Chapter 2

Instructions: Language of the Computer

Introduction

- Computer language
  - Words: instructions
  - Vocabulary: instruction set
  - Similar for all, like regional dialect?
- Design goal of computer language
  - To find an instruction set that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

Instruction Set Architecture, ISA

- A specification of a standardized programmer-visible interface to hardware, comprised of:
  - A set of instructions
    - instruction types
    - with associated argument fields, assembly syntax, and machine encoding.
  - A set of named storage locations
    - registers
    - memory
  - A set of addressing modes (ways to name locations)
  - Often an I/O interface
    - memory-mapped

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

$\downarrow$ **compiler**

Assembly language code: architecture specific statements

$\downarrow$ **assembler**

Machine language code: architecture specific bit patterns

software

Instruction Set Architecture

hardware

ISA Design Issue

- Where are operands stored?
- How many explicit operands are there?
- How is the operand location specified?
- What type & size of operands are supported?
- What operations are supported?

Before answering these questions, let's consider more about

- Memory addressing
- Data operand
- Operations

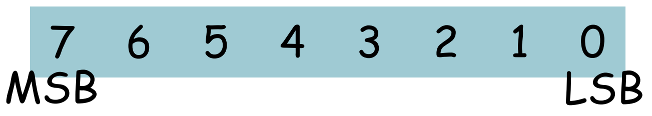
Memory Addressing

- Most CPU are byte-addressable and provide access for
  - Byte (8-bit)
  - Half word (16-bit)
  - Word (32-bit)
  - Double words (64-bit)

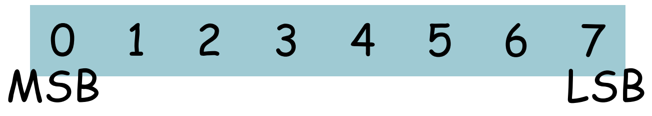
- How memory addresses are interpreted and how they are specified?
  - Little Endian or Big Endian
    - for ordering the bytes within a larger object within memory
  - Alignment or misaligned memory access
    - for accessing to an abject larger than a byte from memory
  - Addressing modes
    - for specifying constants, registers, and locations in memory

Byte-Order ("Endianness")

- **Little Endian**
  - The byte order put the byte whose address is "xx…x000" at the least-significant position in the double word
  - E.g. Intel, DEC, …
  - The bytes are numbered as



- **Big Endian**
  - The byte order put the byte whose address is "xx…x000" at the most-significant position in the double word
  - E.g. MIPS, IBM, Motorolla, Sun, HP, …
  - The byte address are numbered as

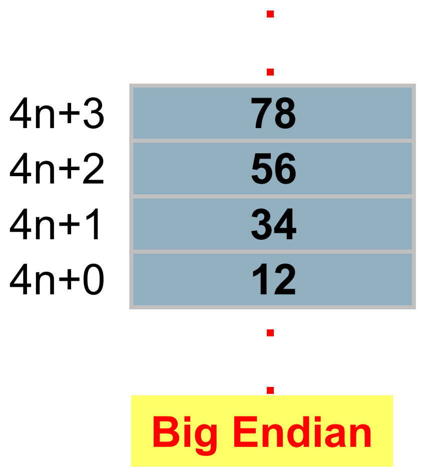


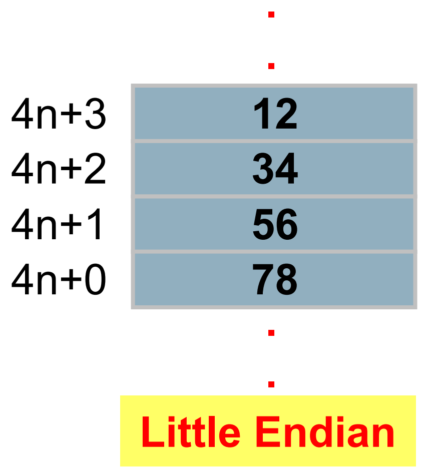
Little or Big Endian ?

- No absolute advantage for one over the other, but

  Byte order is a problem when exchanging data among computers

- Example
  - In C, `int num = 0x12345678; // a 32-bit word`,
  - how is num stored in memory?

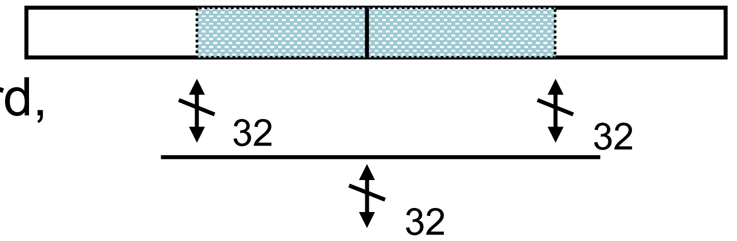




Data Alignment

- The memory is typically aligned on a multiple of a word or double-word boundary.
  - Address resolution is typically one byte
- An access to object of size S bytes at byte address A is called aligned if $A \bmod S = 0$.
- Access to an unaligned operand may require more memory accesses !!
- If computer supports byte, half-word, word accesses in 64-bit register organization, it needs
  - alignment network
    - for the loading of e.g. word to the lower part of a register
  - sign-extended (to be discussed later!!)

Mis-aligned word reference



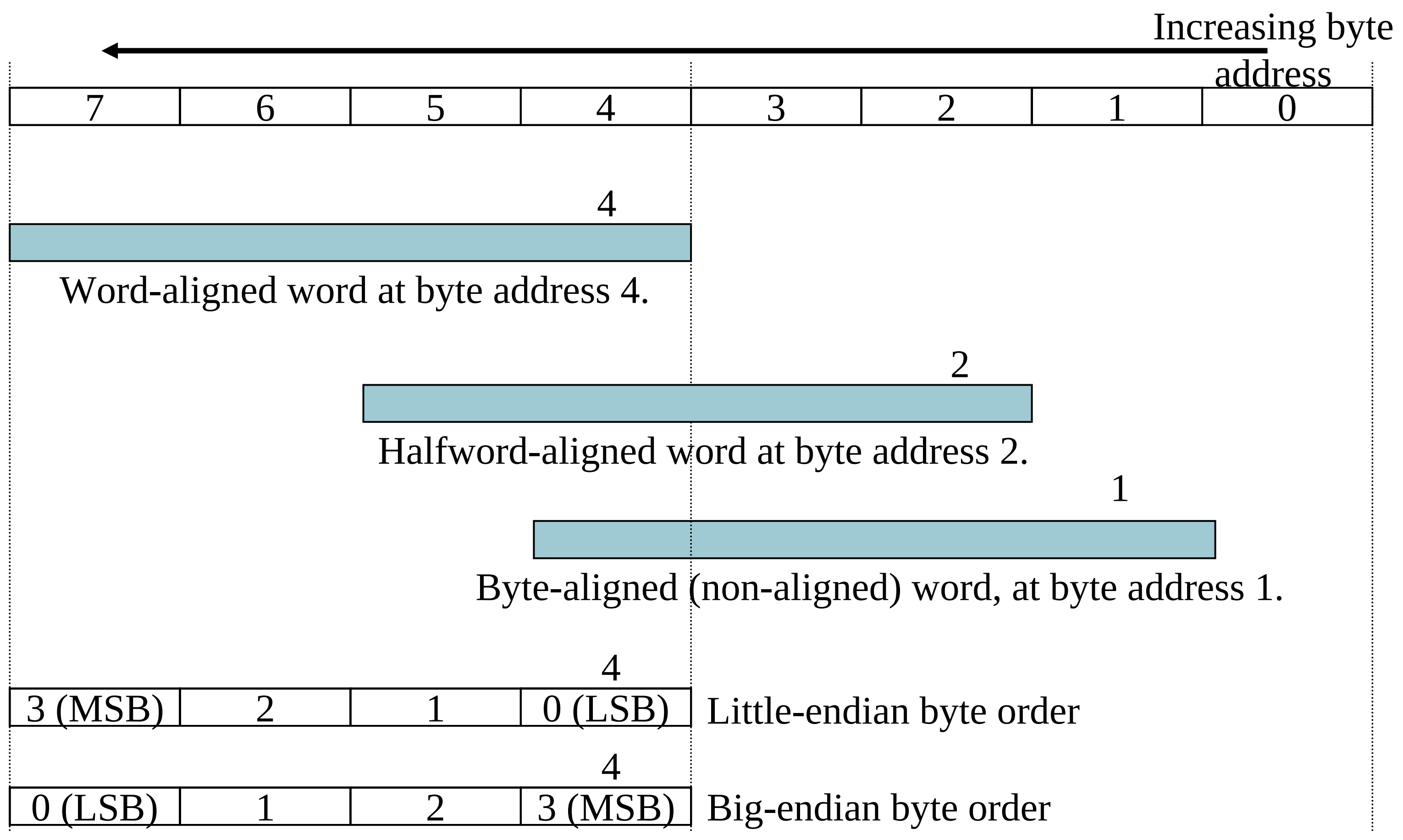
To Processor

Remarks

- Unrestricted Alignment
  - Software is simple
  - Hardware must detect misalignment and make more memory accesses
  - Expensive logic to perform detection
  - Can slow down all references
  - Sometimes required for backwards compatibility
- Restricted Alignment
  - Software must guarantee alignment
  - Hardware detects misalignment access and traps
  - No extra time is spent when data is aligned

- Since we want to ***make the common case fast***, having restricted alignment is often a better choice, unless compatibility is an issue.

