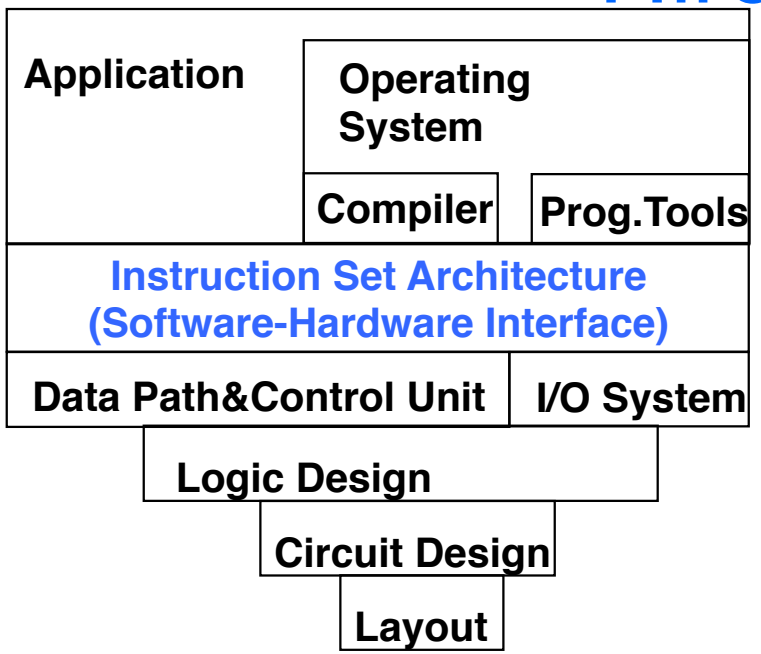
2024年
Architecture(アーキテクチャ学)

Chapter 2
Instruction Set Principles and Examples

Hiroaki Kobayashi
(小林広明)



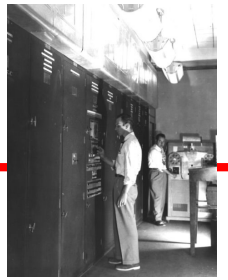
Ver. 12102024

Contents

- Von Neumann Model: An Execution Model of Computers
- Classifying Instruction Set Architectures
- Memory Addressing
- Addressing Mode
- Type and Size of Operands
- Operations in the Instruction Set
- Instructions for Control Flow
- Encoding an Instruction Set
- Example: MIPS Architecture
- Conclusions

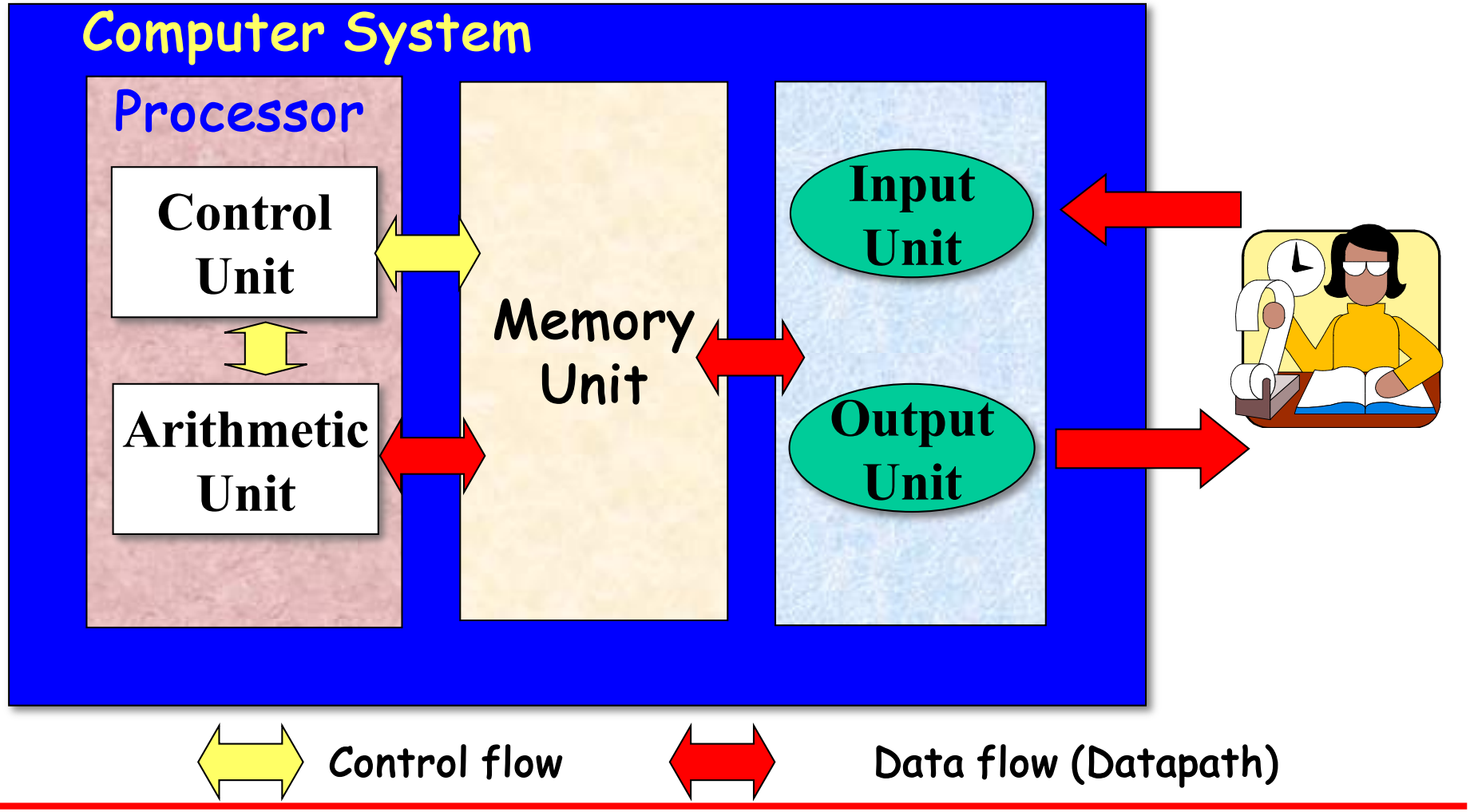
Basic Model of Modern Computers

EDVAC in 1949



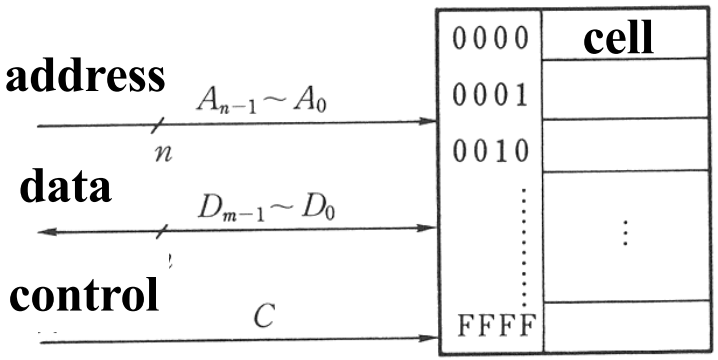
- Von Neumann Computer Model
  - Basic model of modern computers, designed and proposed by John von Neumann in 1945
    - First implementation in 1949; EDVAC (*Electronic Discrete Variable Automatic Computer*)
- Characteristics of Von-Neumann-type Computers
  - Stored program
    - Program and data are stored in memory
    - Instruction read from memory, decode, data read from memory, processing (calculation), and result store to memory
  - Linear memory space
    - Memory cells (unit for data storage) are placed as an 1D array
    - Each cell has its own address to specify a cell for data read and write
  - Simple instructions to control a computer are provided such as
    - Addition, subtraction, shift, logical AND/OR, Data move for read/write to/from memory
    - Execution sequence control
  - Sequentially-controlled computer that processes an instruction one by one
    - Program (machine instructions) is processed sequentially stored on memory
    - Special counter named program counter specifies the address of the current executing instruction.
      - Change value of program counter if you want to change the sequence of execution.

Basic Structure of Von-Neumann Computer

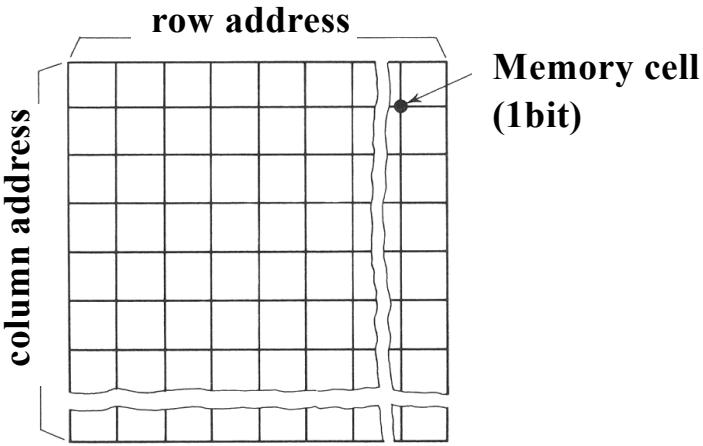


Structure of Memory

- Logical Structure (visible from programmers)
  - 1 dimensional array of memory cells
  - Each cell stores a unit of data and has its own address to specify for data read/write in memory
- Physical Structure (actual implementation)
  - Memory devices are placed in a 2-dimensional and accessed through a combination of row and column address.



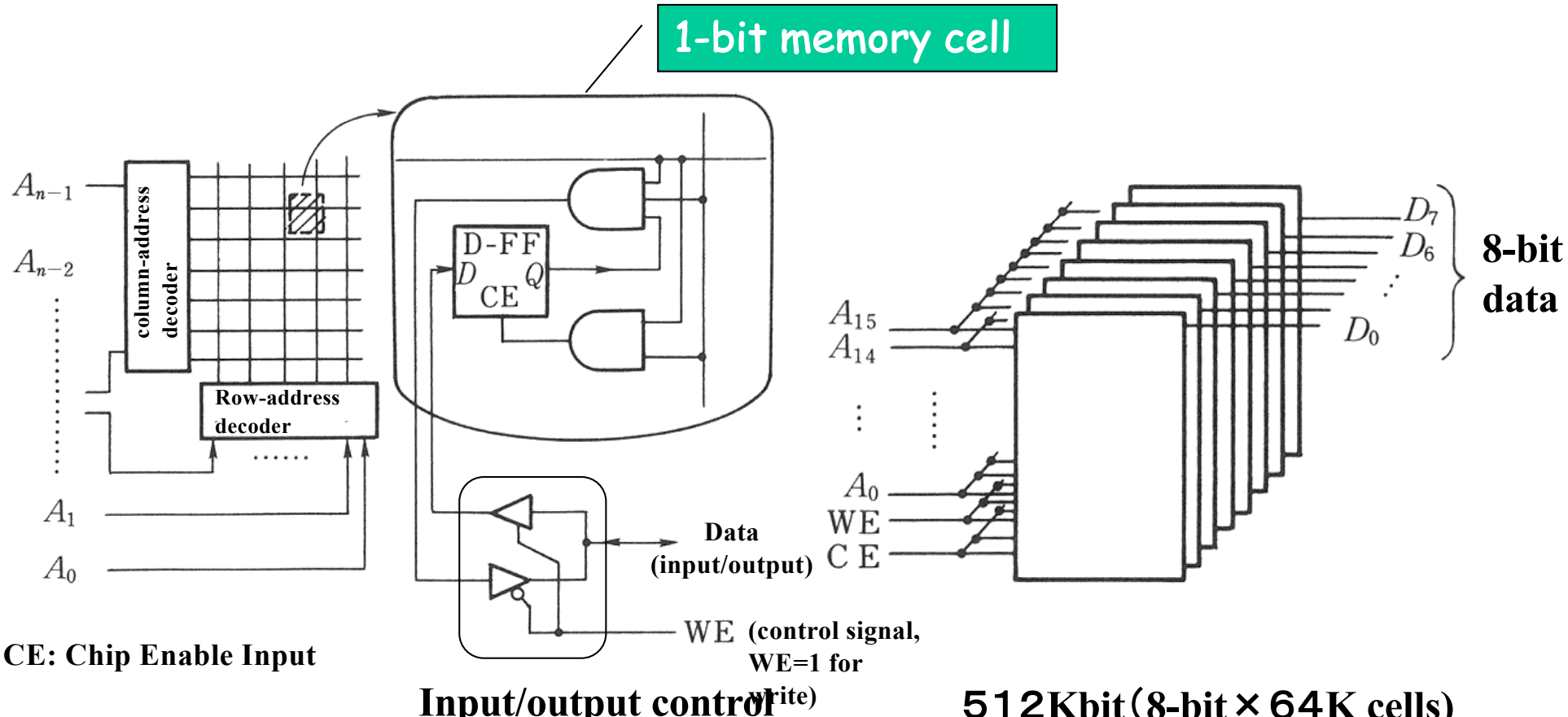
Logical structure



Physical structure (implementation)

data

Physical Memory Structure
SRAM:Static Random Access Memory



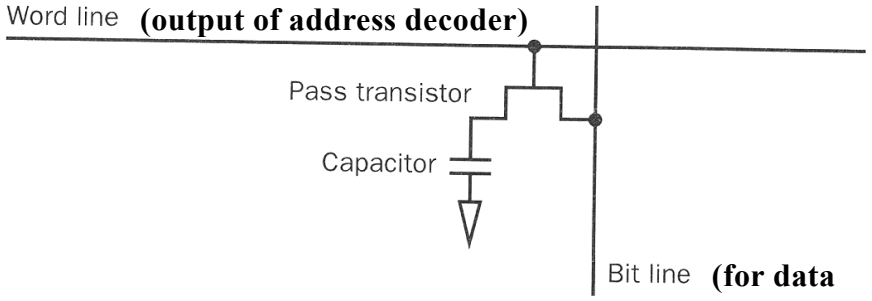
CE: Chip Enable Input

1-bit memory structure (plane)

512Kbit（8-bit×64K cells)
SRAM structure

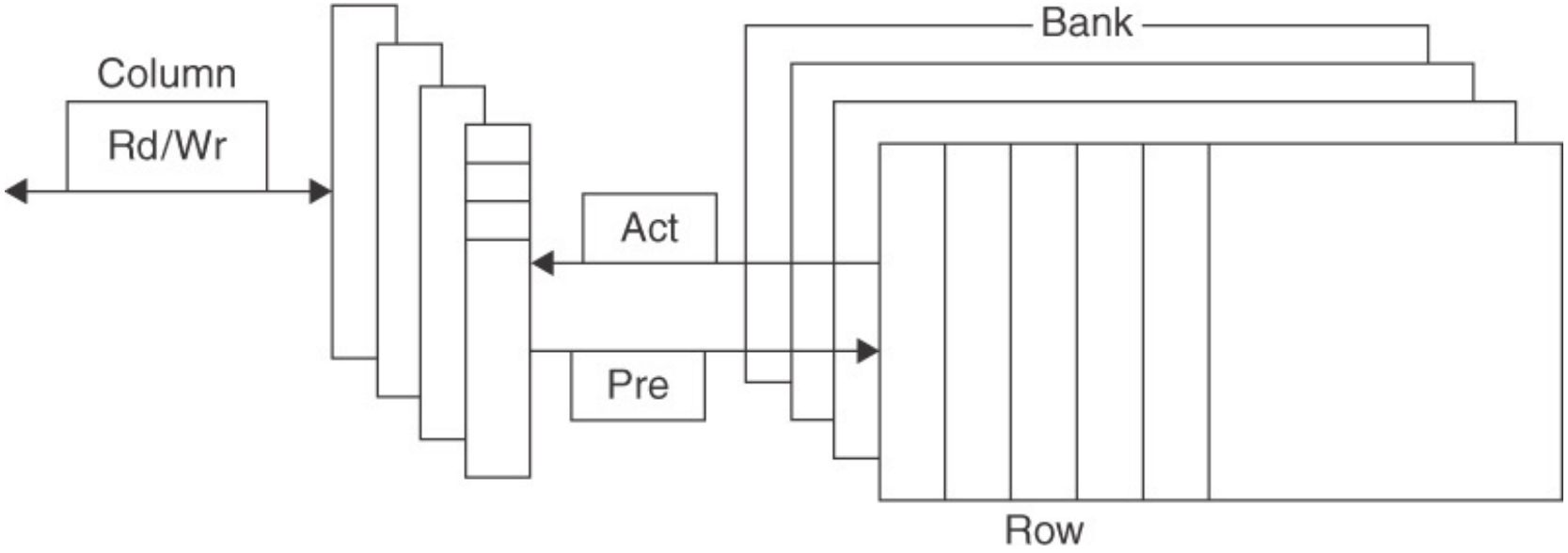
Another Implementation of Memory
DRAM:Dynamic Random Access Memory

