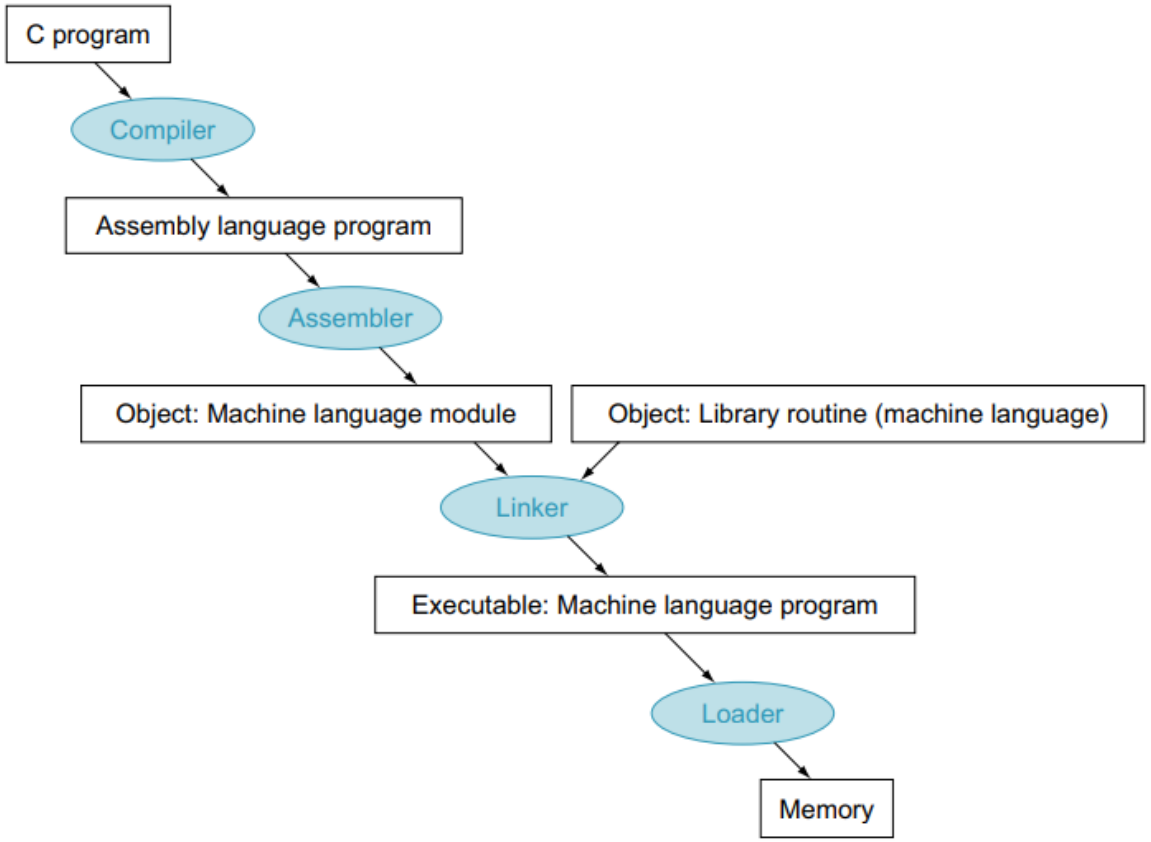
C Program

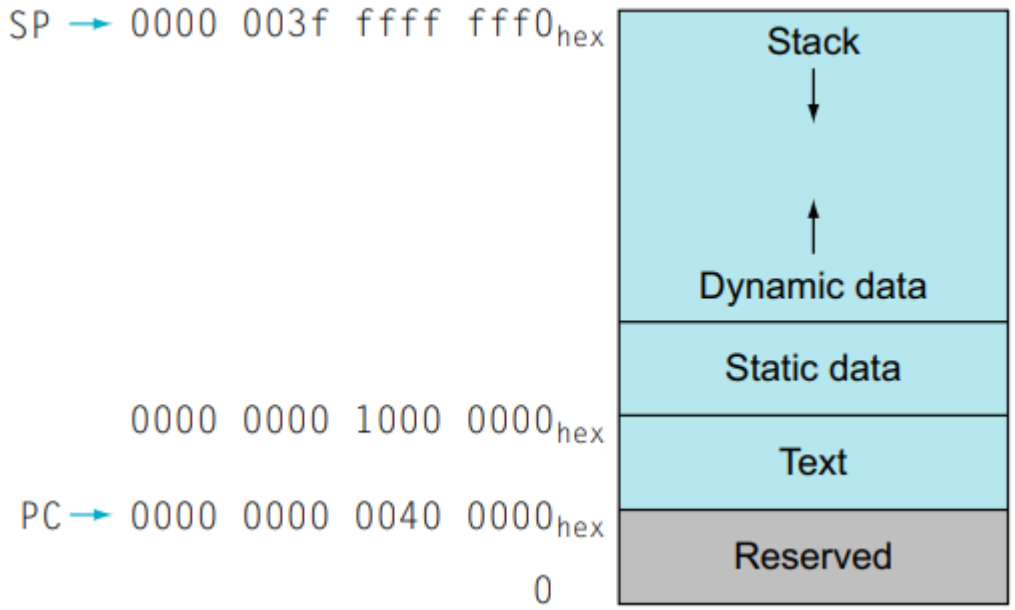
Translating and Starting a Program

- Four steps in transforming a C program into a file in nonvolatile storage (disk or flash memory) into a program running on a computer.
- Some systems combine these steps to reduce translation time, but programs go through these four logical phases.

