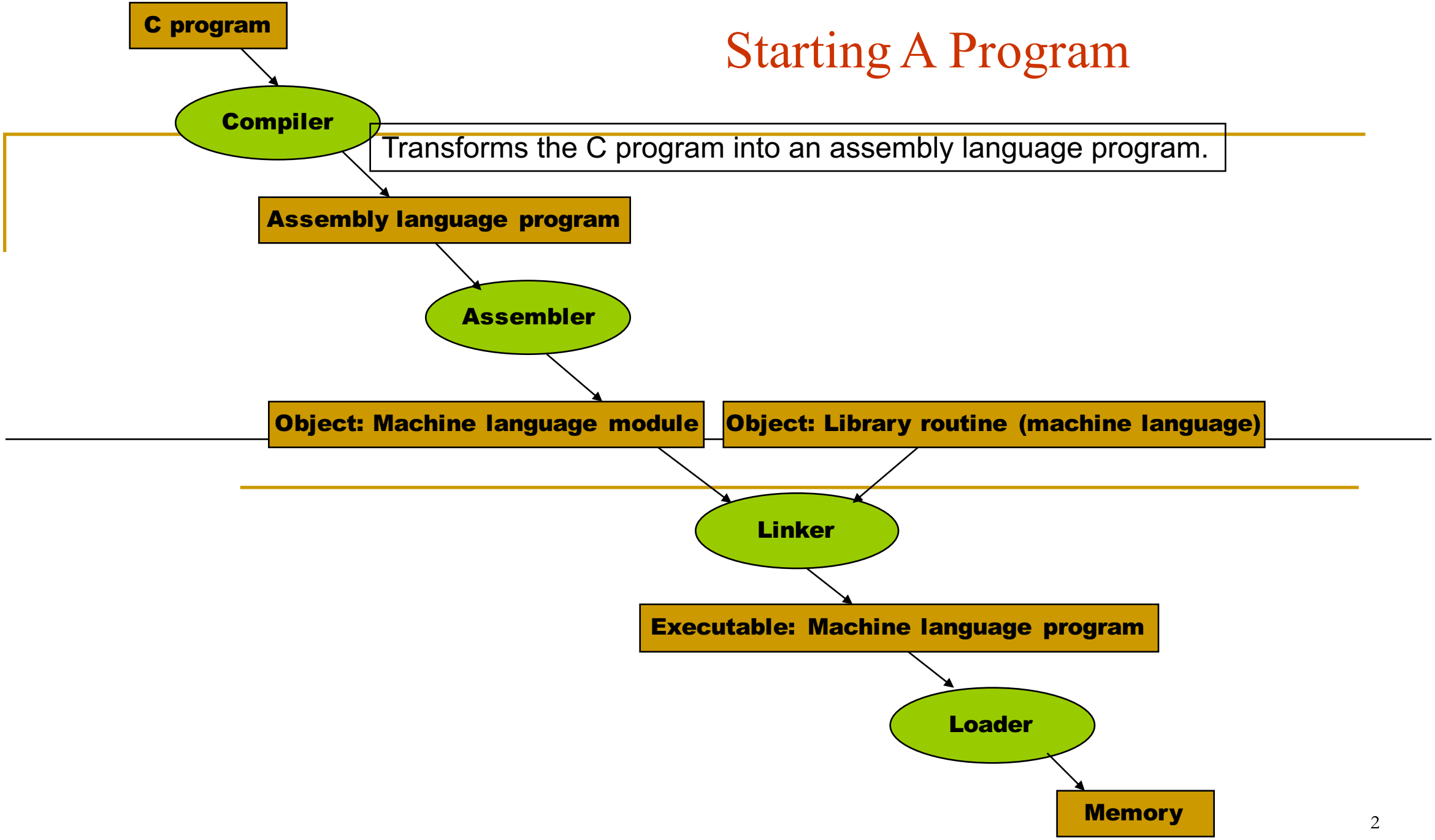
Linker & Loader

Starting A Program



Assembler

- Assembler
  - The assembler turns the assembly language program into an object file.
  - Symbol table: A table that matches names of labels to the addresses of the memory words that instruction occupy.

Object file header			
	~~Name~~	~~Procedure A~~	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	ld x10,0(x3)	
	4	Jal x1, 0	
	…	…	
Data segment	O	(X)	
	…	…	
Relocation information	Address	Instruction Type	Dependency
	0	ld	X
	4	jal	B
Symbol table	Label	Address	
	X	—	
	B	—	

ld x10, x
jal x1, B

Assembler (cont.)

