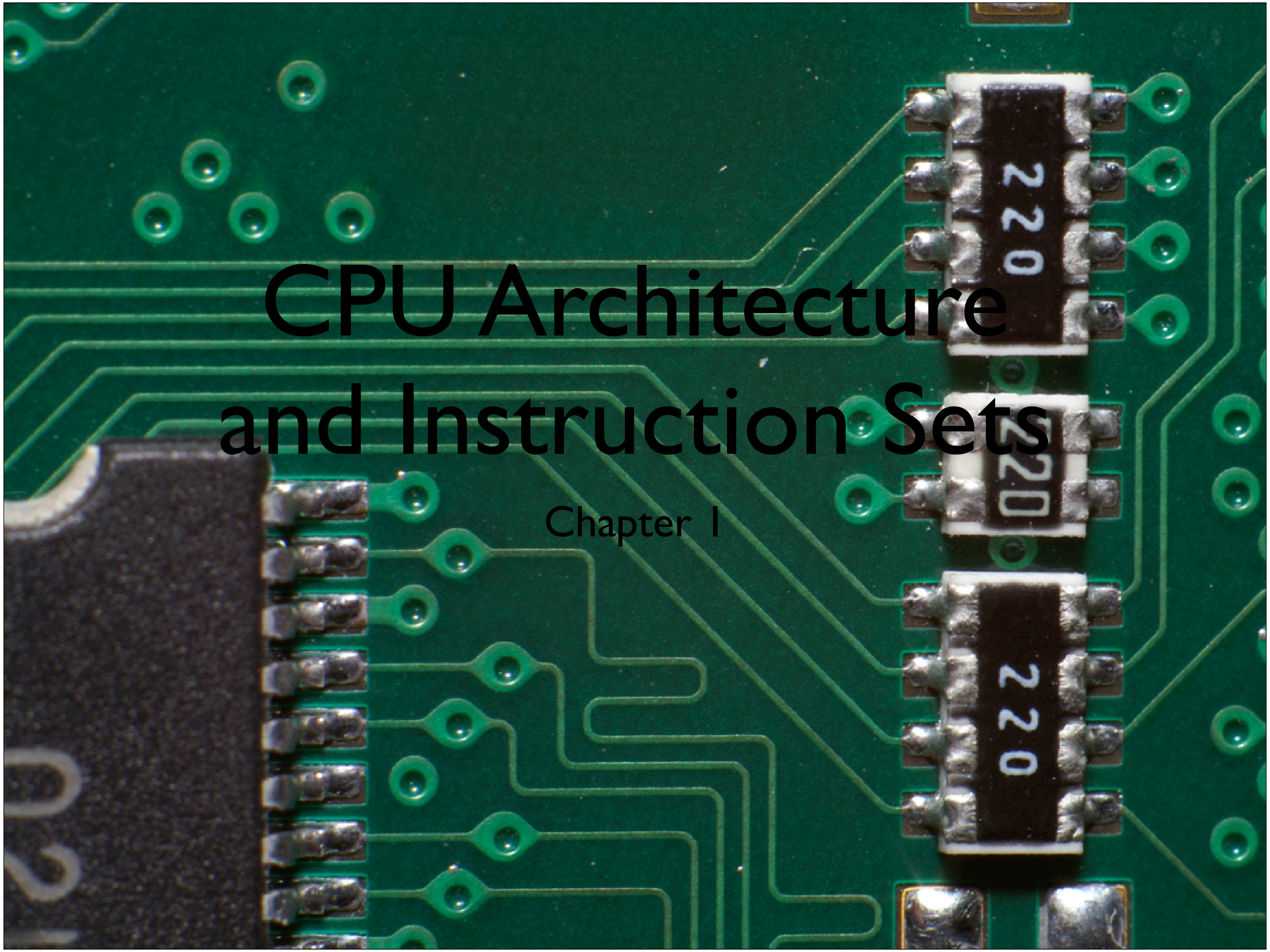


CPU Architecture and Instruction Sets

Chapter 1

Is CPU Architecture Relevant for DBMS?

- CPU design focuses on speed — resulting in a 55%/year improvement since 1987:

  "*If CPU performance in database code really is disappointing, upgrade the database server to the next processor generation.*"

- With the advent of modern multi-core CPUs, all odds are that this trend will continue for the foreseeable future.

