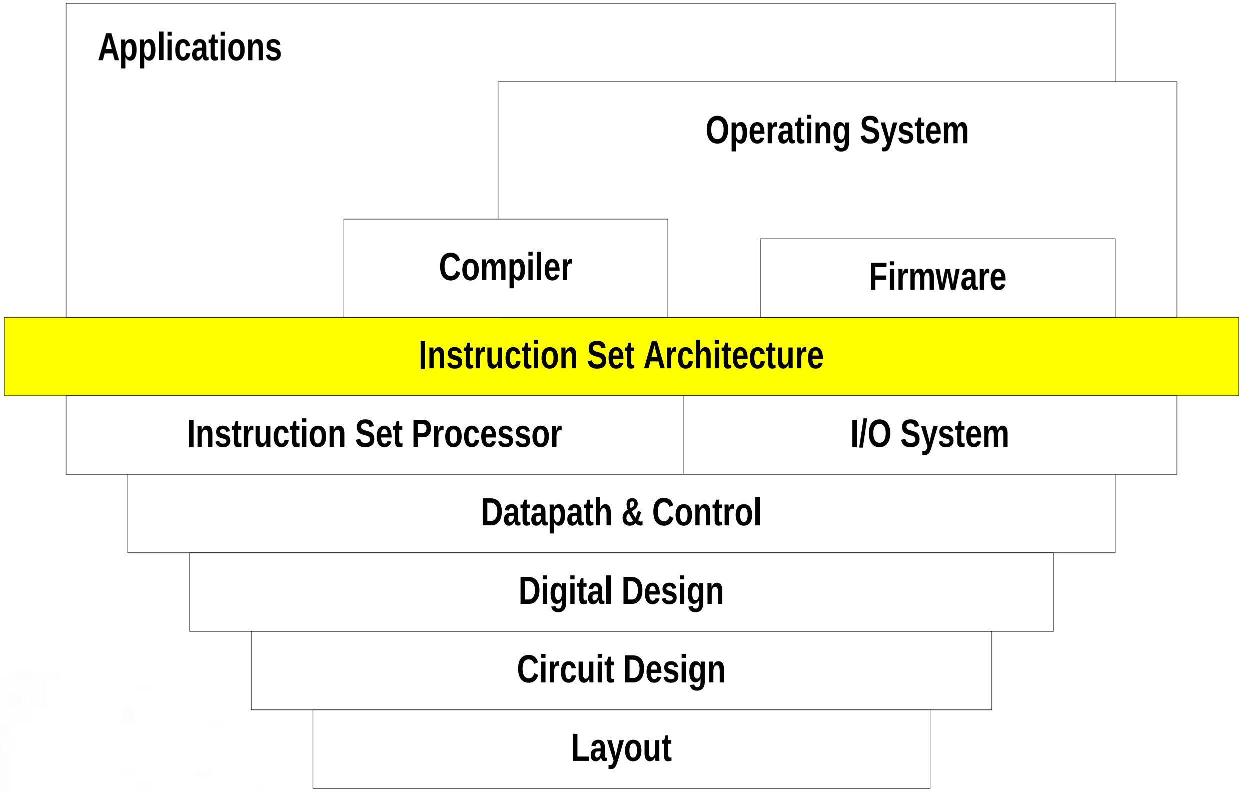
浙江大学
ZheJiang University

Computer Architecture ----A Quantitative Approach

College of Compute of Zhejiang University
CHEN WEN ZHI
chenwz@zju.edu.cn
Room 511, CaoGuangBiao BLD

Instruction Set Architecture



Applications

Operating System

Compiler

Firmware

Instruction Set Architecture

Instruction Set Processor

I/O System

Datapath & Control

Digital Design

Circuit Design

Layout

Assembly Language

|||

Instruction Set Architecture

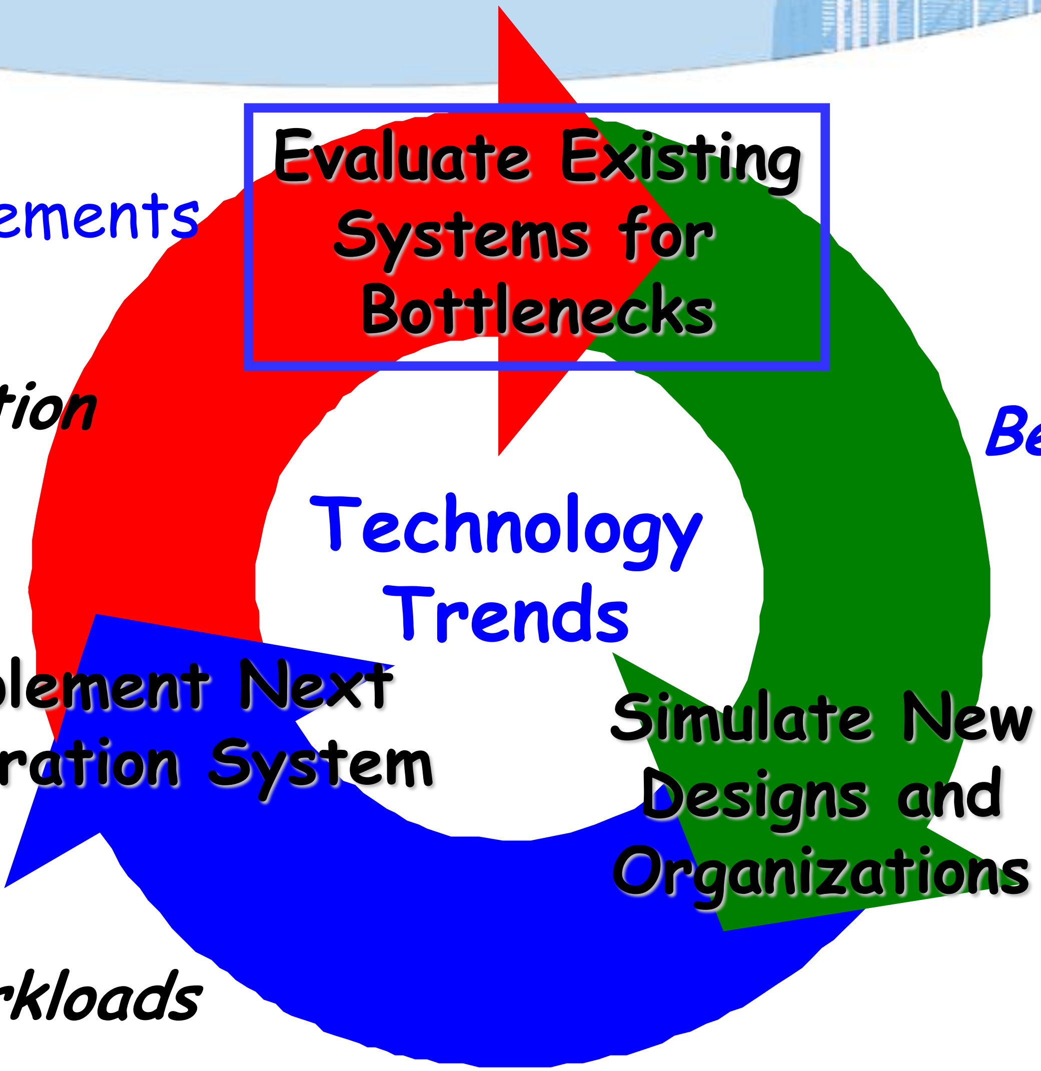
|||

Machine Language

Instruction Set Design Tasks

- Classifying Instruction Set Architectures
- Memory Addressing
- Operations in the Instruction Set
- Type and Size of Operands
- Encoding an Instruction Set
- Optimizing an Instruction Set

Recall: The Design Engineering



Evaluate Existing Systems for Bottlenecks

Simulate New Designs and Organizations

Implement Next Generation System

Technology Trends

Requirements

Implementation Complexity

Benchmarks

Quantitative principle

Workloads

Important step for ISA design

- To analyze and evaluate the existing machines with a large collection of programs before making architectural decisions.
- Compare with research/graduate project
  - reading a large amount of materials in the area
  - evaluating or classifying the existing methods
  - make your focus and your work plan
  - implement your ideas
  - write the report : summary of your work

Recall: Three application area

- Desktop Computing
  - emphasizes performance of programs with integer and floating-point data types, with little regard for program size of processor power consumption
  - integer /floating-point programs
- Servers
  - used primarily for databases, file server and Web applications, plus some time-sharing applications for many users.
  - Time-sharing applications for many users
  - FP performance is less important than that of integer/strings
- Embedded Applications
  - value cost and power, so code size is important because less memory is both cheaper and lower power
  - code size

2.2 Classifying Instruction Set Architectures

The type of internal storage in CPU

- stack
  - The operands are implicitly on the top of the stack :B5000
- accumulator
  - One operand is implicitly the accumulator : PDP-8
- GPR(General-Purpose Register) architecture
  - Have only explicit operands-either registers or memory locations
  - 1975-now all machines use general purpose registers

Three general types of GPR

Max number of operands in ALU instruction.

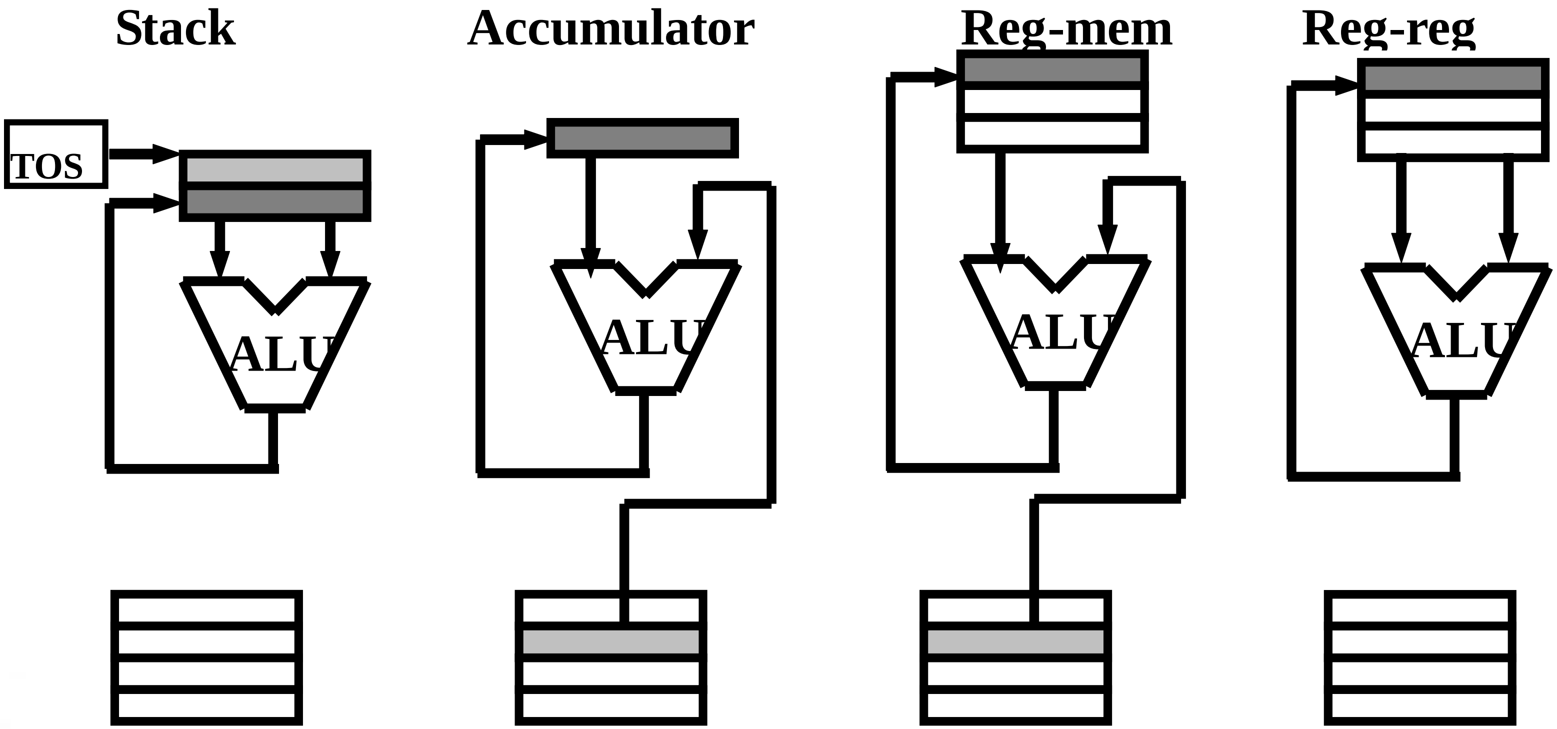
Total memory-address operands in ALU instruction

- Register-Register (0) ------ Load/Store
  - **Data must be explicitly moved between registers and memory.**
  - **ALU operations use register operands only.**
  - **Usually 3 operands, all in registers.**
- Register-Memory (1)
  - **Operations occur between register and memory (one operand in memory).**
  - **Usually 2 operands, one in a register (src and dest) and one in memory (src only).**
- Memory-Memory (2~3)
  - **May have 2 or 3 operands in memory (VAX).**

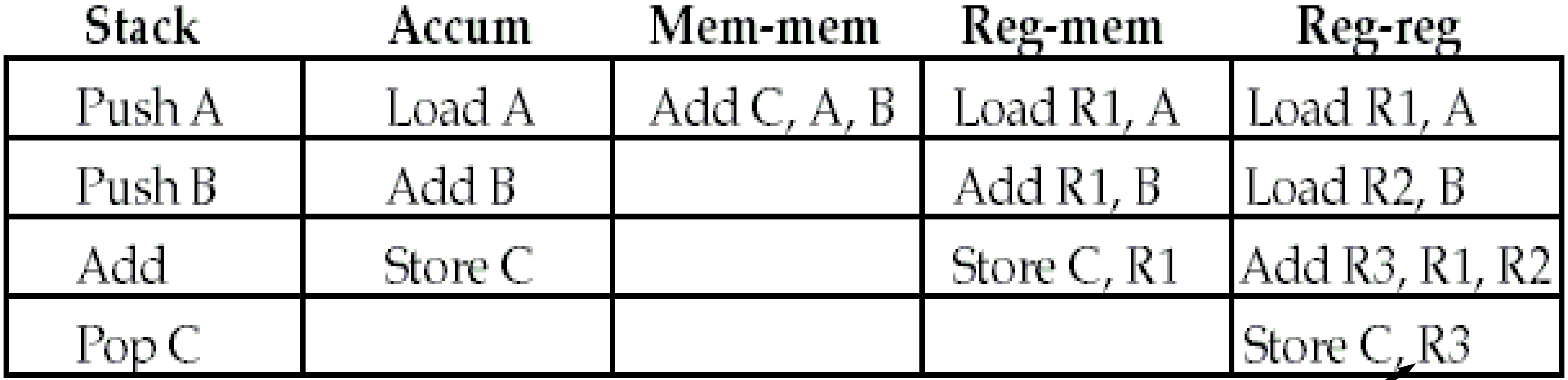
Examples of Computers

Number of Memory addresses	Maximum num. of operands allowed	Type of Architecture	Examples
0	3	Load-store	Alpha, ARM, MIPS, PowerPC, SPARC, superH, TM
1	2	Reg-Mem	IBM360/370, Inter80x86,Ti TM Motorola 6800,
2	2	Mem-Mem	Vas(aoso has three-oprands formats)
3	3	Mem-Mem	Vas(aoso has three-oprands formats)

Operand location for 4 ISA classes



Code sequence of C=A+B



Stack	Accum	Mem-mem	Reg-mem	Reg-reg
Push A	Load A	Add C, A, B	Load R1, A	Load R1, A
Push B	Add B		Add R1, B	Load R2, B
Add	Store C		Store C, R1	Add R3, R1, R2
Pop C				Store C, R3

MIPS is one of these: this is what we'll be learning

Why are GPR ISAs so popular ?

