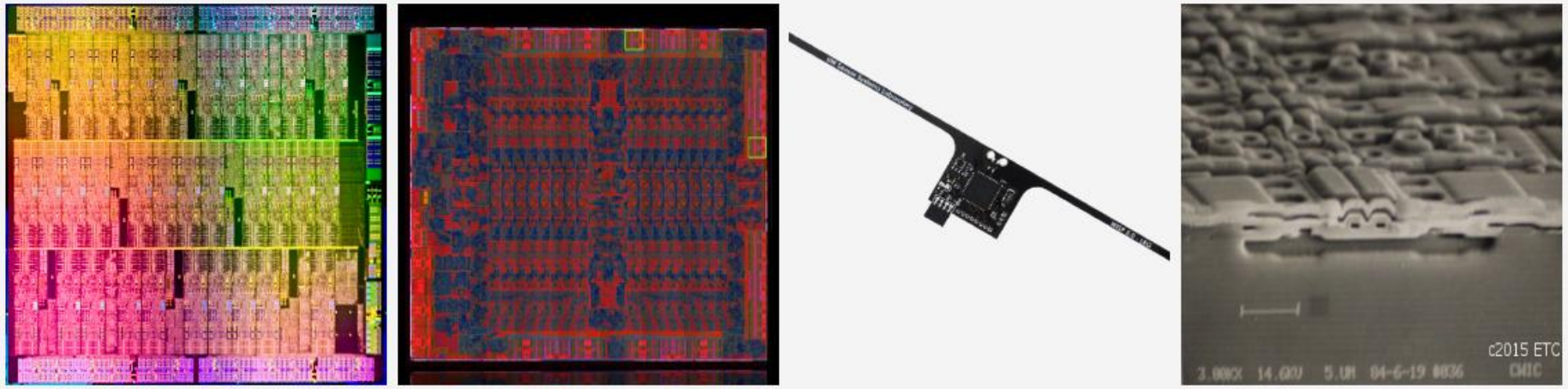
18-344: Computer Systems and the Hardware-Software Interface

Fall 2025



Course Description

Lecture 14: The Compiler Is Here To Help

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

Credit: Brandon Lucia

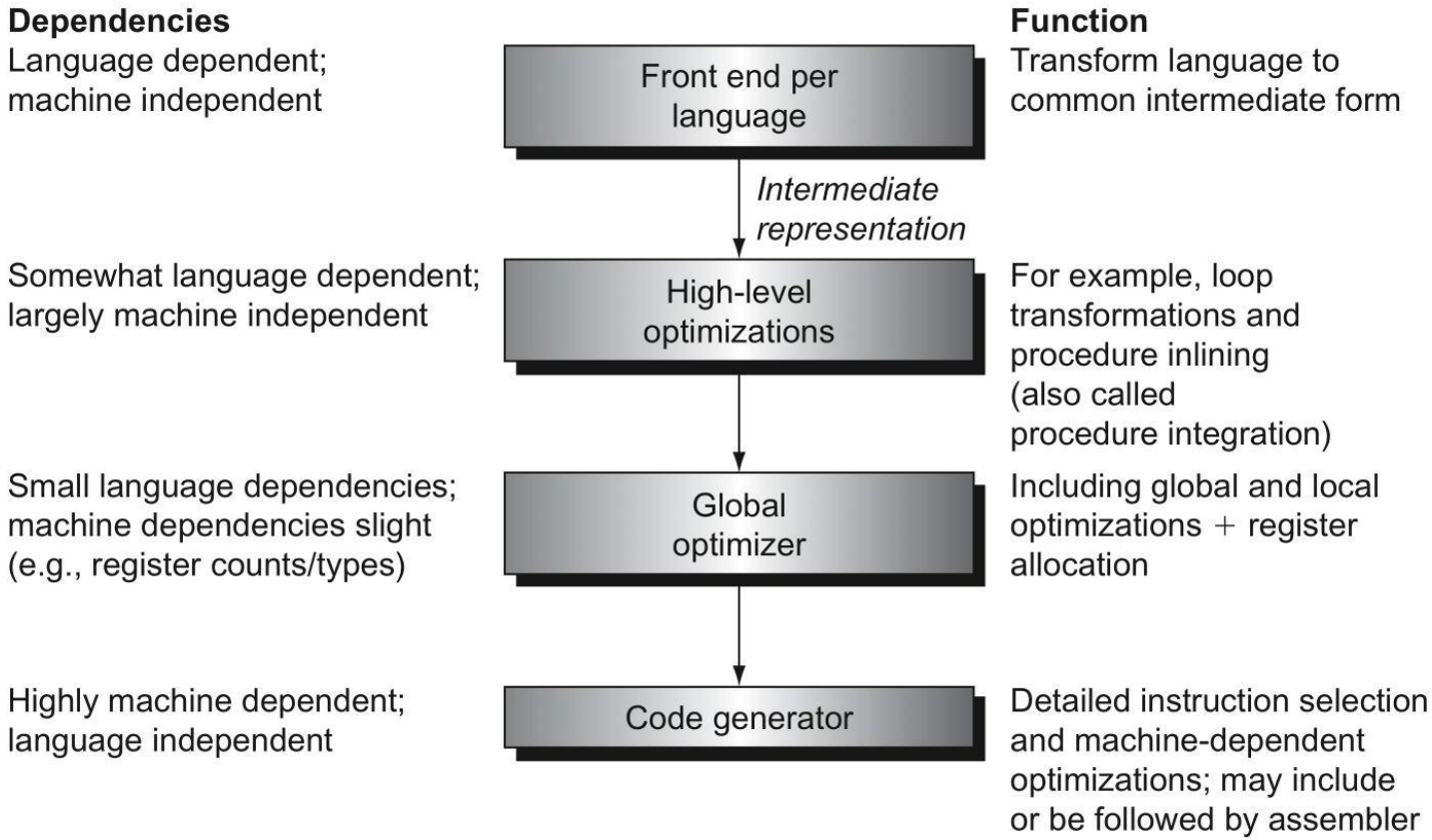
Today: The Compiler is Here to Help

- Next we will look to how the compiler represents and reasons about a program
- We will learn about compiler optimizations and how they get defined and applied

- Note: This lecture was intended to fall after VLIW and vector machines and before VM in the schedule, but was bumped back by one lecture to give you more time on the VM lab.
    - When studying later, you may want to review it at that point in your flow.