A translation hierarchy for C



The RISC-V memory allocation for program and data



Compiler

- The compiler transforms the C program into an assembly language program, a symbolic form of what the machine understands
- assembly language A symbolic language that can be translated into binary machine language

Assembler

- Since assembly language is an interface to higher-level software, the assembler can also treat common variations of machine language instructions as if they were instructions in their own right
- The hardware need not implement these instructions; however, their appearance in assembly language simplifies translation and programming. Such instructions are called pseudoinstructions
- pseudoinstruction A common variation of assembly language instructions often treated as if it were an instruction in its own right.

- RISC-V hardware makes sure that register x0 always has the value 0
- Register x0 is used to create the assembly language instruction that copies the contents of one register to another
- li x9, 123 // load immediate value 123 into register x9 is accepted even though it is not found in the RISC-V machine language
- It is converted to
- addi x9, x0, 123 // register x9 gets register x0 + 123

- The RISC-V assembler also converts mv (move) into an addi instruction.
- mv x10, x11 // register x10 gets register x11 becomes

addi x10, x11, 0 // register x10 gets register x11 + 0

- The assembler accepts j Label to unconditionally branch to a label, as a stand-in for jal x0, Label
- It also converts branches to faraway locations into a branch and a jump
- Li allows larger immediates to be handled compared to addi
- the load address (la) macro works similarly for symbolic addresses

- RISC-V assembler does not require the programmer to specify the immediate version of the instruction when using a constant for arithmetic and logical instructions; it just generates the proper opcode
- and x9, x10, 15 // register x9 gets x10 AND 15

becomes

- andi x9, x10, 15 // register x9 gets x10 AND 15
- “i” on the instructions are used to remind the reader that andi produces a different opcode in a different instruction format than the and instruction with no immediate operands.

- pseudoinstructions give RISC-V a richer set of assembly language instructions than those implemented by the hardware

- The assembler turns the assembly language program into an object file, which is a combination of machine language instructions, data, and information needed to place instructions properly in memory
- To produce the binary version of each instruction in the assembly language program, the assembler must determine the addresses corresponding to all labels.
- Assemblers keep track of labels used in branches and data transfer instructions in a symbol table
- symbol table A table that matches names of labels to the addresses of the memory words that instructions occupy

The object file for UNIX systems typically contains six distinct pieces:

- The object file header describes the size and position of the other pieces of the object file.
- The text segment contains the machine language code
- The static data segment contains data allocated for the life of the program. (UNIX allows programs to use both static data, which is allocated throughout the program, and dynamic data, which can grow or shrink as needed by the program

- The relocation information identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.
- The symbol table contains the remaining labels that are not defined, such as external references
- The debugging information contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable

Linker

- linker Also called link editor. A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.
- Compilers were compiling all procedures for any change
- An alternative is to compile and assemble each procedure independently, so that a change to one line would require compiling and assembling only one procedure

- 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

The linker uses the relocation information and symbol table in each object module to resolve all undefined labels. Such references occur in branch instructions and data addresses, so the job of this program is much like that of an editor: it finds the old addresses and replaces them with the new addresses. Editing is the origin of the name “link editor,” or linker for short

- The linker produces an executable file that can be run on a computer
- executable file A functional program in the format of an object file that contains no unresolved references. It can contain symbol tables and debugging information. A “stripped executable” does not contain that information. Relocation information may be included for the loader

Linking Object Files

- Link the two object files below. Show updated addresses of the first few instructions of the completed executable file. We show the instructions in assembly language just to make the example understandable; in reality, the instructions would be numbers

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw x10, 0(x3)`	
	4	`jal x1, 0`	
	. . .	. . .	
Data segment	0	(X)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw x11, 0(x3)`	
	4	`jal x1, 0`	
	. . .	. . .	
Data segment	0	(Y)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0000\ 0000\ 0040\ 0000_{hex}$	`lw x10, 0(x3)`
	$0000\ 0000\ 0040\ 0004_{hex}$	`jal x1,` 252_{ten}
	. . .	. . .
	$0000\ 0000\ 0040\ 0100_{hex}$	`sw x11, 32(x3)`
	$0000\ 0000\ 0040\ 0104_{hex}$	`jal x1,` -260_{ten}
	. . .	. . .
Data segment	Address	
	$0000\ 0000\ 1000\ 0000_{hex}$	(X)
	. . .	. . .
	$0000\ 0000\ 1000\ 0020_{hex}$	(Y)
	. . .	. . .

Loader

- loader A systems program that places an object program in main memory so that it is ready to execute.
- The loader follows these steps in UNIX systems:

1. Reads the executable file header to determine size of the text and data segments.
2. Creates an address space large enough for the text and data.
3. Copies the instructions and data from the executable file into memory

