RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

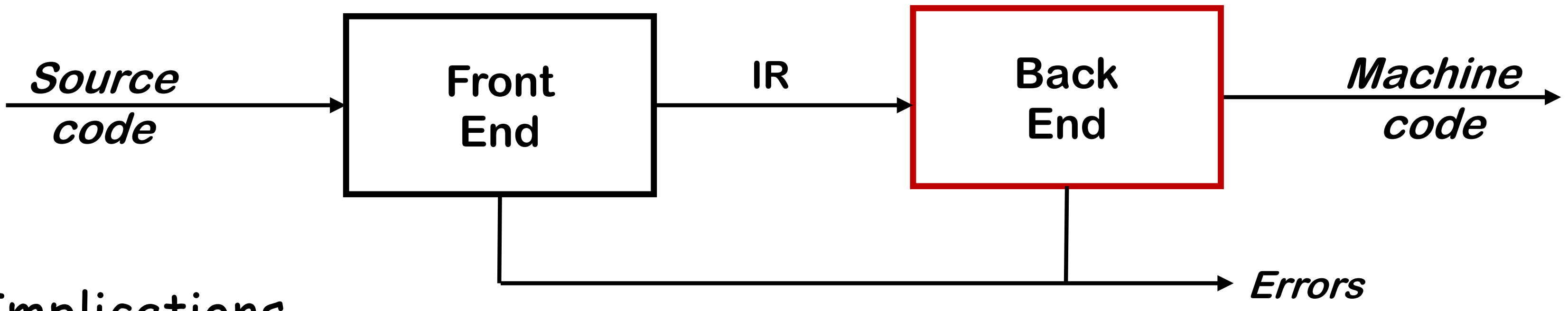
CS415 Compilers ILOC, Code Shape, and Instruction Scheduling

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Announcements

- Recitations and office hours start this week
- Office hours will be posted soon
- Please go to https://www.cs.rutgers.edu/courses/415/classes/spring_2022_kremer/ to download lecture slides
- Lecture videos for first three lectures are/will be available on canvas https://rutgers.instructure.com/courses/160913
- Please go to piazza for questions https://rutgers.instructure.com/courses/160913/external_tools/1590
- Reminder: Get ilab account

