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
  - Orgnization (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
  - Thoughput, bandwidth

Application	⟸	Answers/month
Programming	⟸	Response time (seconds)
language	⟸	Operations/second
Compiler		
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

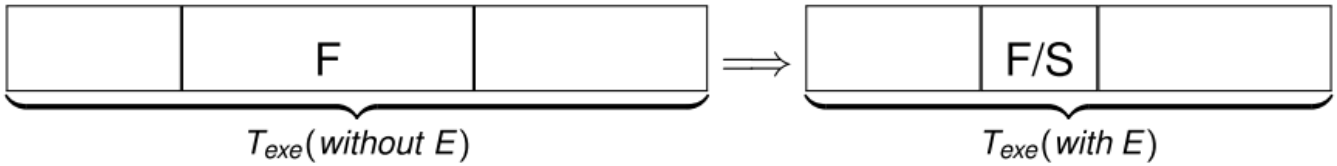
Quantitative Principles

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 makeing a design trade-off, favor the frquent case ove the infrequent case
- Amdahl's Law
  - The performance improvement gained from uisng 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:

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

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

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

Best you could ever hope to do:

$$Speedup_{maximum} = \frac{1}{(1 - Fraction_{enhanced})}$$

Amdahl's Law: example

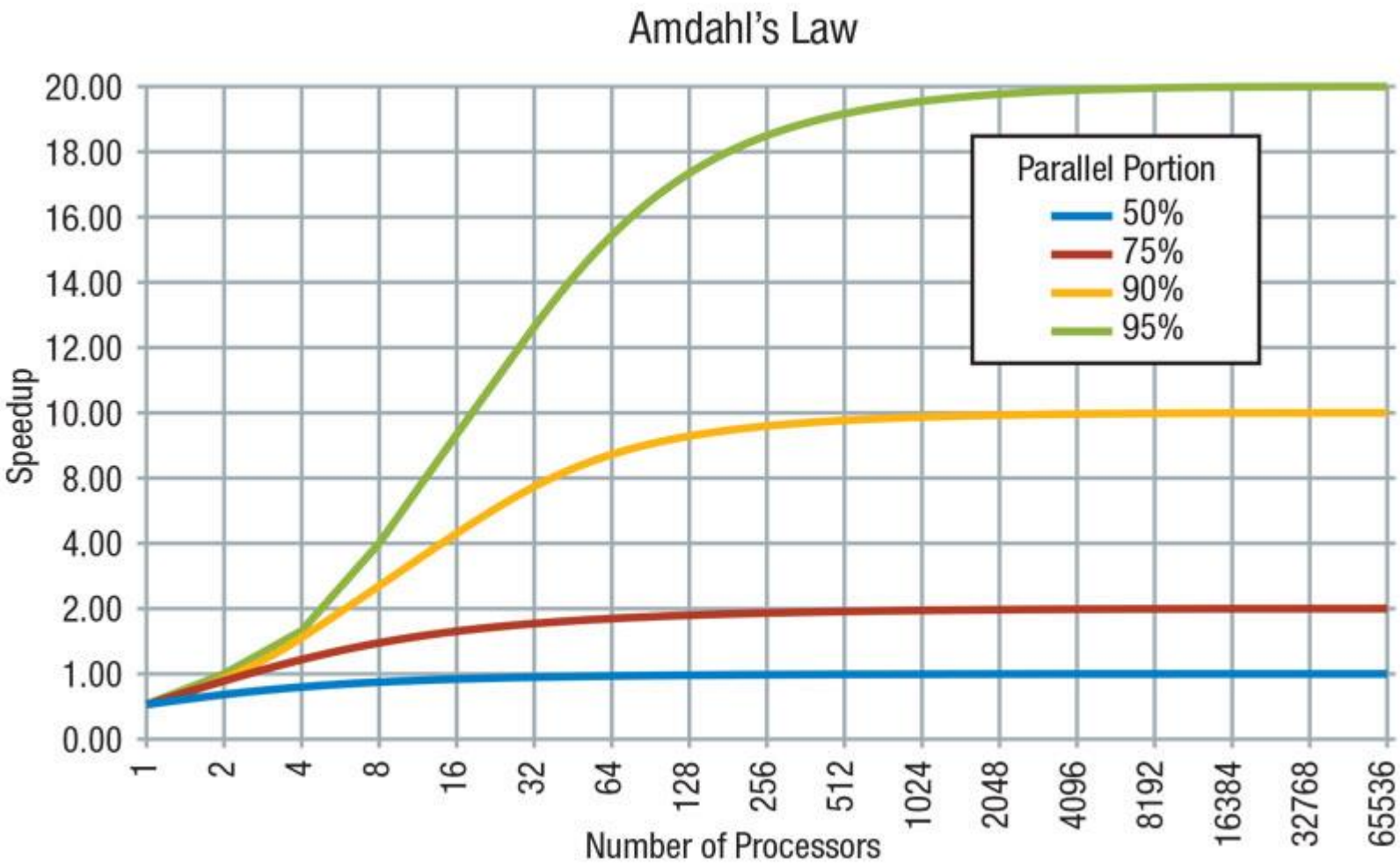
- New CPU is 10 times faster!
- 60% for I/O which remains almost the same…

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

$$= \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = \frac{1}{0.64} = 1.56$$

Apparently, its human nature to be attracted by 10X faster, vs. keeping in perspective its just 1.6X faster

Amdahl's Law: example



http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf

Aspect of CPU performance

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

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

	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 %)

$\overline{CPI} = 1.5$

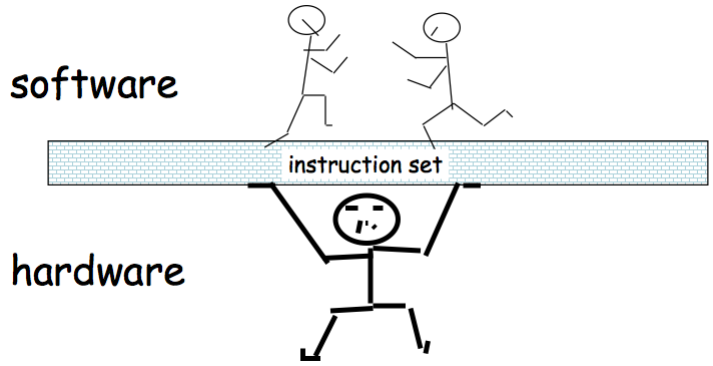
Invest resources where time is spent!