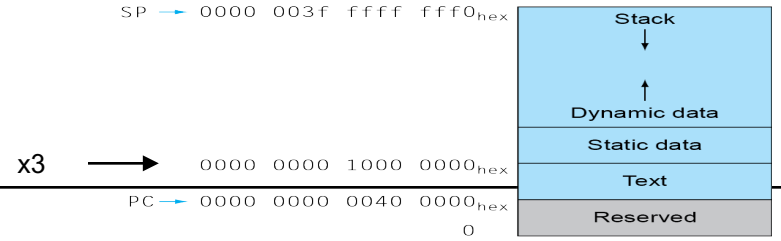
- Psudoinstruction: a common variation of assembly language instructions often treated as if it were an instruction in its own right.
  - li x9, 123 -> addi x9, x0, 123
  - mv x10, x11 -> addi x10, x11, 0

Linker (Link editor)

- Linker takes all the independently assembled machine language programs and “stitches” them together to produce an executable file that can be run on a computer.
- There are three steps for the linker:
  1. Place code and data modules symbolically in memory.
  2. Determine the addresses of data and instruction labels.
  3. Patch both the internal and external references.

Object file header			
	Name	Procedure A	
	Text size	100hex	
	Data size	20hex	
Text segment	Address	Instruction	
	0	ld x10 , 0(x3)	
	4	jal x1,0	
	...	...	
Data segment	O	(X)	
	...	...	
Relocation information	Address	Instruction Type	Dependency
	0	ld	X
	4	jal	B
Symbol table	Label	Address	
	X	—	
	B	—	

Object file header			
	Name	Procedure B	
	Text size	200hex	
	Data size	30hex	
Text segment	Address	Instruction	
	0	sd $x11, 0(x3)	
	4	jal x1, 0	
	...	...	
Data segment	O	(Y)	
	...	...	
Relocation information	Address	Instruction Type	Dependency
	0	sd	Y
	4	jal	A
Symbol table	Label	Address	
	Y	—	
	A	—	



Executable file header		
	Text size	300hex
	Data size	50hex
Text segment	Address	Instruction
	0040 0000hex	ld $x10, 0 (x3)
	0040 0004hex	Jal x1, 252ten
	...	...
	0040 0100hex	sd $x11, 32 (x3)
	0040 0104hex	jal x1, -260ten
	...	...
Data segment	Address	
	1000 0000hex	(X)
	...	...
	1000 0020hex	(Y)
	...	...

Loader

- Read the executables file header to determine the size of the text and data segments
- Creates an address space large enough for the text and data
- Copies the instructions and data from the executable file into memory
- Copies the parameters (if any) to the main program onto the stack
- Initializes the machine registers and sets the stack pointer the first free location
- Jump to a start-up routine

```
main();

_start_up:
  lw   a0, offset($sp)   ## load arguments
  jal   main;
  exit
```

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 x10, … and calls main
     - When main returns, do exit syscall

```
main();

_start_up:
  lw   x10, offset($sp)   ## load arguments
  jal  x1, main;
  exit
```

Dynamically Linked Libraries (DLL)

- Disadvantages with traditional statically linked library
  - Library updates
  - Loading the whole library even if all of the library is not used
- Dynamically linked library
  - The libraries are not linked and loaded until the program is run.
  - Lazy procedure linkage
    - Each routine is linked only after it is called.