- represent 1-bit data whether a capacitor is charged or not.
  - Need only one transistor and one capacitor for 1-bit memory
  - ☺ Less hardware compared with SRAM (1/4)
    - ✓ More memory capacity on the same area
  - ✓ Since real capacitors leak charge, the information eventually fades unless the capacitor charge is refreshed periodically.
  - ☹ Need periodical refreshment of the memory content
    - So named dynamic memory
    - Longer access time for read/write of DRAM



1-bit cell of DRAM

DRAM Internal Structure

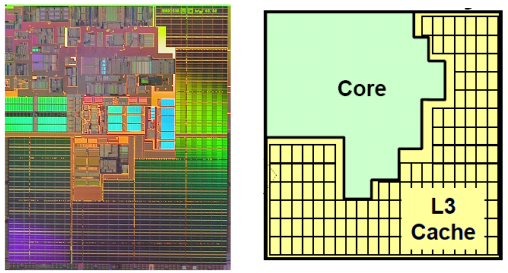


Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

DIMM

SRAM vs. DRAM

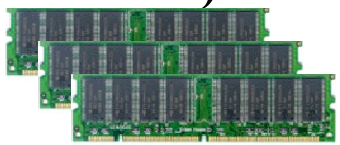
- SRAM:Static Random Access Memory
  - Use D-FF(equivalent to 4 Transistors) for 1-bit storage
    - Stable(static) memory
    - Fast memory access
      - Access time: 0.5ns-5ns
    - High (hardware) cost/1bit



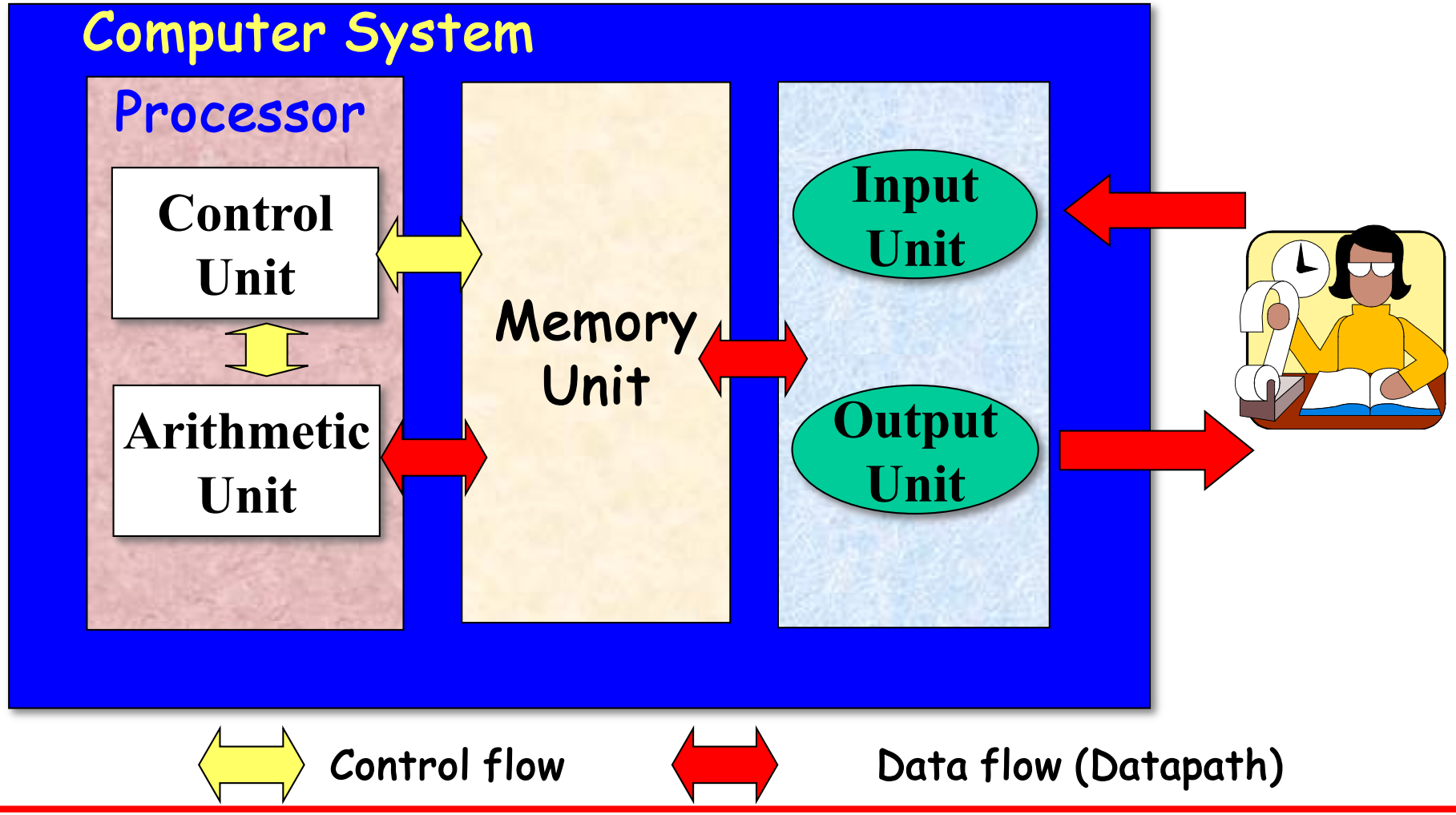
For speed-oriented memory
⇨on-chip(processor) memory such as register and cache

- DRAM:Dynamic Random Access Memory
  - Use one transistor and one capacitor for 1-bit storage
    - More memory capacity at lower cost
    - Unstable and need refreshment of contents
      - Dynamic operation
    - Refresh mechanism enlarges memory access time
      - Access time: 50～70ns

For capacity oriented memory
⇨off-chip(processor) memory such as main memory (SIMM/DIMM)

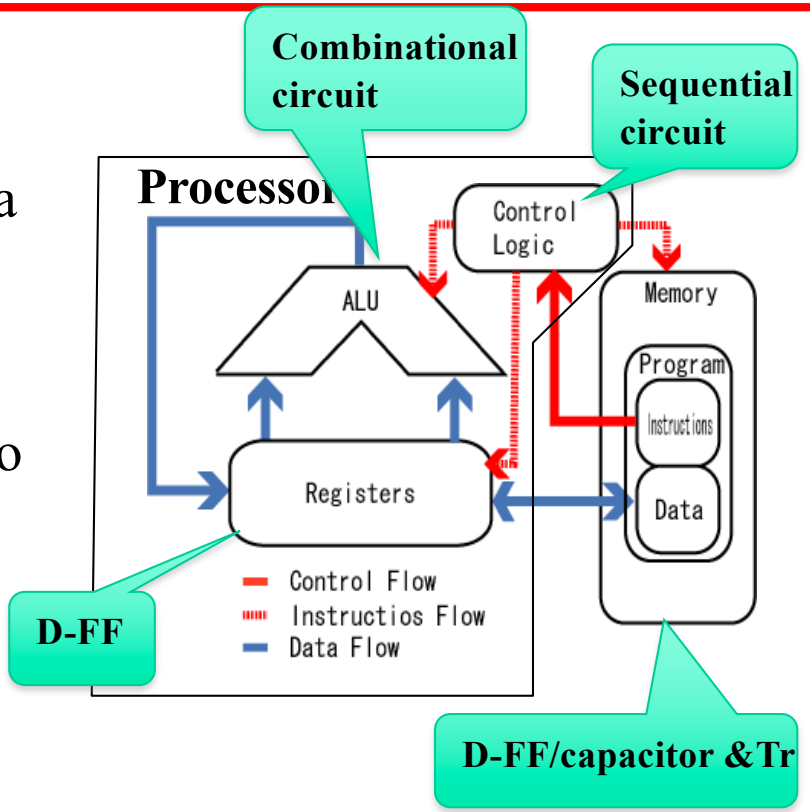


Basic Structure of Von-Neumann Computer



Memory Operations

- Data read from memory
  - Specify an address to read data and a register to have data in processor
    - A register is an internal memory in processor
  - Move data from memory specified to the register
- Data write to memory
  - prepare data (calculation result) in a register
  - Specify an address to write data
  - Move data from the register to memory specified.

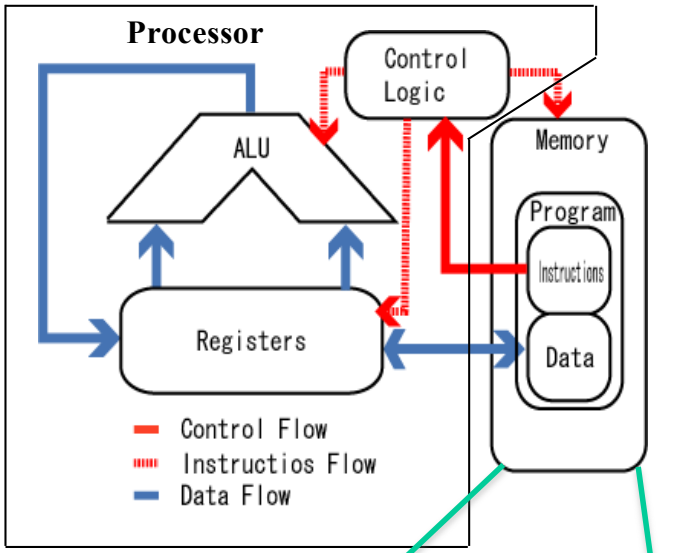


Basic Operation of Computer : Data Processing

- Basic behavior for data processing
  - Move data from memory to processor,
  - Perform an operation on data
  - Store the operation result to memory

- Example: perform an addition of two values
  - y = a + b
  - Data y,a,b are placed in memory
  - Addition is performed in a processor by using an adder (combinational circuit)

- Processing of y=a+b is realized as a sequence of basic instructions
  - Execute the following four instructions sequentially
    - Instructions 1 and 2 for data movement from memory locations a and b to internal registers
    - Instruction 3 to perform addition of data stored in the registers, and store the result in (another) register
    - Instruction 4 for data movement from register to memory location for y



Program counter

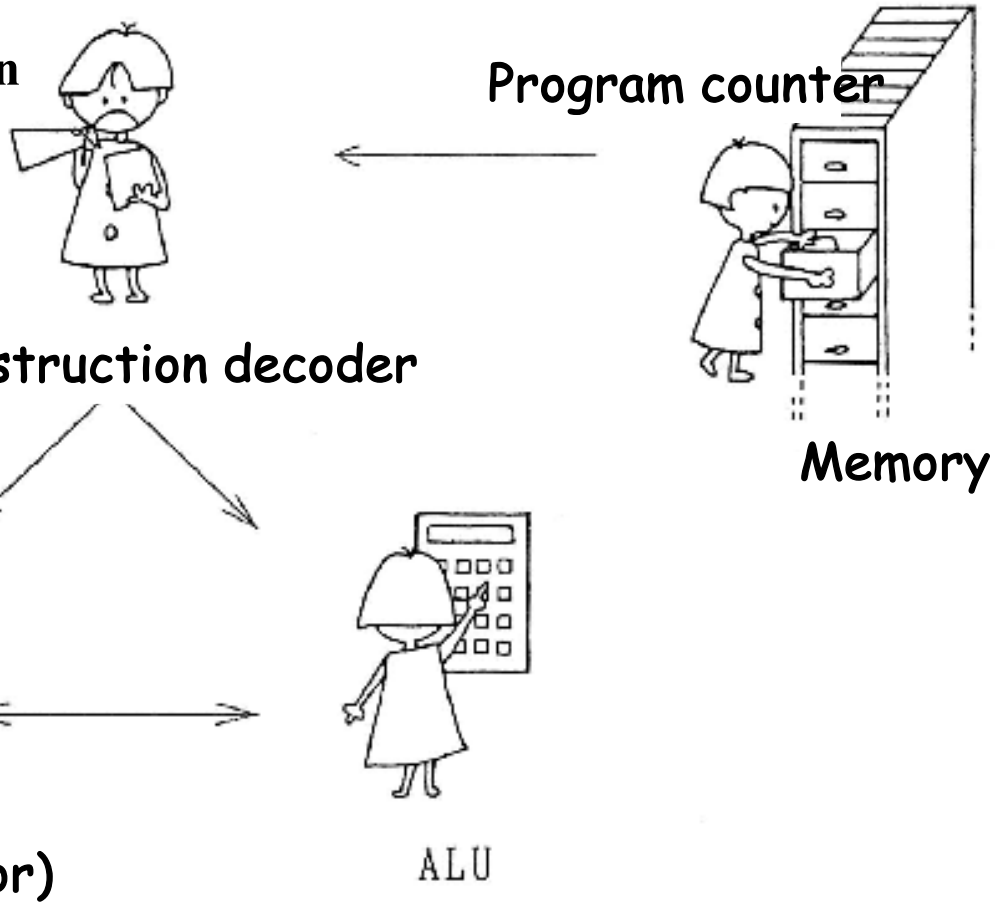
	Address	Contents
0	0	Instruction1
1	1	Instruction2
2	2	Instruction3
3	3	Instruction4
	...	
	a	Data of a
	b	Data of b
	y	Data of y

Sequential Processing: Basic Execution Control for Computer

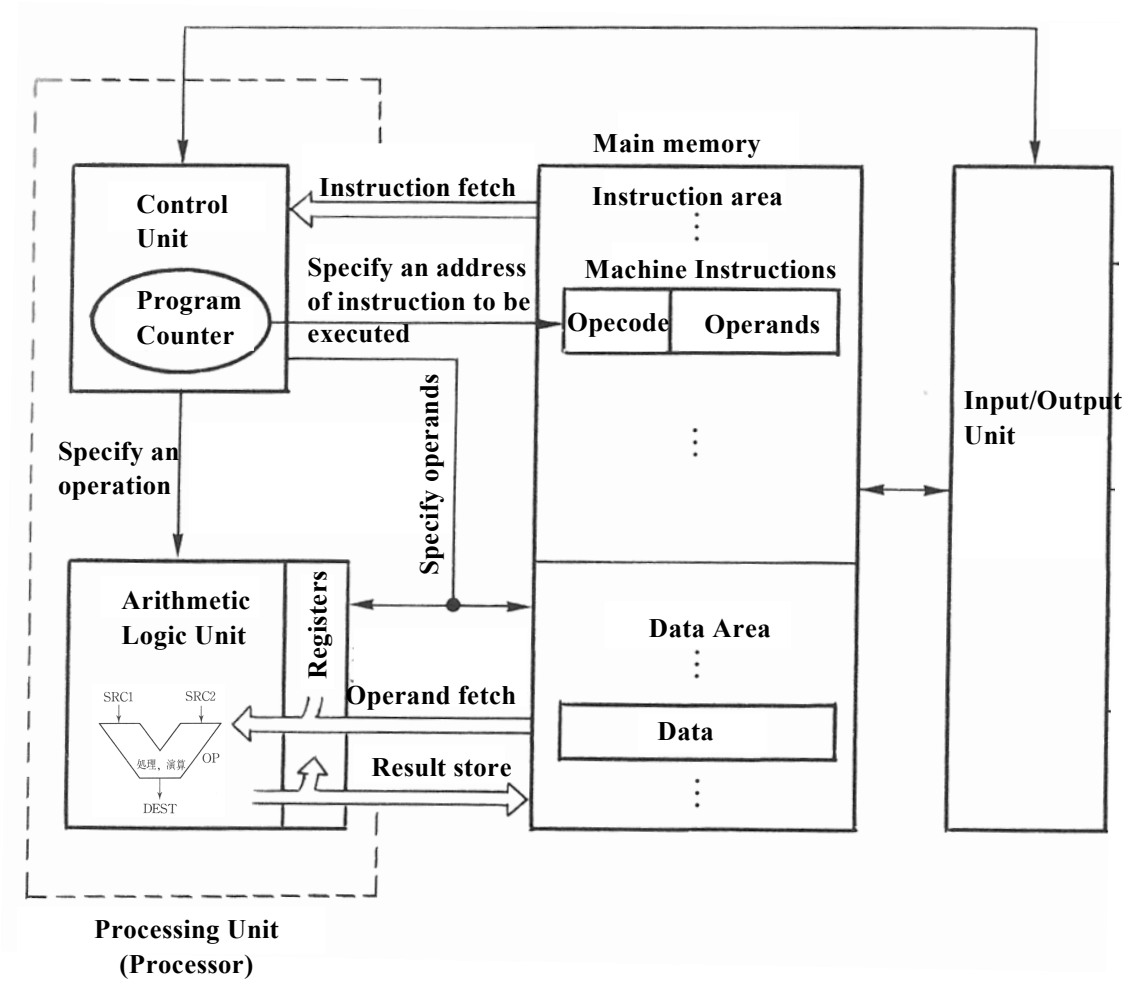
1. Computer sequentially processes instructions in order of memory locations
   - Sequential processing
   - Current memory location of an instruction processed by computer is held by a program counter (PC)
   - Sequential processing is carried out by incrementing the content of PC
   - ➢ Basic operations specified by instructions are: data movement to/from memory, arithmetic operations
2. In addition, special instructions (named JUMP and BRANCH) for execution control are prepared to change the order of instruction execution for handling
   - if…then…else, and
   - loop (iteration)
   - Action: change the content of PC to the destination address specified by JUMP or BRANCH instruction

