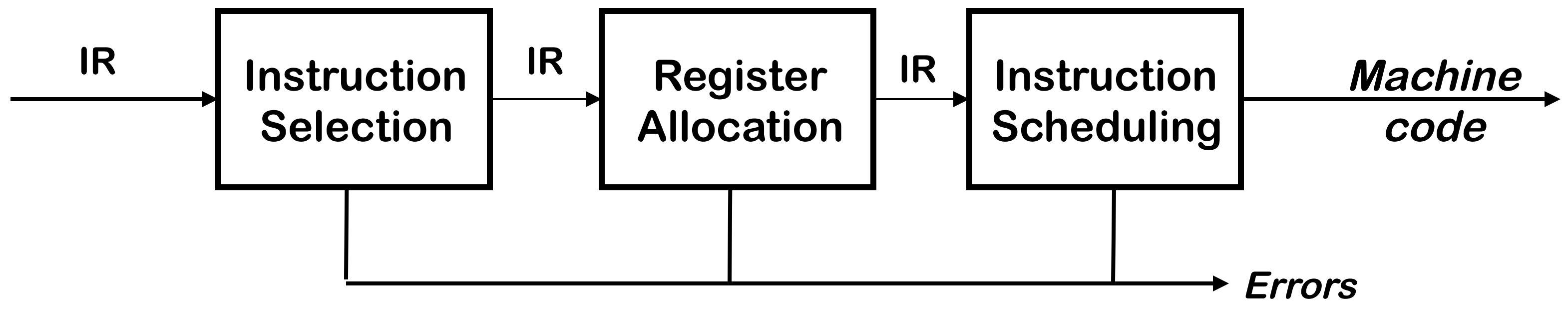
RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Instruction Scheduling and Introduction to Lexical Analysis

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

Review: The Back End



Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces

Automation has been *less* successful in the back end

RUTGERS

Local Instruction Scheduling

Readings: EaC 12.1-12.3

Local: within single basic block
Global: across entire procedure

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 (or power requirements)
- Static schedule – insert NOPs to preserve correctness
- Dynamic schedule – hardware pipeline stalls
- Preserve correctness
- 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

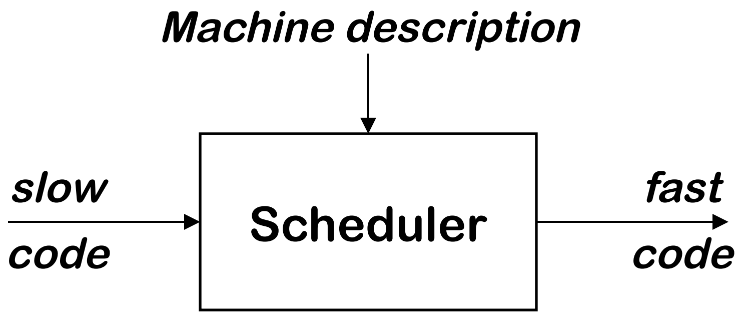
- Reorder instructions to reduce execution time (or power requirements)
- Static schedule – insert NOPs to preserve correctness
- Dynamic schedule – hardware pipeline stalls
- Preserve correctness
- Interactions with other optimizations (register allocation!)
- Note: code shape contains real, not virtual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

Instruction Scheduling (Engineer's View)

The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time

The Concept



The task

- **Produce correct code**
- **Minimize wasted (idle) cycles**
- **Operate efficiently**

Data Dependences (stmt./instr. level)

Dependences ⇒ defined on memory locations / registers and not values

Statement/instruction b depends on statement/instruction a if there exists:

- true of flow dependence
  a writes a location/register that b later reads (RAW conflict)
- anti dependence
  a reads a location/register that b later writes (WAR conflict)
- output dependence
  a writes a location/register that b later writes (WAW conflict)

Dependences define ORDER CONSTRAINTS that need to be respected in order to generate correct code.

true	anti	output
a =	= a	a =
= a	a =	a =

RUTGERS

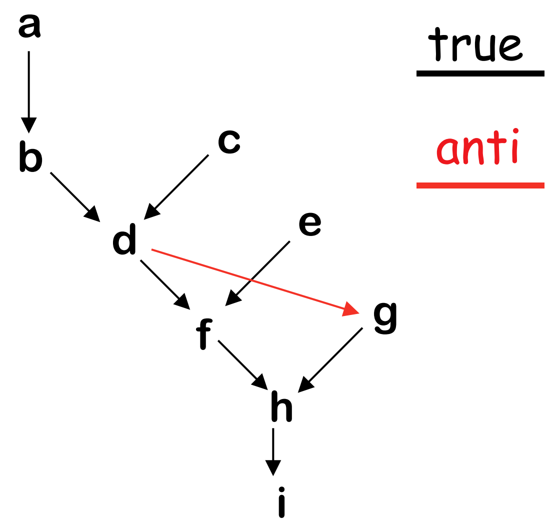
Instruction Scheduling (The Abstract View)

