

The MIPS Instruction Set

Syntax Based Tools and Compilers COMP SCI 4TB3 / 6TB3

"Do you program in Assembly?" she asked. "NOP", he said.

"Low-level programming is good for the programmer's soul."

Emil Sekerinski, McMaster University, Winter 2016/17

Instruction Set

The “language” of the hardware; varies from historically simple (8 bit microcontrollers) to complex (CISC – Complex Instruction Set Computer) to modern (RISC – Reduced Instruction Set Computer)

- **MIPS**: elegant RISC architecture from 80's; now mainly in embedded systems like network, storage, cameras, printers, consumer (Sony Playstation)
- **ARMv7**: related to MIPS, 9 billion in 2011, most popular in the world; embedded devices, smartphones
- **Intel x86**: legacy 32 bit architecture; PC's
- **Intel x64**: 64 bit architecture based on RISC (with x86 emulation): PC's, servers
- **Digital Equipment VAX**: defunct CISC architecture
- **ARMv8**: extends ARMv7 to 64 bit, but close to MIPS; high-end embedded devices, smartphones
- **RISC-V**: successor to MIPS designed for low power consumption, license free and open-sourced, modular instruction set architecture supporting different applications

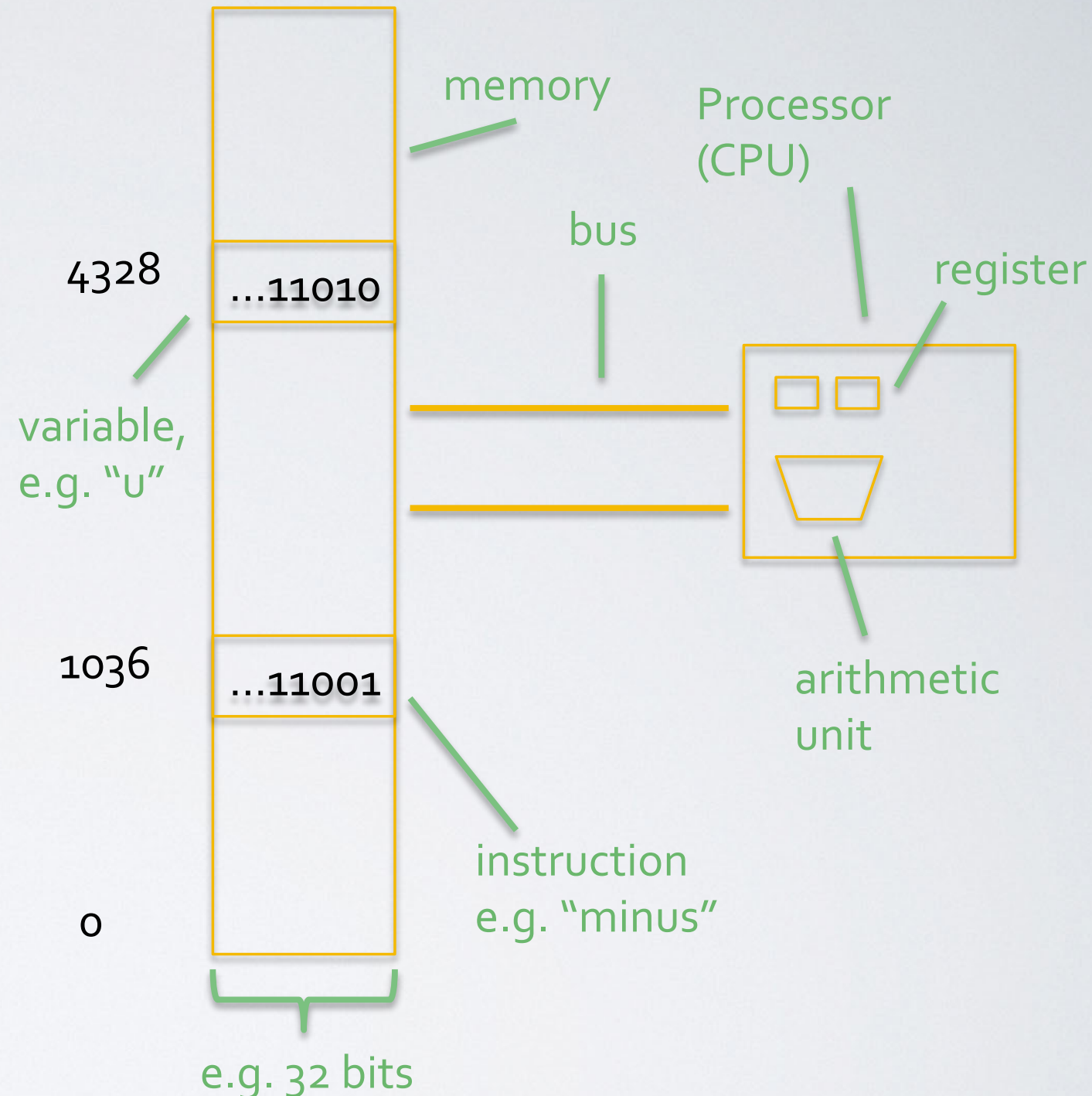
We focus on MIPS and discuss differences to other instruction set architectures.

Registers and Memory

Data and programs are stored in memory (stored-program concept, von Neumann architecture)

MIPS can only operate on data in registers: there are 32 registers (r0 .. r31) of 32 bits (1 word, 4 bytes)

In many languages (C, Java), an integer is stored with 32 bits



Arithmetic Instructions in MIPS

Arithmetic instructions have two source operands (registers rs, rt or constant imm) and one destination register (register rd). The registers are R[0] ... R[31]

add rd, rs, rt $R[rd] \leftarrow R[rs] + R[rt]$

sub rd, rs, rt $R[rd] \leftarrow R[rs] - R[rt]$

addi rd, rs, imm $R[rd] \leftarrow R[rs] + \text{imm}$

For example, the C assignments

`a = b + c;`

`d = a - e;`

All arithmetic operations have this form, following the principle: “simplicity favours regularity”
Simplicity here allows higher performance

assuming that a, b, c, d, e are associated with registers, is translated by a compiler to:

add a, b, c

sub d, a, e

Register Usage

How would a compiler translate a complex C assignment?

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

Assuming we preserve the structure and order of the evaluation, we need to use **temporary registers**:

```
add t0, g, h    # t0 is temporary register
add t1, i, j    # t1 is temporary register
sub f, t0, t1
```

An **optimizing compiler** may do better:

```
add t0, g, h    # t0 is temporary register
add f, i, j
sub f, t0, f
```

RISC architectures depend (and were developed together with) optimizing compilers, unlike CISC architectures. It is easier for humans to write and compilers to generate CISC code.

3 Questions: How many temporary registers are needed ...

1. ... for C assignment

$f = g + h + i + j;$

- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

2. ... for C assignment

$f = ((f + g) - i) + ((f - g) + j);$

- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

3. ... for C assignment

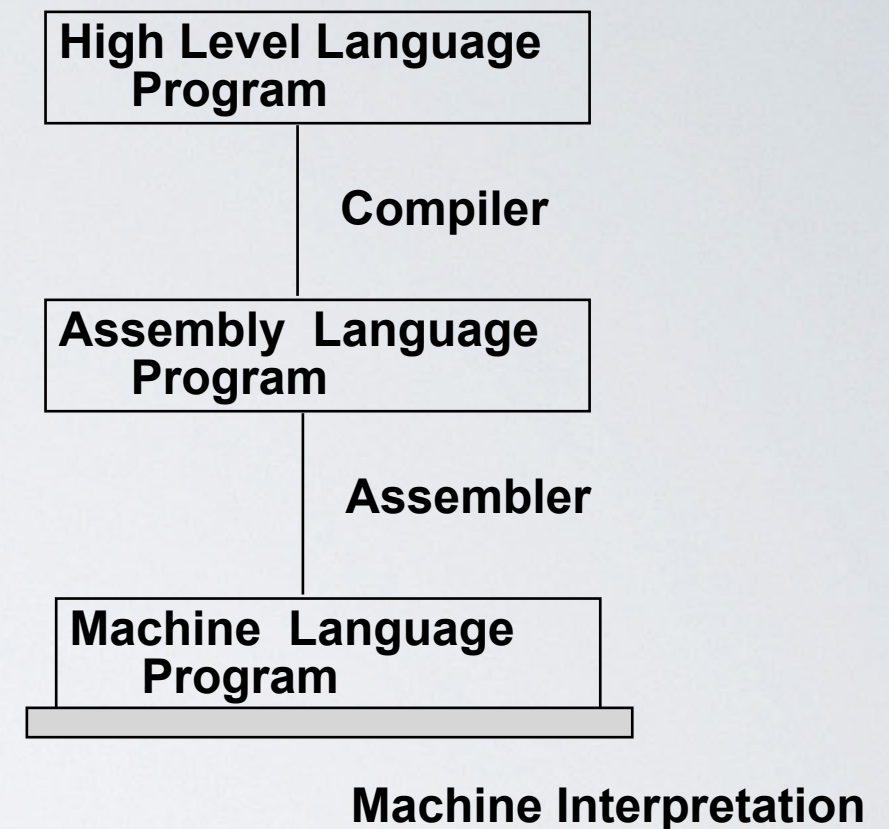
$f = ((f + g) - i) + ((f + g) - j);$

- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

MIPS Assembler: see Green Card at the front of the book

The MIPS assembler uses register names reflecting their typical usage.

The assembler also allows comments to be included (which are ignored) and memory locations to be named.



MIPS operands

Name	Example	Comments
32 registers	<code>\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at</code>	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register <code>\$zero</code> always equals 0, and register <code>\$at</code> is reserved by the assembler to handle large constants.
2^{30} memory words	<code>Memory[0], Memory[4], ..., Memory[4294967292]</code>	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.

Load and Store Instructions in MIPS

Load and store instructions require that the memory address is in one register (*rs*) and an offset (constant *imm*) is specified, which can be 0. The memory has 2^{30} words, or 2^{32} bytes, $M[0] \dots M[2^{32} - 1]$, i.e. is **byte-addressed**

`lw rt, rs, imm`

$R[rt] \leftarrow M[R[rs] + imm]$

`sw rt, rs, imm`

$M[R[rs] + imm] \leftarrow R[rt]$

These instructions load or store a whole word. Similar instructions exist for loading and storing bytes and **halfwords** (2 bytes).

For example, the C assignment

`g = h + A[8];`

assuming that *g*, *h* are associated with registers $\$s1$, $\$s2$, the base address of integer array *A* is in $\$s3$, and the array elements are stored **consecutively** $A[0], A[1], \dots$, is compiled to:

`lw $t0, 32($s3) # A[0] at $s3, A[1] at $s3+4, A[2] at $s3+8, ...`

`add $s1, $s2, $t0`

Review: Signed and Unsigned Binary Numbers

An **n-bit unsigned number** $x_{n-1}x_{n-2}\dots x_1 x_0$ is interpreted as:

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

The range is $0 \dots 2^n - 1$. For example

$$\begin{aligned} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_{\text{two}} \\ &= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 0 + \dots + 8 + 0 + 2 + 1 = 11_{\text{ten}} \end{aligned}$$

Using 32 bits, the range is $0 \dots +4,294,967,295$

A **2's complement n-bit signed number** $x_{n-1}x_{n-2}\dots x_1 x_0$ is interpreted as:

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

The range is $-2^{n-1} \dots 2^{n-1} - 1$. For example

$$\begin{aligned} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{\text{two}} \\ &= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= -2,147,483,648 + 2,147,483,644 = -4_{\text{ten}} \end{aligned}$$

Using 32 bits, the range is $-2,147,483,648 \dots +2,147,483,647$

Two'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 (neither can 2^n for unsigned numbers)

Non-negative numbers have the same unsigned and 2s-complement representation

- 0: 0000 0000 ... 0000
- -1: 1111 1111 ... 1111
- Most-negative: 1000 0000 ... 0000
- Most-positive: 0111 1111 ... 1111

To calculate $-x$, complement and 1; **complement** means $0 \rightarrow 1, 1 \rightarrow 0$

- $+2 = 0000\ 0000\ \dots\ 0010_{\text{two}}$
- $-2 = 1111\ 1111\ \dots\ 1101_{\text{two}} + 1$
 $= 1111\ 1111\ \dots\ 1110_{\text{two}}$

Sometimes a value is stored in fewer bits and then extended. **Sign extension** replicates the bit on the left. For examples from 8-bit to 16-bit

- $+2$: 0000 0010 \Rightarrow 0000 0000 0000 0010
- -2 : 1111 1110 \Rightarrow 1111 1111 1111 1110

Hexadecimal Numbers

Base 16, compact representation of bit strings with 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