Execution Control of a Computer

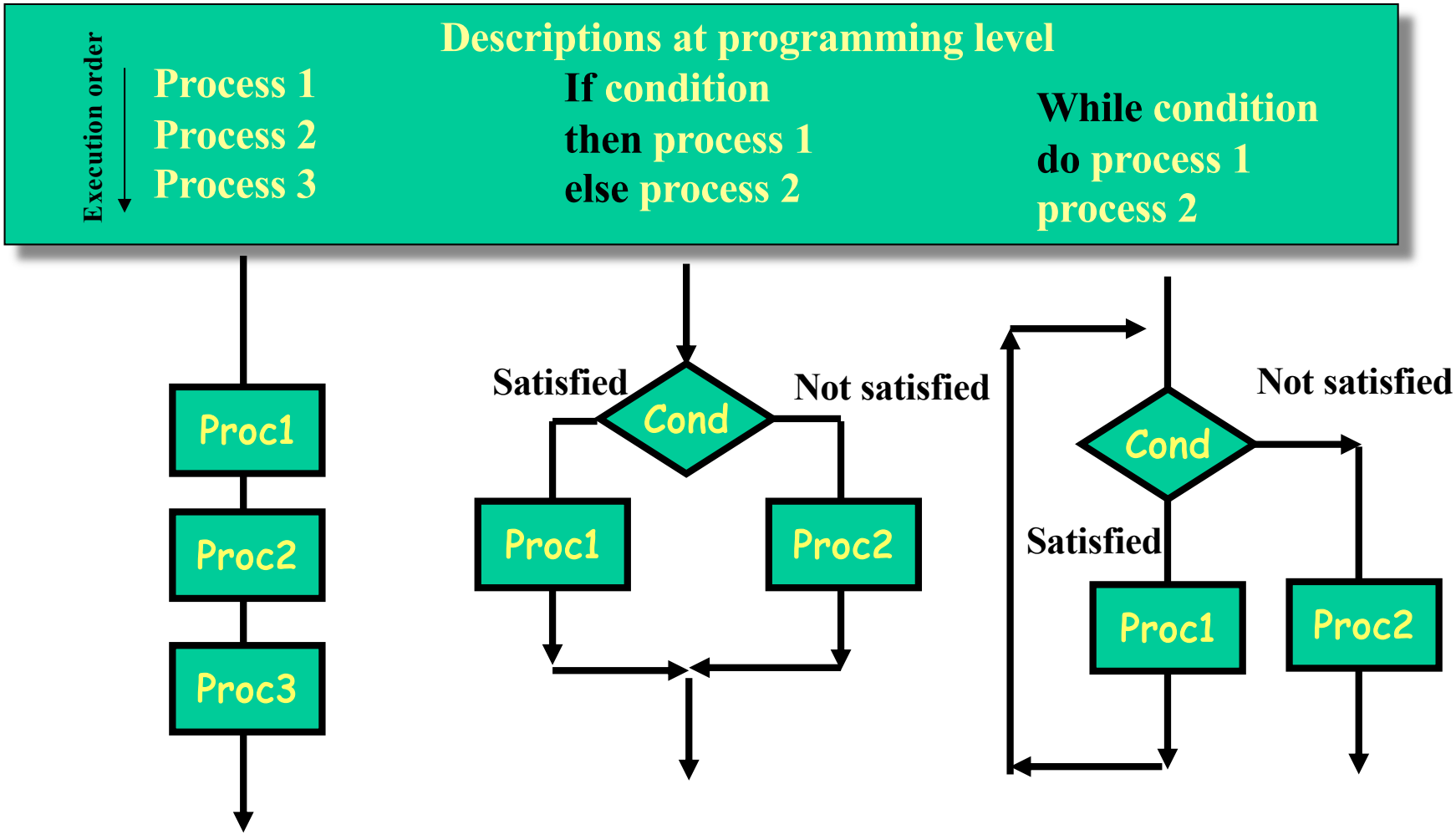
Flow of instruction execution (Fetch/Decode/Exection of Instructions)



Control and Data Path of a Computer



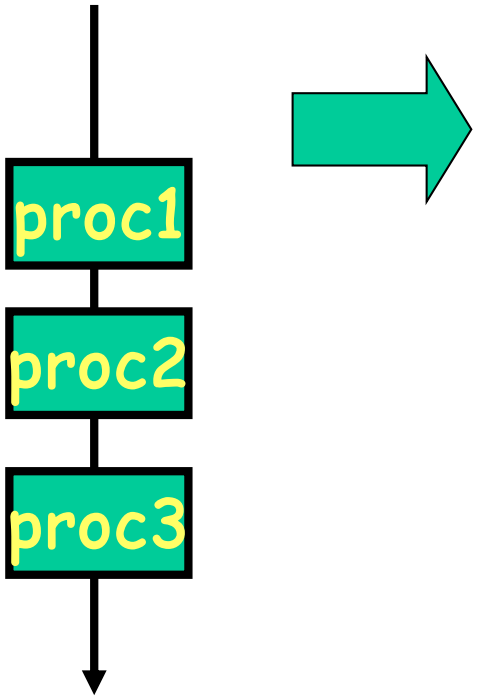
Three Basic Execution Flow Available on Computers



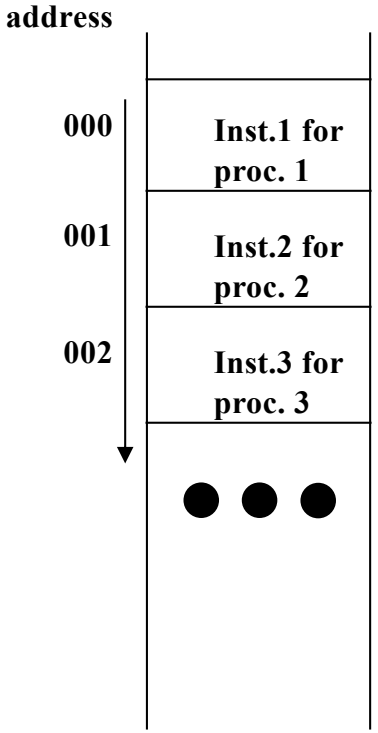
Execution Control of Computer: Sequential Execution of Instructions

Program Description

Sequential execution of process1, process 2, process 3



Memory allocation for instructions



Behavior of computer for instructions execution

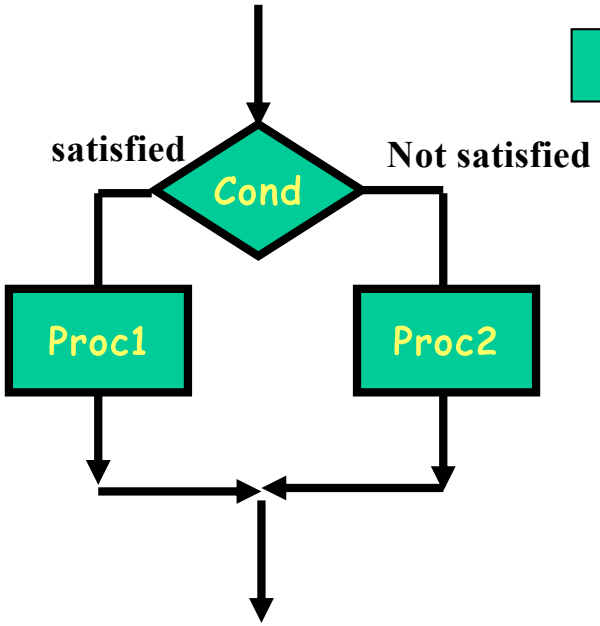
Step1：Fetch/decode/execute instruction1
Step2：Fetch/decode/execute instruction2
Step3：Fetch/decode/execute instruction3

・・・（Fetch/decode/execute following instructions）

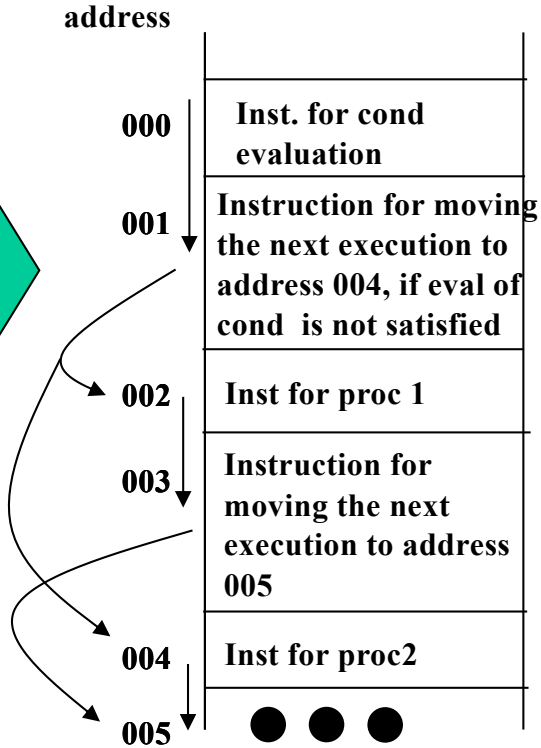
Execution Control of Computer: Execution of Conditional Branches

Program description

If condition
then proc 1
else proc 2



Memory allocation for instructions



Behavior of computer for instructions execution

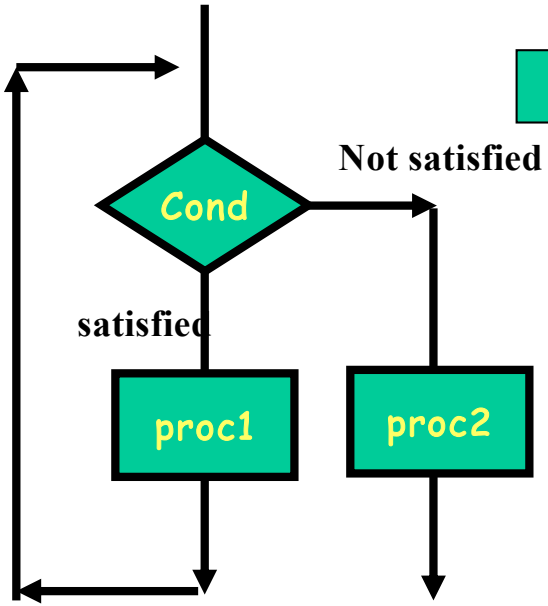
Step1: fetch/decode/execute instruction for evaluating the condition at address 00(
Step2: fetch/decode/execute instruction for instruction for moving the next execution at address 004, if condition is not satisfied, otherwise, execute the next instruction at 002
(if condition satisfied)
Step3: fetch/decode/execute instruction for proc1
Step4: fetch/decode/execute instruction for moving the next execution at address 005
(if condition not satisfied)
Step3: fetch/decode/execute instruction for proc2

・・・（Fetch/decode/execute following instructions at address 005 or later）

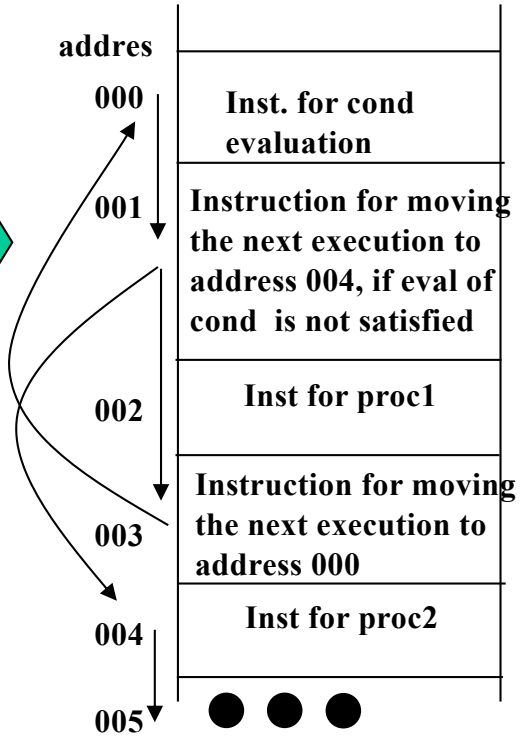
Execution Control of Computer: Execution of Loop (Iterations)

Program description

While condition
do proc1
proc2



Memory allocation for instructions



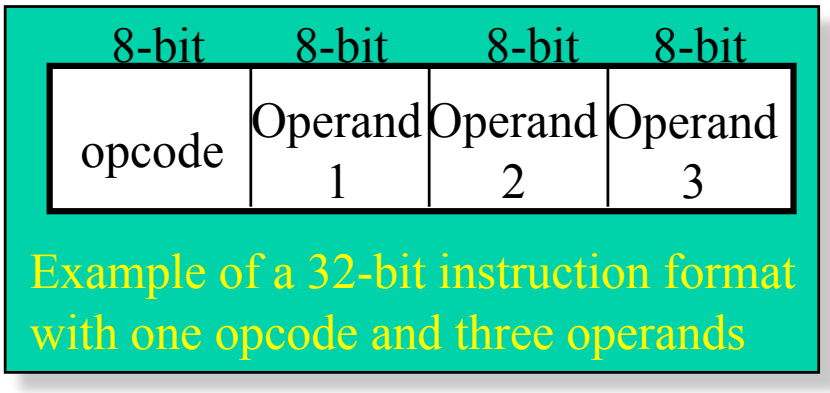
Behavior of computer for instructions execution

Step1: fetch/decode/execute instruction for evaluating the condition at address 000
Step2: fetch/decode/execute instruction for moving the next execution at address 004, if condition is not satisfied, otherwise, execute the next instruction at 002
(if condition satisfied)
Step3: fetch/decode/execute instruction for proc1
Step4: fetch/decode/execute instruction for moving the next execution at address 000
・・・Repeat Steps 1 to 4・・・
(if condition not satisfied after several iteration)
Step3: fetch/decode/execute instruction for proc2
・・・(Fetch/decode/execute following instructions at address 005 or later)

How Machine Instructions is Organized

An instruction has a fixed bit width (for example, 32-bit) and consists of an operation code and its operands

- Opecode (OPEration CODE) specifies an operation to be performed
  - Arithmetic operations
    - AND, OR, ＋, －, ×, …
  - Data movement
    - load/store instructions
  - Execution Control
    - Unconditional Jump
    - Conditional Branch