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



High level language code : C, C++, Java, Fortran,

compiler

Assembly language code: architecture specific statements

assembler

Machine language code: architecture specific bit patterns

software

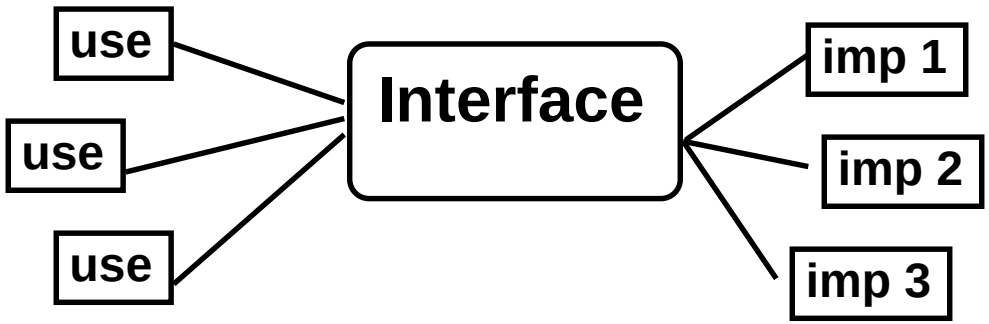
instruction set

hardware

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

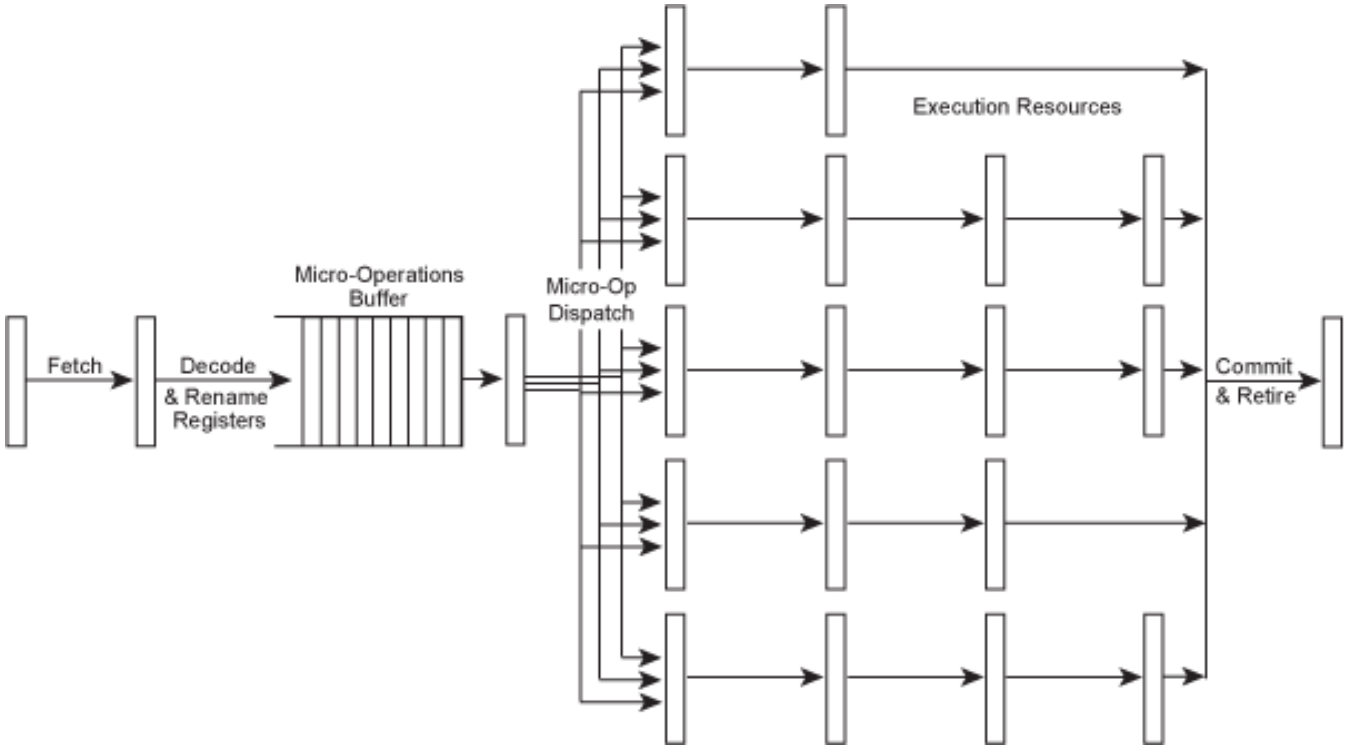


Example: RISC-CICS

- **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/risccisc/

Example: RISC-CICS



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

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

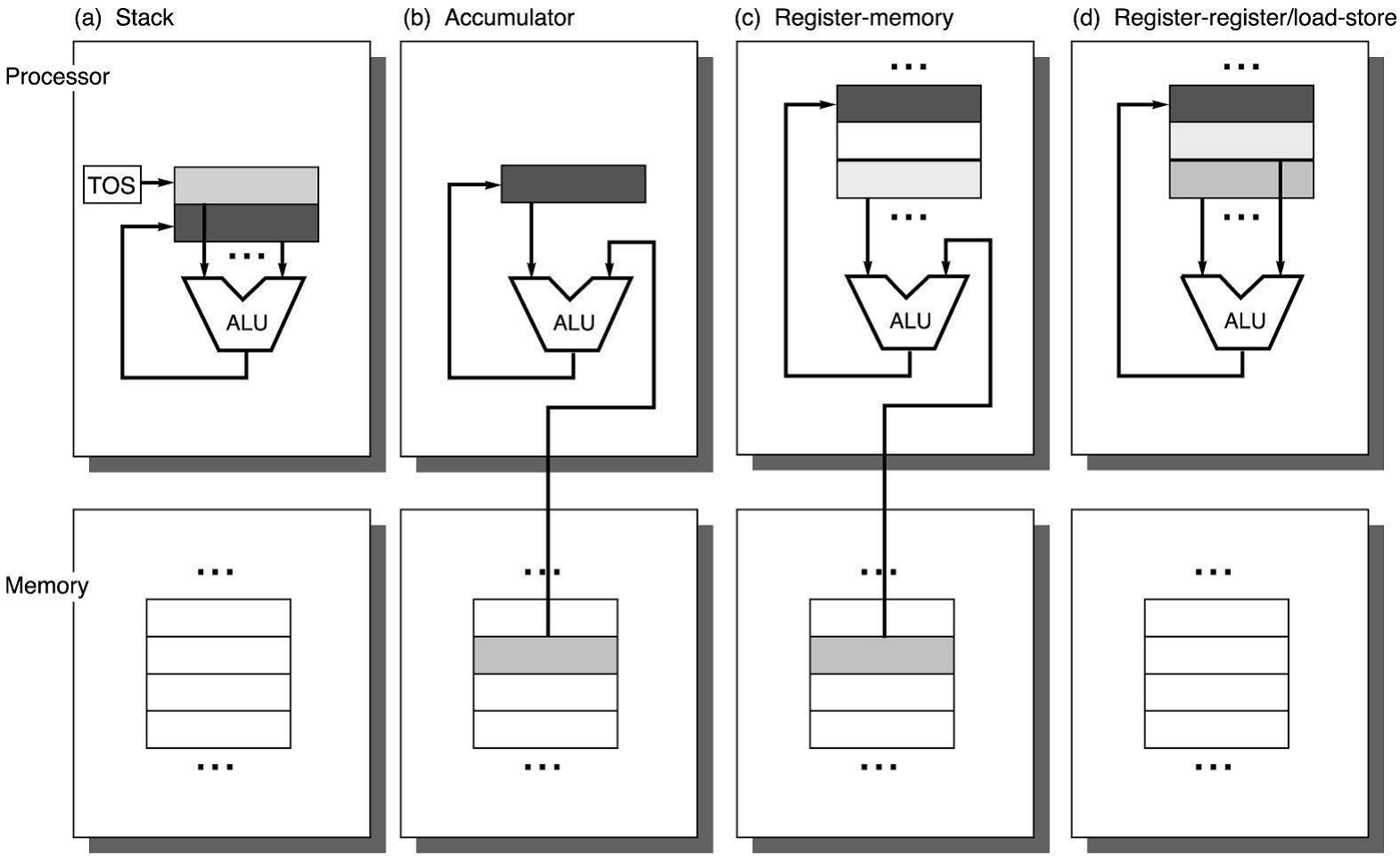
ISA Classification

- **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

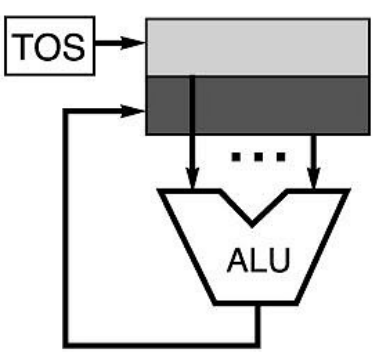
How are operands accessed by the CPU?

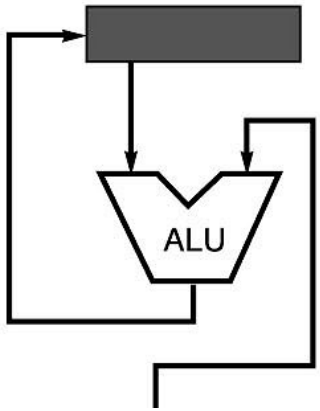
Temporary storage	Examples	Explicit operands	Result destination	Accessing operands
stack	B5500	0	stack	Push, Pop
Accumulator	PDP-8	1	Accumulator	Load/Store
Register (GPR)	IBM 360 VAX MIPS 80x86	2 or 3	Register or Memory	Load/Store or Reg/Mem

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

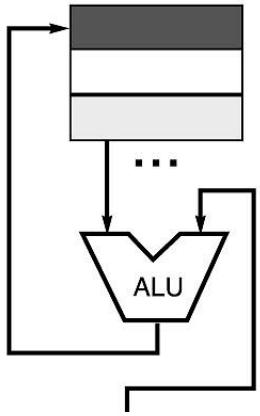
Example: C=A+B

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

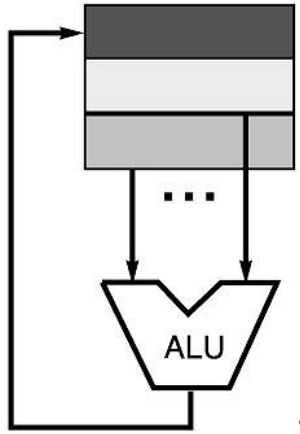




acc = acc + mem[C]



R1 = R1 + mem[C]



R3 = R1 + R2

ISA Classes

Accumulator (before 1960):

1-address	add A	acc ← acc + mem[A]

Stack (1960s to 1970s):

0-address	add	tos ← tos + next

Memory-Memory (1970s to 1980s):

2-address	add A, B	mem[A] ← mem[A] + mem[B]
3-address	add A, B, C	mem[A] ← mem[B] + mem[C]

Register-Memory (1970s to present, e.g. 80x86):

2-address	add R1, A	R1 ← R1 + mem[A]
	load R1, A	R1 ← mem[A]

Register-Register (Load/Store) (1960s to present, e.g. MIPS):

3-address	add R1, R2, R3	R1 ← R2 + R3
	load R1, R2	R1 ← mem[R2]
	store R1, R2	mem[R1] ← R2

Evolution of ISA

Single Accumulator (EDSAC 1950, Maurice Wilkes)

Accumulator + Index Registers
(Manchester Mark I, IBM 700 series 1953)

Separation of Programming Model from Implementation

Stack Architecture
(Burroughs Corporation B5000 1963)

Concept of a Family (RR,RM,MM)
(IBM 360 1964)

General Purpose Register Machines

Complex Instruction Sets
(Vax, Intel 432 1977-80)

CISC
Intel x86, Pentium

Load/Store Architecture
(CDC 6600, Cray 1 1963-76)

RISC
(MIPS,SUN- Sparc,HP-PA,
IBM RS6000,PowerPC . . .1987)

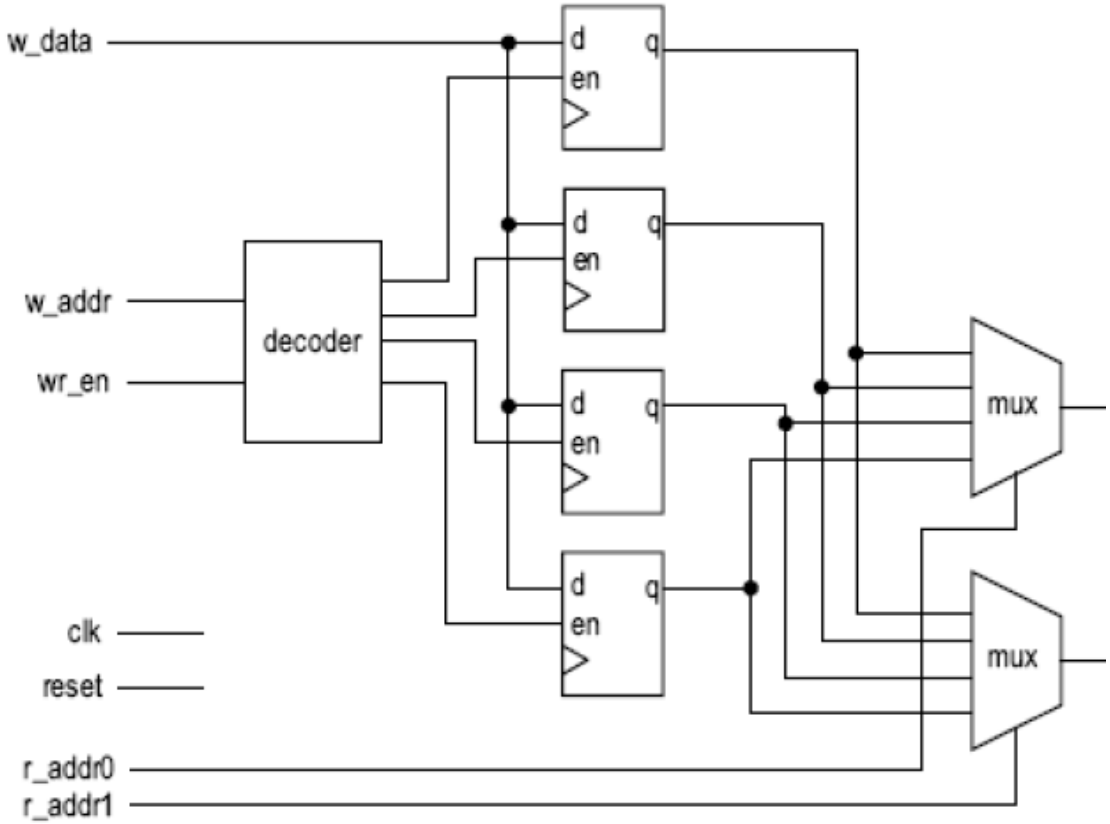
GPR (General Purpose Register)

- **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)

