Compiler Backend

Martin Kellogg

Course Announcements

- PA3 deadline is **today** (AoE)
  - How is it going?

Course Announcements

- PA3 deadline is **today** (AoE)
  - How is it going?
- I will hold an extra office hour today **11:30-12:30** for those who would like to see a PA3 test case
  - You may also be able to catch me either between 2 and 2:30 in my office or at the CS seminar this afternoon, but no promises

Course Announcements

- PA3 deadline is **today** (AoE)
  - How is it going?
- I will hold an extra office hour today **11:30-12:30** for those who would like to see a PA3 test case
  - You may also be able to catch me either between 2 and 2:30 in my office or at the CS seminar this afternoon, but no promises
- PA4 is still Coming Soon™ (I'm actually trying to get this autograder right the first time…)

Course Announcements

- PA3 deadline is **today** (AoE)
  - How is it going?
- I will hold an extra office hour today **11:30-12:30** for those who would like to see a PA3 test case
  - You may also be able to catch me either between 2 and 2:30 in my office or at the CS seminar this afternoon, but no promises
- PA4 is still Coming Soon™ (I'm actually trying to get this autograder right the first time…)
- Note that PA4c1's specification is TAC -> TAC
  - that is, the input is also a `.cl-tac` file
  - PA4c1 is due April 28, and is mostly optional

Agenda

- Interprocedural optimizations (and analysis)
- Compiler backend overview
  - Instruction selection
  - Instruction scheduling
  - Register allocation basics
- Next time:
  - Deep dive into register allocation

Agenda

- **Interprocedural optimizations (and analysis)**
- Compiler backend overview
  - Instruction selection
  - Instruction scheduling
  - Register allocation basics
- Next time:
  - Deep dive into register allocation

Interprocedural Optimizations

Interprocedural Optimizations

- Dividing the program up into procedures give one big benefit: **separate compilation**

Interprocedural Optimizations

- Dividing the program up into procedures give one big benefit: **separate compilation**
  - we can also optimize each procedure **independently** using global analyses like those we've discussed today

Interprocedural Optimizations

- Dividing the program up into procedures give one big benefit: **separate compilation**
  - we can also optimize each procedure **independently** using global analyses like those we've discussed today
- However, procedure calls also introduce **significant overhead**
  - pre-call/post-return bookkeeping, prologue/epilogue, jump

Interprocedural Optimizations

- Dividing the program up into procedures give one big benefit: **separate compilation**
  - we can also optimize each procedure **independently** using global analyses like those we've discussed today
- However, procedure calls also introduce **significant overhead**
  - pre-call/post-return bookkeeping, prologue/epilogue, jump
- Calls are also **hard to reason about** in global optimizations
  - compiler doesn't know what will happen inside the call

Interprocedural Optimizations

- Dividing the program up into procedures give one big benefit: **separate compilation**
  - we can also optimize each procedure **independently** using global analyses like those we've discussed today
- However, procedure calls also introduce **significant overhead**
  - pre-call/post-return bookkeeping, prologue/epilogue, jump
- Calls are also **hard to reason about** in global optimizations
  - compiler doesn't know what will happen inside the call
- These downsides of procedure calls motivate ***interprocedural*** (or "***whole-program***") optimizations that span procedure boundaries

Interprocedural Optimizations

- Dividing the program up into procedures give one big benefit: **separate compilation**
  - we can also optimize e
    global analyses like tho
- However, procedure calls
  - pre-call/post-return bo
- Calls are also **hard to reason about** in global optimizations
  - compiler doesn't know what will happen inside the call
- These downsides of procedure calls motivate ***interprocedural*** (or "***whole-program***") optimizations that span procedure boundaries

Today we will take a brief look at two interprocedural optimizations:

- inlining
- tail call optimization

Interprocedural Optimizations: Inlining

- Consider the following procedure:

```
int add(int x, int y):
   return x + y
```

Interprocedural Optimizations: Inlining

- Consider the following procedure:

```
int add(int x, int y):
   return x + y
```

- Imagine generating code for a call to this procedure:

Interprocedural Optimizations: Inlining

- Consider the following procedure:

```
int add(int x, int y):
   return x + y
```

- Imagine generating code for a call to this procedure:
  - the actual procedure body is only one instruction

Interprocedural Optimizations: Inlining

- Consider the following procedure:

```
int add(int x, int y):
   return x + y
```

- Imagine generating code for a call to this procedure:
  - the actual procedure body is only one instruction
  - the prologue and epilogue dominate, we may have to spill registers at call sites, etc.