To capture properties of the code, build a <u>dependence graph</u> *G*

- Nodes $n \in G$ are operations with *type(n)* and *delay(n)*
- An edge $e = (n_1,n_2) \in G$ iff n_2 uses the result of n_1

```
a:  loadAI   r0,@w   ⇒ r1
b:  add      r1,r1   ⇒ r1
c:  loadAI   r0,@x   ⇒ r2
d:  mult     r1,r2   ⇒ r1
e:  loadAI   r0,@y   ⇒ r3
f:  mult     r1,r3   ⇒ r1
g:  loadAI   r0,@z   ⇒ r2
h:  mult     r1,r2   ⇒ r1
i:  storeAI  r1      ⇒ r0,@w
```

The Code



The Dependence Graph

(all output dependences are covered, i.e., are satisfied through other dependences)

RUTGERS

Instruction Scheduling (Definitions)

A correct schedule S maps each $n \in N$ into a non-negative integer representing its cycle number such that

1. $S(n) \geq 0$, for all $n \in N$, obviously
2. If $(n_1,n_2) \in E$, $S(n_1) + delay(n_1) \leq S(n_2)$
3. For each type t, there are no more operations of type t in any cycle than the target machine can issue

The length of a schedule S, denoted $L(S)$, is

$L(S) = max_{n \in N} (S(n) + delay(n))$

The goal is to find the shortest possible correct schedule.

S is time-optimal if $L(S) \leq L(S_1)$, for all other schedules S_1

A schedule might also be optimal in terms of registers, power, or space....

Instruction Scheduling (What's so difficult?)

Critical Points

- All operands must be available
- Multiple operations can be *ready*
- Moving operations can lengthen register lifetimes
- Placing uses near definitions can shorten register lifetimes
- Operands can have multiple predecessors

Together, these issues make scheduling *hard* (NP-Complete)

Local scheduling is the simpler case

- Restricted to straight-line code (single basic block)
- Consistent and predictable latencies

RUTGERS

Instruction Scheduling

The big picture

1. Build a dependence graph, *P*
2. Compute a <u>*priority function*</u> over the nodes in *P*
3. Use list scheduling to construct a schedule, one cycle at a time (can only issue/schedule at most one instructions per cycle)
    a. Use a queue of operations that are ready
    b. At each cycle
        I. Choose a ready operation and schedule it
        II. Update the ready queue

Local list scheduling

- The dominant algorithm for twenty years
- A greedy, heuristic, local technique

RUTGERS

Local (Forward) List Scheduling

```
Cycle ← 1
Ready ← leaves of P
Active ← Ø

while (Ready ∪ Active ≠ Ø)
   if (Ready ≠ Ø) then
      remove an op from Ready
      S(op) ← Cycle
      Active ← Active ∪ op

   Cycle ← Cycle + 1

   for each op ∈ Active
        if (S(op) + delay(op) ≤ Cycle) then
           remove op from Active
           for each successor s of op in P
                if (s is ready) then
                   Ready ← Ready ∪ s
```

Removal in priority order

op has completed execution

If successor's operands are ready, put it on Ready

RUTGERS

Scheduling Example

Operation	Cycles
load	3
loadI	1
loadAI	3
store	3
storeAI	3
add	1
mult	2
fadd	1
fmult	2
shift	1
branch	0 to 8

- Loads & stores may or may not block
  - Non-blocking ⇒fill those issue slots
- Branches typically have delay slots
  - Fill slots with operations unrelated to branch condition evaluation
  - Percolates branch upward
- Branch Prediction may hide branch latencies (hardware feature)

Build a simple local scheduler (basic block)

- non-blocking loads & stores
- different latencies load/store vs. arith. etc. operations
- different heuristics
- forward / backward scheduling