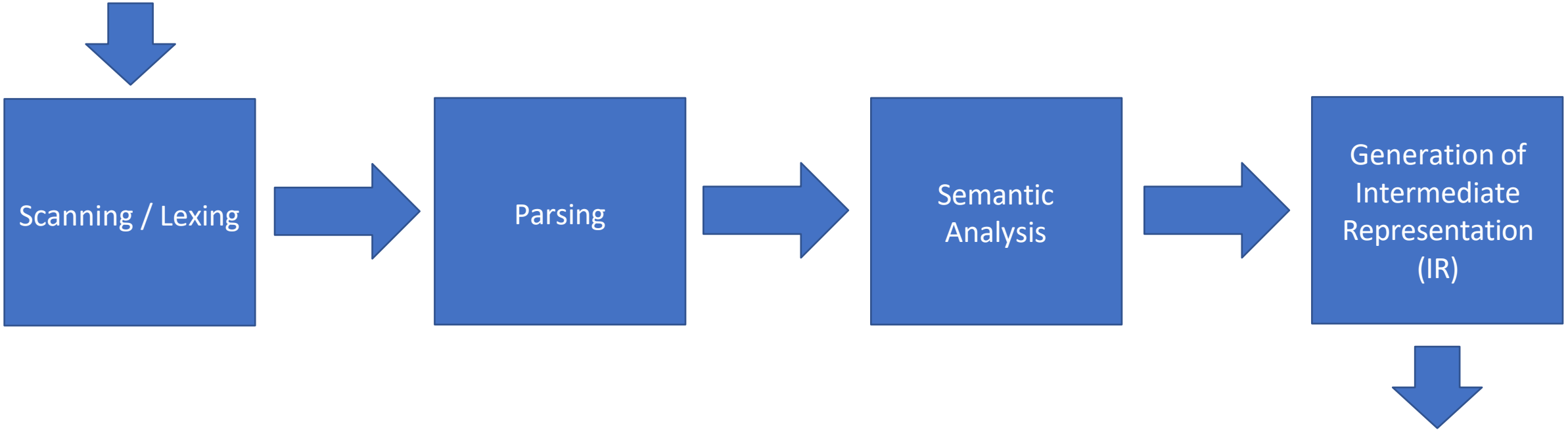
Purpose & Operation of a Compiler

- Compilers process high-level language code to (eventually) produce machine code
- Organized into a front-end, “middle-end”, and back-end.
- Middle-end analyzes/transforms program represented as an “Intermediate Representation”



Front-end Phases in the Compiler

Input: High-level
Language / C Program



Output: IR to process in middle-end

Scanning / Lexing: Reading in raw characters

- Process input string from code file into characters
- Use finite state machine to process *characters* into *lexemes* to match *tokens*
- Maximal Munch Rule: Always match longest available lexeme to token
- Output: series of tokens identified by a (string) name that can be parsed to give semantic meaning

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

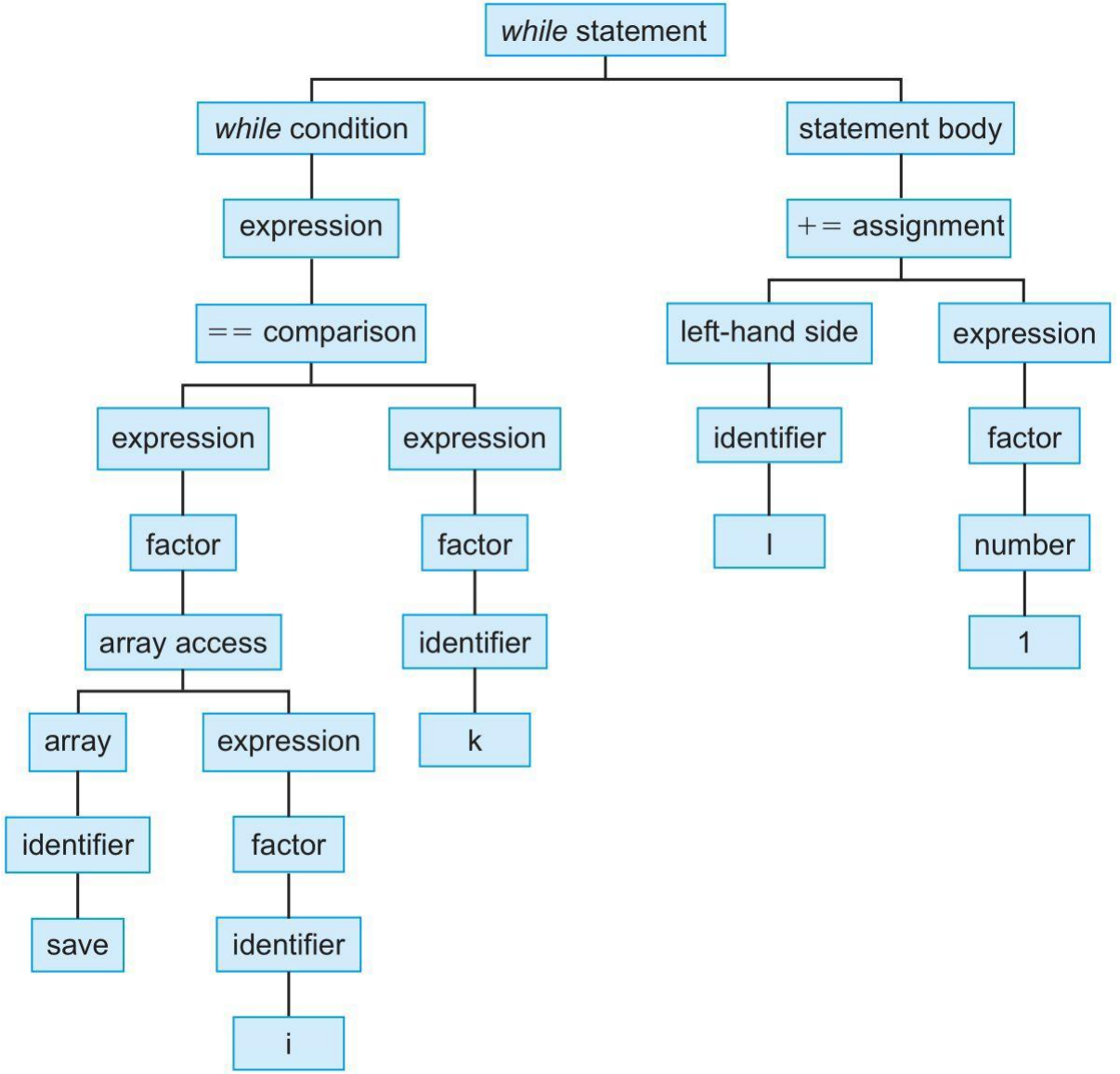
```
‘w’ ‘h’ ‘i’ ‘l’ ‘e’ ‘(‘
‘s’ ‘a’ ‘v’ ‘e’ ‘[‘ ‘I’
‘]’ ‘=‘ ‘=‘  ‘k’ ‘)’
‘:’
    ‘i’ ‘+’ ‘=‘ ‘1’ ‘;’
```