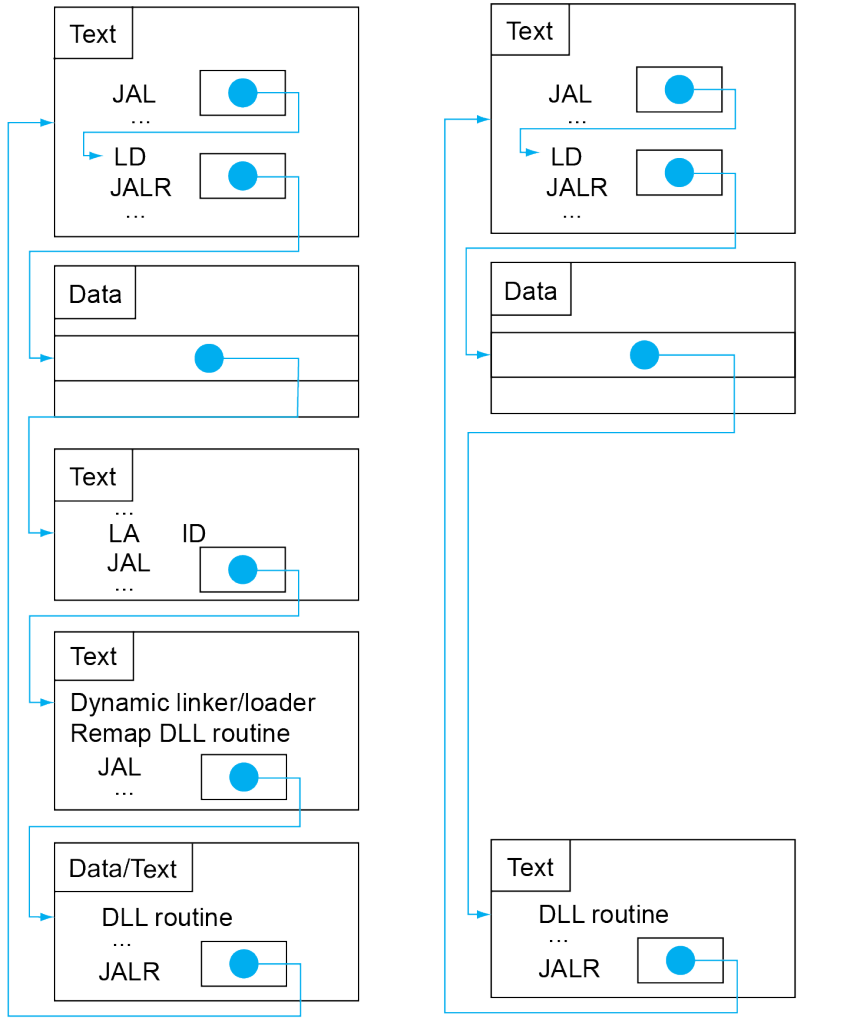
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

