



**LUND**  
UNIVERSITY

# EITF20: Computer Architecture

## Part2.1.1: Instruction Set Architecture

Liang Liu  
liang.liu@eit.lth.se



# Outline

- Reiteration
- Instruction Set Principles
- The Role of Compilers
- MIPS



# Main Content

- Computer Architecture
- Performance
- Quantitative Principles



# What Computer Architecture?

## □ Design

- ISA
- Organization (microarchitecture)
- Implementation

## □ To meet requirements of

- Functionality (application, standards...)
- Price
- Performance
- Power
- Reliability
- Compatability
- Dependability
- ..



# Performance

## □ Time to complete a task ( $T_{exe}$ )

- Execution time, response time, latency

## □ Task per day, hour...

- Total amount of tasks for given time
- Throughput, bandwidth

Application	⇐	Answers/month
Programming language	⇐	Response time (seconds)
Compiler	⇐	Operations/second
Instruction set	⇐	MIPS/MFLOPS
Data-path control	⇐	Megabytes/second
Functional units		
Transistors, wires, pins	⇐	Cycles per second (clock rate)

MIPS = millions of instructions per second

MFLOPS = millions of FP operations per second



# Quantitative Principles

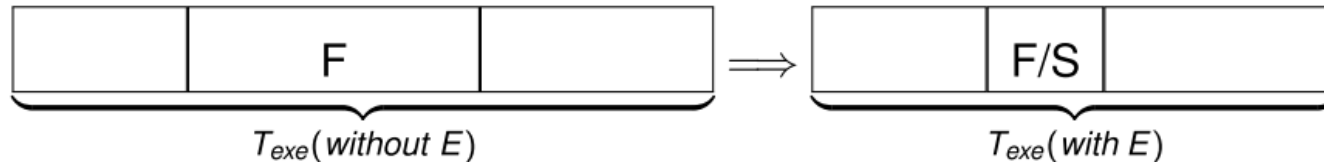
## □ This is intro to design and analysis

- Take advantage of parallelism
  - ILP, DLP, TLP, ...
- Principle of locality
  - 90% of execution time in only 10% of the code
- Focus on the common case
  - In making a design trade-off, favor the frequent case over the infrequent case
- Amdahl's law
  - The performance improvement gained from using faster mode is limited by the fraction of the time the faster mode can be used
- The processor performance equation



# Amdahl's Law

Enhancement E accelerates a fraction F of a program by a factor S



Speedup due to enhancement E:

$$\text{Speedup}(E) = \frac{T_{exe}(\text{without } E)}{T_{exe}(\text{with } E)} = \frac{\text{Performance}(\text{with } E)}{\text{Performance}(\text{without } E)}$$

$$T_{exe}(\text{with } E) = T_{exe}(\text{without } E) * [(1 - F) + F/S]$$

$$\text{Speedup}(E) = \frac{T_{exe}(\text{without } E)}{T_{exe}(\text{with } E)} = \frac{1}{(1-F)+F/S}$$

Best you could ever hope to do:

$$\text{Speedup}_{\text{maximum}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}})}$$



# Aspect of CPU performance

CPUtime = Execution time =  
*seconds/program* =

$$\underbrace{(\textit{executed})\textit{instr./program}}_{IC} * \underbrace{\textit{cycles/instr.}}_{CPI} * \underbrace{\textit{seconds/cycle}}_{T_c}$$

	IC	CPI	$T_c$
Program	X		
Compiler	X	(X)	
Instr. Set	X	X	
Organization		X	X
Technology			X





# Instructions are not created equally

“Average Cycles per Instruction”

$CPI_{op}$  = Cycles per Instruction of type  $op$

$IC_{op}$  = Number of executed instructions of type  $op$

$$CPUtime = T_c * \sum (CPI_{op} * IC_{op})$$

“Instruction frequency”

$$\overline{CPI} = \sum (CPI_{op} * F_{op}) \text{ where } F_{op} = IC_{op} / IC$$



# Average CPI: example

Op	$F_{op}$	$CPI_{op}$	$F_{op} * CPI_{op}$	% time
ALU	50 %	1	0.5	(33 %)
Load	20 %	2	0.4	(27 %)
Store	10 %	2	0.2	(13 %)
Branch	20 %	2	0.4	(27 %)

$$\overline{CPI} = 1.5$$

**Invest resources where time is spent!**



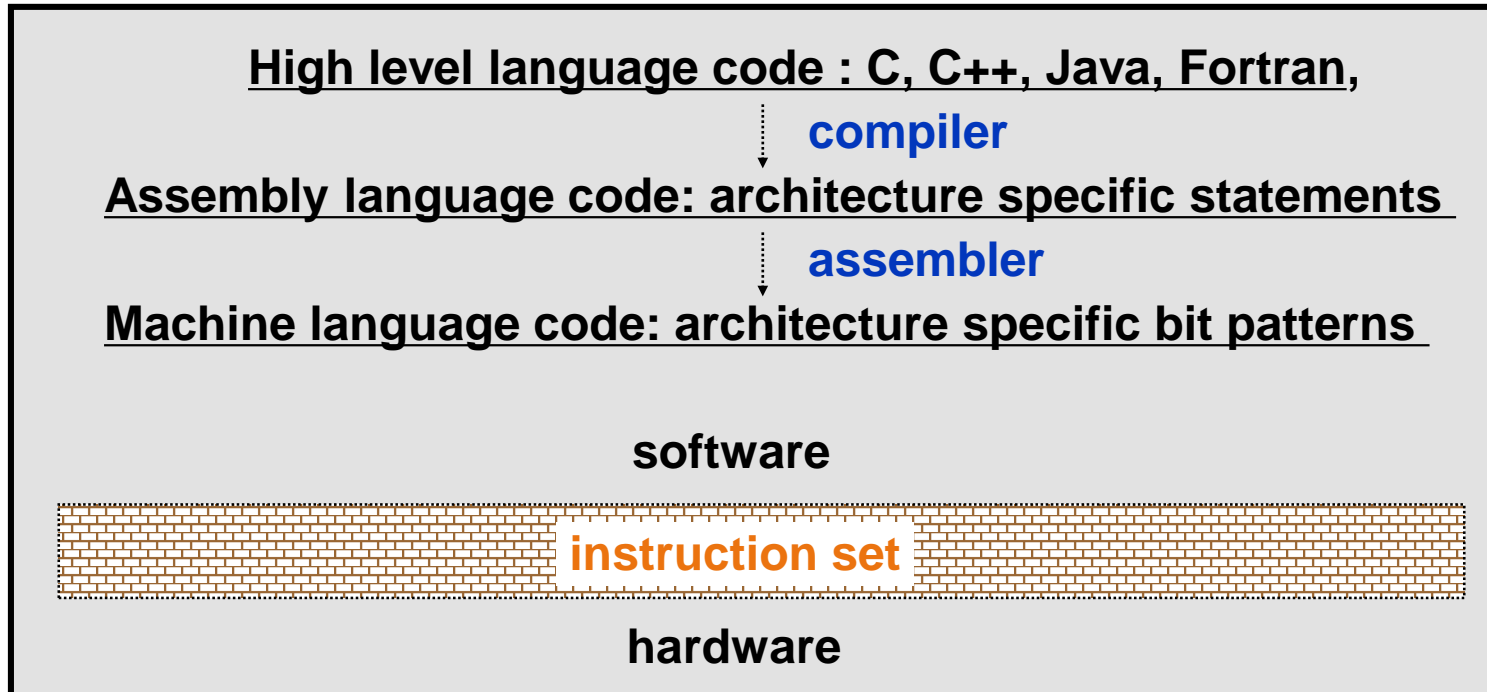
# Outline

- Reiteration
- **Instruction Set Principles**
- The Role of Compilers
- MIPS



# Instruction Set

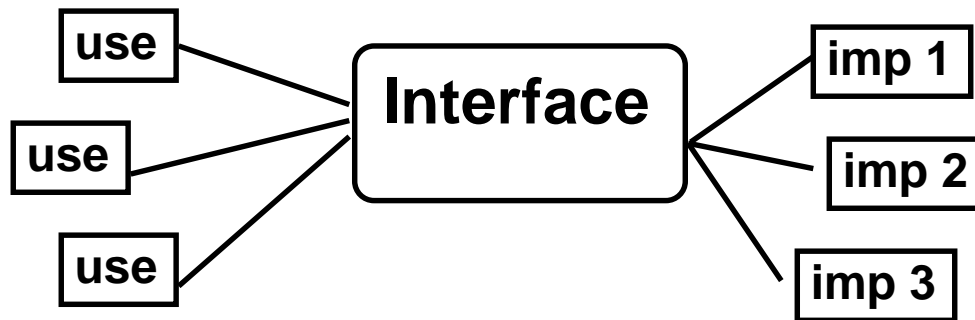
- ❑ Serves as an **interface** between software and hardware
- ❑ Provides a mechanism by which the software **tells the hardware what should be done**



# Interface Design

## □ A good interface

- Lasts through many implementations (portability, compatibility)
- Can be used in many different ways (generality)
- Provides sufficient functionality to higher levels
- Permits an efficient implementation at lower levels



# ISA Classification

## □ What's needed in an instruction set?

- Addressing
- Operands
- Operations
- Control Flow

## □ Classification of instruction sets

- Register model
- The number of operands for instructions
- Addressing modes
- The operations provided in the instruction set
- Type and size of operands
- Control flow instructions
- Encoding

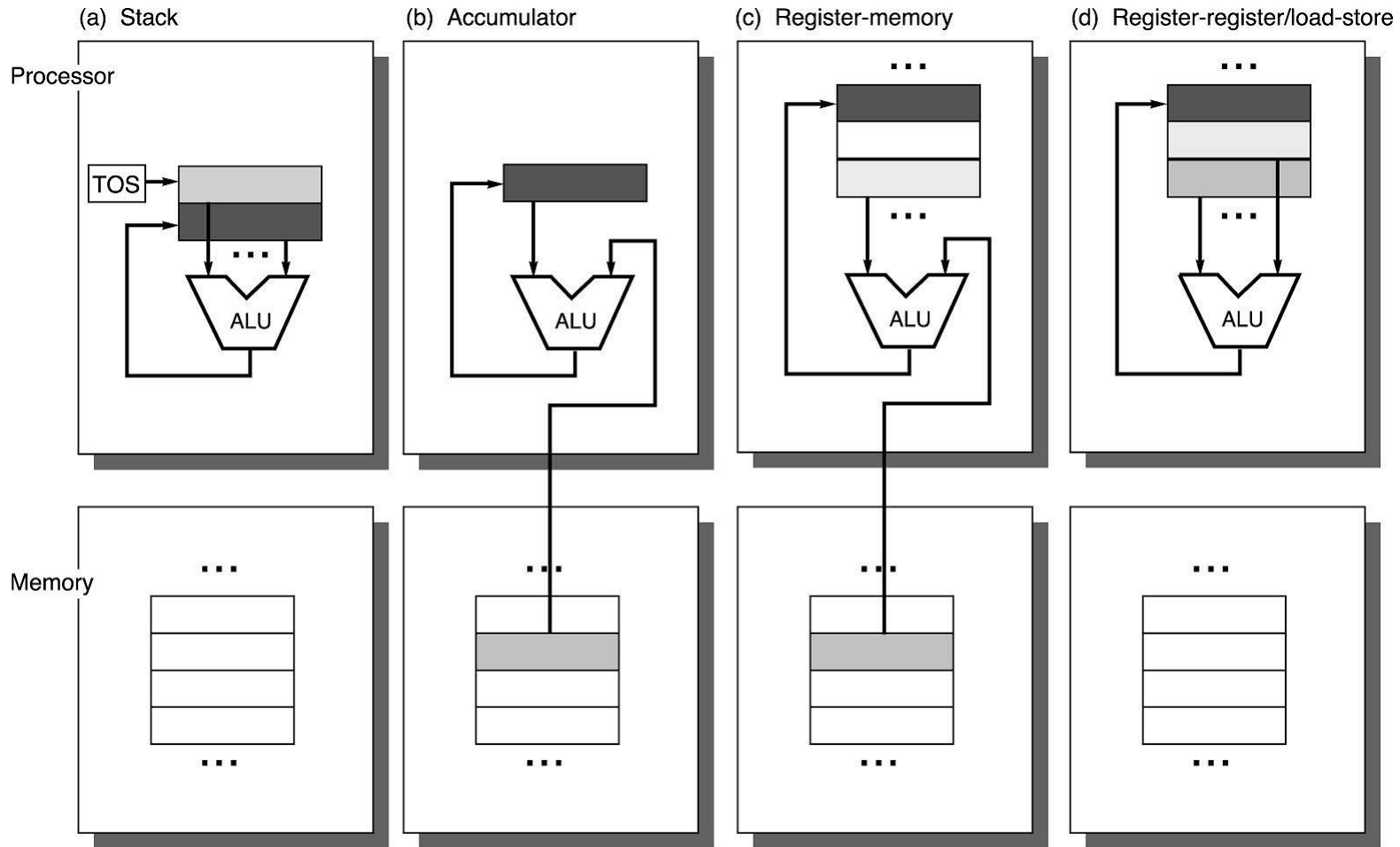


# ISA Design Issues

- ❑ **Where are operands stored?**
  - registers, memory, stack, accumulator
- ❑ **How many explicit operands are there?**
  - 0, 1, 2, or 3
- ❑ **How is the operand location specified?**
  - register, immediate, indirect, . . .
- ❑ **What type & size of operands are supported?**
  - byte, int, float, double, string, vector. . .
- ❑ **What operations are supported?**
  - add, sub, mul, move, compare . . .
- ❑ **How is the operation flow controlled?**
  - branches, jumps, procedure calls . . .
- ❑ **What is the encoding format**
  - fixed, variable, hybrid...