Summary: Endians & Alignment


Increasing byte address
7
6
5
4
3
2
1
0
4
Word-aligned word at byte address 4.
2
Halfword-aligned word at byte address 2.
1
Byte-aligned (non-aligned) word, at byte address 1.
4
3 (MSB)
2
1
0 (LSB)
Little-endian byte order
4
0 (LSB)
1
2
3 (MSB)
Big-endian byte order

Addressing Mode ?

- It answers the question:
  - Where can operands/results be located?

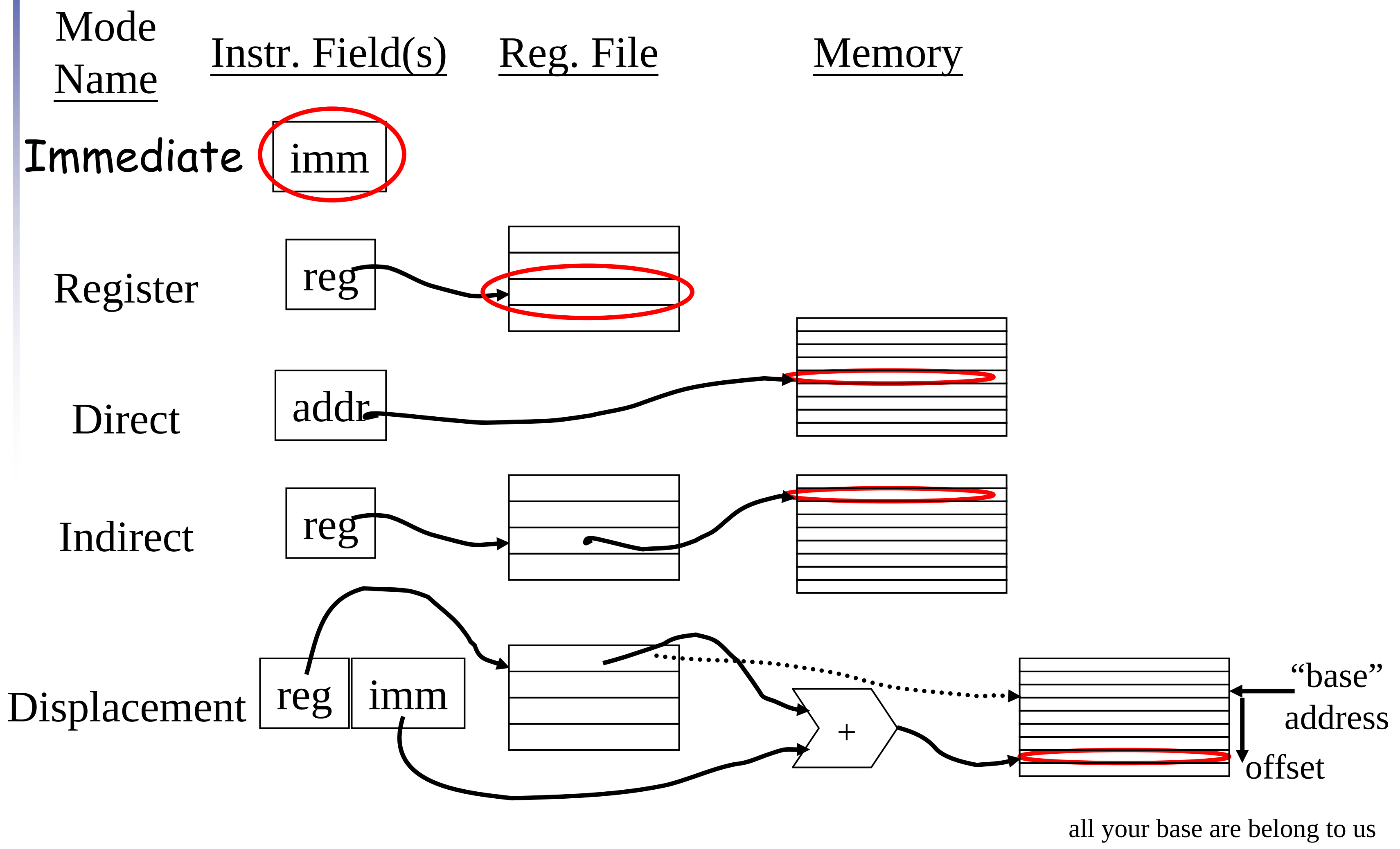
- Recall that we have two types of storage in computer : registers and memory
  - A single operand can come from either a register or a memory location
  - Addressing modes offer various ways of specifying the specific location

Types of Addressing Mode

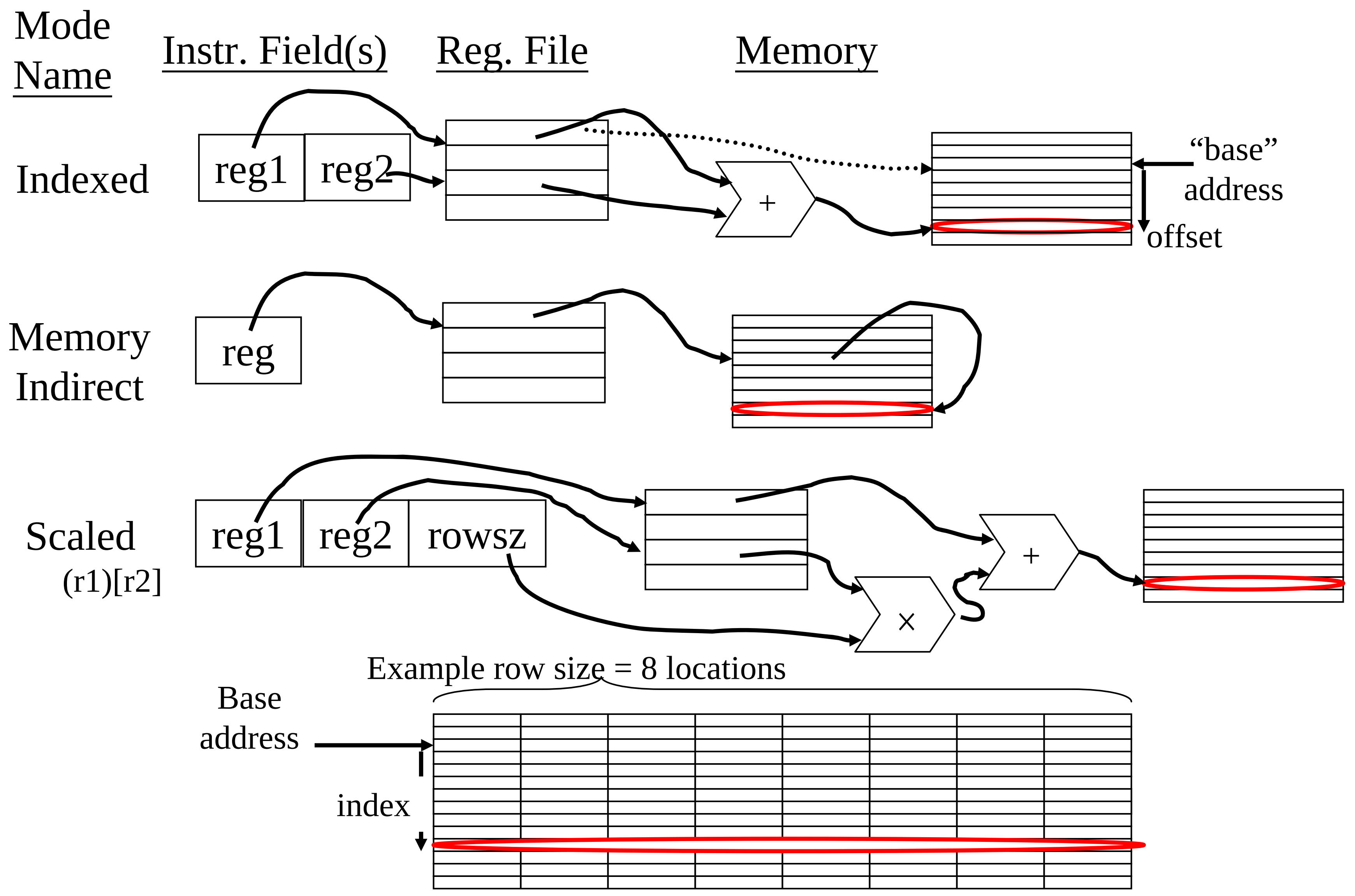
Addressing Mode	Example	Action
1. Register direct	`Add R4, R3`	`R4 <- R4 + R3`
2. Immediate	`Add R4, #3`	`R4 <- R4 + 3`
3. Displacement	`Add R4, 100(R1)`	`R4 <- R4 + M[100 + R1]`
4. Register indirect	`Add R4, (R1)`	`R4 <- R4 + M[R1]`
5. Indexed	`Add R4, (R1 + R2)`	`R4 <- R4 + M[R1 + R2]`
6. Direct	`Add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

R: Register, M: Memory

Addressing Modes Visualization (1)



Addressing Modes Visualization (2)



How Many Addressing Modes?