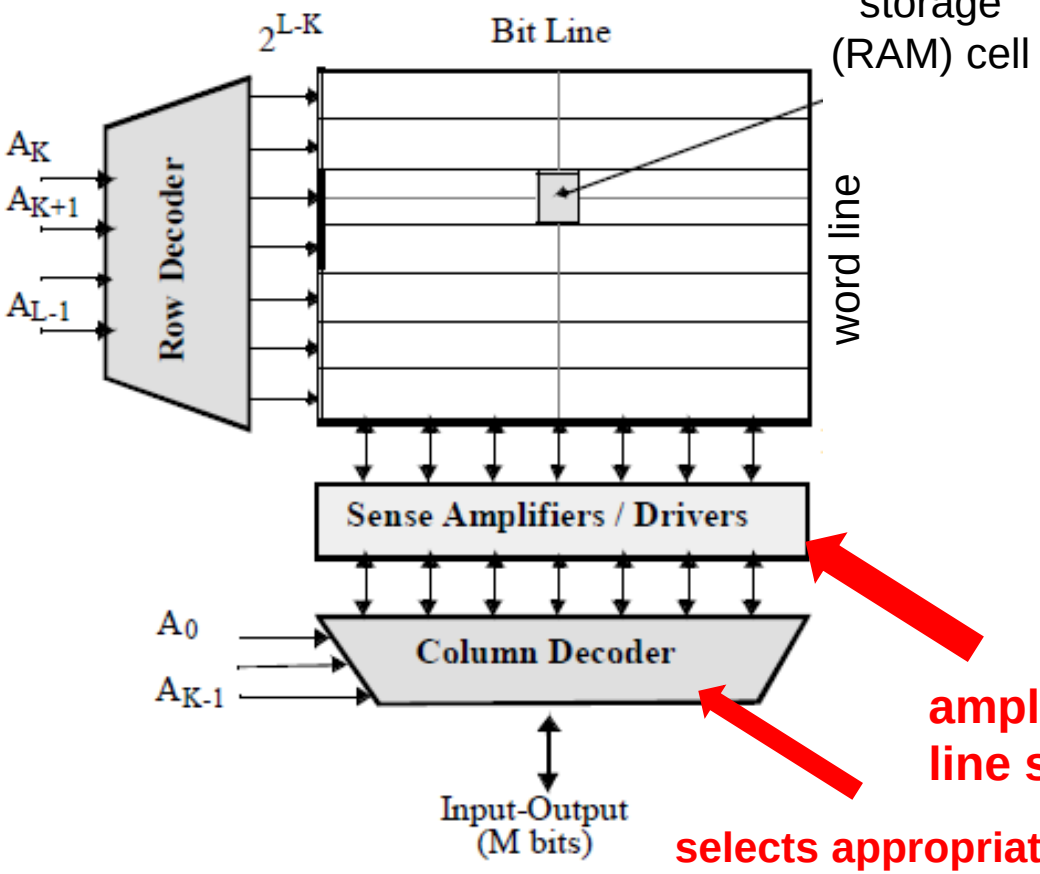
Register File

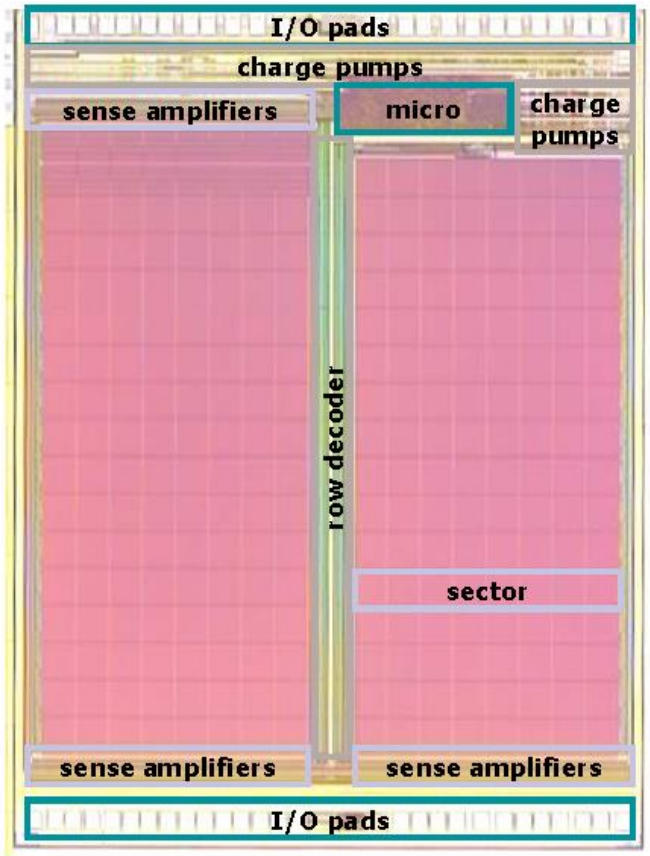
- **Example:** 4-word register file with 1 write port and two read ports

- **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



Memory Architecture





amplifies bit line swing

selects appropriate word from memory row

Reg v.s. Mem (65nm CMOS)

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

Stack v.s. GPR

Advantages

- Very compact object code

Push A	**Load R1,A**
Push B	**Load R2, B**
Add	**Add R3, R1, R2**

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

Disadvantages

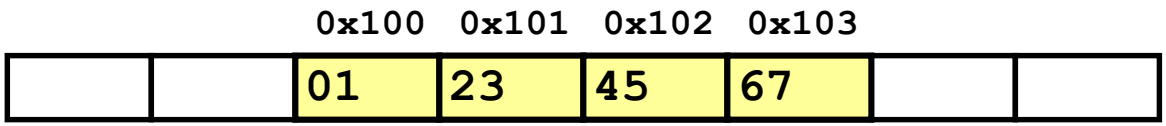
- More memory references (if stack is implemented with MEM)

X+1	**load X, push to memory** **load 1, push to memory** **pop 2 values from memory, add, and** **push result to memory**

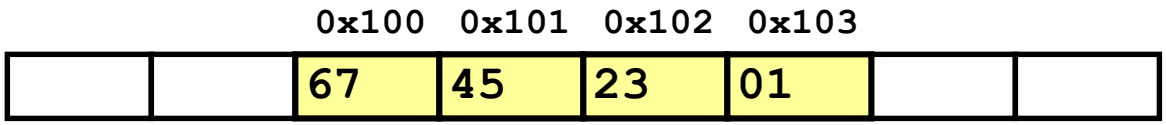
- Factoring out common subexpressions has high cost
- Can add some registers (hybrid)

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**

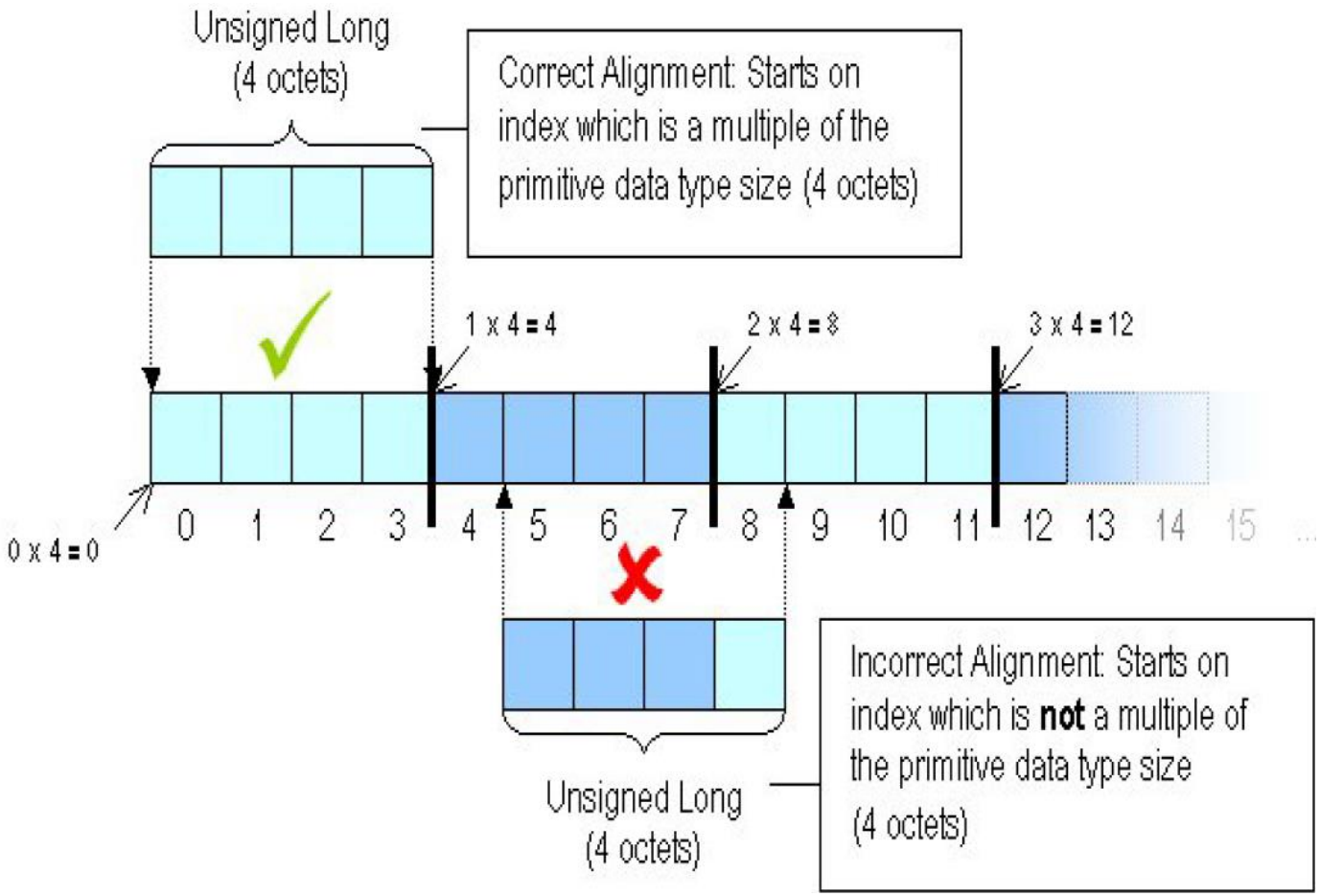


  - **Important for exchange of data**, (and strings)

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:**
  - Adreess index is multiple of date 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