# ISA Classes: Where are operands stored





# ISA Classes

## How are operands accessed by the CPU?

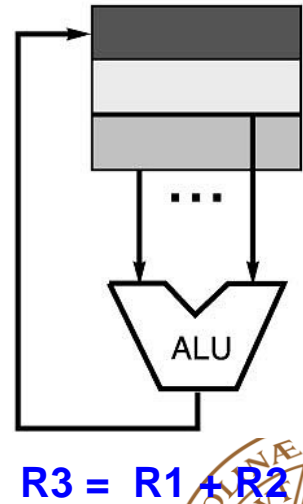
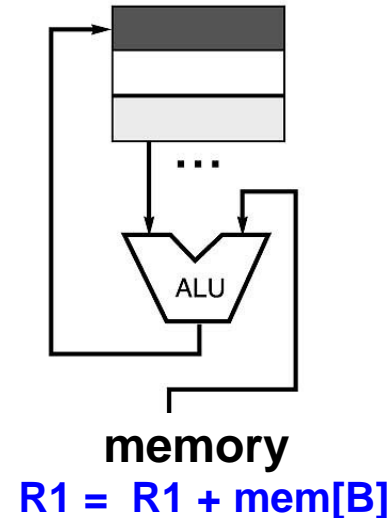
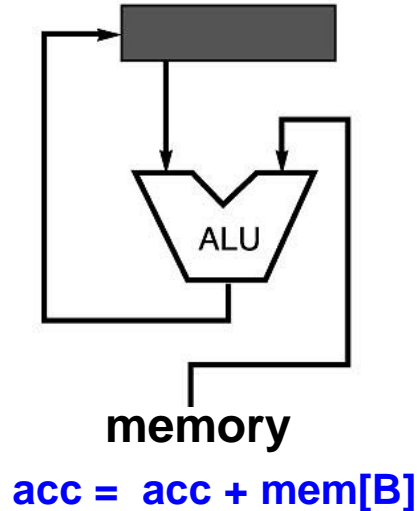
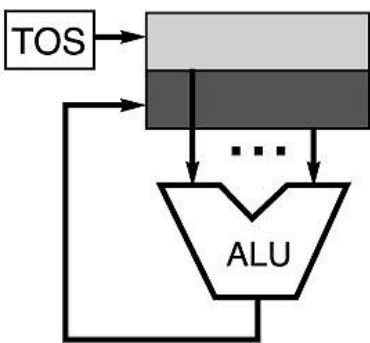
Temporary storage	Examples	Explicit operands	Result destination	Accessing operands
stack	B5500	0	stack	Push, Pop
Accumulator	PDP-8	1	Accumulator	Load/Store
Register (GPR)	IBM 360 VAX MIPS 80x86	2 or 3	Register or Memory	Load/Store or Reg/Mem

Only GPR (General Purpose Register) architectures are used today.

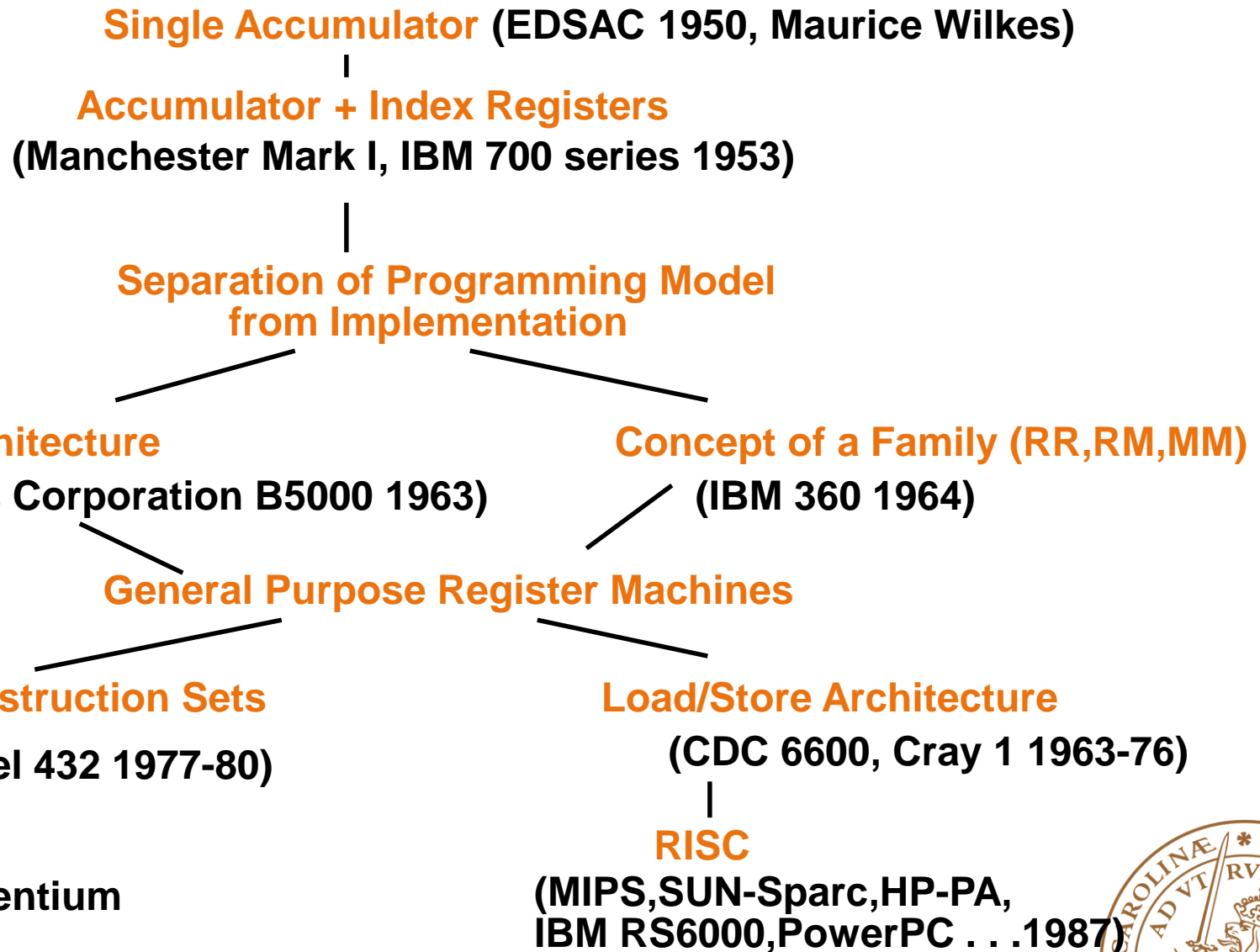


# Example: C=A+B

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1, A Load R2, B Add R3, R1, R2 Store C, R3



# Evolution of ISA



# GPR (General Purpose Register)

## □ Registers are much faster than memory (even cache)

- Register values are available “immediately”
- When memory isn’t ready, processor must wait (“stall”)

## □ Registers are convenient for variable storage

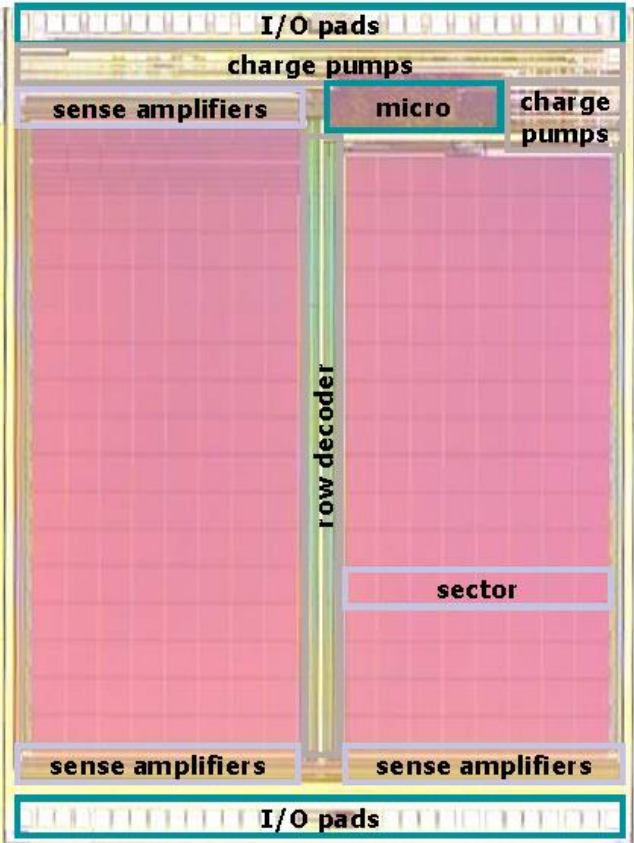
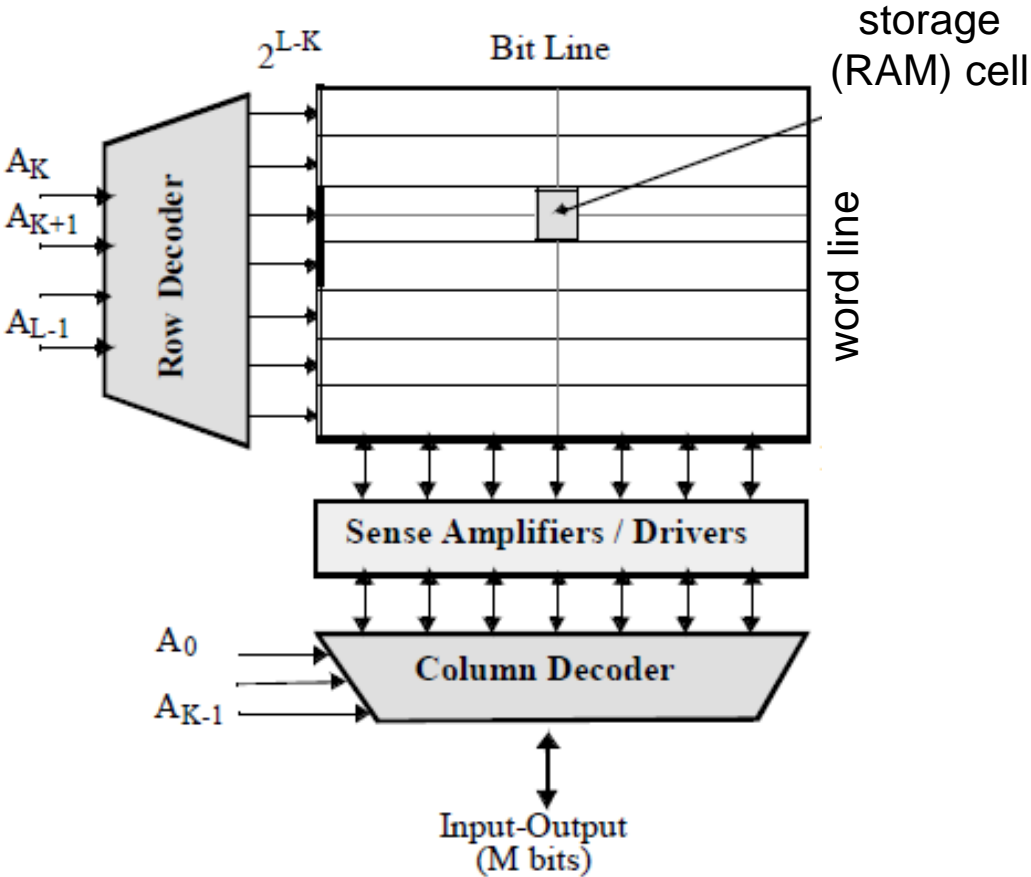
- Compiler assigns some variables (especially frequently used ones) just to registers
- More compact code since small fields specify registers (compared to memory addresses)