- Simple addressing modes
  - Register, Direct, Immediate
- Register indirect addressing modes
  - Register Indirect, Displacement, Indexed, Memory Indirect
- Advanced addressing modes
  - Auto-increment, Auto-decrement, Scaled
- A Tradeoff: complexity vs. instruction count
  - Should we add more modes?
    - Depends on the application class
    - Special addressing modes for DSP processors
      - Modulo or circular addressing
      - Bit reverse addressing
      - Stride, gather/scatter addressing

Summary – Memory Addressing

- Need to support at least three addressing modes
  - Displacement, immediate, and register indirect
  - They represent 75% -- 99% of the addressing modes in benchmarks
- The size of the address for displacement mode to be at least 12—16 bits (75% – 99%)
- The size of immediate field to be at least 8 – 16 bits (50%— 80%)
- DSPs rely on hand-coded libraries to exercise novel addressing modes

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  # a = b + c
```

- All arithmetic operations have this form
- Design Principle 1: Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled MIPS code:
  - break a C statement into several assembly instructions
  - introduce temporary variables

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- Design Principle 2: Smaller is faster
  - e.g. main memory: millions of locations

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

operands are all registers !!

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

Memory Operand Example 1

addressing mode

- C code:

  ```
  g = h + A[8];
  ```

  - g in $s1, h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)     # load word
add $s1, $s2, $t0
```

offset

base register

Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  - h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)     # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)     # store word
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

MIPS Registers

- 32 32-bit Registers with R0:=0
  - These registers are general purpose, any one can be used as an operand/result of an operation
  - But making different pieces of software work together is easier if certain conventions are followed concerning which registers are to be used for what purposes.
- Reserved registers: R1, R26, R27
  - R1 for assembler, R26-27 for OS
- Special usage:
  - R28: pointer register
  - R29: stack pointer
  - R30: frame pointer
  - R31: return address

Policy of Use Conventions

Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Register 1 ($at) reserved for assembler, 26-27 for operating system

These conventions are usually suggested by the vendor and supported by the compilers

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant
- Design Principle 3: Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 ~ $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 ~ +4,294,967,295

Binary Representation of Integers

- Number can be represented in any base
- Hexadecimal/Binary/Decimal representations

  $\texttt{ACE7}_{\texttt{hex}} = \texttt{1010 1100 1110 0111}_{\texttt{bin}} = \texttt{44263}_{\texttt{dec}}$

  - most significant bit, MSB, usually the leftmost bit
  - least significant bit, LSB, usually the rightmost bit
- Ideally, we can represent any integer if the bit width is unlimited
- Practically, the bit width is limited and finite…
  - for a 8-bit byte ➔ 0~255 ($0 \sim 2^8 - 1$)
  - for a 16-bit halfword ➔ 0~65,535 ($0 \sim 2^{16} - 1$)
  - for a 32-bit word ➔ 0~4,294,967,295 ($0 \sim 2^{32} - 1$)

Signed Number

- Unsigned number is mandatory
  - Eg. Memory access, PC, SP, RA
- Sometimes, negative integers are required in arithmetic operation
  - a representation that can present both positive and negative integers is demanded

→signed integers

3 well-known methods

- Sign and Magnitude
- 1’s complement
- 2’s complement

Sign and Magnitude

- Use the MSB as the **sign** bit
  - 0 for positive and 1 for negative
- If the bit width is n
  - range ➔ $-(2^{n-1} - 1) \sim 2^{n-1} - 1$; $2^n - 1$ different numbers
  - e.g., for a byte ➔ –127 ~ 127
- Examples
  - 00000110 ➔ +6
  - 10000111 ➔ –7
- Shortcomings
  - 2 0’s; positive 0 and negative 0; 00000000 and 10000000
  - relatively complicated HW design (e.g., adder)

1's Complement

+7 ➔ 0000 0111

–7 ➔ 1111 1000 (bit inverting)

- If the bit width is n
  - range ➔ $-(2^{n-1} - 1) \sim 2^{n-1} - 1$; $2^n - 1$ different numbers
  - e.g., for a byte ➔ –127 ~ 127
- The MSB implicitly serves as the sign bit
  - except for –0
- Shortcomings
  - 2 0's; positive 0 and negative 0; 00000000 and 11111111
  - relatively complicated HW design (e.g., adder)

2’s Complement

+7 ➔ 0000 0111

–7 ➔ 1111 1001 (bit inverting first then add 1)

- The MSB implicitly serves as the sign bit
- 2’s complement of 10000000 ➔ 10000000
  - this number is defined as –128
- If the bit width is n
  - range ➔ **-2^{n-1}** ~ $2^{n-1} - 1$; **2^n** different numbers
  - e.g., for a byte ➔ **–128** ~ 127
- Relatively easy hardware design

- Virtually, all computers use 2’s complement representation nowadays

2's-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: $-2^{n-1} \sim +2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2,147,483,648 + 2,147,483,644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 ~ +2,147,483,647

2's-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \overline{x} = 1111...111_2 = -1$$

$$\overline{x} + 1 = -x$$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$

    $= 1111\ 1111\ \ldots\ 1110_2$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
  - e.g. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

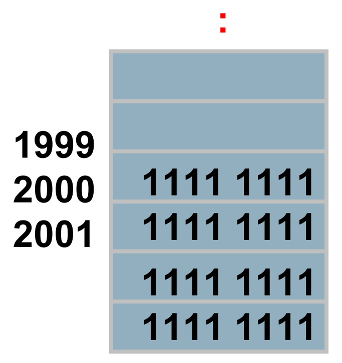
lbu vs lb

- We want to load a BYTE into `$s3` from the address 2000

  After the load, what is the value of `$s3` ?

  - A1: 0000 0000 0000 0000 0000 0000 1111 1111 (255) ?
  - A2: 1111 1111 1111 1111 1111 1111 1111 1111 (–1) ?

- Signed (A2) ➔ `lb $s3, 0($s0)`
- Unsigned (A1) ➔ `lbu $s3, 0($s0)`



Assume
$s0 = 2000

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers (5-bit representation)
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

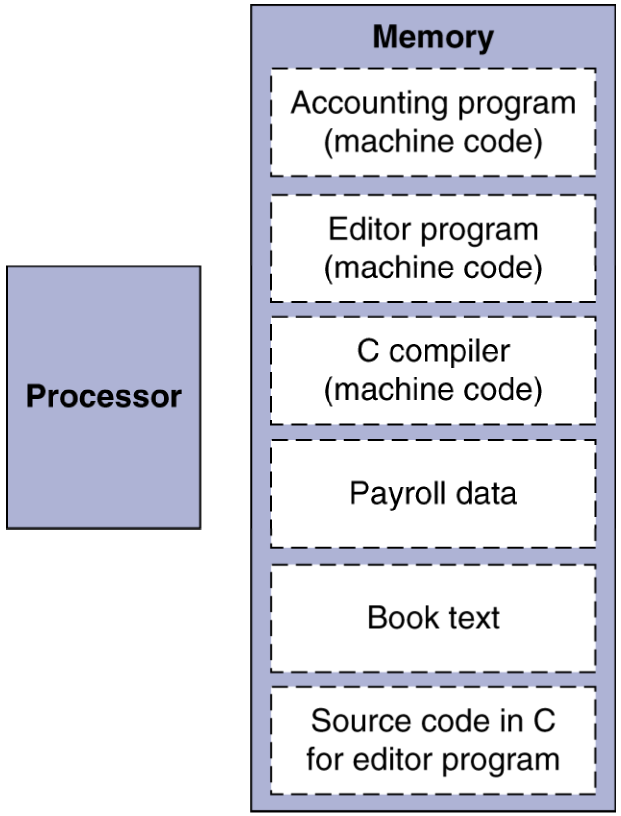
MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: $–2^{15}$ to $+2^{15} – 1$
  - Address: offset added to base address in rs
- Design Principle 4: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Stored Program Computers

The BIG Picture


Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

- Useful for extracting and inserting groups of bits in a word

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and $t0, $t1, $t2
```

```
$t2   0000 0000 0000 0000 0000 1101 1100 0000

$t1   0000 0000 0000 0000 0011 1100 0000 0000

$t0   0000 0000 0000 0000 0000 1100 0000 0000
```

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1101 1100 0000

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Program Flow Control

- Decision making instructions
  - alter the control flow, i.e., change the "next" instruction to be executed
- Branch classifications
  - Unconditional branch
    - Always jump to the desired (specified) address
  - Conditional branch
    - Only jump to the desired (specified) address if the condition is true; otherwise, continue to execute the next instruction
- **Destination addresses** can be specified in the same way as other operands (*combination of register, immediate constant, and memory location*), depending on what addressing modes are supported in the ISA

Conditional Operations in MIPS

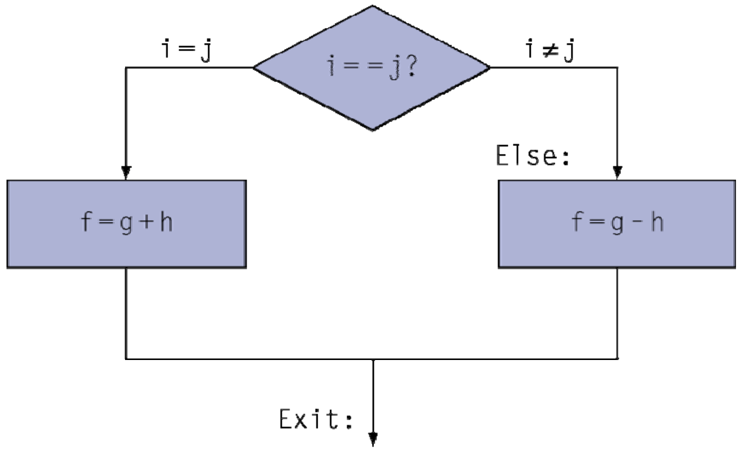
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