4. Copies the parameters (if any) to the main program onto the stack.

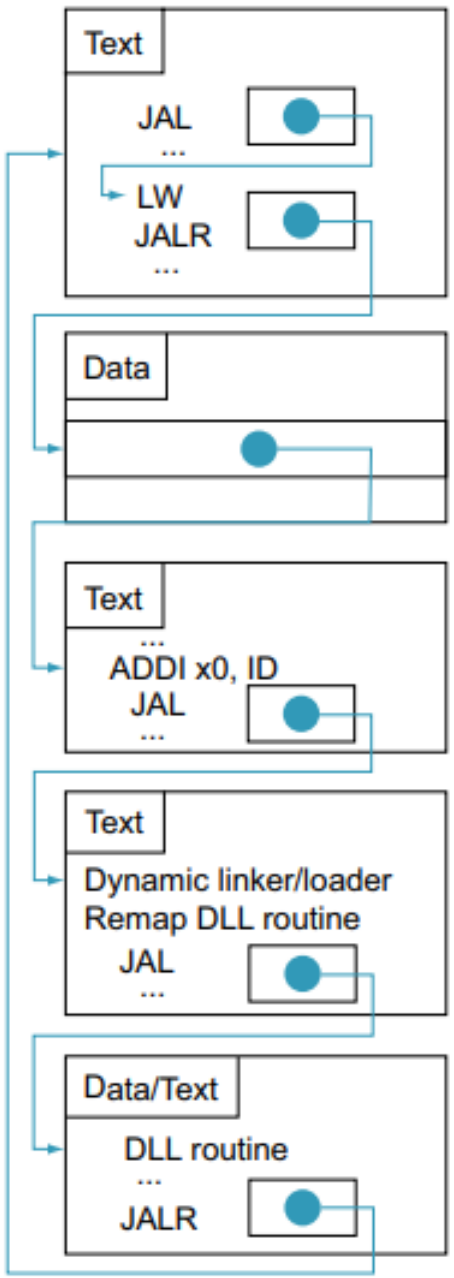
5. Initializes the processor registers and sets the stack pointer to the first free location.

6. Branches to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an exit system call

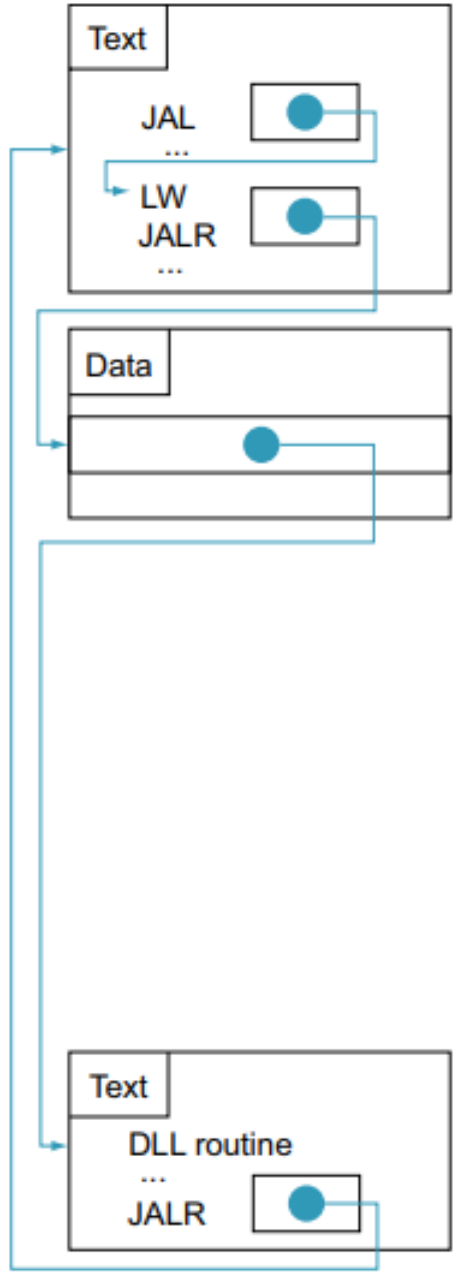
Dynamically linked libraries

- Disadvantages with static
- The library routines become part of the executable code. If a new version of the library is released that fixes bugs or supports new hardware devices, the statically linked program keeps using the old version.
- It loads all routines in the library that are called anywhere in the executable, even if those calls are not executed. The library can be large relative to the program; for example, the standard C library on a RISC-V system running the Linux operating system is 1.5MiB

- dynamically linked libraries (DLLs) Library routines that are linked to a program during execution
- Both the program and library routines keep extra information on the location of nonlocal procedures and their names
- The downside of the initial version of DLLs was that it still linked all routines of the library that might be called, versus just those that are called during the running of the program