## □ Disadvantages

- Higher instruction count
- Dependent on good compiler (Reg. assignment)
- Higher hardware cost (comparing to MEM)



# Memory Architecture



# Register File

□ **Example:** 4-word register file with 1 write port and two read ports

□ **Register array:**

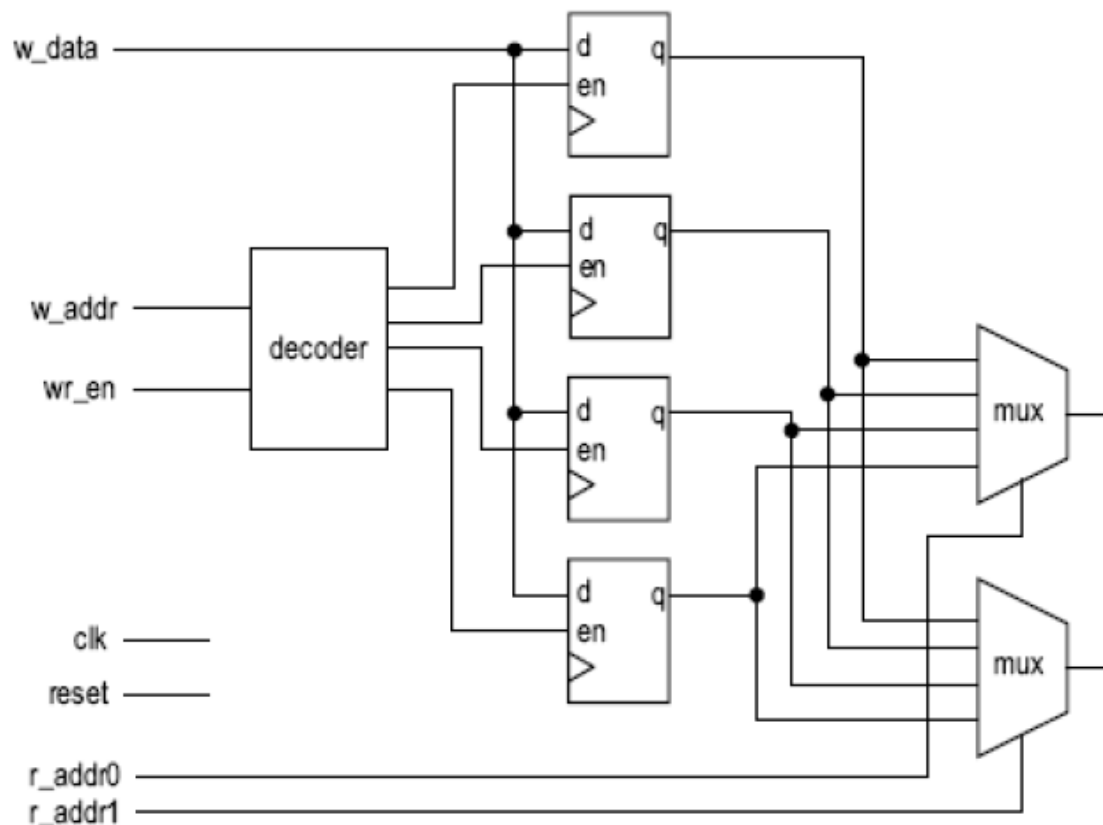
- 4\*16bit registers
- Each register has an enable signal

□ **Write decoding circuit:**

- 0000 if wr\_en is 0
- 1 bit asserted according to w\_addr if wr\_en is 1

□ **Read circuit:**

- A mux for each read port



# Reg v.s. Mem (65nm CMOS)

	Register Bank	Memory
Size	256*4Byte	1K*4Byte
Area	0.14mm <sup>2</sup>	0.04mm <sup>2</sup>
Density	7KB/mm <sup>2</sup>	100KB/mm <sup>2</sup>



# Example: RISC-CICS

MULT 2:3, 5:2

LOAD A, 2:3

LOAD B, 5:2

PROD A, B

STORE 2:3, A

## CISC

Emphasis on hardware

Includes multi-clock  
complex instructions

Memory-to-memory:  
"LOAD" and "STORE"  
incorporated in instructions

Small code sizes,  
high cycles per second

Transistors used for storing  
complex instructions

## RISC

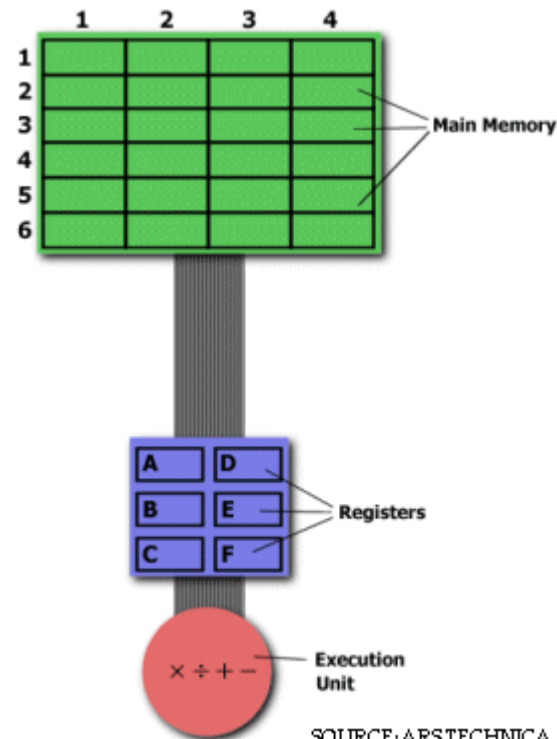
Emphasis on software

"Single"-clock,  
reduced instruction only

Register to register:  
"LOAD" and "STORE"  
are independent instructions

Low cycles per second,  
large code sizes

Spends more transistors  
on memory registers



SOURCE: ARSTECHNICA





# Example: RISC-CICS

## □ RISC (Reduced Instruction Set Computing)

- Simple instructions
- MIPS, ARM, ...
- Easier to design, build
- Less power
- Larger code size (IC), but in total (byte)?
- “Easier” for compiler, but for optimization?

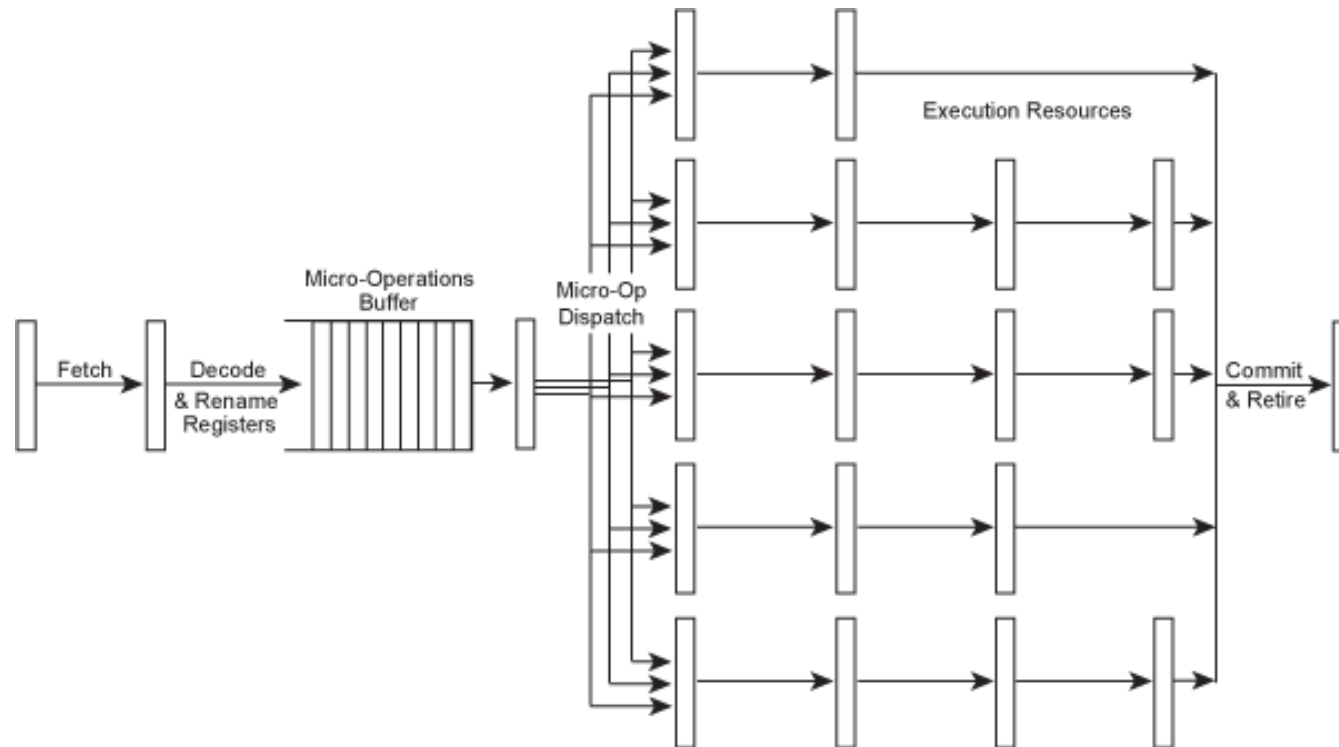
## □ CISC (Complex Instruction Set Computing)

- Complex instructions
- VAX, Intel 80x86 (now RISC-like internally), ...