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

A “scanner” or “lexer” processes text into tokens

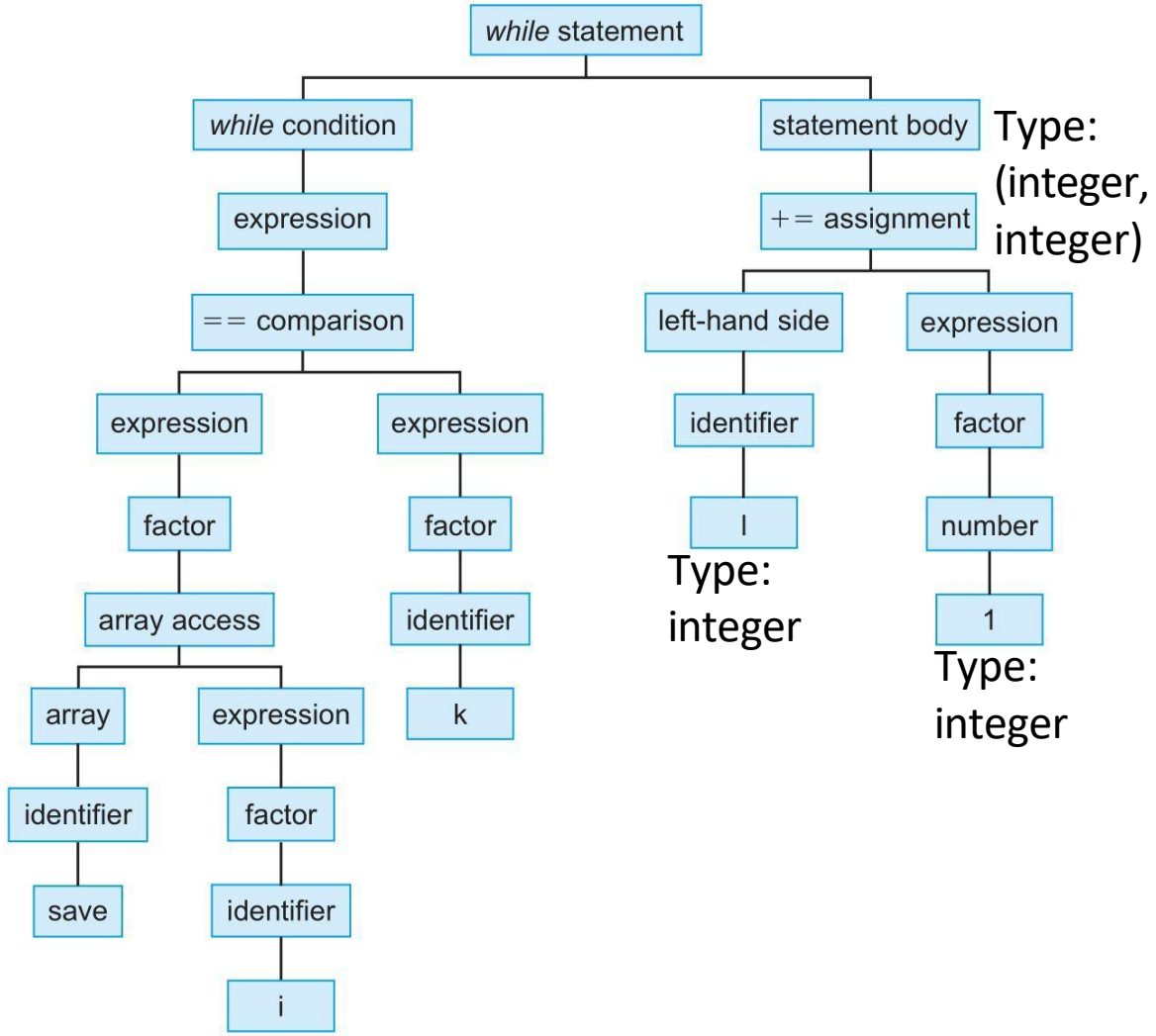
Parsing tokens

- Assemble *tokens* into an *abstract syntax tree* (AST) that represents the syntactic structure of the program
- Tree represents recursive substructure of program
- Includes sufficient information to assign *semantics* to components of the tree for *analysis, optimization,* and *mapping* to machine instructions

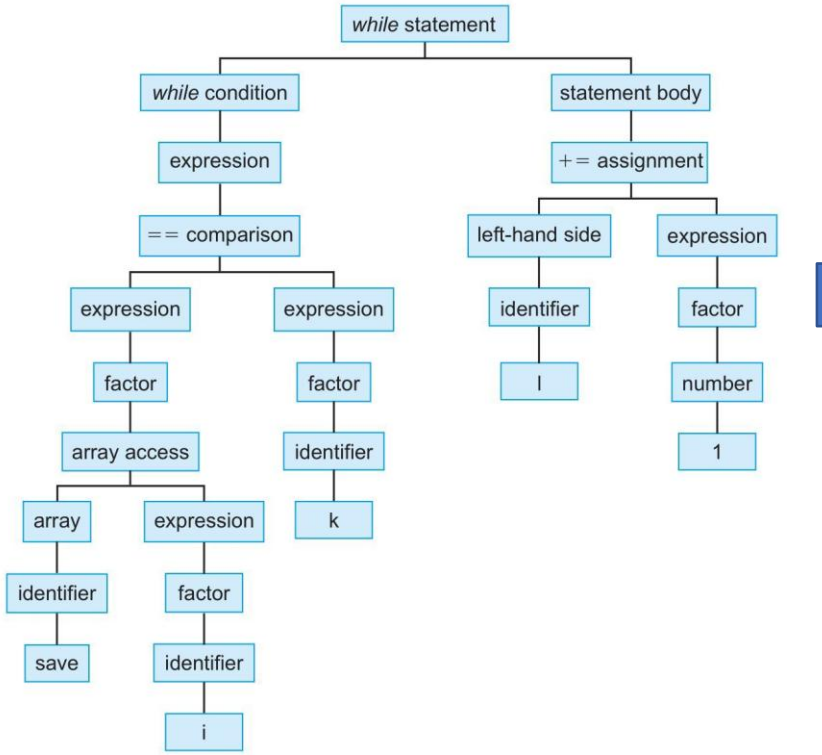


Semantic Checking & Analysis

- Programmatically examine the AST to find a broad class of common errors
- Creates a *symbol table* mapping names & types in program
- **Type checking (most):** make sure computations only applied to correct kinds of values
- **Ownership/Borrow Checking (some):** make sure only zero or one writable reference to a memory location
- **Data-race checking (some):** make sure that no code allows data races
- **Security / data integrity checking:** make sure that non-trustworthy operations to do not see secret data



Converting to Intermediate Representation (IR)



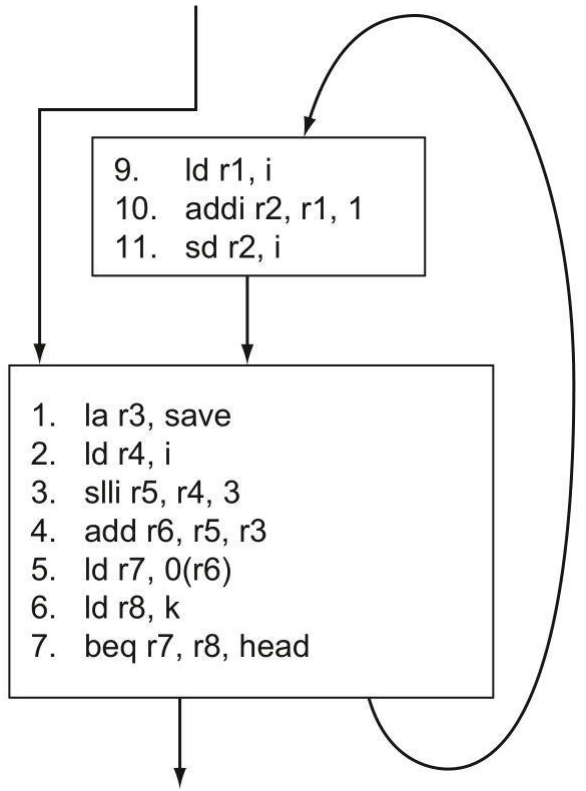

```
loop:
  # comments are written like this--source code often included
  # while (save[i] == k)
  la   r100, save          # r100 = &save[0]
  ld   r101, i
  li   r102, 8
  mul  r103, r101, r102
  add  r104, r103, r100
  ld   r105, 0(r104)       # r105 = save[i]
  ld   r106, k
  bne  r105, r106, exit
  # i += 1
  ld   r106, i
  addi r107, r106, i       # increment
  sd   r107, i
  j    loop                # next iteration
exit:
```