For example,

eca8 6420_{hex}

= 1110 1100 1010 1000 0110 0100 0010 0000_{two}

Interlude: C Programming Language

C was developed in the early 70's at AT&T Bell Labs. It was used to implement the **Unix** operating system (Turing Award 1983 for Ritchie and Thompson)

C is a **general-purpose imperative** language: it does support **data type declarations**, **scoping**, **recursion**, and **structured programming**

C is **statically typed**. The compiler can map C constructs directly to machine instructions, such that the programmer has control over efficiency

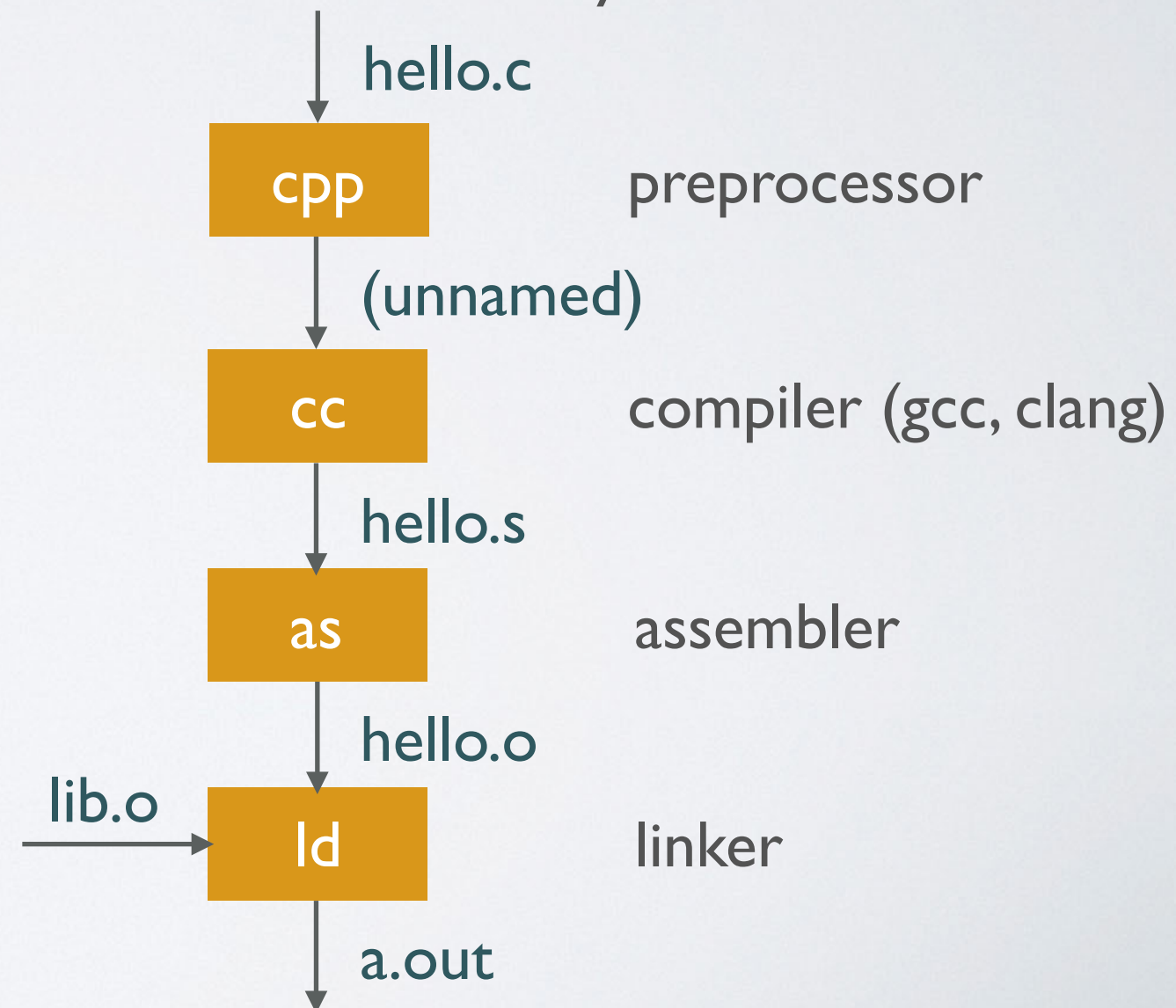
The language is minimal, relying on other tools, typically used in a **pipeline**, one of the features of Unix:

`cc hello.c` invokes all of these.

`cc -o hello hello.c` names the object file differently

`cc -c hello.c` generates `hello.o`

`cc -S hello.c` generates `hello.s`



C Program Structure

The C syntax is **free-form** (no fixed indentations), but emphasizes brevity

The size of **basic data types** depends on ISA, can be obtained with `sizeof(int)`:

`int`: 4 bytes

`char`: 1 bytes

`short`: 2 bytes

`float`: 4 bytes

`double`: 8 bytes

`unsigned int`: 4 bytes

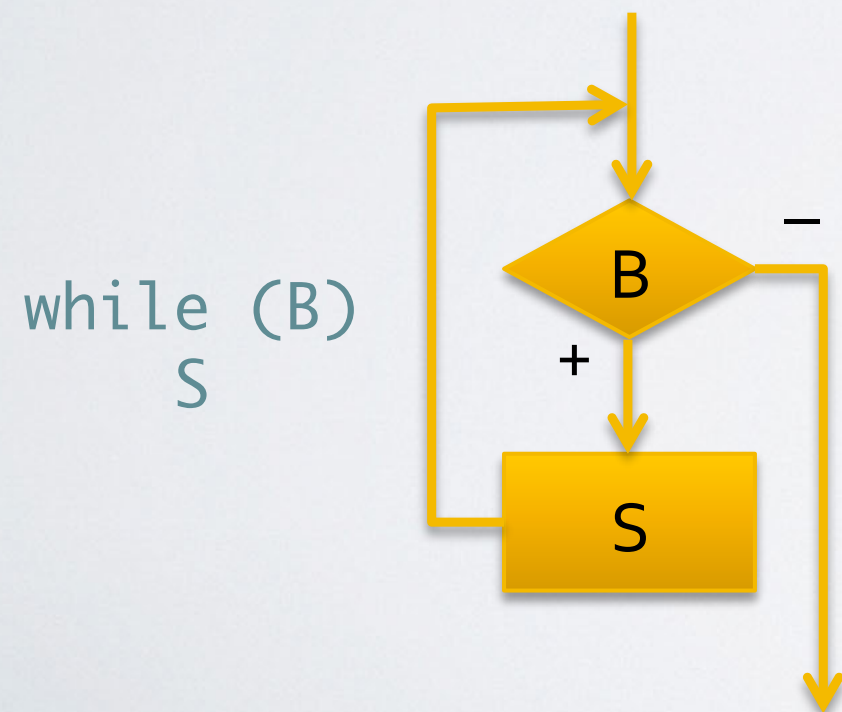
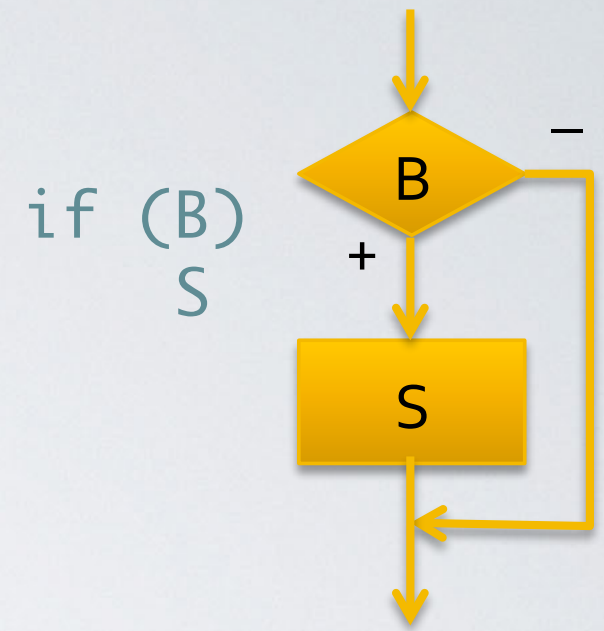
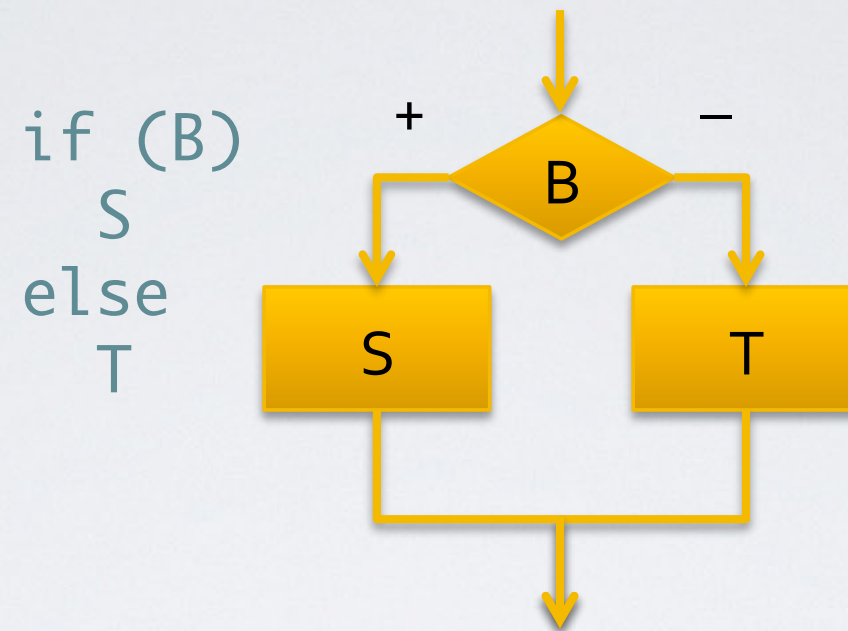
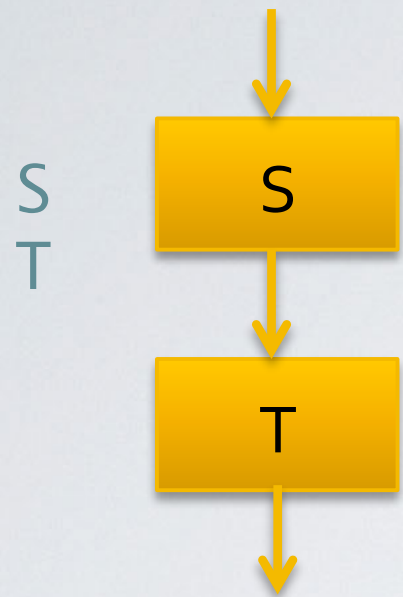
`unsigned short`: 2 bytes

```
#include <stdio.h>
    /* print Fahrenheit-Celsius table
     *      for fahr = 0, 20, ..., 300 */
int main() {
    float fahr, celsius;
    int lower, upper, step;

    lower = 0;      /* lower limit of */
    upper = 300;    /* upper limit */
    step = 20;      /* step size */

    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

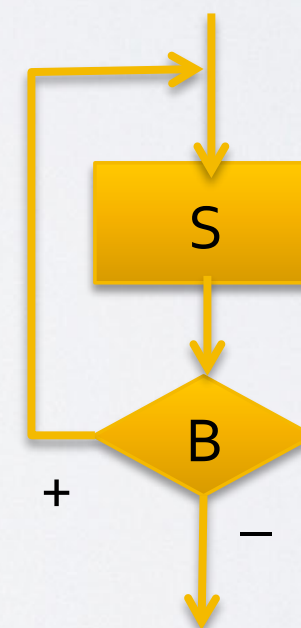
C Control Structures



$\begin{matrix} \text{do} \\ S \\ \text{while } (B) \end{matrix}$

=

$\begin{matrix} S \\ \text{while } (B) \end{matrix}$



C Pointers

The unary **operator &** gives the **address** (memory location) of a variable:

```
p = &c;
```

The unary operator ***** stands for **indirection** or **dereferencing**

```
int x = 1, y = 2, z[10];
int *ip;      /* ip is a pointer to int */
ip = &x;      /* ip now points to x */
y = *ip;      /* y is now 1 */
*ip = 0;      /* x is now 0 */
ip = &z[0];   /* ip now points to z[0] */
```

WRONG:

```
void swap(int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

CORRECT:

```
void swap(int *px, int *py) {
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

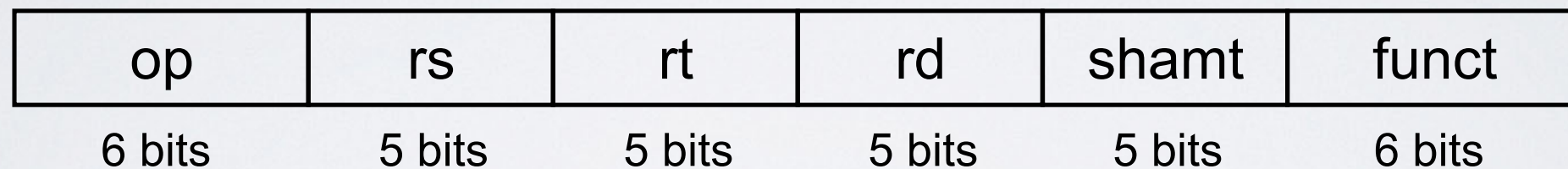
Machine Code

Each instruction is encoded as a 32-bit word

Only small number of **formats** encoding operation code, register numbers, ...

Register numbers \$t0 – \$t7 are 8 – 15, \$t8 – \$t9 are 24 – 25, \$s0 – \$s7 are 16 – 23

R-Format Instructions



- 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)

Machine Code

I-Format Instructions

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

- op: operation code (opcode)
- rs: source register number
- rt: destination register number

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32 _{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34 _{ten}	n.a.
add immediate	I	8 _{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35 _{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43 _{ten}	reg	reg	n.a.	n.a.	n.a.	address

There is no “**subi**” instruction: use **addi** with a negative operand: the operand is sign extended from 16 to 32 bits.

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
```

0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

000000|0001|100|00|00000000|00000_{two} = 02324020_{hex}

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32 _{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34 _{ten}	n.a.
add immediate	I	8 _{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35 _{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43 _{ten}	reg	reg	n.a.	n.a.	n.a.	address

Question: Converting Code to Assembly

Machine code

0010 0001 0001 0000 0000 0000 0010 0010

corresponds to

- A. `addi r16, r8, #34`
- B. `addi r8, r16, #34`
- C. `sub r8, r16, r31`
- D. `sub r31, r8, r16`
- E. None of the above

Question: Converting Code to Assembly

Machine code

1000 1101 0001 0000 0000 0000 0000 1000

corresponds to

- A. `jr r16`
- B. `jr r8`
- C. `lw r16, 8 (r8)`
- D. `lw r8, 8 (r16)`
- E. None of the above

Logical (bitwise) Operators

Useful for control of I/O devices (memory-mapped I/O), specific arithmetic expressions, efficient storage (using parts of a word), image manipulation, compression, encryption, ...

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

Shift Operations

R-Format:

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 rd, rs, shamt` is $rd = rs * 2^{shamt};$
or $rd = rs \ll shamt;$

- Shift right logical

Shift right and fill with 0 bits

`srl rd, rs, shamt` is $rd = rs / 2^{shamt};$
or $rd = rs \gg shamt;$

(for unsigned only)

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

AND, OR Operations

AND: 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: 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 Operation

NOT: useful to **invert** bits in a word

MIPS does not have NOT, but NOR, defined as $a \text{ NOR } b = \sim(a \mid b)$

NOT is expressed with one operand zero

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

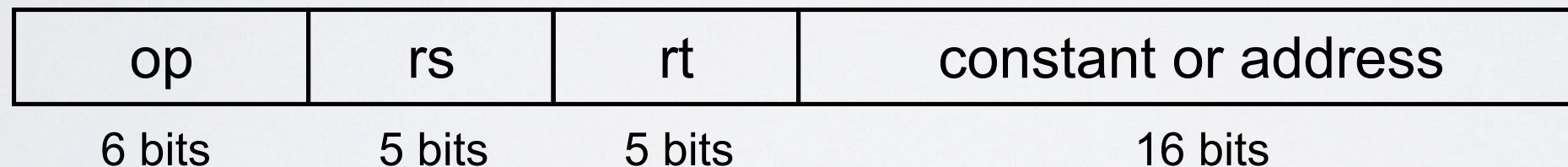
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
------	-----------------------------------------

\$t0	1111 1111 1111 1111 1100 0011 1111 1111
------	-----------------------------------------

Branch and Jump Instructions

Branch to a **labeled instruction** if a condition is true, otherwise, continue with the next instruction

- **beq** *rs*, *rt*, *L1*
if (*rs* == *rt*) branch to instruction labeled *L1*;
- **bne** *rs*, *rt*, *L1*
if (*rs* != *rt*) branch to instruction labeled *L1*;
- **beq** and **bne** are I-Format instructions



- **j** *L1*
unconditional jump to instruction labeled *L1*;
j is an **J-Format** instruction

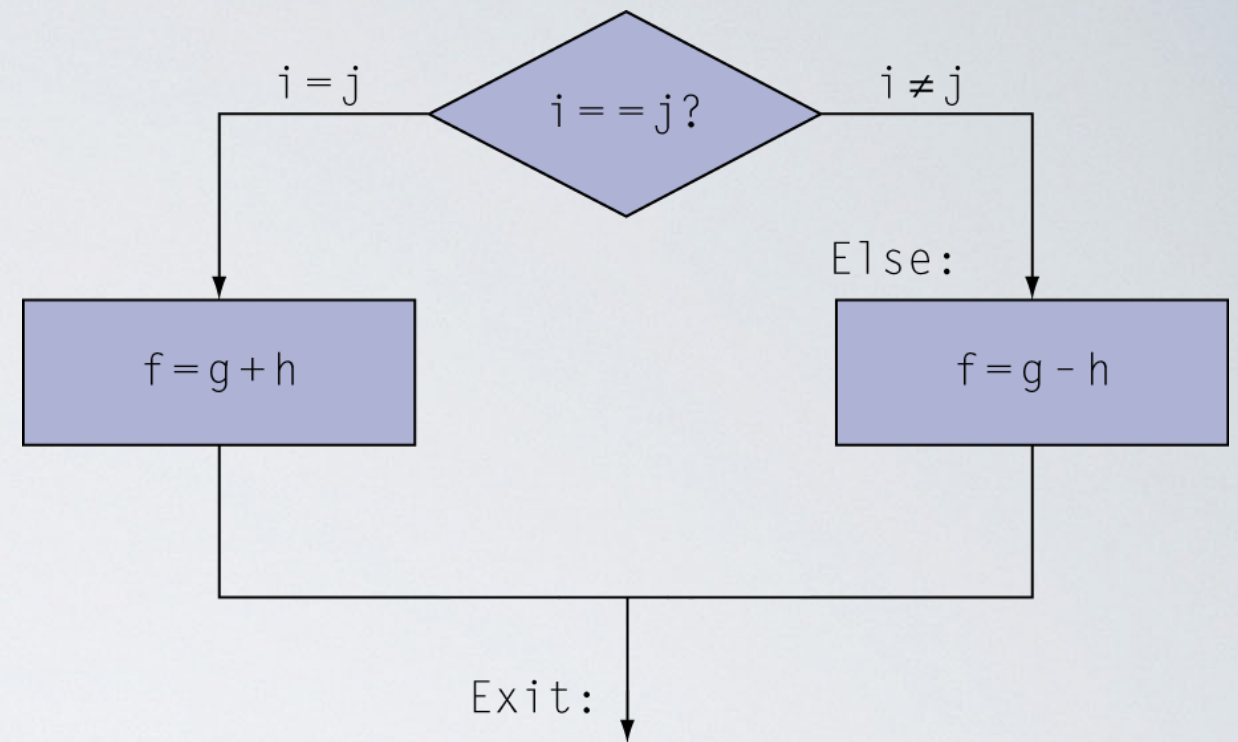


Compiling if-Statements

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

Assuming f, g, h, i, j in \$s0, \$s1, \$s2, \$s3, \$s4:

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



Compiling for-Statements

A for-statement is first translated to a while-statement

```
short A[10];  
...  
for (int i = 0; i < 10; i++) {  
    A[i] = i;  
}
```

```
short A[10];  
...  
int i = 0;  
while (i < 10) {  
    A[i] = i;  
    i++;  
}
```

Question: suppose that the base address of A is in \$s0 and has the value of 1000 (base ten). What byte address(es) correspond to A[5]?

- A. 1005
- B. 1010
- C. 1005-1006
- D. 1010-1011
- E. None of the above

Compiling while-Statements

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

Assuming i in \$s3, k in \$s5, address of save in \$s6:

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


More Conditional Operators

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, imm`
if ($rs < imm$) $rt = 1$; else $rt = 0$;

Use in combination with `beq`, `bne`

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

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

Signed vs Unsigned Comparison

Signed comparison: `slt`, `slti`

Unsigned comparison: `sltu`, `sltui`

`$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 → $t0 = 1`

`sltu $t0, $s0, $s1 # unsigned`

`+4,294,967,295 > +1 → $t0 = 0`

Procedure Calling

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

Steps required for a call:

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

Procedure Call Instructions

The **program counter** (**pc**) points to the next instruction to be executed. It is a register that is manipulated by branch and jump instructions.

Procedure call: jump and link

`jal Label`

Address of following instruction
put in \$ra

Jumps to Label: sets program
counter to Label

Procedure return: jump register

`jr $ra`

Copies \$ra to program counter

Can also be used for computed
jumps, e.g., for case/switch statements

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)

Leaf Procedure Example

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

leaf:

addi	\$sp,	\$sp,	-4
sw	\$s0,	0(\$sp)	
<hr/>			
add	\$t0,	\$a0,	\$a1
add	\$t1,	\$a2,	\$a3
sub	\$s0,	\$t0,	\$t1
<hr/>			
add	\$v0,	\$s0,	\$zero
<hr/>			
lw	\$s0,	0(\$sp)	
addi	\$sp,	\$sp,	4
<hr/>			
jr	\$ra		

Arguments g, ..., j in \$a0, ..., \$a3

f in \$s0 (hence, need to save \$s0 on stack)

Result in \$v0

Save \$s0 on stack

Procedure body

Result

Restore \$s0

Return

Non-Leaf Procedures

Procedures that call other procedures: caller needs to save on the stack:

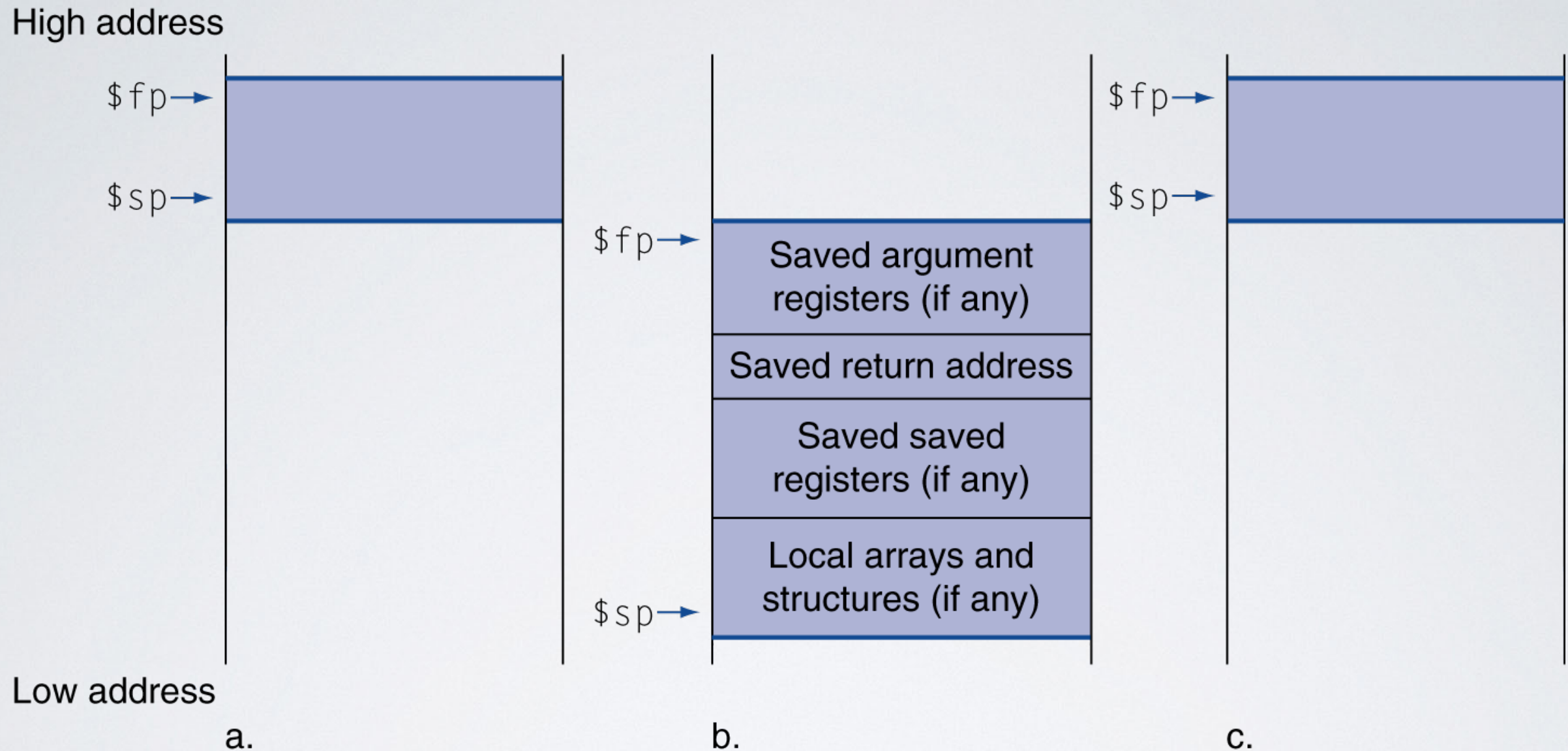
- Its return address
- Any arguments and temporaries needed after the call

