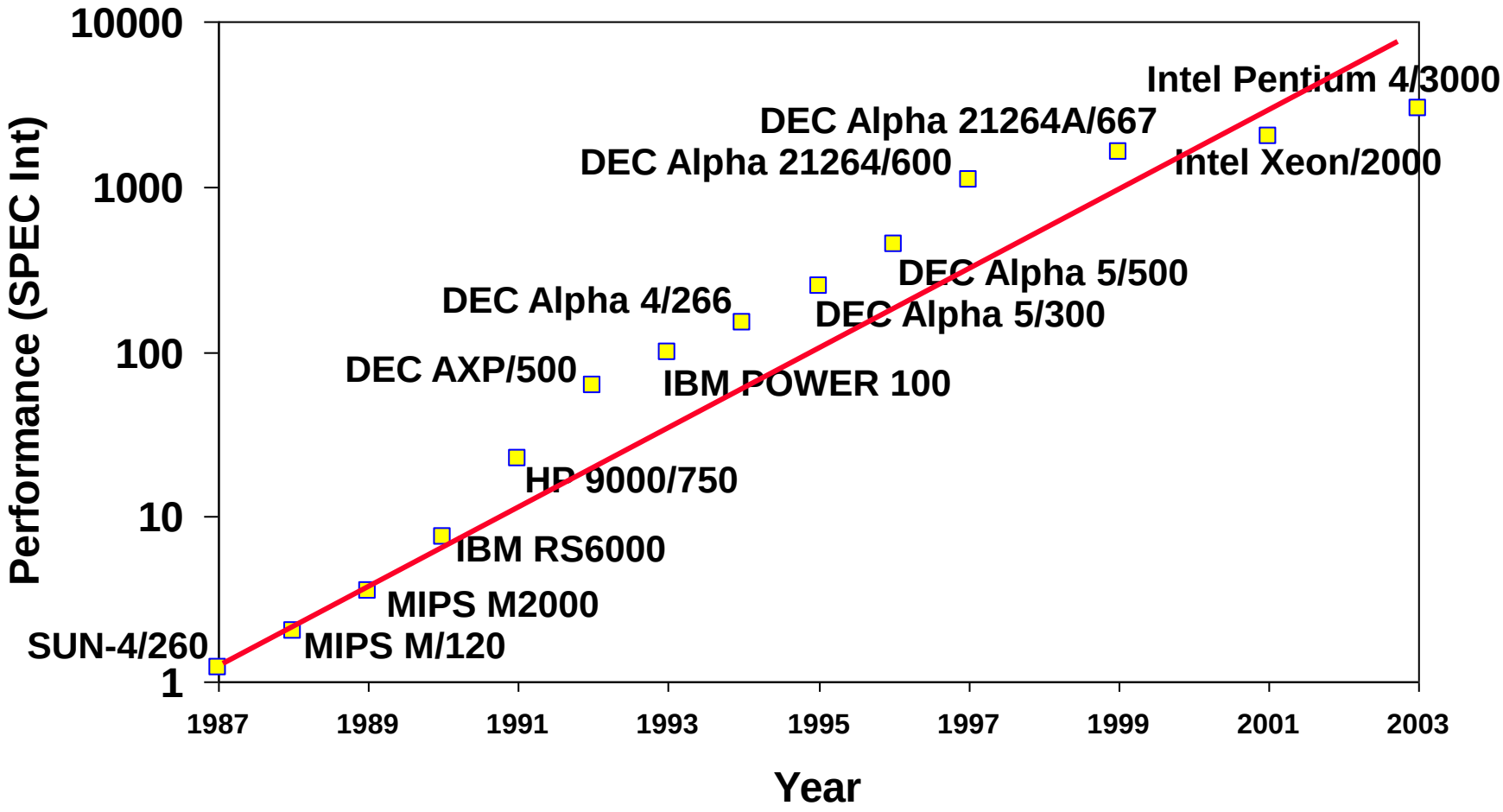


**–Since then:**

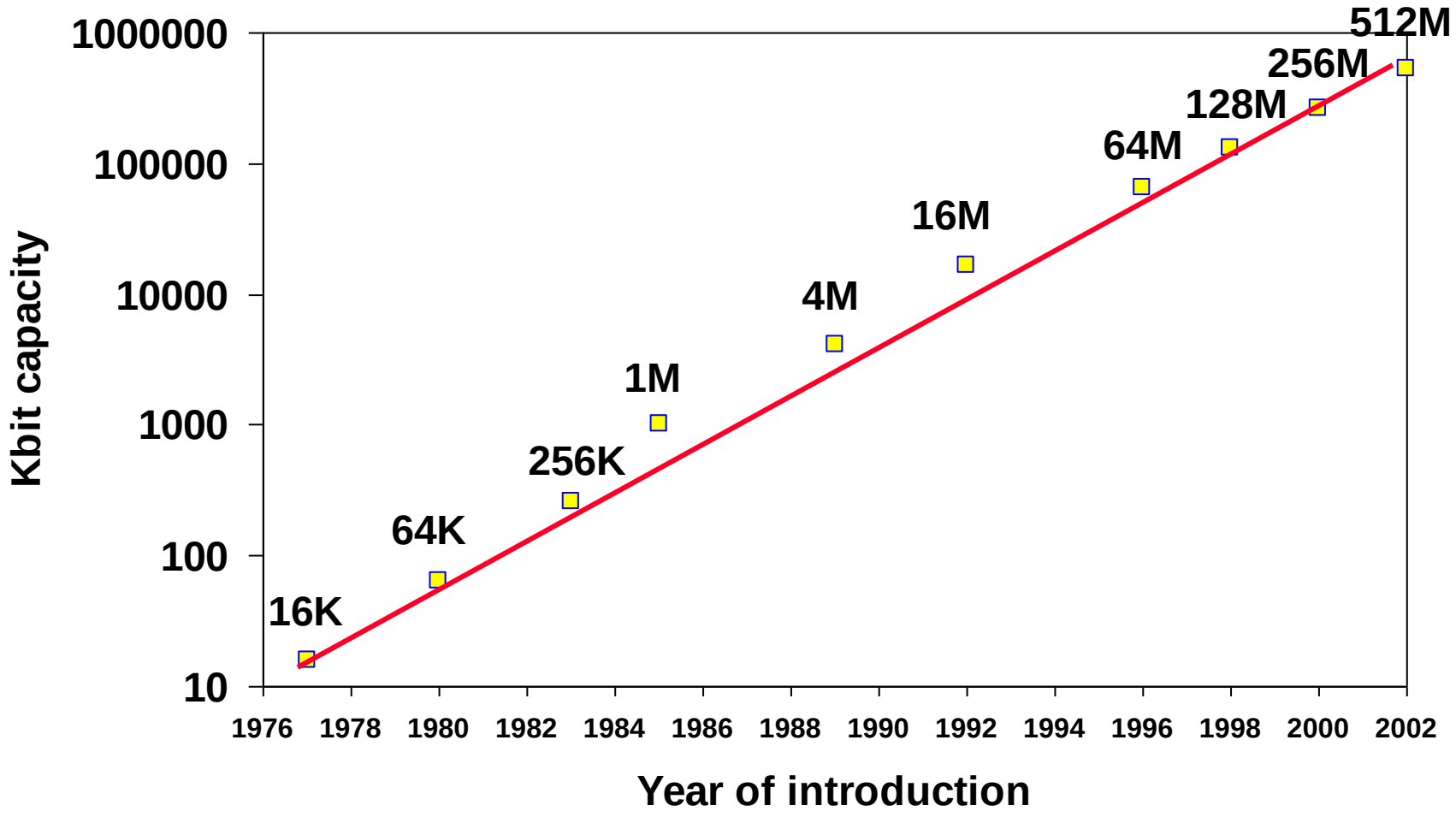
**Moore's Law:**

**transistor capacity doubles every 18-24 months**

# Processor Performance Increase



# DRAM Capacity Growth



# Impacts of Advancing Technology

- Processor
  - logic capacity: increases about 30% per year
  - performance: 2x every 1.5 years