Purpose of IR: simple set of semi-universal, language- and machine-independent operations to which to apply analysis & optimizations

Structure of IR: varies depending on the IR model used by compiler, usually looks assembly-ish & has hierarchical structure (functions, blocks, instructions)

IR: Linear View vs. Control-flow Graph View

```
loop:
  # comments are written like this--source code often included
  # while (save[i] == k)
  la   r100, save          # r100 = &save[0]
  ld   r101, i
  li   r102, 8
  mul  r103, r101, r102
  add  r104, r103, r100
  ld   r105, 0(r104)       # r105 = save[i]
  ld   r106, k
  bne  r105, r106, exit
  # i += 1
  ld   r106, i
  addi r107, r106, i       # increment
  sd   r107, i
  j    loop                # next iteration
exit:
```



Linear: Directly maps to high-level code, no recursive/hierarchical sub-structure

Control-flow Graph: Break code into basic blocks (linear, single entry point, single exit point) with arcs describing possible control flows

Single Static Assignment (SSA)

All registers in IR are "fresh" virtual registers, assigned to *exactly once* anywhere in the code. These registers do not correspond to any machine.

Why?

```
loop:
  # comments are written like this--source code often included
  # while (save[i] == k)
  la   r100, save          # r100 = &save[0]
  ld   r101, i
  li   r102, 8
  mul  r103, r101, r102
  add  r104, r103, r100
  ld   r105, 0(r104)       # r105 = save[i]
  ld   r106, k
  bne  r105, r106, exit
  # i += 1
  ld   r106, i
  addi r107, r106, i       # increment
  sd   r107, i
  j    loop                # next iteration
exit:
```

SSA Makes Dependence Chains Easy

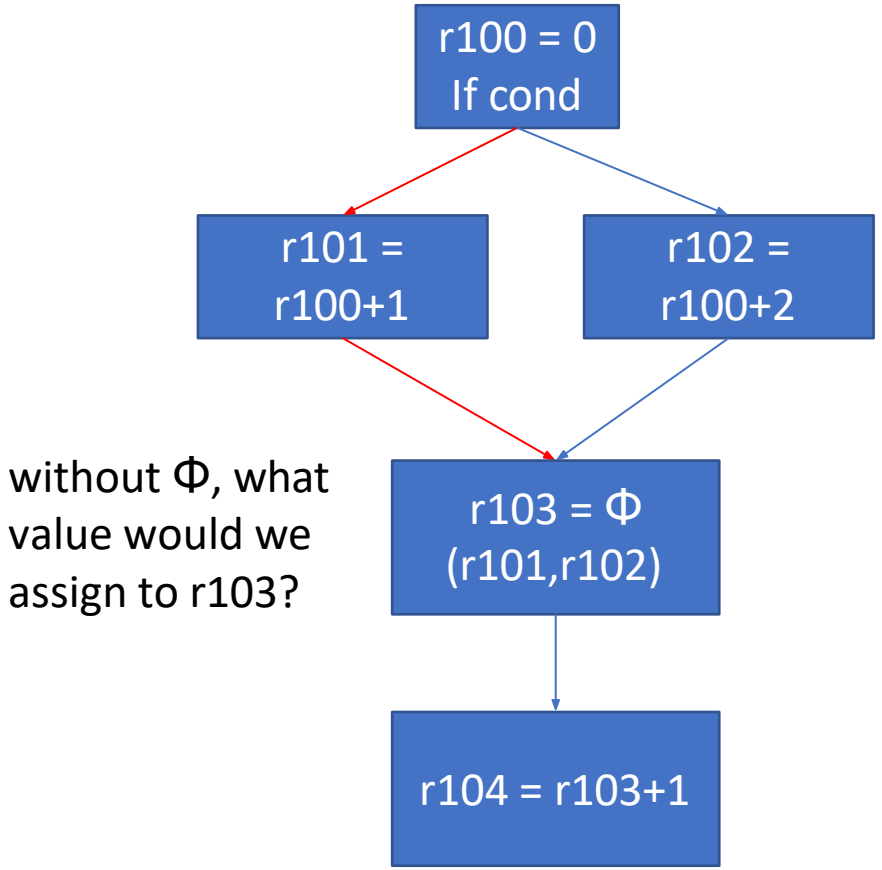
Definition – Use Chains (or Def-Use Chains) emerge directly from analysis of virtual register assignments and uses

```
loop:
  # comments are written like this--source code often included
  # while (save[i] == k)
  la   r100, save          # r100 = &save[0]
  ld   r101, i
  li   r102, 8
  mul  r103, r101, r102
  add  r104, r103, r100
  ld   r105, 0(r104)       # r105 = save[i]
  ld   r106, k
  bne  r105, r106, exit
  # i += 1
  ld   r106, i
  addi r107, r106, i       # increment
  sd   r107, i
  j    loop                # next iteration
exit:
```

Φ –nodes: Handling Control-flow Divergence

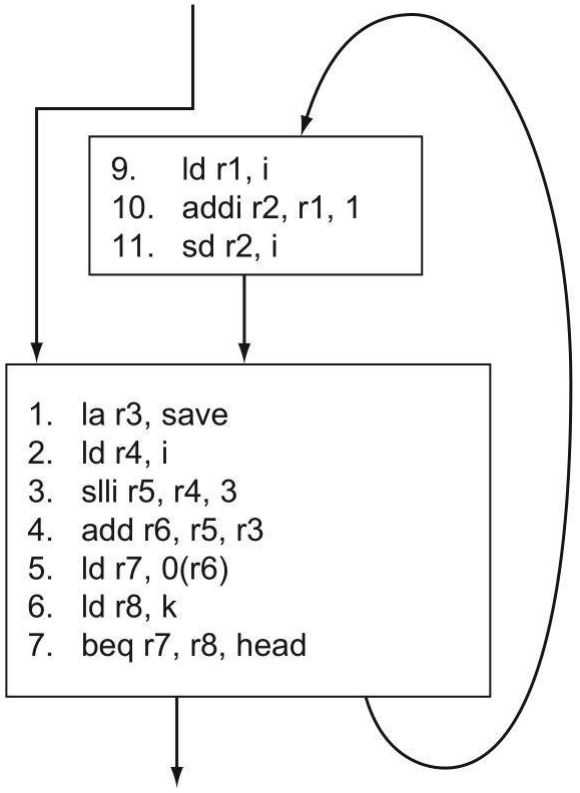
- When control paths diverge, need to re-converge eventually
- If single variable conditionally assigned on both sides of branch, how to decide its value?

```
x = 0;
if(cond){x+=1}
     else{x+=2};
y = x + 1;
```



Compiler Optimizations

- **Local Optimization**: optimize operations within a single basic block; “cleanup” before global optimizations
- **Global Optimization:** optimization across basic blocks
- **Global register allocation:** Register mapping pass required for making code fast



Optimization Example: Dead Code Elimination

```
x = 0;
if(x>0){x+=1}
    else{x+=2};
y = x + 1;
```

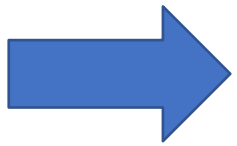
```
x = 0;
x += 2;
y = x+1;
```

Why would you have a program with weird code like this in it?