(a) First call to DLL routine



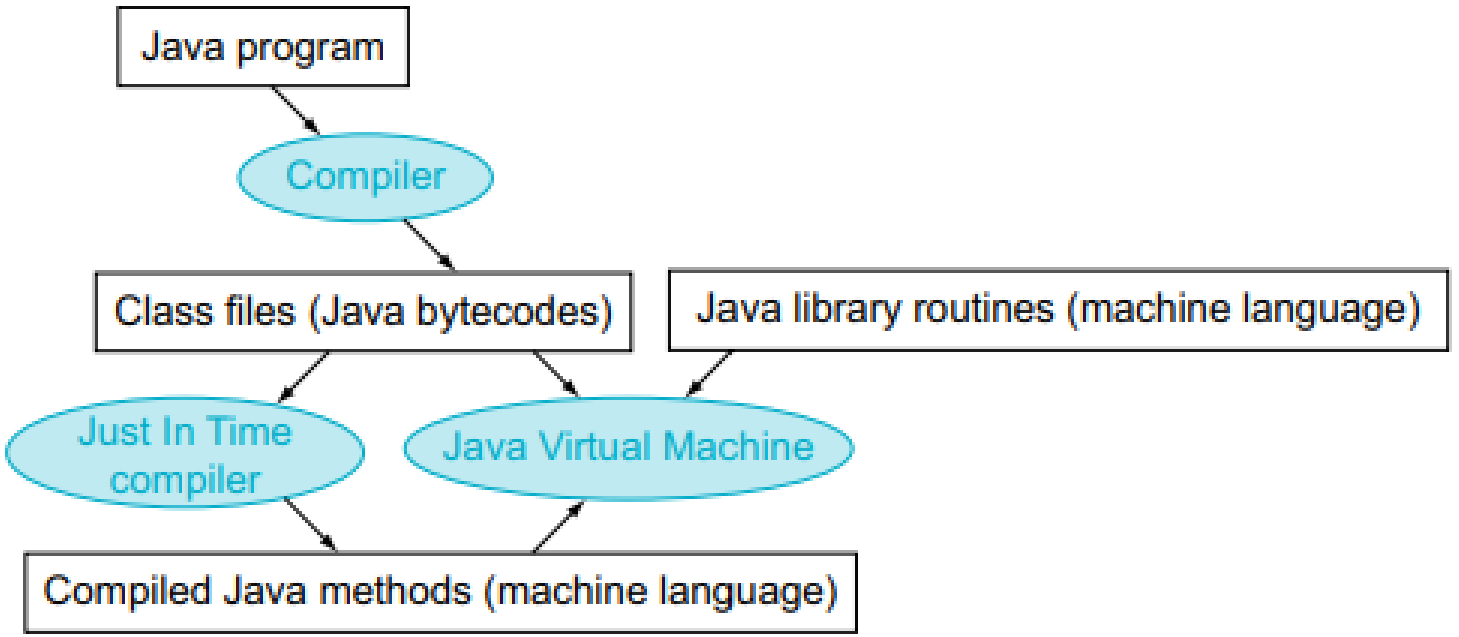
(b) Subsequent calls to DLL routine

- The first time the library routine is called, the program calls the dummy entry and follows the indirect branch.
- It points to code that puts a number in a register to identify the desired library routine and then branches to the dynamic linker/ loader.
- The linker/loader finds the wanted routine, remaps it, and changes the address in the indirect branch location to point to that routine.
- It then branches to it. When the routine completes, it returns to the original calling site. Thereafter, the call to the library routine branches indirectly to the routine without the extra hops

- DLLs require additional space for the information needed for dynamic linking, but do not require that whole libraries be copied or linked
- Microsoft’s Windows relies extensively on dynamically linked libraries

Starting a Java Program

- traditional model of executing a program, where the emphasis is on fast execution time for a program targeted to a specific instruction set architecture, or even a particular implementation of that architecture Ex:C
- Java program run safely on any computer, even if it might slow execution time.


Java program
Compiler
Class files (Java bytecodes)
Java library routines (machine language)
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)

- Java bytecode Instruction from an instruction set designed to interpret Java programs
- Virtually no optimizations are performed
- A software interpreter, called a Java Virtual Machine (JVM), can execute Java bytecodes. An interpreter is a program that simulates an instruction set architecture
- Java virtual machines are found in billions of devices, in everything from cell phones to Internet browsers

- The downside of interpretation is lower performance
- To preserve portability and improve execution speed, the next phase of Java's development was compilers that translated while the program was running. Such Just In Time compilers (JIT)
- Just In Time compiler (JIT) The name commonly given to a compiler that operates at runtime, translating the interpreted code segments into the native code of the computer

A C Sort Example to Put it All Together

- A C procedure that swaps two locations in memory

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

RISC-V convention on parameter passing is to use registers x10 to x17. Since swap has just two parameters, v and k, they will be found in registers x10 and x11. The only other variable is temp, which we associate with register x5 since swap is a leaf procedure

```
swap:
    slli    x6, x11, 2    // reg x6 = k * 4
    add     x6, x10, x6   // reg x6 = v + (k * 4)
    lw      x5, 0(x6)     // reg x5 (temp) = v[k]
    lw      x7, 4(x6)     // reg x7 = v[k + 1]
    sw      x7, 0(x6)     // v[k] = reg x7
    sw      x5, 4(x6)     // v[k+1] = reg x5 (temp)
    jalr    x0, 0(x1)     // return to calling routine
```

The Procedure sort

- sorting an array of integers, using bubble or exchange sort, which is one of the simplest if not the fastest sorts.

```
void sort (int v[], size_t int n)
{
    size_t 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);
        }
    }
```

Saving registers

```
sort:   addi sp, sp, -20        # make room on stack for 5 registers
        sw x1, 16(sp)           # save return address on stack
        sw x22, 12(sp)          # save x22 on stack
        sw x21, 8(sp)           # save x21 on stack
        sw x20, 4(sp)           # save x20 on stack
        sw x19, 0(sp)           # save x19 on stack
```

Procedure body		
Move parameters	`addi x21, x10, 0`	`# copy parameter x10 into x21`
	`addi x22, x11, 0`	`# copy parameter x11 into x22`
Outer loop	`addi x19,x0, 0`	`# i = 0`
	`for1tst:bge x19, x22, exit1`	`# go to exit1 if i >= n`
Inner loop	`addi x20, x19, -1`	`# j = i - 1`
	`for2tst:blt x20, x0, exit2`	`# go to exit2 if j < 0`
	`slli x5, x20, 2`	`# x5 = j * 4`
	`add x5, x21, x5`	`# x5 = v + (j * 4`
	`lw x6, 0(x5)`	`# x6 = v[j]`
	`lw x7, 4(x5)`	`# x7 = v[j + 1]`
	`ble x6, x7, exit2`	`# go to exit2 if x6 < x7`
Pass parameters and call	`addi x10, x21, 0`	`# first swap parameter is v`
	`addi x11, x20, 0`	`# second swap parameter is j`
	`jal x1, swap`	`# call swap`
Inner loop	`addi x20, x20, -1`	`j for2tst`
	`jal, x0 for2tst`	`# go to for2tst`
Outer loop	`exit2: addi x19, x19, 1`	`# i += 1`
	`jal, x0 for1tst`	`# go to for1tst`