Example of a 32-bit instruction format with one opcode and three operands

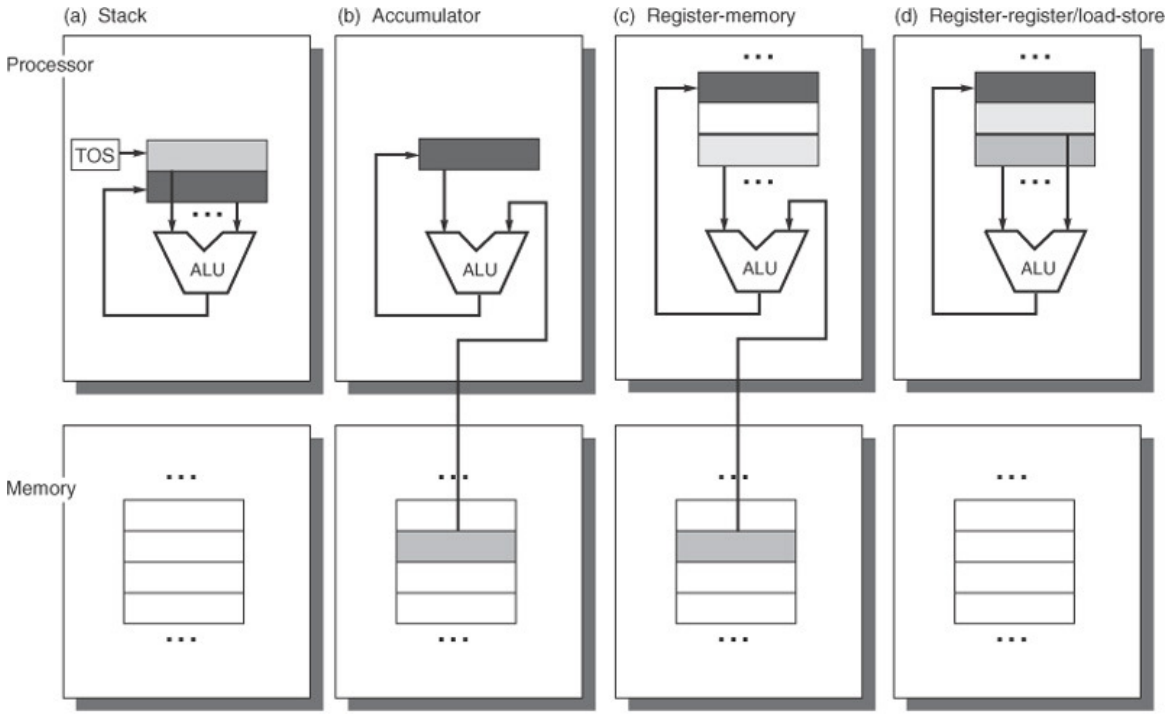
- Operands is a one of inputs/output (arguments) of an opcode
  - 0 ~ 3 operands are specified depending on the type of architectures
- Bits of an instruction are divided into the opcode field and operand fields

Seven Dimension of an ISA

- Class of ISA
  - General-purpose register architectures
  - ✓ Register-memory ISAs of 80x86
  - ✓ Load-store ISAs of MIPS
- Memory addressing
  - Byte-addressing
  - Aligned (MIPS)
  - Not aligned (80x86)
- Addressing modes
  - register, immediate, displacement, register indirect, based with scaled index
- Types and sizes of operands
  - 8-bit (char), 16-bit (unicode char, half-word), 32-bit (integer or word), 64-bit (double word or long int), and IEEE 745 fp in 32-bit(single)&64-bit(dobule), 80-bit fp in 80x86)
- Operations
  - Data transfer, arithmetic logical, control, and floating point
  - MIPS is simple but 80x86 has a much richer and larger set
- Control flow instructions
  - Cond branch, uncond jump, procedure call and return
  - PC-relative addressing
  - MIPS cond branches test the contents of registers
  - 80x86 branches test condition code bits set as side effects of operations
- Encoding an ISA
  - Fixed length (32-bit fix in MIPS) and variable length (1 to 18 bytes in 80x86)
  - Variable length inst. can take less space, but needs complicated decoding
  - # of registers and # of addressing modes have a significant impact on size of instructions

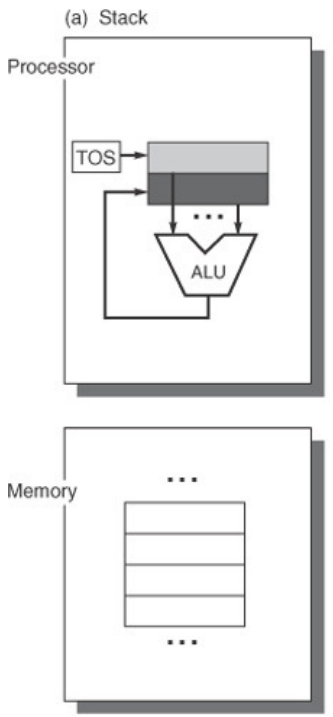
Classifying Instruction Set Architecture

- Four Instruction Set Architecture Classes
  - The difference in the classes is operand locations!
  - (Operands may be named explicitly or implicitly)

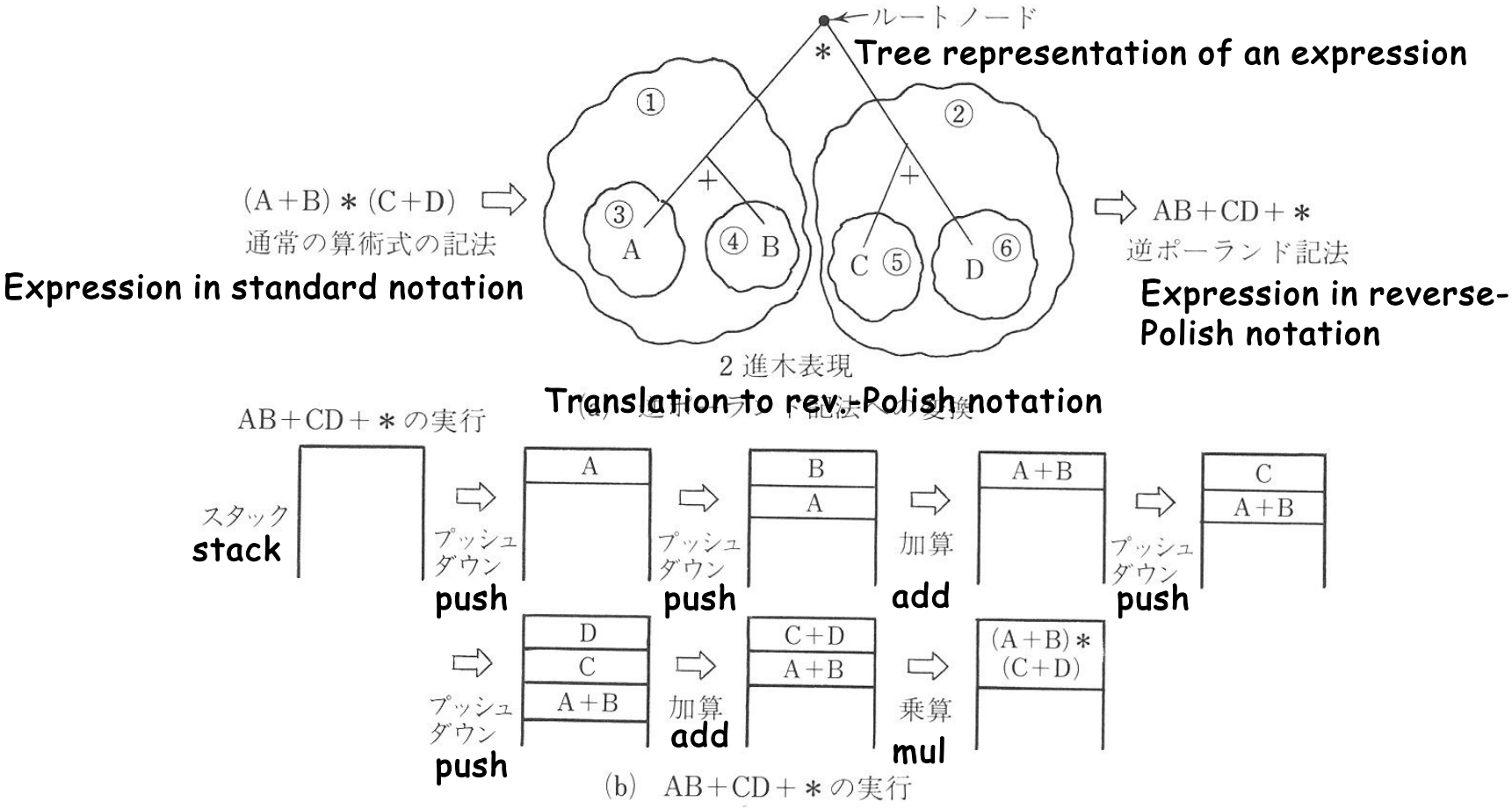


Stack Architecture

- ✓ The operands are implicitly on the top of stack
  - A Top of Stack register (TOS) points to the top input operand and the second input operand below.
  - The first operand is removed from the stack
  - The result is written back to the place of the second operand, and
  - TOS is updated to point to the result
  - ◆ Push/Pop are separate instructions for data transfer between the stack and memory
  - ◆ 0-operand instructions for operations and 1-operand instructions for push/pop operations are used.
    - ◆ Ex. ADD, Push (memory), Pop (memory)
- Examples:
  - Burroughs B5000(1961) , Intel X87/X86FPU(1980), Java virtual Machine (1995)



Expression Evaluation on the Stack Architecture



Tree representation of an expression

Expression in standard notation

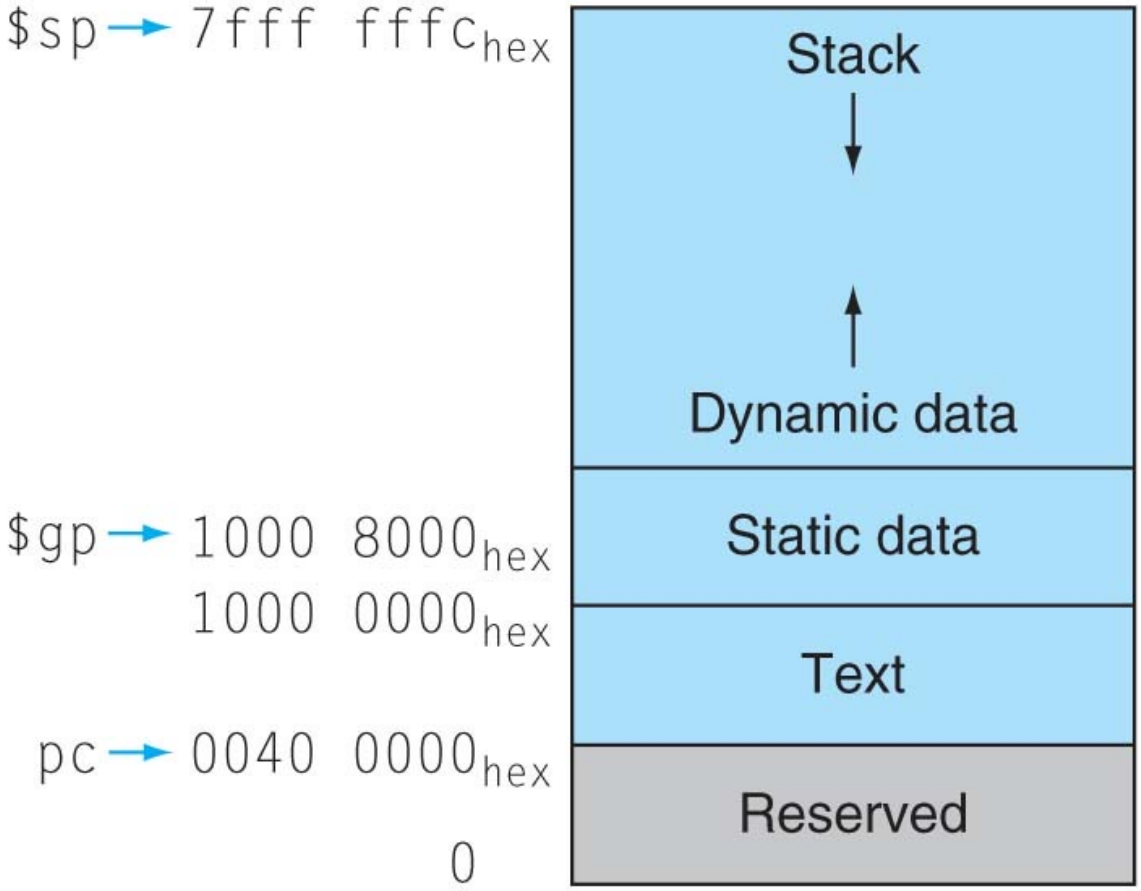
Expression in reverse-Polish notation

Translation to rev.-Polish notation

Execution of an expression in rev.-Polish notation

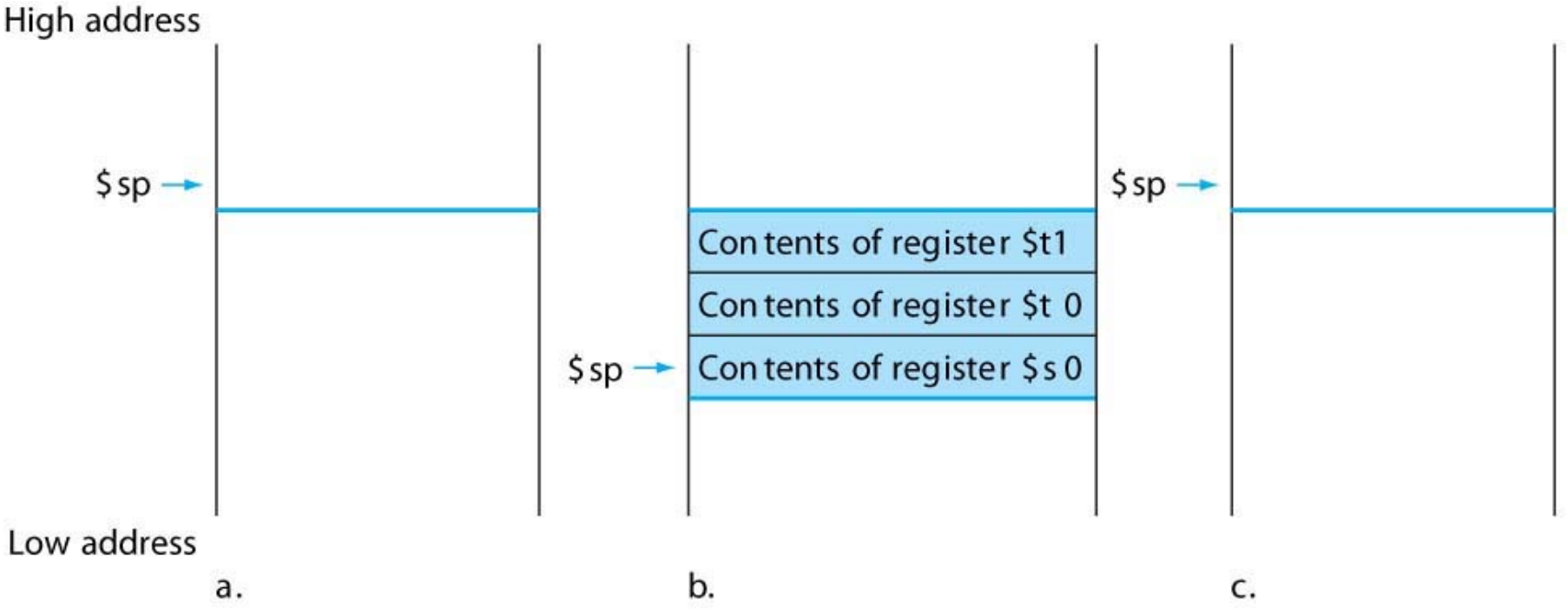
Stack Control for Procedure Calls (1/2)

- Memory map of the MIPS processor



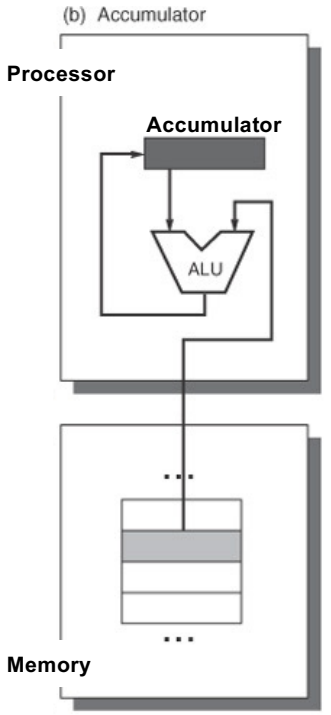
Stack Control for Procedure Calls (2/2)

Stack (a) before, (b) during, and (c) after the procedure call.