Optimization Example: Constant / Copy Propagation

```
x = 0;
if(x>0){x+=1}

else{x+=2};
y = x + 1;
```

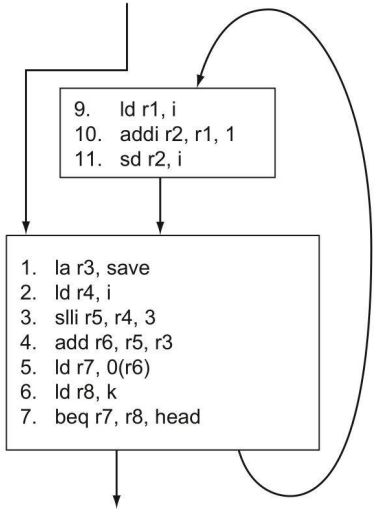


```
x=0
x+=2
y = x+1
```

```
y = 3;
```

Significantly simplified from our original code by applying a sequence of optimizations

Optimization Example: Common Subexpression Elimination



x[i] = x[i] + 4

```
// x[i] + 4
la r100,x
ld r101,i
mul r102,r101,8
add r103,r100,r102
ld r104, 0(r103)
//
addi r105, r104,4
la r106,x
ld r107,I
mul r108,r107,8
add r109,r106,r107
sd r105,0(r109)
```

```
// x[i] + 4
la r100,x
ld r101,i
slli r102,r101,3
add r103,r100,r102
ld r104, 0(r103)
// x[i] is in r104
addi r105, r104,4
sd r105, 0(r103)
```

Optimization Example: Strength Reduction

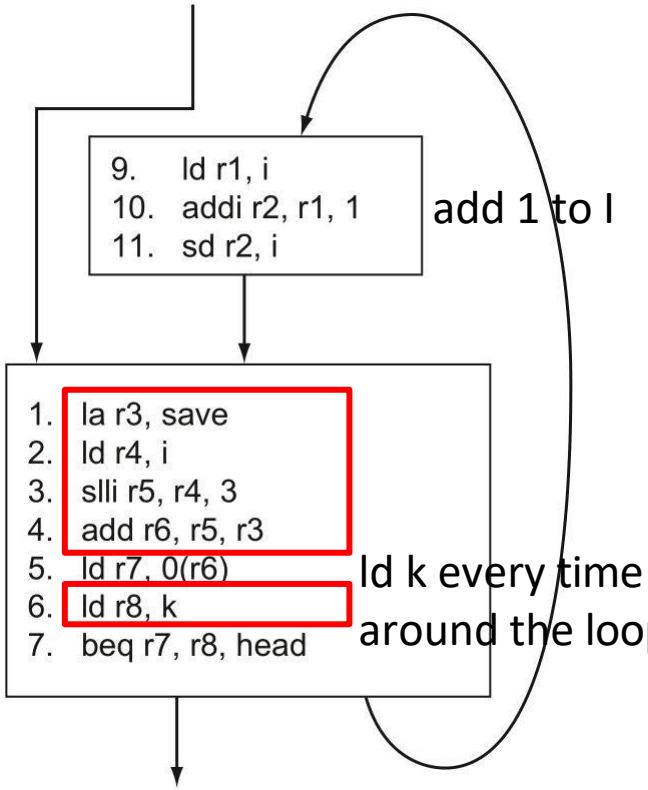
x[i] = x[i] + 4

```
// x[i] + 4
la r100,x
ld r101,i
mul r102,r101,8
add r103,r100,r102
ld r104, 0(r103)
//
addi r105, r104,4
la r106,x
ld r107,I
mul r108,r107,8
add r109,r106,r107
sd r105,0(r109)
```

```
// x[i] + 4
la r100,x
ld r101,i
slli r102,r101,3
add r103,r100,r102
ld r104, 0(r103)
// x[i] is in r104
addi r105, r104,4
sd r105, 0(r103)
```

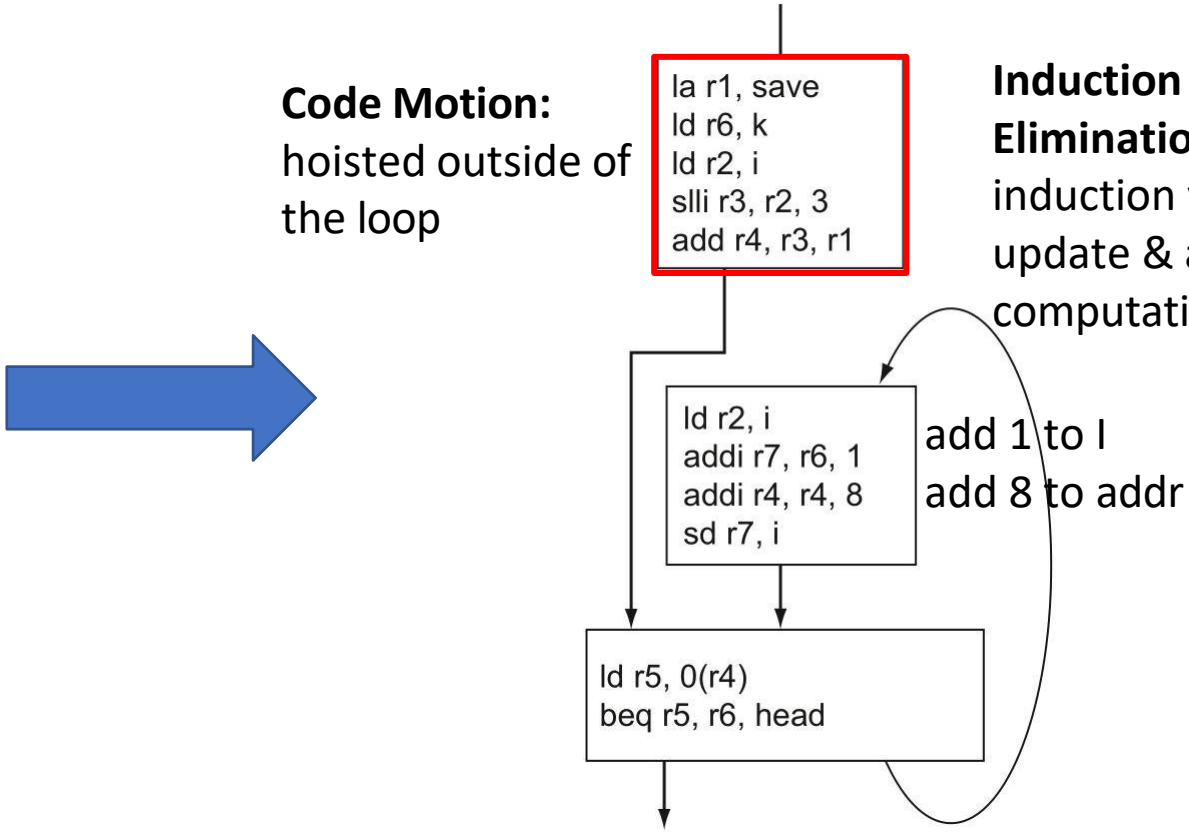
Global Optimization Example: Code Motion / Induction Variable Elimination