Restoring registers

```
exit1:  lw x19, 0(sp)           # restore x19 from stack
        lw x20, 4(sp)           # restore x20 from stack
        lw x21, 8(sp)           # restore x21 from stack
        lw x22, 12(sp)          # restore x22 from stack
        lw x1, 16(sp)           # restore return address from stack
        addi sp, sp, 20         # restore stack pointer
```

Procedure return

```
        jalr x0, 0(x1)          # return to calling routine
```

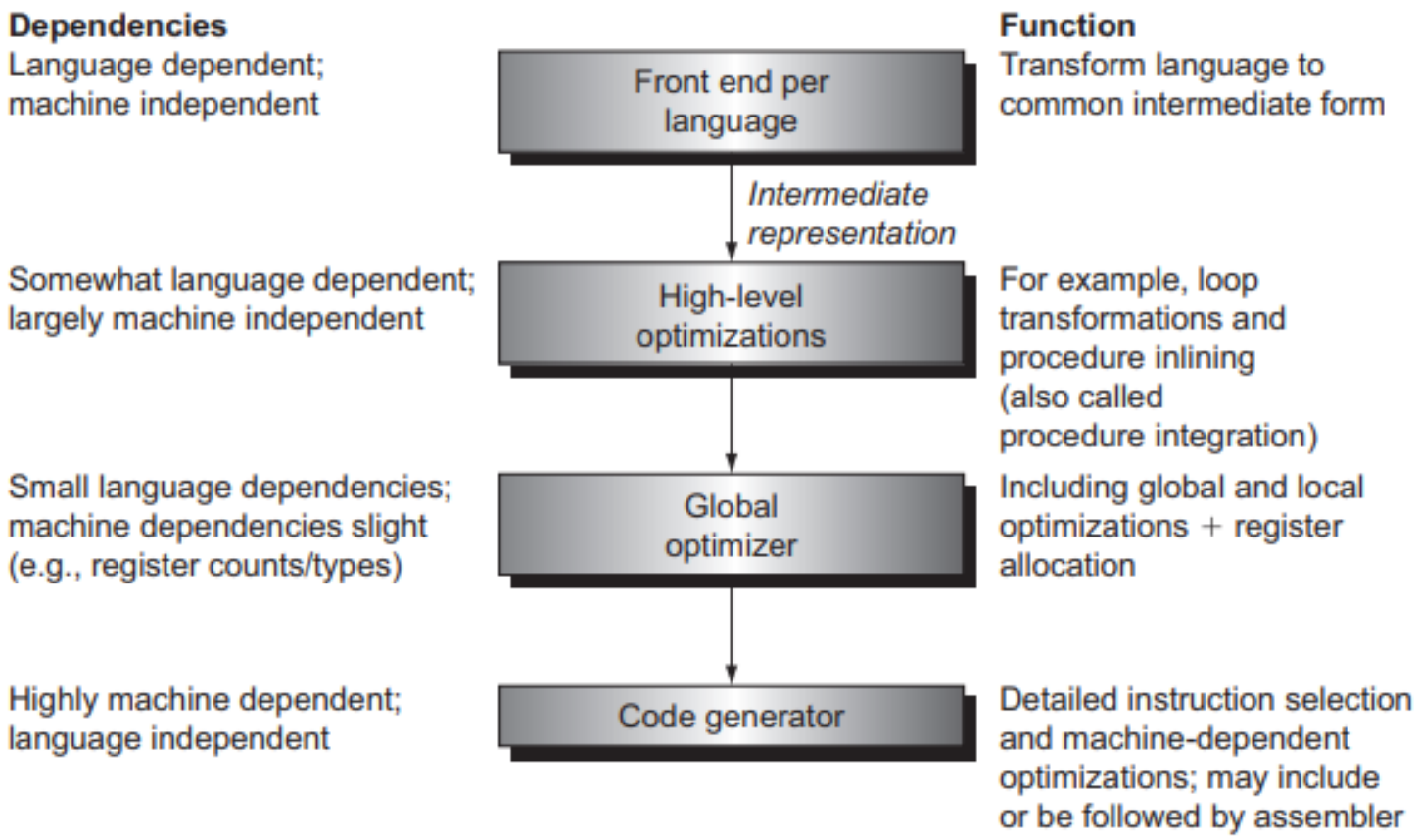
Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort.