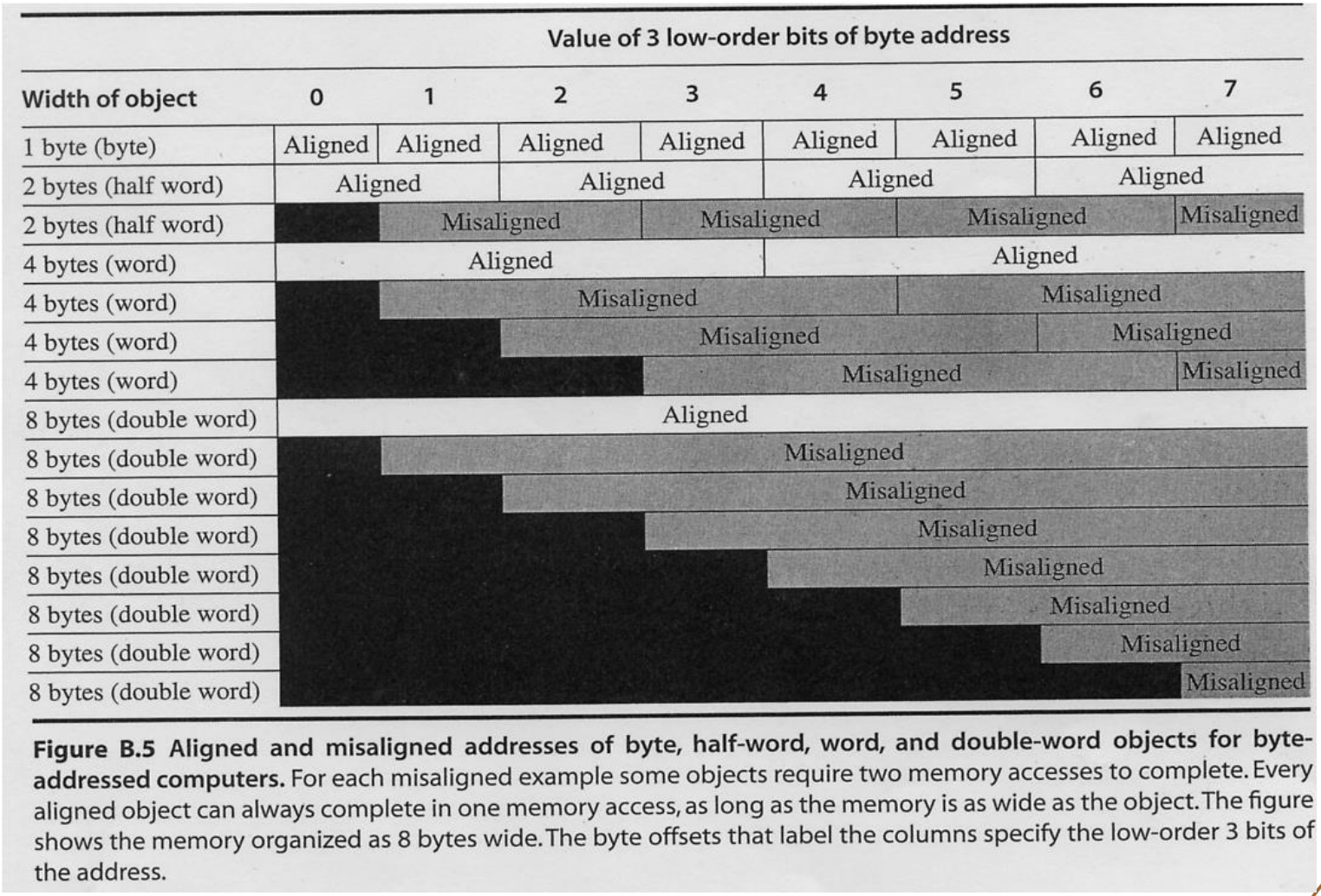


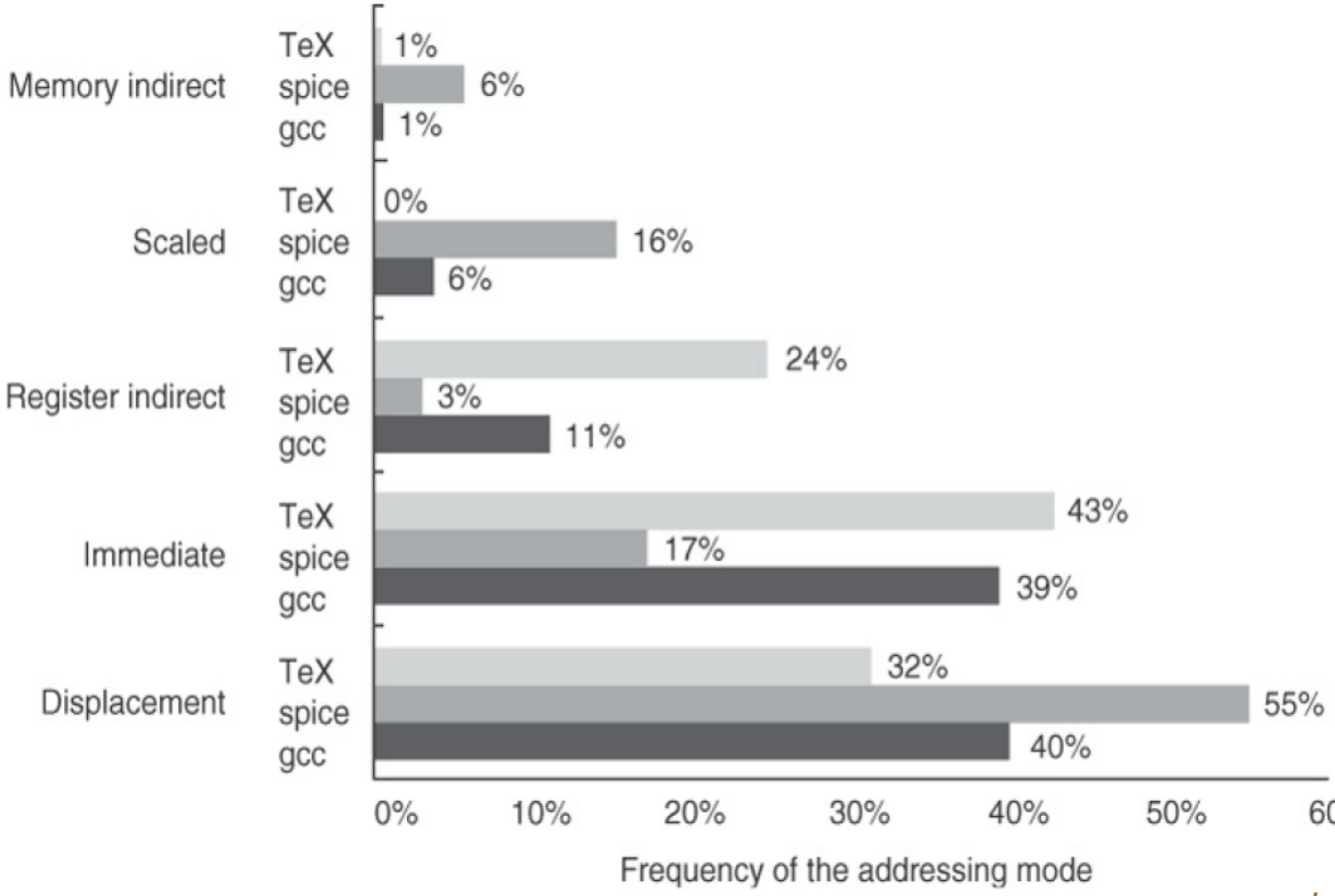
Figure B.5 Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide 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 <- R4 + R3
2. Immediate	Add R4, #3	R4 <- R4 + 3
3. Displacement	Add R4, 100(R1)	R4 <- R4 + M[100 + R1]
4. Register indirect	Add R4, (R1)	R4 <- R4 + M[R1]
5. Indexed	Add R4, (R1 + R2)	R4 <- R4 + M[R1 + R2]
6. Direct	Add R4, (1000)	R4 <- R4 + M[1000]
7. Memory Indirect	Add R4, @(R3)	R4 <- R4 + M[M[R3]]
8. Auto-increment	Add R4, (R2)+	R4 <- R4 + M[R2] R2 <- R2 + d
9. Auto-decrement	Add R4, (R2)-	R4 <- R4 + M[R2] R2 <- R2 - d
10. Scaled	Add R4, 100(R2)[R3]	R4 <- R4 + M[100 + R2 + R3*d]

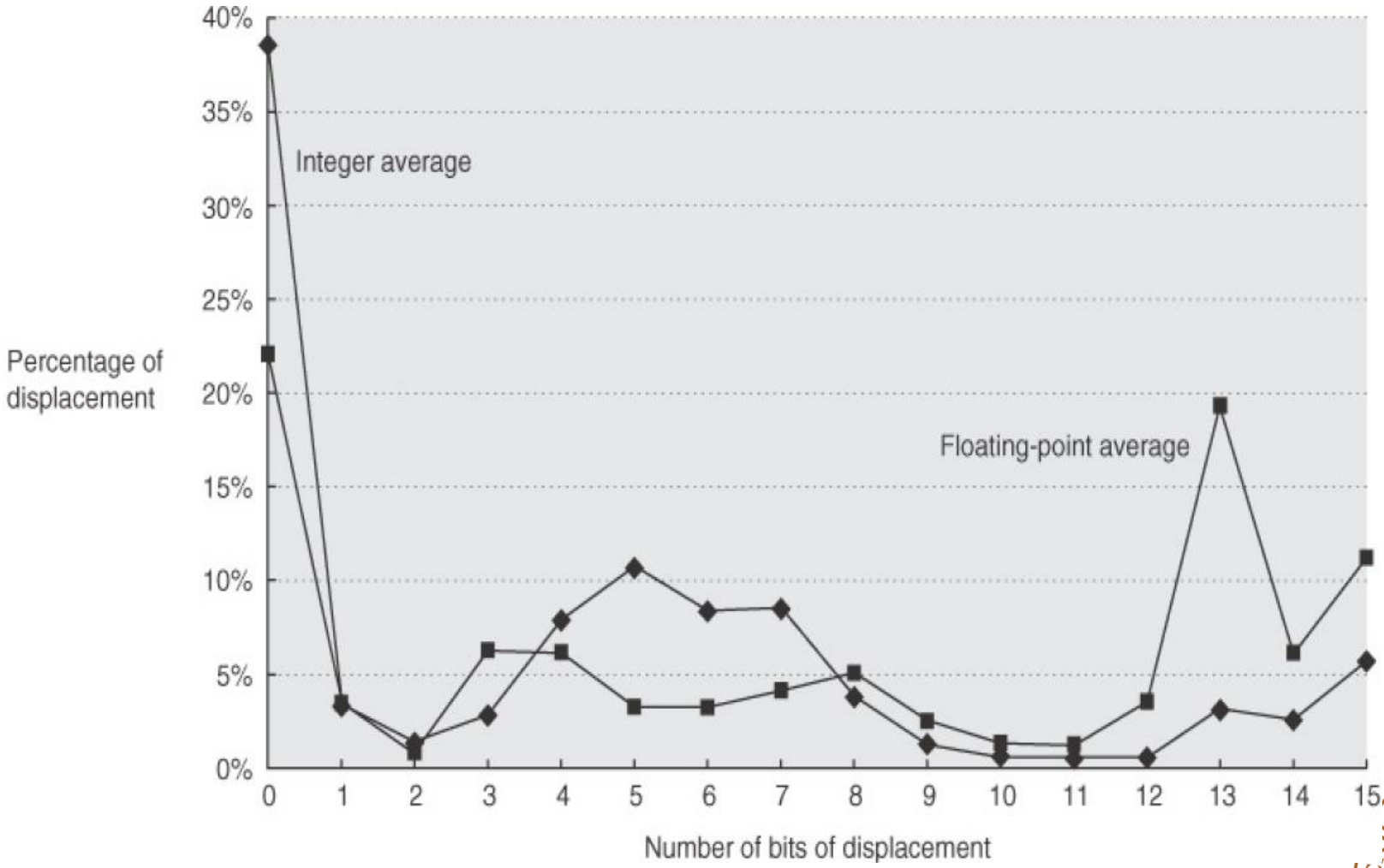
Memory addressing mode (one VAX)

- Are all these addressing modes needed?