<http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/riscisc/>



# Example: RISC-CICS



$$\text{CPUtime} = \text{Execution time} = \text{seconds/program} =$$

$$\underbrace{(\text{executed}) \text{instr.} / \text{program}}_{IC} * \underbrace{\text{cycles} / \text{instr.}}_{CPI} * \underbrace{\text{seconds} / \text{cycle}}_{T_c}$$



# Stack v.s. GPR

## □ Advantages

- Very compact object code

<b>Push A</b>	<b>Load R1, A</b>
<b>Push B</b>	<b>Load R2, B</b>
<b>Add</b>	<b>Add R3, R1, R2</b>

- Simple compilers (no reg. assignment)
- Minimal processor state (simple hardware)

## □ Disadvantages

- More memory references (if stack is implemented with MEM)

	<b>load X, push to memory</b>
	<b>load 1, push to memory</b>
X+1	<b>pop 2 values from memory, add, and push result to memory</b>

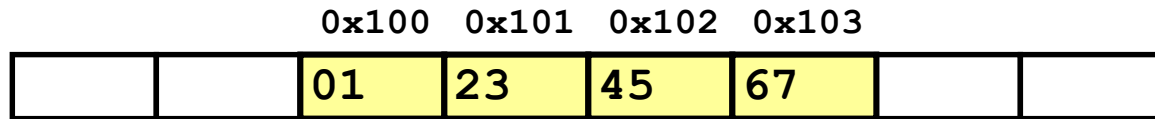
- Factoring out common subexpressions has high cost
- Can add some registers (hybrid)



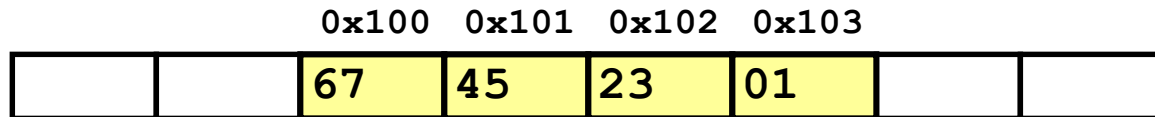
# Memory Addressing

□ A 32 bit (4Byte) integer variable (0x01234567) stored at address 0x100

- Big Endian
  - Least significant byte has highest address



- Little Endian
  - Least significant byte has lowest address



- **Important for exchange of data**, (and strings)

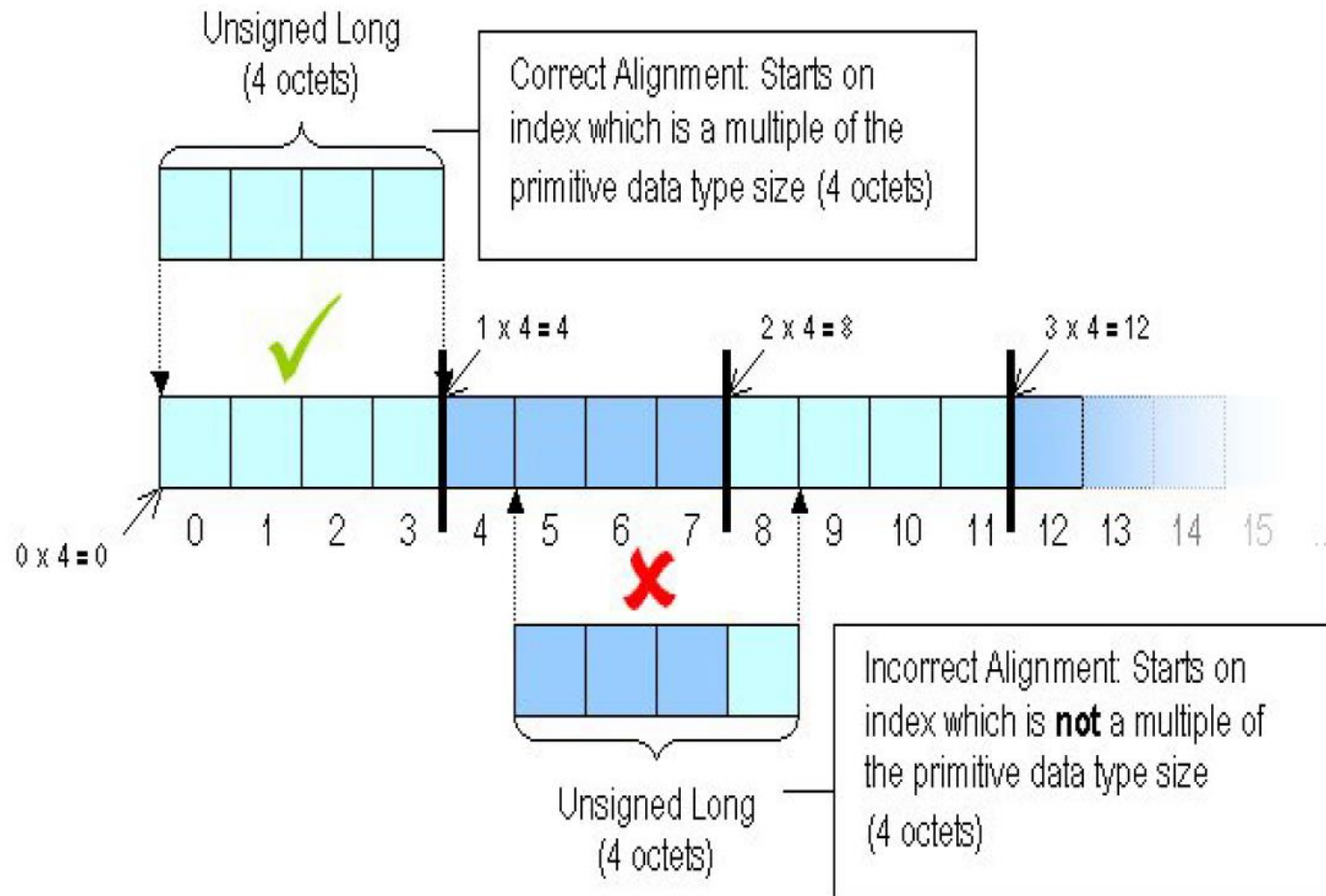


# Memory Addressing

- ❑ **Memory is generally byte addressed and provides access for**
  - bytes (8 bits), half words (16 bits), words (32 bits), and double words(64 bits)
- ❑ **Access to data-objects > 1 byte?**
- ❑ **An architecture may require that data is aligned:**
  - Address index is multiple of data type size ([depending on memory implementation](#))
  - byte always aligned
  - half word (16 bits) aligned at byte offsets 0,2,4,6,...
  - word (32 bits) aligned at byte offsets 0,4,8,12,...
  - double word (64 bits) aligned at byte offsets 0,8,16,24,...



# Memory Alignment



# Memory Alignment

Width of object	Value of 3 low-order bits of byte address								
	0	1	2	3	4	5	6	7	
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned		
2 bytes (half word)		Misaligned	Misaligned	Misaligned	Misaligned	Misaligned	Misaligned	Misaligned	
4 bytes (word)	Aligned				Aligned				
4 bytes (word)		Misaligned			Misaligned				
4 bytes (word)			Misaligned				Misaligned		
4 bytes (word)				Misaligned					Misaligned
8 bytes (double word)	Aligned								
8 bytes (double word)		Misaligned							
8 bytes (double word)			Misaligned						
8 bytes (double word)				Misaligned					
8 bytes (double word)					Misaligned				
8 bytes (double word)						Misaligned			
8 bytes (double word)							Misaligned		
8 bytes (double word)								Misaligned	

**Figure B.5** Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order 3 bits of the address.





# Memory Addressing Mode

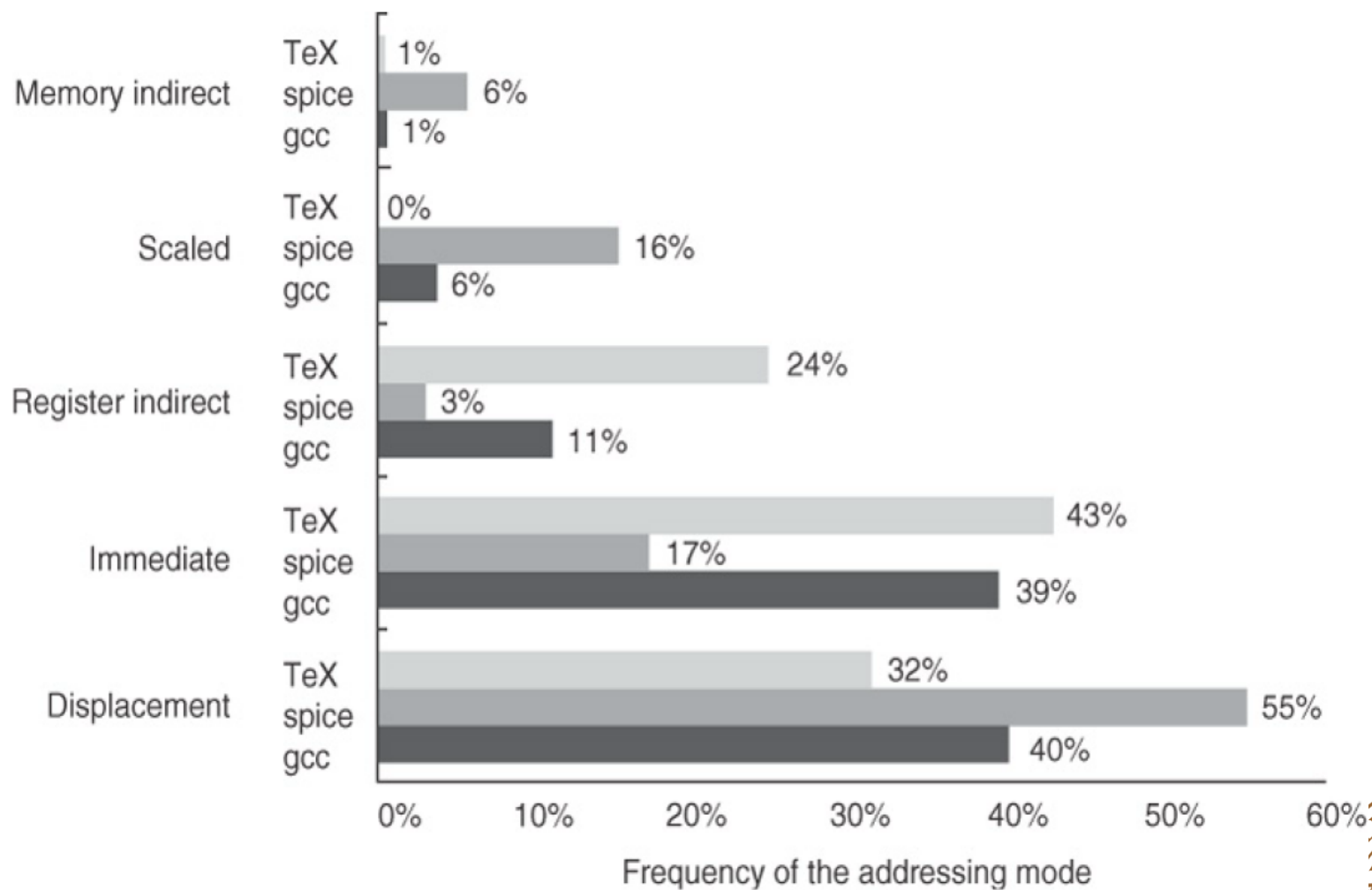
Addressing Mode	Example	Action
1. Register direct	Add R4, R3	$R4 \leftarrow R4 + R3$
2. Immediate	Add R4, #3	$R4 \leftarrow R4 + 3$
3. Displacement	Add R4, 100(R1)	$R4 \leftarrow R4 + M[100 + R1]$
4. Register indirect	Add R4, (R1)	$R4 \leftarrow R4 + M[R1]$
5. Indexed	Add R4, (R1 + R2)	$R4 \leftarrow R4 + M[R1 + R2]$
6. Direct	Add R4, (1000)	$R4 \leftarrow R4 + M[1000]$
7. Memory Indirect	Add R4, @(R3)	$R4 \leftarrow R4 + M[M[R3]]$
8. Auto-increment	Add R4, (R2)+	$R4 \leftarrow R4 + M[R2]$ $R2 \leftarrow R2 + d$
9. Auto-decrement	Add R4, (R2)-	$R4 \leftarrow R4 + M[R2]$ $R2 \leftarrow R2 - d$
10. Scaled	Add R4, 100(R2)[R3]	$R4 \leftarrow R4 +$ $M[100 + R2 + R3 * d]$