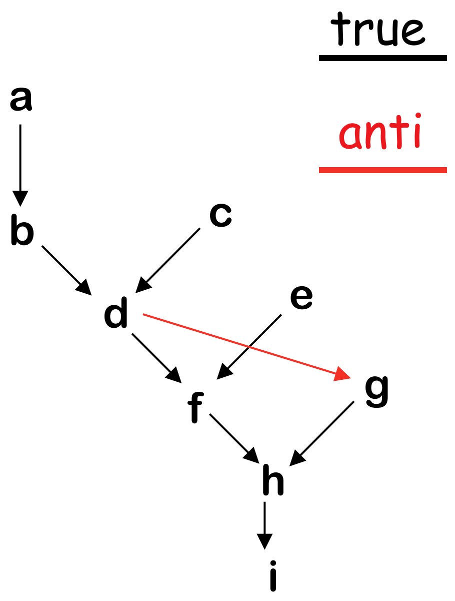
Scheduling Example

1. Build the dependence graph

```
1   a:  loadAI   r0,@w   ⇒ r1
4   b:  add      r1,r1   ⇒ r1
5   c:  loadAI   r0,@x   ⇒ r2
8   d:  mult     r1,r2   ⇒ r1
9   e:  loadAI   r0,@y   ⇒ r3
12  f:  mult     r1,r3   ⇒ r1
13  g:  loadAI   r0,@z   ⇒ r2
16  h:  mult     r1,r2   ⇒ r1
18  i:  storeAI  r1      ⇒ r0,@w
21
```

The Code

⇒ 20 cycles



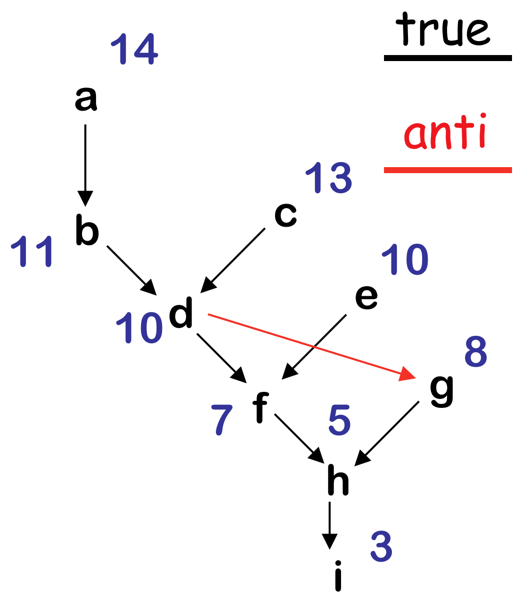
The Dependence Graph

Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r3
f:  mult     r1,r3  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



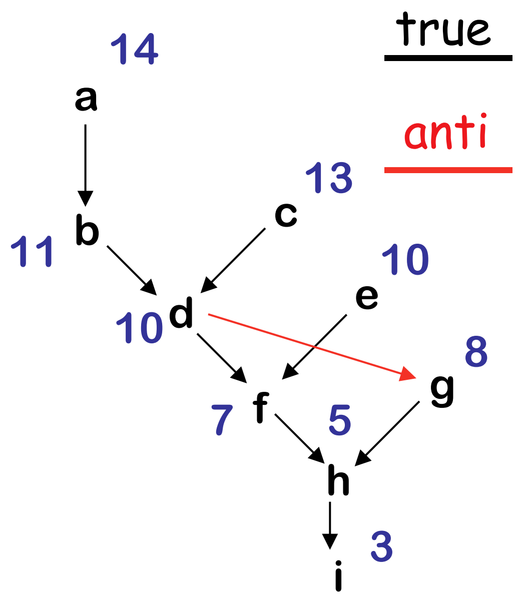
The Dependence Graph

Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r3
f:  mult     r1,r3  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



The Dependence Graph

Note: Here we assume that operation has to finish to satisfy an anti dependence.
Our ILOC simulator takes only one cycle to satisfy an anti dependence since read-stage is executed before write stage

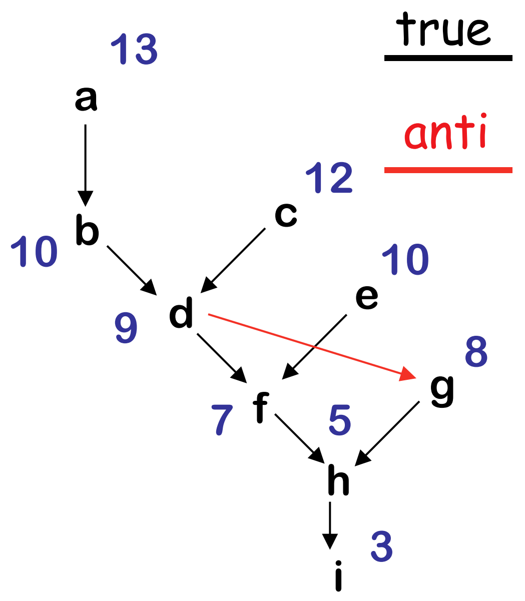
RUTGERS

Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

```
a:   loadAI    r0,@w   ⇒ r1
b:   add       r1,r1   ⇒ r1
c:   loadAI    r0,@x   ⇒ r2
d:   mult      r1,r2   ⇒ r1
e:   loadAI    r0,@y   ⇒ r3
f:   mult      r1,r3   ⇒ r1
g:   loadAI    r0,@z   ⇒ r2
h:   mult      r1,r2   ⇒ r1
i:   storeAI   r1      ⇒ r0,@w
```

The Code



The Dependence Graph

Note: Here we assume that operation has to finish to satisfy an anti dependence.
Our ILOC simulator **takes only one cycle to satisfy an anti dependence** since read-stage is executed before write stage (EaC).