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (procedure integration)	2.41	65,747	44,993	1.46

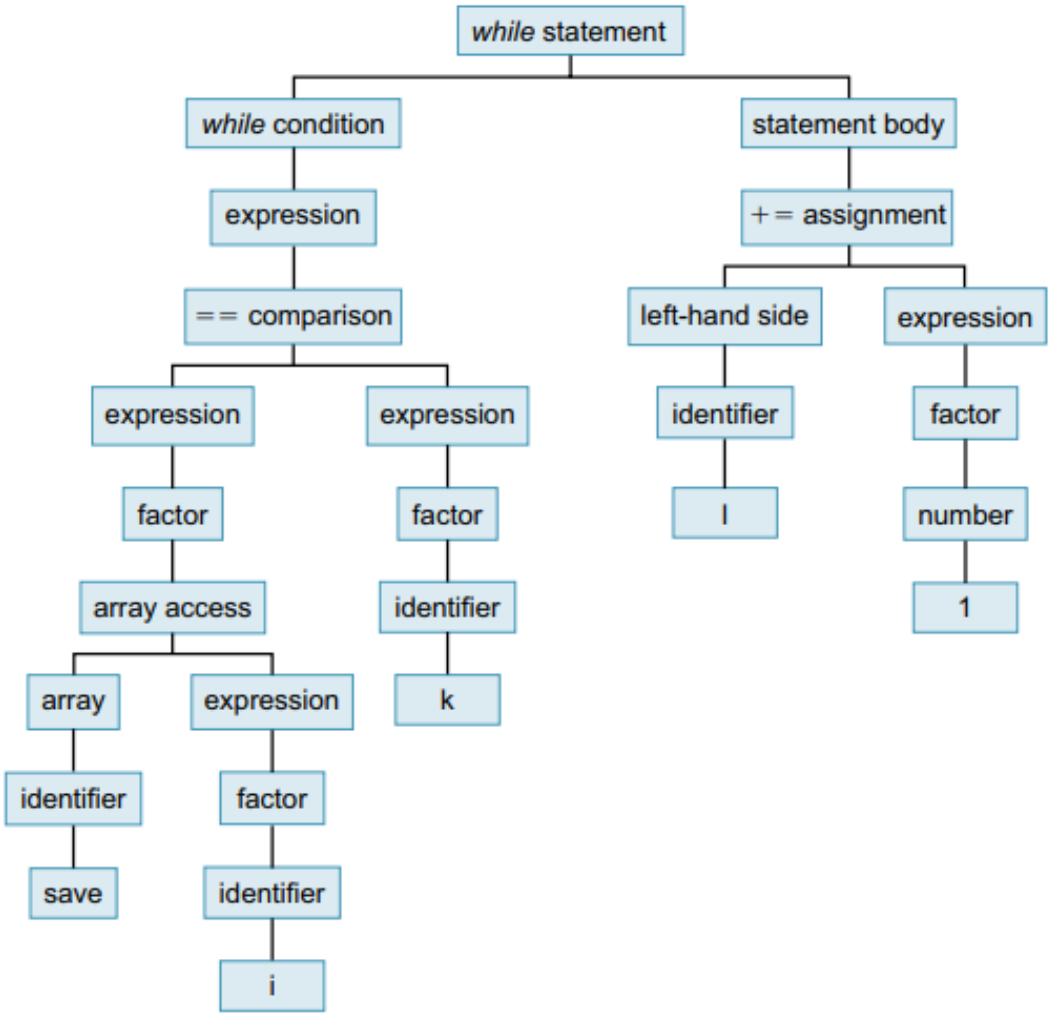
Performance of two sort algorithms in C and Java using interpretation and optimizing compilers relative to unoptimized C version

Language	Execution method	Optimization	Bubble Sort relative performance	Quicksort relative performance	Speedup Quicksort vs. Bubble Sort
C	Compiler	None	1.00	1.00	2468
	Compiler	O1	2.37	1.50	1562
	Compiler	O2	2.38	1.50	1555
	Compiler	O3	2.41	1.91	1955
Java	Interpreter	–	0.12	0.05	1050
	JIT compiler	–	2.13	0.29	338

Compiling C


Dependencies
Language dependent; machine independent
Front end per language
Function
Transform language to common intermediate form
Intermediate representation
Somewhat language dependent; largely machine independent
High-level optimizations
For example, loop transformations and procedure inlining (also called procedure integration)
Small language dependencies; machine dependencies slight (e.g., register counts/types)
Global optimizer
Including global and local optimizations + register allocation
Highly machine dependent; language independent
Code generator
Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler

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



High-Level Optimizations

- High-level optimizations are transformations that are done at something close to the source level
- The most common high-level transformation is probably procedure inlining, which replaces a call to a function by the body of the function, substituting the caller’s arguments for the procedure’s parameters.
- Other high-level optimizations involve loop transformations that can reduce loop overhead, improve memory access, and exploit the hardware more effectively

- loop-unrolling A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

The while loop example is shown using a typical intermediate representation.

```
loop:
  # comments are written like this--source code often included
  # while (save[i] == k)
  addi r100, x0, save          # r100 = &save[0]
  lw   r101, i
  addi r102, x0, 4
  mul  r103, r101, r102
  add  r104, r103, r100
  lw   r105, 0(r104)           # r105 = save[i]
  lw   r106, k
  bne  r105, r106, exit
  # i += 1
  lw   r106, i
  addi r107, r106, i           # increment
  sw   r107, i
  jal x0 loop                  # next iteration
exit:
```

Local and Global Optimizations

- Within the pass dedicated to local and global optimization, three classes of optimization are performed

1 Local optimization works within a single basic block. A local optimization pass is often run as a precursor and successor to global optimization to “clean up” the code before and after global optimization.

2. Global optimization works across multiple basic blocks; we will see an example of this shortly.

3. Global register allocation allocates variables to registers for regions of the code. Register allocation is crucial to getting good performance in modern processors

- Several optimizations are performed both locally and globally, including common subexpression elimination, constant propagation, copy propagation, dead store elimination, and strength reduction.