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



addr = save + i*8
via
`ld`
`slli`
`addi`

Code Motion: hoisted outside of the loop



Induction Variable Elimination: Re-write induction variable update & address computation directly

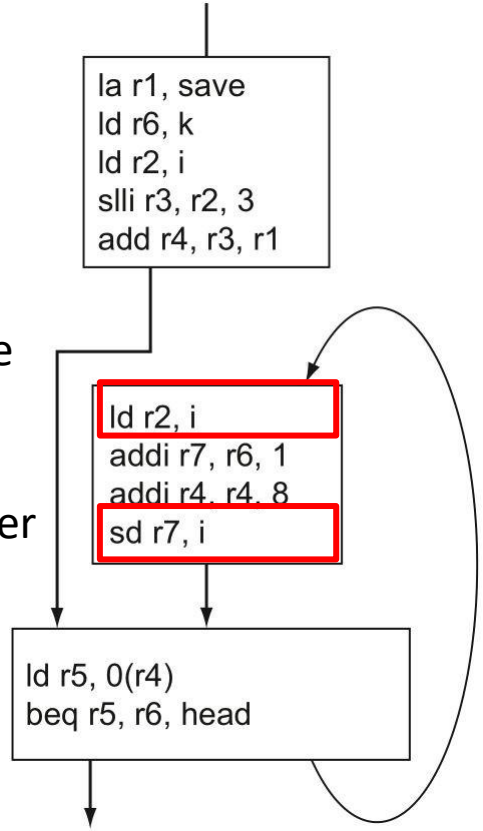
Global optimizations tend to be more complex and involve reasoning that is more difficult to prove correct

Optimization Summary

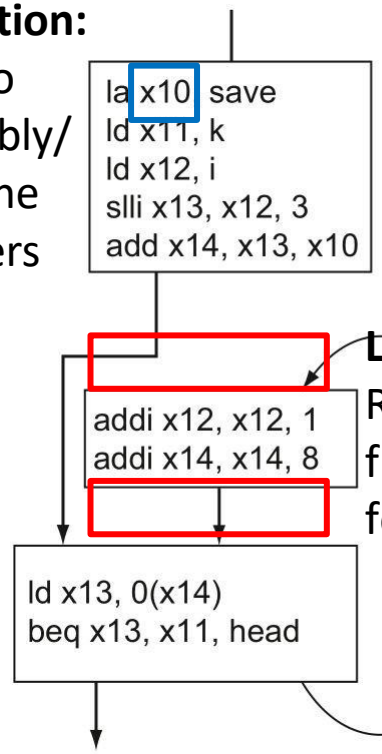
Optimization name	Explanation	gcc level
High level	*At or near the source level; processor independent*	
Procedure integration	Replace procedure call by procedure body	O3
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	O1
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	O1
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	O1
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	O2
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., $A = X$) with X	O2
Code motion	Remove code from a loop that computes the same value each iteration of the loop	O2
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	O2
Processor dependent	*Depends on processor knowledge*	
Strength reduction	Many examples; replace multiply by a constant with shifts	O1
Pipeline scheduling	Reorder instructions to improve pipeline performance	O1
Branch offset optimization	Choose the shortest branch displacement that reaches target	O1

Register Allocation: Eliminating Memory Loads and Stores

Load/Store temp variables before/after use

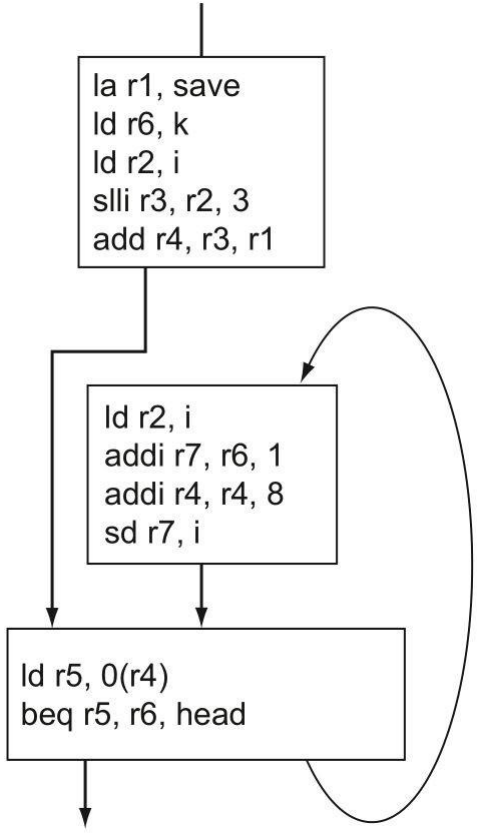


Register Allocation: Map to assembly/ machine registers



Load/Store Elimination: Remove loads from/stores to memory for vars in regs

Register Allocation: Algorithmically



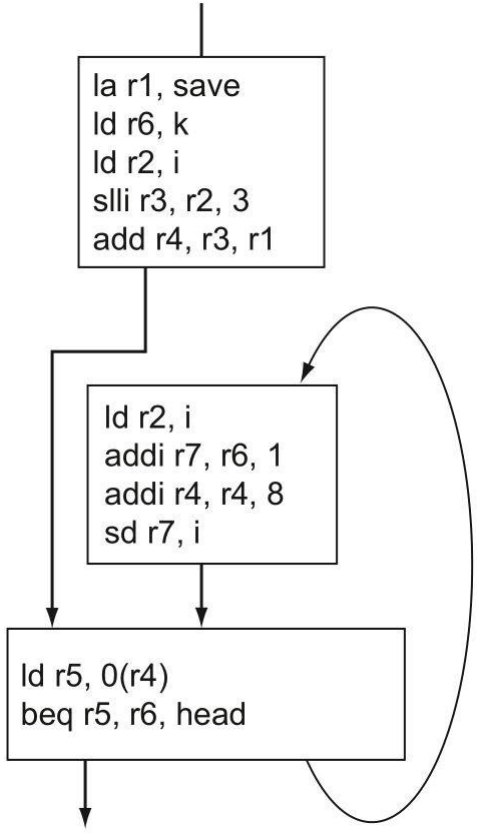
Region-based Register Allocation – Finding Regions:

1. Choose a variable definition, d (assignment); add its basic block to region, R
2. Find uses, u_i of the definition d in other basic blocks, add those basic blocks to R; add any block on a path between a definition's block and the use's block to R too.
3. Find any other definitions that could affect u_i and add the blocks containing those definitions to R
4. Repeat steps 2 and 3 until you do not add any more regions to R.

Property of resulting set of blocks: if the variable subject to definition d is allocated in register in these blocks, then there is no need to load and store the variable after its initial load/definition

Proof Intuition: definitions flow through these and only these blocks along reachable paths to all possible uses. If all uses and defs use the same register name, no need to write to/read from memory).