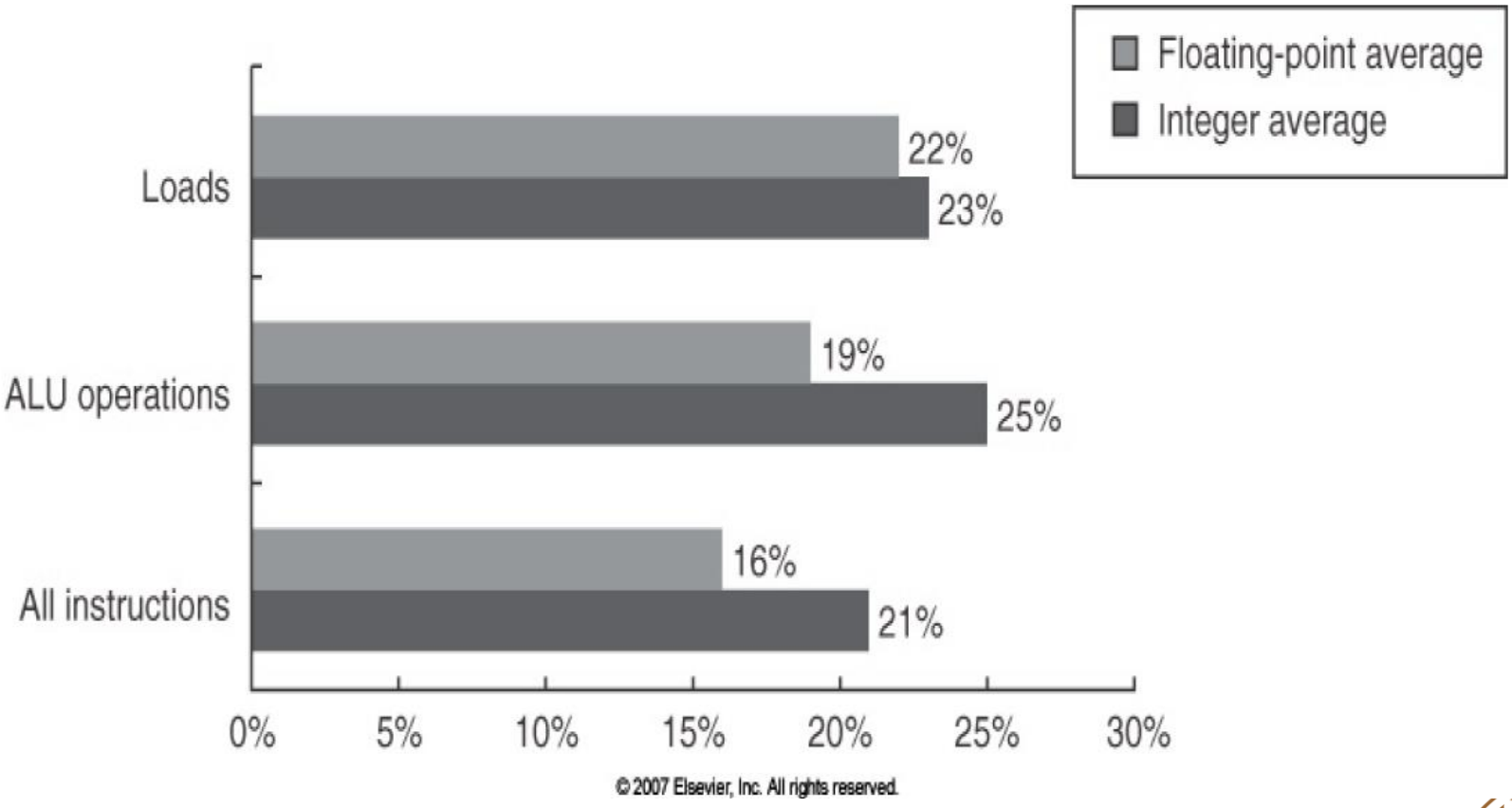
Memory Addressing Mode

- How many bits are needed for address displacement?



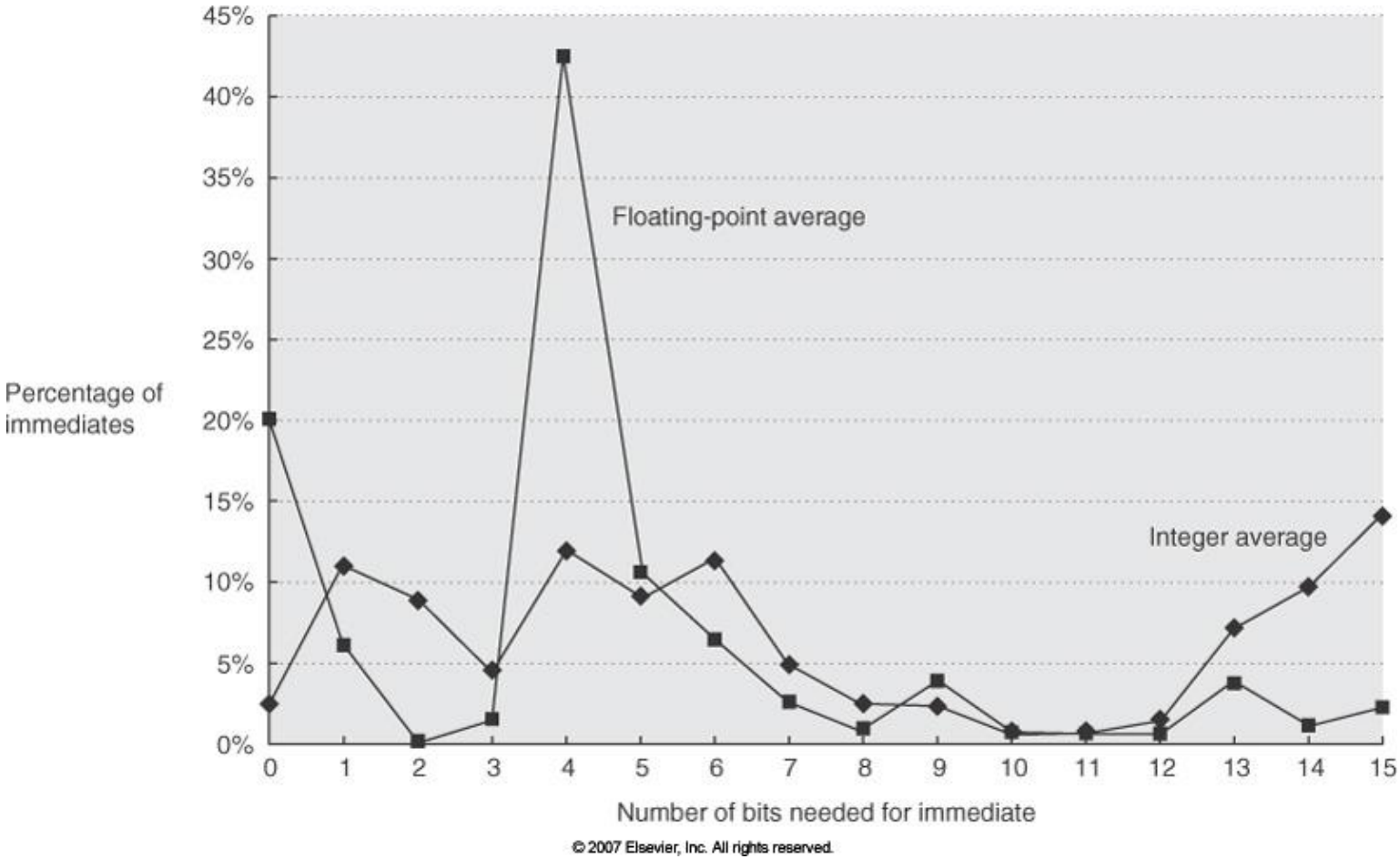
Memory Addressing Mode

- How important are immediats?



Memory Addressing Mode

- How many bits are needed for immediate?



What does it mean?

00100000010010000110100100100001

What does it mean?

00100000010010000110100100100001

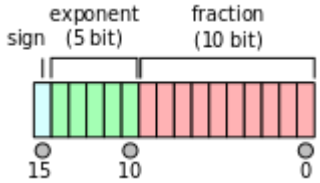
001000	00010	01000	0110100100100001
instruction	source	target	immediate
ADDI	2	8	26913

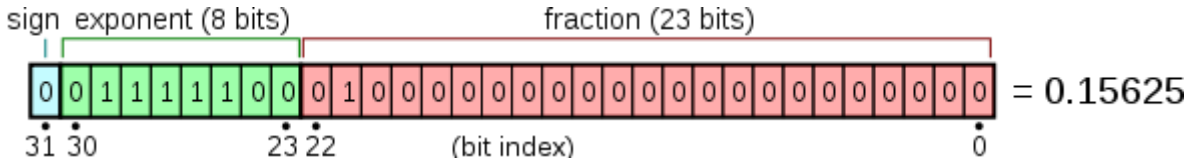
ADDI R8,R2,26913

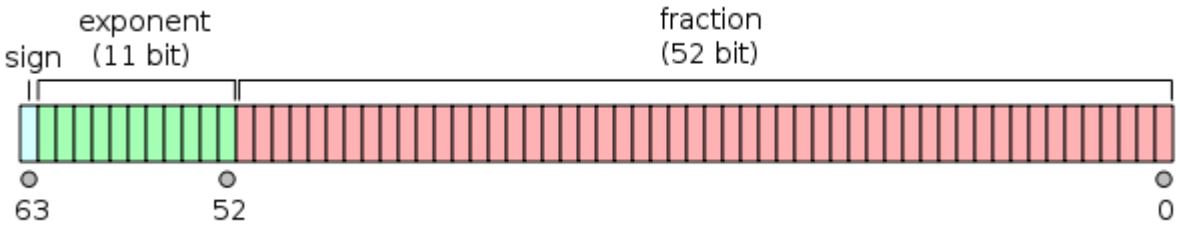
Types and sizes of operands

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

Floating point







Floating point v.s. fixed point

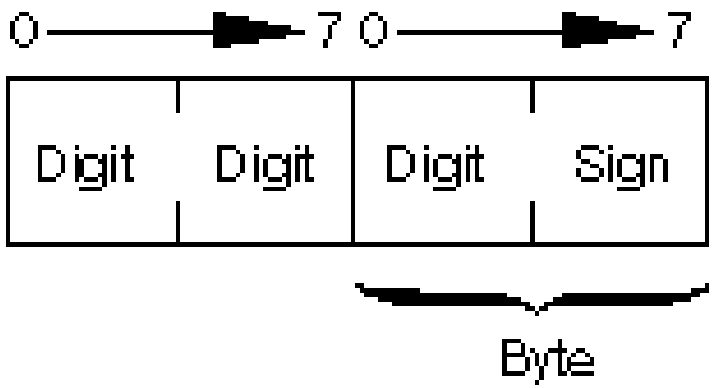
	32, 64bit Fixed-point (with 2, 4 pipeline stages)		32, 64bit Floating-point (with 6, 8 pipeline stages)		32, 64bit Floating-point (with 18, 23 pipeline stages)	
Area (slices)	36	139	293	693	504	1383
Max Freq. (MHz) achievable	250	250	140	130	250	200
Power (mW) at 100MHz	23.48	104	148.7	329	-	-

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)