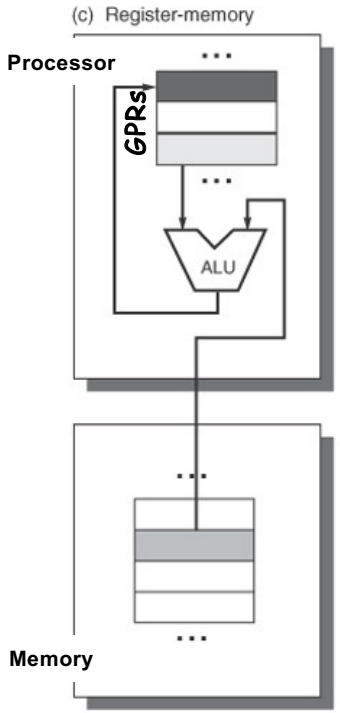
Accumulator Architecture

- ✓ One operand is implicitly coming from the accumulator, and the other operand is explicitly from the memory location.
  - The accumulator is a special register used for both an implicit operand and a result.
  - Data should be loaded/stored into/from the accumulator from/to memory before/after an ALU operation via a separate instruction (load/store)
  - 1-operand instruction format used
    - Ex. ADD (memory location)
  - Examples: almost all early computers are accumulator machines (Intel 8080@1974), but many microcontrollers such as Freescale 68HC12 and PICmicro are still accumulator machines



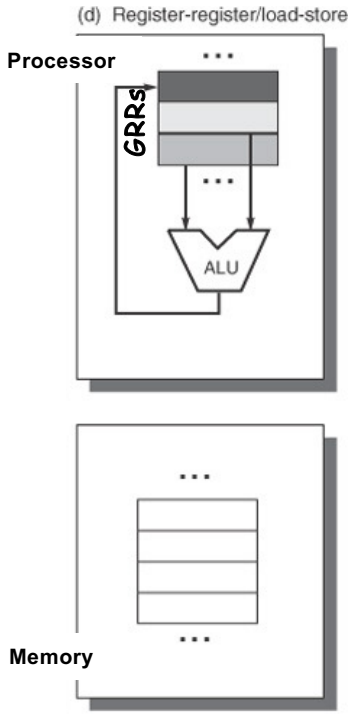
General-Purpose Register Architecture: Register-Memory Architecture

- ✓ Operands are explicitly specified; ether register or memory location
  - More transistors used for internal storage: general purpose registers (GPRs)
  - One input operand is a register, one is in memory, and the result goes to a register
    - Increase the flexibility in register allocation by compiler
    - Provide sophisticated addressing modes for efficient data handling
  - 2- or 3-operand instruction format used
    - Ex. Load (Reg.) (memory),
    - ADD (Reg.2) (Reg.1) (memory).
  - Examples: IBM360/370 (16 registers@1964) , X86(8086@1978), MC68000@1979



General-Purpose Register Architecture: Register-Register Architecture

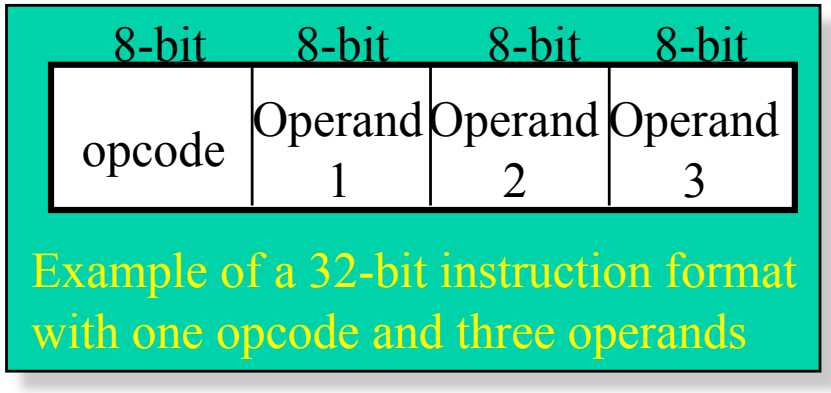
- ✓ Operands are explicitly specified only on GPRs
  - Memory can be accessed only with load/store instructions
  - Virtually every new architecture designed after 1980 uses a load-store register architecture
    - Registers are faster than memory
    - Registers are more efficient for a compiler to use
    - Registers can be used to hold variables
      - Memory traffic reduces
      - Program speeds up
      - Code density improves
  - 2- or 3-operand instruction format
    - Ex. Load Reg. (Memory), ADD Reg. 3 Reg.2 Reg.1
  - Examples: MIPS(1981) , SPARC(1985), PowerPC(1991)



Format of Machine Instructions

An instruction has a fixed bit width (for example, 32-bit) and consists of an operation code and its operands

- Opecode (OPEration CODE) specifies an operation to be performed
  - Arithmetic operations
    - AND, OR, ＋, －, ×, …
  - Data movement
    - load/store instructions
  - Execution Control
    - Unconditional Jump
    - Conditional Branch

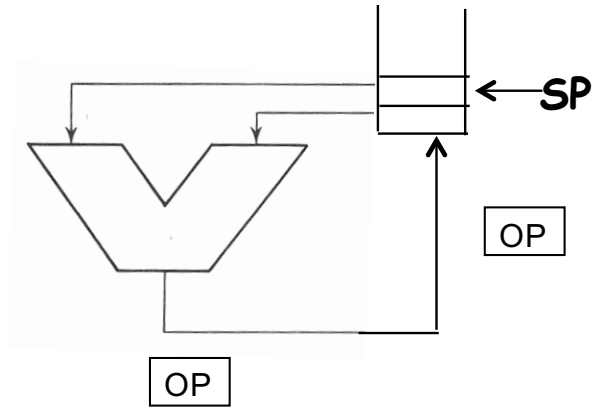


Example of a 32-bit instruction format with one opcode and three operands

- Operands is a one of inputs/output (arguments) of an opcode
- Bits of an instruction are divided into the opcode field and operand fields

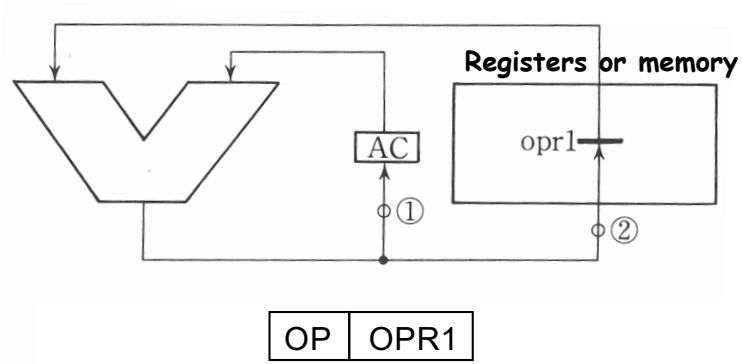
Classification of Instructions based on Number of Operands: Arithmetic Operations

- Zero-operand instruction
  - Operands are implicitly defined by the stack pointer
  - The result also goes to the memory location specified by the stack pointer.



Instruction Format

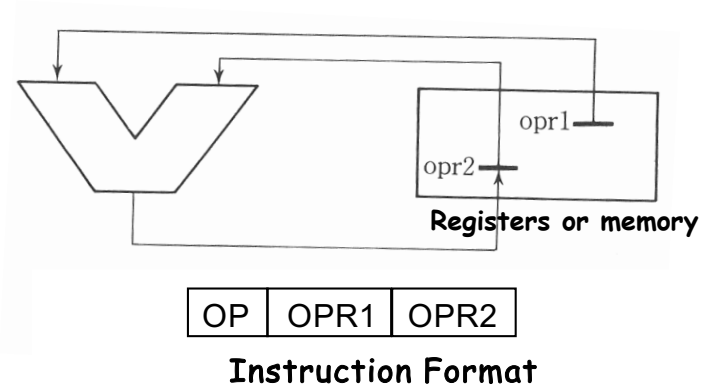
- One-operand instruction
  - Specify one operand and the others are defined implicitly
  - Implicitly defined internal register is called Accumulator



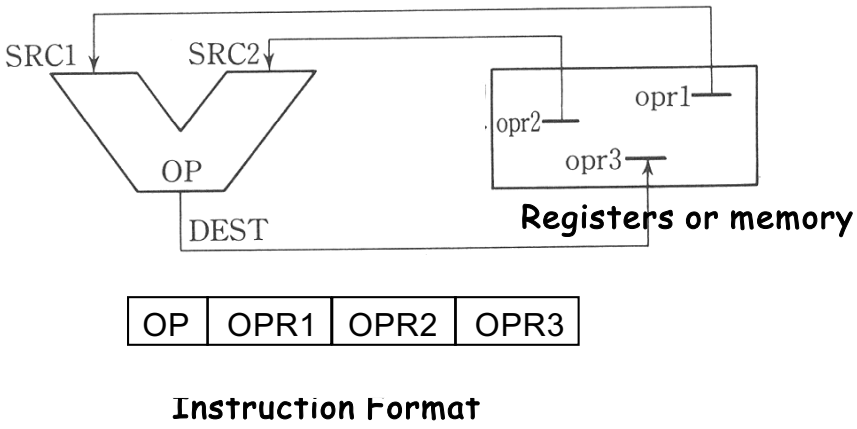
Instruction Format

Classification of Instructions based on Number of Operands : Arithmetic Operations (Cont'd)

- Two-operand instruction
  - Two operands specified, and one of them is also used as a destination (output)



- Three-operand instruction
  - Three operands, two for inputs and one for output, are specified



Code Sequence Comparison

Code sequence for C=A+B for four classes of instruction sets

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

0/1 operand instructions

1 operand instructions

2/3 operand instructions

2/3 operand instructions

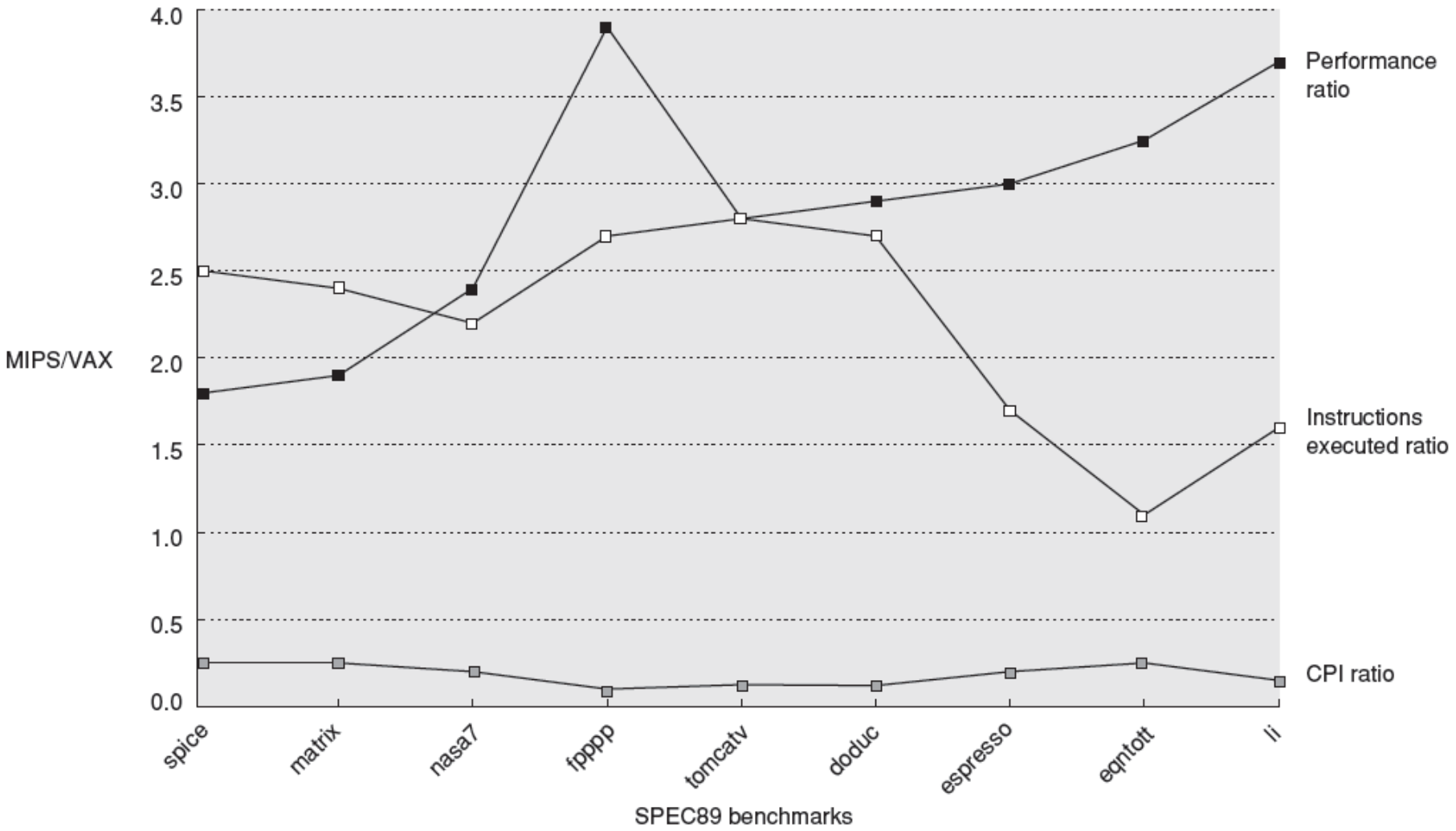
More Implicit operands

More Explicit operands

How many Operations Should be Coded in a Single Opcode? CISC vs. RISC

- CISC: Complex Instruction Set Computer
  - Has rich instruction sets and is designed to simplify compilation of high-level languages
    - Narrowing the gap between high-level languages and machine instruction
    - Optimizing code size because compiled programs were often too large for available memories in 1970s ~ 1980s
    - IBM 360/370, VAX-11/780, Intel X86 ISA
- RISC: Reduced Instruction Set Computer (1980~)
  - Has commonly used instructions only and make them fast in execution.
    - Make common case fast and keep the system simple
      - Rarely executed instructions slow the entire control of the system
      - The increase in memory size on computers eliminated the code size problems arising from high-level languages and enabled operating systems to be written in high-level languages
    - Simple instructions and their fixed size and format make pipelining much more efficient, running at higher frequency.
    - Intel Xeon (Internal translation from X86 inst. to risc-ops), MIPS, SPARC, PowerPC, PA-RISC, DEC Alpha, ARM, SuperH…

Ratio of MIPS M2000 vs. VAX 8700



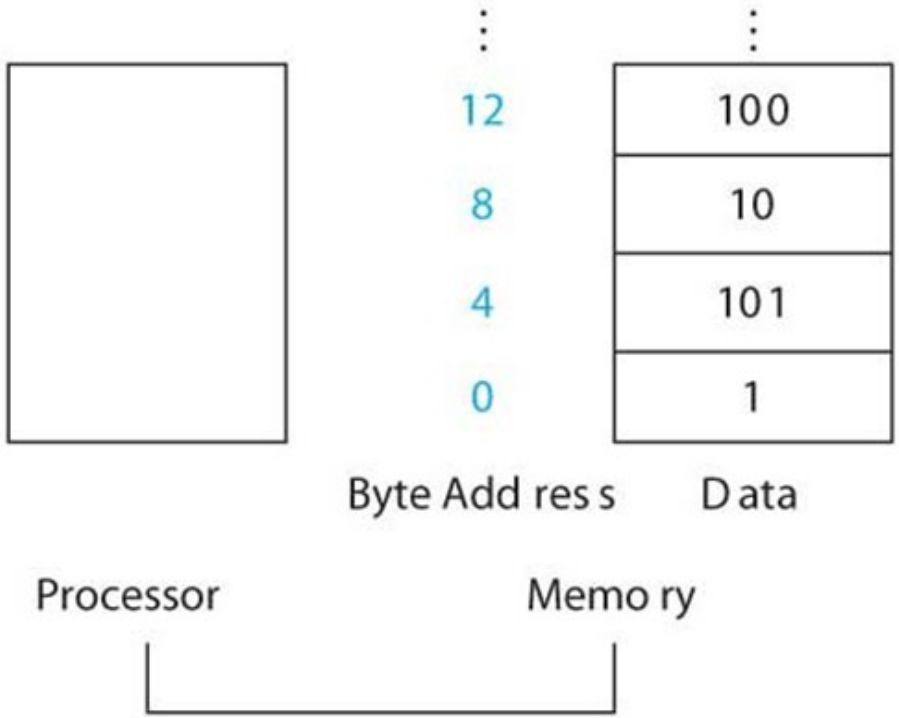
Comparison of GPR Architectures

Number of memory addresses	Maximum number of operands allowed	Type of Architecture	Examples
0	3	Load-store	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH
1	2	Register-memory	IBM 360/370, Intel X86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand format)
3	3	Memory-memory	VAX (also has two-operand format)