```
       bne $s3, $s4, Else
       add $s0, $s1, $s2
       j   Exit
Else:  sub $s0, $s1, $s2
Exit:  …
```

Assembler calculates addresses

Compiling Loop Statements

- C code:

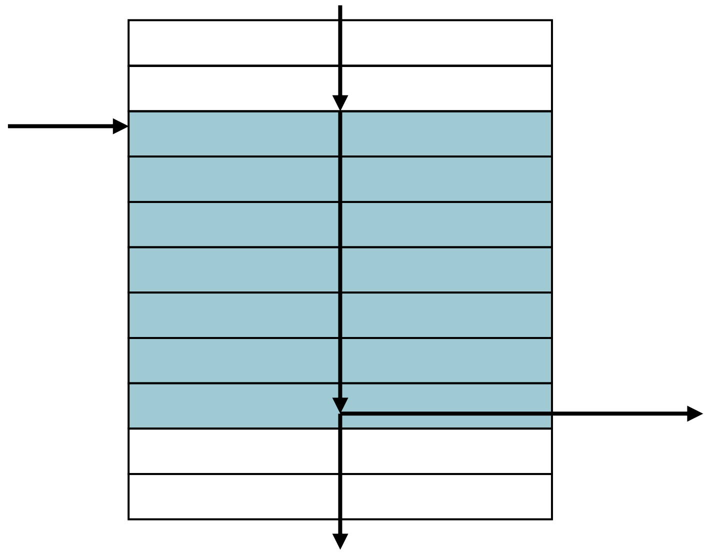
```
while (save[i] == k) i += 1;
```

  - i in $s3, k in $s5, address of save in $s6
- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

- MIPS compiler uses the `slt, beq, bne, $zero` to create =, ≠, <, ≤, >. ≥

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - $-1 < +1 \Rightarrow$ \$t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - $+4{,}294{,}967{,}295 > +1 \Rightarrow$ \$t0 = 0

Branches on LT/LE/GT/GE

- How to implement an equivalent `blt $s0, $s1, L1`?

```
slt $t0, $s0, $s1
bne $t0, $zero, L1    # $zero is always 0
```

- `bge  $s0, $s1, L1`?

```
slt  $t0, $s0, $s1
beq  $t0, $zero, L1
```

- `bgt  $s0, $s1, L1`?

```
slt  $t0, $s1, $s0
bne  $t0, $zero, L1
```

Try ble yourself !!

Branch Instruction Design

- Why not `blt`, `bge`, etc?
- Hardware for $<$, $\geq$, … slower than $=$, $\neq$
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise

Jump Register, jr

A chain of if-then-else,
0≤k<4

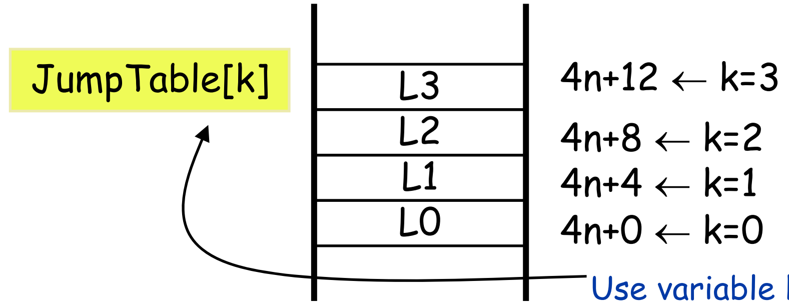
- Case statement in C

```
 switch (k){
    case 0: f=i+j;
    case 1: f=g+h;
    case 2: f=g-h;
    case 3: f=i-j;
 }
```

- Assume `f, g, h,i, j, k` are stored in registers `$s0`, `$s1`,…, and `$s5`, respectively
- Assume `$t2` contains 4
- Assume starting address contained in `$t4`, corresponding to labels L0, L1, L2, and L3, respectively

```
    slt  $t3, $s5, $zero   #test if k<0
    bne  $t3, $zero, Exit  #if k<0,exit
    slt  $t3, $s5, $t2     #test if k<4
    beq  $t3, $zero, Exit  #if k≥4,exit
    add  $t1, $s5, $s5     #2k
    add  $t1, $t1, $t1     #$t1=4k
    add  $t1, $t1, $t4
    lw   $t0, 0($t1)
    jr   $t0
L0:add  $s0, $s3, $s4,
    j    Exit
L1:add  $s0, $s1, $s2
    j    Exit
L2:sub  $s0, $s1, $s2
    j    Exit
L3:sub  $s0, $s3, $s4
Exit:
```

Jump address table in memory



Use variable k to index a jump address tabke

Procedure Calling

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call

Recall: Register Usage

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Procedure Call Instructions

- Procedure call: jump and link

  `jal ProcedureLabel`

  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register

  `jr $ra`

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in \$a0, …, \$a3
  - f in \$s0 (hence, need to save \$s0 on stack)
  - Result in \$v0

Leaf Procedure Example

- MIPS code:

```
leaf_example:
   addi $sp, $sp, -4
   sw   $s0, 0($sp)
   add  $t0, $a0, $a1
   add  $t1, $a2, $a3
   sub  $s0, $t0, $t1
   add  $v0, $s0, $zero
   lw   $s0, 0($sp)
   addi $sp, $sp, 4
   jr   $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

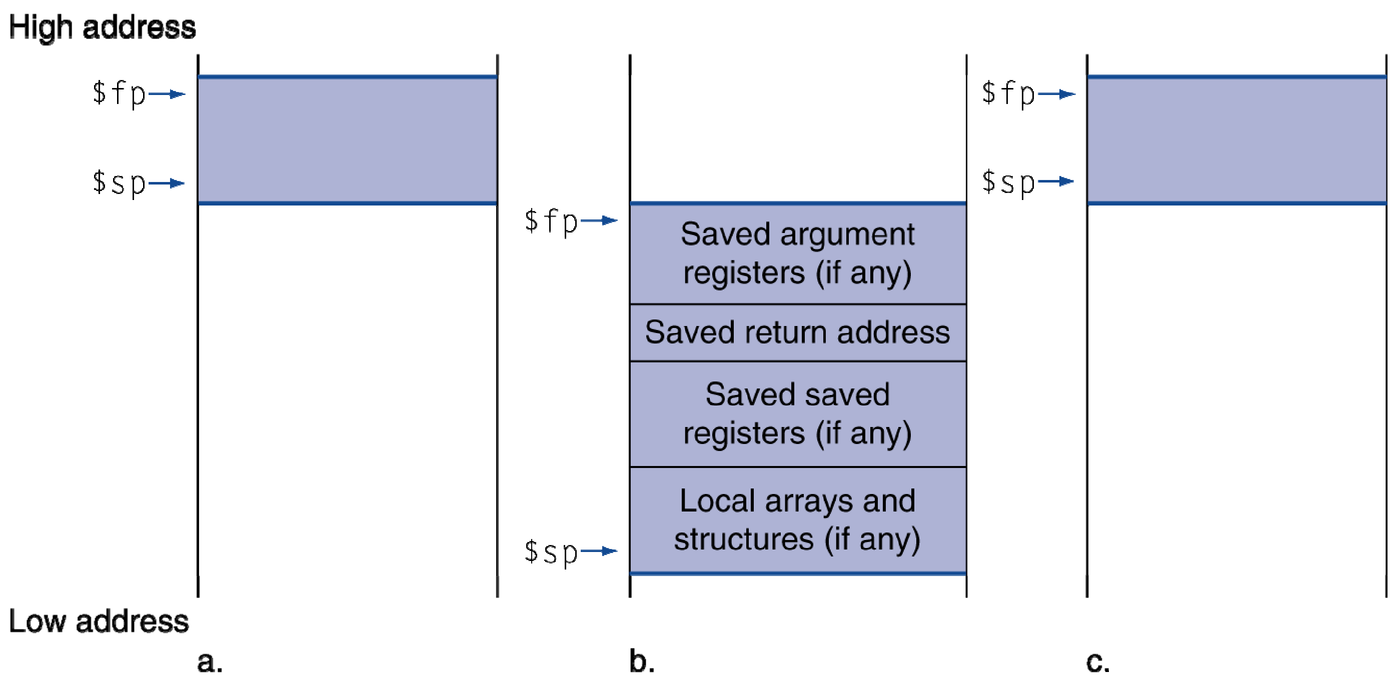
  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

- MIPS code:

```
fact:
     addi $sp, $sp, -8      # adjust stack for 2 items
     sw   $ra, 4($sp)       # save return address
     sw   $a0, 0($sp)       # save argument
     slti $t0, $a0, 1       # test for n < 1
     beq  $t0, $zero, L1
     addi $v0, $zero, 1     # if so, result is 1
     addi $sp, $sp, 8       #   pop 2 items from stack
     jr   $ra               #   and return
L1:  addi $a0, $a0, -1      # else decrement n
     jal  fact              # recursive call
     lw   $a0, 0($sp)       # restore original n
     lw   $ra, 4($sp)       #   and return address
     addi $sp, $sp, 8       # pop 2 items from stack
     mul  $v0, $a0, $v0     # multiply to get result
     jr   $ra               # and return
```

Local Data on the Stack

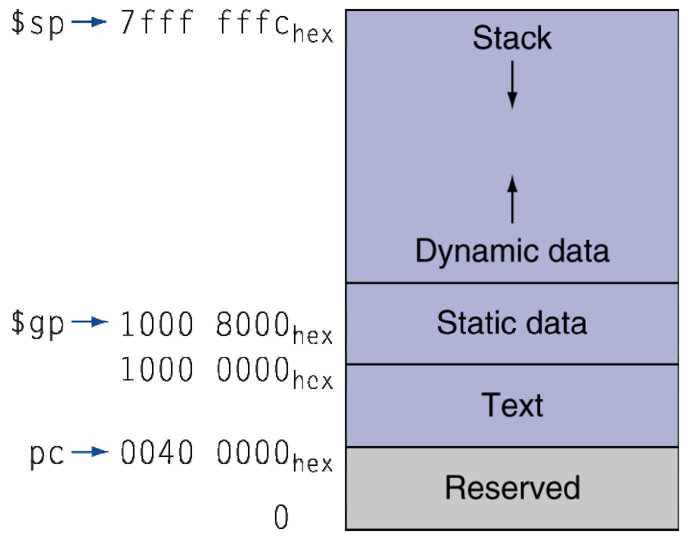


High address
$fp
$sp
Low address
a.
$fp
$sp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
b.
$fp
$sp
c.

- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)      lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)      sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

String Copy Example

- C code (naïve):
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

- MIPS code:

```
strcpy:
     addi $sp, $sp, -4        # adjust stack for 1 item
     sw   $s0, 0($sp)         # save $s0
     add  $s0, $zero, $zero # i = 0
L1:  add  $t1, $s0, $a1       # addr of y[i] in $t1
     lbu  $t2, 0($t1)         # $t2 = y[i]
     add  $t3, $s0, $a0       # addr of x[i] in $t3
     sb   $t2, 0($t3)         # x[i] = y[i]
     beq  $t2, $zero, L2      # exit loop if y[i] == 0
     addi $s0, $s0, 1         # i = i + 1
     j    L1                  # next iteration of loop
L2:  lw   $s0, 0($sp)         # restore saved $s0
     addi $sp, $sp, 4         # pop 1 item from stack
     jr   $ra                 # and return
```

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

```
lui rt, constant
```

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

`lhi $s0, 61`	0000 0000 0111 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- PC-relative addressing
  - Target address = PC + offset $\times$ 4
  - PC already incremented by 4 by this time

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

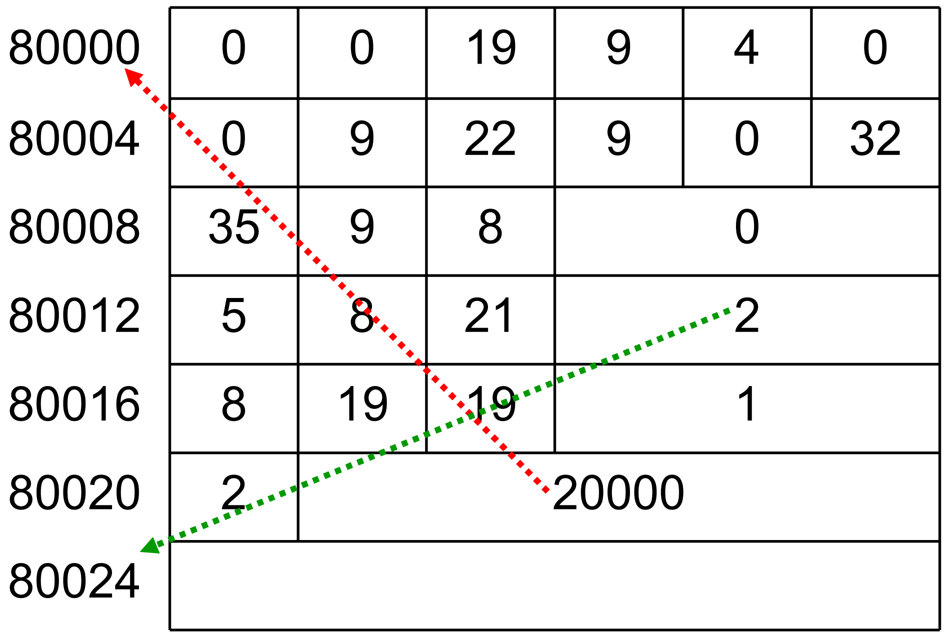
op	address
6 bits	26 bits

- (Pseudo) Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address $\times$ 4)

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```



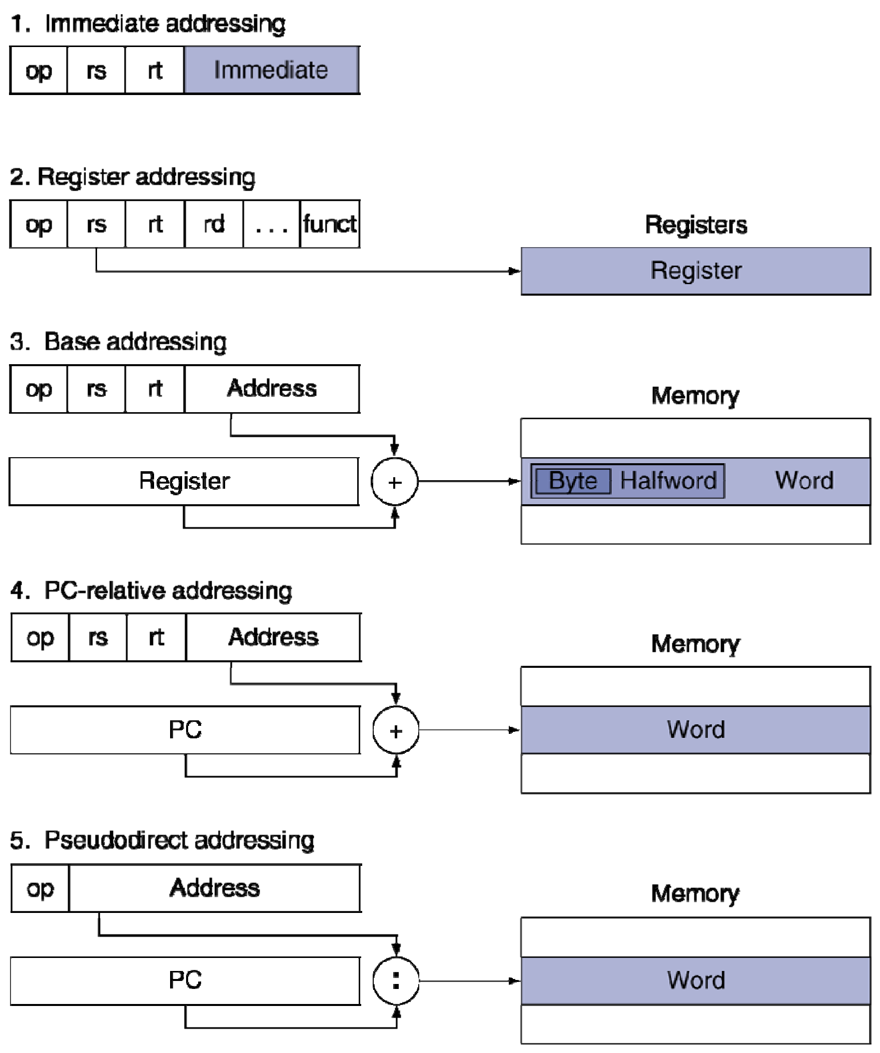
80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
        beq $s0,$s1, L1
               ↓
        bne $s0,$s1, L2
        j L1
    L2: …
```

Addressing Mode Summary



Synchronization

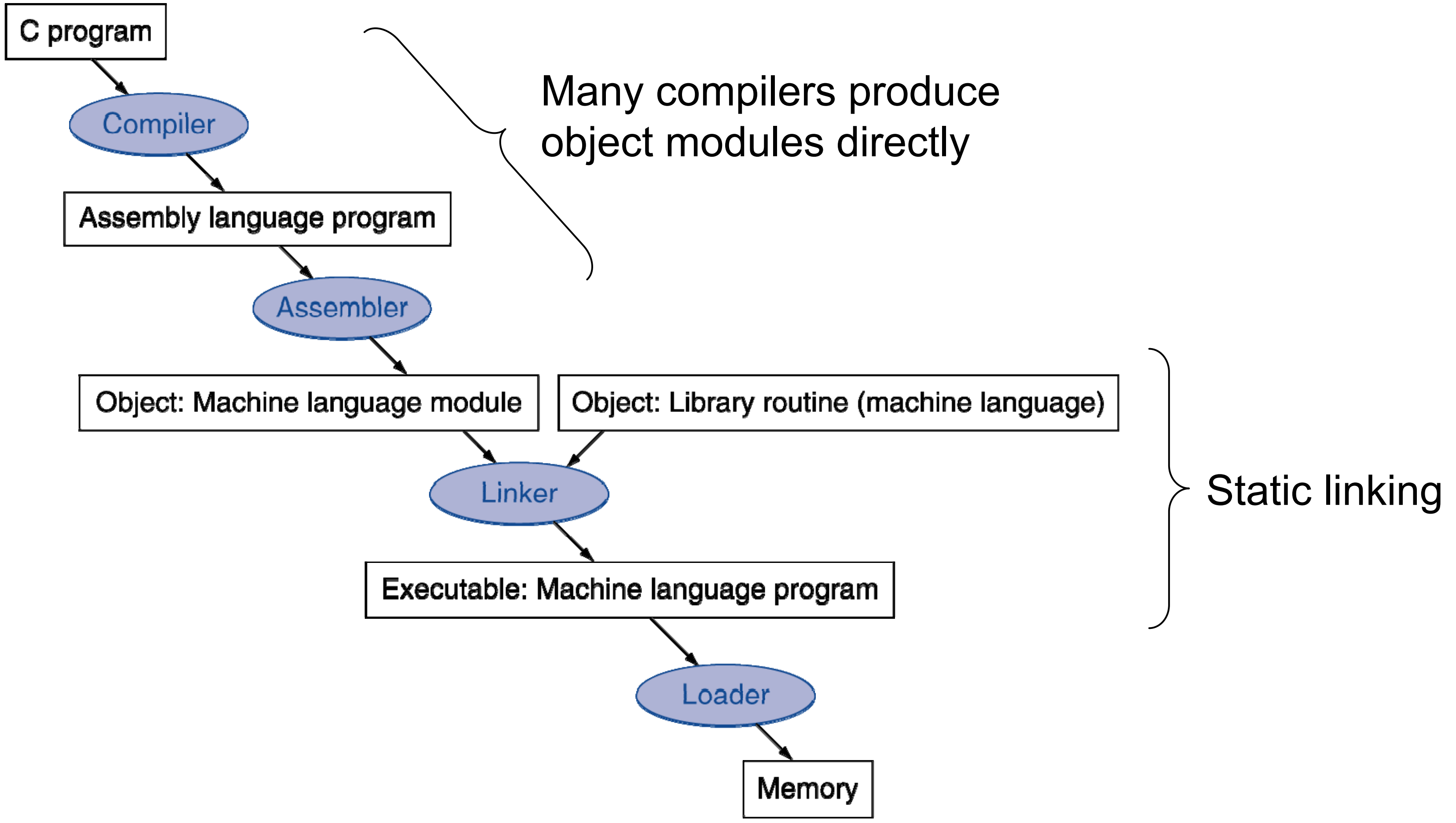
- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