Interprocedural Optimizations: Inlining

- Consider the following procedure:

```
int add(int x, int y):
   return x + y
```

- Imagine generating code for a call to this procedure:
  - the actual procedure body is only one instruction
  - the prologue and epilogue dominate, we may have to spill registers at call sites, etc.
- Key idea of ***inlining***: for such a procedure call, replace the call with the procedure's body

Inlining: Benefits & Risks

- Inlining is useful when:

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedure to be inlined is much shorter than the prologue/epilogue

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedure to be inlined is much shorter than the prologue/epilogue
  - inlining enables **specialization** (e.g., arguments are constants)

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedure to be inlined is much shorter than the prologue/epilogue
  - inlining enables **specialization** (e.g., arguments are constants)
  - inlining enables **other optimizations** (e.g., part of the body is dead at this particular call site)

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedure to be inlined is much shorter than the prologue/epilogue
  - inlining enables **specialization** (e.g., arguments are constants)
  - inlining enables **other optimizations** (e.g., part of the body is dead at this particular call site)
- Inlining also has risks:

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedure to be inlined is much shorter than the prologue/epilogue
  - inlining enables **specialization** (e.g., arguments are constants)
  - inlining enables **other optimizations** (e.g., part of the body is dead at this particular call site)
- Inlining also has risks:
  - increases **code size** (may overflow instruction cache)

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedure to be inlined is much shorter than the prologue/epilogue
  - inlining enables **specialization** (e.g., arguments are constants)
  - inlining enables **other optimizations** (e.g., part of the body is dead at this particular call site)
- Inlining also has risks:
  - increases **code size** (may overflow instruction cache)
  - increases **register pressure**

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedu
    the prologue/epilogue
  - inlining enables **specialization** (e.g., arguments are constants)
  - inlining enables **other optimizations** (e.g., part of the body is dead at this particular call site)
- Inlining also has risks:
  - increases **code size** (may overflow instruction cache)
  - increases **register pressure**

Note similar benefits and risks to **loop unrolling**. Deciding whether to inline is similarly complicated!

Inlining: Common Heuristics

- In practice, production compilers will use **heuristics** to decide when/if to inline, such as:

Inlining: Common Heuristics

- In practice, production compilers will use **heuristics** to decide when/if to inline, such as:
  - Is this procedure a **leaf** in the call graph?
    - That is, does it not call any other procedures itself?

Inlining: Common Heuristics

- In practice, production compilers will use **heuristics** to decide when/if to inline, such as:
  - Is this procedure a **leaf** in the call graph?
    - That is, does it not call any other procedures itself?
  - Is the callee procedure **significantly smaller** than the calling procedure?

Inlining: Common Heuristics

- In practice, production compilers will use **heuristics** to decide when/if to inline, such as:
  - Is this procedure a **leaf** in the call graph?
    - That is, does it not call any other procedures itself?
  - Is the callee procedure **significantly smaller** than the calling procedure?
  - Static ***call count***: the number of distinct sites that call the procedure.
    - Any procedure called just once is a good inlining candidate.

Inlining: Common Heuristics

- In practice, production compilers will use **heuristics** to decide when/if to inline, such as:
  - Is this procedure a **leaf** in the call graph?
    - That is, does it not call any other procedures itself?
  - Is the callee procedure **significantly smaller** than the calling procedure?
  - Static ***call count***: the number of distinct sites that call the procedure.
    - Any procedure called just once is a good inlining candidate.
  - **Profile data**, such as fraction of execution time (if available)

Interprocedural Optimizations: Tail Calls

Interprocedural Optimizations: Tail Calls

- Consider a procedure that calls another procedure and then immediately returns, like this example:

```
int foo(...):
    ...
    return bar(...)
```

- What will happen as `bar` returns?