Packed decimal



0.10_{10} $0.000110011001100110011\ldots_{2}$

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

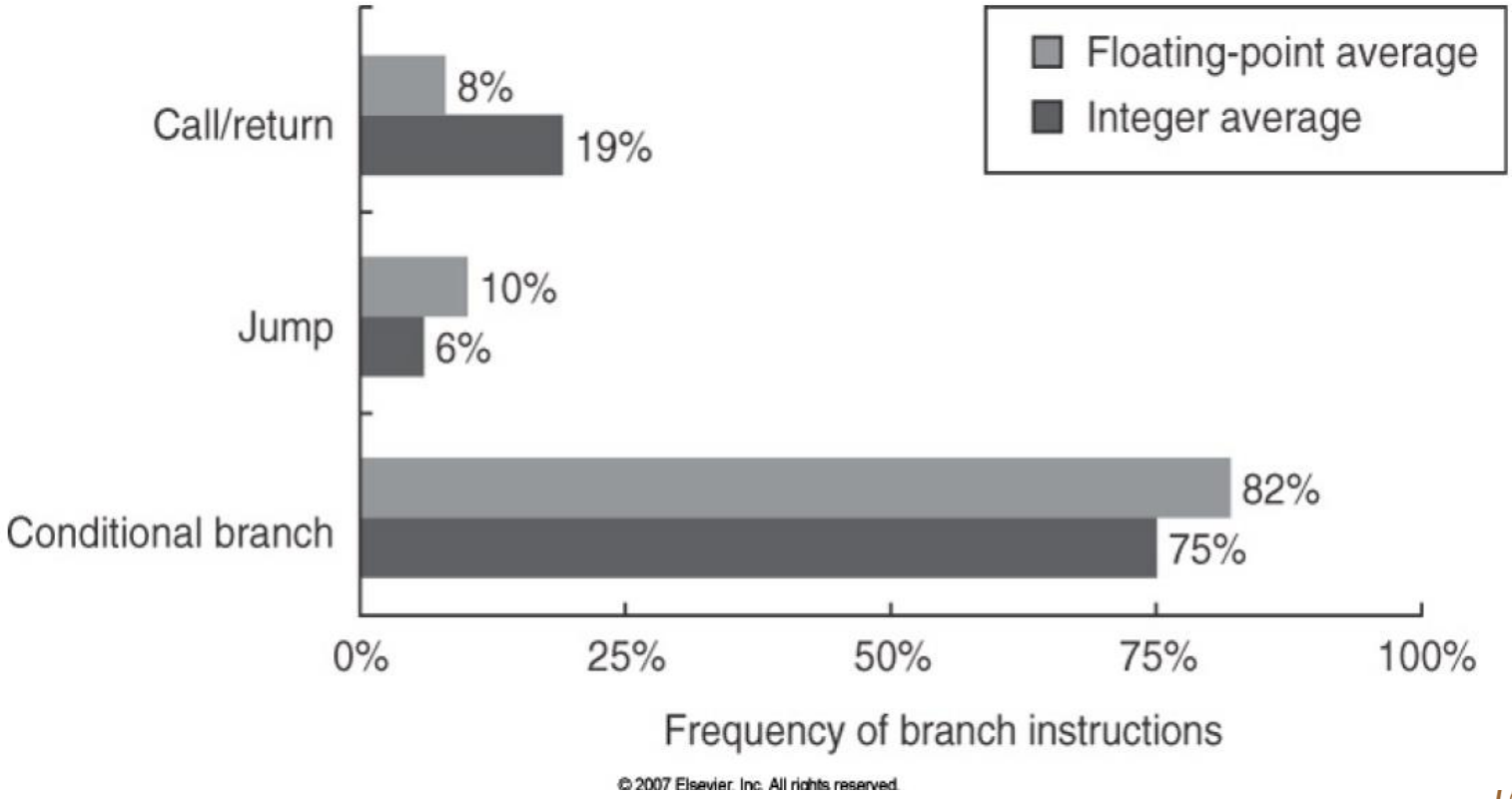
Types of operations (frequency)

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

80x86 Instruction Frequency

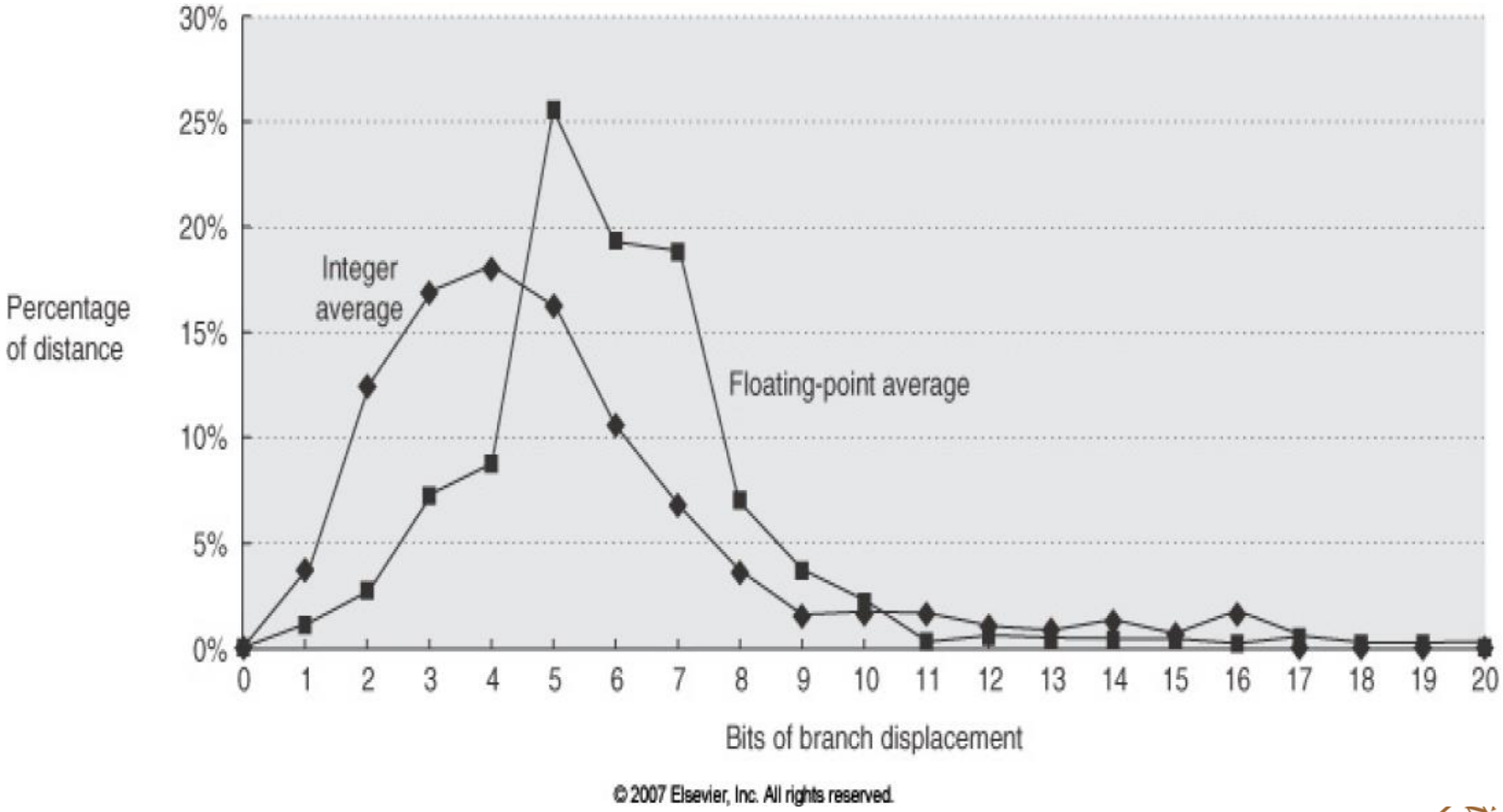
Types of control instructions

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



Types of control instructions

- How large need the branch displacement be?



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

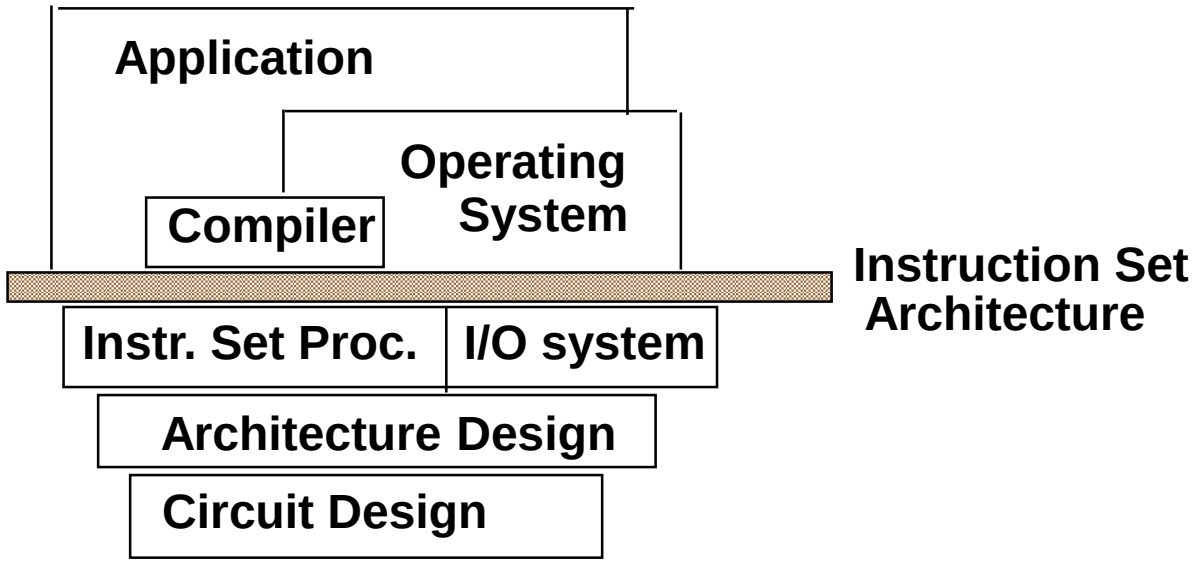
Operation	Address field 1	Address field 2	Address field 3

Outline

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

ISA and compiler

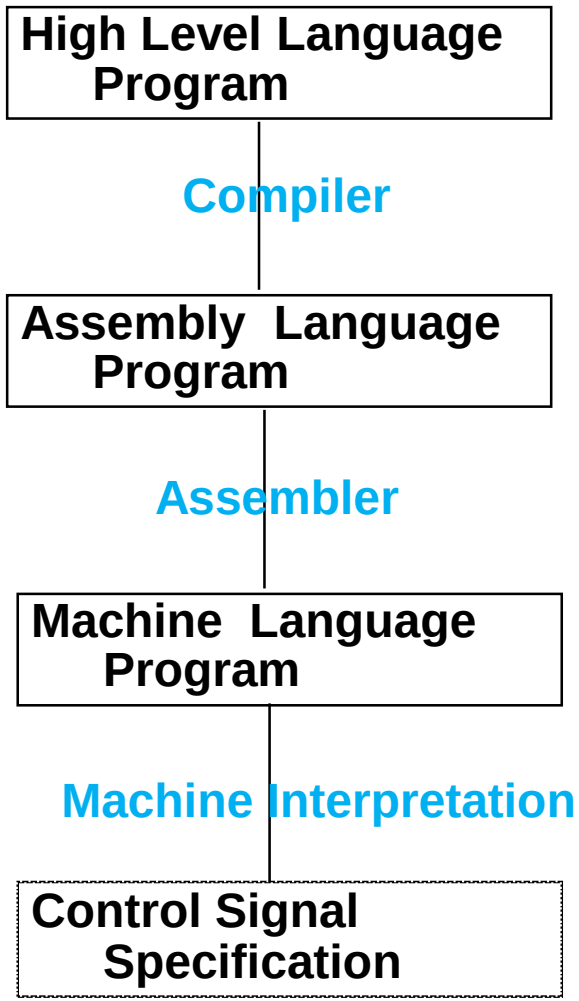
- Instruction set architecture is a compiler target
- By far most instructions executed are generated by a compiler (exception certain special purpose processors)
- Interaction compiler - ISA critical for overall performance