ClockCycle = 1/ClockRate

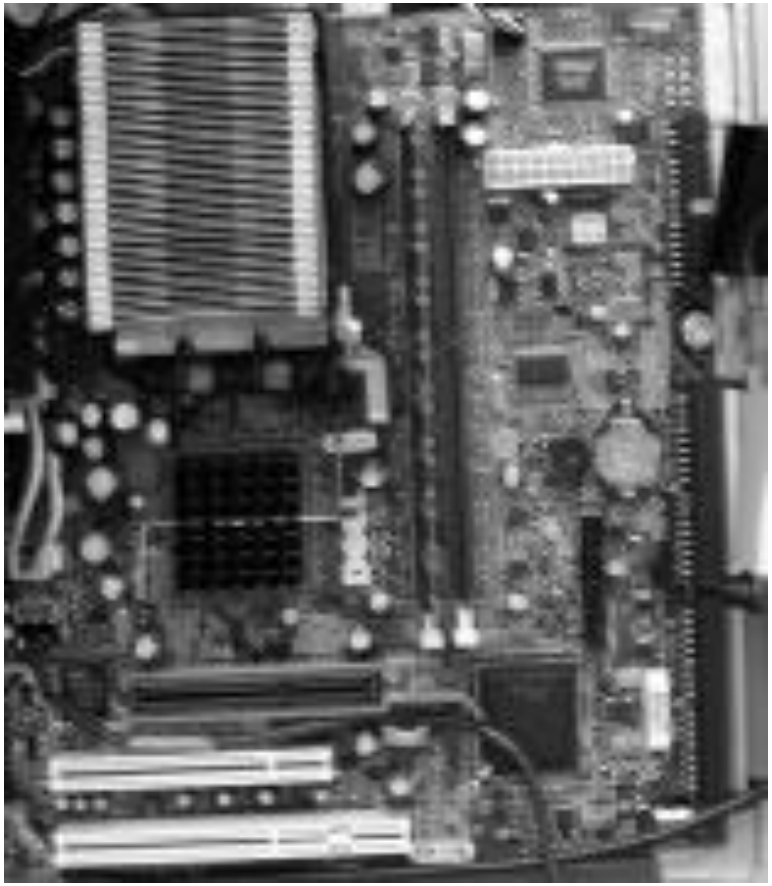
500 MHz ClockRate = 2 nsec ClockCycle

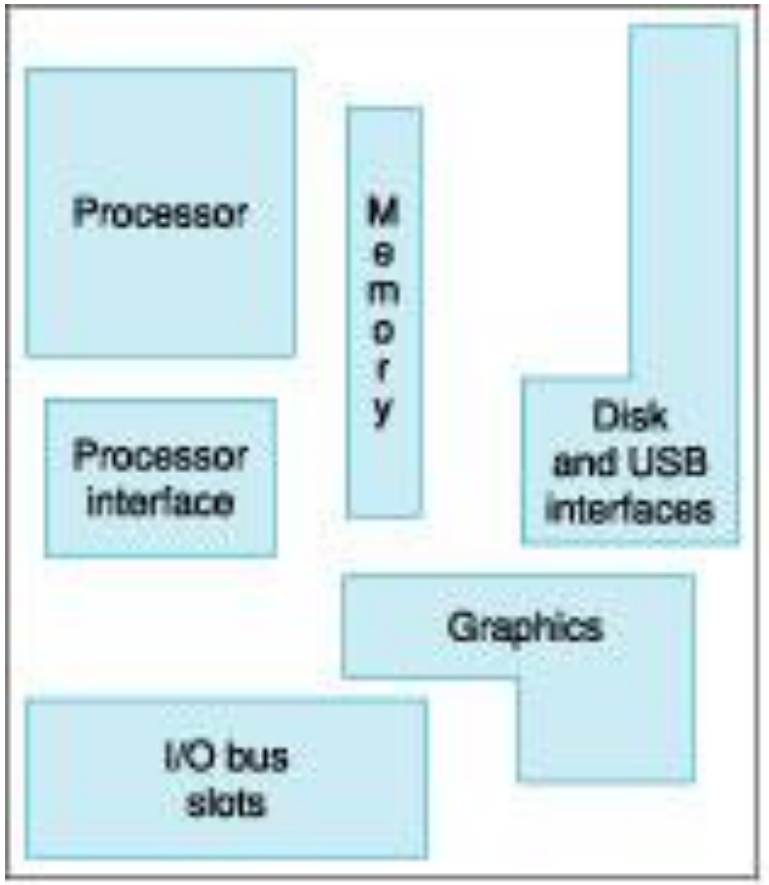
1 GHz ClockRate = 1 nsec ClockCycle

4 GHz ClockRate = 250 psec ClockCycle

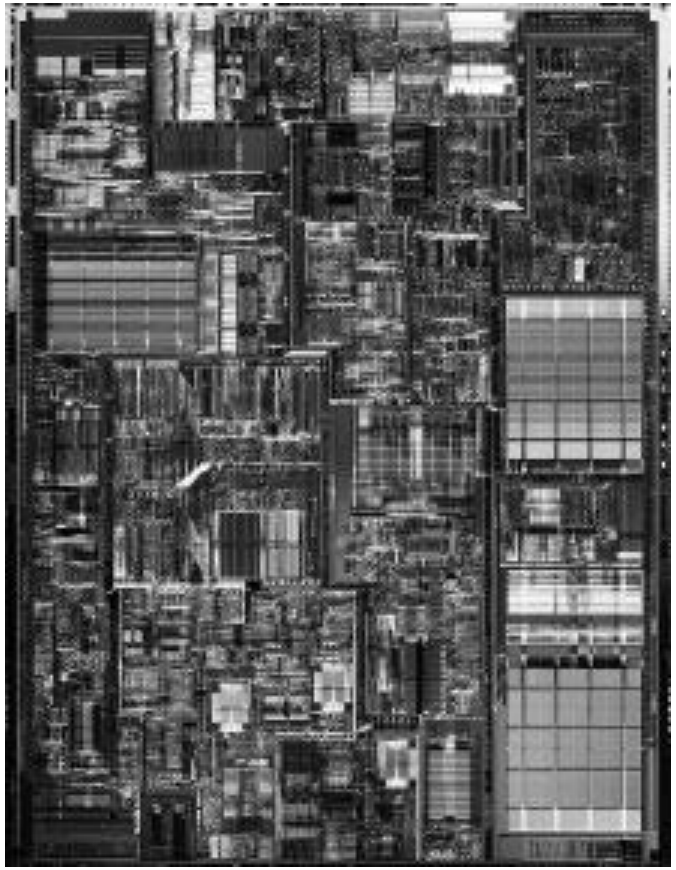
- Memory
  - DRAM capacity: 4x every 3 years, now 2x every 2 years
  - memory speed: 1.5x every 10 years
  - cost per bit: decreases about 25% per year
- Disk
  - capacity: increases about 60% per year