Interprocedural Optimizations: Tail Calls

- Consider a procedure that calls another procedure and then immediately returns, like this example:

```
int foo(...):
    ...
    return bar(...)
```

- What will happen as `bar` returns?
  - We will execute the epilogues of `foo` and `bar` **in sequence**, with no intervening instructions

Interprocedural Optimizations: Tail Calls

- Consider a procedure that calls another procedure and then immediately returns, like this example:

```
int foo(...):
    ...
    return bar(...)
```

- What will happen as `bar` returns?
  - We will execute the epilogues of `foo` and `bar` **in sequence**, with no intervening instructions
    - Including many **redundant operations** (e.g., resetting `%rsp`)

Interprocedural Optimizations: Tail Calls

- ***Tail-call elimination*** is an interprocedural optimization that allows a function called as the last instruction in a procedure to return to the calling procedure's caller directly

Interprocedural Optimizations: Tail Calls

- ***Tail-call elimination*** is an interprocedural optimization that allows a function called as the last instruction in a procedure to return to the calling procedure's caller directly
  - This eliminates redundant operations in the epilogue

Interprocedural Optimizations: Tail Calls

- ***Tail-call elimination*** is an interprocedural optimization that allows a function called as the last instruction in a procedure to return to the calling procedure's caller directly
  - This eliminates redundant operations in the epilogue
- This optimization is most important for ***tail-recursive*** procedures that call **themselves** as the last operation in their body
  - e.g., imagine a naive Fibonacci implementation

Interprocedural Optimizations: Tail Calls

- ***Tail-call elimination*** is an interprocedural optimization that allows a function called as the last instruction in a procedure to return to the calling procedure's caller directly
  - This eliminates redundant operations in the epilogue
- This optimization is most important for ***tail-recursive*** procedures that call **themselves** as the last operation in their body
  - e.g., imagine a naive Fibonacci implementation
  - Tail-call elimination often reduces asymptotic stack space requirements from linear to constant for tail-recursive calls

Interprocedural Optimizations: Tail Calls

- ***Tail-call elimination*** is an interprocedural optimization that allows a function called as the last instruction in a procedure to return to the calling procedure's caller directly
  - This eliminates redundant operations in the epilogue
- This optimization is most important for ***tail-recursive*** procedures that call **themselves** as the last operation in their body
  - e.g., imagine a naive Fibonacci implementation
  - Tail-call elimination often reduces asymptotic stack space requirements from linear to constant for tail-recursive calls
- **Functional languages** practically require tail-call elimination

Interprocedural Optimizations: Issues

Interprocedural Optimizations: Issues

- The biggest difficulty in interprocedural optimization is maintaining support for **separate compilation**
  - Traditional "***compilation unit***" is a procedure or file

Interprocedural Optimizations: Issues

- The biggest difficulty in interprocedural optimization is maintaining support for **separate compilation**
  - Traditional "***compilation unit***" is a procedure or file
- If all of the code is definitely available, it suffices to **track dependencies** between procedures from an optimization perspective, and then re-optimize whenever a dependent procedure changes