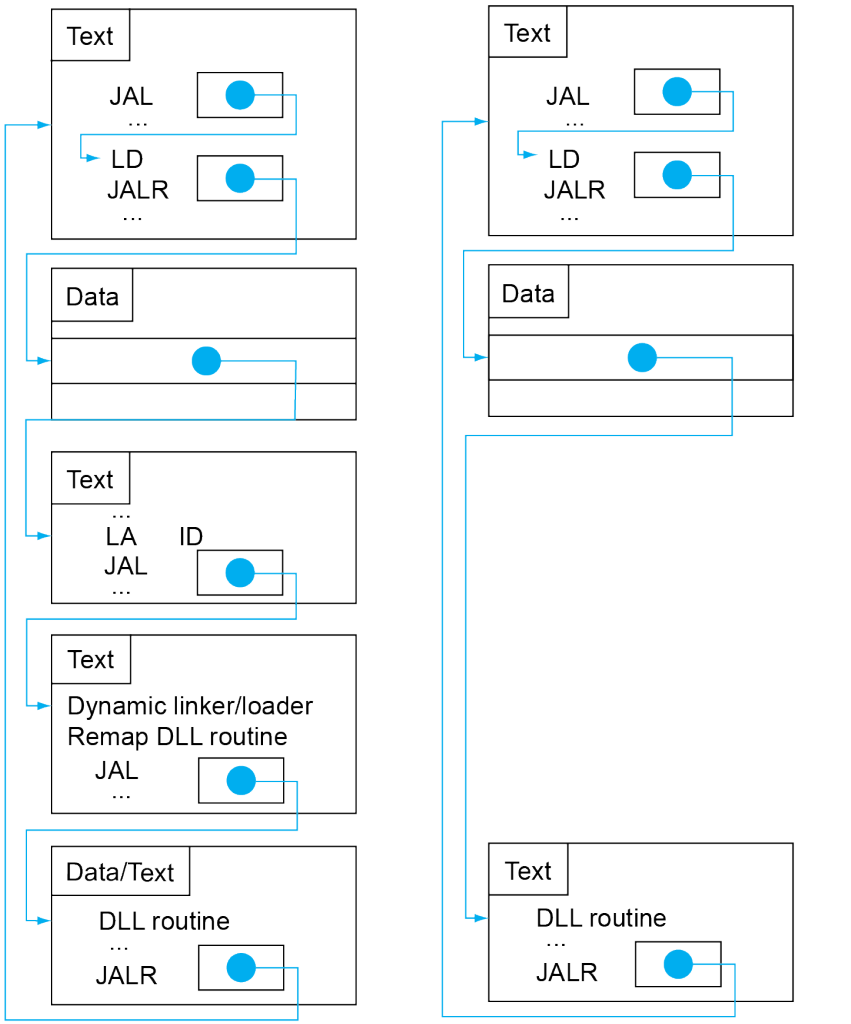
Linker/loader code

Dynamically
mapped code



(a) First call to DLL routine

(b) Subsequent calls to DLL routine



(a) First call to DLL routine (b) Subsequent calls to DLL routine

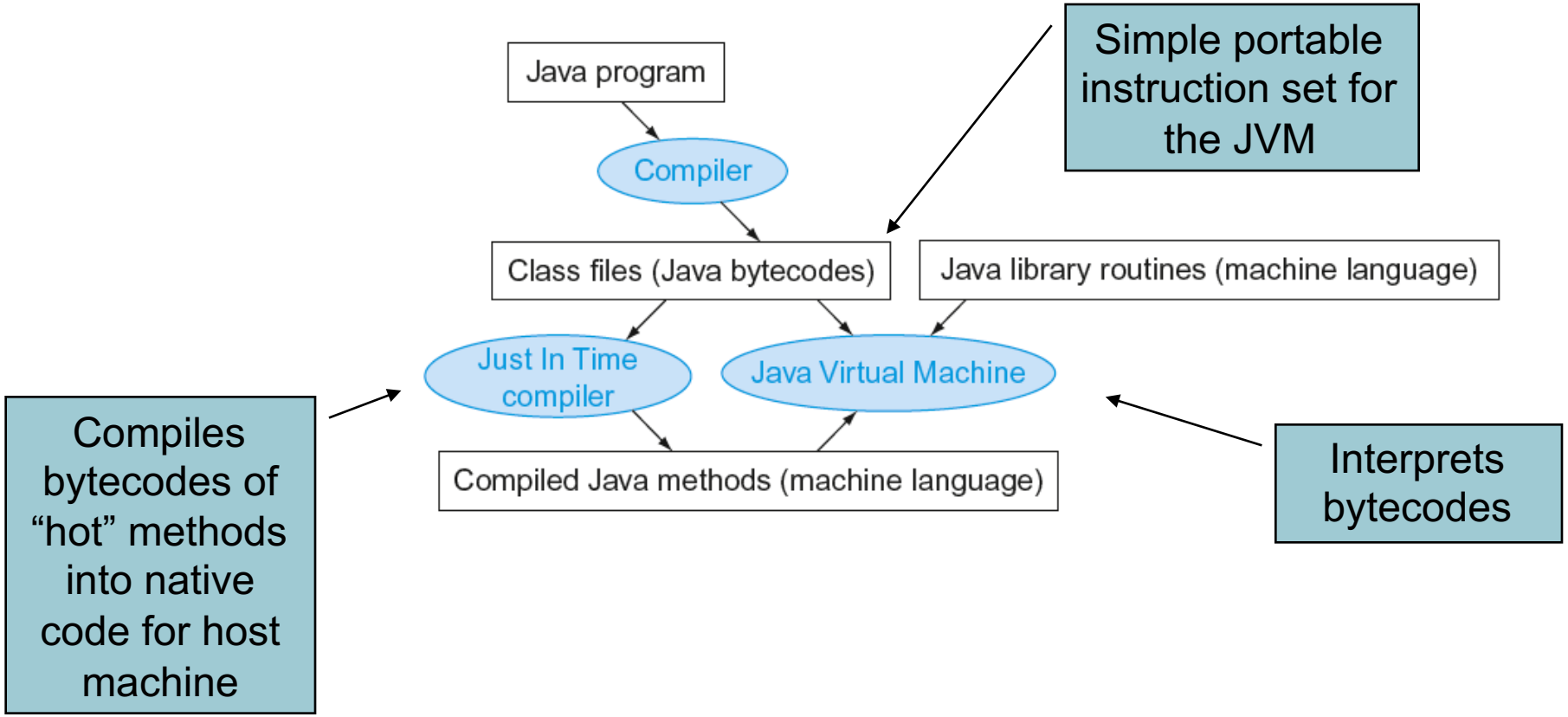
Text
jal printf();
printf ()
{ ID x5, printf_addr
JALR x0, 0(x5)}

Data
printf_addr .word L1 → Data
printf_addr 0x400000

Text
L1: LA ID
JAL x1, DLL;