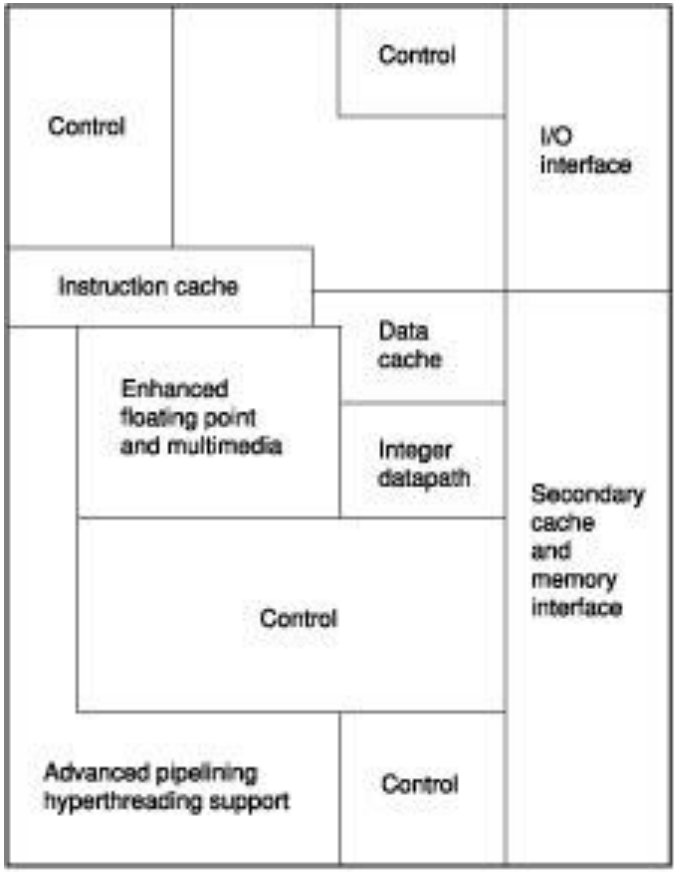
# PC Motherboard Closeup





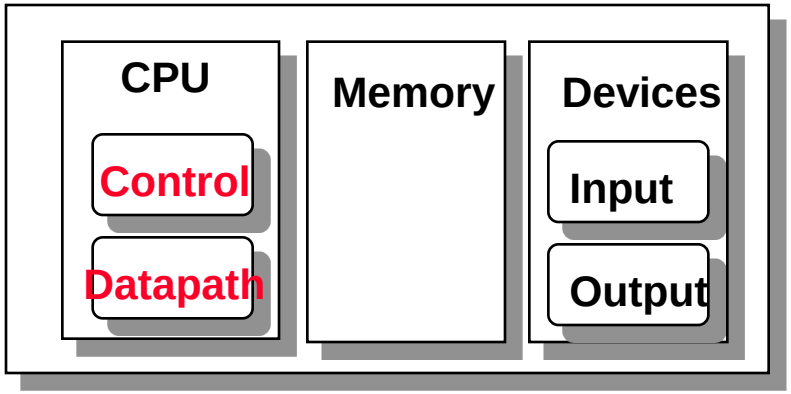
# Inside the Pentium 4 Processor Chip

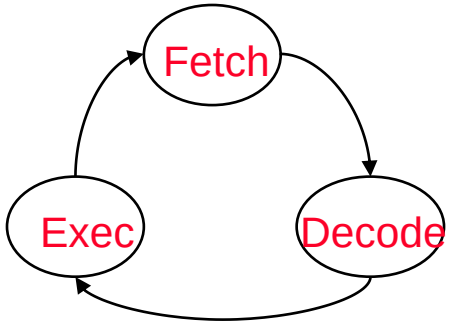




# (vonNeumann) Processor Organization

- Control needs to
  1. input instructions from Memory
  2. issue signals to control the information flow between the Datapath components and to control what operations they perform
  3. control instruction sequencing




- Datapath needs to have the
  - components – the functional units and storage (e.g., register file) needed to execute instructions
  - interconnects - components connected so that the instructions can be accomplished and so that data can be loaded from and stored to Memory

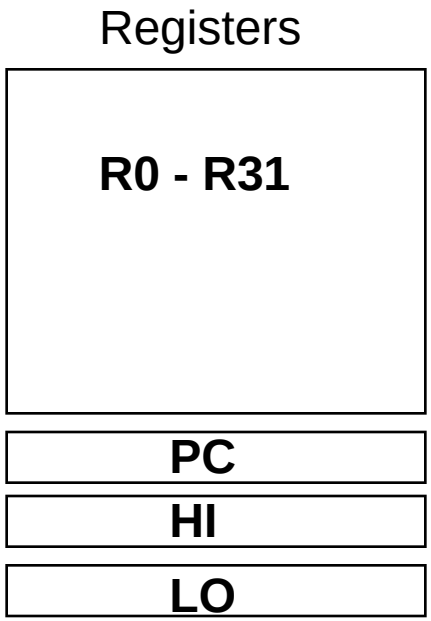
# RISC - Reduced Instruction Set Computer

- RISC philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
- MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC, Intel (Compaq) Alpha, …
- Instruction sets are measured by how well compilers use them as opposed to how well assembly language programmers use them

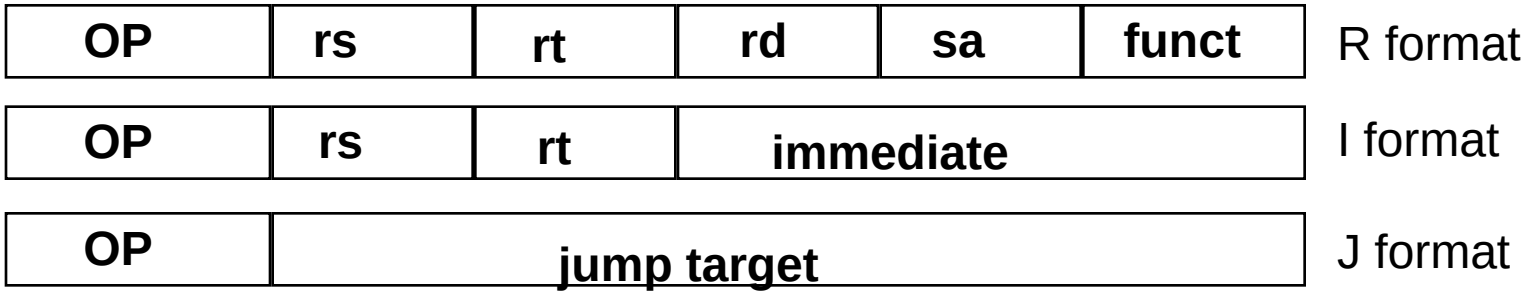
*Design goals: speed, cost (design, fabrication, test, packaging), size, power consumption, reliability, memory space (embedded systems)*

# MIPS R3000 Instruction Set Architecture (ISA)

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special




**3 Instruction Formats: all 32 bits wide**



# Aside: Beyond Numbers

- American Std Code for Info Interchange (ASCII): 8-bit bytes representing characters

| ASCII | Char | | ASCII | Char | ASCII | Char | ASCII | Char | | ASCII | Char | ASCII | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Null | | 32 | space | 48 | 0 | 64 | @ | | 96 | ` | 112 | p |
| 1 | | | 33 | ! | 49 | 1 | 65 | A | | 97 | a | 113 | q |
| 2 | | | 34 | “ | 50 | 2 | 66 | B | | 98 | b | 114 | r |
| 3 | | | 35 | # | 51 | 3 | 67 | C | | 99 | c | 115 | s |
| 4 | EOT | | 36 | $ | 52 | 4 | 68 | D | | 100 | d | 116 | t |
| 5 | | | 37 | % | 53 | 5 | 69 | E | | 101 | e | 117 | u |
| 6 | ACK | | 38 | & | 54 | 6 | 70 | F | | 102 | f | 118 | v |
| 7 | | | 39 | ‘ | 55 | 7 | 71 | G | | 103 | g | 119 | w |
| 8 | bksp | | 40 | ( | 56 | 8 | 72 | H | | 104 | h | 120 | x |
| 9 | tab | | 41 | ) | 57 | 9 | 73 | I | | 105 | i | 121 | y |
| 10 | LF | | 42 | * | 58 | : | 74 | J | | 106 | j | 122 | z |
| 11 | | | 43 | + | 59 | ; | 75 | K | | 107 | k | 123 | { |
| 12 | FF | | 44 | , | 60 | < | 76 | L | | 108 | l | 124 | \| |
| | | | | | | | | | | | | | |
| 15 | | | 47 | / | 63 | ? | 79 | O | | 111 | o | 127 | DEL |

# MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

```
add  $t0, $s1, $s2
sub  $t0, $s1, $s2
```

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

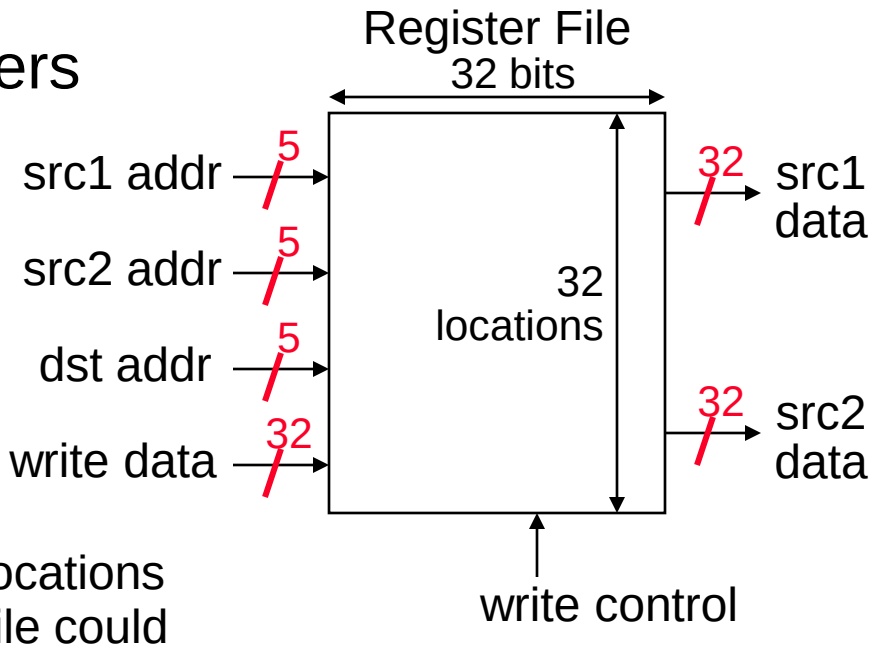

- Operand order is fixed (destination first)
- Those operands are all contained in the datapath's register file (`$t0,$s1,$s2`) – indicated by $