Type	Advantages	Disadvantages
Register-register (0,3)	Simple, fixed-length instruction encoding Simple code generation model. Instructions take similar number of clocks to execute	Higher instruction count than others. More instructions and lower instruction density leads to larger program.
Register-memory(1,2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers Clocks per instruction vary by operand location.
Memory-memory (2,2) or (3,3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (not used today)

Memory Addressing: How Instructions and Data are Placed in the Memory

- **MIPS uses fixed size instructions (4-byte width), its memory addressing is Byte-addressed and Aligned.**



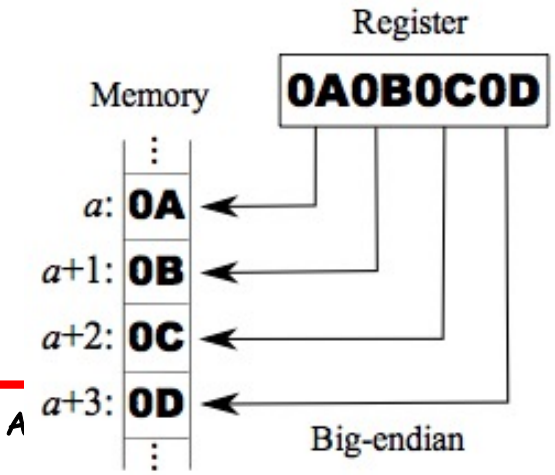
- **X86 uses variable size instructions (1-byte to 17-byte width), its memory addressing is Byte-addressed but Not Aligned.**

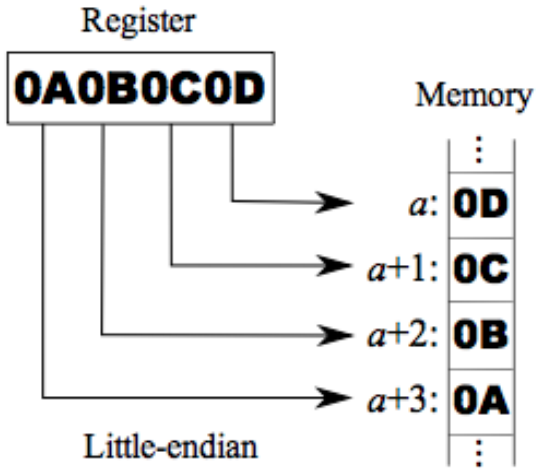
Example of Machine Instructions on X86

```
00401000 <_WinMainCRTStartup>:
 401000:    55                  push   %ebp
 401001:    89 e5               mov    %esp,%ebp
 401003:    83 ec 18            sub    $0x18,%esp
 401006:    83 e4 f0            and    $0xfffffff0,%esp
 401009:    a1 00 30 40 00      mov    0x403000,%eax
 40100e:    85 c0               test   %eax,%eax
 401010:    74 01               je     401013
<_WinMainCRTStartup+0x13>
 401012:    cc                  int3
 401013:    d9 7d fe            fnstcw -0x2(%ebp)
 401016:    0f b7 45 fe         movzwl -0x2(%ebp),%eax
 40101a:    66 25 c0 f0         and    $0xf0c0,%ax
 40101e:    66 89 45 fe         mov    %ax,-0x2(%ebp)
 401022:    0f b7 45 fe         movzwl -0x2(%ebp),%eax
 401026:    66 0d 3f 03         or     $0x33f,%ax
 40102a:    66 89 45 fe         mov    %ax,-0x2(%ebp)
 40102e:    d9 6d fe            fldcw  -0x2(%ebp)
 401031:    c7 04 24 40 10 40 00    movl   $0x401040,(%esp)
 401038:    e8 33 01 00 00      call   401170 <_cygwin_crt0>
 40103d:    c9                  leave
 40103e:    c3                  ret
 40103f:    90                  nop
```

Endianness: Big-Endian vs. Little-Endian (1/2)

- Endianness: the ordering of individually addressable sub-units (words, bytes, or even bits) within a longer data word stored in external memory.
  - How a 16-, 32- or 64-bit word (multi-byte data) is stored in a byte-addressed memory???
- Big-Endian: Most significant byte first
  - MC6800/68000, PowerPC, IBM System/370, SPARC(until version 9) , JAVA VM
- Little-Endian: Least significant byte first
  - x86, 6502, Z80, VAX
- Bi-Endian: switchable endianness
  - ARM, PowerPC, Alpha, SPARC V9, MIPS, PA-RISC, IA-64, Crusoe





Endianness: Big-Endian vs. Little-Endian (2/2)

```c
#include<stdio.h>
union tag_uEndian
{
int in;                  /* 4Byte        */
unsigned short sh[2];    /* 2Byte×2      */
unsigned char ch[4];     /* 1Byte×4      */
};
int main()
{
        union tag_uEndian En;
        En.in = 0x12345678;
        printf("INPUT is 0x%x\n",En.in);
        printf("short is 0x%x, 0x%x\n",En.sh[0],En.sh[1]);
        printf(" char is 0x%x, 0x%x, 0x%x, 0x%x\n",En.ch[0],En.ch[1],En.ch[2],En.ch[3]);

        return 0;
}
```

Output on a big-endian machine

```
INPUT is 0x12345678
short is 0x1234, 0x5678
 char is 0x12, 0x34, 0x56, 0x78
```

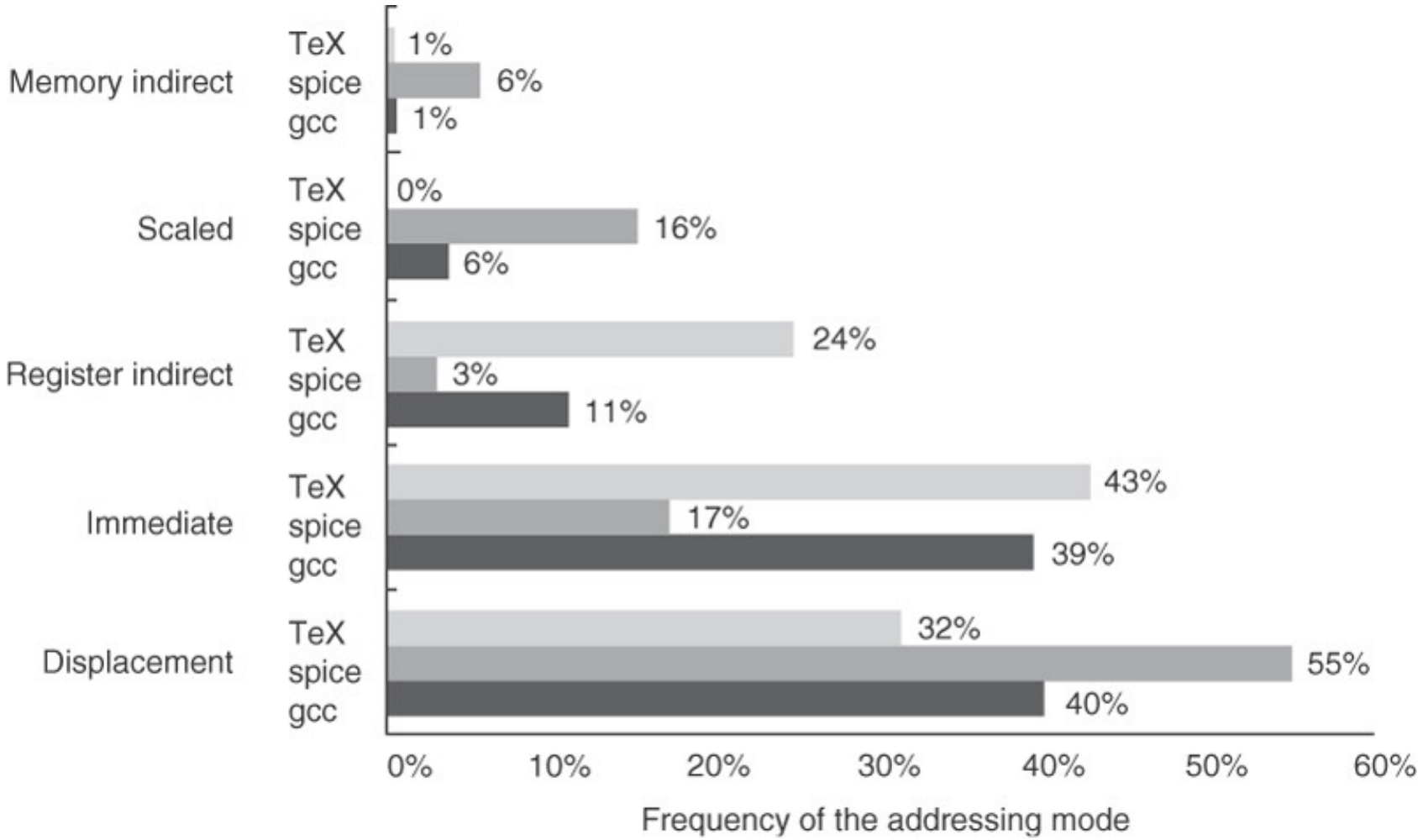
Output on a little-endian machine

```
INPUT is 0x12345678
short is 0x5678, 0x1234
 char is 0x78, 0x56, 0x34, 0x12
```

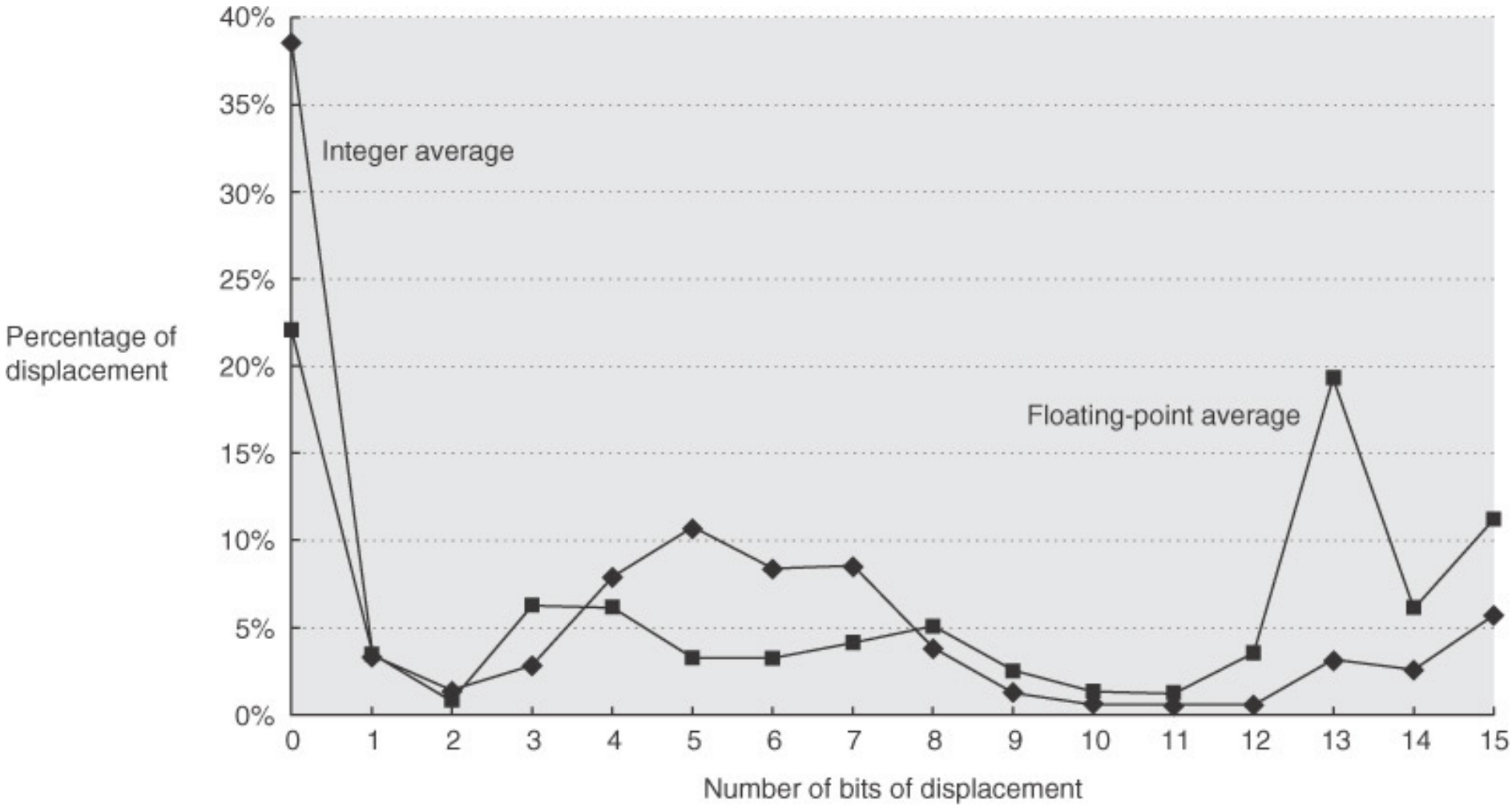
Addressing Mode: how operand data are obtained

Addressing mode	Example	Meaning	When used
Register	Add R4, R3	Regs[R4]←Regs[R4]+Regs[R3]	When a value is in a register
Immediate	Add R4, #3	Regs[R4]←Regs[R4]+3	For constants
Displacement	Add R4, 100(R1)	Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]	Accessing local variables
Register indirect	Add R4, (R1)	Regs[R4]←Regs[R4] +Mem[Regs[R1]]	Accessing using a pointer or a computed address
Indexed	Add R3, (R1+R2)	Regs[R3]←Regs[R3] +Mem[Regs[R1]+Regs[R2]]	Sometimes useful in array adrressing: R1=base of array; R2=index amount
Direct or absolute	Add R1, (1001)	Regs[R1]←Regs[R1] +Mem[1001]	Sometimes useful for accessing static data: address constant may need to be large.
Memory indirect	Add R1, @(R3)	Regs[R1]←Regs[R1]+Mem[Mem[Regs[R3]]]	If R3 is the address of a pointer, the mode yeilds *p
Autoincrement	Add R1, (R2)+	Regs[R1] ←Regs[R1] +Mem[Regs[R2]] Regs[R2]←Regs[R2]+d	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d
Autodecrement	Add R1, -(R2)	Regs[R2]←Regs[R2]-d Regs[R1]←Regs[R1] +Mem[Regs[R2]]	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack
Scaled	Add R1,100(R2)[R3]	Regs[R1] ←Regs[R1] +Mem[100+Regs[R2] +Regs[R3]xd]	Used to index arrays. May be applied to any indexed addressing mode in some computer