ISA and compiler

	VDPU					VCPU			vmacArray				VDEU			
21	Process prologue	Row 0	src	vArrayB	1	Reserved		0	Reserved			0	Reserved			0
20					0			0				0				0
19			opcode	Neg, i	0			0				0				0
18					1			0				0				0
17					0			0				0	Acc	en_r3	Disable	0
16		Row 1	src	vArrayB	1			0	VDDU	opcode	Complex	0		en_r2	Disable	0
15					0			0				0		en_r1	Disable	0
14			opcode	Neg, i	0			0		swap_b	Disable	0		en_r0	Disable	0
13					1	perm_en	Enable	1		swap_a	Disable	0		reg_acc_r3	Direct	0
12					0	perm_imm	Row 0	0	VMAC array	mul_en	Enable	1		reg_acc_r2	Direct	0
11		Row 2	src	vArrayB	1			0		mul_sign_out	Unsigned	1		reg_acc_r1	Direct	0
10					0		Row 0	0		add_en_l1	Enable	1		reg_acc_r0	Direct	0
9			opcode	Neg, i	0			0		add_en_l2	Enable	1	Sum	mux_in_r23	Straight	0
8					1		Row 3	1		add_sign_l1	Signed	0		mux_in_r01	Straight	0
7					0			1		add_sign_l2	Signed	0		mux_out_r23	acc out	0
6		Row 3	src	vArrayB	1		Row 3	1		add_ctrl_l1a	Sub	1		mux_out_r01	acc out	0
5					0			1		add_ctrl_l1b	Add	0	Barral shift	opcode	ASR	0
4			opcode	Neg, i	0	swap_en	Disable	0		add_ctrl_l2a	Sub	1				1
3					1	opcode	None	0		add_ctrl_l2b	Sub	1		shift	1	0
2					0			0		op_coeff	Const mul	0				0
1	Mask		src	None	0			0				1				0
0					0	mask_en	Disable	0		coeff_shape	Column	0				1
Inst [HEX]	00252948					000021E0			00001E5A				00000011			

The role of compilers



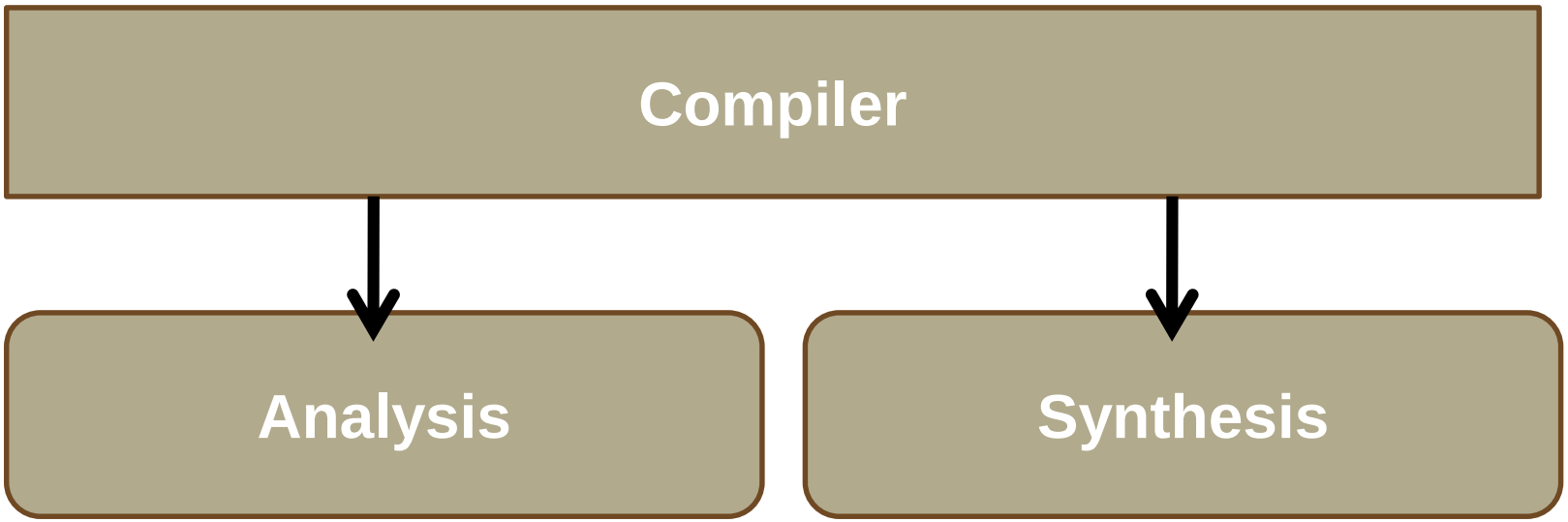

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

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

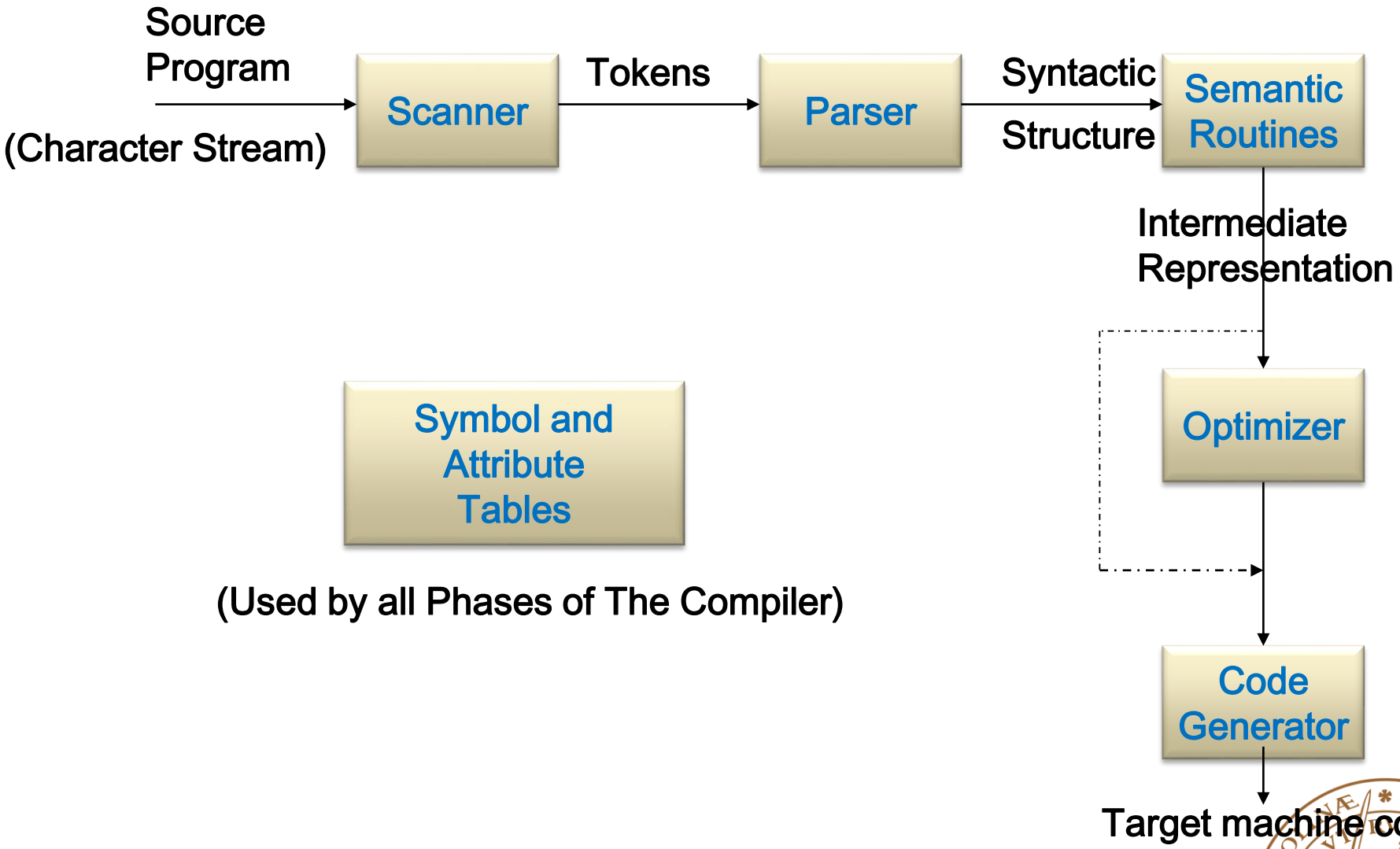
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