Review - Traditional Compiler



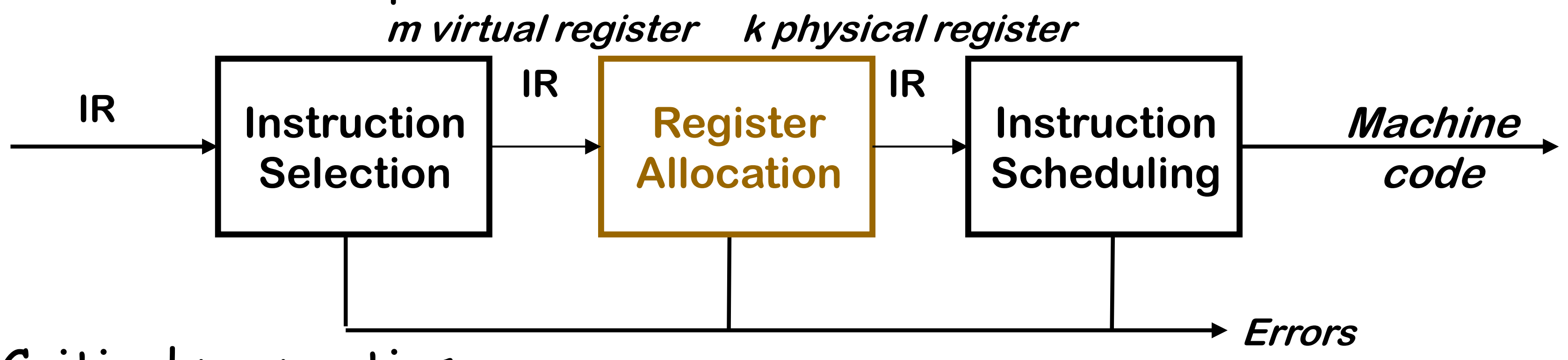
Implications

- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code

Typically, front end is O(n) or O(n log n), while back end is NP-complete

Backend - Register Allocation

Part of the compiler's back end



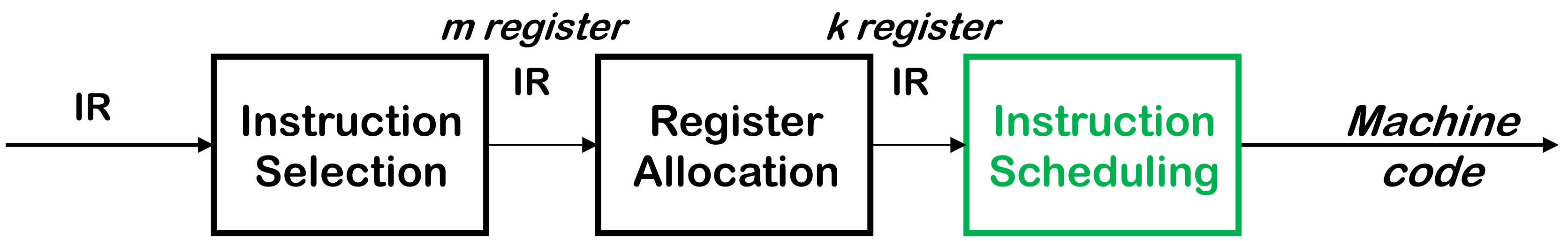
Critical properties

- Produce <u>correct</u> code that uses *k* (or fewer) registers
- Minimize added loads and stores
- Minimize space used to hold *spilled values*
- Operate efficiently

  $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$

Backend - Instruction Scheduling

Part of the compiler's back end



RUTGERS

Local Instruction Scheduling

Readings: EaC 12.1-12.3, Appendix A (ILOC)

Definition

A *basic block* is a maximal length segment of straight-line (*i.e.,* branch free) code. Control can only enter at first instruction of basic block and exit after last instruction.

Local: within single basic block
Global: across procedures/functions

RUTGERS

Instruction Scheduling

Motivation

- Instruction latency (pipelining)
  several cycles to complete instructions; instructions can be issued every cycle
- Instruction-level parallelism (VLIW, superscalar)
  execute multiple instructions per cycle

Issues

- Reorder instructions to reduce execution time
- Static schedule – insert NOPs to preserve correctness
- Dynamic schedule – hardware pipeline stalls
- Preserve correctness, improve performance
- Interactions with other optimizations (register allocation!)

RUTGERS

Instruction Scheduling

Motivation

- Instruction latency (pipelining)
  several cycles to complete instructions; instructions can be issued every cycle
- Instruction-level parallelism (VLIW, superscalar)
  execute multiple instructions per cycle

Issues

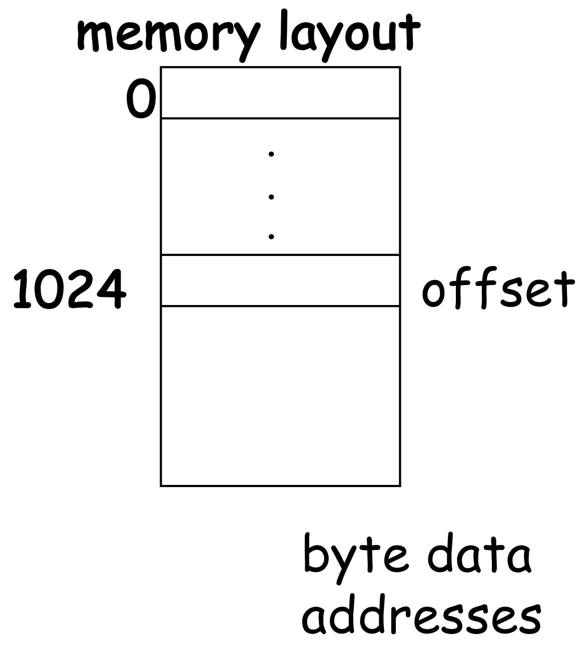
- Reorder instructions to reduce execution time
- Static schedule – insert NOPs to preserve correctness
- Dynamic schedule – hardware pipeline stalls
- Preserve correctness, improve performance
- Interactions with other optimizations (register allocation!)
- Note: After register allocation, code shape contains real, not virtual registers ==> register may be redefined