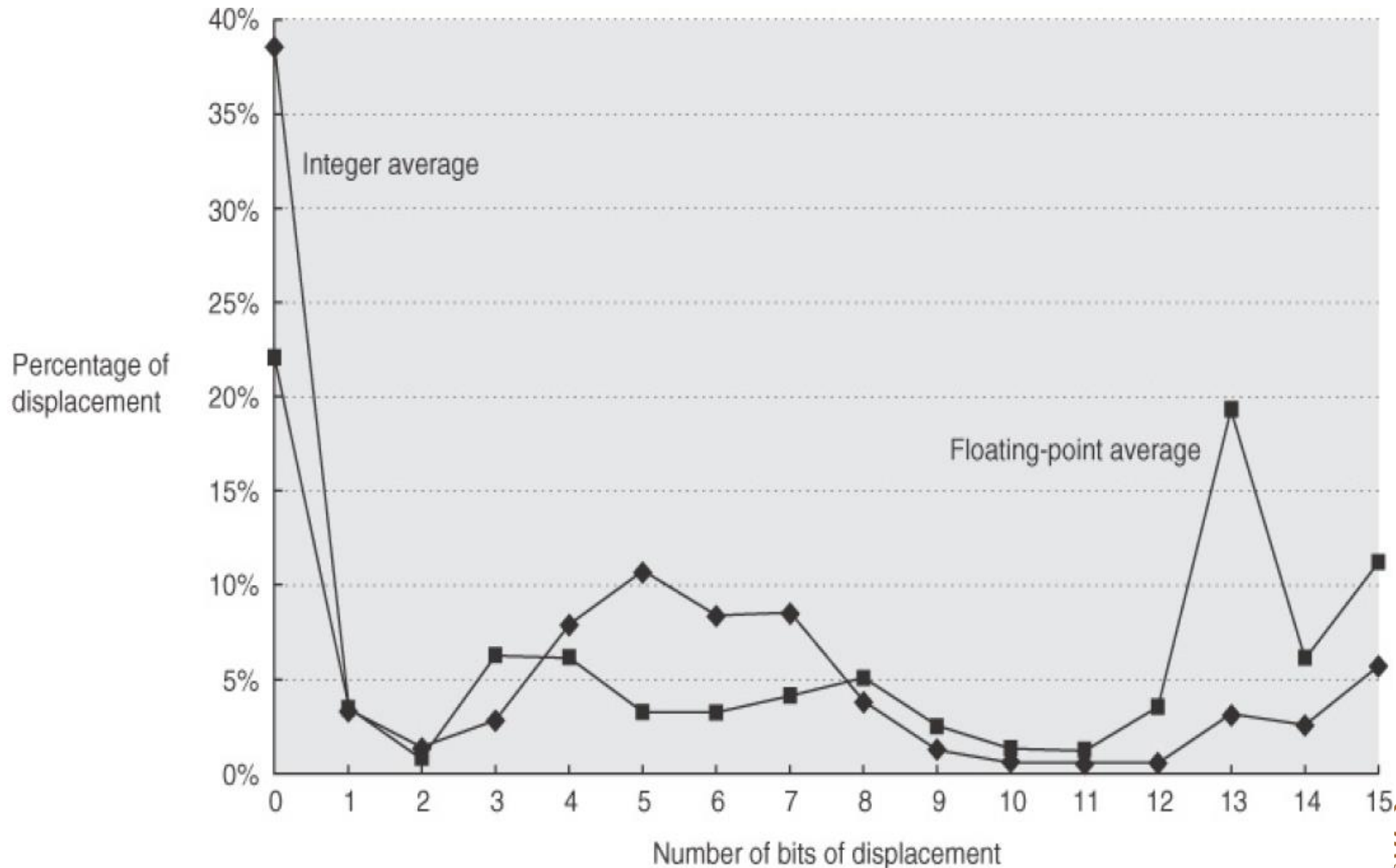
# Memory addressing mode (on VAX)

□ Are all these addressing modes needed?



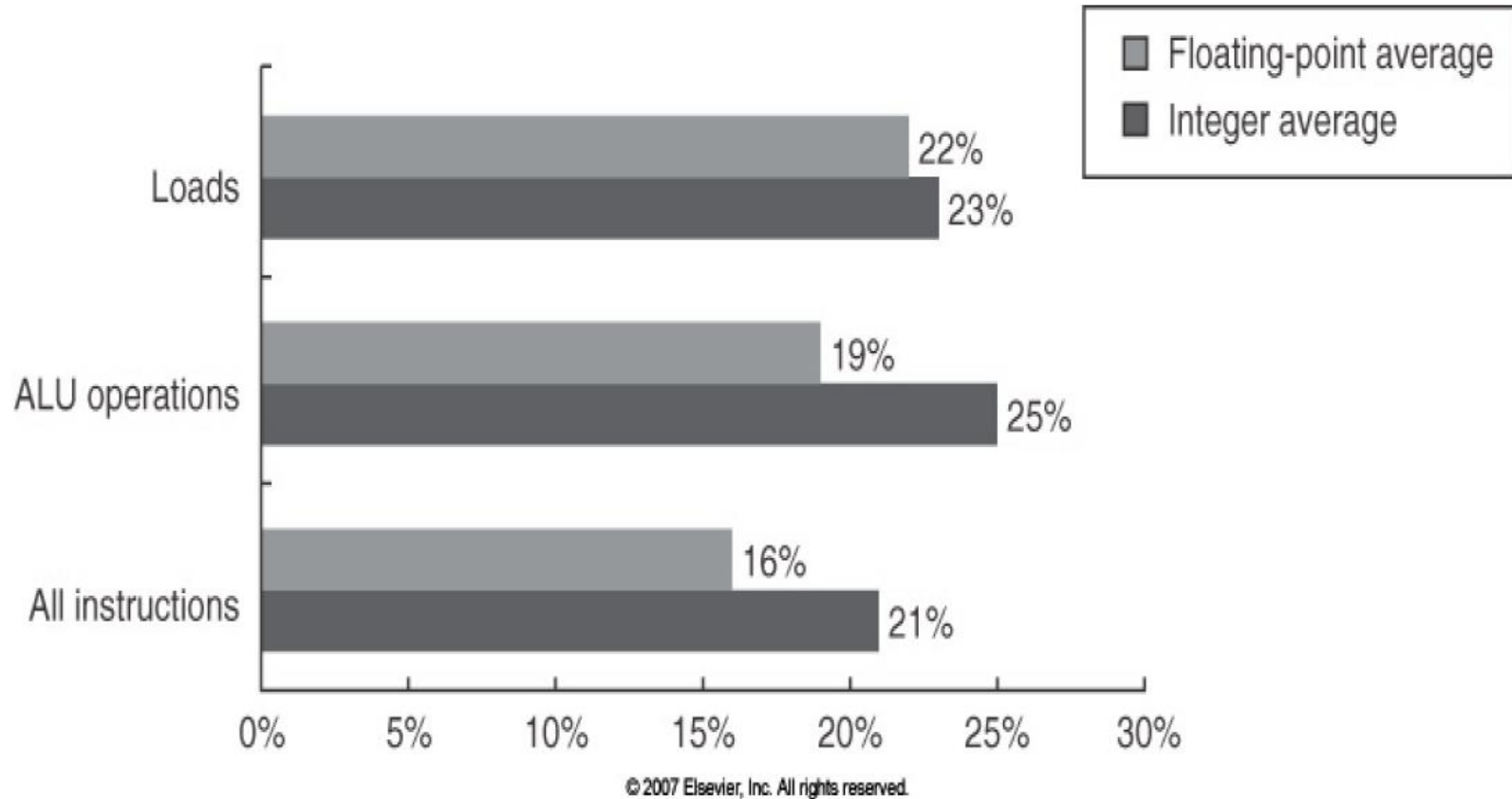
# Memory addressing mode

□ How many bits are needed for address displacement?



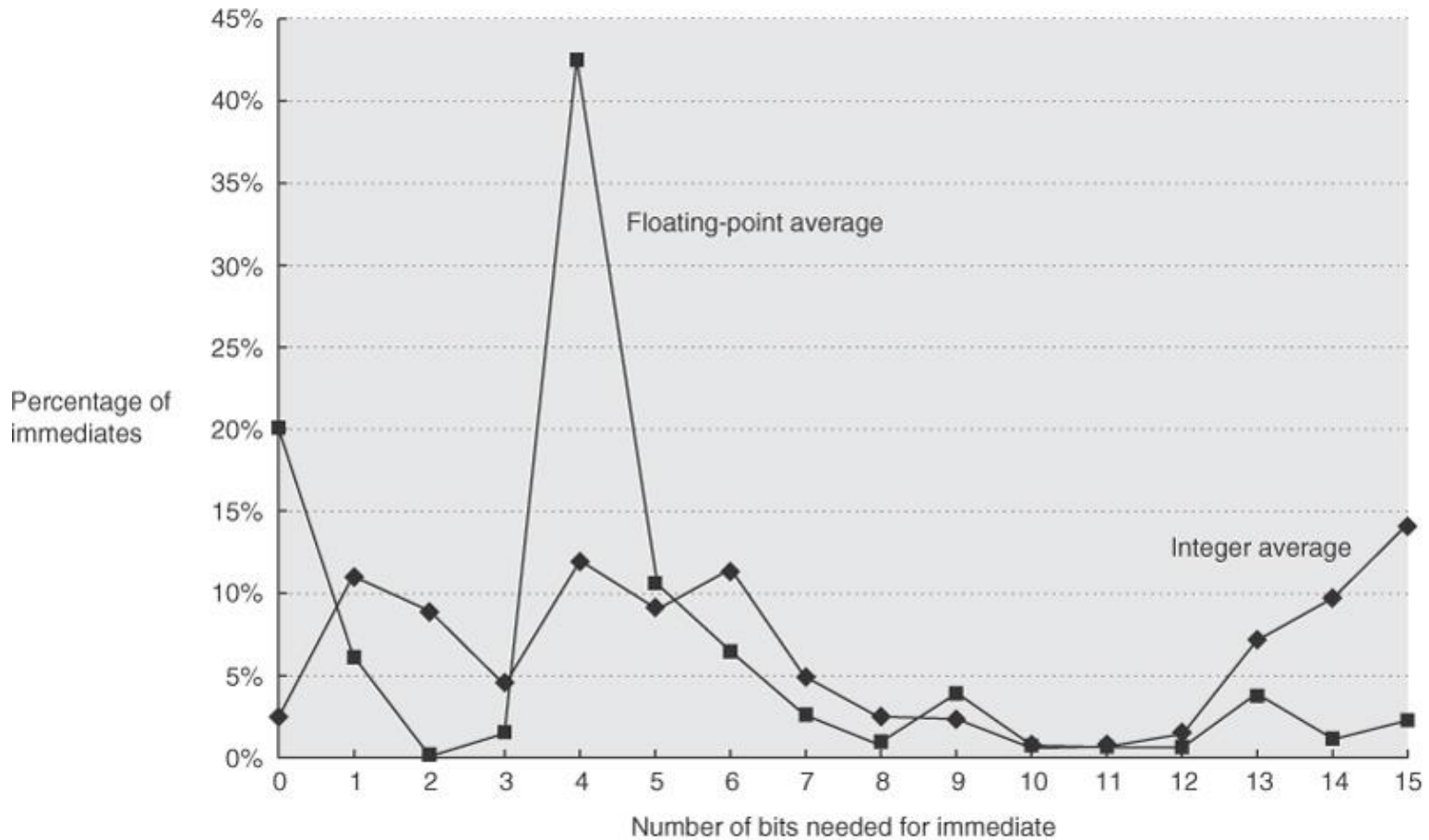
# Memory addressing mode

## How important are immediats?



# Memory addressing mode

□ How many bits are needed for immediate?



© 2007 Elsevier, Inc. All rights reserved.



# ISA Classification

## □ What's needed in an instruction set?

- Addressing
- **Operands**
- Operations
- Control Flow



# What does it mean?

00100000010010000110100100100001

---



# What does it mean?

00100000010010000110100100100001

001000	00010	01000	0110100100100001
instruction	source	target	immediate
ADDI	2	8	26913

ADDI R8,R2,26913



# Types and sizes of operands

- integer
- floating point (single precision)
- character
- packed decimal
- ... etc ...







# Floating point v.s. fixed point

	32, 64bit Fixed-point (with 2, 4 pipeline stages)		32, 64bit Floating-point (with 6, 8 pipeline stages)		32, 64bit Floating-point (with 18, 23 pipeline stages)	
Area (slices)	36	139	293	693	504	1383
Max Freq. (MHz) achievable	250	250	140	130	250	200
Power (mW) at 100MHz	23.48	104	148.7	329	-	-

**Table 1a: A comparison of addition units (Virtex2Pro-c2vp125-7)**

	32, 64bit Fixed-point (with 5, 7 pipeline stages)		32, 64bit Floating-point (with 9, 11 pipeline stages)		64bit Floating-point (with 18, 23 pipeline stages)	
Area (slices)/Embedded multipliers	190 / 4	1024 / 16	249 / 3	775 / 10	492 / 3	1558 / 10
Max Freq. (MHz) achievable	200	130	140	130	200	200
Power (mW) at 100MHz	136.3	804	164.7	424	-	-

**Table 1b: A comparison of multiplication units (Virtex2Pro-c2vp125-7)**



# Different Data representation

$$6/8 = 0.75$$

Binary: 0.11

Stochastic : 10111011,  $p=P(x=1)$

$$3/10 = 0.3$$

Binary: ?