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

Restore from the stack after the call
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



The frame pointer \$fp points to the beginning of the stack frame, while the stack pointer \$sp points to its end; \$fp is needed to access arguments and local variables in case it is not statically known how much the stack will grow (variable length parameters and dynamic local arrays)

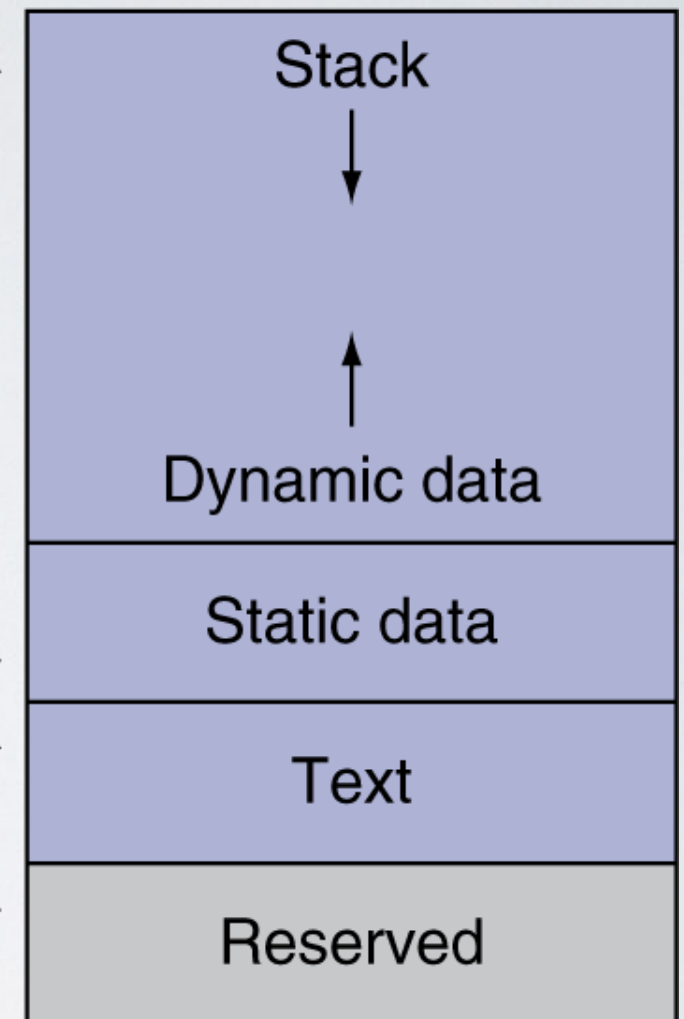
Memory Layout

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

\$sp → 7fff fffc_{hex}

\$gp → 1000 8000_{hex}
1000 0000_{hex}

pc → 0040 0000_{hex}
0



Character Data

ASCII (American Standard Code for Information Interchange) with 128 characters, 95 graphic and 33 control

Used for transmission to **teletypewriters** with 8 bits, 7 bits data plus 1 bit parity (sum of 7 bits even or odd) for error detection.

```
$ man ascii
```



00	nul	01	soh	02	stx	03	etx	04	eot	05	enq	06	ack	07	bel
08	bs	09	ht	0a	nl	0b	vt	0c	np	0d	cr	0e	so	0f	si
10	dle	11	dc1	12	dc2	13	dc3	14	dc4	15	nak	16	syn	17	etb
18	can	19	em	1a	sub	1b	esc	1c	fs	1d	gs	1e	rs	1f	us
20	sp	21	!	22	"	23	#	24	\$	25	%	26	&	27	'
28	(29)	2a	*	2b	+	2c	,	2d	-	2e	.	2f	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3a	:	3b	;	3c	<	3d	=	3e	>	3f	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4a	J	4b	K	4c	L	4d	M	4e	N	4f	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5a	Z	5b	[5c	\	5d]	5e	^	5f	_
60	`	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6a	j	6b	k	6c	l	6d	m	6e	n	6f	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7a	z	7b	{	7c		7d	}	7e	~	7f	del

Character Data

Latin-1 set with 256 characters, ASCII + 96 more graphic characters

Introduced with DEC VT220, used by Windows, Unix, http



```
xterm
bash-3.2$ echo $('\\x77'
w
bash-3.2$ echo $('\\x7'

bash-3.2$ echo $('\\xC4'
Ä
bash-3.2$ echo $LANG
bash-3.2$ █
```

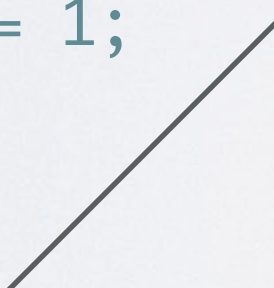
NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
00A0	00A1	00A2	00A3	00A4	00A5	00A6	00A7	00A8	00A9	00AA	00AB	00AC	00AD	00AE	00AF
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
00B0	00B1	00B2	00B3	00B4	00B5	00B6	00B7	00B8	00B9	00BA	00BB	00BC	00BD	00BE	00BF
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
00C0	00C1	00C2	00C3	00C4	00C5	00C6	00C7	00C8	00C9	00CA	00CB	00CC	00CD	00CE	00CF
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
00D0	00D1	00D2	00D3	00D4	00D5	00D6	00D7	00D8	00D9	00DA	00DB	00DC	00DD	00DE	00DF
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
00E0	00E1	00E2	00E3	00E4	00E5	00E6	00E7	00E8	00E9	00EA	00EB	00EC	00ED	00EE	00EF
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
ø	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ
00F0	00F1	00F2	00F3	00F4	00F5	00F6	00F7	00F8	00F9	00FA	00FB	00FC	00FD	00FE	00FF
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Strings

A string is an array of characters. For variable number of character:

- allow only strings of fixed lengths
- first byte of the string has its length
- length of string in accompanying variable
- string terminated with special character: C uses 0 (ASCII nul)

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



Beware, condition with side effect!

Some Python implementations store a string as a C string plus length (plus hash code plus ...). Why?

String Operations

Store 4 chars in a word and could use bitwise operations to access them. Easier:

lb rt, offset(rs) sign extend to 32 bits

lbu rt, offset(rs) zero extend to 32 bits

sb rt, offset(rs) store rightmost byte

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

Assuming addresses of x, y in \$a0, \$a1, i in \$s0:

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
```

Unicode

some 120,000 characters, most of the world's alphabets plus symbols

Used in Python, Java, Unix, ...

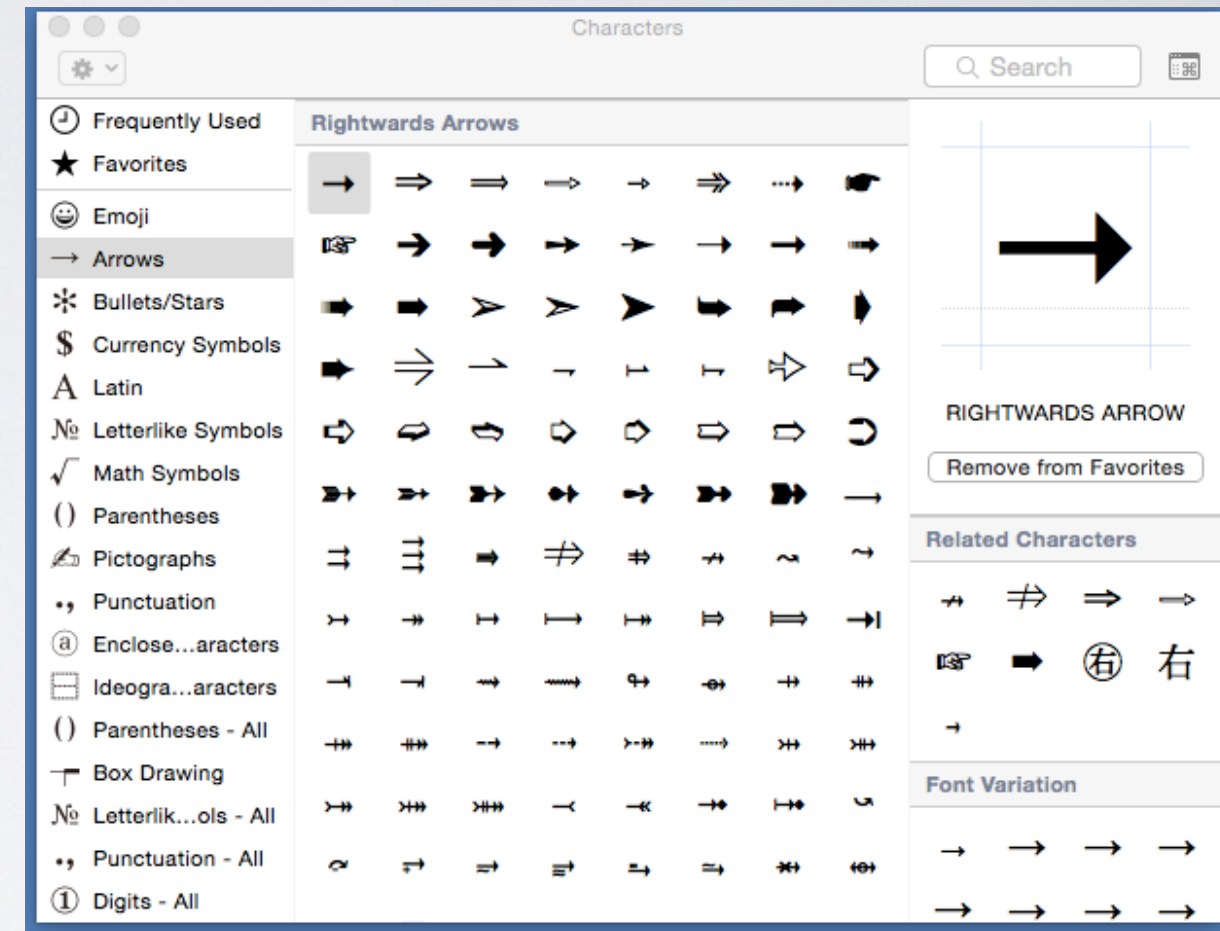
Several encodings: UTF-8, UTF-16, UTF-32

UTF-8, UTF-16: **variable length encoding** used by Python, Java

lh, **lhu**, **sb**: load and store halfwords

UTF-8: compatible with ASCII

Python, Java don't allow assignments to `str[i]`, C does. Why?



Bits of code point	First code point	Last code point	Bytes in sequence	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
7	U+0000	U+007F	1	0xxxxxxx					
11	U+0080	U+07FF	2	110xxxxx	10xxxxxx				
16	U+0800	U+FFFF	3	1110xxxx	10xxxxxx	10xxxxxx			
21	U+10000	U+1FFFFF	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
26	U+200000	U+3FFFFFFF	5	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
31	U+4000000	U+7FFFFFFF	6	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

32-bit Constants

Most constants are small: 16-bit immediate of I-Format is sufficient

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

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

`lui $s0, 61`

0000 0000 0011 1101	0000 0000 0000 0000
---------------------	---------------------

`ori $s0, $s0, 2304`

0000 0000 0011 1101	0000 1001 0000 0000
---------------------	---------------------

Question: what is the resulting value in \$s0?

A. 612304

D. 150995005

B. 230461

E. none of the above

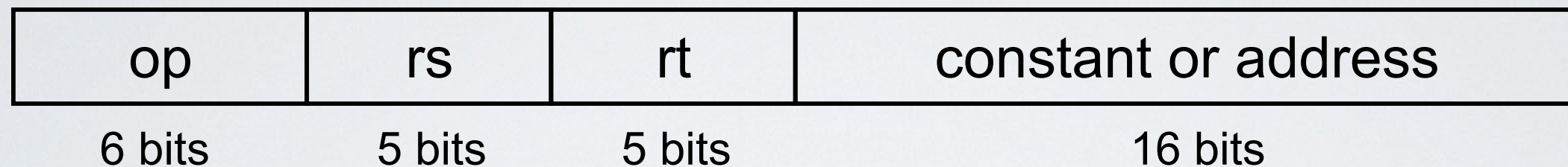
C. 4000000

Branch Addressing

Branch instructions specify opcode, two registers, target address

`beq rs, rt, L1`

Most branch targets are near (`if, while`), forward or backward: I-Format



PC-relative addressing (PC already incremented by 4):

$$\text{target address} = \text{PC} + \text{offset} \times 4$$

Jump Addressing

Jump (j and jal) targets could be anywhere in the code

`jal Label`

Encode full address in instruction: J-Format



(Pseudo) **direct addressing**:

$$\text{target address} = \text{PC}_{31..28} : \text{address} \times 4$$

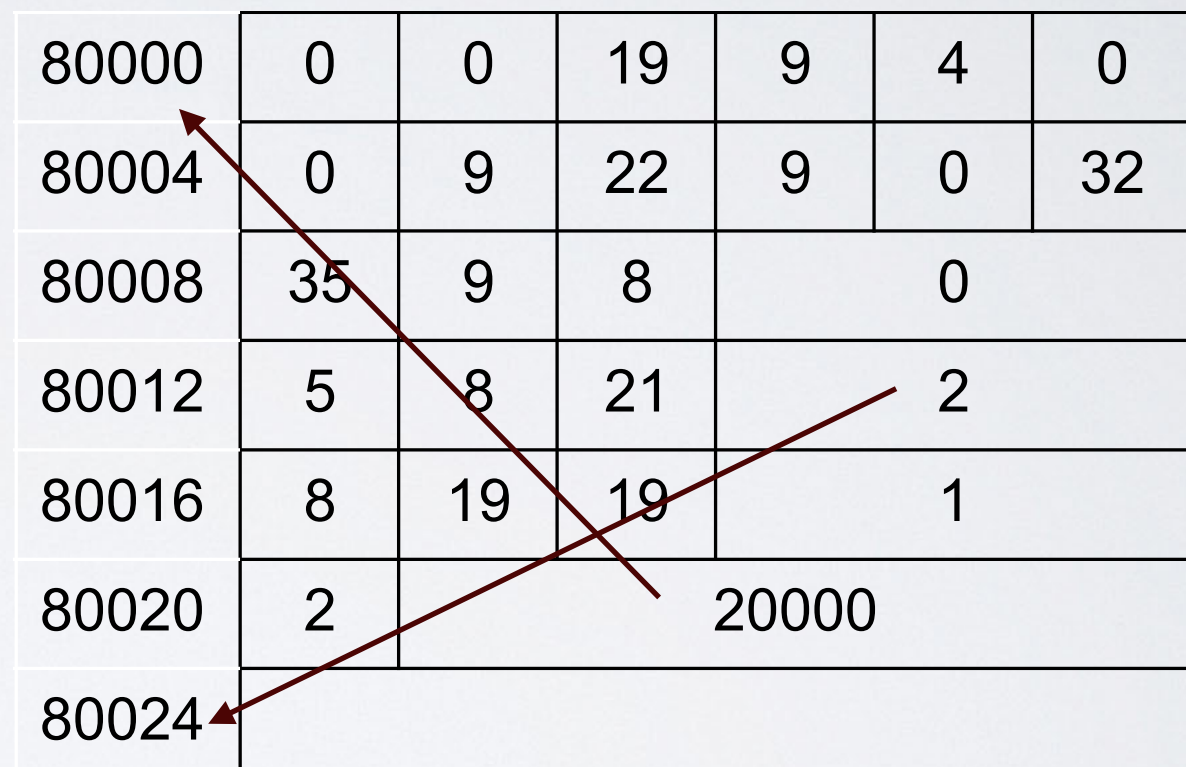
Target Addressing Example

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

Consider above `while`-loop with `i` in `$s3`, `k` in `$s5`, address of `save` in `$s6`:

```
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

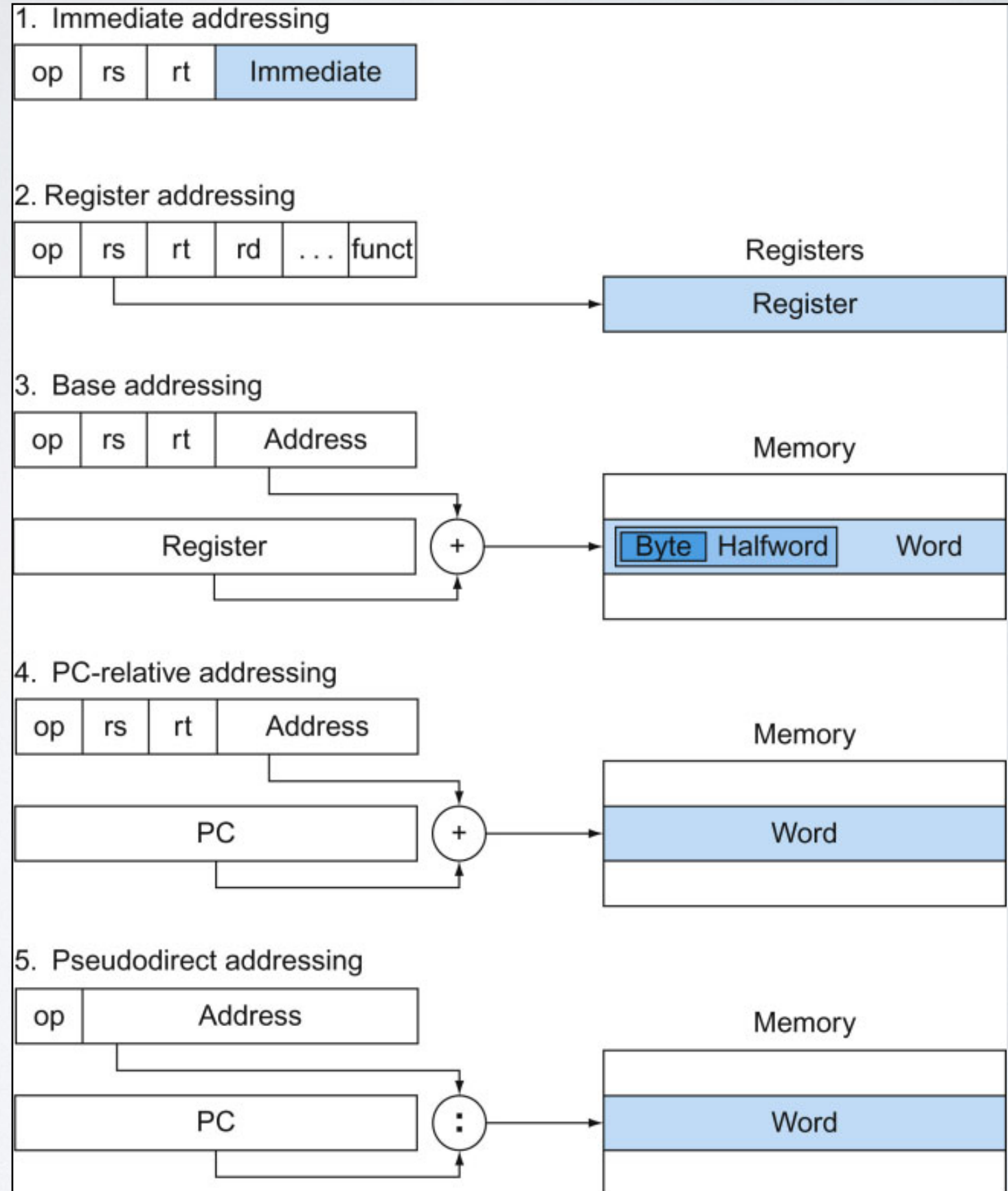
If branch target is too far to encode with 16-bit offset, assembler rewrites the code:

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

Another example of making the common case fast.

Addressing Mode Summary

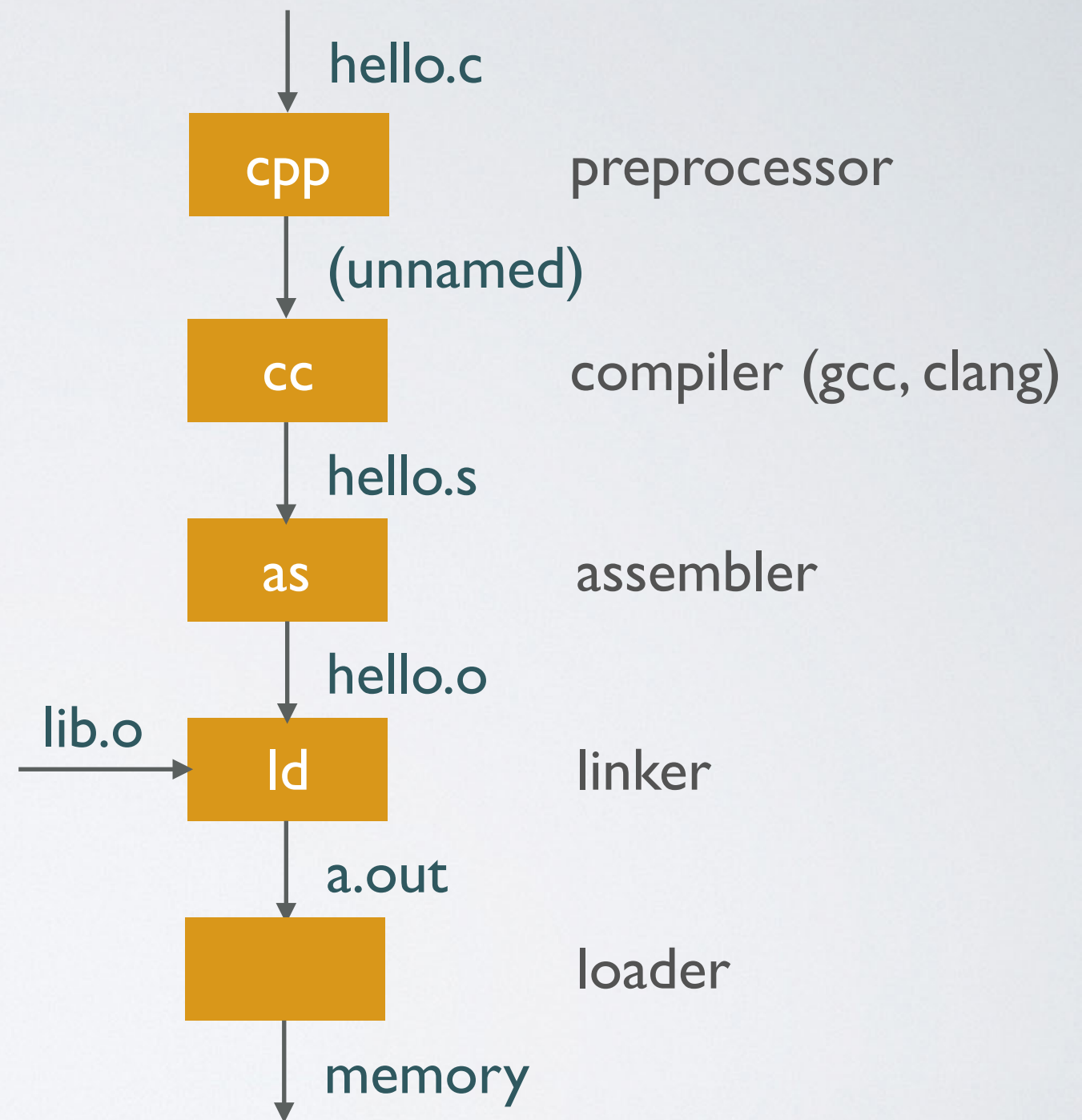
1. `addi $s3, $s3, 1`
2. `sub $s0, $s1, $s2`
3. `lw $t0, 32($s3)`
4. `bne $t0, $zero, L`
5. `j Exit`



Translation and Startup

Compilers may produce object files directly

Static linking produces a self-contained executable file



Assembler Pseudoinstructions

Most assembler instructions represent machine instructions one-to-one

Pseudoinstructions: figments of the assembler's imagination

<code>move \$t0, \$t1</code>	→	<code>add \$t0, \$zero, \$t1</code>
<code>blt \$t0, \$t1, L</code>	→	<code>slt \$at, \$t0, \$t1</code> <code>bne \$at, \$zero, L</code>

`$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 (global variables, constants, in particular strings, floating-point, array constants)
- **Relocation info**: for contents that depend on absolute location of loaded program
- **Symbol table**: global definitions (exported variables, procedures) and external refs (imported variables, procedures)
- **Debug info**: for associating with source code (for range of instructions in text segment, line number in source file)

Linking Object Modules

Produces an **executable image**

1. merges segments
2. resolves labels (determine their addresses)
3. patches location-dependent and external refs

Could leave location dependencies for fixing by a **relocating loader**

- with virtual memory, no need to do this
- program can be loaded into absolute location in virtual memory space