The structure of a compiler

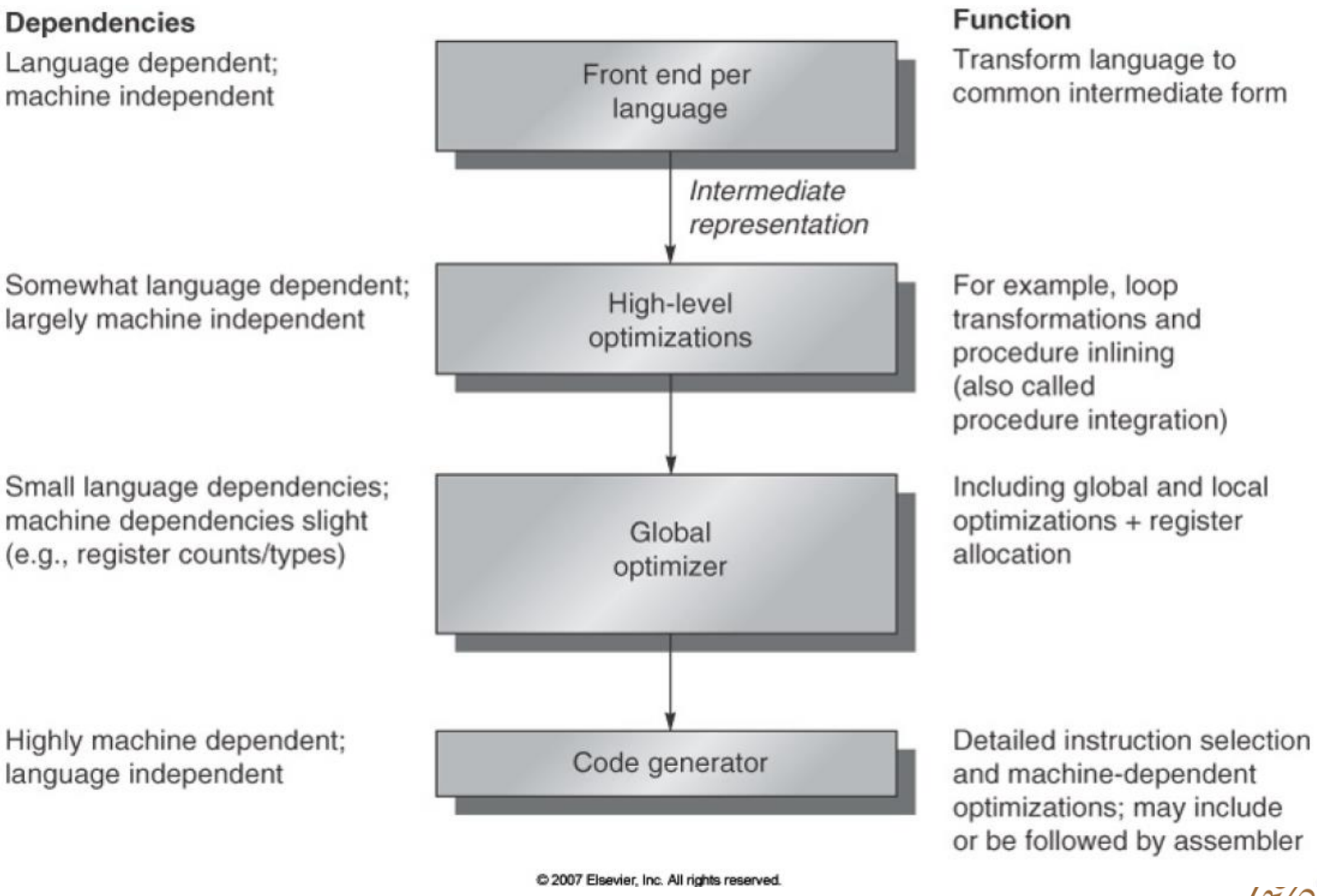
- **Any compiler must perform two major tasks**
  - Analysis of the source program
  - Synthesis of a machine-language program



The structure of a compiler



The structure of a compiler



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**

GCC optimization options (individual opt.)

- Four *gcc* optimizations, all optimizations applied on top -O1
- *-fschedule-insns* – local register allocation followed by basic-block list scheduling
- *-fschedule-insns2* – Postpass scheduling done
- *-finline-functions* – Integrated all simple functions into their callers
- *-funroll-loops* – Perform the optimization of loop unrolling

-falign-functions -falign-jumps -falign-labels -falign-loops -fassociative-math -fauto-inc-dec -fbranch-probabilities

-fbranch-target-load-optimize -fbranch-target-load-optimize2 -fbtr-bb-exclusive -fcaller-saves -fcheck-data-deps -fcprop-registers

-fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fcx-limited-range -fdata-sections -fdce -fdce -fdelayed-branch

-fdelete-null-pointer-checks -fdse -fdse -fearly-inlining -fexpensive-optimizations -ffast-math -ffinite-math-only -ffloat-store

-fforward-propagate -ffunction-sections -fgcse -fgcse-after-reload -fgcse-las -fgcse-lm -fgcse-sm -fif-conversion -fif-conversion2

-finline-functions -finline-functions-called-once -finline-limit -finline-small-functions -fipa-cp -fipa-matrix-reorg -fipa-pta

-fipa-pure-const -fipa-reference -fipa-struct-reorg -fipa-type-escape -fivopts -fkeep-inline-functions -fkeep-static-consts

-fmerge-all-constants -fmerge-constants -fmodulo-sched -fmodulo-sched-allow-regmoves -fmove-loop-invariants -fmudflap

-fmudflapir -fmudflapth -fno-branch-count-reg -fno-default-inline -fno-defer-pop -fno-function-cse -fno-guess-branch-probability

-fno-inline -fno-math-errno -fno-peephole -fno-peephole2 -fno-sched-interblock -fno-sched-spec -fno-signed-zeros

-fno-toplevel-reorder -fno-trapping-math -fno-zero-initialized-in-bss -fomit-frame-pointer -foptimize-register-move

-foptimize-sibling-calls -fpeel-loops -fpredictive-commoning -fprefetch-loop-arrays -fprofile-generate -fprofile-use -fprofile-values

-freciprocal-math -fregmove -frename-registers -freorder-blocks -freorder-blocks-and-partition -freorder-functions

-frerun-cse-after-loop -freschedule-modulo-scheduled-loops -frounding-math -frtl-abstract-sequences -fsched2-use-superblocks

-fsched2-use-traces -fsched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns-dep -fsched-stalled-insns

-fschedule-insns -fschedule-insns2 -fsection-anchors -fsee -fsignaling-nans -fsingle-precision-constant -fsplit-ivs-in-unroller

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;
}
```

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

Procedure inlining

```
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 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

```
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

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

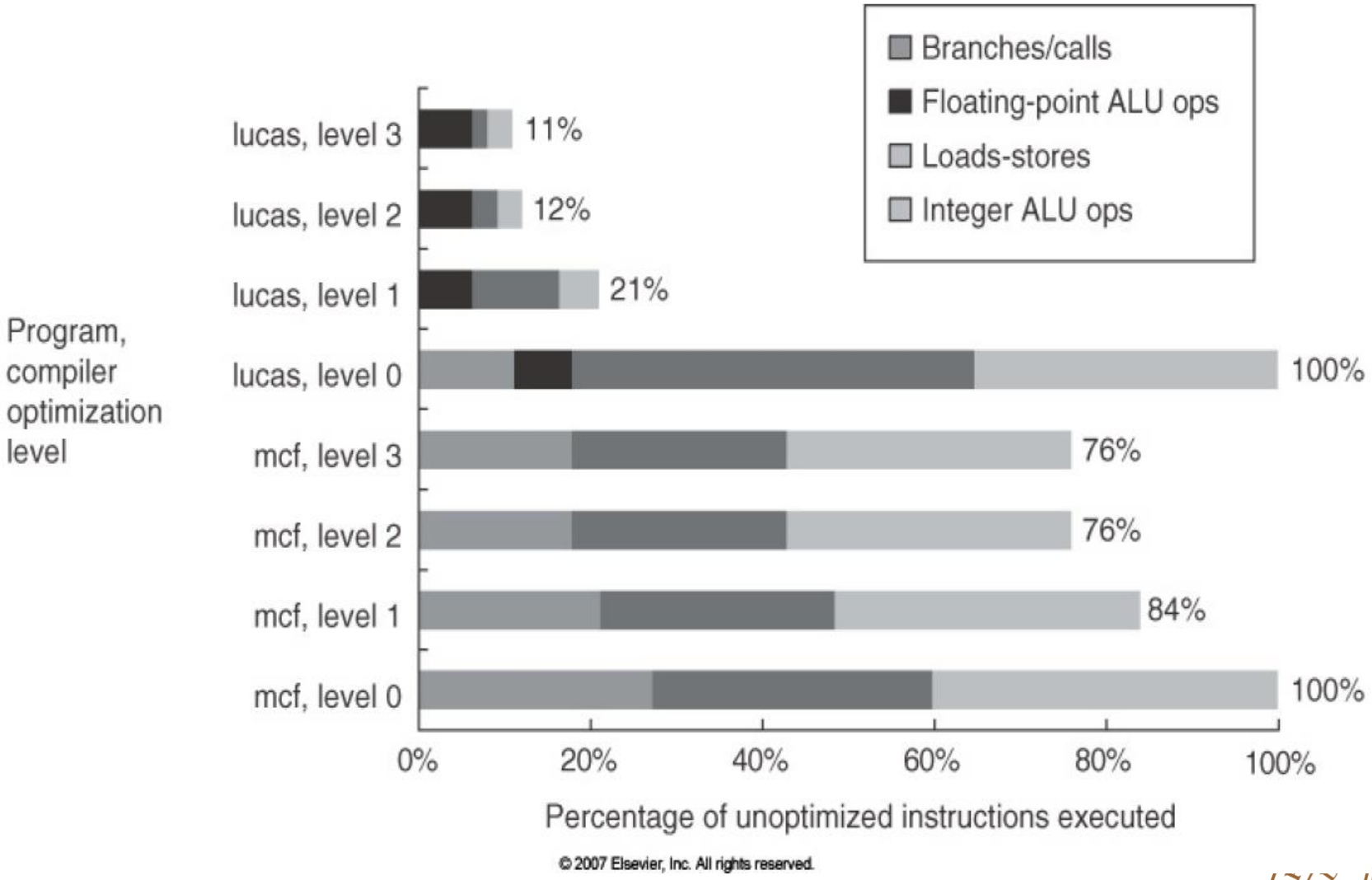
```
tmp = b * c;
a = tmp + g;
d = tmp * e;
```

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



How can you aid compiler

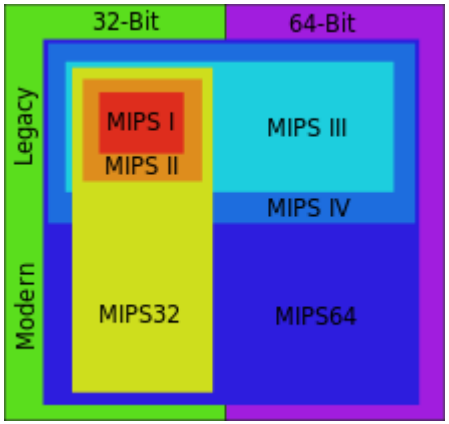
- **Rules of thumb when designing an instruction set (for general purpose processor):**
  - Regularity (operations, data types and addressing modes should be orthogonal)
  - Generality. Provide primitives, not high-level constructs or solutions. Complex instructions are often too specialized.
  - Simplify trade-offs among alternatives
  - Provide instructions that bind quantities known at compile time as constants

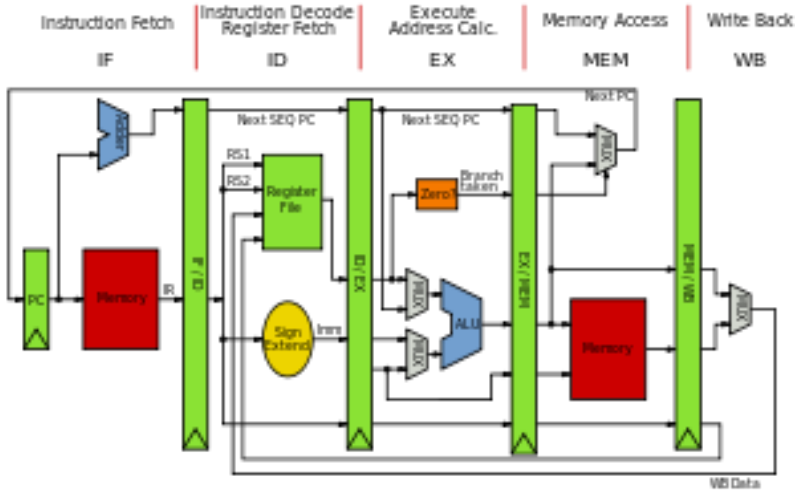
Outline

- 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 ← rs 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 ← rs 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

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**