# Aside: MIPS Register Convention

| Name | Register Number | Usage | Preserve on call? |
|---|---|---|---|
| $zero | 0 | constant 0 (hardware) | n.a. |
| $at | 1 | reserved for assembler | n.a. |
| $v0 - $v1 | 2-3 | returned values | no |
| $a0 - $a3 | 4-7 | arguments | yes |
| $t0 - $t7 | 8-15 | temporaries | no |
| $s0 - $s7 | 16-23 | saved values | yes |
| $t8 - $t9 | 24-25 | temporaries | no |
| $gp | 28 | global pointer | yes |
| $sp | 29 | stack pointer | yes |
| $fp | 30 | frame pointer | yes |
| $ra | 31 | return addr (hardware) | yes |

# MIPS Register File

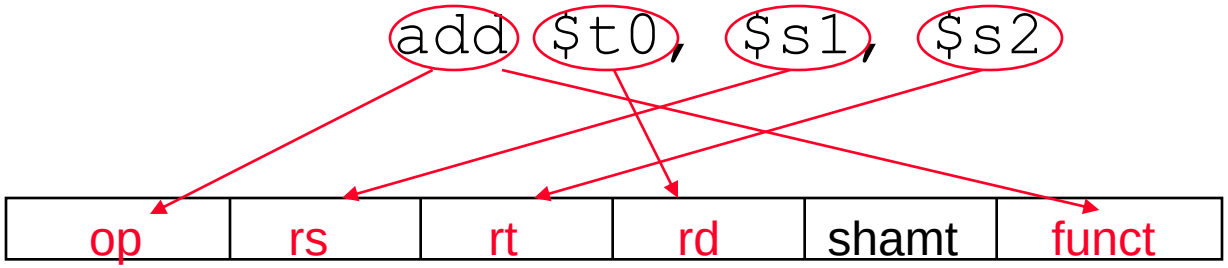
- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port




- Registers are
  - Faster than main memory
    - But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
    - Read/write port increase impacts speed quadratically
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)

# Machine Language - Add Instruction

- Instructions, like registers and words of data, are 32 bits long
- Arithmetic Instruction Format (R format):




| | | |
|---|---|---|
| op | 6-bits | opcode that specifies the operation |
| rs | 5-bits | register file address of the first source operand |
| rt | 5-bits | register file address of the second source operand |
| rd | 5-bits | register file address of the result's destination |
| shamt | 5-bits | shift amount (for shift instructions) |
| funct | 6-bits | function code augmenting the opcode |

# MIPS Memory Access Instructions

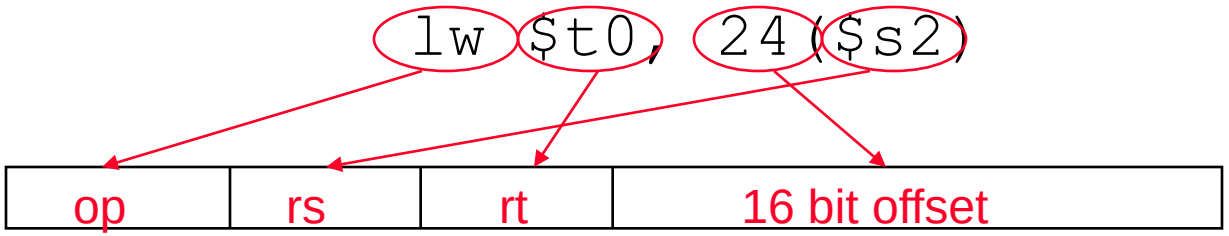
- MIPS has two basic data transfer instructions for accessing memory

```
lw   $t0, 4($s3)   #load word from memory

sw   $t0, 8($s3)   #store word to memory
```

- The data is loaded into (lw) or stored from (sw) a register in the register file – a 5 bit address
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
  - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register
  - Note that the offset can be positive or negative

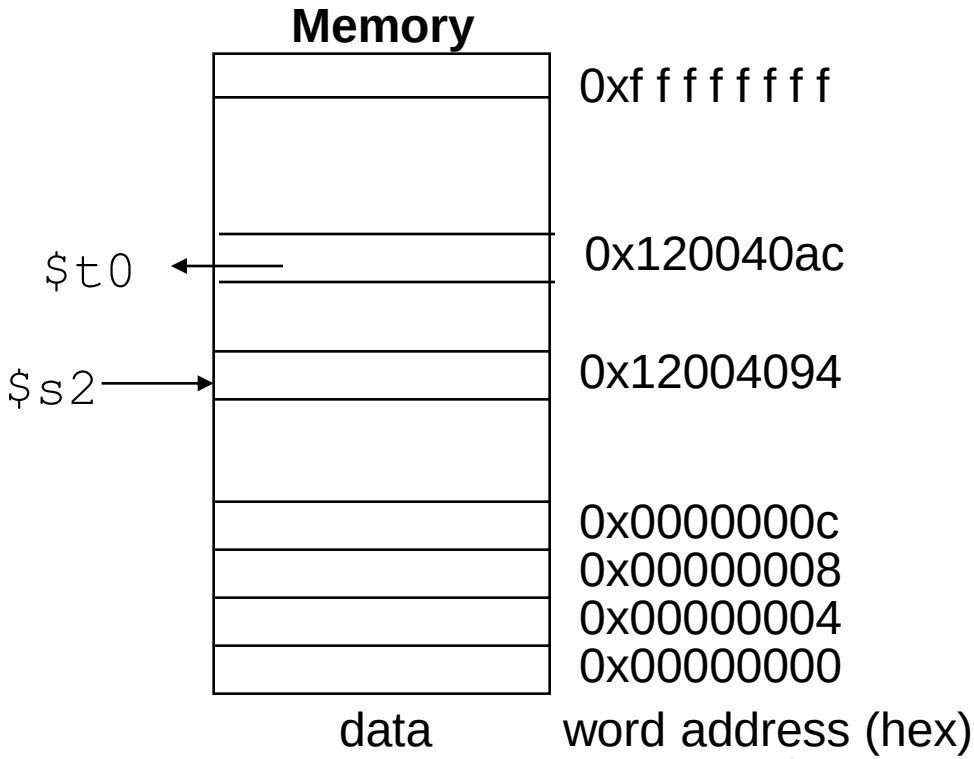
# Machine Language - Load Instruction

- Load/Store Instruction Format (I format):




$24_{10}$ + $s2 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



# MIPS Control Flow Instructions

- MIPS conditional branch instructions:

```
bne $s0, $s1, Lbl #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl #go to Lbl if $s0=$s1
```

  - Ex:

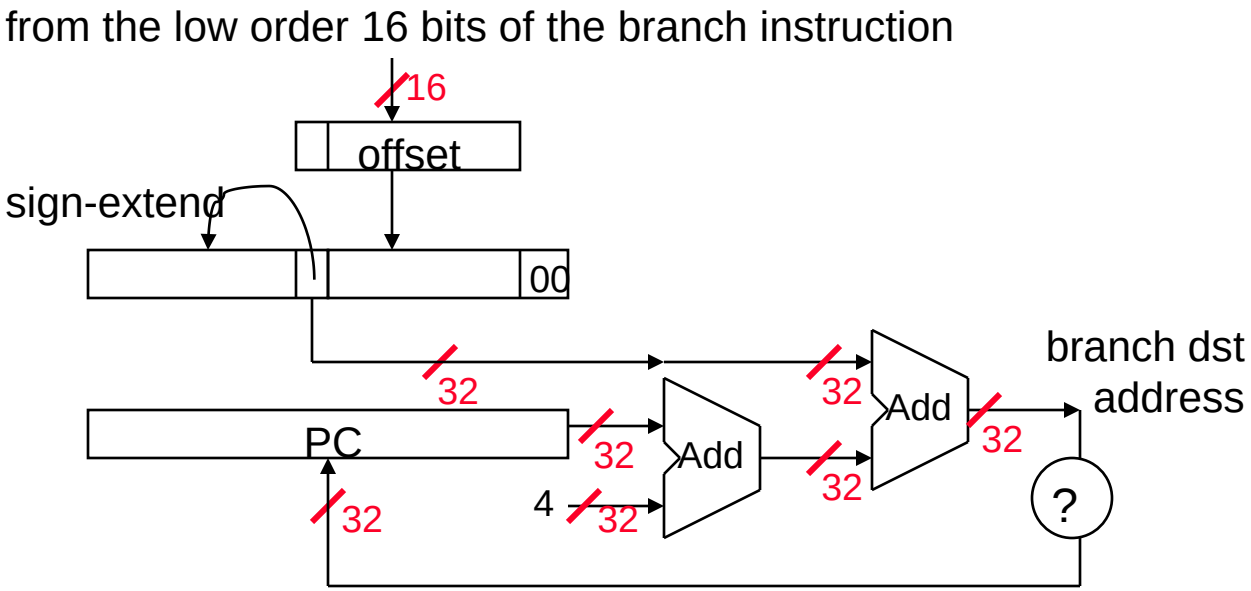
```
        if (i==j) h = i + j;

        bne $s0, $s1, Lbl1
        add $s3, $s0, $s1
Lbl1:   ...
```

- Instruction Format (I format):


- How is the branch destination address specified?

# Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset
  - which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
  - limits the branch distance to $-2^{15}$ to $+2^{15}-1$ instructions from the (instruction after the) branch instruction, but most branches are local anyway



# More Branch Instructions

- We have `beq`, `bne`, but what about other kinds of brances (e.g., branch-if-less-than)? For this, we need yet another instruction, `slt`
- Set on less than instruction:

```
slt $t0, $s0, $s1     # if $s0 < $s1     then
                      # $t0 = 1          else
                      # $t0 = 0
```

- Instruction format (R format):

# More Branch Instructions, Con't

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions
  - less than

```
blt $s1, $s2, Label
```

```
slt  $at, $s1, $s2      #$at set to 1 if
bne  $at, $zero, Label  #  $s1 < $s2
```

  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`

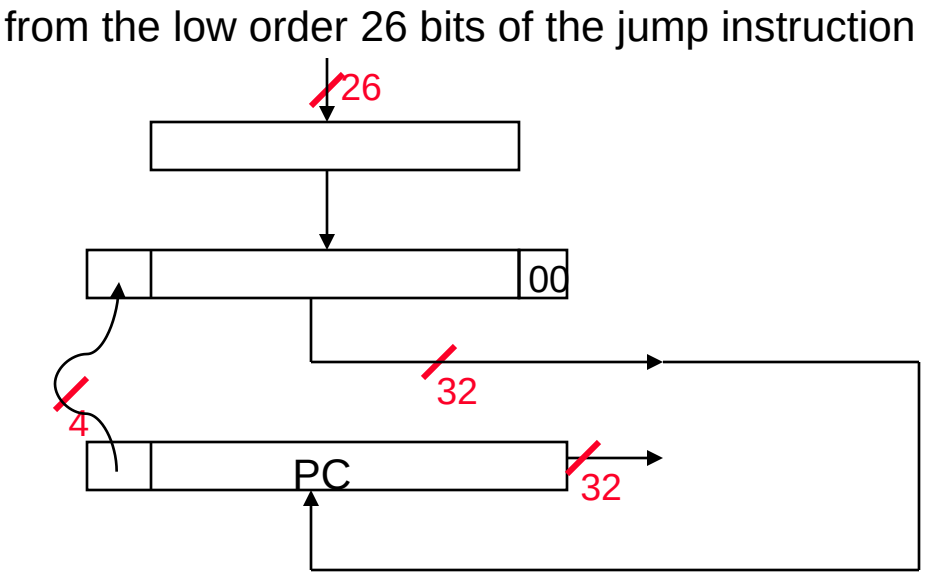
- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - Its why the assembler needs a reserved register (`$at`)

# Other Control Flow Instructions

- MIPS also has an unconditional branch instruction or jump instruction:

```
j   label        #go to label
```

- Instruction Format (J Format):

# Aside: Branching Far Away

- What if the branch destination is further away than can be captured in 16 bits?

- The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition

```
        beq   $s0, $s1, L1
```

becomes

```
        bne   $s0, $s1, L2
        j     L1
L2:
```

# Instructions for Accessing Procedures

- MIPS procedure call instruction:

```
jal   ProcedureAddress     #jump and link
```

- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return
- Machine format (J format):

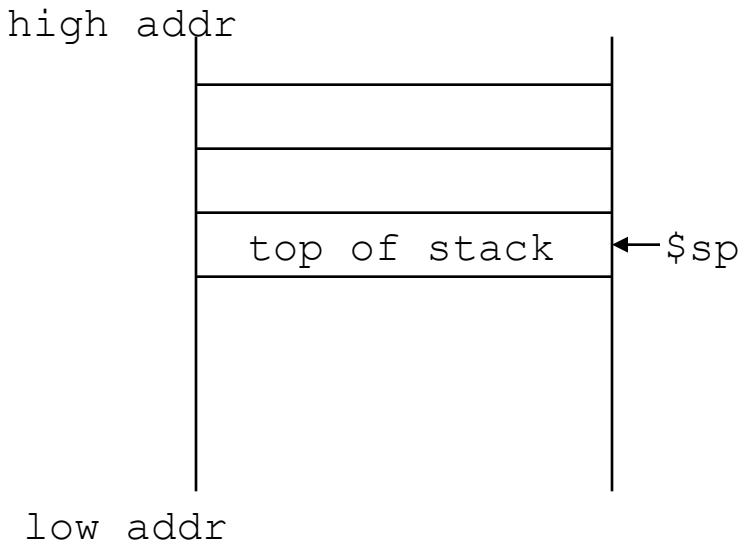

- Then can do procedure return with a

```
jr    $ra                  #return
```

- Instruction format (R format):

# Aside: Spilling Registers

- What if the callee needs more registers? What if the procedure is recursive?
  - uses a stack – a last-in-first-out queue – in memory for passing additional values or saving (recursive) return address(es)




- One of the general registers, $sp, is used to address the stack (which "grows" from high address to low address)
  - add data onto the stack – push

    $sp = $sp – 4
    data on stack at new $sp
  - remove data from the stack – pop

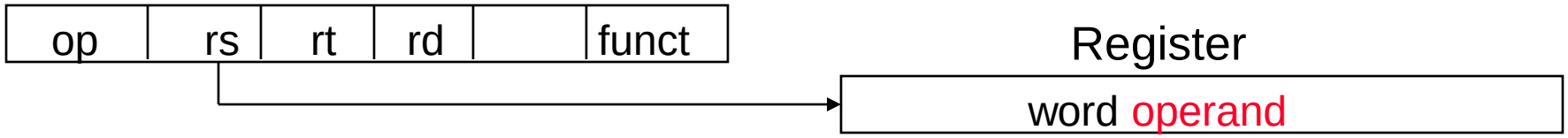
    data from stack at $sp
    $sp = $sp + 4

# MIPS ISA So Far

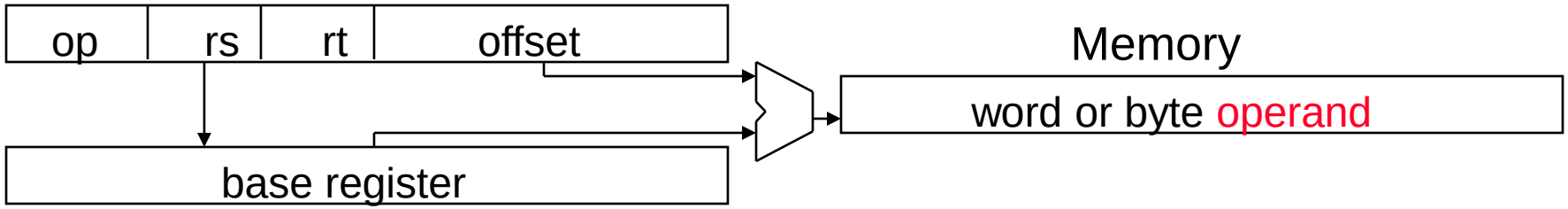
| Category | Instr | Op Code | Example | Meaning |
|---|---|---|---|---|
| Arithmetic (R & I format) | add | 0 and 32 | add $s1, $s2, $s3 | $s1 = $s2 + $s3 |
| | subtract | 0 and 34 | sub $s1, $s2, $s3 | $s1 = $s2 - $s3 |
| | add immediate | 8 | addi $s1, $s2, 6 | $s1 = $s2 + 6 |
| | or immediate | 13 | ori $s1, $s2, 6 | $s1 = $s2 v 6 |
| Data Transfer (I format) | load word | 35 | lw $s1, 24($s2) | $s1 = Memory($s2+24) |
| | store word | 43 | sw $s1, 24($s2) | Memory($s2+24) = $s1 |
| | load byte | 32 | lb $s1, 25($s2) | $s1 = Memory($s2+25) |
| | store byte | 40 | sb $s1, 25($s2) | Memory($s2+25) = $s1 |
| | load upper imm | 15 | lui $s1, 6 | $s1 = 6 * $2^{16}$ |
| Cond. Branch (I & R format) | br on equal | 4 | beq $s1, $s2, L | if ($s1==$s2) go to L |
| | br on not equal | 5 | bne $s1, $s2, L | if ($s1 !=$s2) go to L |
| | set on less than | 0 and 42 | slt $s1, $s2, $s3 | if ($s2<$s3) $s1=1 else $s1=0 |
| | set on less than immediate | 10 | slti $s1, $s2, 6 | if ($s2<6) $s1=1 else $s1=0 |
| Uncond. Jump (J & R format) | jump | 2 | j 2500 | go to 10000 |
| | jump register | 0 and 8 | jr $t1 | go to $t1 |
| | jump and link | 3 | jal 2500 | go to 10000; $ra=PC+4 |