Amdahl's Law

- CPU speed is only one of many aspects of overall system performance.
- Amdahl's law describes the impact of the speedup of a single component (*e.g.*, the CPU) of a complex system.
  - Since the rest of the system remains as is, the return to be expected from the speedup is diminished.

Amdahl's Law

- $\text{Speedup}_{\text{enhanced}} \geqslant 1$:

  Performance of the enhanced component in comparison with the replaced, original component.

- $\text{Fraction}_{\text{enhanced}} \leqslant 1$:

  Fraction of computation time that actually can take advantage of the enhanced component.

Amdahl's Law

$$\text{Exec. time}_{\text{new}}$$
$$= \text{Exec. time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

- The execution time after the enhancement will be
  1. the time spent using the unenhanced portion of the system, plus
  2. the time spent using the enhancement.

Amdahl's Law

- Example:

  Perform a database server upgrade and plug in a new CPU that is 10 times faster. The original system is busy with computation 40% of the time, and is waiting for memory accesses 60% of the time (this seems reasonable in database code, *i.e.*, for a data-intensive application). What is the overall speedup gained?

CPU Time

- It is vital to understand which factors contribute to the CPU time — the overall time the CPU requires to execute a given program:

$$\begin{aligned}\text{CPU time} = {} & \text{Instruction count} && \times \\ & \text{Clock cycle time} && \times \\ & \text{Cycles per Instruction} &&\end{aligned}$$

- Note:
  CPU time is equally dependent on all three factors.

CPU Time

- Improving the CPU time factors calls for action on various levels.

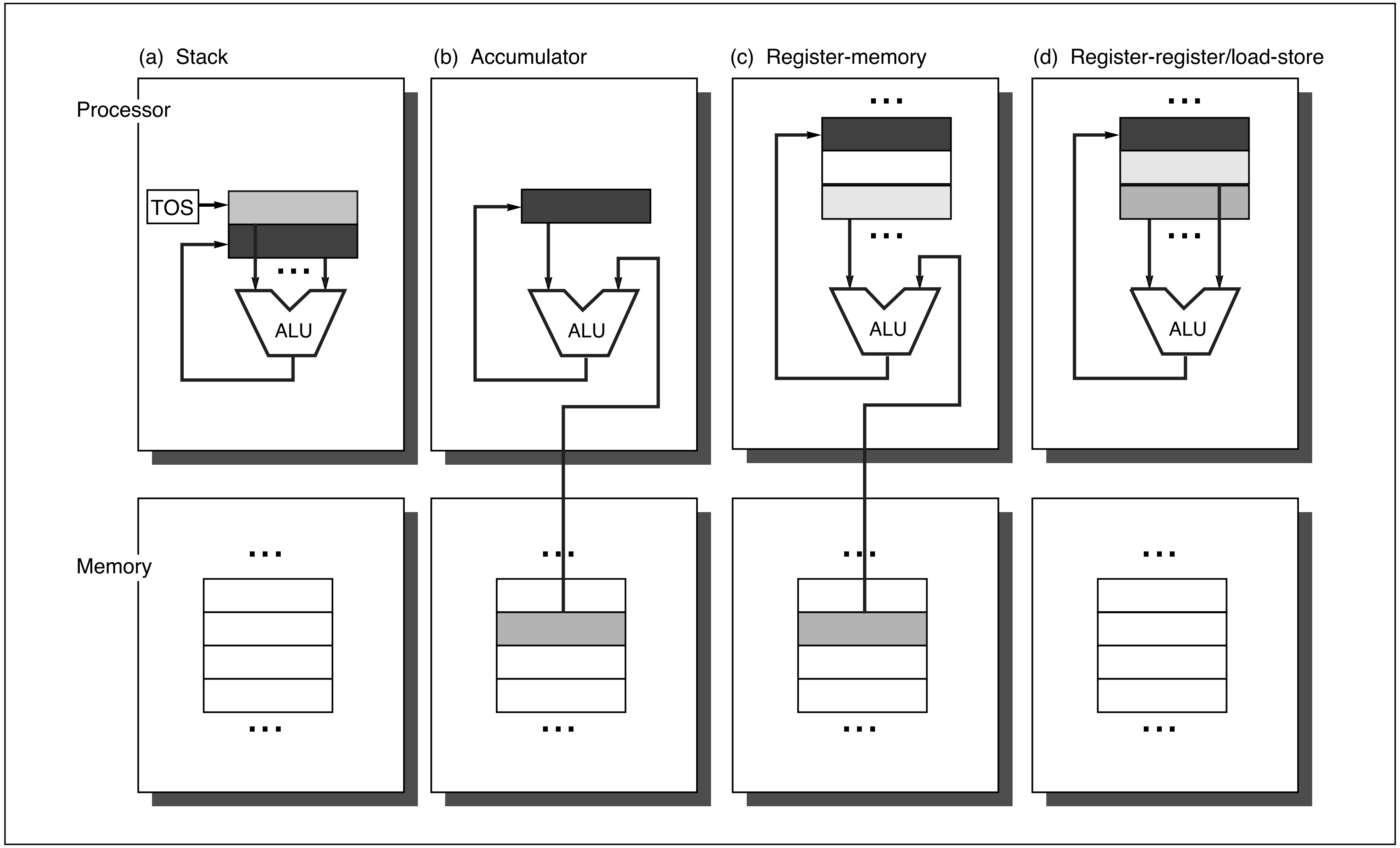
  1. Clock cycle time:
     Hardware technology
     (faster components & signal transfer)
  2. Clocks per instructions (CPI):
     Instruction set and execution (parallelism)
  3. Instruction count:
     Instruction set and compiler technology