- Common subexpression elimination finds multiple instances of the same expression and replaces the second one by a reference to the first. Consider, for example, a code segment to add 4 to an array element
- x[i] = x[i] + 4

Unoptimized vs optimized code

```
// x[i] + 4
addi r100, x0, x
lw r101,i
mul r102,r101,4
add r103,r100,r102
lw r104, 0(r103)
//
addi r105, r104,4
addi r106, x0, x
lw r107,i
mul r108,r107,4
add r109,r106,r107
sw r105,0(r109)
```

```
// x[i] + 4
addi r100, x0, x
lw r101,i
slli r102,r101,2
add r103,r100,r102
lw r104, 0(r103)
// value of x[i] is in r104
addi r105, r104,4
sw r105, 0(r103)
```

other optimizations:

- Strength reduction replaces complex operations by simpler ones and can be applied to this code segment, replacing the mul by a shift left.
- Constant propagation and its sibling constant folding find constants in code and propagate them, collapsing constant values whenever possible.
- Copy propagation propagates values that are simple copies, eliminating the need to reload values and possibly enabling other optimizations, such as common subexpression elimination.

- Dead store elimination finds stores to values that are not used again and eliminates the store; its “cousin” is dead code elimination, which finds unused code—code that cannot affect the result of the program—and eliminates it. With the heavy use of macros, templates, and the similar techniques designed to reuse code in high-level languages, dead code occurs surprisingly often.

- Compilers must be conservative. The first task of a compiler is to produce correct code; its second task is usually to produce fast code, although other factors, such as code size, may sometimes be important as well.
- Code that is fast but incorrect—for any possible combination of inputs—is simply wrong. Thus, when we say a compiler is “conservative,” we mean that it performs an optimization only if it knows with 100% certainty that, no matter what the inputs, the code will perform as the user wrote it.
- Since most compilers translate and optimize one function or procedure at a time, most compilers, especially at lower optimization levels, assume the worst about function calls and about their own parameters

Global Code Optimizations

- There are two other important global optimizations: code motion and induction variable elimination other than the ones used in local optimizations
- Code motion finds code that is loop invariant: a particular piece of code computes the same value on every iteration of the loop and, hence, may be computed once outside the loop.
- Induction variable elimination is a combination of transformations that reduce overhead on indexing arrays, essentially replacing array indexing with pointer accesses

Implementing Local Optimizations

- Local optimizations are implemented on basic blocks by scanning the basic block in instruction execution order, looking for optimization opportunities.

Refer to slide 46

1. Determine that the two addi operations return the same result by observing that the operand x is the same and that the value of its address has not been changed between the two addi operations.
2. Replace all uses of R106 in the basic block by R101.
3. Observe that i cannot change between the two lw instructions that reference it. So replace all uses of R107 with R101.

4. Observe that the mul instructions now have the same input operands, so that R108 may be replaced by R102.

5. Observe that now the two add instructions have identical input operands (R100 and R102), so replace the R109 with R103.

6. Use dead store code elimination to delete the second set of addi,lw, mul, and add instructions since their results are unused.

Implementing Global Optimizations

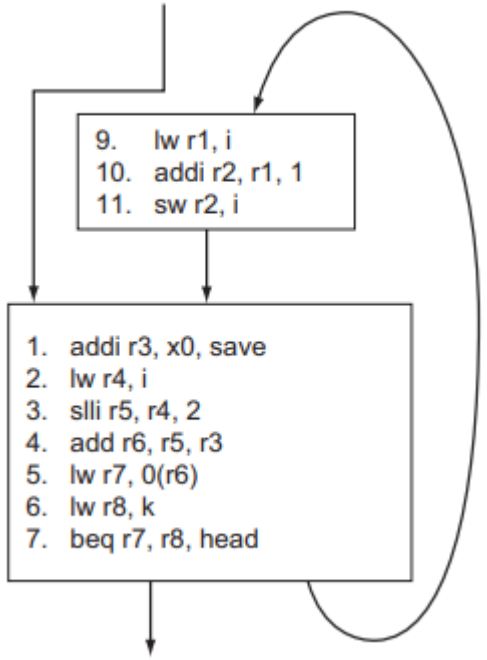
- To understand the challenge of implementing global optimizations, let’s consider a few examples
- Consider the case of an opportunity for common subexpression elimination, say, of an IR statement like add Rx, R20, R50. To determine whether two such statements compute the same value, we must determine whether the values of R20 and R50 are identical in the two statements. In practice, this means that the values of R20 and R50 have not changed between the first statement and the second. For a single basic block, this is easy to decide; it is more difficult for a more complex program structure involving multiple basic blocks and branches.

- Consider the second lw of i into R107 within the earlier example: how do we know whether its value is used again? If we consider only a single basic block, and we know that all uses of R107 are within that block, it is easy to see. As optimization proceeds, however, common subexpression elimination and copy propagation may create other uses of a value. Determining that a value is unused and the code is dead is more difficult in the case of multiple basic blocks.

- Finally, consider the load of k in our loop, which is a candidate for code motion. In this simple example, we might argue that it is easy to see that k is not changed in the loop and is, hence, loop invariant. Imagine, however, a more complex loop with multiple nestings and if statements within the body. Determining that the load of k is loop invariant is harder in such a case.

