RISC V ARCHITECTURE

Rajeshwari B
Department of Electronics and Communication Engineering

PES UNIVERSITY ONLINE

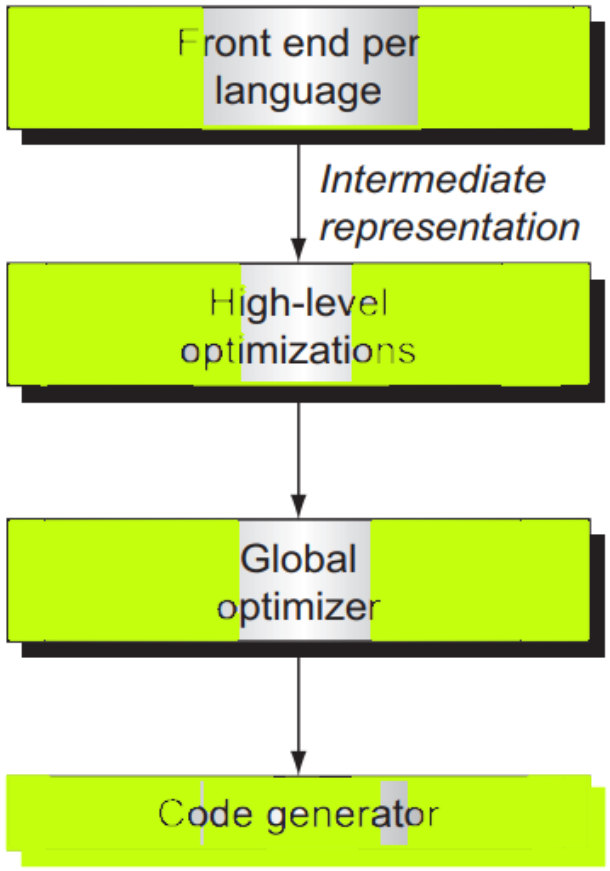
RISC V ARCHITECTURE

Instructions: The Language of Computer –Part-B

Rajeshwari B
Department of Electronics and Communication Engineering

Compiling C

Structure of recent compilers



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

Compiling C

Example:

To illustrate the concepts, the C version of a while loop considered

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

The Front End

The function of the front end is to read in a source program; check the syntax and semantics;

translate the source program to an intermediate form that interprets most of the language-specific operation of the program.

Compiling C

The front end is typically broken into four separate functions:

1. **Scanning** reads in individual characters and creates a string of tokens. Examples of tokens are reserved words, names, operators, and punctuation symbols.

   In the above example, the token sequence is while, (, save, [, i,], ==, k,), i, +=, 1. A word like while is recognized as a reserved word in C, but save, i, and j are recognized as names, and 1 is recognized as a number.

2. **Parsing** takes the token stream, ensures the syntax is correct, and produces an abstract syntax tree, which is a representation of the syntactic structure of the program.

3. **Semantic analysis** takes the abstract syntax tree and checks the program for semantic correctness. Semantic checks normally ensure that variables and types are properly declared and that the types of operators and objects match, a step called type checking.

During this process, a symbol table representing all the named objects—classes, variables, and functions—is usually created and used to type-check the program.

4. Generation of the intermediate representation (IR) takes the symbol table and the abstract syntax tree and generates the intermediate representation that is the output of the front end.

The most common intermediate form looks much like the RISC-V instruction set but with an infinite number of virtual registers;

High-Level Optimizations

High-level optimizations are transformations that are done at 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.

Loop transformations that can reduce loop overhead,-**Loop-unrolling** involves taking a loop, replicating the body multiple times, and executing the transformed loop fewer times.

Local and Global Optimizations

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;

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

Compiling C

Example: Optimization- Common subexpression elimination

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

// 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

Recalculating address x[i] again using different set of registers

```
addi r106, x0, x
lw r107,i
mul r108,r107,4
add r109,r106,r107
```

sw r105,0(r109)

Compiling C

Common subexpression elimination

```
// x[i] + 4

addi r100, x0, x
lw r101,I
slli r102,r101,2          // Strength reduction Replacing mul with shift
add r103,r100,r102
lw r104, 0(r103)

addi r105, r104,4
sw r105, 0(r103)          // value of x[i] is in r104
```

Compiling C

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.

Global Code Optimizations

Many global code optimizations have the same aims as those used in the local case, including common subexpression elimination, constant propagation, copy propagation, and dead store and dead code elimination.