Text
printf()
:::::::
JALR x0, 0(x1) → Assume this is loaded into 0x400000

Starting Java Applications

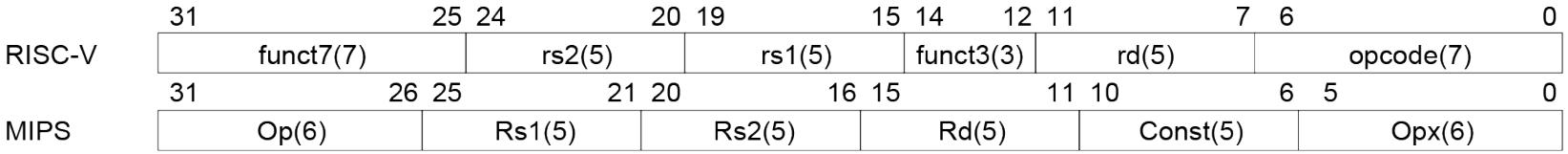


MIPS Instructions

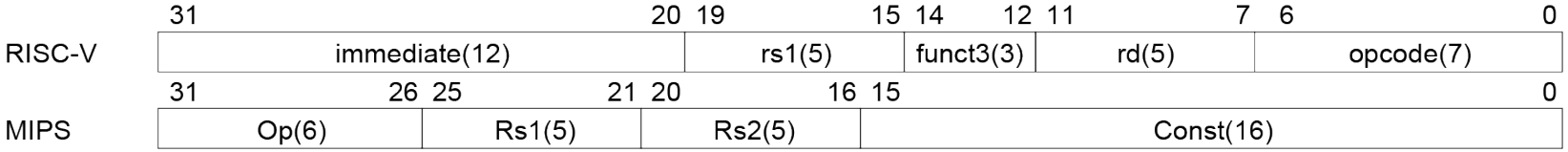
- MIPS: commercial predecessor to RISC-V
- Similar basic set of instructions
  - 32-bit instructions
  - 32 general purpose registers, register 0 is always 0
  - 32 floating-point registers
  - Memory accessed only by load/store instructions
    - Consistent use of addressing modes for all data sizes
- Different conditional branches
  - For <, <=, >, >=
  - RISC-V: blt, bge, bltu, bgeu
  - MIPS: slt, sltu (set less than, result is 0 or 1)
    - Then use beq, bne to complete the branch