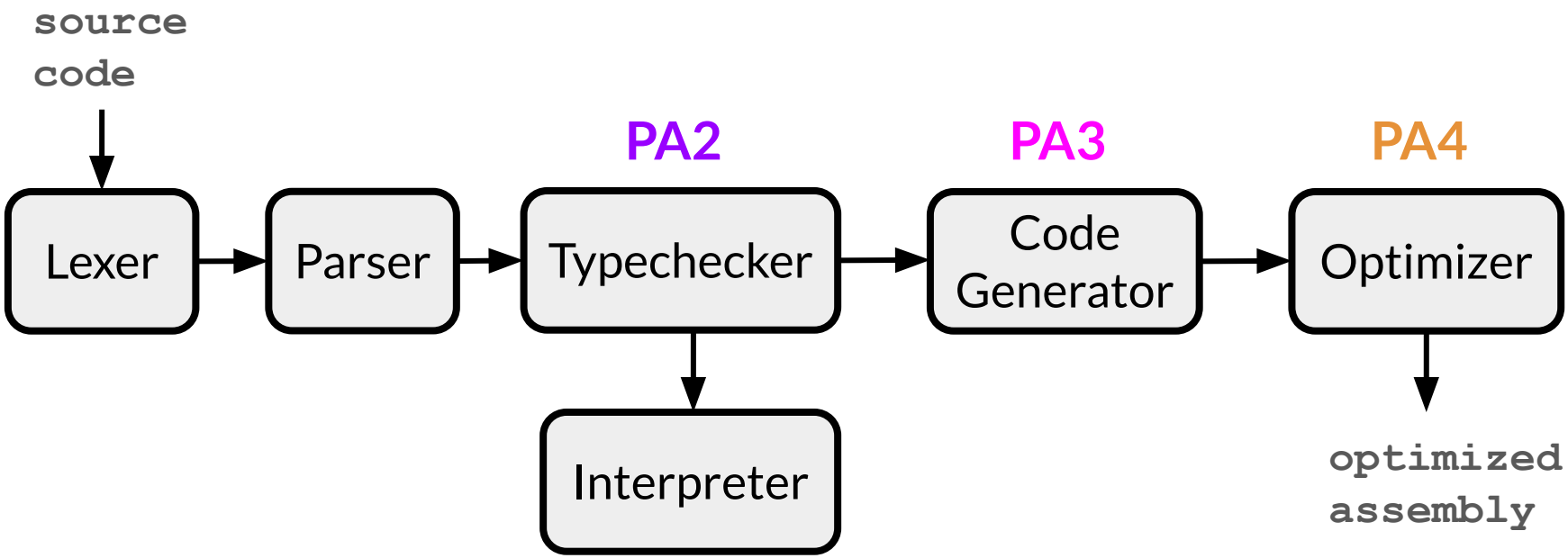
Interprocedural Optimizations: Issues

- The biggest difficulty in interprocedural optimization is maintaining support for **separate compilation**
  - Traditional "***compilation unit***" is a procedure or file
- If all of the code is definitely available, it suffices to **track dependencies** between procedures from an optimization perspective, and then re-optimize whenever a dependent procedure changes
- Alternatively, we can defer interprocedural optimization until ***link time***, when a ***linker*** combines the object files from each compilation unit into a single executable. We'll talk more about this next week.