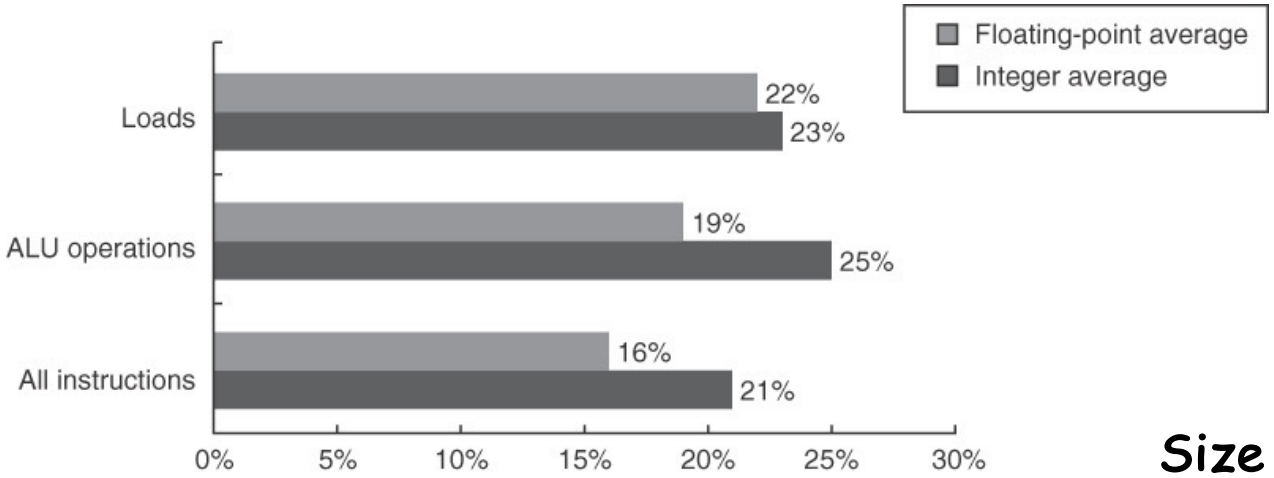
Summary of Use of Memory Addressing Modes



Displacement Addressing Mode

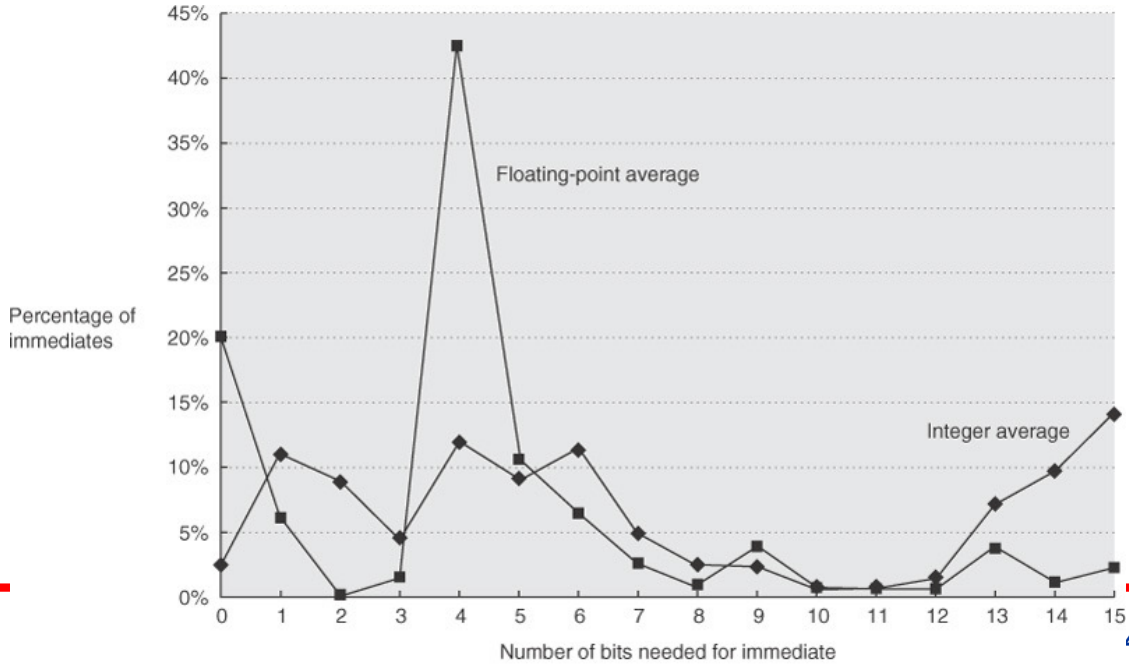


Immediate or Literal Addressing Mode



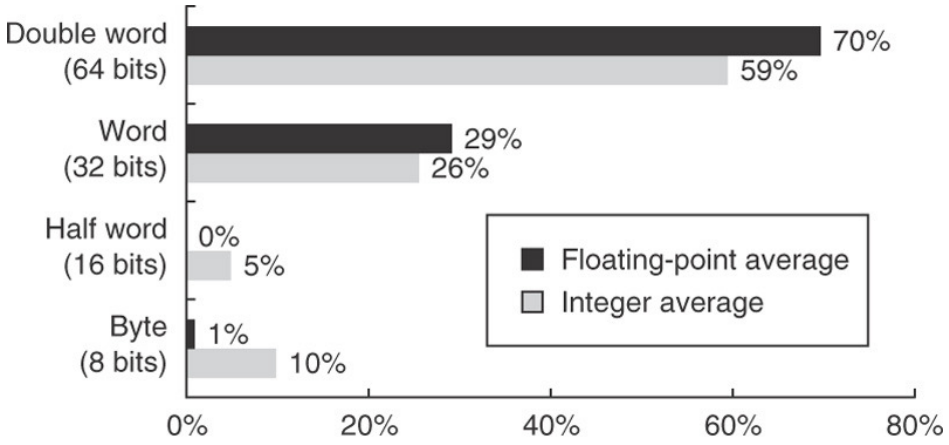
Frequency of Immediate

Size of Immediate



Type and Size of Operands

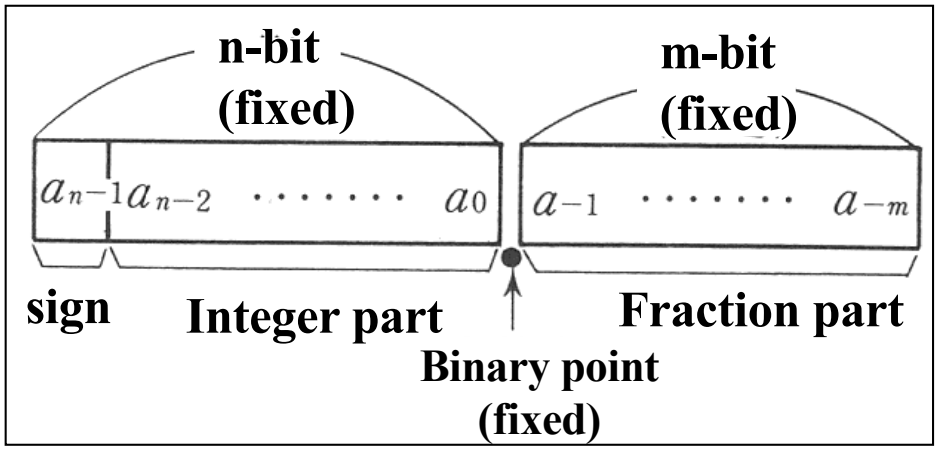
- Common operand types include:
  - Character (8 bits for ASCII, 16bits for Unicode)
  - Half word (16 bits, short integer)
  - Word (32 bits)
    - Integer
    - Single-precision floating point (IEEE standard 754)
  - Double-word (64 bit)
    - Double integer
    - Double-precision floating point (IEEE standard 754)
  - Decimal format for business applications



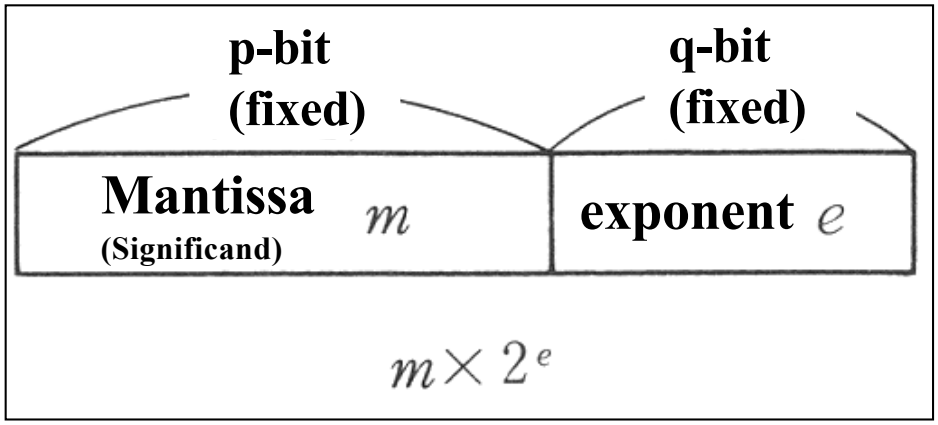
Distribution of data accesses by size for the benchmark programs

Floating-Point Numbers

- Fixed-point notation
  - Binary point is fixed, e.g., rightmost for integers
  - ☺ Easy for arithmetic operations
  - ☹ Representable range limited

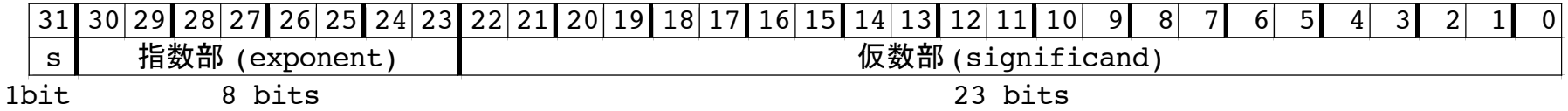


- Floating-point notation
  - Numbers are presented by a mantissa (significand) and an exponent, similar to scientific notation
  - ☺ Representable range extended
  - ☹ Complicated processing needed for arithmetic operations

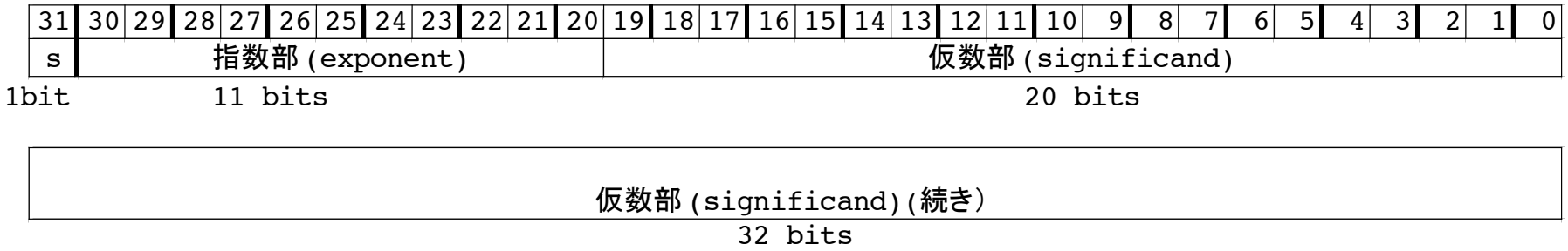


IEEE745: Standard Format for Floating-Point Data Representation

Single-Precision Format $(-1)^S \times (1+\text{significand}) \times 2^{(\text{exponent} - 127)}$



Double-Precision Format $(-1)^S \times (1+\text{significand}) \times 2^{(\text{exponent} - 1023)}$



BCD Representation

- Binary-Coded Decimal (BCD) Representation
  - Decimal numbers are presented digit by digit in the binary form
    - ⇨ 4 bits are needed to represent decimal digit from 0 to 9
    - ☺ No error occur in representation of fraction numbers
    - ✓ 1010 to1111 are not used
      - ⇨ Redundant coding

Example 589. 138

⇩

0101 1000 1001. 0001 0011 1000

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Additions on BCD Numbers

- Add 4-bit (BCD digit) by 4-bit from the lower
- As there is unused range from 1010 to 1111, if a result of addition of each digit is greater than 9, add offset 110(6) for compensation and carry generation

```
  0011 0101  (35)
+ 0010 0100  (24)
-------------------
  0101 1001  (59)
```

```
  1000 0111  (87)
+ 0011 1000  (38)
-------------------
  1011 1111  >1010
     110     <= Add offset 110
-------------------
carry-> 1 0101
     110     <= Add offset 110
-------------------
  1 0010       (125)
```

Subtraction on BCD Numbers

- Subtract 4-bit (BCD digit) by 4-bit from the lower
- If subtraction of each digit needs a borrow from the upper digit and a subtraction result is greater than 9, subtract offset 110(6) for compensation

```
   0011 0101  (35)
 - 0010 0100  (24)
-------------------
   0001 0001  (11)
```

```
   1000 0111  (87)
 - 0011 1000  (38)
-------------------
   0101 1111      >1010
borrow
        1 0110  <=  Sub offset
-------------------
   0100 1001  (49)
```

Operations in the Instruction Set

Categories of instruction operators

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Load-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, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

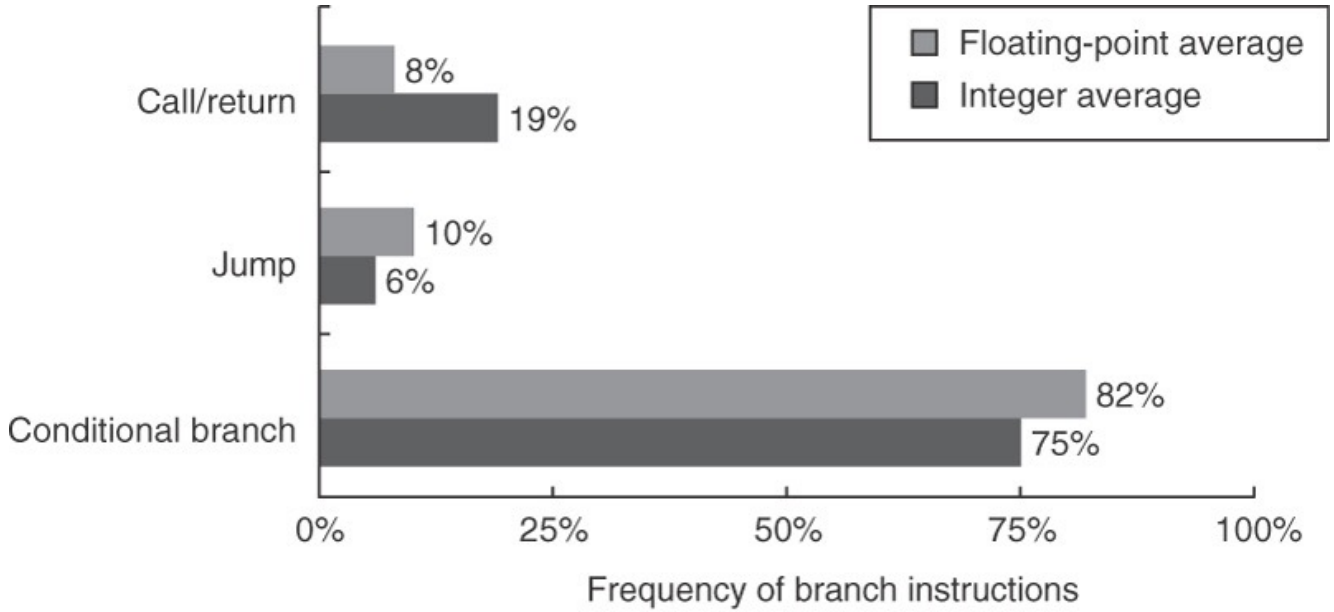
Top 10 instructions for X86

Rank	X86 Instruction	Integer average* (% total executed)
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%

* 5 SPECint92 programs

Instructions for Control Flow

- Four types of control flow change
  - Conditional branches
  - Jumps
  - Procedure calls
  - Procedure returns