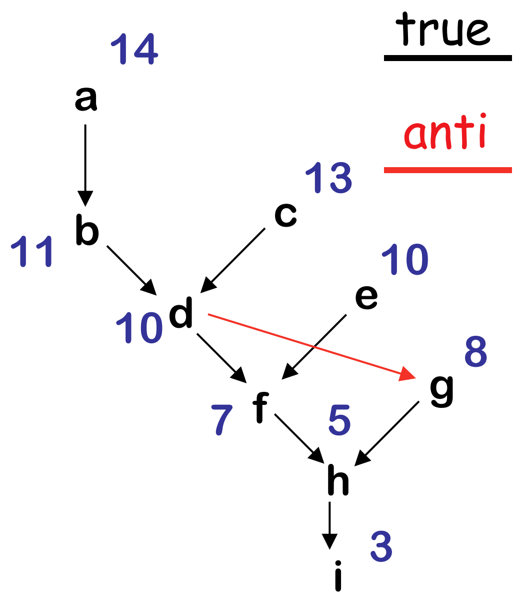
RUTGERS

Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

```
a:  loadAI   r0,@w   ⇒ r1
b:  add      r1,r1   ⇒ r1
c:  loadAI   r0,@x   ⇒ r2
d:  mult     r1,r2   ⇒ r1
e:  loadAI   r0,@y   ⇒ r3
f:  mult     r1,r3   ⇒ r1
g:  loadAI   r0,@z   ⇒ r2
h:  mult     r1,r2   ⇒ r1
i:  storeAI  r1      ⇒ r0,@w
```

The Code



The Dependence Graph

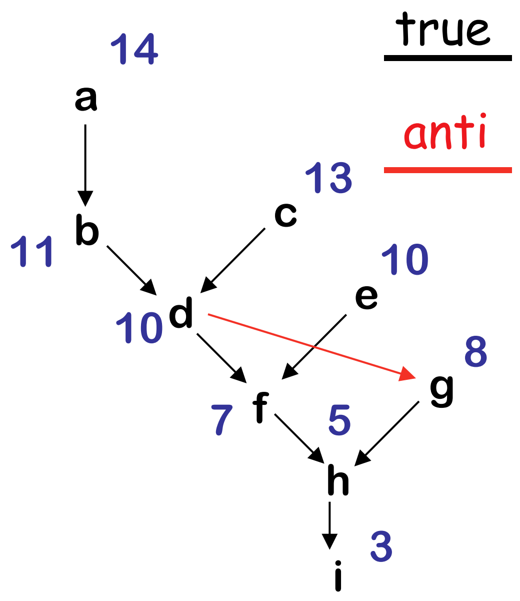
RUTGERS

Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

```
1    a:  loadAI    r0,@w   ⇒ r1
2    c:  loadAI    r0,@x   ⇒ r2
3    e:  loadAI    r0,@y   ⇒ r3
4    b:  add       r1,r1   ⇒ r1
5    d:  mult      r1,r2   ⇒ r1
7    g:  loadAI    r0,@z   ⇒ r2
8    f:  mult      r1,r3   ⇒ r1
10   h:  mult      r1,r2   ⇒ r1
12   i:  storeAI   r1      ⇒ r0,@w
15
```

The Code



The Dependence Graph

⇒ 14 cycles

Our ILOC simulator takes only one cycle to satisfy an anti dependence

RUTGERS

More on Scheduling

Forward list scheduling
- start with available ops
- work forward
- ready ⇒ all operands available

Backward list scheduling
- start with no successors
- work backward
- ready ⇒ latency covers operands

Different heuristics (forward) based on Precedence/Dependence Graph
1. Longest latency weighted path to root (⇒ critical path)
2. Highest latency instructions (⇒ more overlap)
3. Most immediate successors (⇒ create more candidates)
4. Most descendents (⇒ create more candidates)
5. …

Interactions with register allocation
- perform dynamic register renaming (⇒ may require spill code)
- move life ranges around (⇒ may remove or require spill code)
- …

Next class

Lexical Analysis

Read EaC: Chapters 2.1 – 2.5