- registers are faster than memory
  - memory traffic is reduced, so program is speed up (since registers are faster than memory)
- registers can hold variables
  - registers are easier for a compiler to use: e.g., (A*B) – (C*D) –(E*F) can do multiplies in any order vs. stack
  - code density improves (since register named with fewer bits than memory location)

ISA metrics

- Code density :
  - How much space does a program require ?
- Instruction count :
  - How many instructions are necessary for a specific task ?
- Instruction complexity :
  - How much decoding is necessary to interpret an instruction ?
- Instruction length :
  - Is length dependent on the type of instruction and addressing mode ?
- Other metrics
  - Encoding Complexity, CPI

Pros and Cons of the three GPR computers

Metrics	Reg-Reg	Reg-mem	Mem-mem
Code density	lowest	Higher	Highest
Instruction count	Largest	Large	small
Instruction complexity	Simplest	Complex	most complex
Instruction length	Fixed	variable	Large variation
Encoding complexity	Fixed,	Hybrid	Variable
CPI	small	middle	Large variation

- Computers with fewer alternatives simplify the compiler's task.
- The number of registers also affects the instruction size.

2.3 Memory Addressing

- How memory addresses are interpreted ?
- How the memory addresses are specified ?

Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- Can be addressed in
  - **Word: Easy to implement, not support for non-numerical computing**
  - **Bit: variable length computing, waste of address space**
  - **Byte: Most popular, exists data storage and align problems**
    - "Byte addressing" means that the index points to a byte of memory.

Addressed in Words or Bytes

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

Byte addresses
of words
in mem

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
...	

- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
- Words are aligned

Little Endian vs. Big Endian

- Two different conventions for ordering the bytes within a larger object

Little Endian
(Intel)

7	6	5	4	3	2	1	0

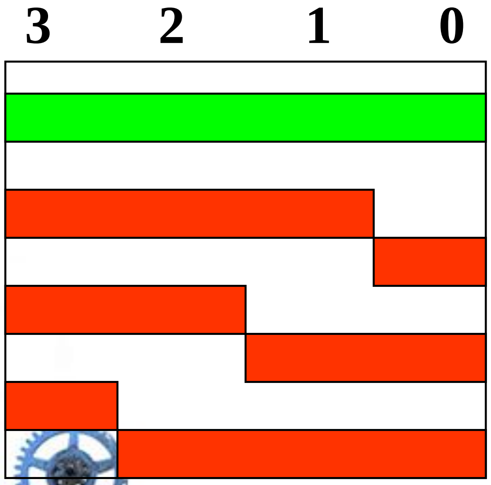
Big Endian
(IBM.Motorola)

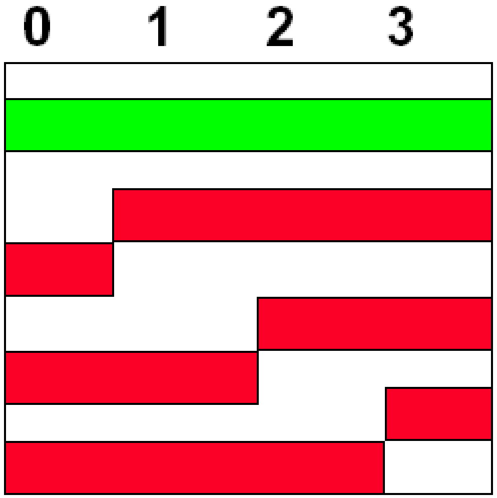
0	1	2	3	4	5	6	7

Aligned Memory Access

- Aligned address of byte, half-word, word, and double-word

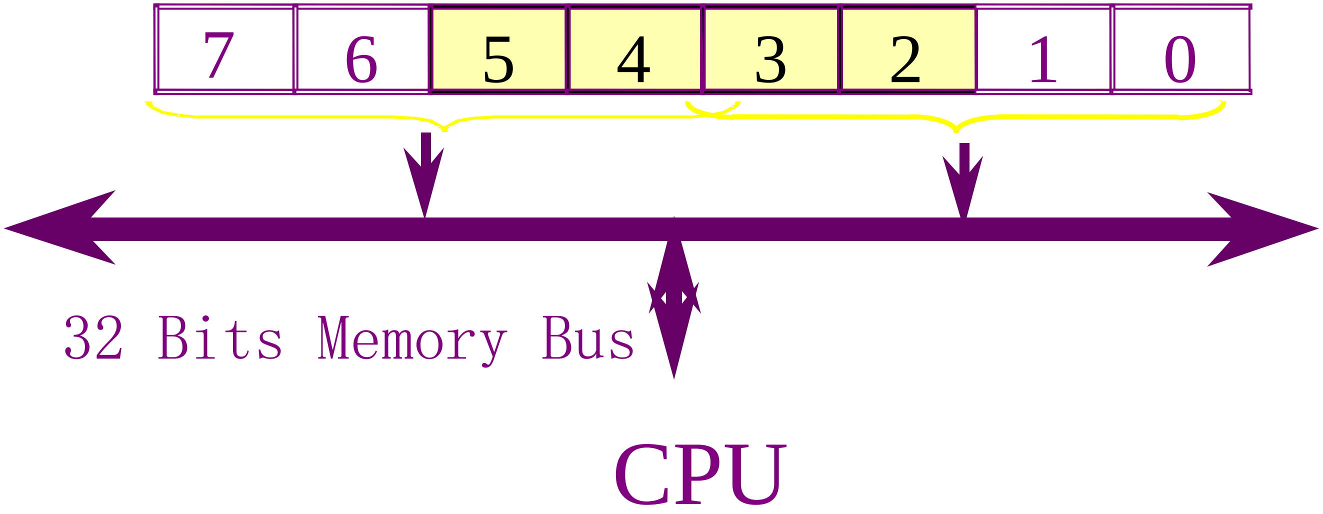
byte	XXXXXXXXXXXX
half-word	XXXXXXXXXXX 0
word	XXXXXXXXXX 0 0
double-word	XXXXXXXXX 0 0 0





Misaligned memory access

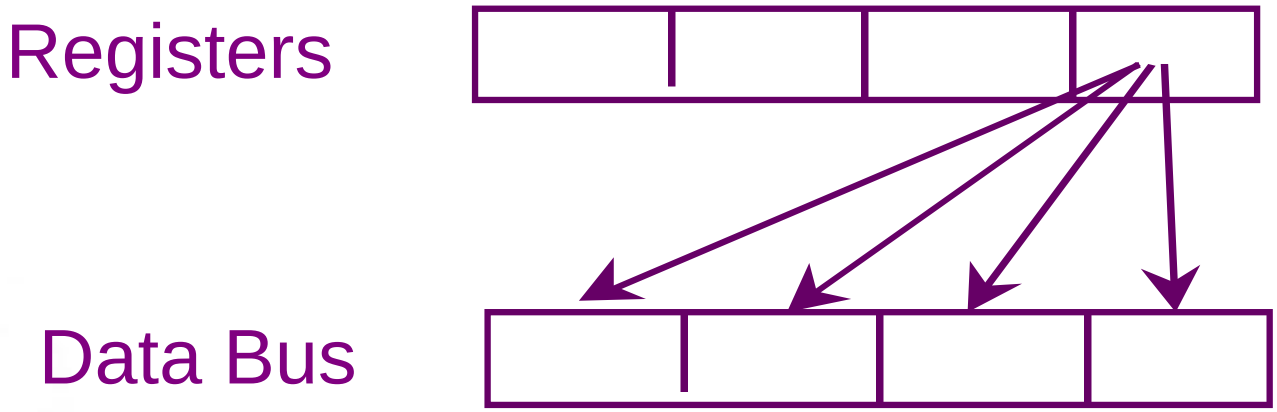
- A misligned memory access may take multiple aligned memory references



- Even in computers that allow misaligned access, programs with aligned accesses run faster.

Hardware Alignment

- Alignment network
  - External alignment： Data exchange between CPU and external storages
  - Internal alignment： Data exchange between CPU's internal data bus and registers



Addressing Modes (Fig B.6)

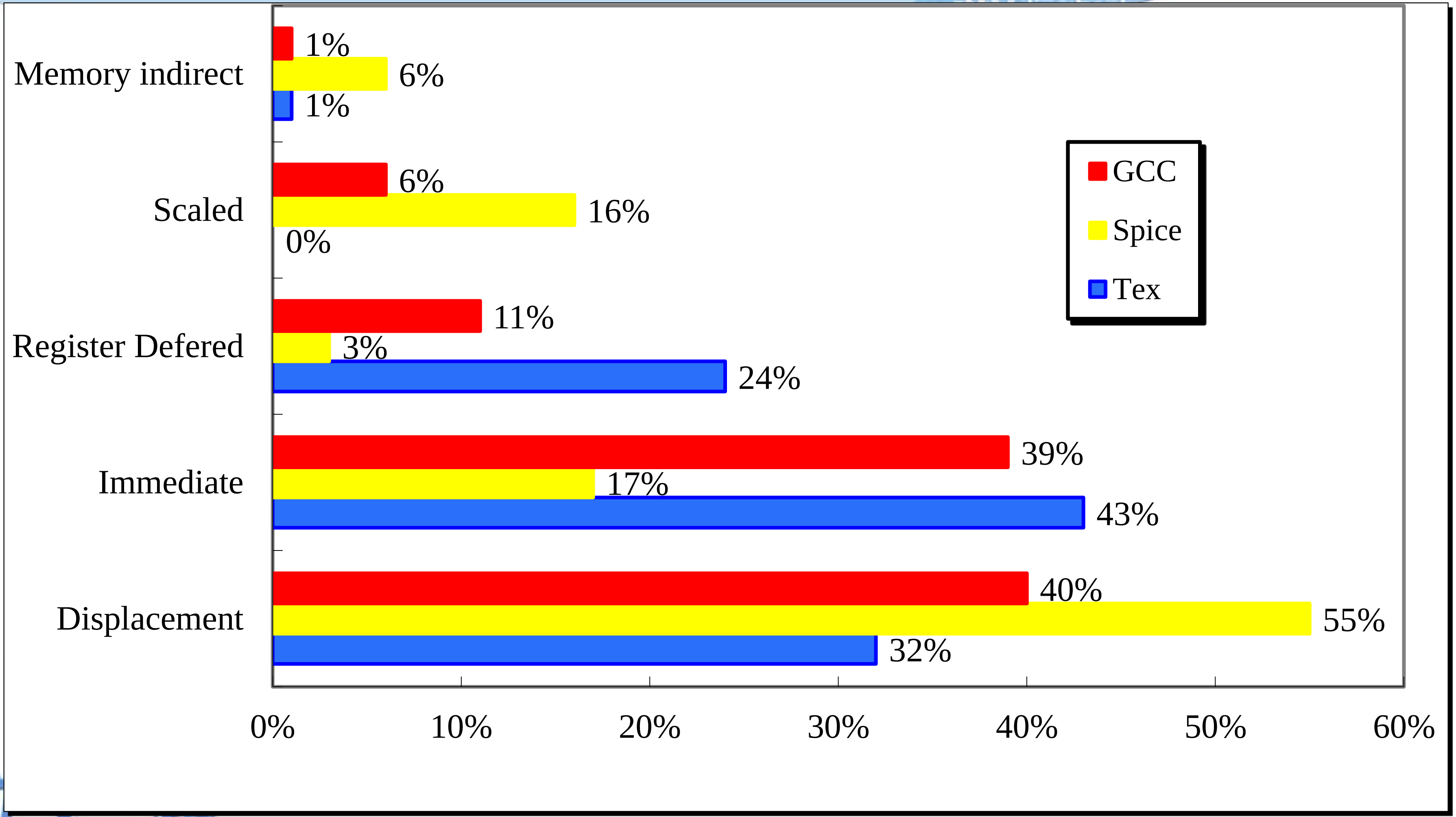
- Addressing Modes

Register	Add R4, R3
Immediate	Add R4, #3
Displacement	Add R4, 100(R1)
Register indirect	Add R4, (R1)
Indexed	Add R3, (R1+R2)
Direct or absolute	Add R1, (1000)
Memory indirect	Add R1, @(R3)
Autoincrement	Add R1, (R2)+
Autodecrement	Add R1, -(R2)
Scaled	Add R1,100(R2)[R3]

Measuring addressing mode

- Addressing modes influence Instruction Set Architecture
  - Significantly reduce instruction counts
  - Add to the complexity of building a computer
  - May increase the average CPI
- So measuring various addressing modes is quite important in helping the architect choose what to include.