RUTGERS

Memory Model / Code Shape

Source code

```
A = 5;
B = 6;
C = A + B;
```



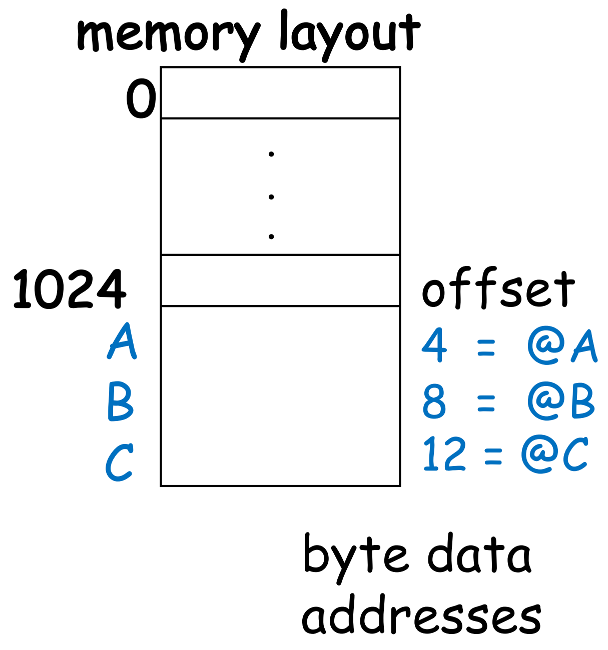
RUTGERS

Memory Model / Code Shape

Source code

```
A = 5;
B = 6;
C = A + B;
```

Example:



Assume A, B, C are integer values of 4 bytes

address(A) = 1024 + offset(A) = 1028
address(B) = 1024 + offset(B) = 1032
address(C) = 1024 + offset(C) = 1036

This convention is used in activation records or stack frames. We use it here for consistency.

More general:

address(X) = base_address + offset(X)

RUTGERS

ILOC (Intermediate Language for Optimizing Compilers)

Instruction scheduling on basic blocks in "ILOC"

- Pseudo-code for a simple, abstracted RISC machine
  - → generated by the instruction selection process
- Simple, compact data structures
- Here: we only use a small subset of ILOC

Naïve Representation:

loadI	2		r1
loadAI	r0	@y	r2
add	r1	r2	r3
loadAI	r0	@x	r4
sub	r4	r3	r5

Quadruples:

- **table of *k x 4* small integers**
- **simple record structure**
- **easy to reorder**
- **all names are explicit**

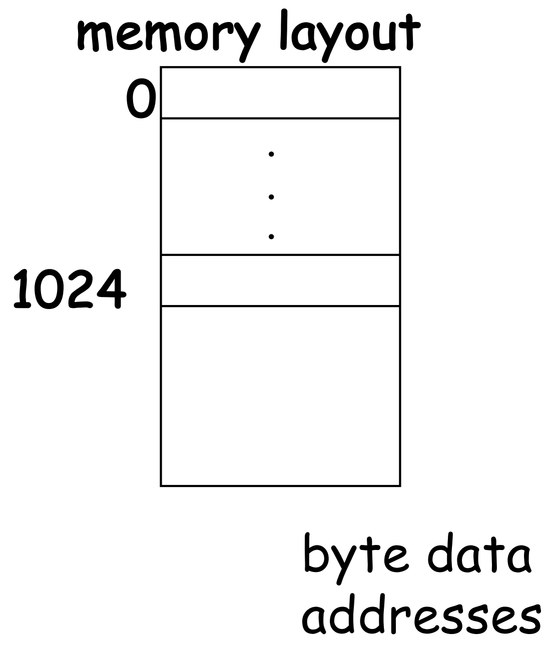
ILOC is described in Appendix A of EAC.