There are two other important global optimizations: code motion and induction variable elimination. Both are loop optimizations; that is, they are aimed at code in loops.

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.

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:

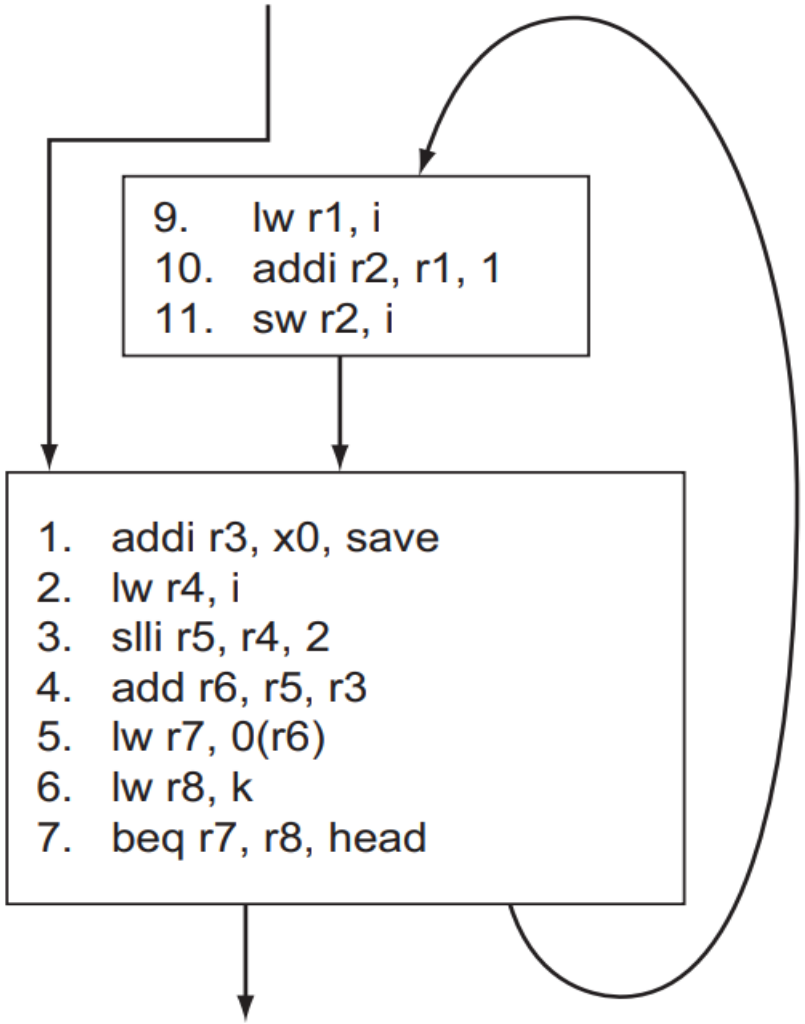
■ Consider the case of an opportunity for common subexpression elimination, say, of an IR statement like add Rx, R20, R50.

■ Consider the second lw of i into R107 within the earlier example: how do we know whether its value is used again?

■ 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.