# Review of MIPS Operand Addressing Modes

- Register addressing – operand is in a register

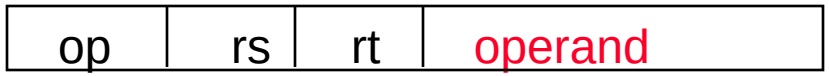



- Base (displacement) addressing – operand is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction

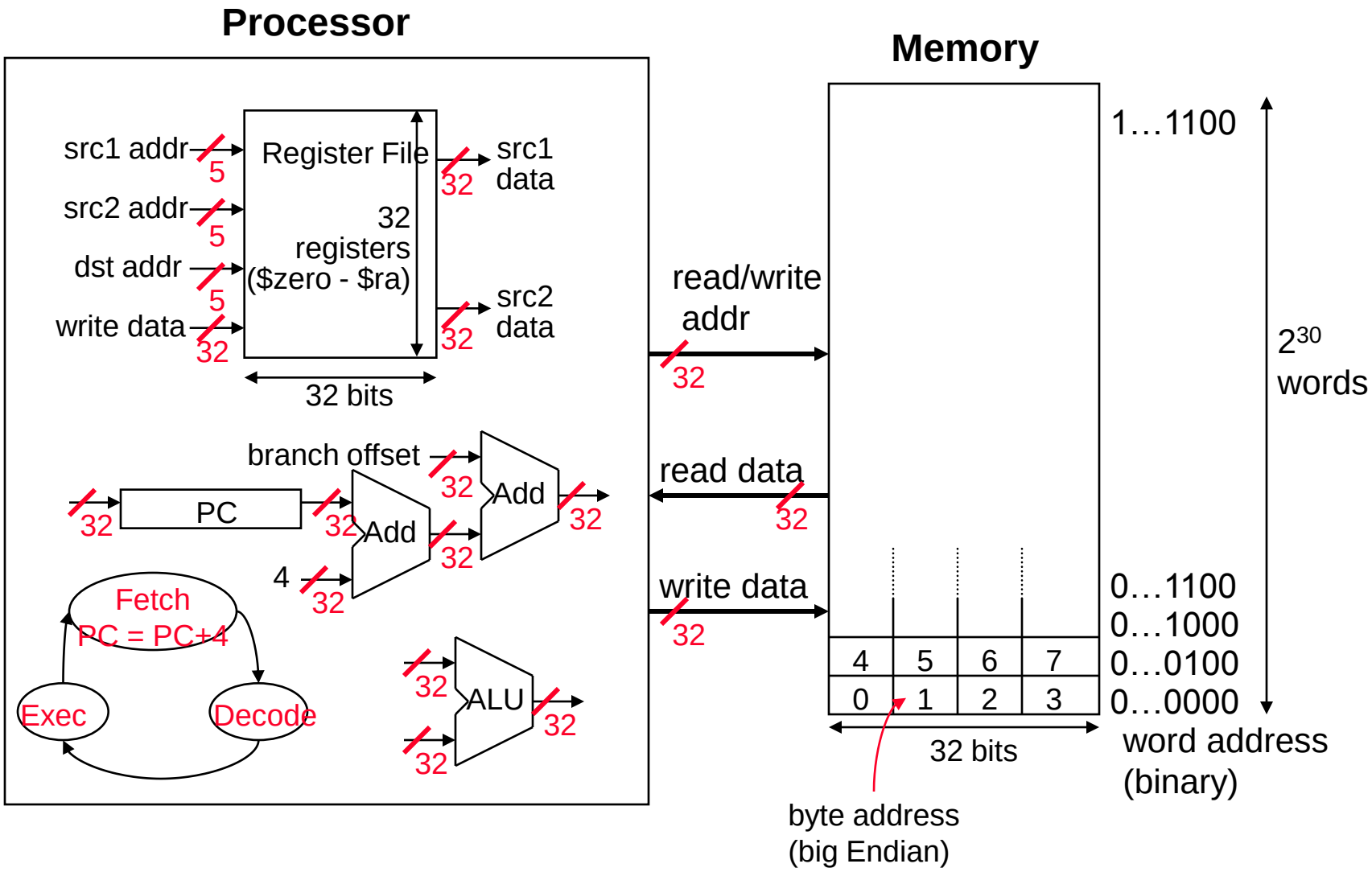



  - Register relative (indirect) with 0($a0)
  - Pseudo-direct with addr($zero)

- Immediate addressing – operand is a 16-bit constant contained within the instruction

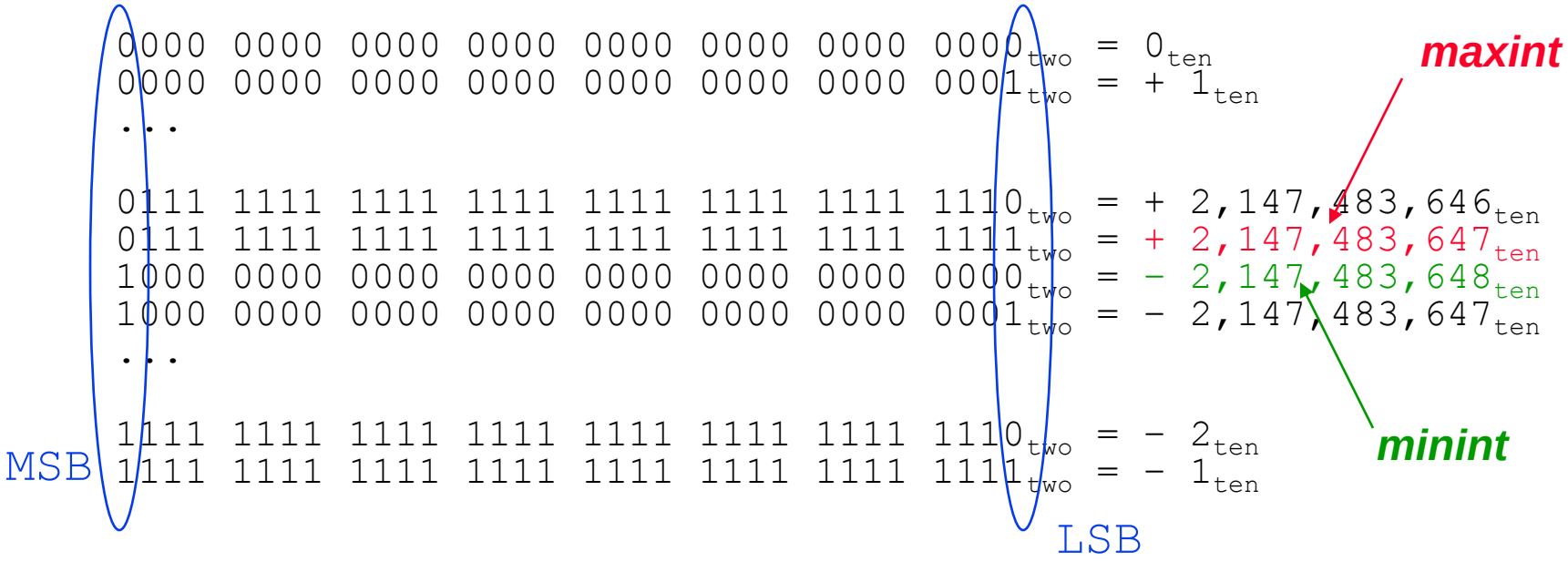


# Review: MIPS Organization



# MIPS Number Representations

- 32-bit signed numbers (2's complement):