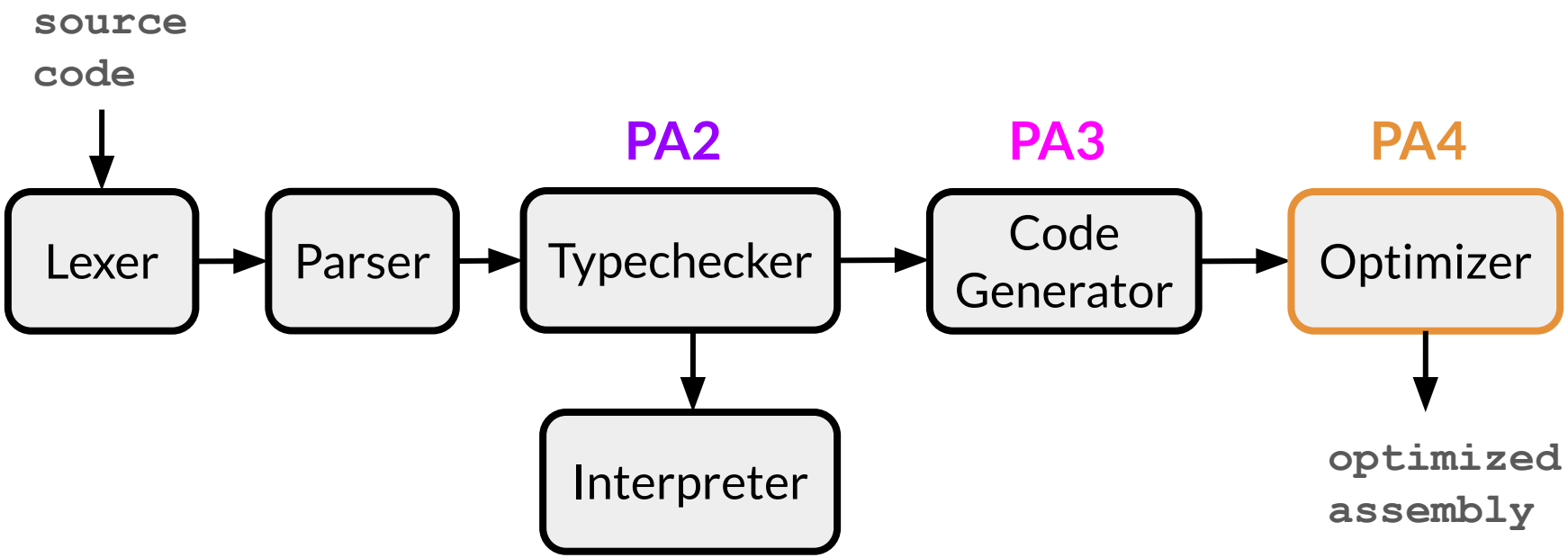
Agenda

- Interprocedural optimizations (and analysis)
- **Compiler backend overview**
  - Instruction selection
  - Instruction scheduling
  - Register allocation basics
- Next time:
  - Deep dive into register allocation

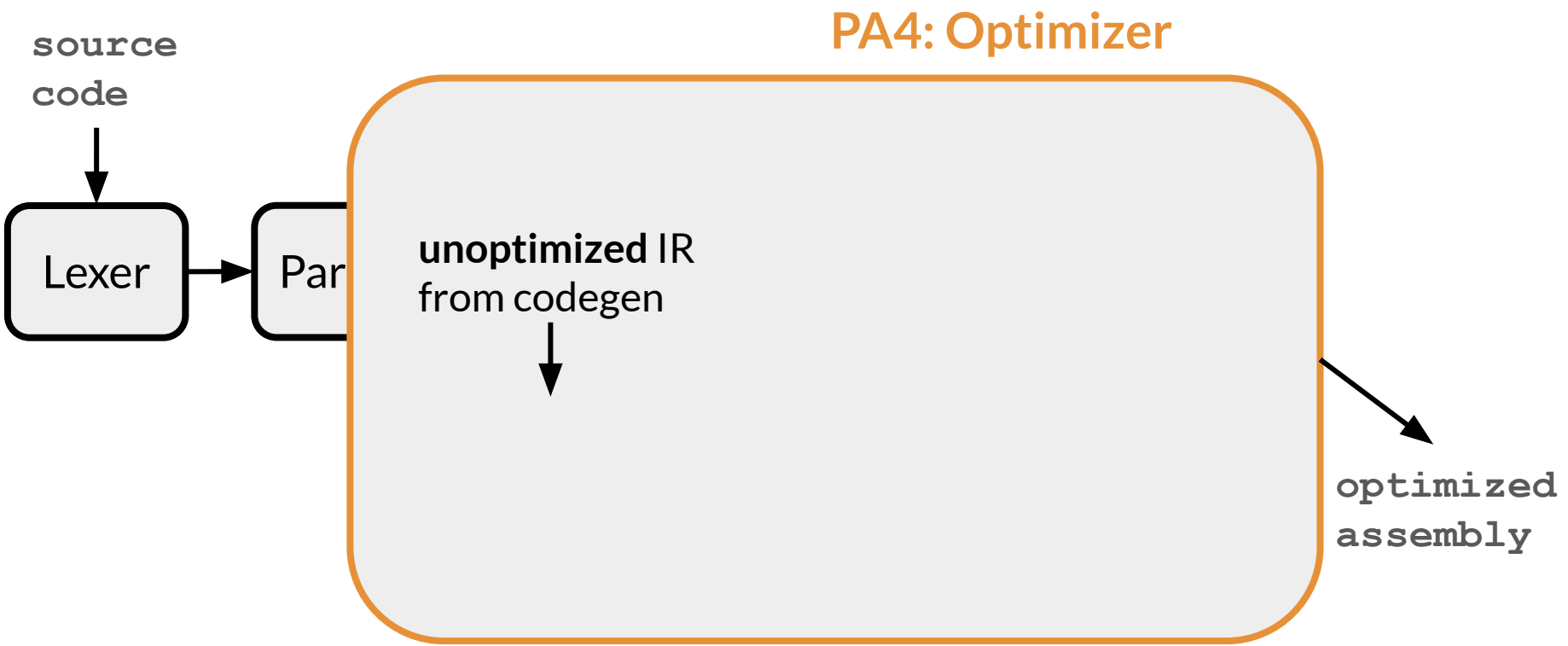
Traditional compiler/interpreter structure