Stochastic : 0010010001



# Advantages: Error tolerant

**Binary:**

3/8: 0.011 → 7/8: 0.111

**Stochastic:**

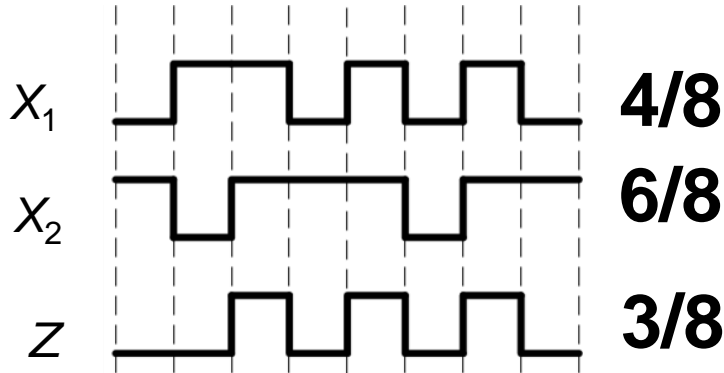
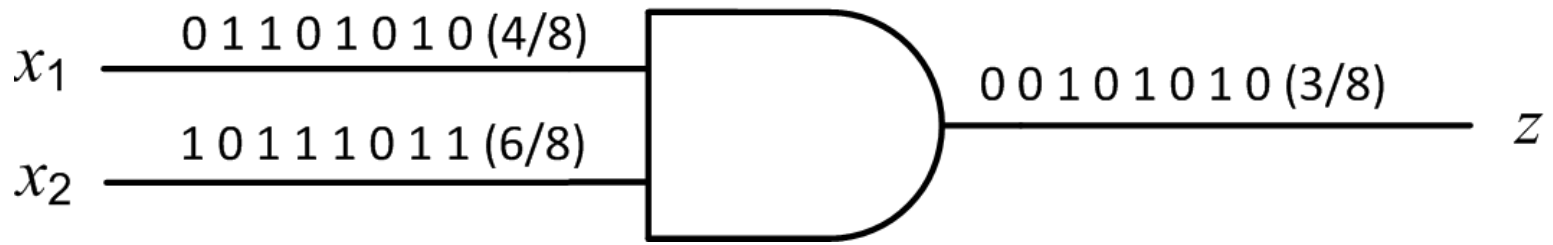
3/8: 00011010 → 4/8: 10011010



# Advantages: Simple arithmetic

$$Z = X_1 \times X_2$$

$$3/8 = 4/8 \times 6/8$$



# ISA Classification

## □ What's needed in an instruction set?

- Addressing
- Operands
- **Operations**
- Control Flow



# Types of operations

- Arithmetic and Logic: AND, ADD
- Data Transfer: MOVE, LOAD, STORE
- Control: BRANCH, JUMP, CALL
- System: OS CALL, VM
- Floating Point: ADDF, MULF, DIVF
- Decimal: ADDD, CONVERT
- String: MOVE, COMPARE, SEARCH
- Graphics: (DE)COMPRESS



# Types of operations (frequency)

<i>Rank</i>	<i>Instruction</i>	<i>Frequency</i>
<b>1</b>	<b>load</b>	<b>22%</b>
<b>2</b>	<b>branch</b>	<b>20%</b>
<b>3</b>	<b>compare</b>	<b>16%</b>
<b>4</b>	<b>store</b>	<b>12%</b>
<b>5</b>	<b>add</b>	<b>8%</b>
<b>6</b>	<b>and</b>	<b>6%</b>
<b>7</b>	<b>sub</b>	<b>5%</b>
<b>8</b>	<b>register move</b>	<b>4%</b>
<b>9</b>	<b>call</b>	<b>1%</b>
<b>10</b>	<b>return</b>	<b>1%</b>
<b>Total</b>		<b>96%</b>

## 80x86 Instruction Frequency





# ISA Classification

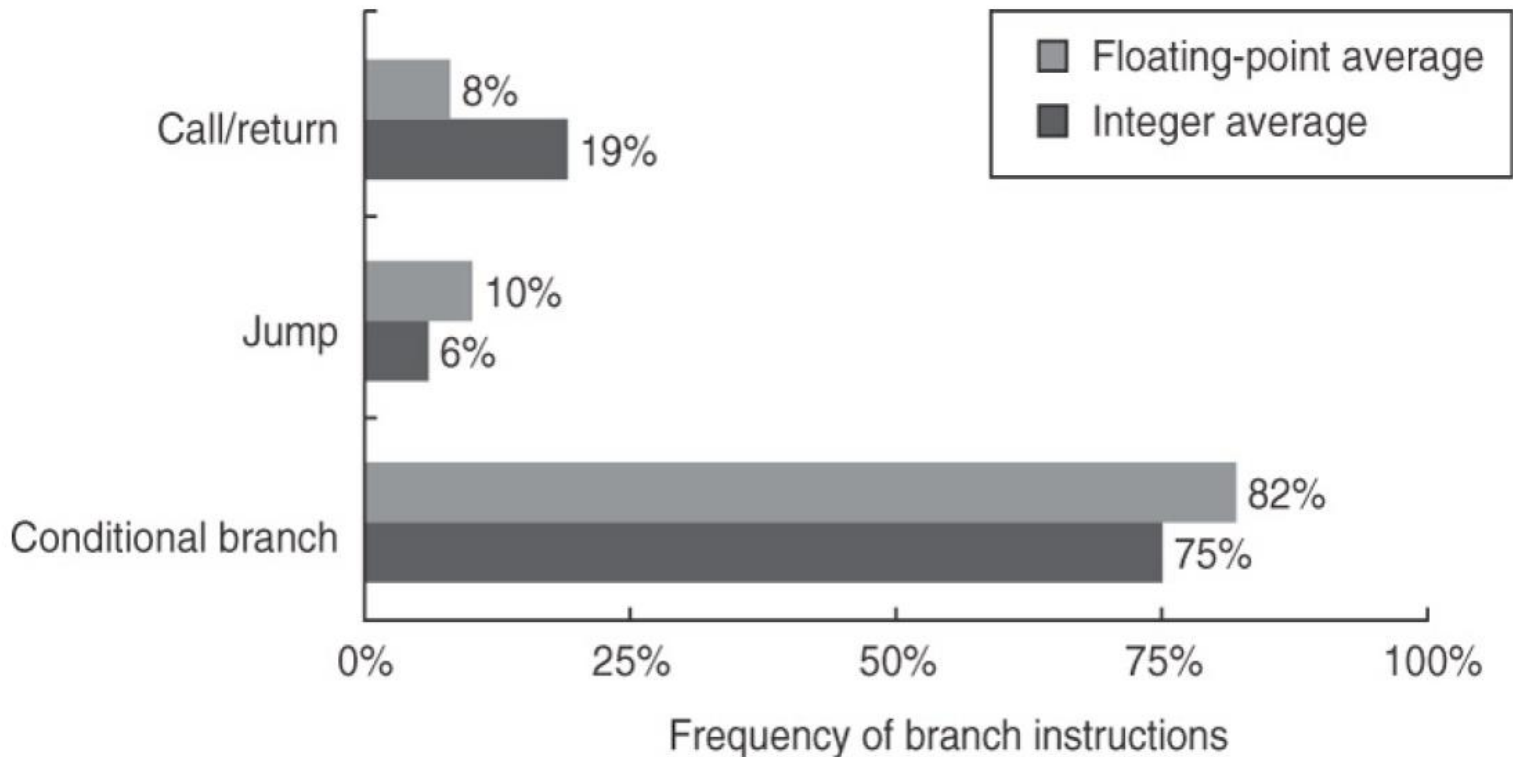
## □ What's needed in an instruction set?

- Addressing
- Operands
- Operations
- **Control Flow**



# Types of control instructions

- Conditional branches
- Unconditional branches (jumps)
- Procedure call/returns

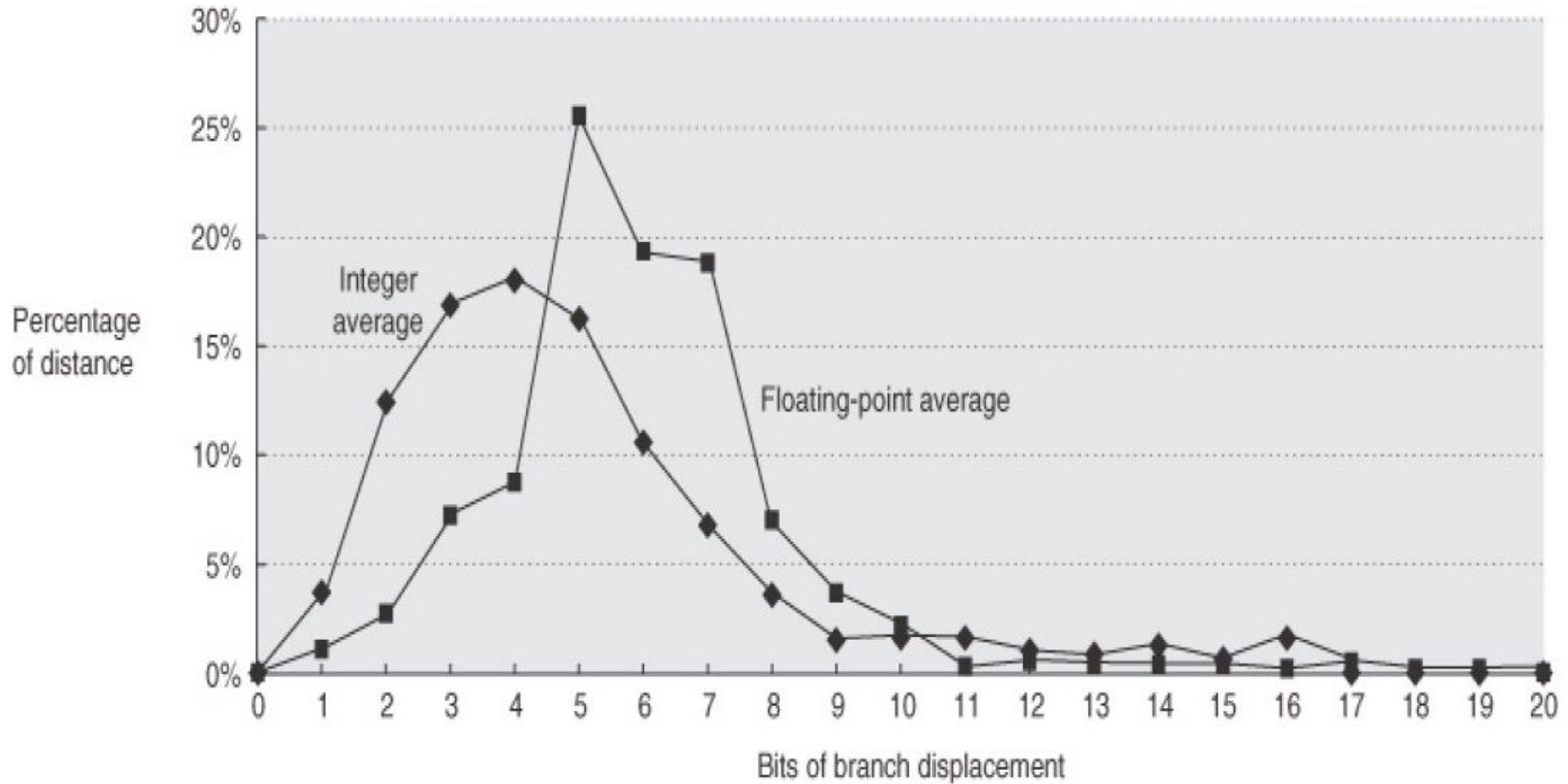


© 2007 Elsevier Inc. All rights reserved.



# Types of control instructions

## How large need the branch displacement be?



© 2007 Elsevier, Inc. All rights reserved.