Register Allocation: Algorithmically



Region-based Register Allocation – Mapping to machine registers:

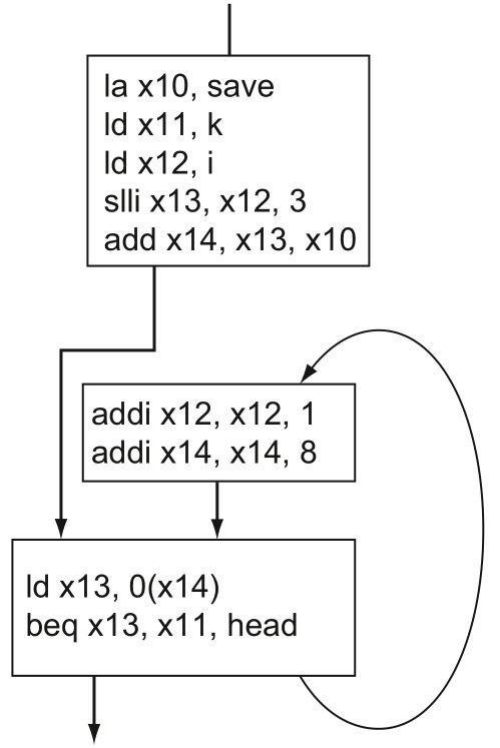
1. Compute regions for each virtual/IR register in program
2. Construct **region interference graph**,
   G=(V,E): V = {regions}, E = { (r1,r2) | r1 and r2 include a basic block in common}
3. Run graphColor(G) to assign a color to each region such that no adjacent vertices in G have the same color
4. If #colors <= #machine registers, assign one machine register per color and register allocate variables in regions.
5. If #colors > #machine registers, need to **spill** register to memory with load/store pair, essentially splitting regions until #colors <= #machine registers

Key property: **regions that do not include overlapping basic blocks can use the same register for a different value/definition**

Is the code you write the code you run?

```
while(save[i] == k]):
  x[i] = x[i] + 4
```

```
loop:
  # comments are written like this--source c
  # while (save[i] == k)
  la   r100, save          # r100 = &save[0]
  ld   r101, i
  li   r102, 8
  mul  r103, r101, r102
  add  r104, r103, r100
  ld   r105, 0(r104)       # r105 = save[i]
  ld   r106, k
  bne  r105, r106, exit
  # i += 1
  ld   r106, i
  addi r107, r106, i       # increment
  sd   r107, i
  j    loop                # next iteration
exit:
```



Key idea: optimize the inner loop automatically!
Inner loop in initial IR representation is 12 instructions long including 7 mem ops (long time if not cache hits).
Inner loop in optimized version is 5 instructions including 1 mem op.

Intermediate Representation Design

- What properties are desirable in an IR? (what properties are *undesirable* in an IR?)
- How to select an IR if you are a compiler writer?
- What aspects of the IR do you care about if you're a programmer?
- What aspects of the IR do you care about if you're an architect?

```
loop:
  # comments are written like this--source code often incl
  # while (save[i] == k)
  la   r100, save          # r100 = &save[0]
  ld   r101, i
  li   r102, 8
  mul  r103, r101, r102
  add  r104, r103, r100
  ld   r105, 0(r104)       # r105 = save[i]
  ld   r106, k
  bne  r105, r106, exit
  # i += 1
  ld   r106, i
  addi r107, r106, i       # increment
  sd   r107, i
  j    loop                # next iteration
exit:
```

Enter: LLVM the “new” great IR ca. 2004

- Designed to support transparent program analysis and transformation for **arbitrary programs**
- Provide high-level information to compiler at compile-time, link-time (and run-time); **always have a CFG**
- Language-independent **type system**
- Explicit support for **typed address arithmetic**
- Uniform abstraction for exception-handling (setjmp / longjmp)
- **Code representation based on Single Static Assignment (SSA) with explicit phi nodes**

LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu
http://llvm.cs.uiuc.edu/

Enter: LLVM the “new” great IR ca. 2004

- LLVM has remained the industry and research standard IR for making custom compilers for nearly two decades
- GCC, which you’re very familiar with also, is less extensible, but more robust and with a longer lifetime of community support
- Recently, new players with applications changing dramatically

LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Vikram Adve
bana-Champaign
s.uiuc.edu
uiuc.edu/

what an awful logo!

Do we need a better IR?MLIR the Multi-Level Intermediate Representation

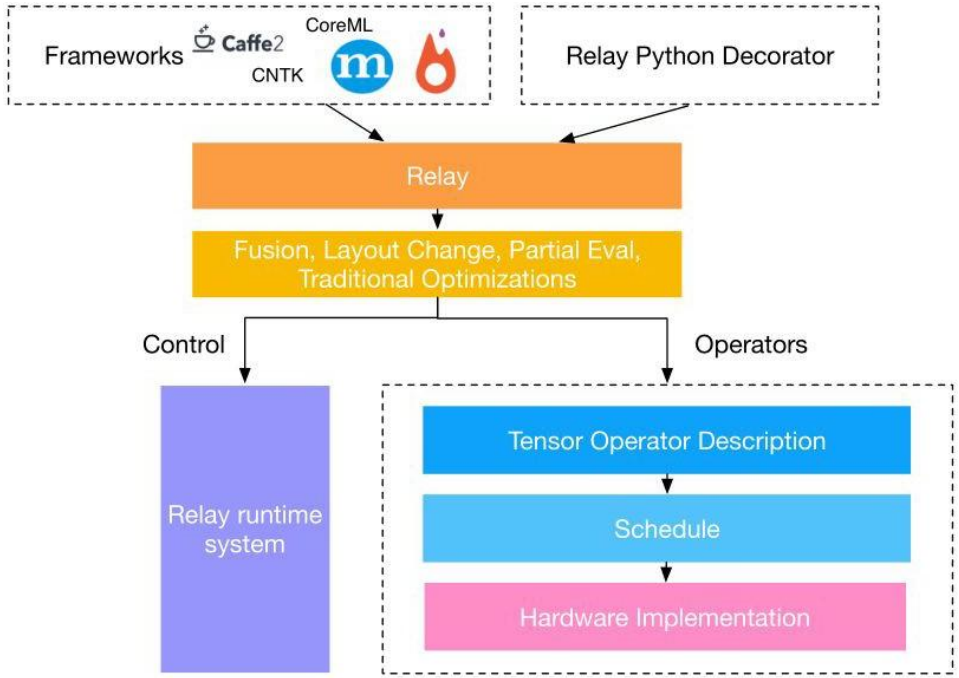
- *"The MLIR project is a novel approach to building reusable and extensible compiler infrastructure. MLIR aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building domain specific compilers, and aid in connecting existing compilers together."*
- Features:
  - Ability to represent dataflows directly (including TensorFlow for ML)
  - Optimization over dataflow graphs directly
  - high-performance-computing-style optimizations across compute kernels
  - Hardware dependent features: DMA insertion, cache management, memory tiling mapping to vector hardware primitives
  - Hardware accelerator specific operations
- Interesting to see where this project goes in the next year or two

Do we need an IR specifically made for machine learning computations?