Summary: use of memory addressing mode



Summary: use of memory addressing modes

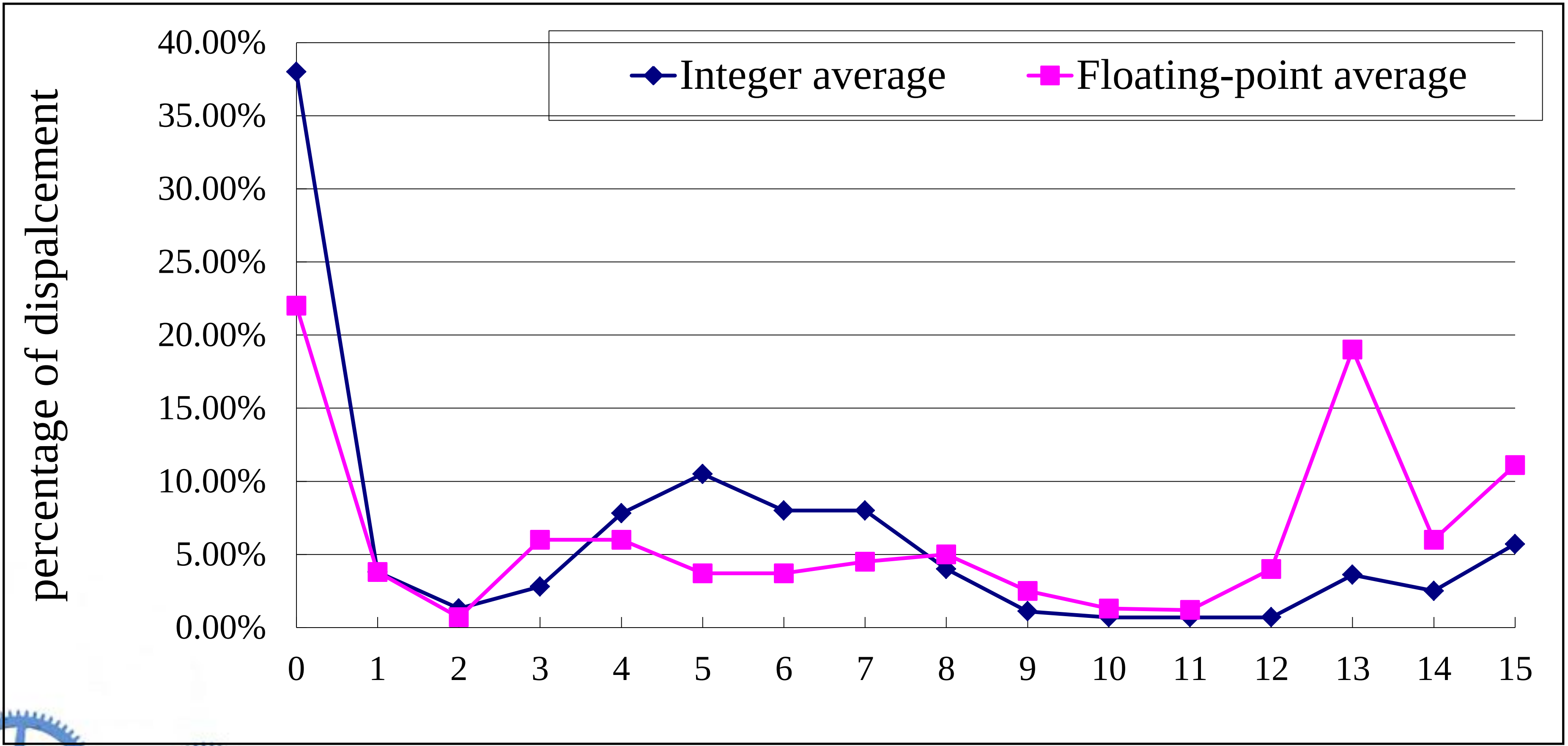
- Register modes, which are not counted, account for one-half of the operand references
- The PC-relative addressing modes, used almost exclusively for branches, are not included;
- Displacement mode includes all displacement lengths (8, 16, and 32 bits)
- Most popular memory addressing modes are:

Displacement	42%
Immediate	33%
Register indirect	13%

Displacement Addressing Mode

- Provides the means of implementing pointers
- Issue: What is the appropriate displacement field size ?
  - Important because it affects instruction length.

Summary of the range of displacement values



Summary of the range of displacement values(2)

- Data were collected on a computer with 16-bit displacements, so can't tell us about longer displacements.
- Data are relative to the policies of compiler optimization.
- The graph does not include the sign bit. Most displacements are positive, but a majority of the largest displacements(14+ bits) are negative.
- Number of bits needed for displacement values:

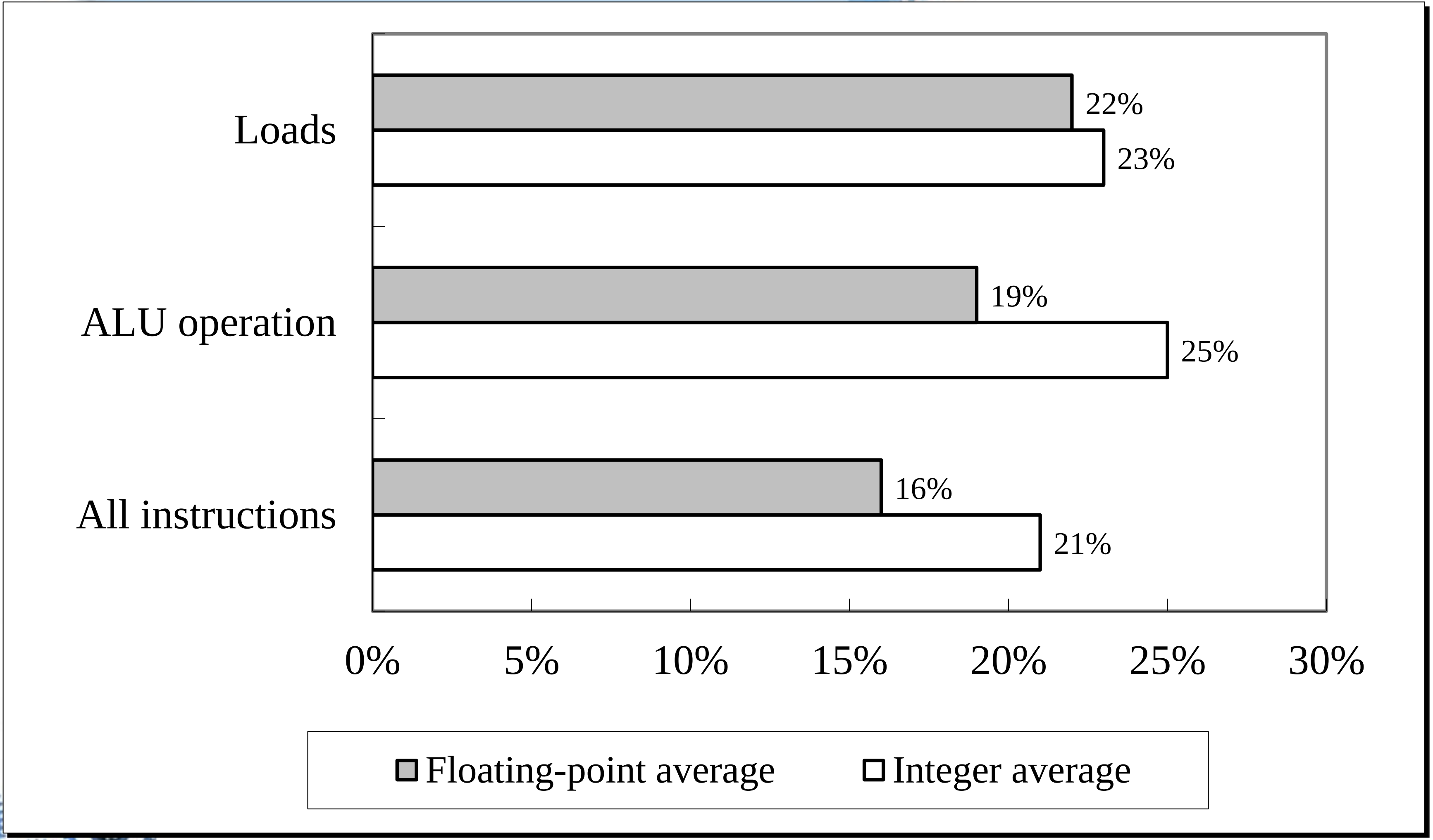
≤12 bits	75%
≤16 bits	99%
16～31 bits	1%

- Therefore, 12-16 bits is probably sufficient.

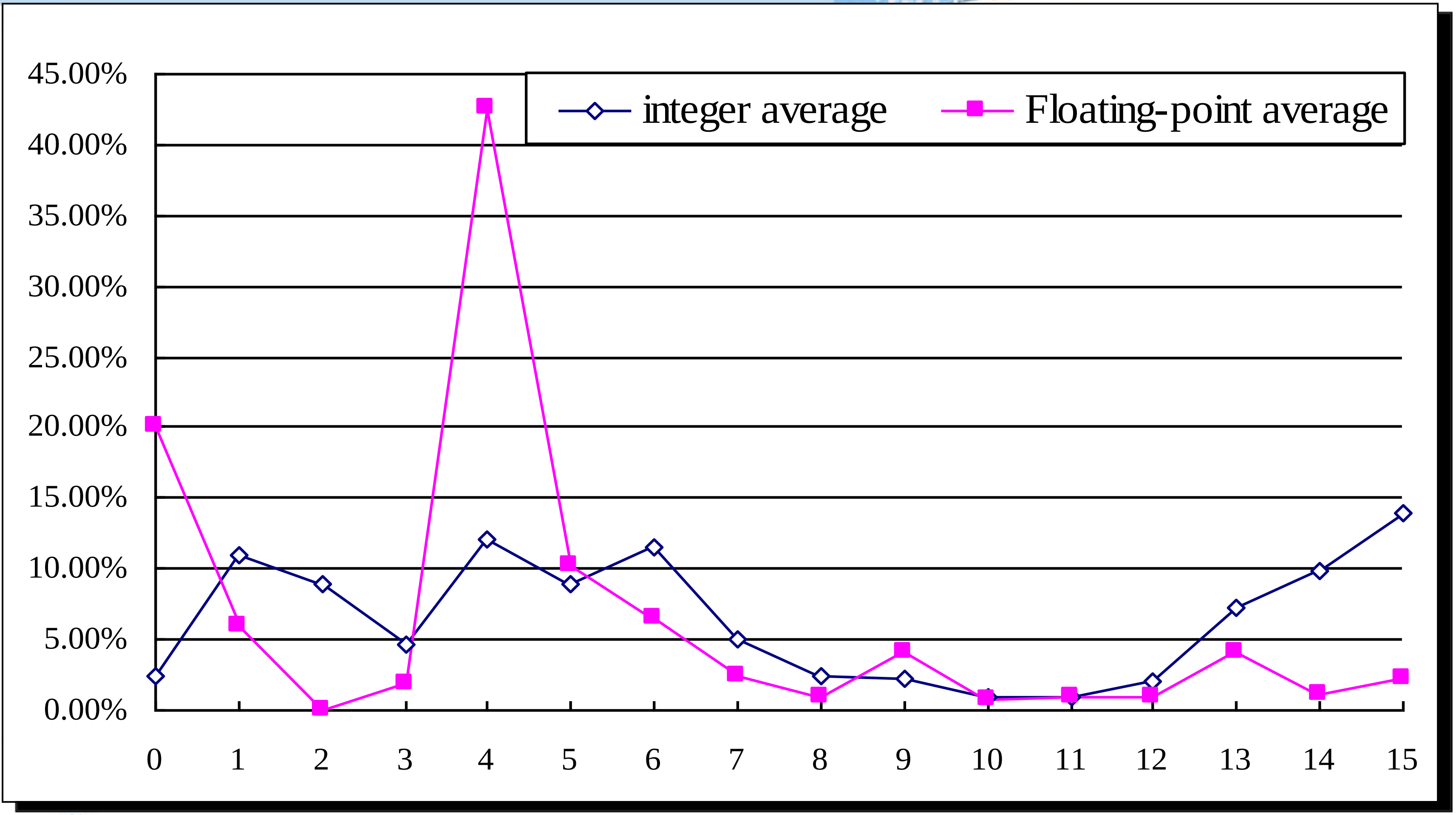
Immediate Addressing Mode

- Immediate are mostly used in: arithmetic operations, comparisons, and data moves;
- The last case occurs for constants written in the code-which tend to be small, and for address constants, which tend to be large;
- Issue:
  - What is the appropriate immediate field size ?
    - Important because it affects instruction length.
  - Support all operations or only a subset?

Percent of instr. which provide immediates



The distribution of immediate values



Summary of Immediate mode

- percent of instructions which provide immediate addressing mode

Integer ALU	21%	1/5
Floating-Point	16%	1/6

- range of values for immediates

<8 Bits	65%~ 90%
<16 Bits	82%~ 99%

- Therefore, 8-16 bits is probably sufficient.
- Other addressing modes are certainly useful, but are they worth the chip space and design complexity ？

2.4 Addressing Modes for Signal Processing

- **Several novel addressing modes for DSPs**:
  - Modulo or circular addressing mode
  - Bit reverse addressing

    $X_1X_2X_3....X_n$ ----> $X_nX_2X_3....X_1$

- There is often a mismatch between what programmers and compilers actually use versus what architects expect.

Summary: Memory Addressing

- Support at least 3 addressing mode
  - Register indirect, displacement, immediate
  - Fig B.7, 75%~99%
- The size of the address for displacement mode to be at least 12-16 bits
  - FigB.8, 75%~99%
- The size of the immediate field to be at least 8-16 bits
  - FigB.10, 50%~80%

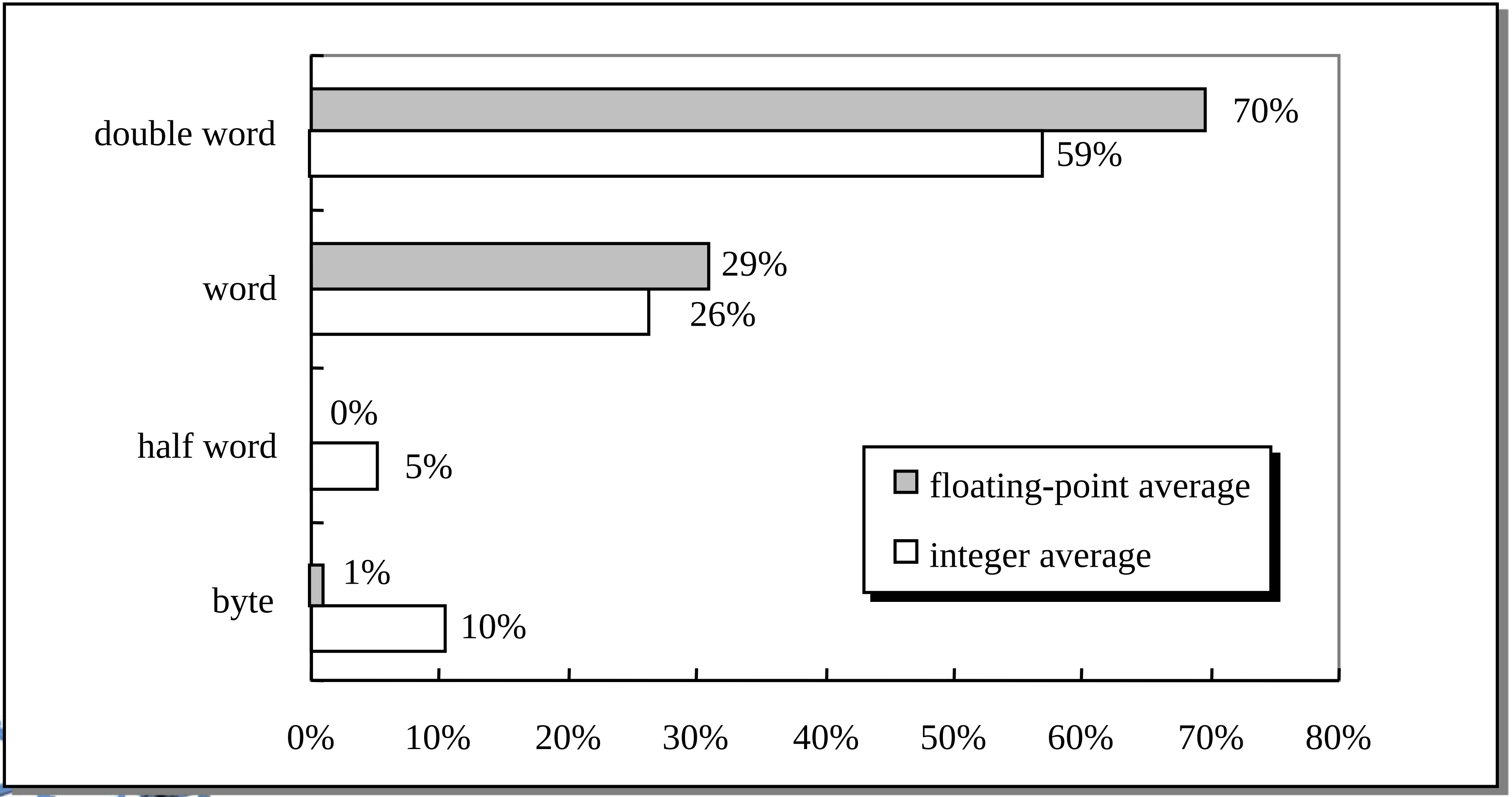
2.5 Type and Size of Operands

- How is the type of an operand designated?
  - Encode in the opcode
  - Annotated with tags (interpreted by the hardware)
- Common used operand types include:
  - byte(1B)、half word(2B)、word(4B)、single-precision floating point(4B)、double-precision floating point(8B)
  - packed decimal, character strings