Instruction Set Architectures

- We will now investigate CPU instructions sets to understand why they look as they do today.
- The type of interal storage in the CPU is the most basic differentiation among instruction set architectures:

Where does an instruction find its operands?

Instruction Set Architectures



Implementing `C=A+B`

Stack	Accumulator	Register-memory	Load-store

`Push A`
`Push B`
`Add`
`Pop  C`
	`Load  A`
	`Add   B`
	`Store C`
		`Load  R1,A`
		`Add   R3,R1,B`
		`Store R3,C`
			`Load  R1,A`
			`Load  R2,B`
			`Add   R3,R1,R2`
			`Store R3,C`

Intel 80x86 could be classified as an *extended accumulator* (or *special purpose register*) architecture.

Load-Store Architectures

- Most modern CPU instruction sets follow the load-store (register-register) architecture:
  - Register access is faster than memory access
  - Compiled code for general-purpose register machines tends to be more efficient.

    Consider the compilation of the arithmetic expression

```
(A*B) - (B*C) - (A*D)
```

Load-Store Architectures

- General-purpose register machines comes with further advantages:
  - When variables are allocated to registers, memory traffic reduces — programs speed up.
  - Code density improves — a register can be named with fewer bits than a memory location.
  - Fixed-length instruction encodings simplify CPU internals.
  - Instructions take similar numbers of clock cycles to execute — simplifies parallelization and scheduling.

Memory Addressing

- When the CPU accesses memory, two parameters determine what object is loaded into the CPU registers:
  1. Memory address, and
  2. Object size (measured in bytes, usually 1,2,4,8).
- Object size is typically encoded in the instruction itself (*e.g.*, MIPS load instructions: `LB`, `LH`, `LW` , `LD`)

Byte Ordering

- Byte ordering determines the layout of a multi-byte object (size ≥ 1) in memory.

  Layouts of a 32-bit value `0x12345678` at address `0x100`):

  - Big Endian:
    (e.g., Sparc)

`0x12`	`0x100`
`0x34`	
`0x56`	
`0x78`	`0x103`

  - Little Endian:
    (e.g., Intel)