```
move $t0, $t1        →   add $t0, $zero, $t1
blt $t0, $t1, L      →   slt $at, $t0, $t1
                         bne $at, $zero, L
```

  - $at (register 1): assembler temporary

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do exit syscall

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

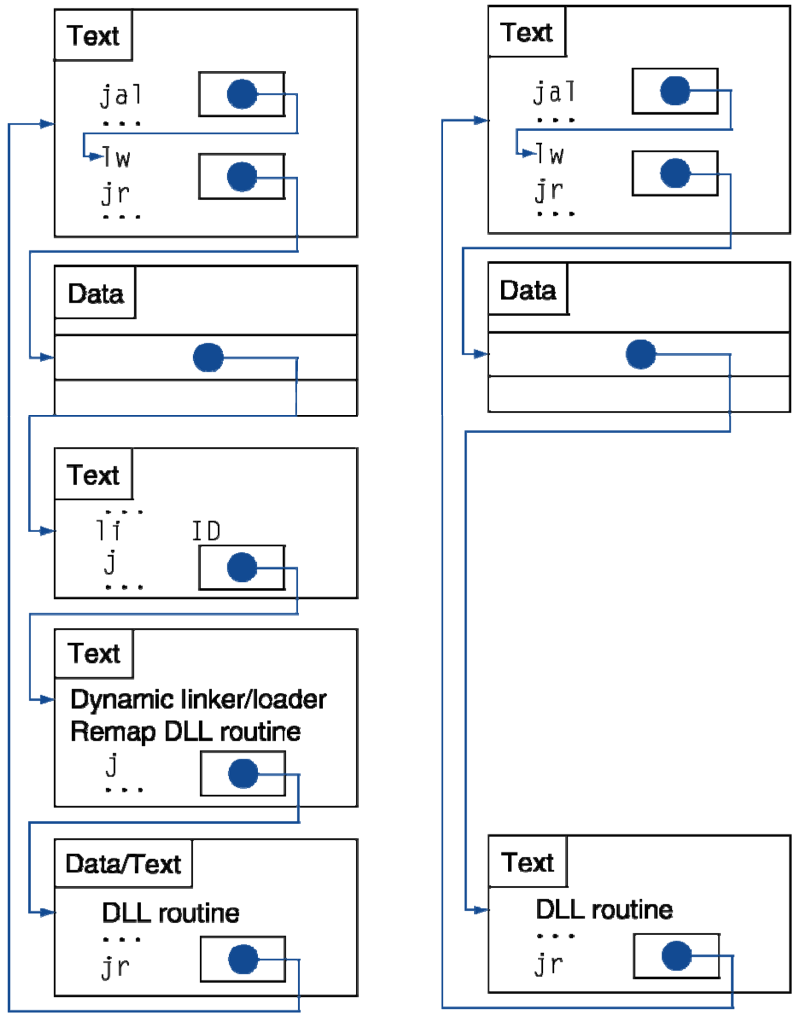
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
swap: sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in $a0, k in $a1, i in $s0, j in $s1

The Procedure Body

```
         move $s2, $a0          # save $a0 into $s2
         move $s3, $a1          # save $a1 into $s3
```
Move params

```
         move $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3     # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
```
Outer loop

```
         beq  $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
         addi $s1, $s0, –1      # j = i – 1
for2tst: slti $t0, $s1, 0       # $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
         sll  $t1, $s1, 2       # $t1 = j * 4
         add  $t2, $s2, $t1     # $t2 = v + (j * 4)
         lw   $t3, 0($t2)       # $t3 = v[j]
         lw   $t4, 4($t2)       # $t4 = v[j + 1]
         slt  $t0, $t4, $t3     # $t0 = 0 if $t4 ≥ $t3
         beq  $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
```
Inner loop

```
         move $a0, $s2          # 1st param of swap is v (old $a0)
         move $a1, $s1          # 2nd param of swap is j
         jal  swap              # call swap procedure
```
Pass params & call