Frequency of access to different data types

- What types are most popular used which need to be supported by hardware?
- Should the computer have a 64-bit access path, or would taking two cycles to access a double word be satisfactory?
- How important is it to support bytes as primitives?

Most Common Used Data Types Statistics



Summary of the usage of integer Data types

- Bytes or half words access accounts for no more than 12% of register references, or roughly 6% of all operand accesses (VAX)
- Use more than one instructions to implement access of bytes and half words (Alpha)
- Double words access frequency will be increased with the development of 64 bits computers

2.6 Operands for Media and Signal Processing

- Data types used in 2D & 3D images:
  - Vertex (32-bit floating-point values)
    - **x-coordinate, y-coordinate, z-coordinate, w (help with color or hidden surfaces)**
  - Triangle (3 vertices)
  - Pixel (32bits)
    - **R, G, B, A**
- Fixed point(special data type used in DSP, low-cost floating point)
  - Has a binary point just to the right of the sign bit
  - Fixed-point data are fractions between –1 and +1

Summary of Type and Size of Operands

- A new 32-bit architecture to support 8-,16-,32-bit integer and 32- and 64-bit IEEE floating-point data.
- A new 64-bit address architecture need to support 64-bit integer.
- Support for decimal data is less clear.
- DSPs need wider accumulating registers than the size in memory to aid accuracy in Fixed-point arithmetic.

2.7 Operations in the Instruction Set

- **Categories of instruction operators:**
  - **Basic instruction operators**
    - Arithmetic and logical
    - Data transfer
    - Control
  - **Special instruction operators**
    - Floating point(scientific calculation)
    - Decimal(commercial)
    - String
    - Graphics
  - **Privileged instruction operators**
    - Virtual memory management instructions
    - Operating system call

Categories of instruction operators and examples of each

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiple, divide
Data transfer	Loads-stores(move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiple, divide, compare
Decimal	Decimal add, decimal multiple, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

Instruction Operations

- All machines generally provide a full set of operations for the first three categories.
- All machines MUST provide instruction support for basic system functions.
- Floating point instructions are optional but are commonly provided.
- Decimal and string instructions are optional, because they can be easily emulated by sequences of simpler instructions.
- Graphic instructions are optional.

Decide which operations to support

- **Rule of thumb**

  most widely executed instructions are the simple operations of an instruction set. Hence, the implementor of these instructions should be sure to make these fast.

- Remember MAKE THE COMMON CASE FAST ?
- **How to get the statistic data ?**

  usually use benchmarks

The top 10 instructions for the 80x86

Rank	80x86 instruction	Integer average
1	Load	22%
2	Conditional branch	20%
3	compare	16%
4	Store	12%
5	Add	8%
6	And	6%
7	Sub	5%
8	Move register-register	4%
9	Call	1%
10	Return	1%
Total		96%

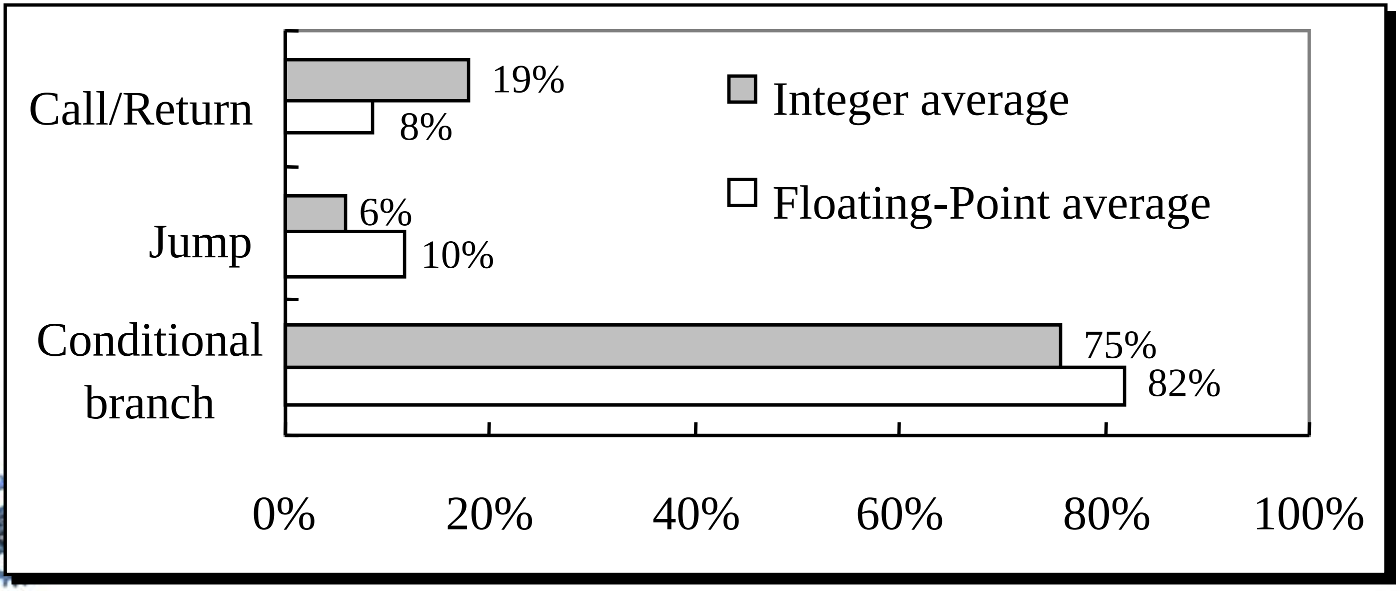
2.8 Operations for Media and Signal Processing

- **Some special operations to Support media and signal processing:**
  - Single-instruction multiple-data (SIMD), vector(Fig2.17)
    - fixed-width operations, performing multiple narrow operations on either a 64-bit or 128-bit ALU
    - partitioned add
    - paired single operations
  - Saturating arithmetic (for DSP)
  - Multiply-accumulate(MAC) instruction (for DSP)

2.9 Instructions for Control Flow

- Conditional branches branch
- Unconditional jumps
- Procedure calls call
- Procedure returns return

Frequencies of these control flow instructions



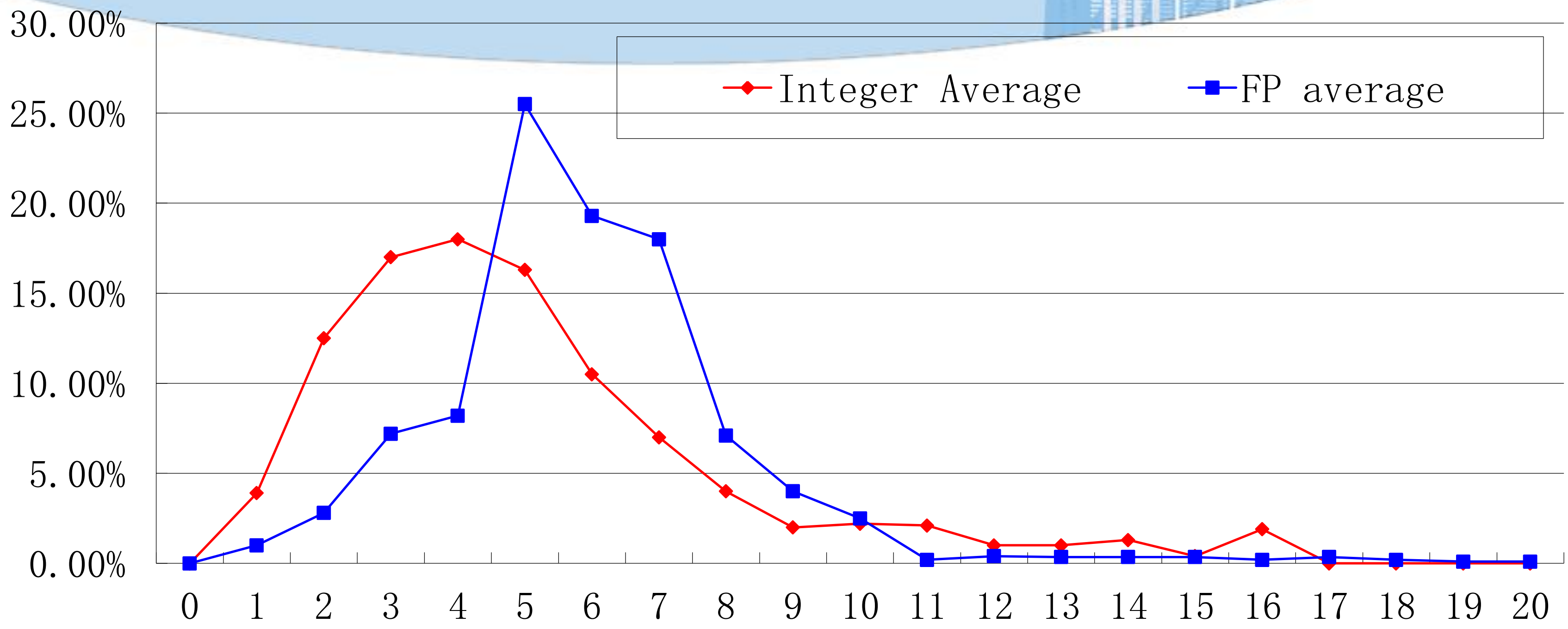
Specification of destination address

- The destination is specified explicitly in the instruction in the vast majority of cases
  - Procedure return is the major exception, since for return the target is not known at compiler time,

How to specify the destination?

- **PC-relative**
  - The target is often near the current instruction, so it requires fewer bits
  - Position independence(permit the code to run independently of where it is loaded)
  - How to do with procedure return or indirect jump?
- **Register indirect**
  - case or switch
  - virtual functions or methods
  - high-order functions or function pointers
  - dynamically shared libraries

Bits of branch displacement



- Most displacement can be encoded in 2~7 bits

  ≤ 7 bits：93%

- About 75% of the branches are in the forward direction

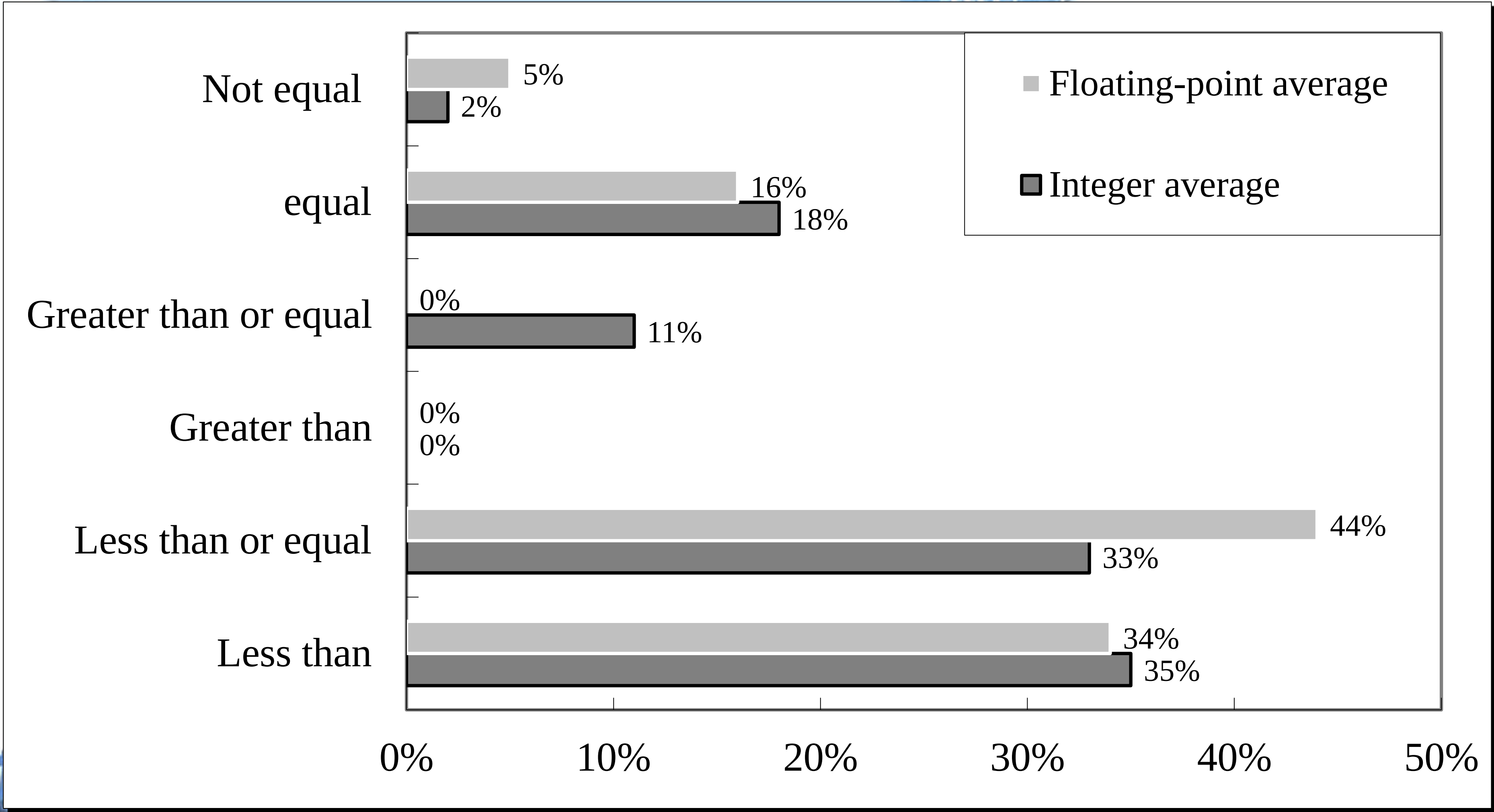
Conditional Branch Options

- Since most changes in control flow are braches, deciding how to specify the branch conditions is important.
- Three techniques to specify the branch conditions
  - condition code tests special bits set by ALU operations, possibly under program control.
  - condition register tests arbitrary register with the result of a comparison.
  - compare and branch compare is part of the branch. Often compare is limited to subset.