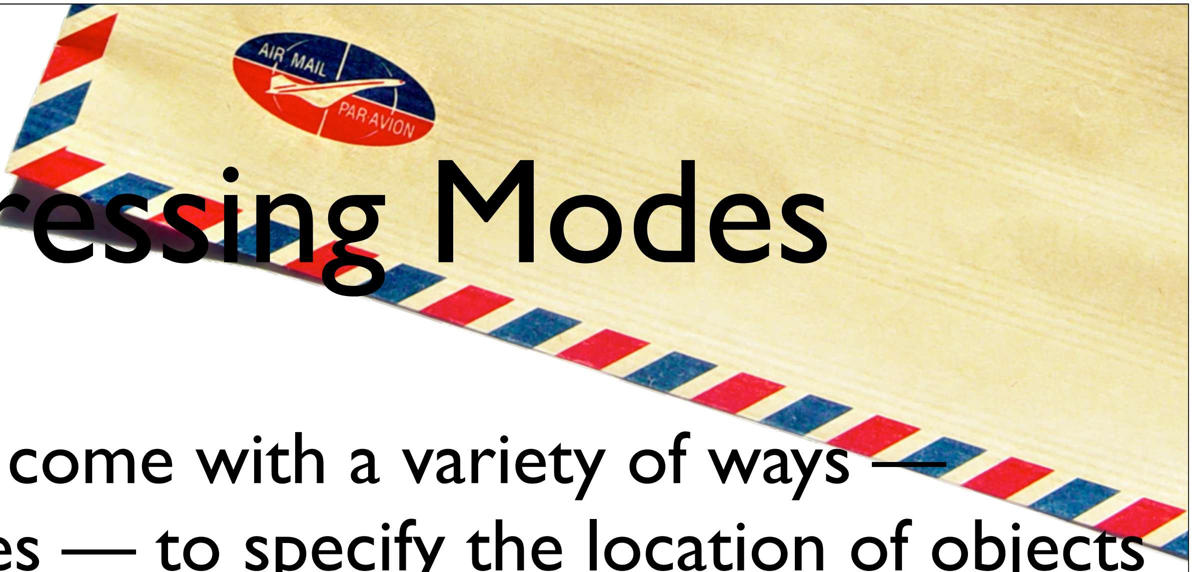
`0x78`	`0x100`
`0x56`	
`0x34`	
`0x12`	`0x103`

Alignment

- Most CPU architectures require aligned memory accesses for all objects of size ≥ 1. Alignment makes memory hardware more simple.

  Access to an object of size *s* at address *A* is aligned, if *A mod s = 0.*

- A misaligned memory access may
  - lead to a CPU exception (e.g., Motorola 68K), or
  - lead to *two* aligned accesses, plus bit shifting (e.g., Intel 80x86).



Addressing Modes

- Instruction sets come with a variety of ways — addressing modes — to specify the location of objects in memory.
- Addressing modes reflect the different methods of how memory is accessed in higher-level programming languages, for example via
  - array indexing, or `a[i]`
  - pointer dereferencing. `*p`

Addressing Modes

Addressing Mode	Sample Instruction	Semantics
Register	`Add R4,R3`	`Regs[R4]← Regs[R4]+Regs[R3]`
Immediate	`Add R4,#3`	`Regs[R4]← Regs[R4]+3`
Displacement	`Add R4,100(R1)`	`Regs[R4]← Regs[R4] +Mem[100+Regs[R1]]`

Addressing Modes

Addressing Mode	Sample Instruction	Semantics
Register indirect	`Add R4,(R1)`	`Regs[R4]←` `Regs[R4]+Mem[Regs[R1]]`
Indexed	`Add R3,(R1+R2)`	`Regs[R3]←` `Regs[R3]` `+Mem[Regs[R1]+Regs[R2]]`
Direct (absolute)	`Add R1,(1001)`	`Regs[R1]←` `Regs[R1]+Mem[1001]`
Memory indirect	`Add R1,@(R3)`	`Regs[R1]←` `Regs[R1]` `+Mem[Mem[Regs[R3]]]`

Addressing Modes

Addressing Mode	Sample Instruction	Semantics

Autoincrement (postincrement)	`Add R1,(R2)+`	`Regs[R1]←`
		`  Regs[R1]+Mem[Regs[R2]]`
		`Regs[R2]←Regs[R2]+d`

Autodecrement (predecrement)	`Add R1,-(R2)`	`Regs[R2]←Regs[R2]-d`
		`Regs[R1]←`
		` +Regs[R1]+Mem[Regs[R2]]`

Scaled	`Add R1,`
	`  100(R2)[R3]`
		`Regs[R1]←Regs[R1]`
		` +Mem[100+Regs[R2]+`
		`      Regs[R3]*d]`

Addressing Modes

- Addressing modes may significantly reduce instruction counts. Consider:

```
LD R1,100(R2)[R3]
```

vs.

```
LW    R4,#8
MULTU R4,R4,R3
LW    R5,#100
ADD   R4,R4,R5
ADD   R4,R4,R2
LD    R1,(R4)
```

- Complex addressing modes may increase CPI (clock cycles per instruction), though.