- Converting <32-bit values into 32-bit values
  - copy the most significant bit (the sign bit) into the "empty" bits

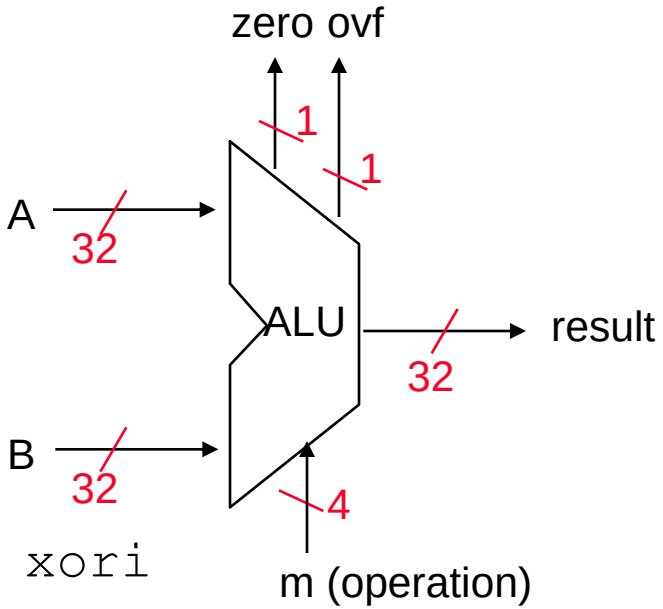
```
0010  -> 0000 0010
1010  -> 1111 1010
```

  - sign extend versus zero extend (`lb` vs. `lbu`)

# MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

```
add, addi, addiu, addu
sub, subu, neg
mult, multu, div, divu
sqrt
and, andi, nor, or, ori, xor, xori
beq, bne, slt, slti, sltiu, sltu
```




- With special handling for
  - sign extend – `addi, addiu andi, ori, xori, slti, sltiu`
  - zero extend – `lbu, addiu, sltiu`
  - no overflow detected – `addu, addiu, subu, multu, divu, sltiu, sltu`

# Review: 2's Complement Binary Representation

- Negate

1011

and add a 1

1010

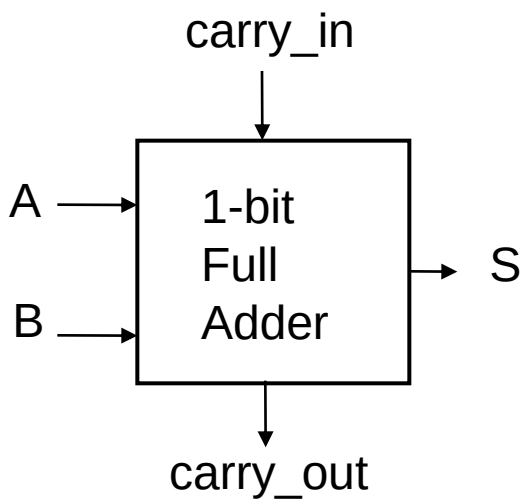
complement all the bits

Bits are inverted whereas numbers are negated!

- Note: negate and invert are different!

| | 2'sc binary | decimal |
|---|---|---|
| $-2^3$ = | 1000 | -8 |
| $-(2^3 - 1)$ = | 1001 | -7 |
| | 1010 | -6 |
| | 1011 | -5 |
| | 1100 | -4 |
| | 1101 | -3 |
| | 1110 | -2 |
| | 1111 | -1 |
| | 0000 | 0 |
| | 0001 | 1 |
| | 0010 | 2 |
| | 0011 | 3 |
| | 0100 | 4 |
| | 0101 | 5 |
| | 0110 | 6 |
| $2^3 - 1$ = | 0111 | 7 |

# Review: A Full Adder




| A | B | carry_in | carry_out | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

$S = A \oplus B \oplus carry\_in$ (odd parity function)

carry_out = A&B | A&carry_in | B&carry_in
(majority function)

- How can we use it to build a 32-bit adder?
- How can we modify it easily to build an adder/subtractor?

# Tailoring the ALU to the MIPS ISA

- Need to support the logic operation (`and,nor,or,xor`)
  - Bit wise operations (no carry operation involved)
  - Need a logic gate for each function, mux to choose the output
- Need to support the set-on-less-than instruction (`slt`)
  - Use subtraction to determine if (a – b) < 0 (implies a < b)
  - Copy the sign bit into the low order bit of the result, set remaining result bits to 0
- Need to support test for equality (`bne, beq`)
  - Again use subtraction: (a - b) = 0 implies a = b
  - Additional logic to “nor” all result bits together
- Immediates are sign extended outside the ALU with wiring (i.e., no logic needed)

# Shift Operations

- Also need operations to pack and unpack 8-bit characters into 32-bit words
- Shifts move all the bits in a word left or right

```
sll   $t2, $s0, 8    #$t2 = $s0 << 8 bits
srl   $t2, $s0, 8    #$t2 = $s0 >> 8 bits
```


- Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or 31 bit positions
- Such shifts are logical because they fill with zeros

# Shift Operations, con't

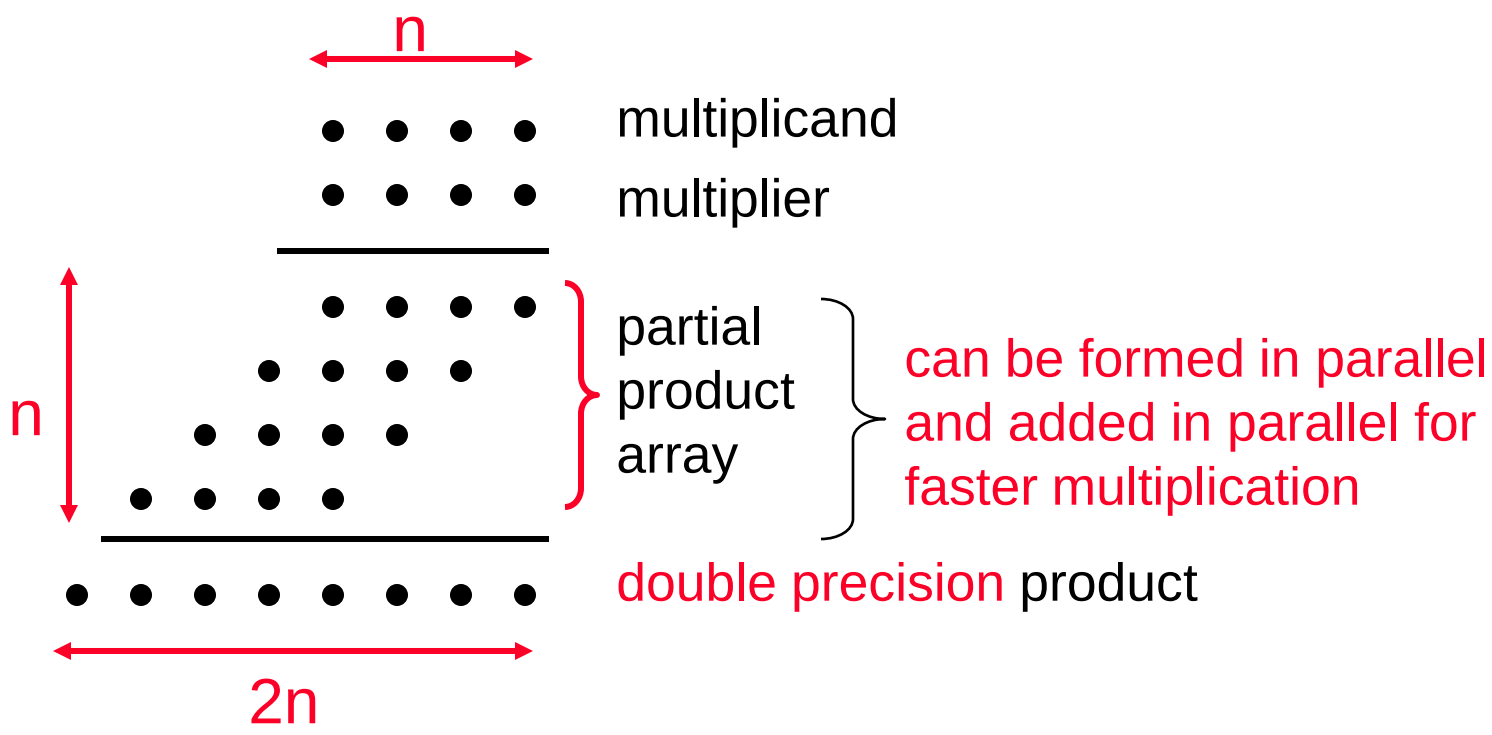
- An arithmetic shift (`sra`) maintain the arithmetic correctness of the shifted value (i.e., a number shifted right one bit should be ½ of its original value; a number shifted left should be 2 times its original value)
  - so `sra` uses the most significant bit (sign bit) as the bit shifted in
  - note that there is no need for a `sla` when using two's complement number representation

```
sra    $t2, $s0, 8    #$t2 = $s0 >> 8 bits
```