Pros and cons of three methods

Name	Examples	Advantages	Disadvantages
Condition Code(CC)	80x86,ARM, PowerPC, SPARC, SuperH	Sometimes condition is set for free	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch
Condition register	Alpha, MIPS	Simple	Uses up a register
Compare and branch	PA-RISC, VAX	One instruction rather than two for a branch	May be too much work per instruction for pipelined execution

Frequency of different types of compares in conditional branches



Analysis of different types of compares in conditional branches

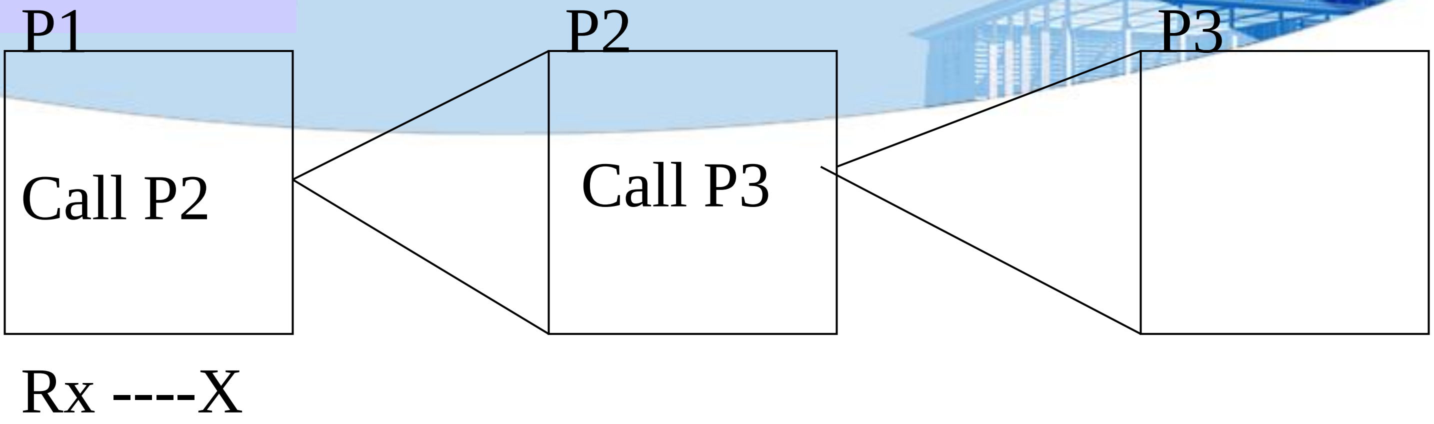
- Less than (or Equal) branches dominate this combination of compiler and architecture
- Comparisons with 0: ≥50% is " =0 "

  (this leads to third method to specify the branch condition, "compare and branch")
- A special branch instruction: not only makes comparisons, but also branches
- DSP add repeat instruction to avoid loop overhead.

Procedure invocation options (call & return)

- **State saving** (at a minimum the return address must be saved somewhere)
- **Save registers**
  - Provide a mechanism to save many registers
  - Require the complier to generate stores and loads for each register saved and restored
- caller-saving: Caller saves any registers that it wants to use after the call, then invoke.
- callee-saving: first invoke, then callee saves the registers.

Sometimes, caller save must be used



- Caller save: store x to a location,which is known by P2.
- Compiler should discover a called procedure may access register-allocated quantities. ------complicated by separate compilation.
- Many compilers conservatively caller save any variable that may be accessed during a call.
- Most real systems today use a combination of the two conventions.

Summary: Instructions for Control flow

- Common used instructions shall be considered firstly： Load, store, add, sub, move R-R, and, shift, ＝, ≠, branch and etc.
- Conditional branch: displacement 100 <=2^7 PC-relative branch: displacement > 8 bits
- PC-relative conditional branches dominate the control instructions.
- jump and link instruction for procedure call
- register indirect jump for procedure return;

2.10 Encoding an Instruction Set

- Encoding affect:
  - Size of compiled program
  - the implementation of the CPU
- Balancing forces:
  - From the compiler viewpoint: to have as many registers and addressing modes as possible
  - Impact of register size and addressing mode fields on Average instruction size、average program size
  - Easy to implement

Key factors for Encoding

- Key Factors
  - The range of addressing modes
  - The degree of independence between opcodes and addressing modes

Three popular choices for instruction encoding

Operation & no. of operands	Address specifier1	Address field 1	…	Address specifier n	Address field n

(a) Variable(e. g. , VAX, Intel 80x86)

Operation & no. of operands	Address field 1	Address field 2	Address field 3

(b) Fixed(e. g. , Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

Operation	Address specifier	Address field

Operation	Address specifier1	Address specifier2	Address field

Operation	Address specifier	Address field 1	Address field 2

(c) Hybrid(e. g. , IBM360/70, MIPS16, Thumb, TI TMS320C54x)

Comparison of three instruction encoding formats

- **Variable length:**
  - Number of operations and addressing modes is big
  - Small program size, high code density, a variety of formats for one instruction
- **Fixed length:**
  - Number of operations and addressing modes is small
  - Large program size, low code density, fixed format, easy to implement
- **Hybrid**
  - Has multiple formats specified by the opcode, adding or or two fields to specify the addressing mode and one or two fields to specify the operand address

Reduced Code Size in RISCs

- New hybrid version of RISC instructions, with both 16-bit and 32-bit instructions.
  - ARM Thumb , MIPS16
- IBM CodePack : compress standard instruction set
  - full 32-bit instruction in instruction cache
  - compressed code kept in main memory, ROM, disk.
  - Hash table (TLB)
- Hitachi: special RISC instruction set for embedded applications.
  - SuperH

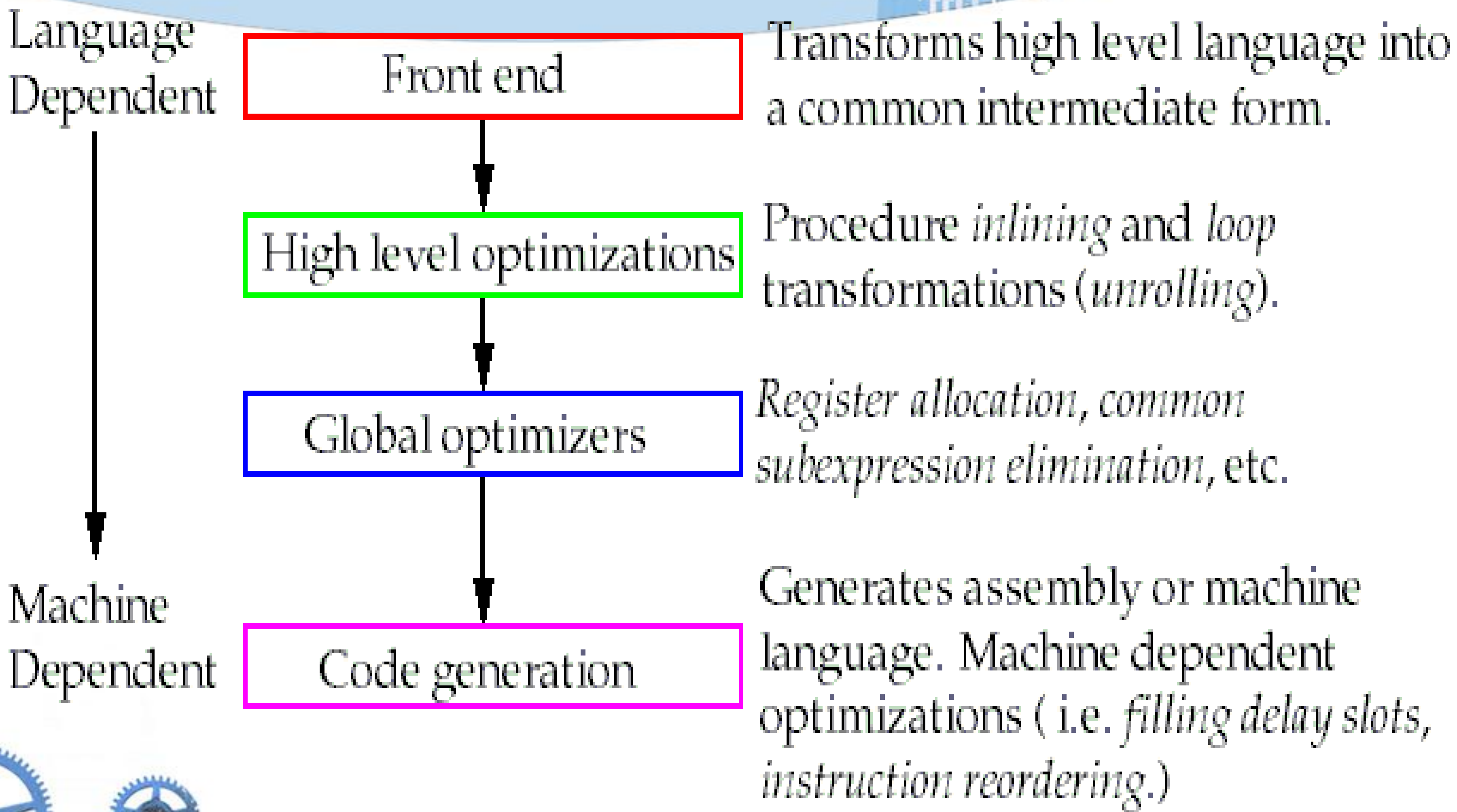
2.11 The Role of Compilers

- Understanding compiler technology is critical to designing an effective instruction set.
- Assembly language programming has been largely replaced by compilers which work together with the hardware to optimize performance.
- Therefore, design architectures to be compiler targets .

Compilers and Architecture

- What features of an architecture lead to high quality code ?
- What "makes it easy" to write efficient compilers for an architecture ?

The Structure of Recent Compilers



Language Dependent

Machine Dependent

Front end

Transforms high level language into a common intermediate form.

High level optimizations

Procedure *inlining* and *loop* transformations (*unrolling*).

Global optimizers

Register allocation, *common subexpression elimination*, etc.

Code generation

Generates assembly or machine language. Machine dependent optimizations (i.e. *filling delay slots*, *instruction reordering*.)

About compiler

- **The goals of compiler**
  - All valid programs must be compiled correctly
  - Fast speed of the compiled code
  - fast compilation, debugging support, interoperability among languages
- **Multiple-pass structure's advantage:**
  - Reduce compiler complexity
  - Easy writing a bug-free compiler
- **Disadvantages:**
  - Phase-ordering problem

    e.g. global common subexpression elimination

Optimizations Classification

- High-level optimizations
  - Procedure inlining
- Local optimizations within a straight-line code fragment
  - Common subexpression elimination、constant propagation、
- Global optimizations extend the local optimizations across branches and introduce a set of transformations aimed at optimizing loops
- Register allocation associates registers with operands
  - Calculate expressions、transfer parameters、store variables
- Processor-dependent optimizations attempt to take advantage of specific architectural knowledge

The Impact of Compiler Technology on the Architect's Decisions

- **Two important questions:**
  - **How are variables allocated and addressed?**
  - **How many registers are needed to allocate variables appropriately?**
- **Three areas in which current high-level languages allocate the data:**
  - Stack: local variables; scalars(single variables)
  - Global data area: global variables, constants; arrays
  - Heap: dynamic objects; accessed with pointers
- At least 16 GPRs + separate floating-point registers

How the Architect Can Help the Compiler Writer ?

- **The difficulties of compiler**
  - Big program size
  - Interactive
  - complexity of compiler's structure
- **Basic principle of the compiler**
  - Make the frequent case fast and the rare case correct

Architect’s Guidelines

- **Provide regularity**
- **Provide primitives**, not solutions
  - Providing special features that "match" language constructs is NOT a good idea.
  - These features may be good only for a certain language.
  - And, worse, they may match but do more or less than what's required.
- **Simplify trade-offs** among alternatives
  - If there are 20 ways to implement an instruction sequence, it makes it difficult for the compiler writer to choose which is the most efficient.
- provide instructions that bind quantities known at compile time as constants.