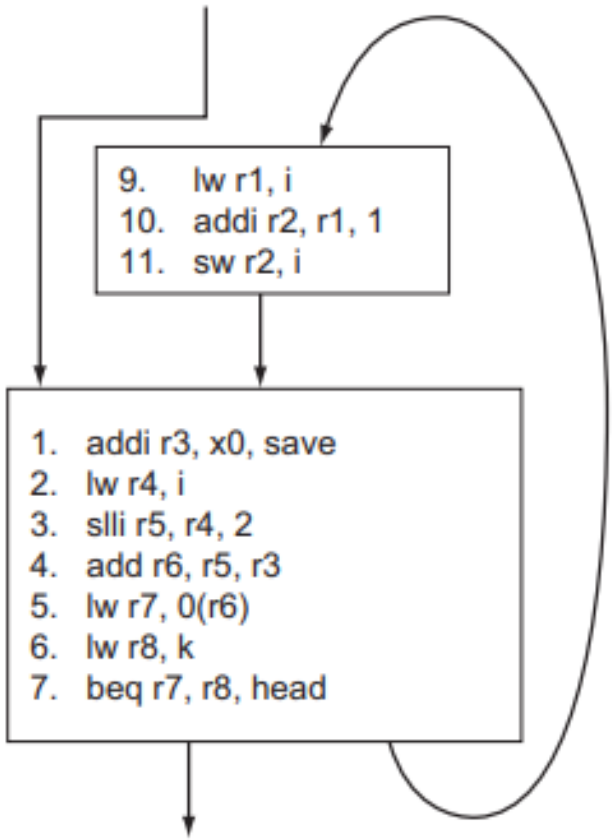
A control flow graph for the while loop example.

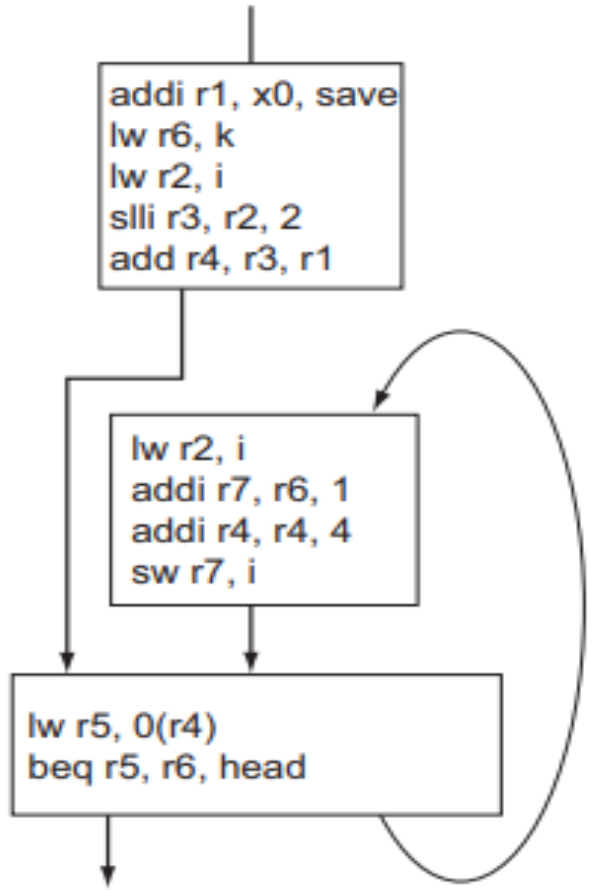
- Global optimizations and data flow analysis operate on a control flow graph, where the nodes represent basic blocks and the arcs represent control flow between basic blocks.



The control flow graph showing the representation of the while loop example after code motion and induction variable elimination

- Consider IR statements number 1 and 6: in both cases, the use-definition information tells us that there are no definitions (changes) of the operands of these statements within the loop
- these IR statements can be moved outside the loop
- consider IR statement 2, which loads the value of i. The definitions of i that affect this statement are both outside the loop, where i is initially defined, and inside the loop in statement 10 where it is stored. Hence, this statement is not loop invariant.


9. lw r1, i
10. addi r2, r1, 1
11. sw r2, i
1. addi r3, x0, save
2. lw r4, i
3. slli r5, r4, 2
4. add r6, r5, r3
5. lw r7, 0(r6)
6. lw r8, k
7. beq r7, r8, head


addi r1, x0, save
lw r6, k
lw r2, i
slli r3, r2, 2
add r4, r3, r1
lw r2, i
addi r7, r6, 1
addi r4, r4, 4
sw r7, i
lw r5, 0(r4)
beq r5, r6, head

- Is there any way that the variable k could possibly ever change in this loop? Unfortunately, there is one way. Suppose that the variable k and the variable i actually refer to the same memory location, which could happen if they were accessed by pointers or reference parameters.

Register Allocation

- Register allocation is perhaps the most important optimization for modern load-store architectures.
- register allocation enhances the value of other optimizations, such as common subexpression elimination
- Register allocation is done on both a local basis and a global basis, that is, across multiple basic blocks but within a single function
- Local register allocation is usually done late in compilation, as the final code is generated

- Modern global register allocation uses a region-based approach, where a region (sometimes called a live range) represents a section of code during which a particular variable could be allocated to a particular register
- How is a region selected? The process is iterative:

1. Choose a definition (change) of a variable in a given basic block; add that block to the region.

2. Find any uses of that definition, which is a data flow analysis problem; add any basic blocks that contain such uses, as well as any basic block that the value passes through to reach a use, to the region.

3. Find any other definitions that also can affect a use found in the previous step and add the basic blocks containing those definitions, as well as the blocks the definitions pass through to reach a use, to the region.

4. Repeat steps 2 and 3 using the definitions discovered in step 3 until convergence

The set of basic blocks found by this technique has a special property: if the designated variable is allocated to a register in all these basic blocks, then there is no need for loading and storing the variable