- Relay IR for Machine Learning / Tensor processing frameworks
- Map from any high-level ML framework to IR
- Generate orthogonal control (runtime code) and scheduling / hardware mapping (data layout + operators)
- Highly specialized for ML & Tensor-specific optimization

Relay: A New IR for Machine Learning Frameworks

Jared Roesch jroesch@cs.uw.edu
Steven Lyubomirsky sslyu@cs.uw.edu
Logan Weber weberlo@cs.uw.edu
Josh Pollock joshpoll@cs.uw.edu
Marisa Kirisame jerry96@cs.uw.edu
Tianqi Chen tqchen@cs.uw.edu
Zachary Tatlock ztatlock@cs.uw.edu
Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle, WA, USA

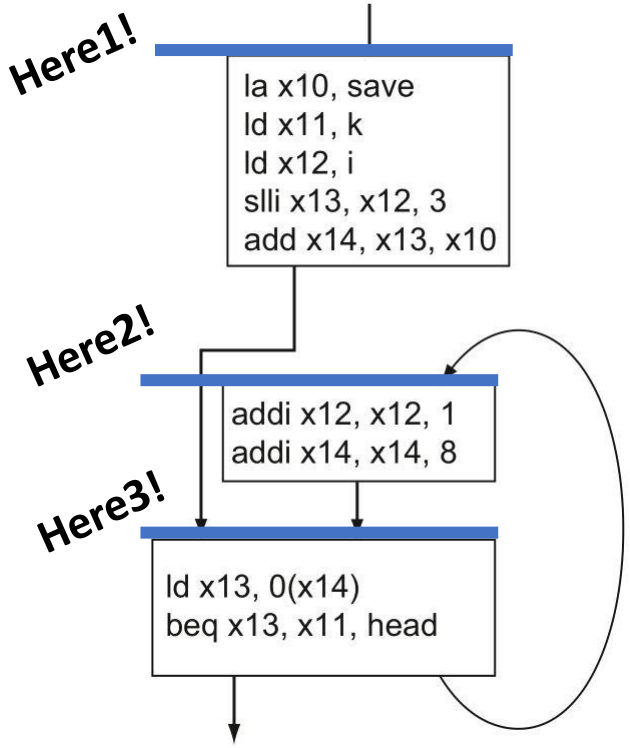


What else can compilers do for you?

- Compilers do more than just map to machines and optimize for you
- Support for **debugging and profiling instrumentation**
- Analysis to check **security properties**

Compiling to profile (like gprof)

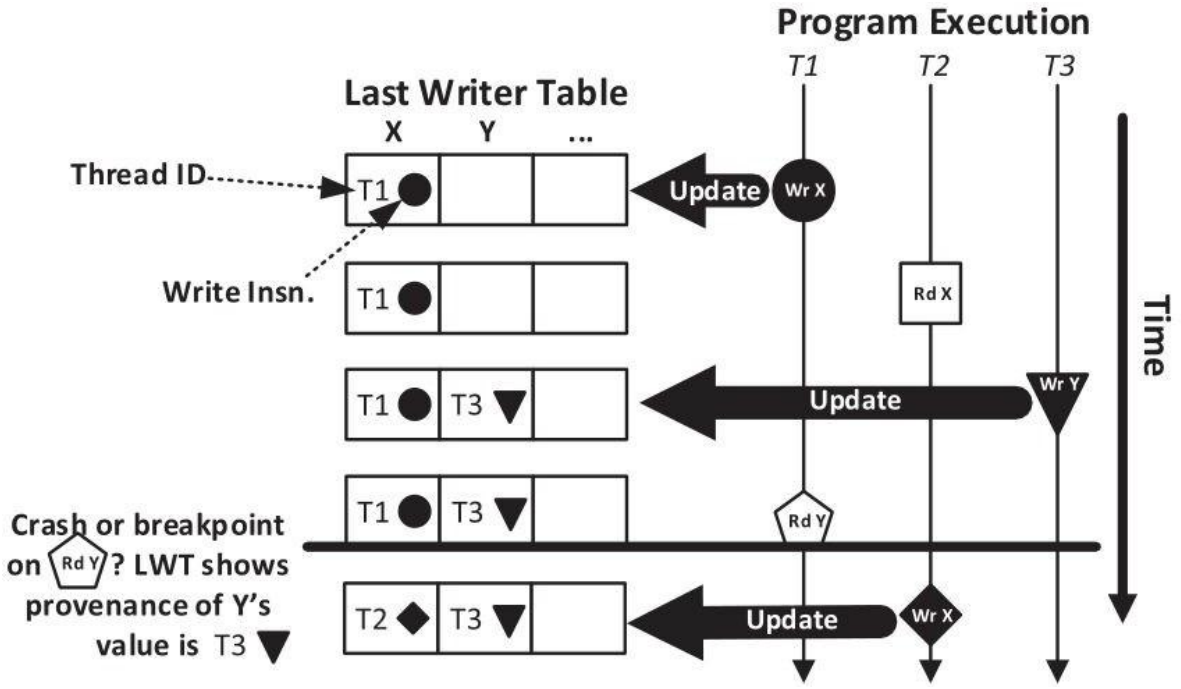
- Insert code that tracks transitions into every basic block
  - (reality: track at *path* granularity to cut down on tracking code)
- Runtime: randomly sample using a timer which block active & make histogram
  - More frequent block in histogram == hot code/inner loop



Compiling to help with debugging: Concurrent Data Provenance Tracking

- Compiler inserts code on every write operation to dynamically track the last writer instruction and thread ID of a variable
- On crash or breakpoint, last writer table helps understand *provenance* of value in crash/op
- Useful for concurrency bug debugging

- `https://github.com/blucia0a/CTraps-gcc`



Information Flow Control Analysis for Security

- Non-privileged operations not allowed to access/be influenced by secret data
- Secret number should never flow to any operation unless we trust that operation with the secret (i.e., correct guess)
- Compiler uses “taint propagation” analysis to tag data and operations
- **Rule:** untrusted op never directly or indirectly affected by confidential data
- **Equivalent rule:** High-assurance op never directly or indirectly affected by untrusted data

```
[1]  public class GuessANumber {
[2]      int secret;
[3]      int tries;
         ...
[4]      void makeGuess ( Integer num )
           throws NullPointerException
         {
[5]          int i = 0;
[6]          if ( num != null ) i = num.intValue();
[7]          if ( i >= 1 && i <= 10 ) {
[8]              if ( tries > 0 && i == secret ) {
[9]                  tries = 0;
[10]                 finishApp(”You win!”);
                 }
[11]             else {
[12]                 tries– ;
[13]                 if ( tries > 0 )
[14]                     message.setText(”Try again”);
[15]                 else
[16]                     finishApp(”Game over!”);
                 }
             }
[17]         else message.setText(”Out of range”);
         }
     }
```

What did we just learn?

- A whirlwind tour of compilers and their design and purpose
- Compilers analyze and translate code you write to be code you run
- Many types of optimization get applied to your code
- Register allocation (and other steps that we omitted) map your code down to the machine
- Compilers are good for a host of other interesting analysis
- Learning LLVM or another compiler-building framework is like a low-grade super-power. Very worth the effort!

What to think about next?

- Next up: Sparse problem optimization
- Further along: Parallelism/concurrency

REMINDER:

Today's lecture logically falls after the advanced architecture lectures, including VLIW and Vector Machines, which we covered before break.

We intentionally misplaced it to let us cover VM and start the VM project earlier, just to give you a little more time on the project.