Linking Example

assuming \$gp set correspondingly

Object file header			
	Name	Procedure A	
	Text size	100 _{hex}	
	Data size	20 _{hex}	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(X)	
	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	-	
	B	-	

Executable file header		
	Text size	300 _{hex}
	Data size	50 _{hex}
Text segment	Address	Instruction
	0040 0000 _{hex}	lw \$a0, 8000 _{hex} (\$gp)
	0040 0004 _{hex}	jal 40 0100 _{hex}

	0040 0100 _{hex}	sw \$a1, 8020 _{hex} (\$gp)
Data segment	0040 0104 _{hex}	jal 40 0000 _{hex}

	1000 0000 _{hex}	(X)

	1000 0020 _{hex}	(Y)

Object file header			
	Name	Procedure B	
	Text size	200 _{hex}	
	Data size	30 _{hex}	
Text segment	Address	Instruction	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	
Data segment	0	(Y)	
	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	-	
	A	-	

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
4. Set up arguments on stack
5. Initialize registers (including \$sp, \$fp, \$gp)
6. Jump to startup procedure (copies arguments to \$a0, ... and calls main)

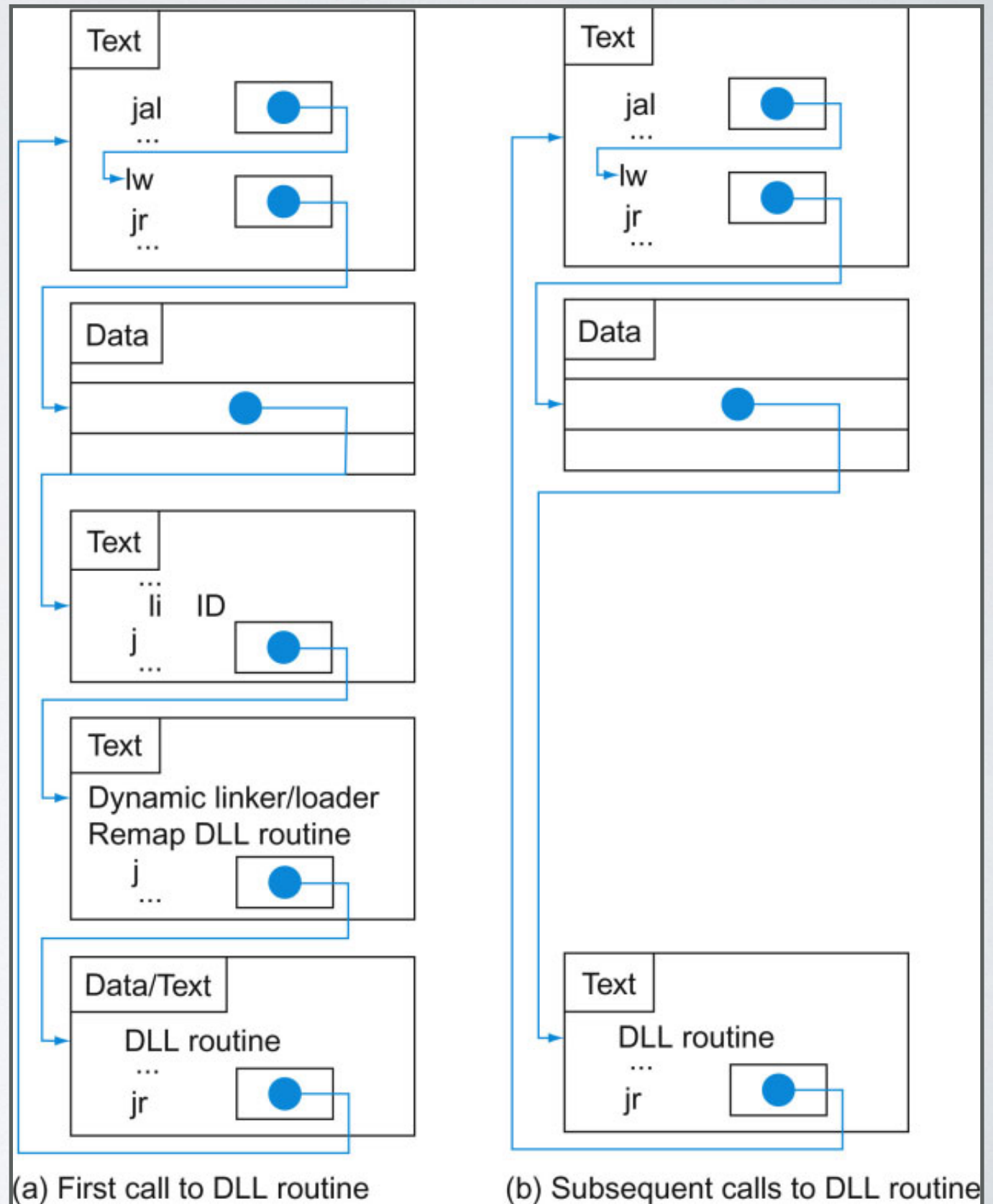
Dynamic Linking

Static linking can lead to bloated executables because all (transitively) references libraries are included.

Dynamic linking only links and loads a library procedure when it is called

Requires procedure code to be **relocatable**

Automatically picks up new library versions



Virtual Machines

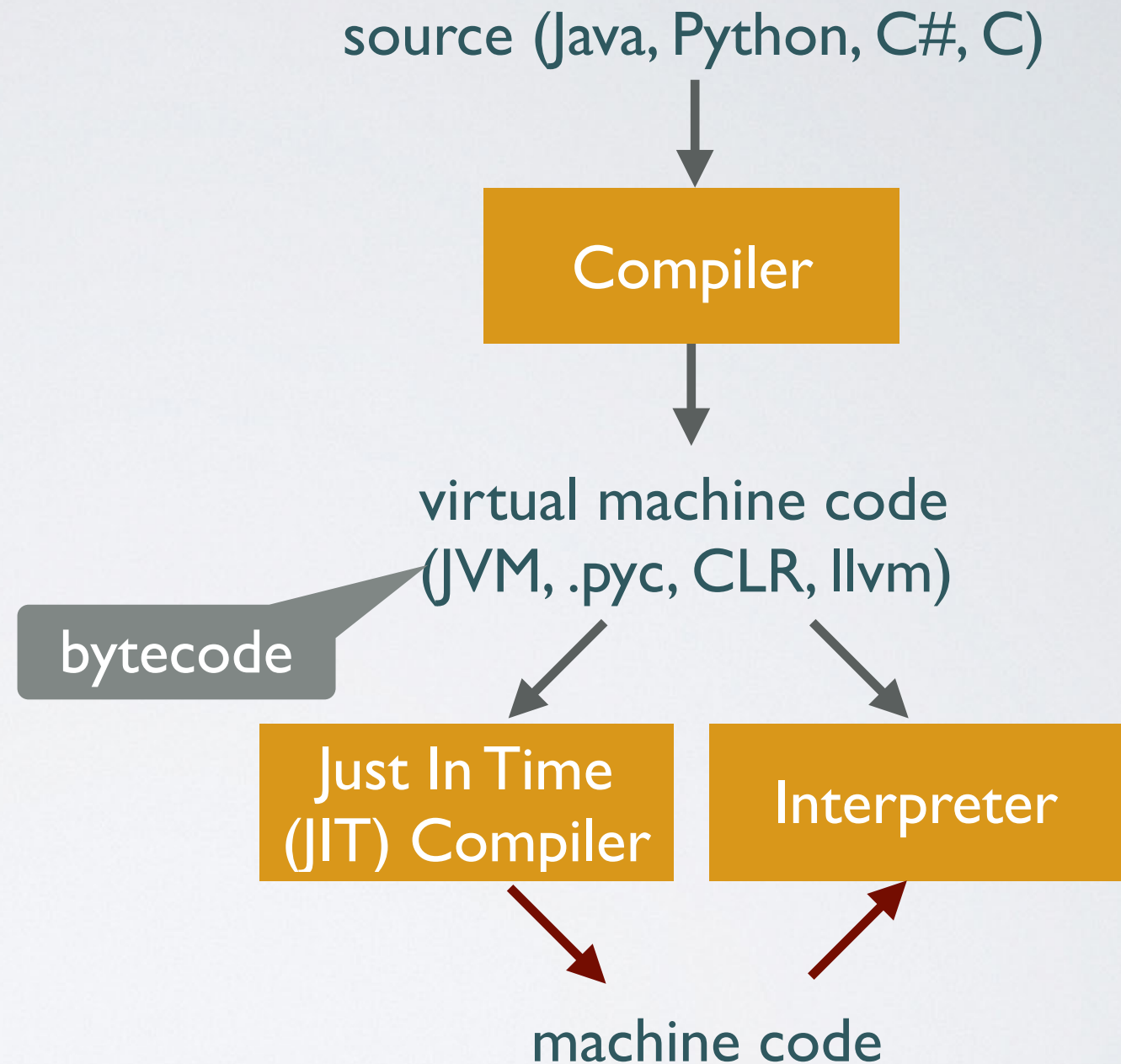
Reasons for virtual machines:

Compactness: virtual machines are typically stack architectures with one byte instructions; RISC code is several times larger.

Abstractness: virtual machines may have provisions for method lookup, garbage collection, threads, arithmetic.

Portability: interpretation and compilation of virtual machine code can take place on any host machine.

Security: typed virtual machine code is more easily checked for type and memory violations.