Operations

Operator type	Examples
Arithmetic, logical	Integer arithmetic and logical operations: add, subtract, multiply, divide, and, or
Data transfer	Loads, stores
Control	Branch, jump, procedure call/return, traps
System	Operating system call, virtual memory mgmt
Floating point	FP operations: add, multiply, divide, compare
String	String move, compare, search

Operation Distribution

Typical operation distribution for SPECint92 programs:

Rank	80x86 Instruction	% total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	subtract	5%
8	move register-register	4%
9	call	1%
10	return	1%

Branch Instructions

- Branch instructions typically specify the branch destination address using PC-relative addressing:

$$PC_{new} \leftarrow PC_{current} + offset \ (\times \ instruction\text{-}length)$$

  - Branch targets near $PC_{current}$ can be specified using few bits (usually ≤ 8 bits).
  - PC-relative addressing makes code position independent — saves linker effort.

(Statically) Unknown Branch Targets

- Jumps to target addresses not known at compile time make branch prediction even more challenging.

  1. Multi-way branches: `case` or `switch` statements
  2. Virtual functions or methods (in OOPLs)
  3. Higher-order functions, function pointers (e.g., in C)
  4. Dynamically loaded shared libraries

Encoding Instructions

- CPU instructions are encoded via a bit pattern that specifies
  1. operation type, and
  2. addressing mode and operand addresses.
- This encoding has a siginificant impact on
  - the CPU-internal instruction decoder, and
  - the size of compiled programs.

Encoding Instructions

- Variable-length instructions encodings can help to reduce code size but are complex to decode.
  - Example: Intel 80x86 instructions occupy 1...17 bytes (*e.g.*, `add EAX,1000(EBX)` uses 6 bytes).
- Fixed-length instructions allow for less addressing modes but are more efficient to decode.
  - If addresses (registers) are encoded at fixed bit positions, the CPU can decode and access registers in parallel.

MIPS Instruction Encoding

31			0
110111	base	rt	offset
6 bits	5	5	16

Encodes the `LD` instruction (addr. mode: *displacement*)

```
LD rt,offset(base)     ; Regs[rt]←
                          Mem[offset+Regs[base]]
```

Note: This also implements addressing modes *register indirect* and *direct (absolute)*.