```
         addi $s1, $s1, –1      # j –= 1
         j    for2tst           # jump to test of inner loop
```
Inner loop

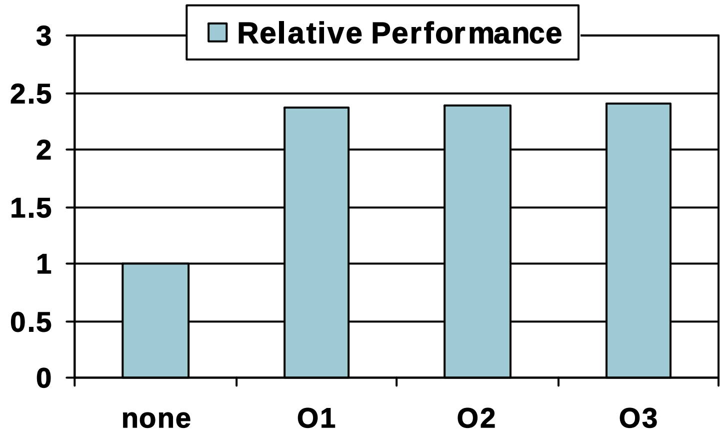
```
exit2:   addi $s0, $s0, 1       # i += 1
         j    for1tst           # jump to test of outer loop
```
Outer loop

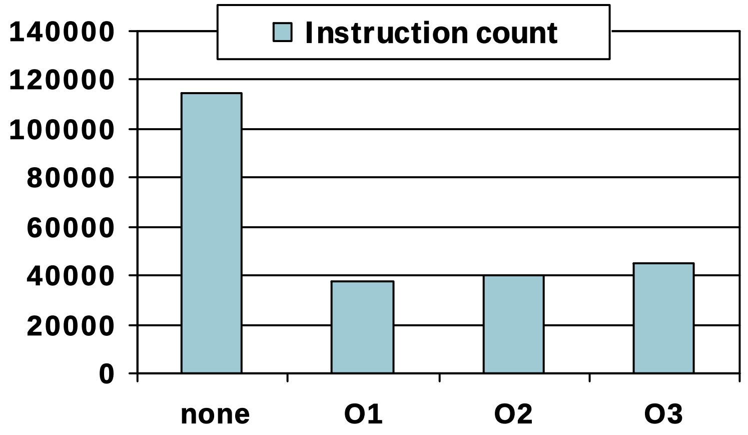
The Full Procedure

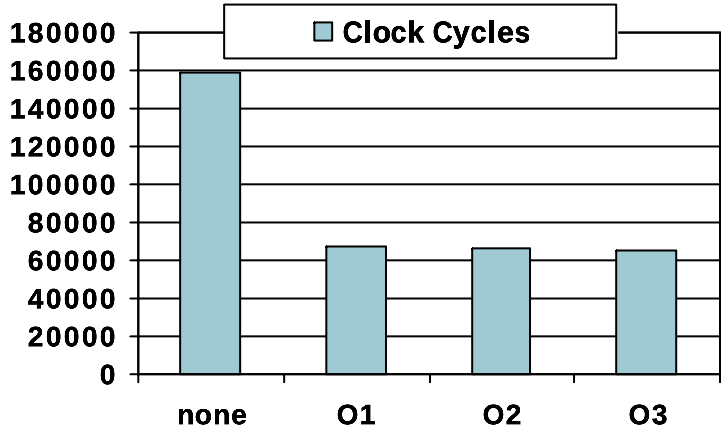
```
sort:    addi $sp,$sp, -20        # make room on stack for 5 registers
         sw $ra, 16($sp)          # save $ra on stack
         sw $s3,12($sp)           # save $s3 on stack
         sw $s2, 8($sp)           # save $s2 on stack
         sw $s1, 4($sp)           # save $s1 on stack
         sw $s0, 0($sp)           # save $s0 on stack
         …                        # procedure body
         …
         exit1: lw $s0, 0($sp)    # restore $s0 from stack
         lw $s1, 4($sp)           # restore $s1 from stack
         lw $s2, 8($sp)           # restore $s2 from stack
         lw $s3,12($sp)           # restore $s3 from stack
         lw $ra,16($sp)           # restore $ra from stack
         addi $sp,$sp, 20         # restore stack pointer
         jr $ra                   # return to calling routine
```

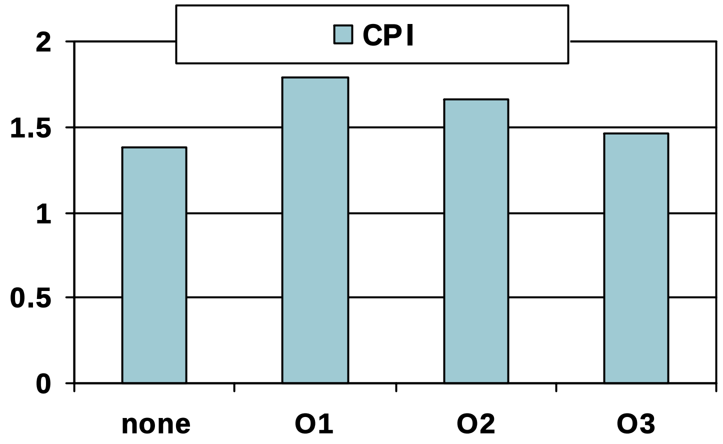
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

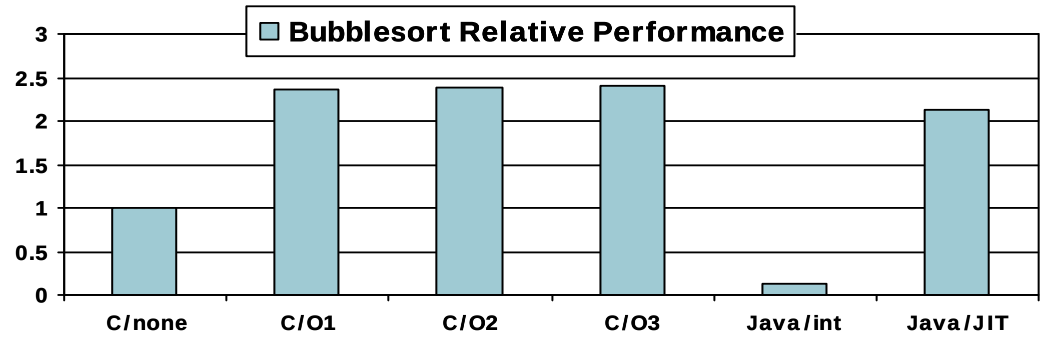


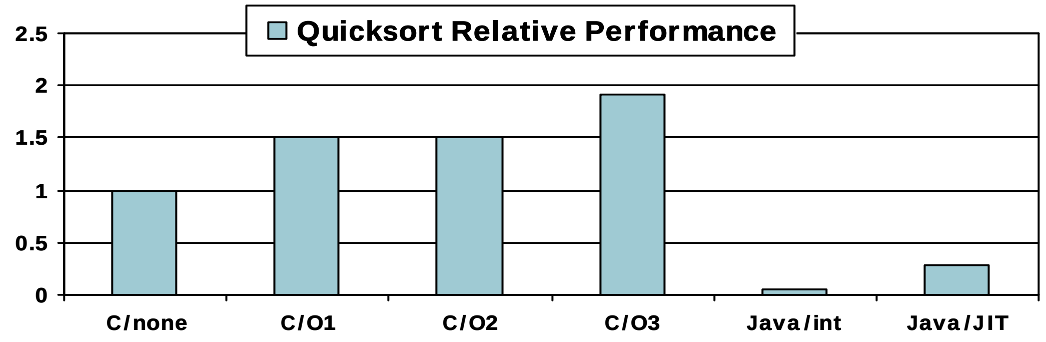


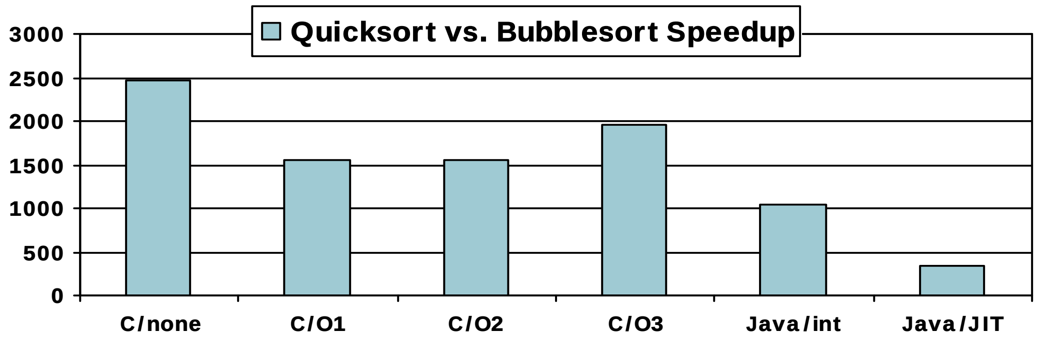




Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing and Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
     *p = 0;
}
```

```
       move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2    # $t1 = i * 4
       add $t2,$a0,$t1  # $t2 =
                        #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
       slt $t3,$t0,$a1  # $t3 =
                        #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

```
       move $t0,$a0     # p = & array[0]
       sll $t1,$a1,2    # $t1 = size * 4
       add $t2,$a0,$t1  # $t2 =
                        #   &array[size]
loop2: sw $zero,0($t0)  # Memory[p] = 0
       addi $t0,$t0,4   # p = p + 4
       slt $t3,$t0,$t2  # $t3 =
                        #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

ARM & MIPS Similarities

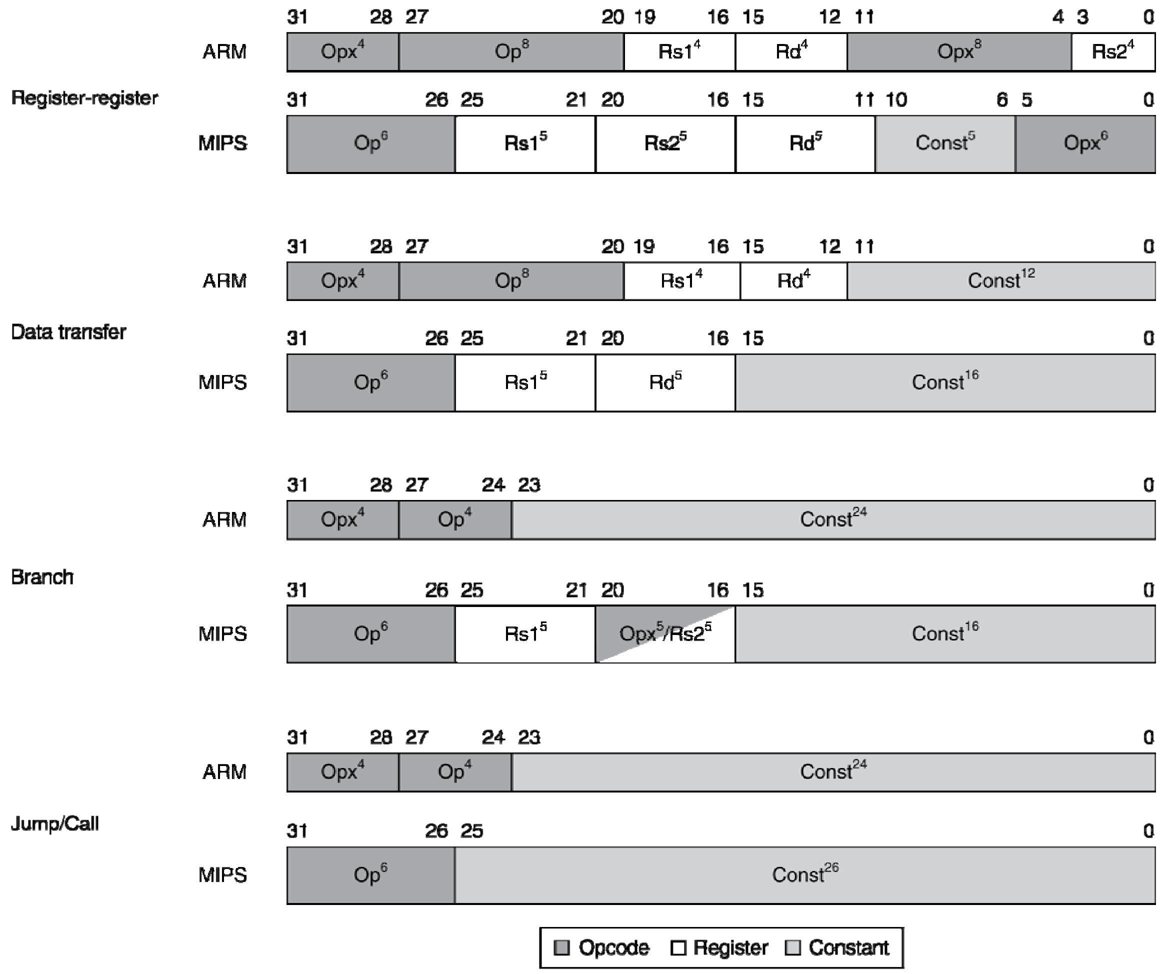
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