Summary for compiler's role

- At least 16 gerneral-purpose registers
- all supported addressing modes apply to all instructions that transfer data
- provide primitives instead of solutions
- simplify trade-offs between alternatives
- KEEP IT SIMPLE , Less is more
- SIMD extensions are examples of good marketing than that of hardware-software codesign

2.12 The MIPS Architecture

- **MIPS emphasizes:**
  - A simple load-store instruction set
  - Design for pipelining efficiency, including a fixed instruction set encoding
  - Efficiency as a complier target
- MIPS provides a good architectural model for study, because of:
  - Popularity of this type of processor
  - An easy architecture to understand

Summary of the statistic data from above sections

- B.2 Use GPRs with a load-store architecture
- B.3 Addressing modes:

  displacement(12-16), immediate(8-16), register indirect
- B.4 Support the data size and types:

  8-, 16-, 32-, and 64-bit integers and 64-bit IEEE 754 floating-point numbers
- B.5 Support the simple instructions:

  load, store, add, subtract, move register-register, and shift
- B.6 compare equal, compare not equal, compare less, branch, jump, call, and return
- B.7 Use fixed instruction encoding
- B.8 Provide at least 16 GPRs, and all addressing modes apply to all data transfer instructions

MIPS emphasizes

- A simple load-store instruction set
- Design for pipelining efficiency, fixed instruction set encoding
- Efficiency as a compiler target

MIPS Architecture

- **Registers for MIPS**
  - R0~R31, F0~F31, a few special registers
- **Data Types for MIPS**
  - 8-bit bytes, 16-bit half words, 32-bit words, and 64-bit double words for integer data
  - 32-bit single precision and 64-bit double precision for floating point
- **Addressing Modes for MIPS Data Transfers**
  - Immediate, displacement
  - (register indirect ~ D=0

    absolute addressing ~ base register=R0)
  - PC-relative addressing

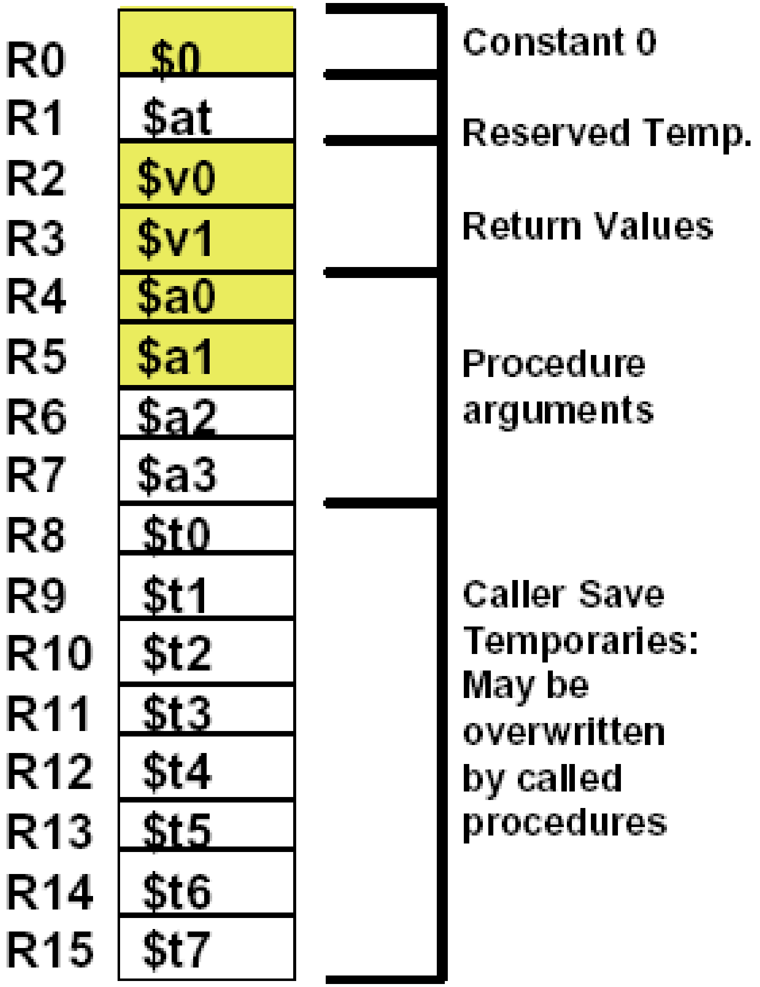
MIPS Register Conventions

Conventions

- This is an agreed upon "**contract**" or
- "**protocol**" that everybody follows

Specifies correct (and expected) **usage**, and some **naming** conventions

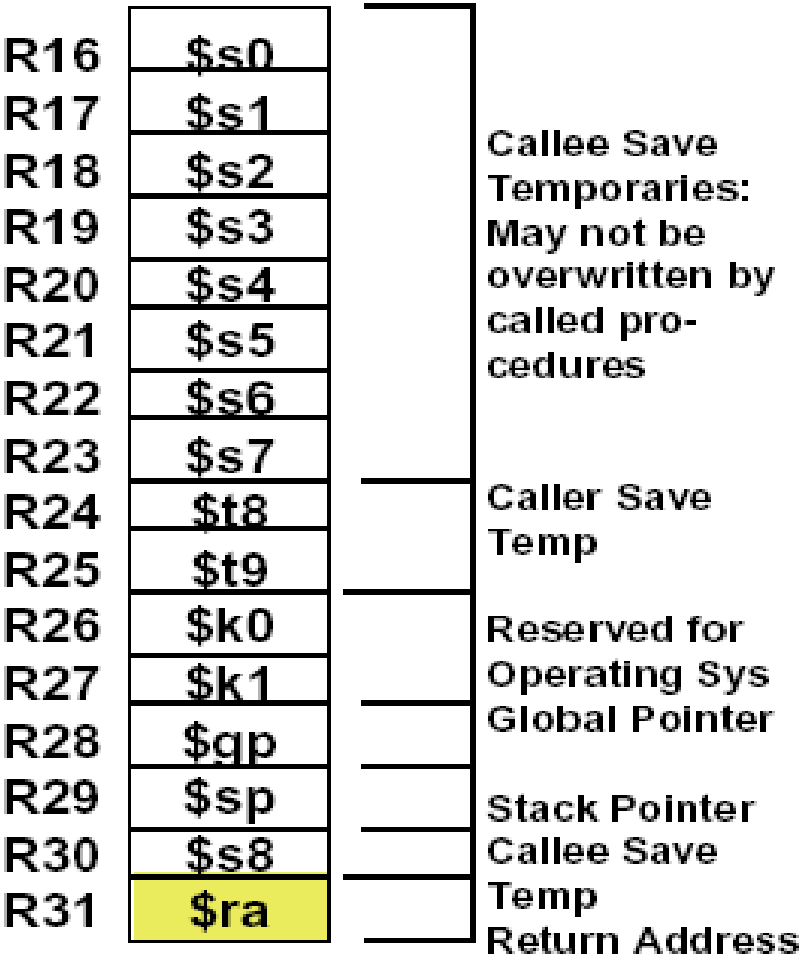
- **Established part of architecture**
- Used by all compilers, programs, and libraries
- Assures **compatibility**



R0	$0	Constant 0
R1	$at	Reserved Temp.
R2	$v0	Return Values
R3	$v1	
R4	$a0	Procedure arguments
R5	$a1	
R6	$a2	
R7	$a3	
R8	$t0	Caller Save Temporaries: May be overwritten by called procedures
R9	$t1	
R10	$t2	
R11	$t3	
R12	$t4	
R13	$t5	
R14	$t6	
R15	$t7	

MIPS Register Convention (cont.)

- Important Ones for Now (shaded)
- **R0** Constant 0
- **R2** Return Value
- **R3** Can use as temporary
- **R4** First argument
- **R5** Second argument
- **R31** Return address



R16	$s0	Callee Save Temporaries: May not be overwritten by called procedures
R17	$s1	
R18	$s2	
R19	$s3	
R20	$s4	
R21	$s5	
R22	$s6	
R23	$s7	
R24	$t8	Caller Save Temp
R25	$t9	
R26	$k0	Reserved for Operating Sys
R27	$k1	
R28	$gp	Global Pointer
R29	$sp	Stack Pointer
R30	$s8	Callee Save Temp
R31	$ra	Return Address

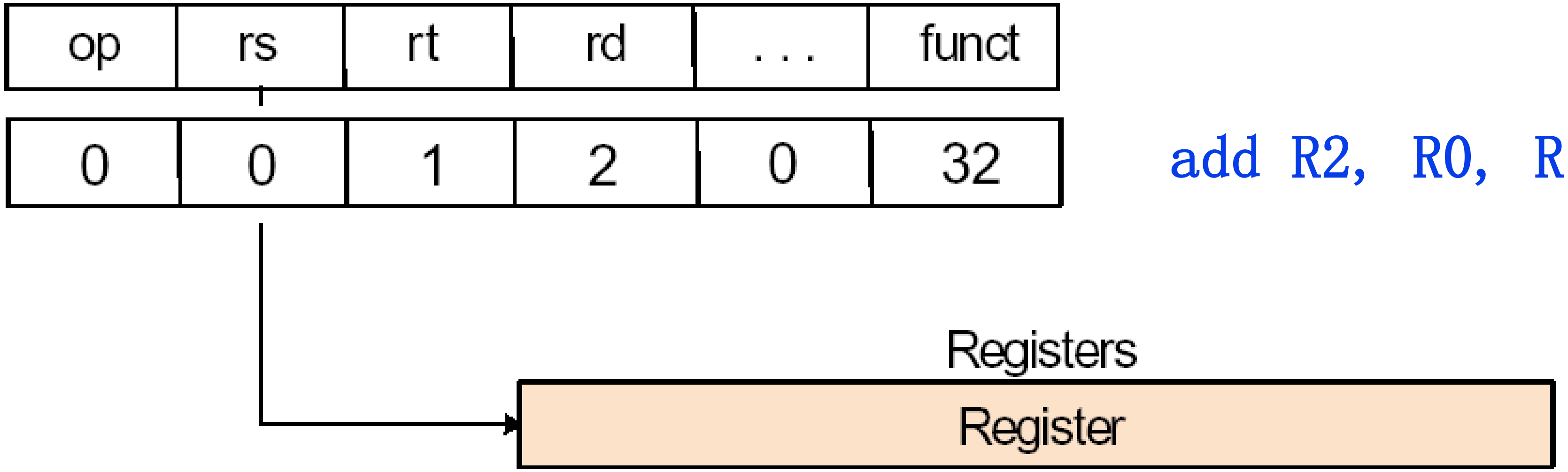
MIPS Addressing Modes

1. Immediate addressing (I-Format)

op	rs	rt	Immediate
8	0	1	10

addi R1, R0, 10

2. Register addressing (R-Format)

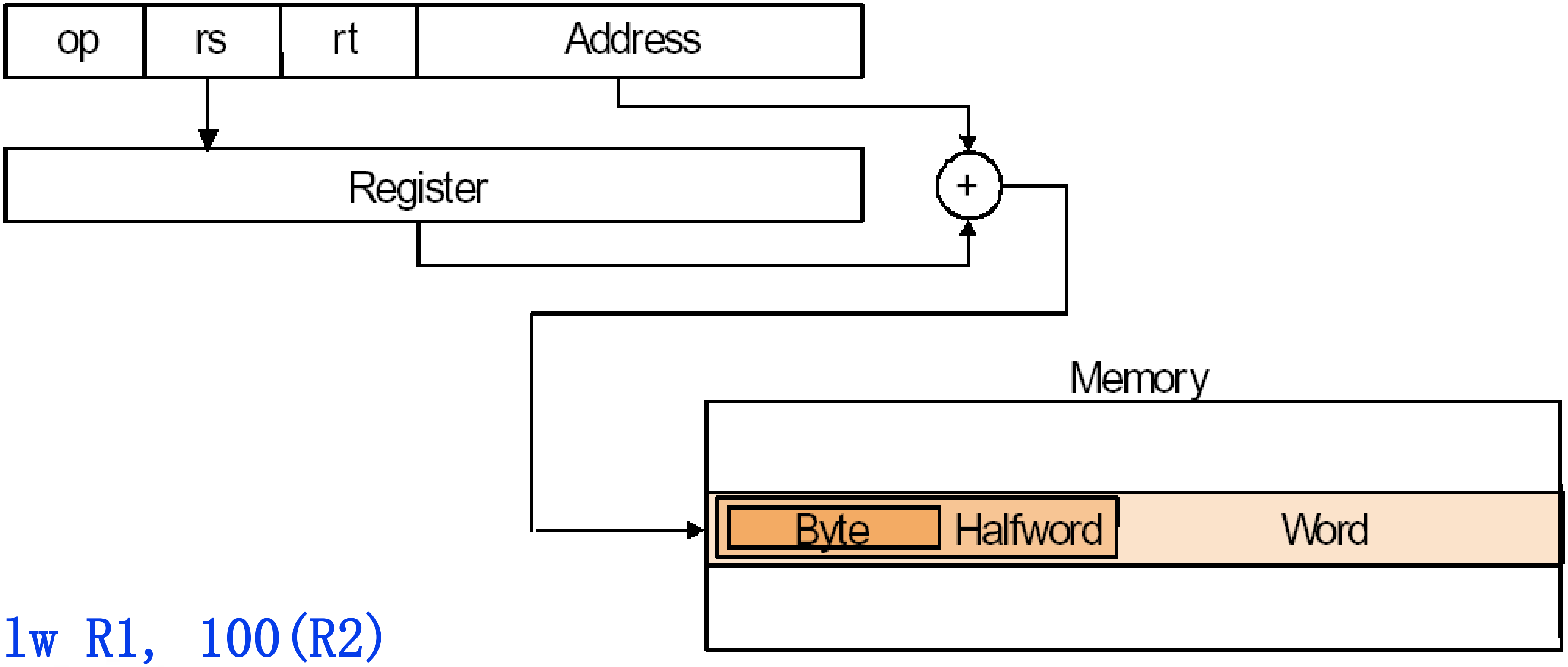


op	rs	rt	rd	. . .	funct
0	0	1	2	0	32

add R2, R0, R1

MIPS Addressing Modes

3. Base addressing (I-Format)

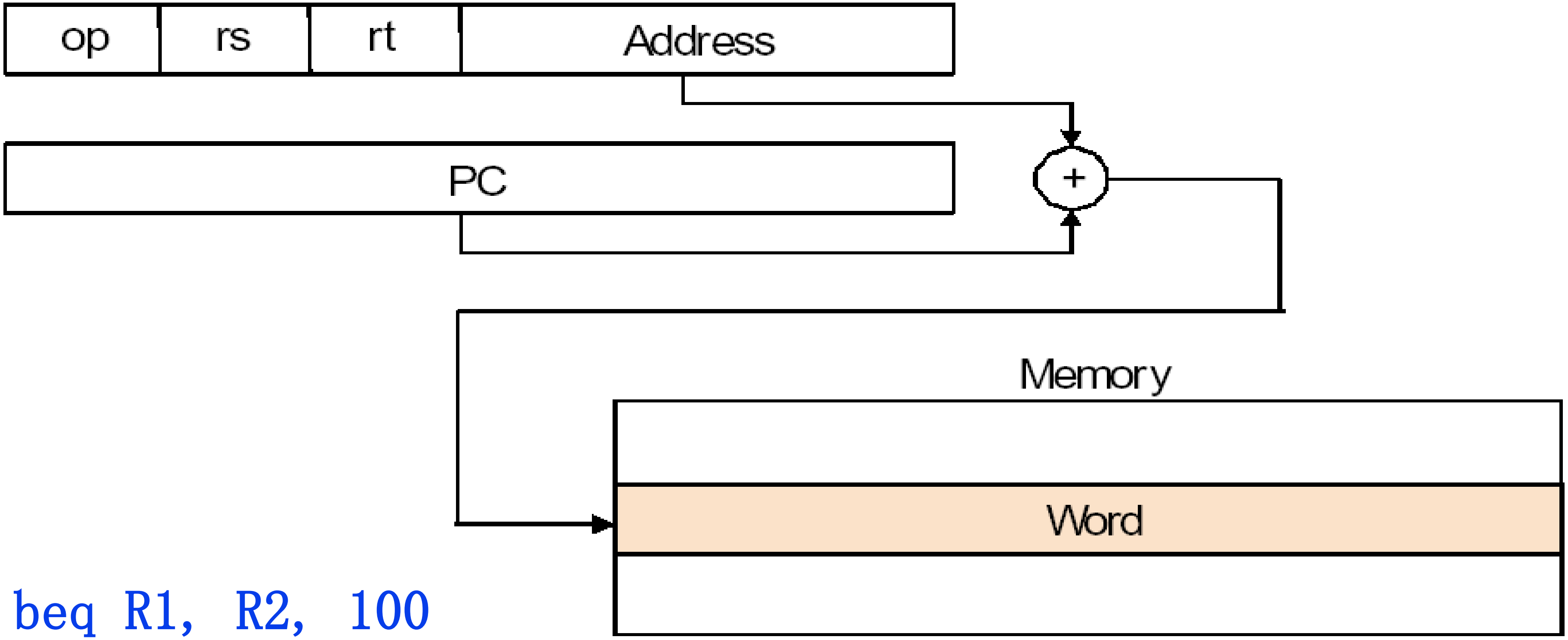


lw R1, 100(R2)