# Instruction format

## □ Variable instruction format

- Compact code but the instruction decoding is more complex and thus slower
- Examples: VAX, Intel 80x86 (1-17 byte)

Operation # operands	Address specifier 1	Address field 1	...	Address specifier x	Address field x
-------------------------	------------------------	--------------------	-----	------------------------	--------------------

## □ Fixed instruction format

- Easy and fast to decode but gives large code size
- Examples: Alpha, ARM, MIPS (4byte), PowerPC, SPARC

Operation	Address field 1	Address field 2	Address field 3
-----------	--------------------	--------------------	--------------------



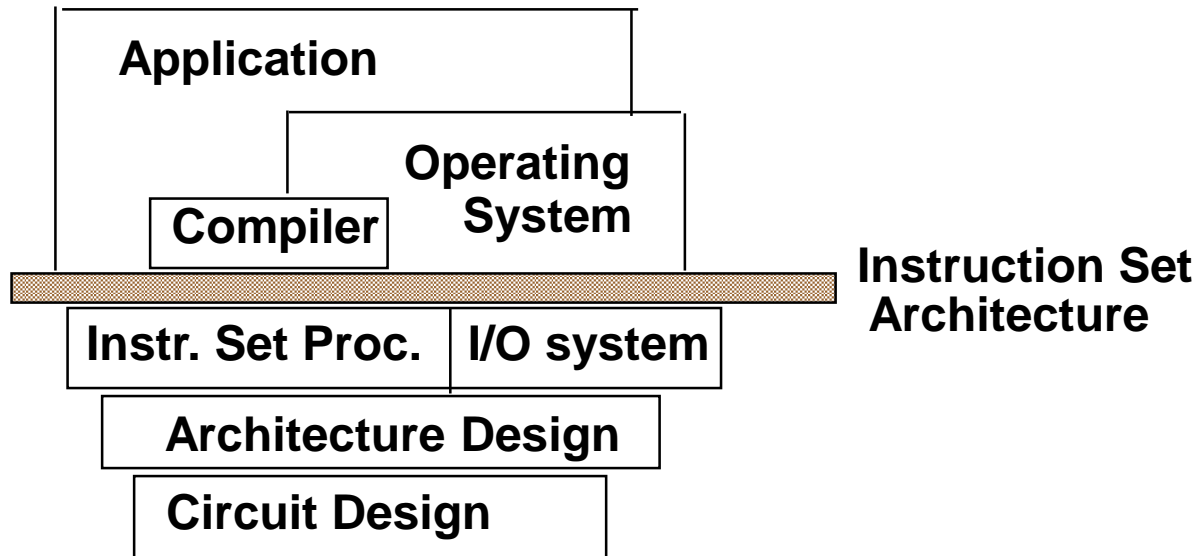
# Outline

- Reiteration
- Instruction Set Principles
- **The Role of Compilers**
- MIPS

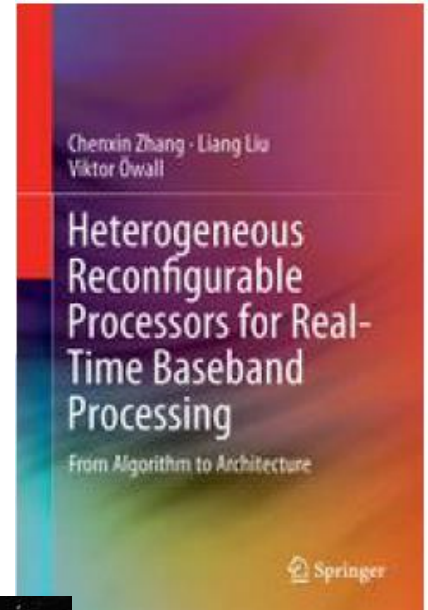
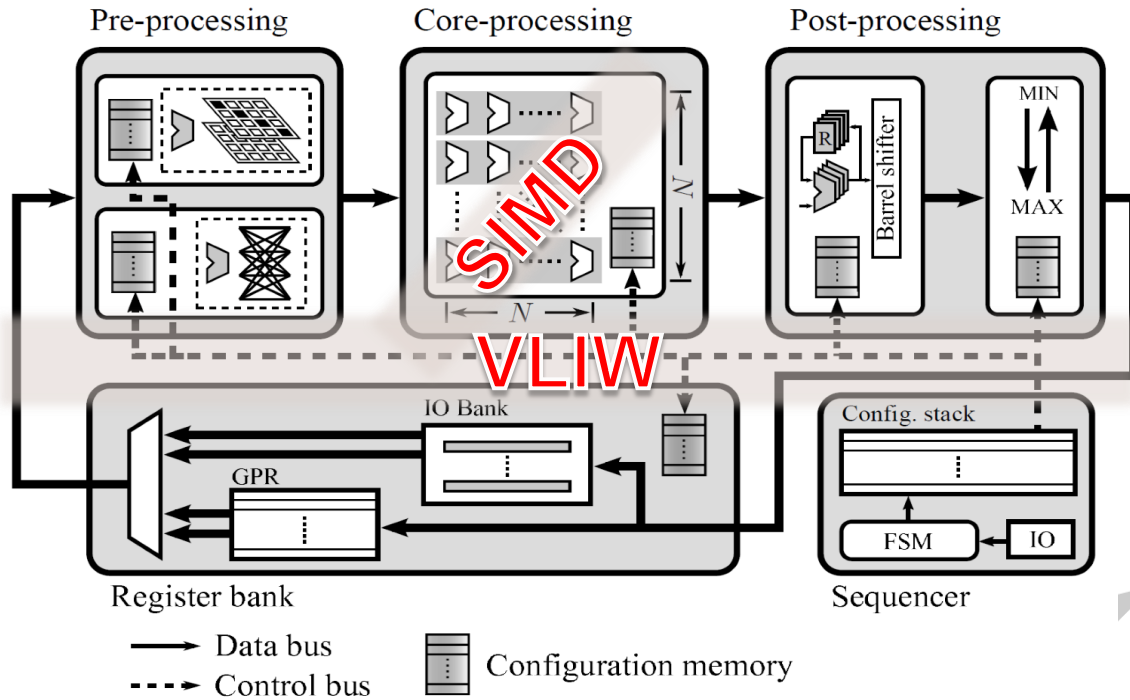


# ISA and compiler

- ❑ Instruction set architecture is a compiler target
- ❑ By far most instructions executed are generated by a compiler (exception certain special purpose processors)
- ❑ **Interaction compiler - ISA critical for overall performance**



# ISA and compiler: a “good or bad?” example



# ISA and compiler: a “good or bad?” example

A	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z																		
	VDPU						VCPU					vmacArray						VDEU																				
21	Process prologue	Row 0	src	vArrayB	1	Reserved			0	Reserved				0	Reserved					0																		
20			0	0	0				0					0						0		0	0	0	0	0	0	0	0	0	0	0	0	0	0			
19			0	0	0				0					0						0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
18			opcode	Neg, i	1				0					0						0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
17			0	0	0				0					0						0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
16			0	0	0				0					0						0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
15		Row 1	src	vArrayB	1	perm_en	Enable	1	Row 0	0	VDDU		opcode	Complex	0	Acc		en_r3	Disable	0	en_r2	Disable	0															
14			0	0	0					0					0									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13			opcode	Neg, i	1					0					0									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0															
11		Row 2	src	vArrayB	1	perm_imm			Row 0	0	VMAC array		mul_en	Enable	1	Sum		reg_acc_r3	Direct	0	reg_acc_r2	Direct	0															
10			0	0	0					0					0									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9			opcode	Neg, i	1					0					0									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0															
7		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0															
6		Row 3	src	vArrayB	1		Row 3	1	Row 3	1			add_sign_l1	Signed	0	Barral shift		mux_in_r23	Straight	0	mux_in_r01	Straight	0															
5	0		0	0	0					0					0									0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	0		0	0	0					0					0									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	opcode	Neg, i	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0																
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0																
1	Mask	src	None	0	opcode	None	0	0	0			op_coeff	Const mul	1		shift	1	0	0	0	0	0																
0														0									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Inst [HEX]	00252948					000021E0				00001E5A					00000011																							





# The role of compilers

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

Control Signal Specification

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

```
lw $15, 0($2)  
lw $16, 4($2)  
sw $16, 0($2)  
sw $15, 4($2)
```

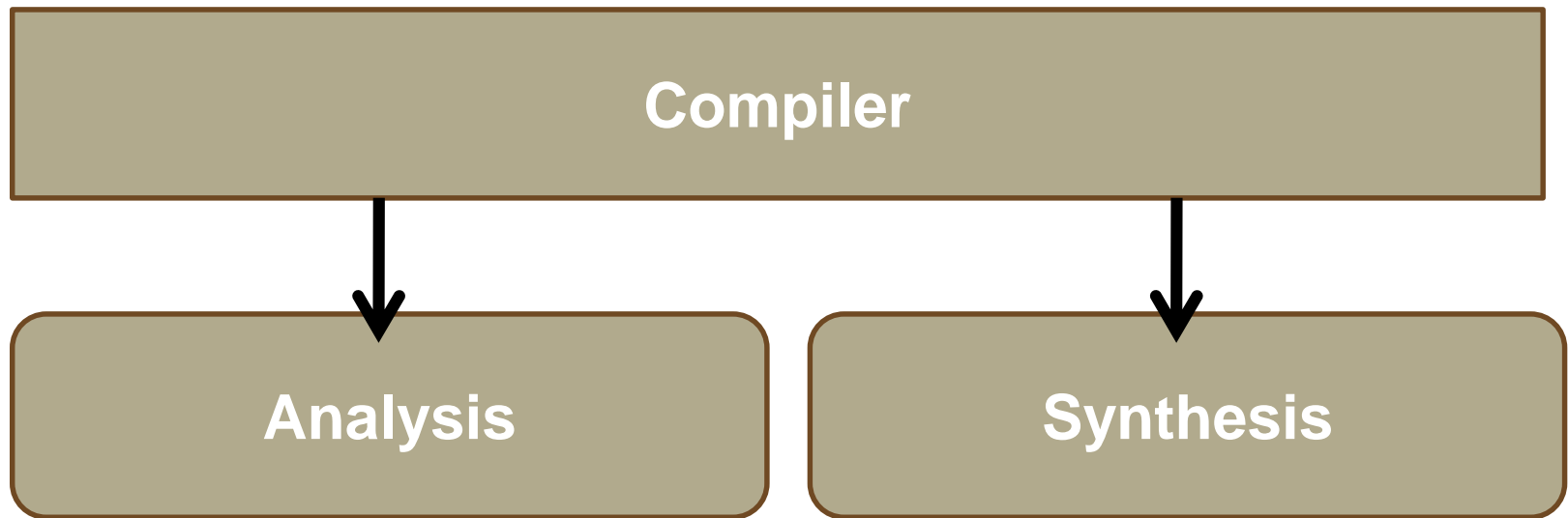
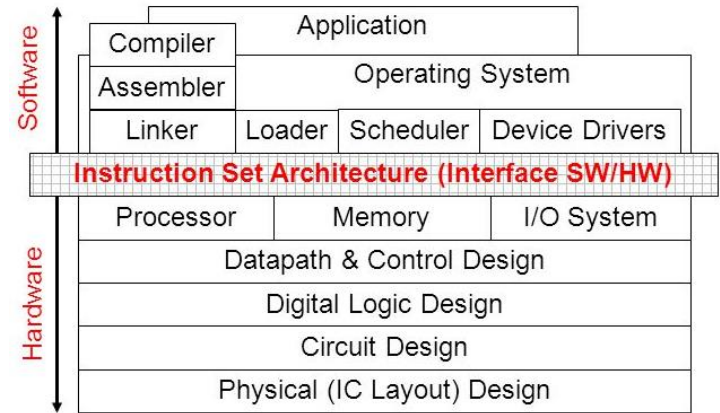
```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```



# The structure of a compiler

## □ Any compiler must perform two major tasks

- Analysis of the source program
- Synthesis of a machine-language program



# The structure of a compiler

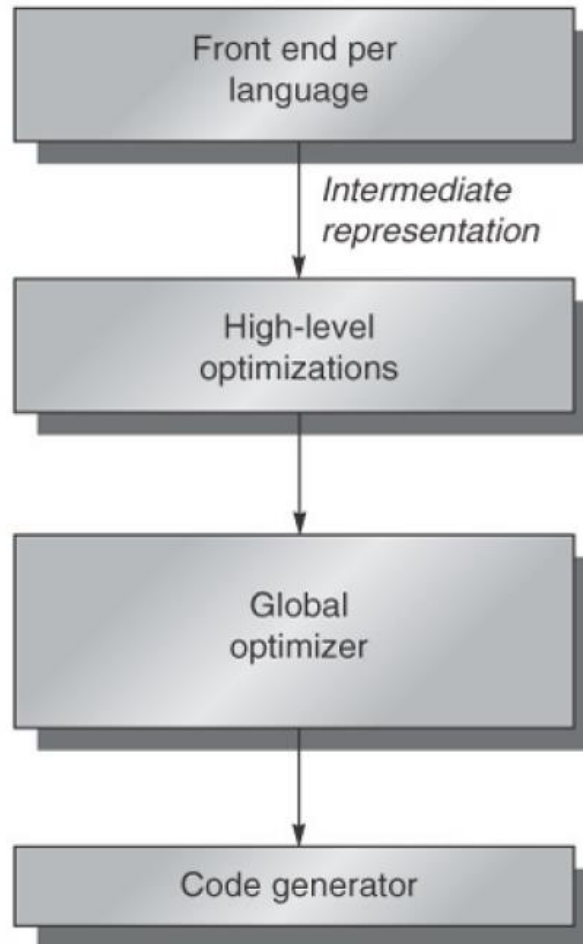
## Dependencies

Language dependent;  
machine independent

Somewhat language dependent;  
largely machine independent

Small language dependencies;  
machine dependencies slight  
(e.g., register counts/types)

Highly machine dependent;  
language independent



## Function

Transform language to  
common intermediate form

For example, loop  
transformations and  
procedure inlining  
(also called  
procedure integration)

Including global and local  
optimizations + register  
allocation

Detailed instruction selection  
and machine-dependent  
optimizations; may include  
or be followed by assembler

© 2007 Elsevier, Inc. All rights reserved.



# GCC optimization options

**O0** – No optimizations performed

**O1** – Local optimizations such as common subexpression elimination, copy propagation, dead-code elimination etc

**O2** – Global optimization, aggressive instruction scheduling, global register allocation

**O3** – Inlining of procedures



# Example of compiler optimization

- **Code improvements made by the compiler are called optimizations and can be classified:**
  - High-order transformations: procedure inlining
  - Optimizations: dead code elimination
  - Constant propagation
  - Common sub-expression elimination
  - Loop-unrolling
  - Register allocation (almost most important)
  - Machine-dependent optimizations: takes advantage of specific architectural features



# Example of compiler optimization