RISC: Reduced Instruction Set Computers

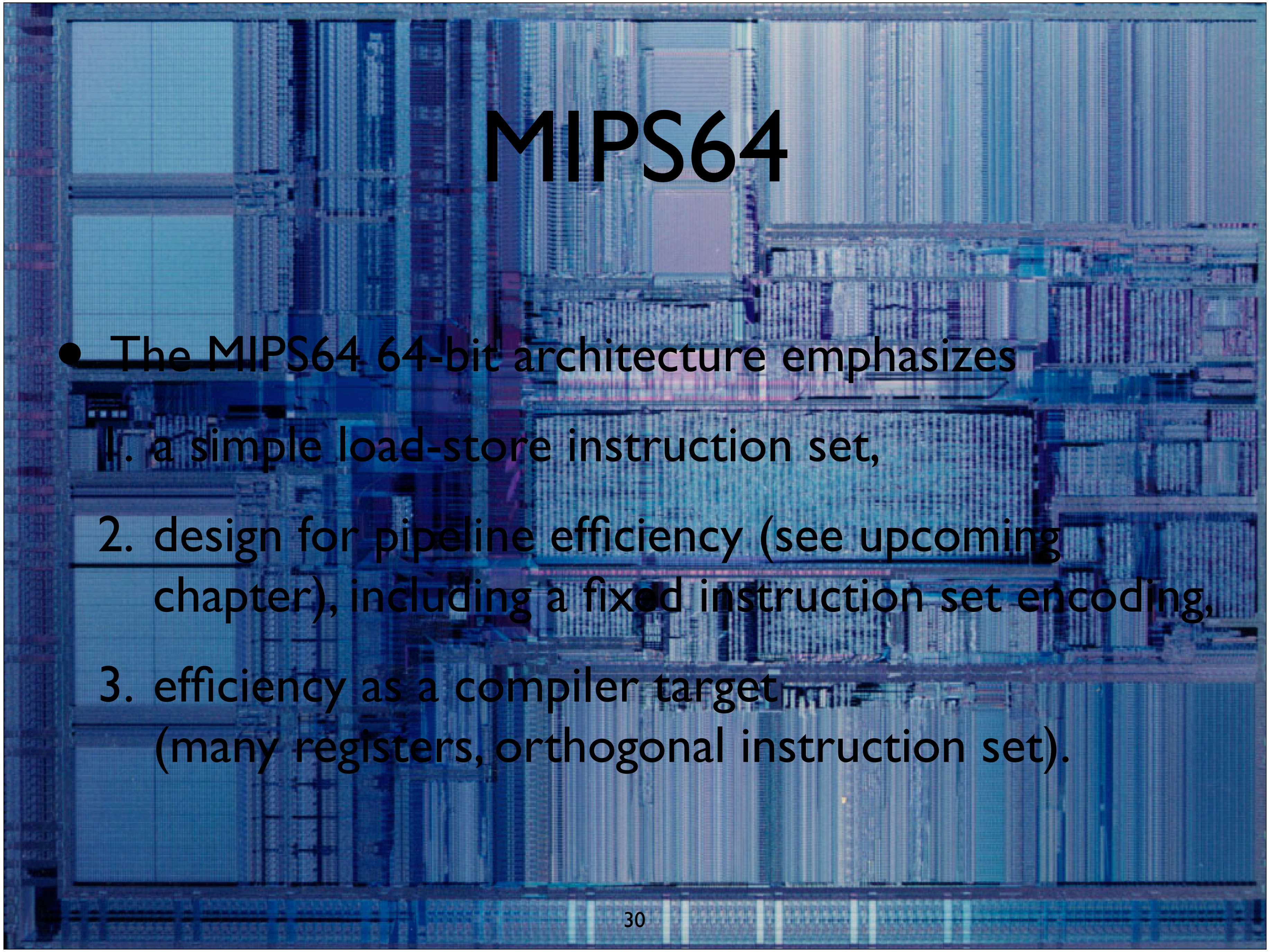
- RISC architectures, like MIPS, offer a comparatively small number of (primitive) instructions but implement these efficiently.
- Typically narrow, fixed-length encoding and orthogonal instruction set.

+ Pseudo instructions (expanded on assembly language level, uses “reserved” registers). Consider:

```
LW R4,0x12345678(R0)
```

$\rightarrow$

```
LUI R1,0x1234
LW  R4,0x5678(R1)
```



MIPS64

- The MIPS64 64-bit architecture emphasizes
  1. a simple load-store instruction set,
  2. design for pipeline efficiency (see upcoming chapter), including a fixed instruction set encoding,
  3. efficiency as a compiler target (many registers, orthogonal instruction set).

MIPS64

- Registers:
  - 32 64-bit general-purpose registers (GPRs):
    `R0`, ..., `R31` (`R0` ≡ `0`)
  - 32 64-bit floating-point registers (FPRs):
    `F0`, ..., `F31` (IEEE 754 format)
- Data types (bit width):
  - Byte (8), half word (16), word (32), double word (64)
  - 64-bit GPRs padded with 0 or sign bit

MIPS64

- Load-store architecture
- Addressing modes:
  - *Immediate* (16 bits): `ADD R4,R4,#`*<16 bit>*
  - *Displacement* (16 bits): `ADD R4,R4,`*<16 bit>*`(R1)`
  - *Register indirect*, *absolute* available via `R0`
  - All memory accesses must be aligned

Operations: Notation

$t \leftarrow_n s$	Transfer *n* bits from *s* to *t*
`Regs[R1]`$_{n..m}$	Selection of bits *n..m* of register `R1` (bit 0 is most significant)
`Mem[`a`]`	Address *a* of byte-organized main memory array, can transfer any number of bytes
x^n	Value *x*, replicated *n* times
x ## y	Concatenate *x* and *y* (may appear left and right of $\leftarrow$)