Instruction Encoding



The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

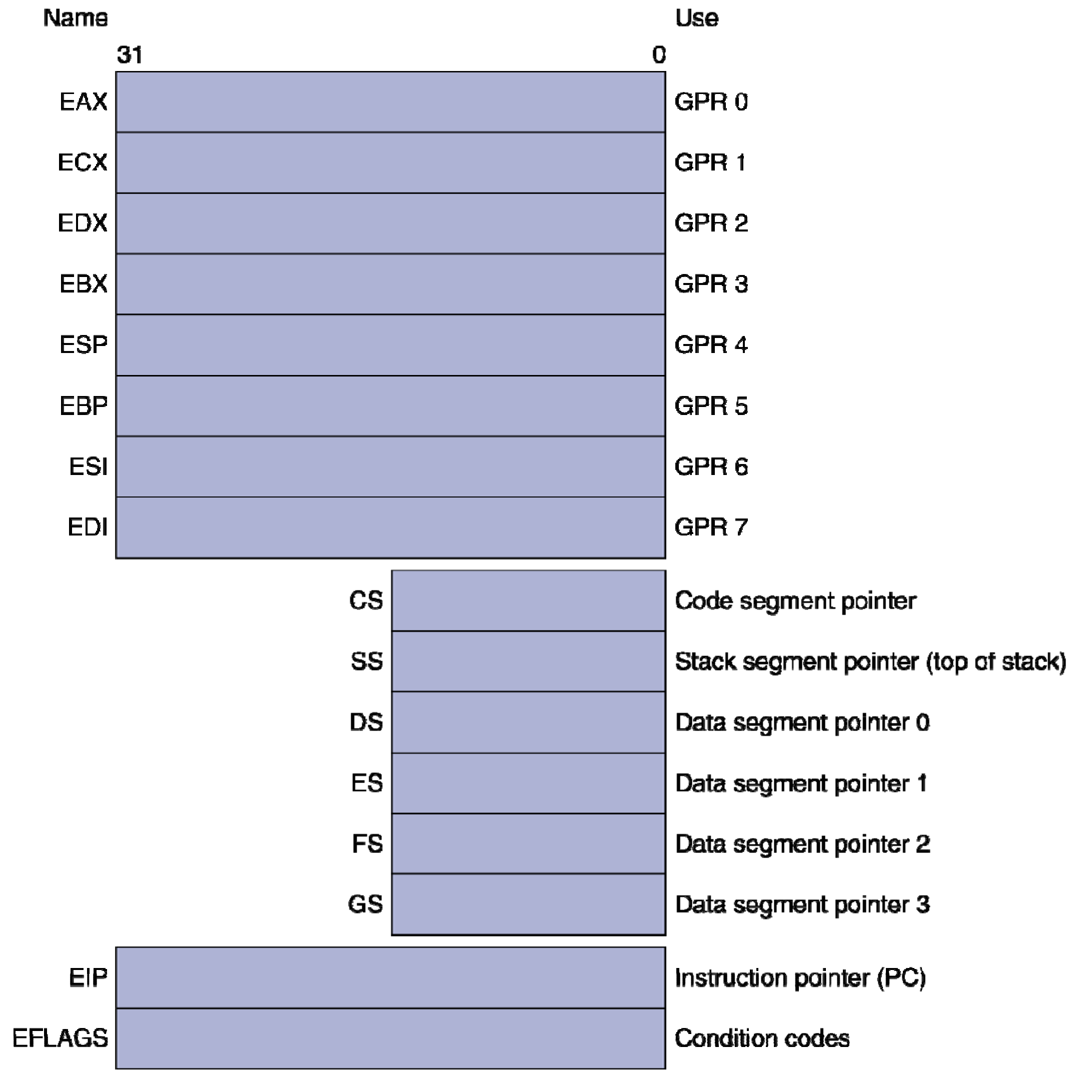
The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Basic x86 Registers



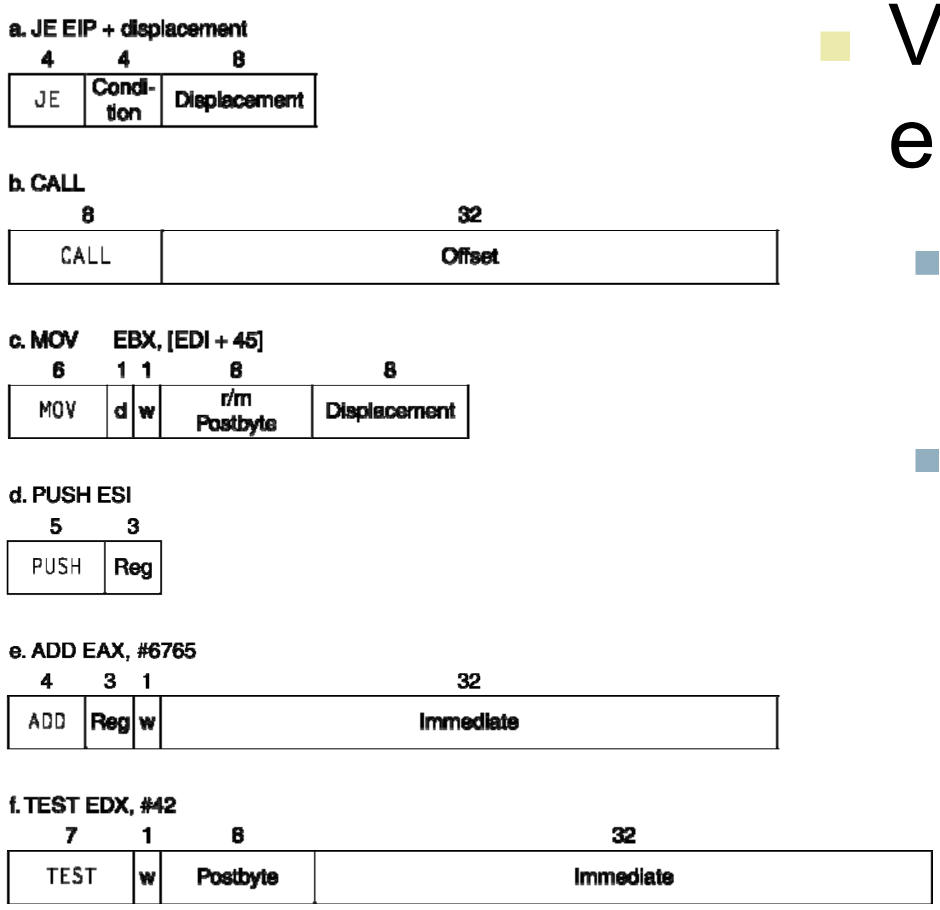
Basic x86 Addressing Modes

- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

Implementing IA-32

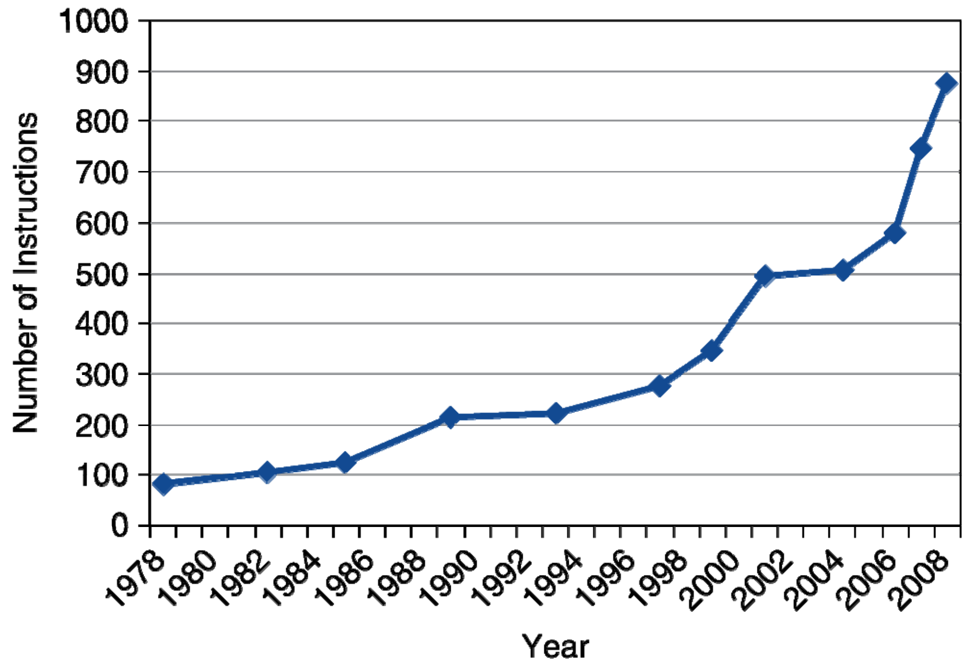
- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies

- Backward compatibility $\Rightarrow$ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`add, sub, addi`	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	12%	4%
Cond. Branch	`beq, bne, slt, slti, sltiu`	34%	8%
Jump	`j, jr, jal`	2%	0%