ILOC simulator "sim" is available on ilab:

~uli/cs415/ILOC_Simulator/sim

Memory Model / Code Shape

ILOC: EaC Appendix A



Source code

```
A = 5;
B = 6;
C = A + B;
```

ILOC code

```
loadI 5   ⇒ r1
// compute address of A in r2
 . . .
store r1  ⇒ r2  // content(A) = r1
loadI 6   ⇒ r3
// compute address of B in r4
   . . .
store r3  ⇒ r4  // content(B) = r3
add r1, r3 ⇒ r5
// compute address of C in r6
   . . .
store r5  ⇒ r6  // content(C) = r1 + r3
```

Is this code correct?

RUTGERS

Memory Model / Code Shape

ILOC: EaC Appendix A

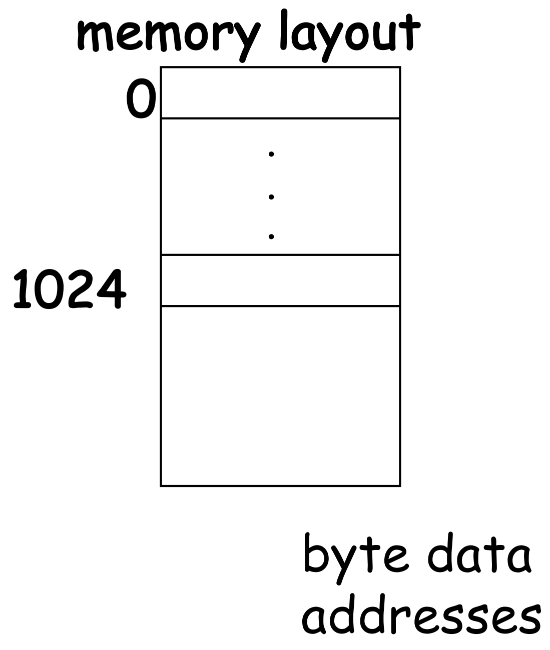
<u>Source code</u>

```
foo (var A, B)
   A = 5;
   B = 6;
   C = A + B;
end foo;

X = 1
call foo(X,X);
print C;
```

<u>ILOC code</u>

```
loadI 5   ⇒ r1
// compute address of A in r2
   . . .
store r1  ⇒ r2  // content(A) = r1
loadI 6   ⇒ r3
// compute address of B in r4
   . . .
store r3  ⇒ r4  // content(B) = r3
add r1, r3 ⇒ r5
// compute address of C in r6
   . . .
store r5  ⇒ r6  // content(C) = r1 + r3
```



Incorrect for call-by-reference!

Is this code correct?

RUTGERS

Aliasing Problem

Aliasing: Two variables or source-code names may refer to the same memory location.

Examples:

- formal call-by-reference parameters a and b
- pointers a->f and b->f
- array elements: a(i, j) and a(k, l)

Challenge: When is it safe to keep a variable's value in a register across STORE instructions, i.e., while other STORE instructions are executed?

RUTGERS

Memory Model / Code Shape

- register-register model ← Will use this one from now on
  - → Values that may safely reside in registers are assigned to a unique virtual register (alias analysis)
  - → Register allocation/assignment maps virtual registers to limited set of physical registers
  - → Register allocation/assignment pass needed to make code "work"
- memory-memory model
  - → All values reside in memory, and are only kept in registers as briefly as possible (load operands from memory, perform computation, store result into memory)
  - → Register allocation/assignment has to try to identify cases where values can be safely kept in registers
  - → Safety verification is hard at the low levels of program abstraction
  - → Even without register allocation/assignment, code will "work"

Next topic

More instruction scheduling

EaC 12.1 – 12.3