Instruction Encoding

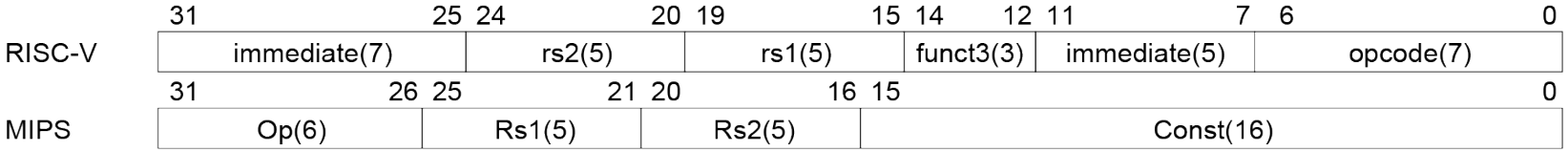
Register-register



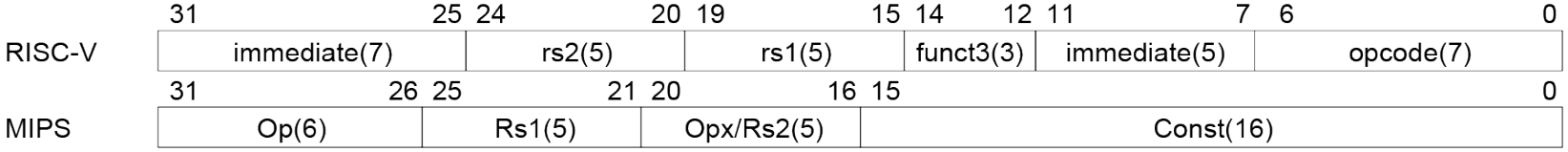
Load



Store



Branch



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

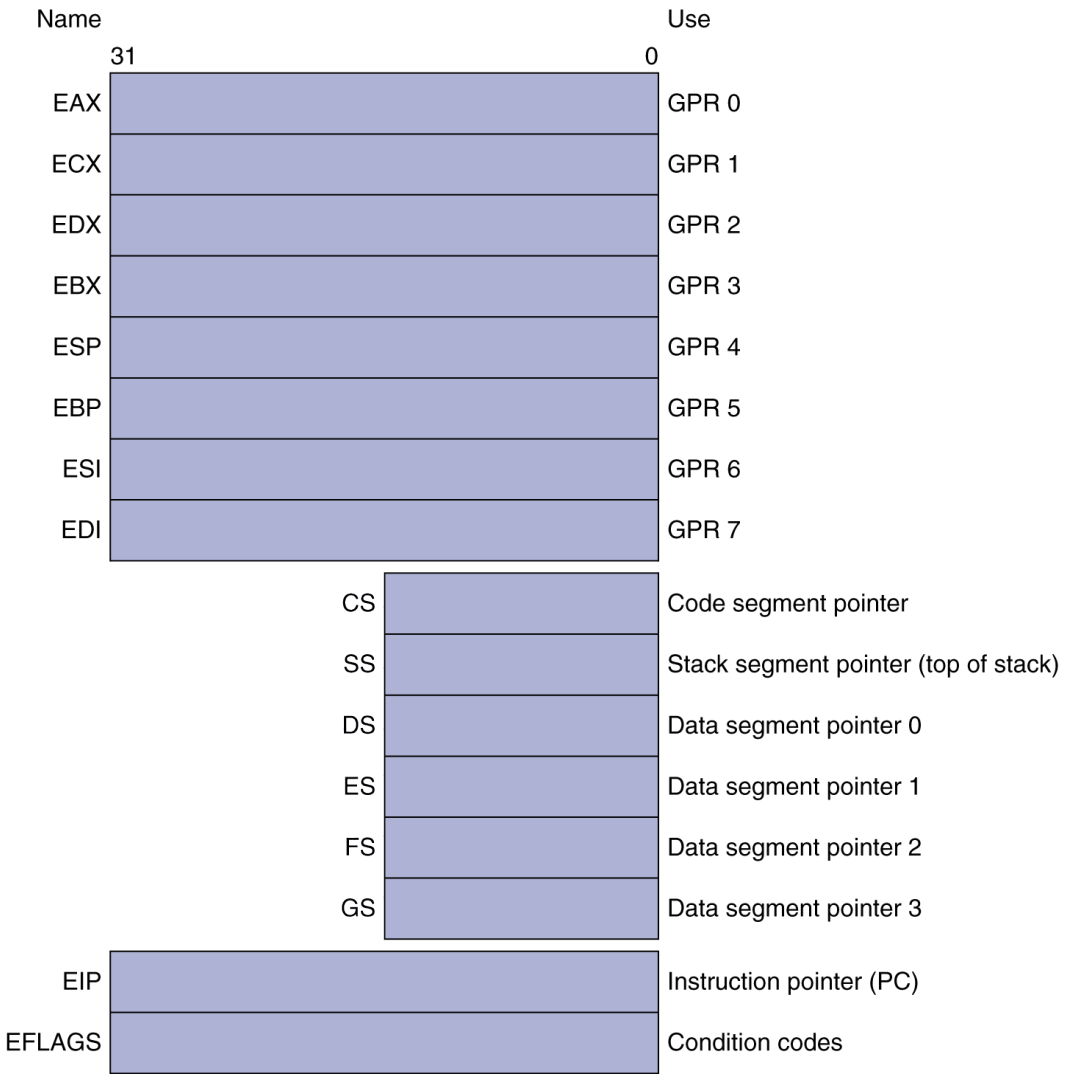
The Intel x86 ISA

- 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 ~ 57 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 ~ 70 instructions
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions ~ 144 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