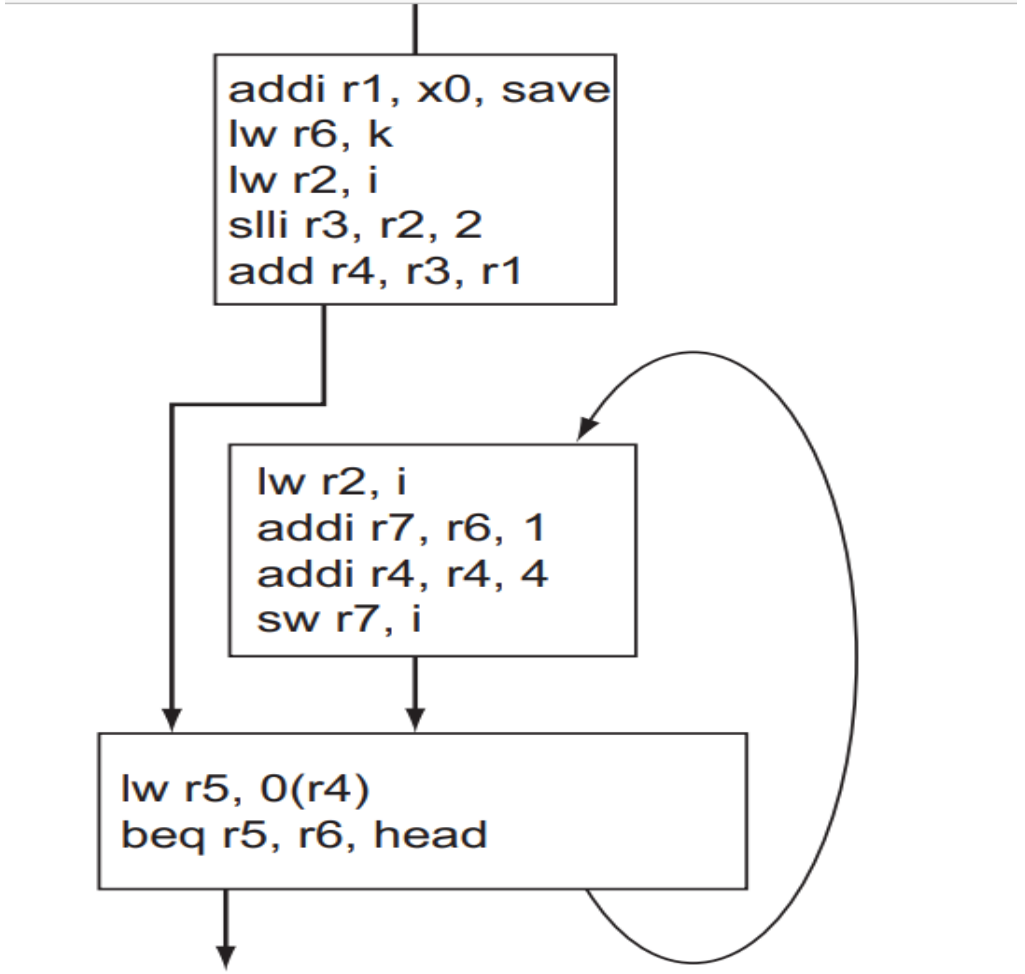
Compiling C

A control flow graph for the while loop example.



Compiling C

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



Register Allocation

Register allocation is perhaps the most important optimization for modern load-store architectures.

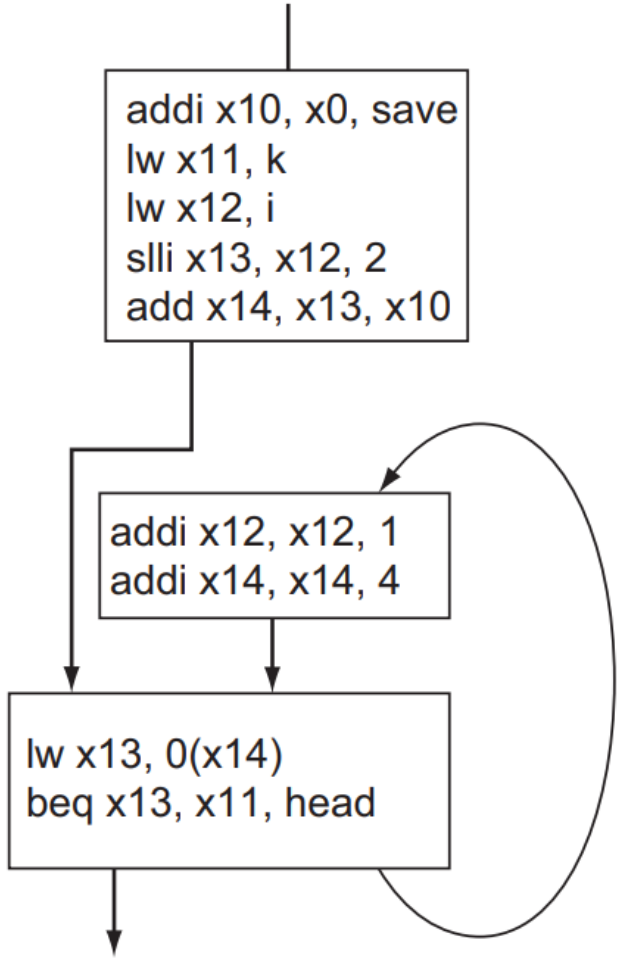
Furthermore, register allocation enhances the value of other optimizations, such as common subexpression elimination.

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.

Compiling C

Region selection- The process is iterative:

The control flow graph showing the representation of the while loop example after code motion and induction variable elimination and register allocation, using the RISC-V register names.



Compiling C

Compiling C:Major types of optimizations and explanation of each class

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

THANK YOU

Rajeshwari B
Department of Electronics and Communication Engineering
rajeshwari@pes.edu