```
int pred(int x) {  
    if (x == 0)  
        return 0;  
    else  
        return x - 1;  
}
```

## Procedure inlining

```
int f(int y) {  
    return pred(y) + pred(0) + pred(y+1);  
}
```

```
int f(int y) {  
    int temp;  
    if (y == 0) temp = 0; else temp = y - 1;  
    if (0 == 0) temp += 0; else temp += 0 - 1;  
    if (y+1 == 0) temp += 0; else temp += (y + 1) - 1;  
    return temp;  
}
```

```
int foo(void)  
{  
    int a = 24;  
    int b = 25; /* Assignment to dead variable */  
    int c;  
    c = a << 2;  
    return c;  
    b = 24; /* Unreachable code */  
    return 0;  
}
```

## Dead code elimination

```
int x = 14;  
int y = 7 - x / 2;  
return y * (28 / x + 2);
```

```
int x = 14;  
int y = 7 - 14 / 2;  
return y * (28 / 14 + 2);
```

```
int x = 14;  
int y = 0;  
return 0;
```

## Constant propagation

```
a = b * c + g;  
d = b * c * e;
```

```
tmp = b * c;  
a = tmp + g;  
d = tmp * e;
```

## Common expression elimination

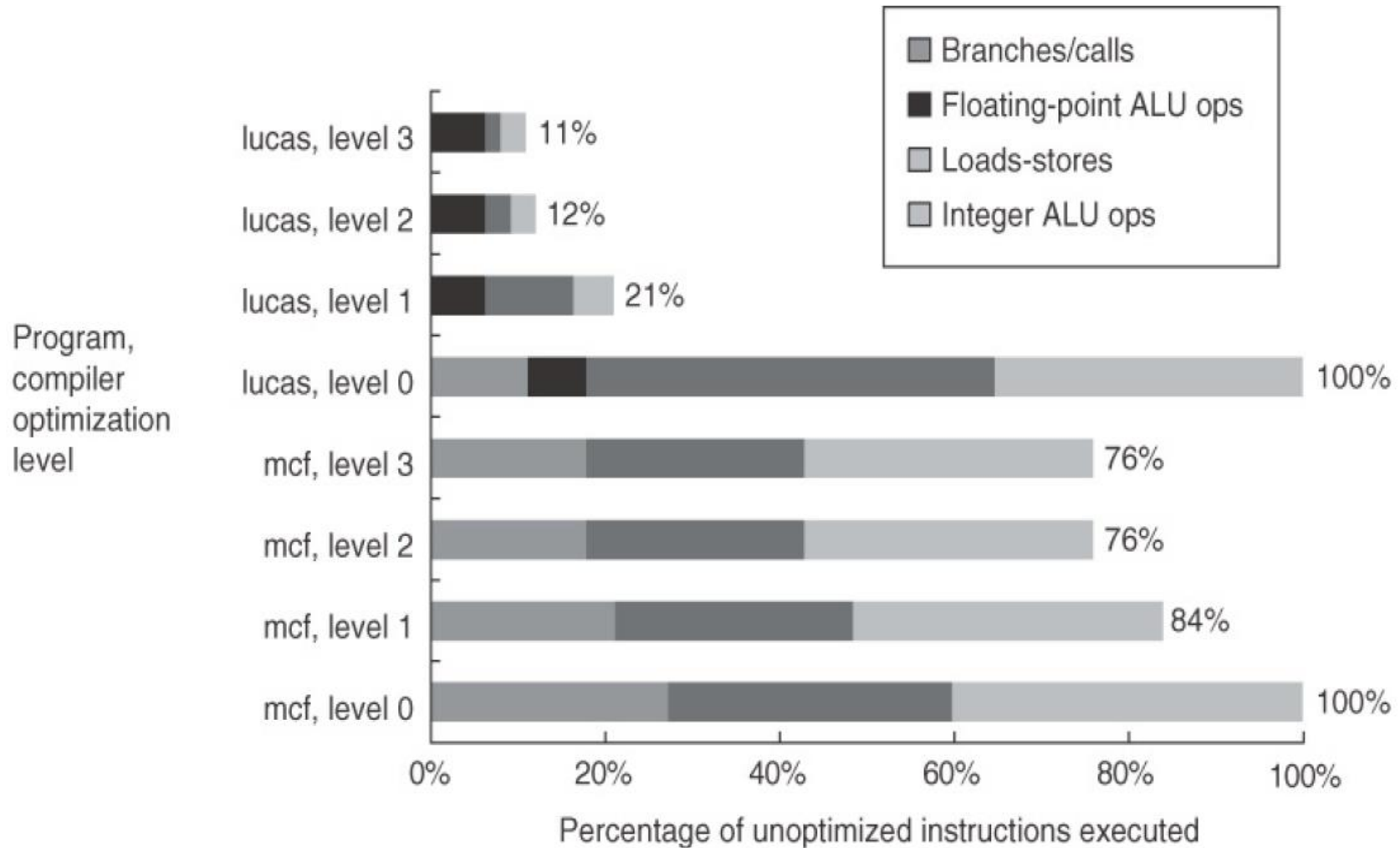


# Example of compiler optimization

- ❑ **Code improvements made by the compiler are called optimizations and can be classified:**
  - High-order transformations: procedure inlining
  - Optimizations: dead code elimination
  - Constant propagation
  - Common sub-expression elimination
  - Loop-unrolling
  - Register allocation (almost most important)
  - Machine-dependent optimizations: takes advantage of specific architectural features
- ❑ **Almost all of these optimizations are easier to do if there are many general registers available!**
  - E.g., common sub/expression elimination stores temporary value into a register
  - Loop-unrolling
  - Procedure inlining



# The impact of compiler optimization



© 2007 Elsevier, Inc. All rights reserved.





# How can you aid compiler

## □ Rules of thumb when designing an instruction set (for general purpose processor):

- Regularity
  - operations, data types and addressing modes should be orthogonal
- Generality
  - Provide primitives, not high-level constructs or solutions
  - Complex instructions are often too specialized
- Simplify trade-offs among alternatives



# Outline

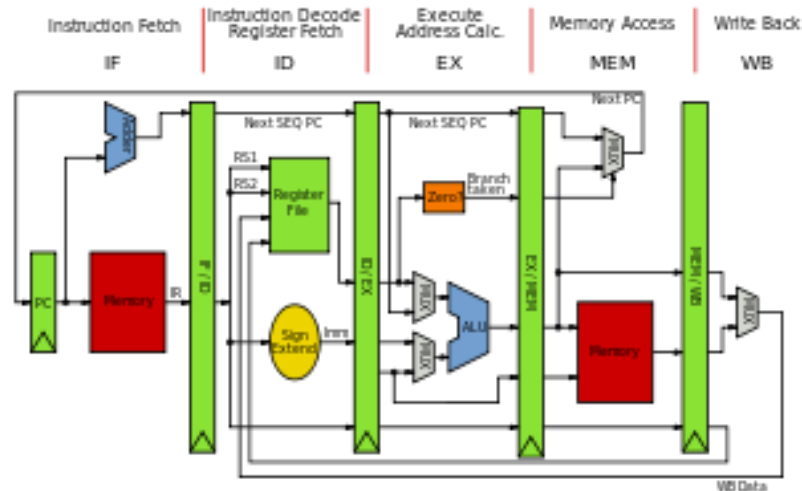
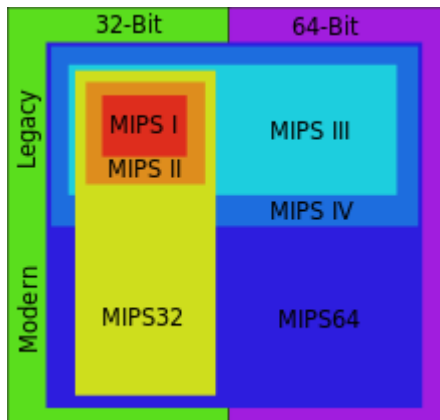
- Reiteration
- Instruction Set Principles
- The Role of Compilers
- **MIPS**



# The MIPS64 architecture

## □ An architecture representative of modern ISA:

- 64 bit load/store GPR architecture
- 32 general integer registers (R0 = 0) and 32 floating point registers
- Supported data types: bytes, half word (16 bits), word (32 bits), double word (64 bits), single and double precision IEEE floating points
- Memory byte addressable with 64 bit addresses
- Addressing modes: immediate and displacement



# MIPS instructions

## □ MIPS instructions fall into 5 classes:

- Arithmetic/logical/shift/comparison
- Control instructions (branch and jump)
- Load/store
- Other (exception, register movement to/from GP registers, etc.)



# MIPS instruction example

LW	R1,60(R7)	Load word
SB	R2,41(R5)	Store byte
MUL	R2,R1,R3	Integer multiply
AND	R3,R2,R1	Logical AND
DADDI	R5,R6,#17	Add immediate
J	lable	Jump
BEQZ	R4,lable	Branch if R4 zero
JALR	R7	Procedure call



# MIPS instruction format

## I-type instruction



Encodes: Loads and stores of bytes, half words, words, double words. All immediates ( $rt \leftarrow rs \text{ op immediate}$ )

Conditional branch instructions (rs is register, rd unused)  
Jump register, jump and link register  
(rd = 0, rs = destination, immediate = 0)

## R-type instruction



Register-register ALU operations:  $rd \leftarrow rs \text{ funct } rt$   
Function encodes the data path operation: Add, Sub, . . .  
Read/write special registers and moves

## J-type instruction



Jump and jump and link  
Trap and return from exception

© 2007 Elsevier Inc. All rights reserved



# Summary

- ❑ **The instruction set architecture have importance for the performance**
- ❑ **The important aspects of an ISA are:**
  - register model
  - addressing modes
  - types of operations
  - data types
  - encoding
- ❑ **Benchmark measurements can reveal the most common case**
- ❑ **Interaction compiler - ISA important**