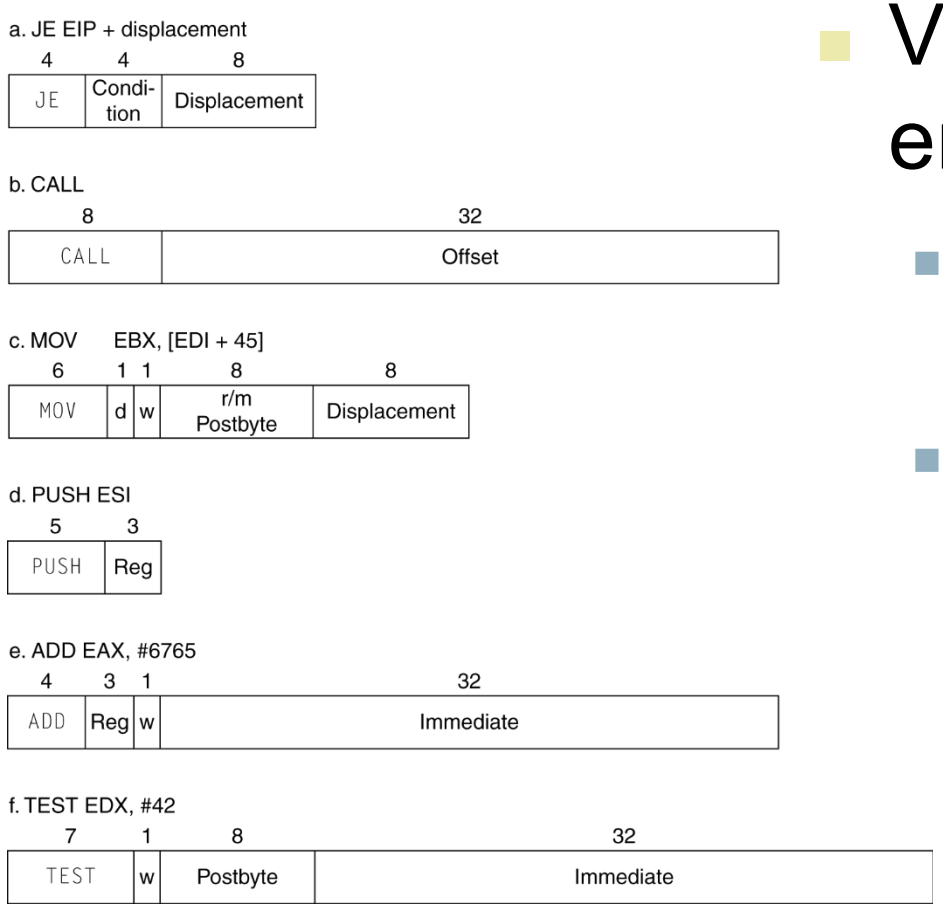
Basic x86 Addressing Modes

- 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



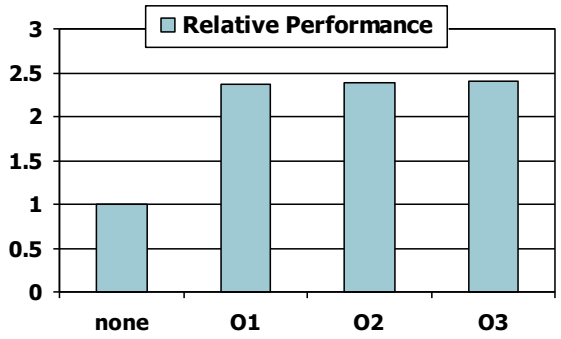
- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

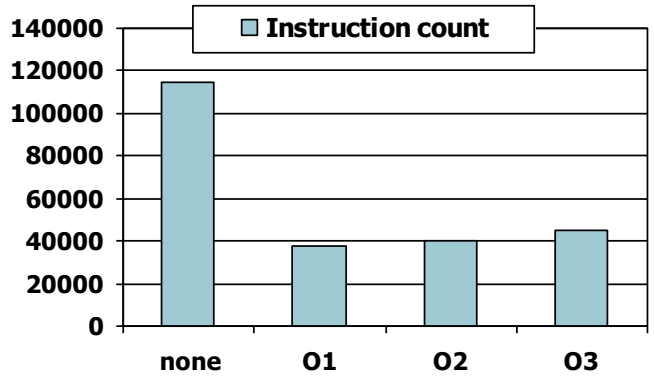
Implementing IA-32

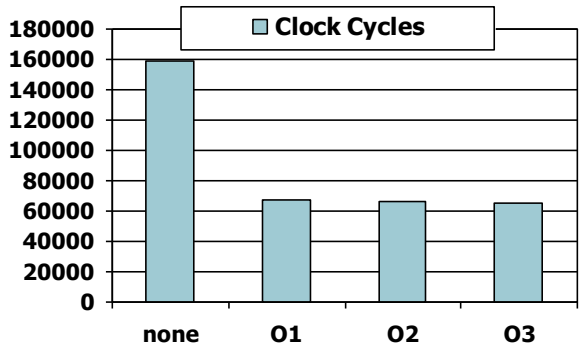
- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
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

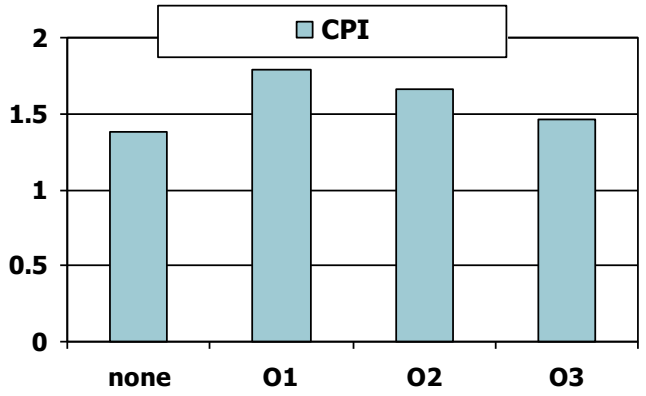
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