Operations: Notation

- Example (move byte at address `(R8)` into lower 32-bit half of `R10` with sign extension):

$$\texttt{Regs[R10]}_{32..63} \leftarrow_{32} (\texttt{Mem[Regs[R8]]}_{0})^{24} \ \texttt{\#\#}\ \texttt{Mem[Regs[R8]]}$$

MIPS64: Load-store instructions

`LD R1,30(R2)`	**Load double word** $\texttt{Regs[R1]} \leftarrow_{64} \texttt{Mem[30+Regs[R2]]}$
`LW R1,1000(R0)`	**Load word** $\texttt{Regs[R1]} \leftarrow_{64} (\texttt{Mem[1000+0]}_0)^{32}$ $\texttt{\#\# Mem[1000+0]}$
`LH R1,60(R2)`	**Load half word** $\texttt{Regs[R1]} \leftarrow_{64} (\texttt{Mem[60+Regs[R2]]}_0)^{48}$ $\texttt{\#\# Mem[60+Regs[R2]]}$
`LBU R1,40(R3)`	**Load byte unsigned** $\texttt{Regs[R1]} \leftarrow_{64} 0^{56}$ $\texttt{\#\# Mem[40+Regs[R3]]}$

MIPS64: Load-store instructions

`SD R3,500(R4)`	**Store double word** `Mem[500+Regs[R4]]` $\leftarrow_{64}$ `Regs[R3]]`
`SW R3,500(R4)`	**Store word** `Mem[500+Regs[R4]]` $\leftarrow_{32}$ `Regs[R3]]`$_{32..63}$
`SH R3,502(R2)`	**Store half word** `Mem[502+Regs[R2]]` $\leftarrow_{16}$ `Regs[R3]]`$_{48..63}$
`SB R2,41(R3)`	**Store byte** `Mem[41+Regs[R3]]` $\leftarrow_{8}$ `Regs[R2]]`$_{56..63}$

MIPS64: Arithmetic/Logical Instructions

`DADDU R1,R2,R3`	**Add unsigned** `Regs[R1] ← Regs[R2] + Regs[R3]`
`DADDIU R1,R2,#3`	**Add immediate unsigned** `Regs[R1] ← Regs[R2] + 3`
`LUI R1,#42`	**Load upper immediate** `Regs[R1] ←` 0^{32} `## 42 ##` 0^{16}
`DSLL R1,R2,#5`	**Shift left logical** `Regs[R1] ← Regs[R2] << 5`

`SLT R1,R2,R3`	**Set less than**
	`if (Regs[R2] < Regs[R3])`
	`  Regs[R1] ← 1 else Regs[R1] ← 0`

MIPS64: Jump Instructions

`J label`	**Jump** $\mathtt{PC}_{36..63} \leftarrow_{28}$ `label` `label` $\in$ $[\mathtt{PC+4-2^{27}},\ \mathtt{PC+4+2^{27}})$
`JAL label`	**Jump and link** `Regs[R31]` $\leftarrow$ `PC+4;` $\mathtt{PC}_{36..63} \leftarrow_{28}$ `label` `label` $\in$ $[\mathtt{PC+4-2^{27}},\ \mathtt{PC+4+2^{27}})$
`JALR R2`	**Jump and link register** `Regs[R31]` $\leftarrow$ `PC+4;` `PC` $\leftarrow$ `Regs[R2]`
`JR R3`	**Jump register** `PC` $\leftarrow$ `Regs[R3]`

MIPS64: Branch Instructions

`BEQZ R4,label`	**Branch equal zero** `if (Regs[R4] == 0)` $\mathtt{PC}_{46..63} \leftarrow_{18}$ `label` `label` $\in$ $[\mathtt{PC}+4-2^{17}, \mathtt{PC}+4+2^{17})$
`BNE R3,R4,label`	**Branch not equal** `if (Regs[R3]` $\neq$ `Regs[R4])` $\mathtt{PC}_{46..63} \leftarrow_{18}$ `label` `label` $\in$ $[\mathtt{PC}+4-2^{17}, \mathtt{PC}+4+2^{17})$

`MOVZ R1,R2,R3`	**Conditional move if zero**
	`if (Regs[R3] == 0)`
	  `Regs[R1]` $\leftarrow$ `Regs[R2]`