35	2	1	100

MIPS Addressing Modes

4. PC-relative addressing (I-Format)

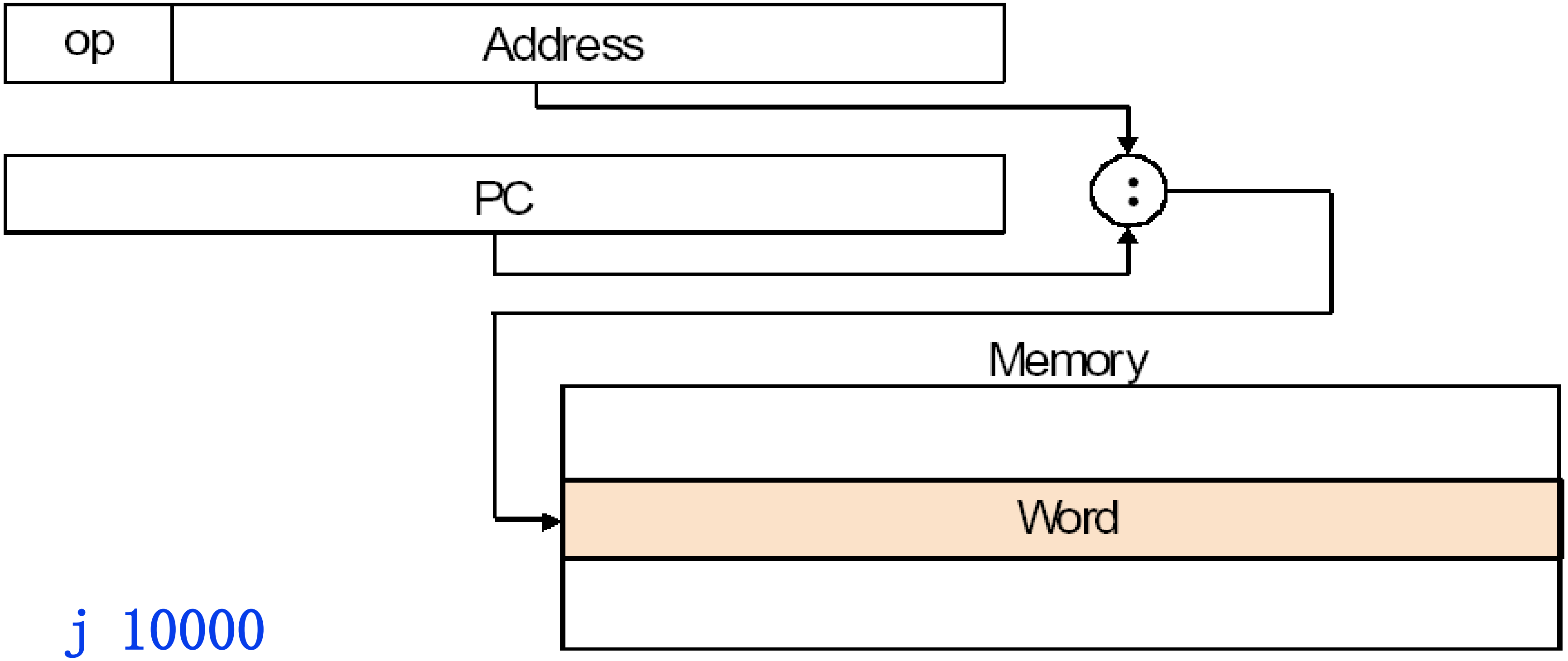


beq R1, R2, 100

4	1	2	100

MIPS Addressing Modes

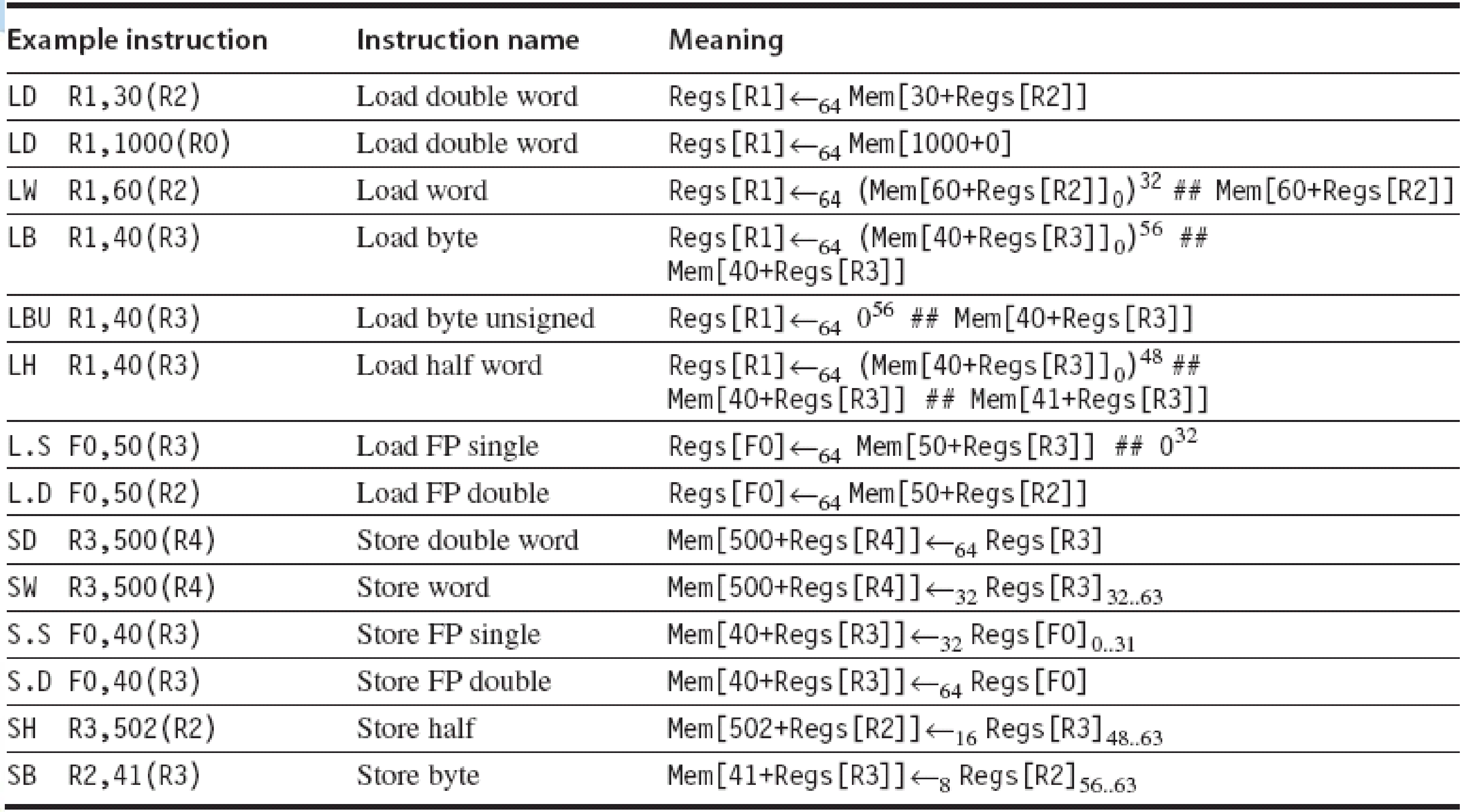
5. Pseudodirect addressing (J-Format)



j 10000

2	10000

MIPS Operations(Fig.B.23, pB-37)

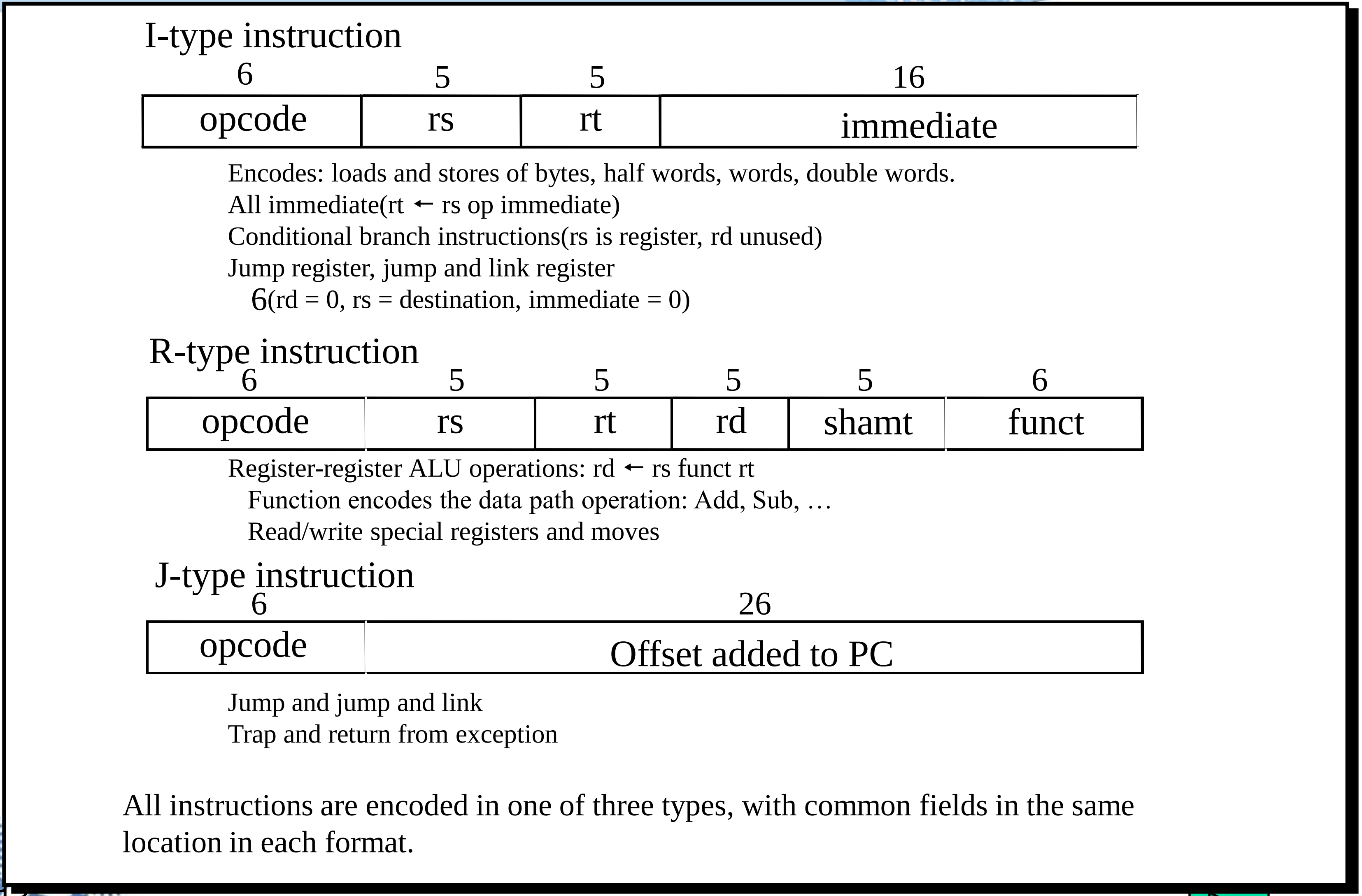


Example instruction	Instruction name	Meaning
LD R1,30(R2)	Load double word	Regs[R1]$\leftarrow_{64}$Mem[30+Regs[R2]]
LD R1,1000(R0)	Load double word	Regs[R1]$\leftarrow_{64}$Mem[1000+0]
LW R1,60(R2)	Load word	Regs[R1]$\leftarrow_{64}$ (Mem[60+Regs[R2]]$_{0}$)^{32} ## Mem[60+Regs[R2]]
LB R1,40(R3)	Load byte	Regs[R1]$\leftarrow_{64}$ (Mem[40+Regs[R3]]$_{0}$)^{56} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1]$\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1]$\leftarrow_{64}$ (Mem[40+Regs[R3]]$_{0}$)^{48} ## Mem[40+Regs[R3]] ## Mem[41+Regs[R3]]
L.S F0,50(R3)	Load FP single	Regs[F0]$\leftarrow_{64}$ Mem[50+Regs[R3]] ## 0^{32}
L.D F0,50(R2)	Load FP double	Regs[F0]$\leftarrow_{64}$Mem[50+Regs[R2]]
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]$_{32..63}$
S.S F0,40(R3)	Store FP single	Mem[40+Regs[R3]]$\leftarrow_{32}$Regs[F0]$_{0..31}$
S.D F0,40(R3)	Store FP double	Mem[40+Regs[R3]]$\leftarrow_{64}$Regs[F0]
SH R3,502(R2)	Store half	Mem[502+Regs[R2]]$\leftarrow_{16}$Regs[R3]$_{48..63}$
SB R2,41(R3)	Store byte	Mem[41+Regs[R3]]$\leftarrow_{8}$Regs[R2]$_{56..63}$

Figure B.23 The load and store instructions in MIPS. All use a single addressing mode and require that the memory value be aligned. Of course, both loads and stores are available for all the data types shown.