A Sorting Example

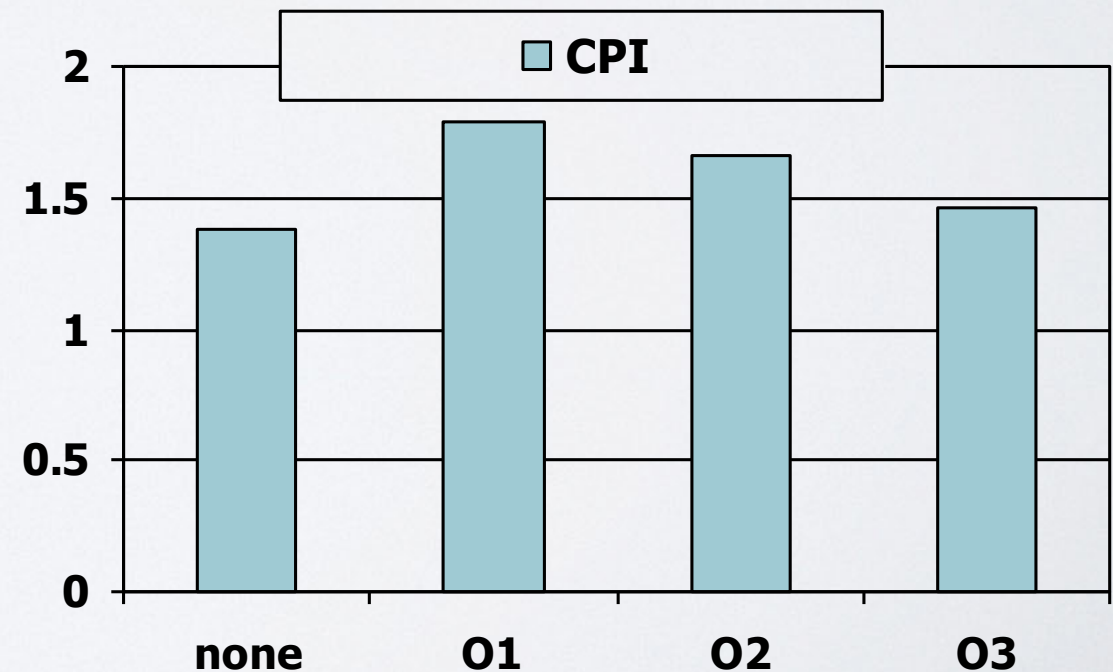
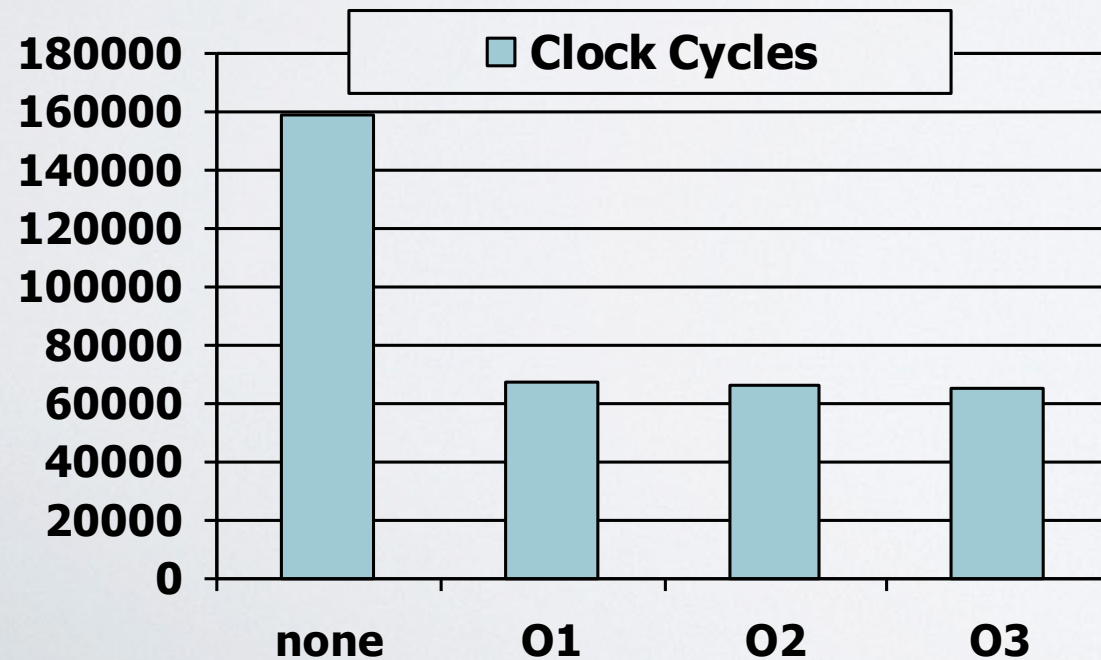
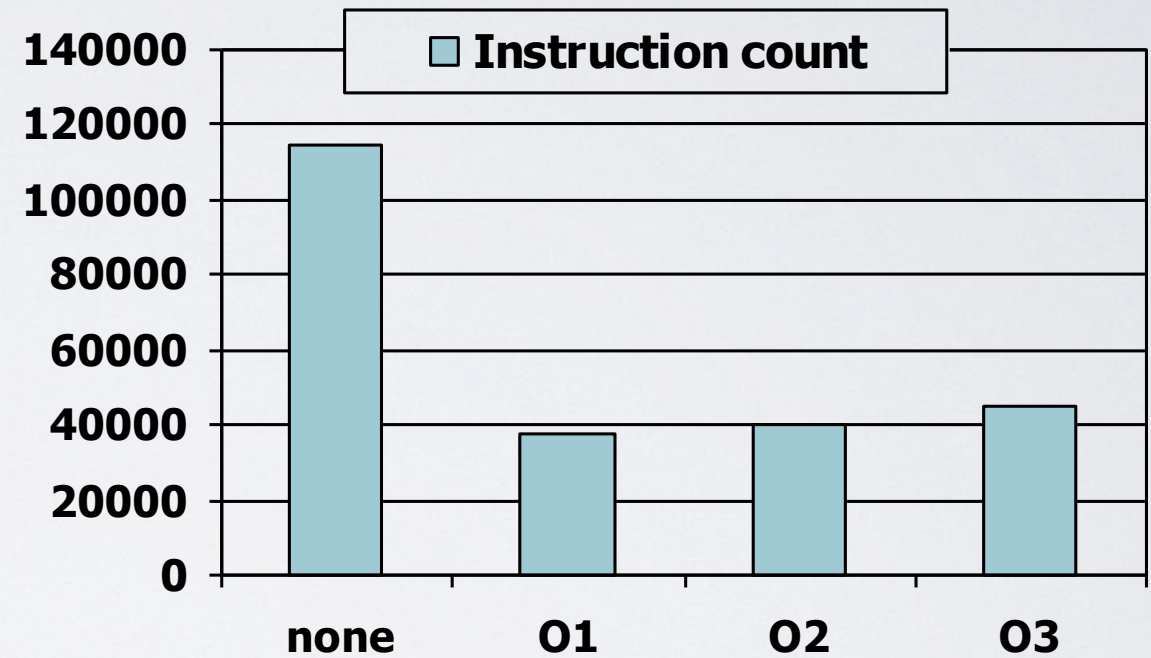
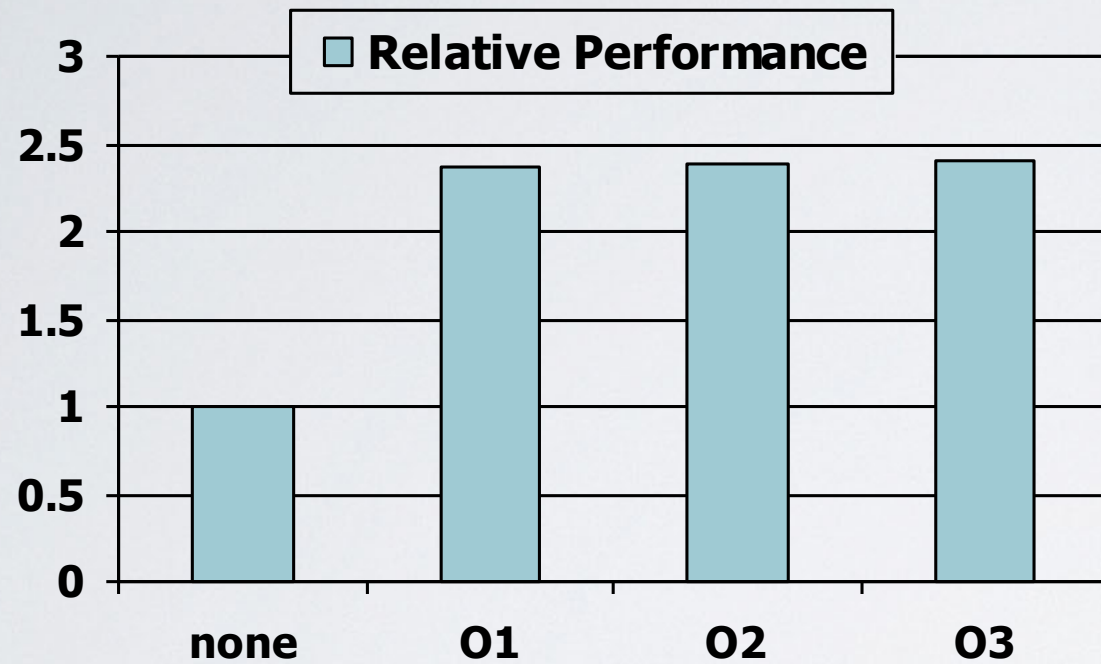
```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

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

v[7]	44	44	44	44	44	44	44	94
v[6]	55	55	55	55	55	55	94	67
v[5]	12	12	12	12	12	94	67	55
v[4]	42	42	42	42	94	67	55	44
v[3]	94	94	94	94	67	42	42	42
v[2]	18	18	67	67	42	18	18	18
v[1]	06	67	18	18	18	12	12	12
v[0]	67	06	06	06	06	06	06	06

Effect of Compiler Optimization

Using gcc for Pentium 4 under Linux

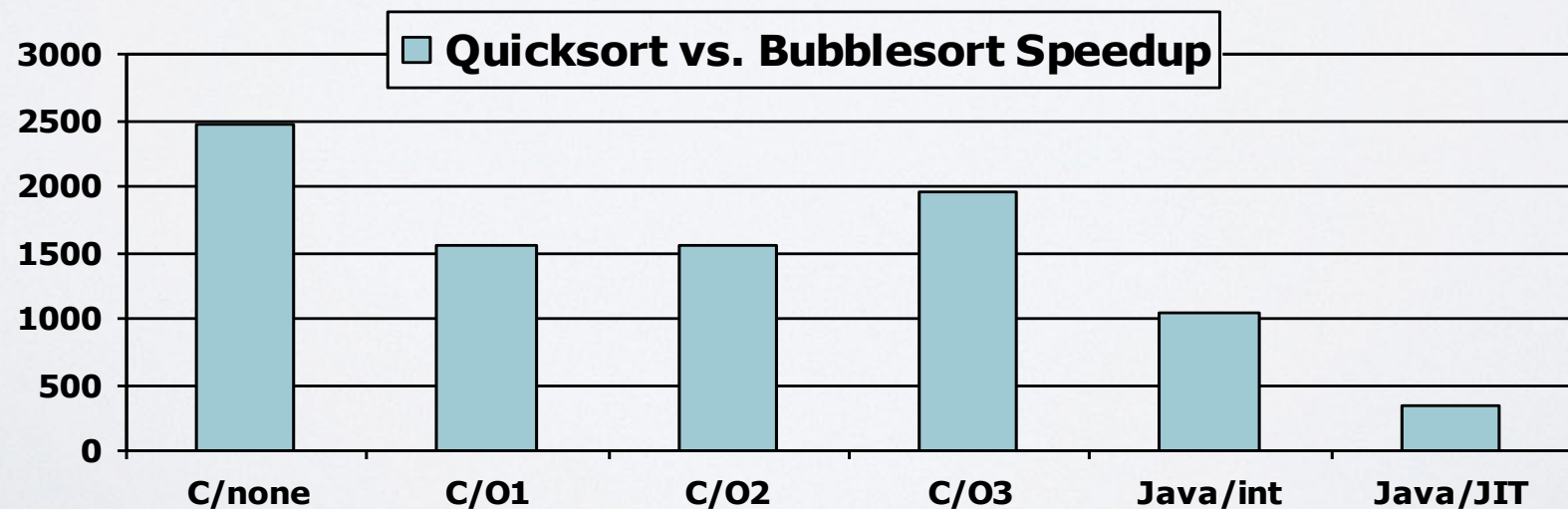
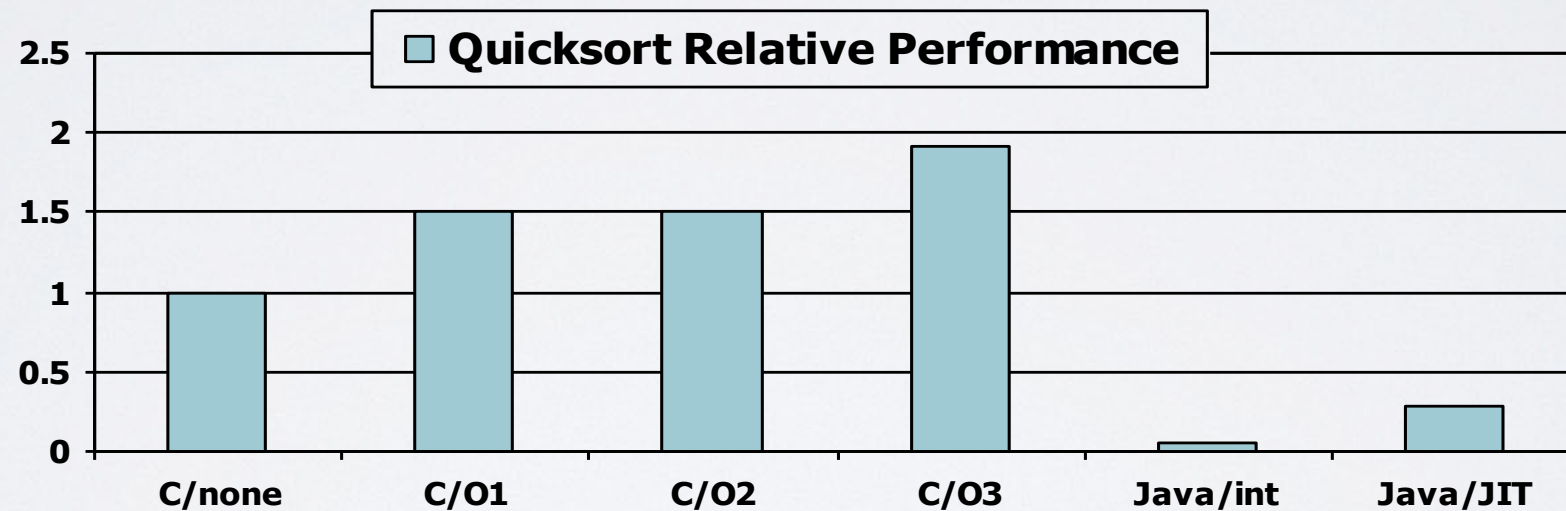
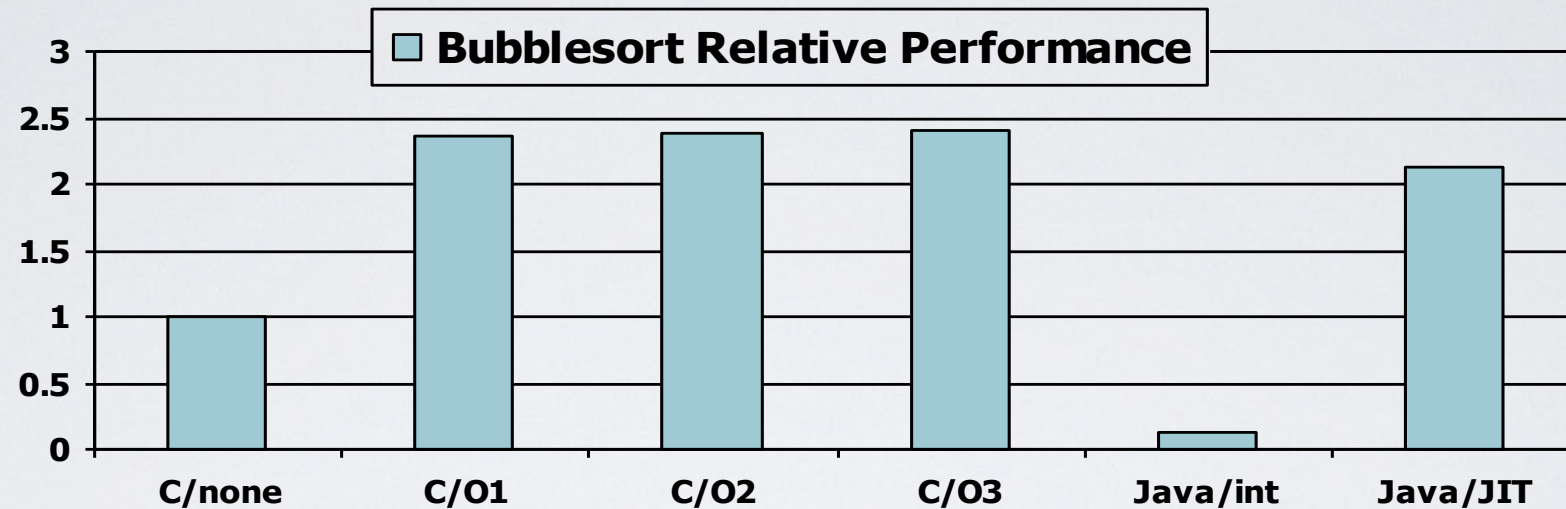


Background: Compiler Optimization in gcc

Optimization name	Explanation	gcc level
<i>High level</i> Procedure integration	<i>At or near the source level; processor independent</i> Replace procedure call by procedure body	03
<i>Local</i> Common subexpression elimination Constant propagation Stack height reduction	<i>Within straight-line code</i> Replace two instances of the same computation by single copy Replace all instances of a variable that is assigned a constant with the constant Rearrange expression tree to minimize resources needed for expression evaluation	01 01 01
<i>Global</i> Global common subexpression elimination Copy propagation Code motion Induction variable elimination	<i>Across a branch</i> Same as local, but this version crosses branches Replace all instances of a variable A that has been assigned X (i.e., $A = X$) with X Remove code from a loop that computes same value each iteration of the loop Simplify/eliminate array addressing calculations within loops	02 02 02 02
<i>Processor dependent</i> Strength reduction Pipeline scheduling Branch offset optimization	<i>Depends on processor knowledge</i> Many examples; replace multiply by a constant with shifts Reorder instructions to improve pipeline performance Choose the shortest branch displacement that reaches target	01 01 01

Effect of Language and Algorithm

(Python sort uses a variant of quicksort)



Some Conclusions

- 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: modern Java/JIT compilers generate code that is comparable to and may outperform compiled C code!
- Nothing can fix a dumb algorithm

Arrays vs. Pointers in C

Array indexing involves

- multiplying index by element size
- adding to array base address

Pointers correspond directly to memory addresses

- can reduce cost of indexing

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

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