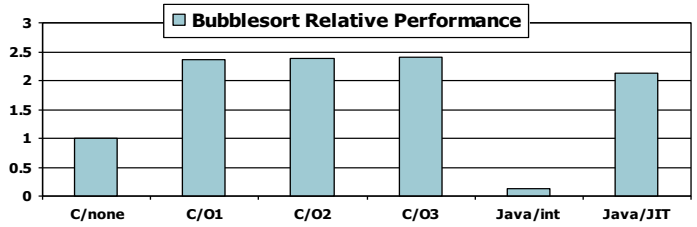


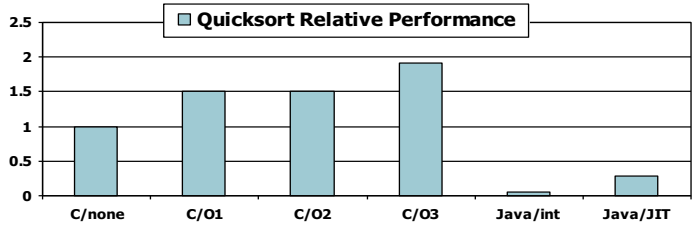


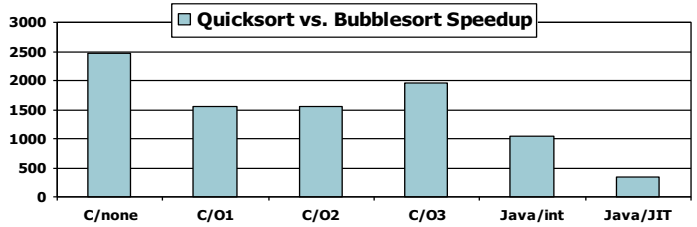




Effect of Language and Algorithm





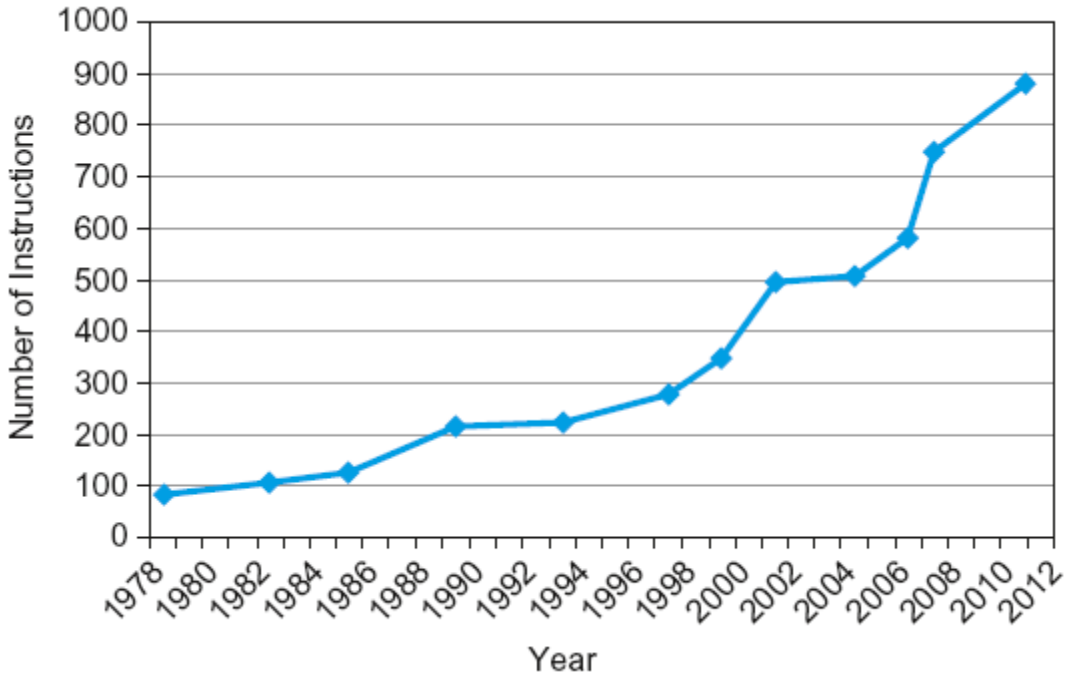


Fallacies

- 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 ⇒ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set