- The shift operation is implemented by hardware separate from the ALU
  - using a barrel shifter (which would takes lots of gates in discrete logic, but is pretty easy to implement in VLSI)

# Multiply

- Binary multiplication is just a *bunch* of right shifts and adds



# MIPS Multiply Instruction

- Multiply produces a double precision product

```
mult   $s0, $s1        # hi||lo = $s0 * $s1
```


  - Low-order word of the product is left in processor register `lo` and the high-order word is left in register `hi`
  - Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file

- Multiplies are done by fast, dedicated hardware and are much more complex (and slower) than adders
- Hardware dividers are even *more* complex and even slower; ditto for hardware square root

# MIPS Divide Instruction

- Divide generates the reminder in `hi` and the quotient in `lo`

```
div    $s0, $s1        # lo = $s0 / $s1

                       # hi = $s0 mod $s1
```


  - Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and reminder to (user accessible) registers in the register file

- As with multiply, divide ignores overflow so software must determine if the quotient is too large. Software must also check the divisor to avoid division by 0.

# Representing Big (and Small) Numbers

- What if we want to encode the approx. age of the earth?

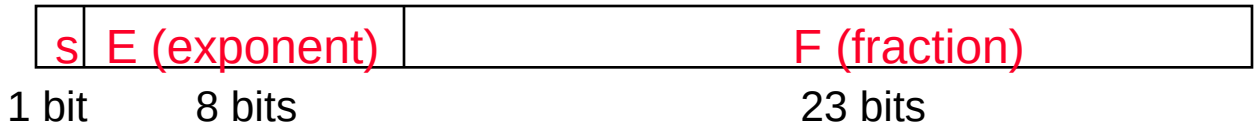
  4,600,000,000 or $4.6 \times 10^9$

  or the weight in kg of one a.m.u. (atomic mass unit)

  0.000000000000000000000000166 or $1.6 \times 10^{-27}$

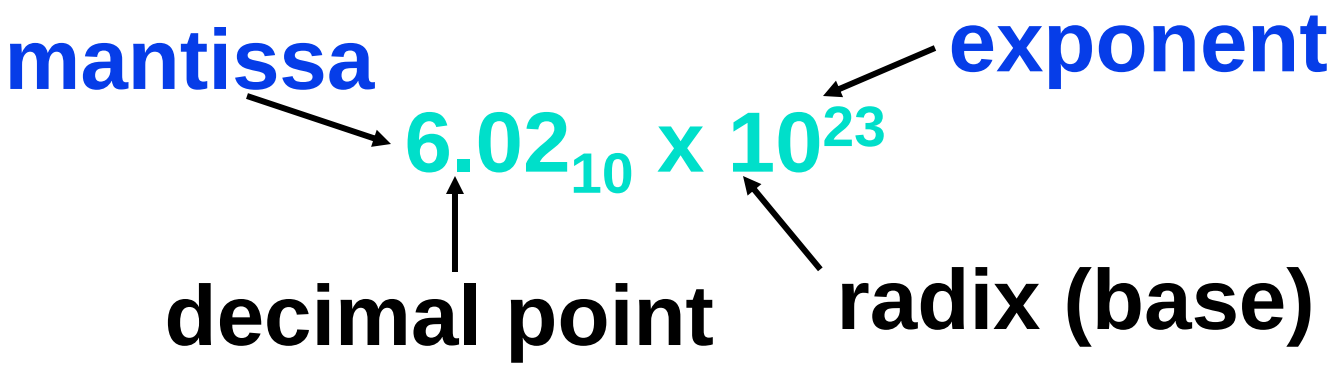
  There is no way we can encode either of the above in a 32-bit integer.

- Floating point representation $(-1)^{sign} \times F \times 2^E$
  - Still have to fit everything in 32 bits (single precision)



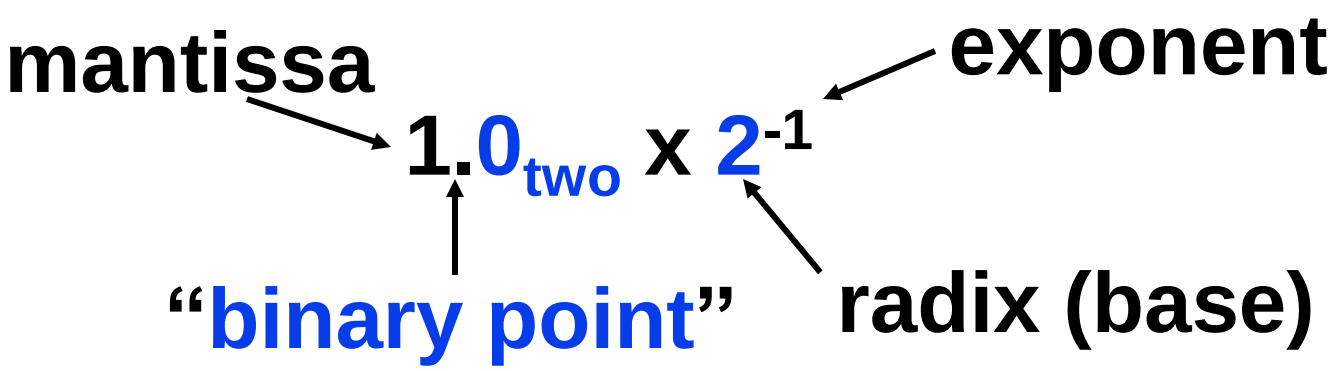

  - The base (2, *not* 10) is hardwired in the design of the FPALU
  - More bits in the fraction (F) or the exponent (E) is a trade-off between precision (accuracy of the number) and range (size of the number)

# Scientific Notation (in Decimal)




- Normalized form: no leadings 0s (exactly one digit to left of decimal point)
- Alternatives to representing 1/1,000,000,000
  - Normalized: $1.0 \times 10^{-9}$
  - Not normalized: $0.1 \times 10^{-8}$, $10.0 \times 10^{-10}$

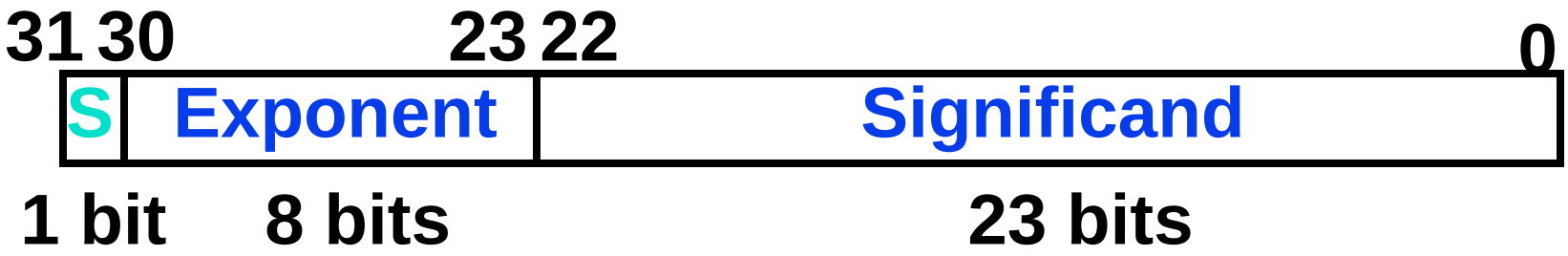
# Scientific Notation (in Binary)




- Computer arithmetic that supports it called floating point, because it represents numbers where the binary point is not fixed, as it is for integers
  - Declare such variable in C as `float`

# Floating Point Representation

- Normal format: $+1.xxxxxxxxxx_{two} * 2^{yyyy_{two}}$
- Multiple of Word Size (32 bits)




- S represents Sign
- Exponent represents y's
- Significand represents x's
- Represent numbers ranging from $2^{-126}(1.0)$ to $2^{+127}(2-2^{-23})$ ie. from $1.18 \times 10^{-38}$ to $3.40 \times 10^{38}$

# Next Lecture and Reminders

- Next lecture
  - Addressing and understanding performance