```
        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. Pointers

Multiply **strength reduced** to shift

Array version requires shift to be inside loop

- part of index calculation for incremented i
- c.f. incrementing pointer

The the possibility to write “optimized” code contributed to the popularity of C

Modern compilers achieve same effect as manual use of pointers (**induction variable elimination**)

Use of pointers is a common source of difficult to detect errors:

- **better to make program clearer and safer**

ARM vs MIPS

ARM most popular embedded and mobile ISA:

	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

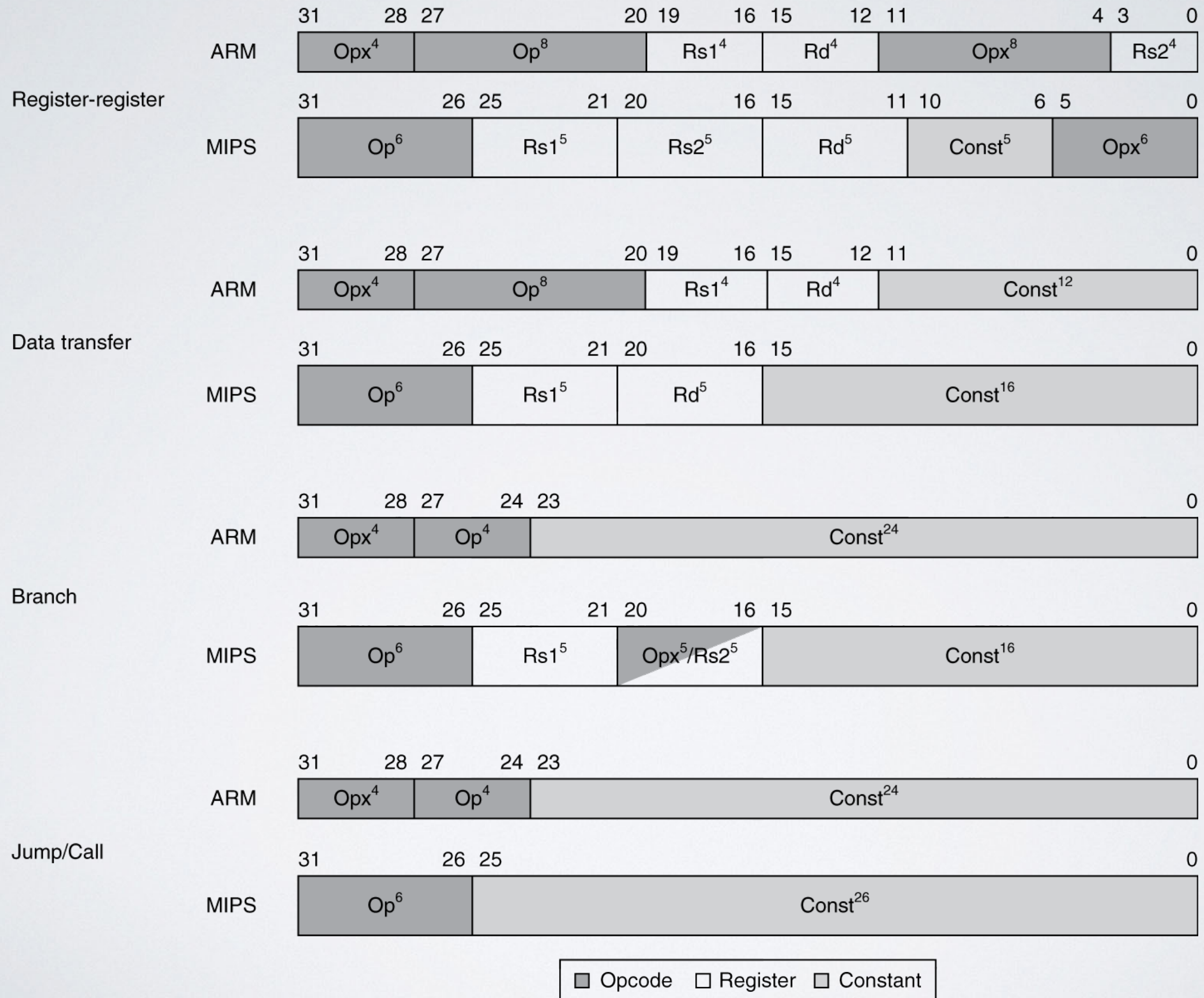
Uses **condition codes** for result of an arithmetic/logical instruction

- Negative, zero, carry, overflow
- Compare instructions set condition codes without keeping the result

Each instruction can be conditional

- Top 4 bits of instruction word are the 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
- 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
- Pentium III (1999): Added SSE (Streaming SIMD Extensions) and associated registers
- Pentium 4 (2001) New microarchitecture: Added SSE2 instructions

... The Intel x86 ISA

- 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 \neq market success

Basic x86 Registers

Name	31	0	Use
EAX			GPR 0
ECX			GPR 1
EDX			GPR 2
EBX			GPR 3
ESP			GPR 4
EBP			GPR 5
ESI			GPR 6
EDI			GPR 7
	CS		Code segment pointer
	SS		Stack segment pointer (top of stack)
	DS		Data segment pointer 0
	ES		Data segment pointer 1
	FS		Data segment pointer 2
	GS		Data segment pointer 3
EIP			Instruction pointer (PC)
EFLAGS			Condition codes

Basic x86 Addressing Modes

Two operands per instruction (one operand can be source and destination):

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

Memory addressing modes:

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

x86 Instruction Encoding

a. JE EIP + displacement

4	4	8
JE	Condition	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

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

ARM v8 ISA

In moving to 64-bit, ARM did a complete overhaul: ARM v8 resembles MIPS

Changes from v7:

- No conditional execution field
- Immediate field is 12-bit constant
- Dropped load/store multiple
- PC is no longer a GPR
- GPR set expanded to 32
- Addressing modes work for all word sizes
- Divide instruction
- Branch if equal/branch if not equal instructions

Fallacies

Powerful instruction \Rightarrow higher performance

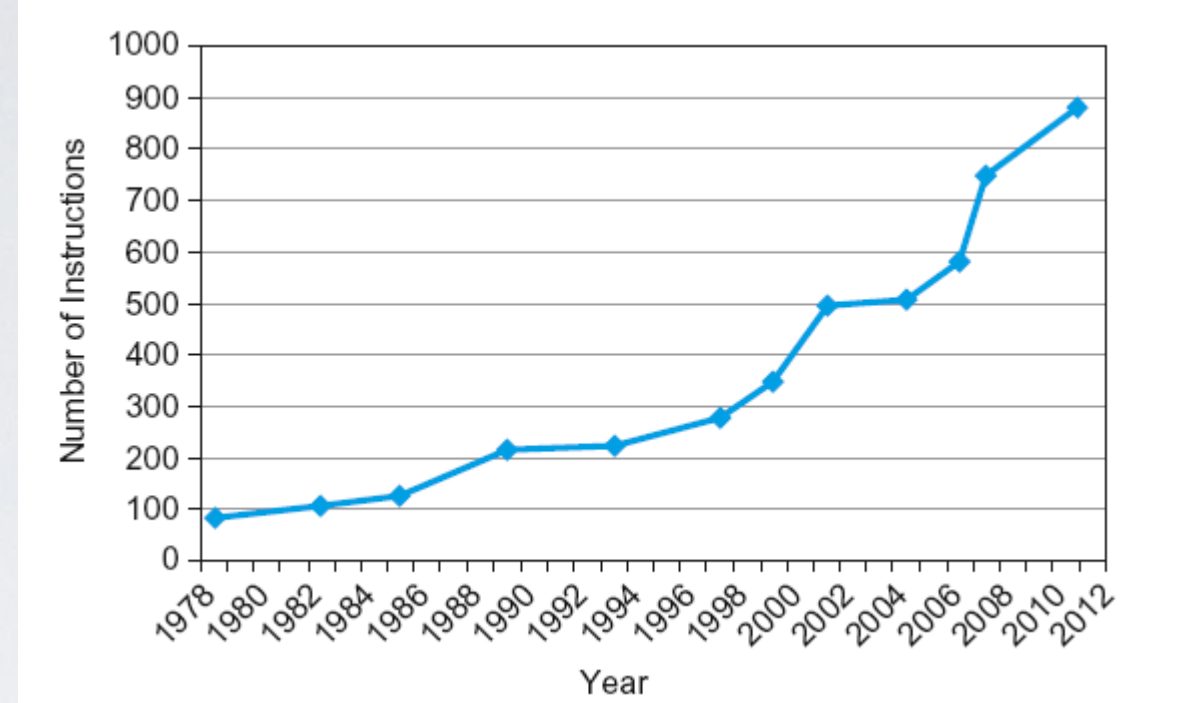
- Fewer instructions required
- But complex instructions are hard to implement
- May slow down all instructions, including simple ones
- Compilers are good at making fast code from simple instructions

Use assembly code for high performance

- But modern compilers are better at dealing with modern processors
- More lines of code \Rightarrow more errors and less productivity

Backward compatibility \Rightarrow instruction set doesn't change

- But they do accrete more instructions



x86 instruction set

Concluding Remarks

Design principles

- Simplicity favours regularity
- Smaller is faster
- Make the common case fast
- Good design demands good compromises

Layers of software/hardware: Compiler, assembler, hardware

MIPS: typical of RISC ISAs vs x86

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%

Measure MIPS instruction executions in benchmark programs

- Consider making the common case fast
- Consider compromises