Addressing Modes for Control Flow Instructions

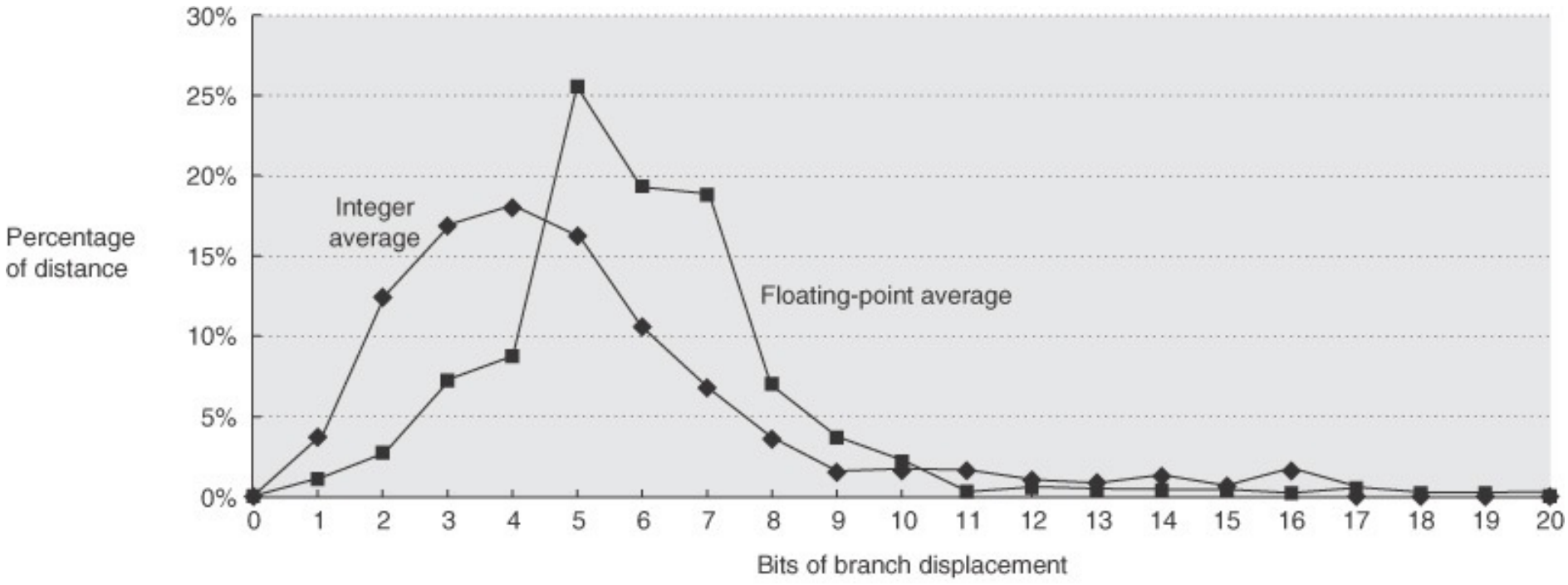
- The most common way to specify the destination is to supply a displacement that is added to the program counter (PC)
  - PC-relative
- Advantages:
  - The target is often near the current instruction
  - Specifying the position relative to the current PC requires fewer bits.
  - Allow the code to run independently of where it is loaded
    - Position independence
    - Eliminate some work when the program is linked and is also useful in programs linked dynamically during execution

Register Indirect Jump

- Registers are used to implement returns and indirect jumps when the target is not known at compile time
  - Case or switch statements
  - Virtual functions or methods in object-oriented languages like C++ or Java
    - which allow different routines to be called depending on the type of the argument
  - Higher-order functions or function pointers in languages like C or C++
    - which allow functions to be passed as arguments
  - Dynamic shared libraries
    - Which allow a library to be loaded and linked at runtime only when it is actually invoked by the program rather than loaded and linked statically before the program is run.

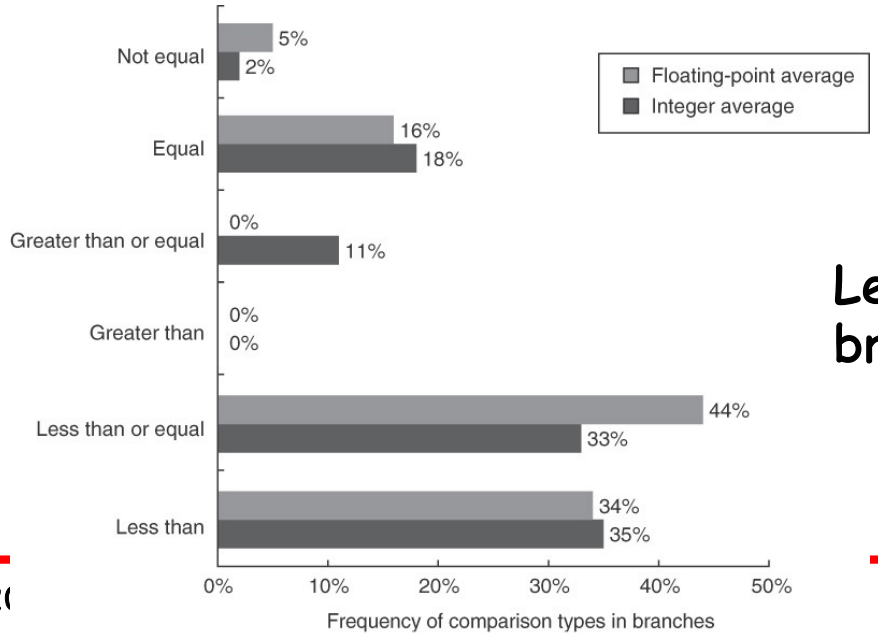
Distribution of Displacement for PC-Relative Branches

- 4-8 bits are enough to cover the most frequent PC-relative branches.



Conditional Branch Options

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition Code (CC)	X86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control	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	Tests arbitrary register with the result of a comparison	Simple	Uses up a register
Compare and branch	PA-RISC, VAX	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch	May be too much work per instruction for pipelined execution



Less than (or equal) branches are dominant!

Encoding an Instruction Set

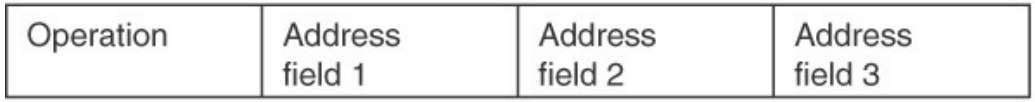
- The architect must balance several competing forces
  - The desire to have as many registers and addressing modes as possible
    - The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size
  - A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation
    - Multiples of bytes, rather than a arbitrary bit length
    - Fixed-length instruction to gain implementation benefits while sacrificing average code size.

Three Basic Variations in Instruction Encoding

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

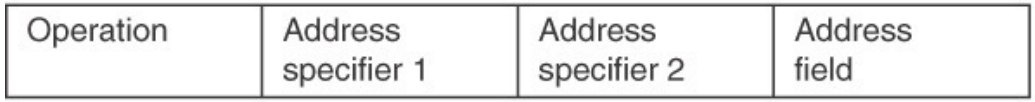
The length of X86 instructions varies between 1 and 17 bytes

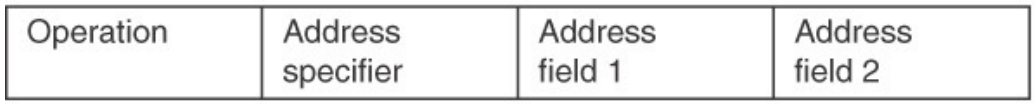
More focusing on code size:
Try to use as few bits as possible to represent the program.



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

More focusing on performance:
Try to make decoding as easy as possible to gain pipelined-implementation benefits.



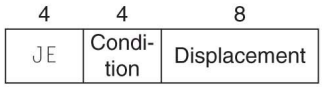


(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

Example: X86 Architecture

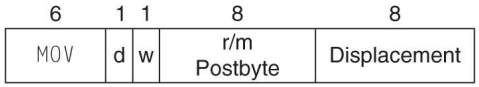
- Typical X86 Instruction Formats

a. JE EIP + displacement

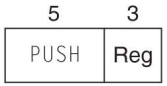


b. CALL

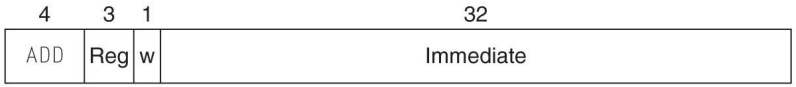
c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42

Example of Machine Instructions on X86

```
00401000 <_WinMainCRTStartup>:
 401000:    55                  push   %ebp
 401001:    89 e5               mov    %esp,%ebp
 401003:    83 ec 18            sub    $0x18,%esp
 401006:    83 e4 f0            and    $0xfffffff0,%esp
 401009:    a1 00 30 40 00      mov    0x403000,%eax
 40100e:    85 c0               test   %eax,%eax
 401010:    74 01               je     401013
<_WinMainCRTStartup+0x13>
 401012:    cc                  int3
 401013:    d9 7d fe            fnstcw -0x2(%ebp)
 401016:    0f b7 45 fe         movzwl -0x2(%ebp),%eax
 40101a:    66 25 c0 f0         and    $0xf0c0,%ax
 40101e:    66 89 45 fe         mov    %ax,-0x2(%ebp)
 401022:    0f b7 45 fe         movzwl -0x2(%ebp),%eax
 401026:    66 0d 3f 03         or     $0x33f,%ax
 40102a:    66 89 45 fe         mov    %ax,-0x2(%ebp)
 40102e:    d9 6d fe            fldcw  -0x2(%ebp)
 401031:    c7 04 24 40 10 40 00    movl   $0x401040,(%esp)
 401038:    e8 33 01 00 00      call   401170 <_cygwin_crt0>
 40103d:    c9                  leave
 40103e:    c3                  ret
 40103f:    90                  nop
```

Example: MIPS Architecture

- MIPS emphasizes
  - a simple load-store instruction set
  - design for pipelining efficiency, including a fixed instruction set encoding
  - efficiency as a compiler target

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

MIPS Instruction Summary

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

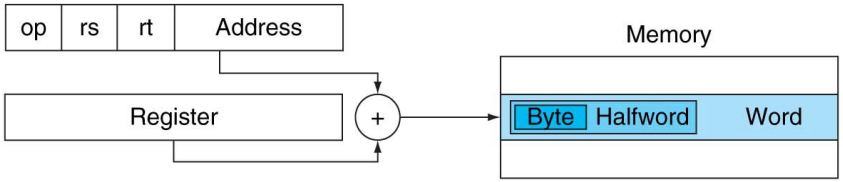
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Addressing Mode

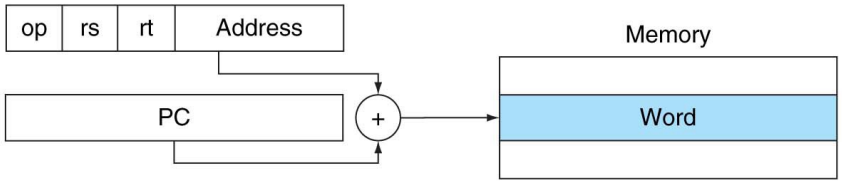
1. Immediate addressing

2. Register addressing

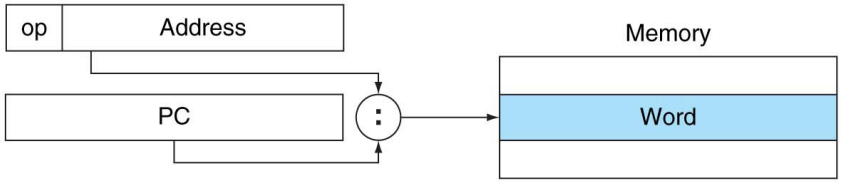
3. Base addressing



4. PC-relative addressing



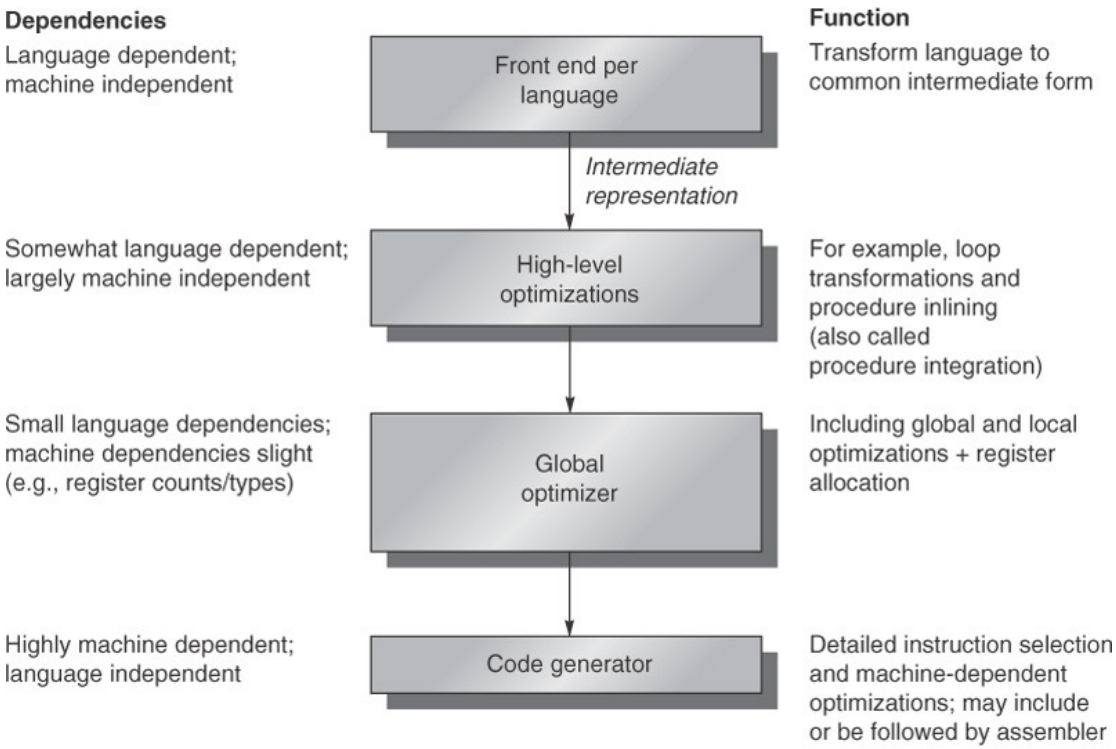
5. Pseudodirect addressing



Reduced Code Size in RISCs

- As now RISCs are used in embedded applications, smaller codes are important.
  1. A hybrid encoding approach use multiple formats specified by the opcode.
     - The narrow instructions support fewer operations, smaller address and immediate fields, fewer registers, and two-address format rather than the classic three-address format of RISC computer
     - Effective use of small limited memory, in particular on-chip memory, in MIPS16 and Thumb
  2. A compressed instructions approach compresses its standard instruction set and then adds hardware to decompress instructions as they are fetched from memory.
     - Compressed code is kept in main memory, ROMs and the disk
     - PowerPC uses run-length encoding compression

Compiler Structure



- How the architect can help the compiler writer
  - ✓ Provide regularity
  - ✓ Provide primitives, not solutions
  - ✓ Simplify trade-offs among alternatives
  - ✓ Provide instructions that bind the quantities known at compile time as constants

Fallacies and Pitfalls

Pitfall	Designing a “high-level” instruction set feature specifically oriented to supporting a high-level language structure - making semantic gap small leads to semantic clash by giving too much semantic content to the instruction, the computer designer make it possible to use instruction only in limited context. CISC (Complex-Instruction Set Architecture) to RISC (Reduced Instruction Set Architecture)
Fallacy	There is such a thing as a typical program Many variation in operations and data size of different applications
Pitfall	Innovating at the instruction set architecture to reduce code size without accounting for the compiler Compilers play a key roll in extracting the potentials of computers
Fallacy	An architecture with flaws cannot be successful X86 is lasting for more than 30 years…
Fallacy	You can design a flawless architecture Stack→Memory/Memory→Memory/Register→Register/Register

Conclusions

- Trend in computer architectures changed from CISC to RISC.
  - In the 1960s, stack architectures were viewed as being a good match for high-level languages
  - In the 1970s, the main concern of architects was how to reduce software costs.
    - High-level language computer architecture and powerful architectures like VAX, which has a large number of addressing modes, multiple data types, and a highly orthogonal architecture
  - In the 1980s, more sophisticated compiler technology and a renewed emphasis on processor performance saw a return to simpler architectures
  - In the 1990s, the following instruction set architectures changes occurred.
    - Address size double from 32 bits to 64 bits
    - Optimization of conditional branches via conditional execution
    - Optimization of cache performance via prefetch
    - Support for multimedia
    - Faster floating-point operations