Instruction layout of MIPS

Fig.B.22, pB-35



I-type instruction

6	5	5	16
opcode	rs	rt	immediate

Encodes: loads and stores of bytes, half words, words, double words.
All immediate(rt ← rs op immediate)
Conditional branch instructions(rs is register, rd unused)
Jump register, jump and link register
6(rd = 0, rs = destination, immediate = 0)

R-type instruction

6	5	5	5	5	6
opcode	rs	rt	rd	shamt	funct

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

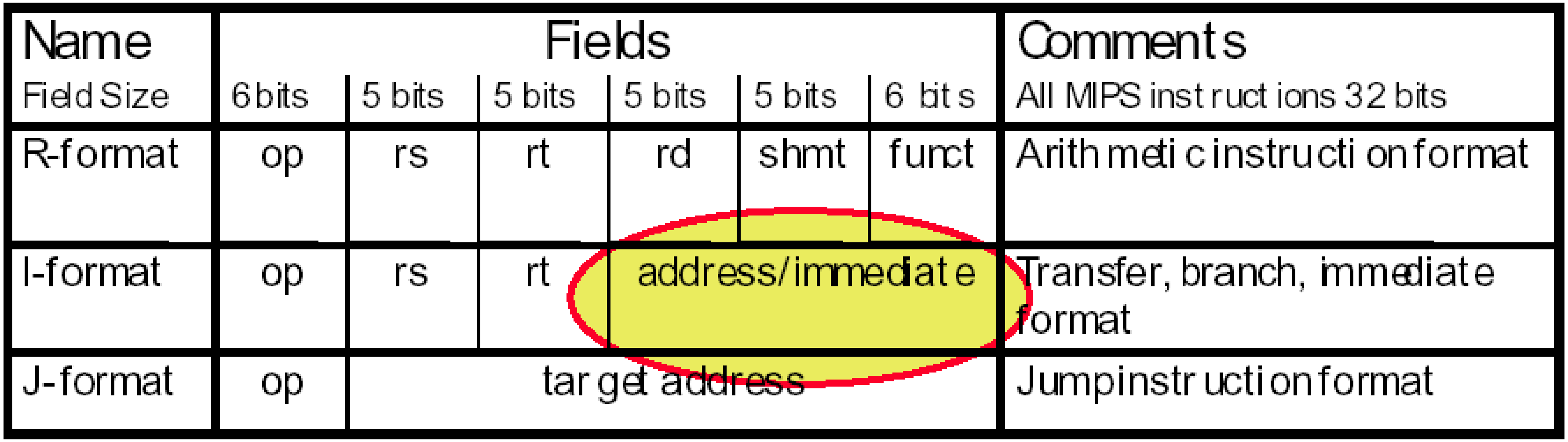
6	26
opcode	Offset added to PC

Jump and jump and link
Trap and return from exception

All instructions are encoded in one of three types, with common fields in the same location in each format.

Loading Immediate Values

- What's the largest immediate value that can be loaded into a register?

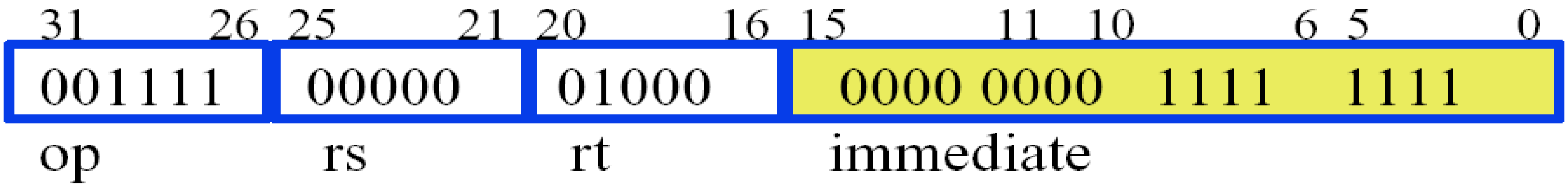


Name	Fields						Comments
Field Size	6bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shmt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, immediate format
J-format	op	target address					Jump instruction format

- But, how do we load larger numbers?

Load Upper Immediate

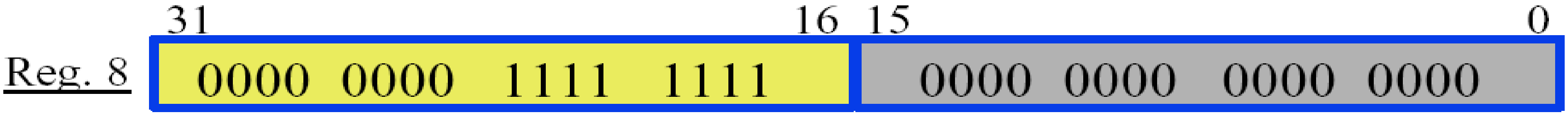
- Example: lui R8, 255



31 26	25 21	20 16	15 11 10 6 5 0
001111	00000	01000	0000 0000 1111 1111
op	rs	rt	immediate

Transfers the immediate field into the register's top 16 bits and fills the register's lower 16 bits with zeros

```
R8[31:16] <--  IR[15:0]       ; top 16 bits of R8 <-- bottom 16 bits of the IR
R8[15:0] <-- 0                ; bottom 16 bits of R8 are zeroed
```

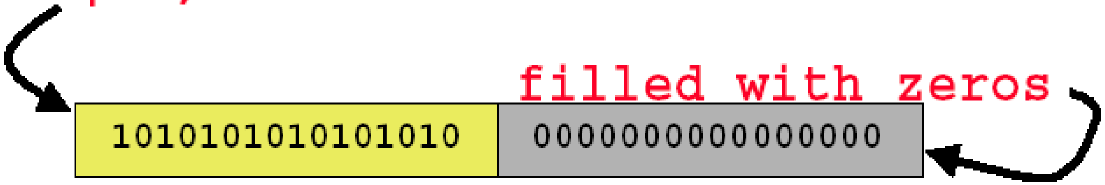


	31 16	15 0
Reg. 8	0000 0000 1111 1111	0000 0000 0000 0000

Larger Constants?

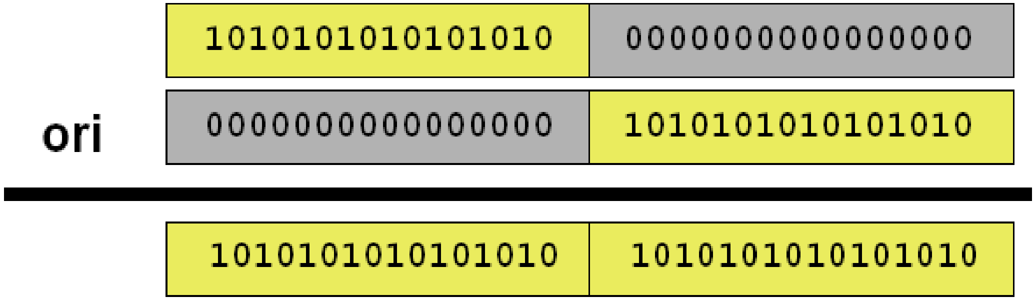
- We'd like to be able to load a 32 bit constant into a register
- Must use 2 instructions: first, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```


filled with zeros

- Second, must then get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```



Procedure calls

- Steps followed in executing a procedure call:
  - Place parameters in a place where the procedure (callee) can access them
  - Transfer control to the procedure
  - Acquire the storage resources needed for the procedure
  - Perform desired task
  - Place results in a place where the calling program (caller) can access them
  - Return control to the point of origin

Resources Involved

- Registers used for procedure calling:
  - $a0 - $a3 : four argument registers in which to pass parameters
  - $v0 - $v1 : two value registers in which to return values
  - $ra : one return address register to return to the point of origin

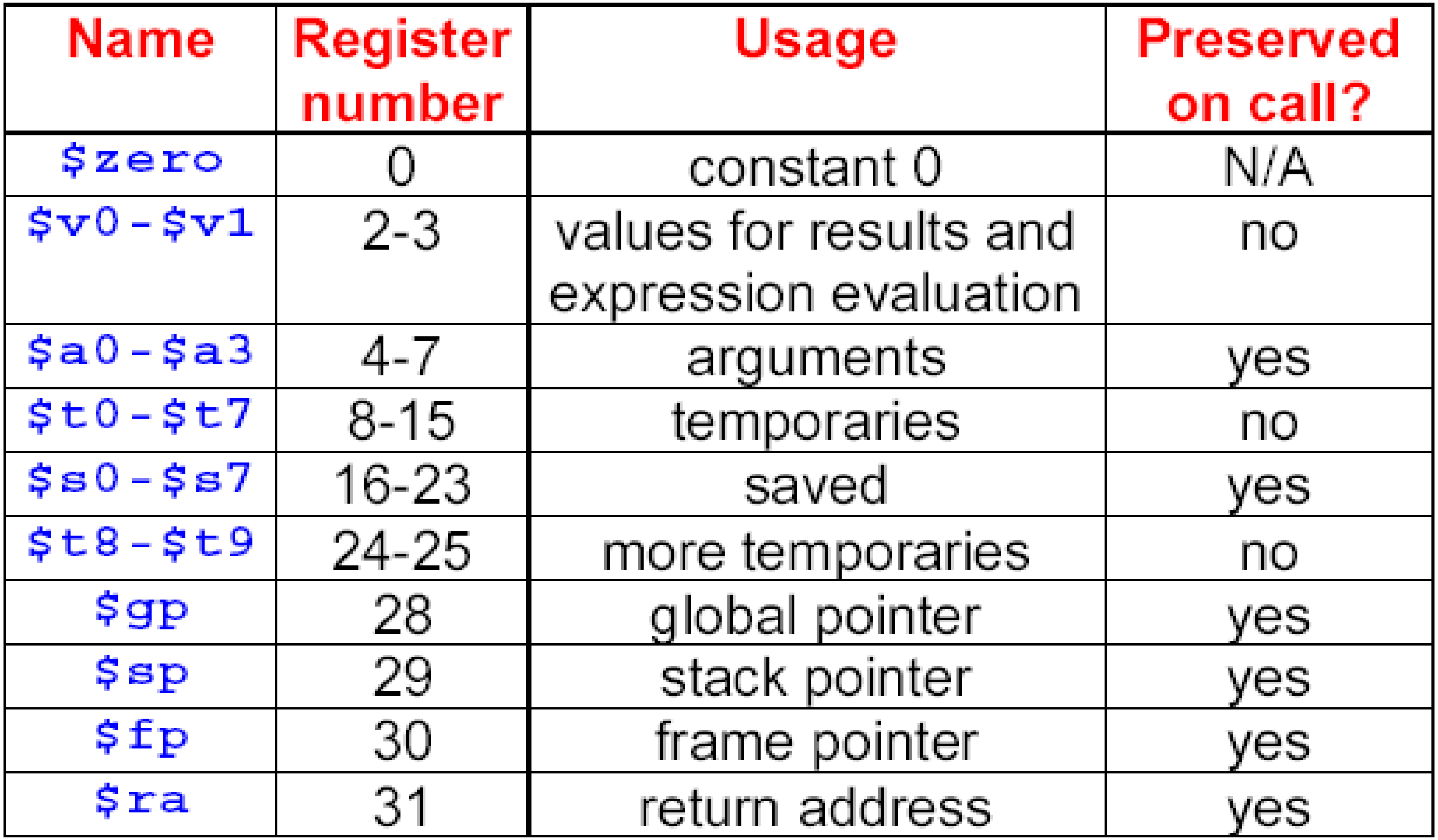
- Transferring the control to the callee:

```
jal ProcedureAddress     ; jump-and-link to the procedure address
                         ; the return address (PC+4) is saved in $ra
```

- Returning the control to the caller:

```
jr $ra                   ; instruction following jal is executed next
```

MIPS Register Convention



Name	Register number	Usage	Preserved on call?
`$zero`	0	constant 0	N/A
`$v0-$v1`	2-3	values for results and expression evaluation	no
`$a0-$a3`	4-7	arguments	yes
`$t0-$t7`	8-15	temporaries	no
`$s0-$s7`	16-23	saved	yes
`$t8-$t9`	24-25	more temporaries	no
`$gp`	28	global pointer	yes
`$sp`	29	stack pointer	yes
`$fp`	30	frame pointer	yes
`$ra`	31	return address	yes

Alternative Architectures

- **About MIPS as an ISA**
  - It's a simple/small ISA;
  - It emphasizes a small number of instructions, formats, address modes
- **Design alternative:**
  - Provide more powerful operations, and many of them
  - Goal is to reduce number of instructions executed
  - Danger is a slower cycle time and/or a higher CPI
- **Sometimes referred to as "RISC vs. CISC"**
  - RISC: Reduced Instruction Set Computing
  - CISC: Complex Instruction Set Computing

CISC vs. RISC

- CISC (Complex Instruction Set Computer)
  - Enhance the function of instructions, many kinds of operations, each instruction's function is strong
- RISC(Reduced Instruction Set Computer)
  - Reduce the function of instructions, Provide basic instructions, each instruction's function is weak
- **Two completely different directions for Instruction Set Architectures**

CISC

- **Complex Instruction Set Computer**
  - **Background:** lack of storage resource, emphasize compiler optimization
  - **Techniques:** Enhance the function of the instructions, Design some complex instructions, instead of some functions which are originally implemented by software
- CISC example was DEC VAX: min code size, make asm easy ***instructions from 1 to 54 bytes long!***

RISC

- Reduce CPI:

  CPUtime=Instr_Count * CPI * Clock_cycle

- Reduce the instruction set:

  only keep the most basic ones

- Load/Store architecture
- Simple instructions, simple addressing modes, fixed-length instruction format…

A Brief history of RISC

- Load/Store architecture:

  CDC6600(1963)--CRAY1(1976)
- IBM801(1979年),

  first RISC computer
- 1980, Patterson(Berkeley) & Ditzel

  first put forward RISC, RISC-I,Π
- 1981, Hennessy(Stanford)

  MIPS
- Commercial RISC CPU after 1985:

  MIPS1(1986) & SPARC V1(1987) ...

Some typical high performance RISC CPU

- SUN, SPARC(1987)
- MIPS, SGI:MIPS(1986)
- HP, PA-RISC,
- IBM, Motorola, PowerPC
- DEC、Compaq, Alpha AXP
- IBM RS6000(1990) first Superscalar RISC

Summary of ISA

- **Architecture = what's visible to the program about the machine**
  - Not everything in the deep implementation is "visible"
  - The name for this invisible stuff is "the implementation"
- **A big piece of the ISA = assembly language structure**
  - Primitive instructions, execute sequentially, atomically
  - Issues are formats, computations, addressing modes, encoding etc

Summary of ISA

- Two broad flavors:
  - CISC: lots of complicated instructions
  - RISC: a few, essential instructions
  - Basically all recent machines are RISC, but the dominant machine of
  - today, Intel x86, is still CISC (though they do RISC tricks in the guts…)
- Example: MIPS

History of ISA

- 60’ Stack----reduce the gap between high-level programming language and machine language.
- 70’ reduce the software cost, replacing software with hardware
- 80’ processor performance → simple ISA

90’s ISA

- Address size doubles: 32bit→ 64bit
- Optimization conditional branch via conditional execution
- Optimization of cache performance via prefetch
- Support of multimedia
- Faster floating-point operations
- Long instruction word
- Increased Conditional Execution

Homework

再见，谢谢！