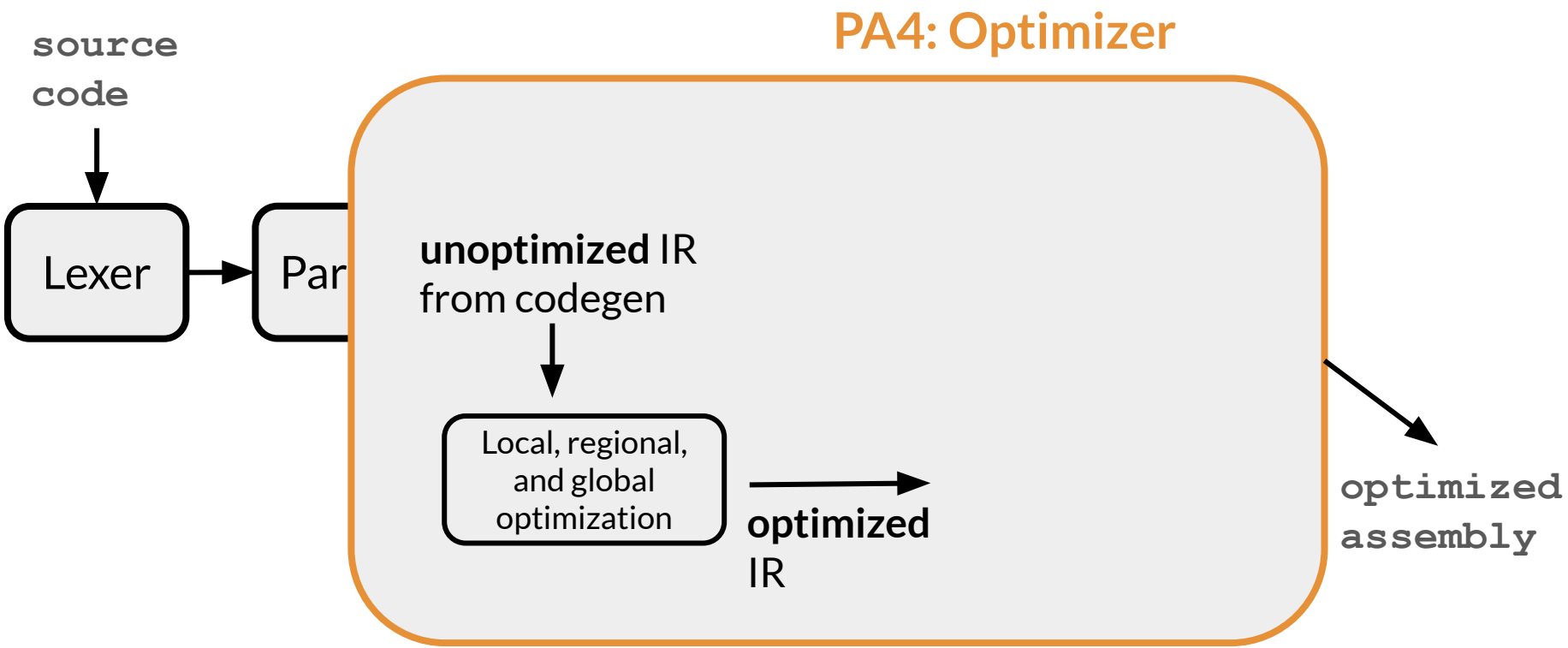
Traditional compiler/interpreter structure



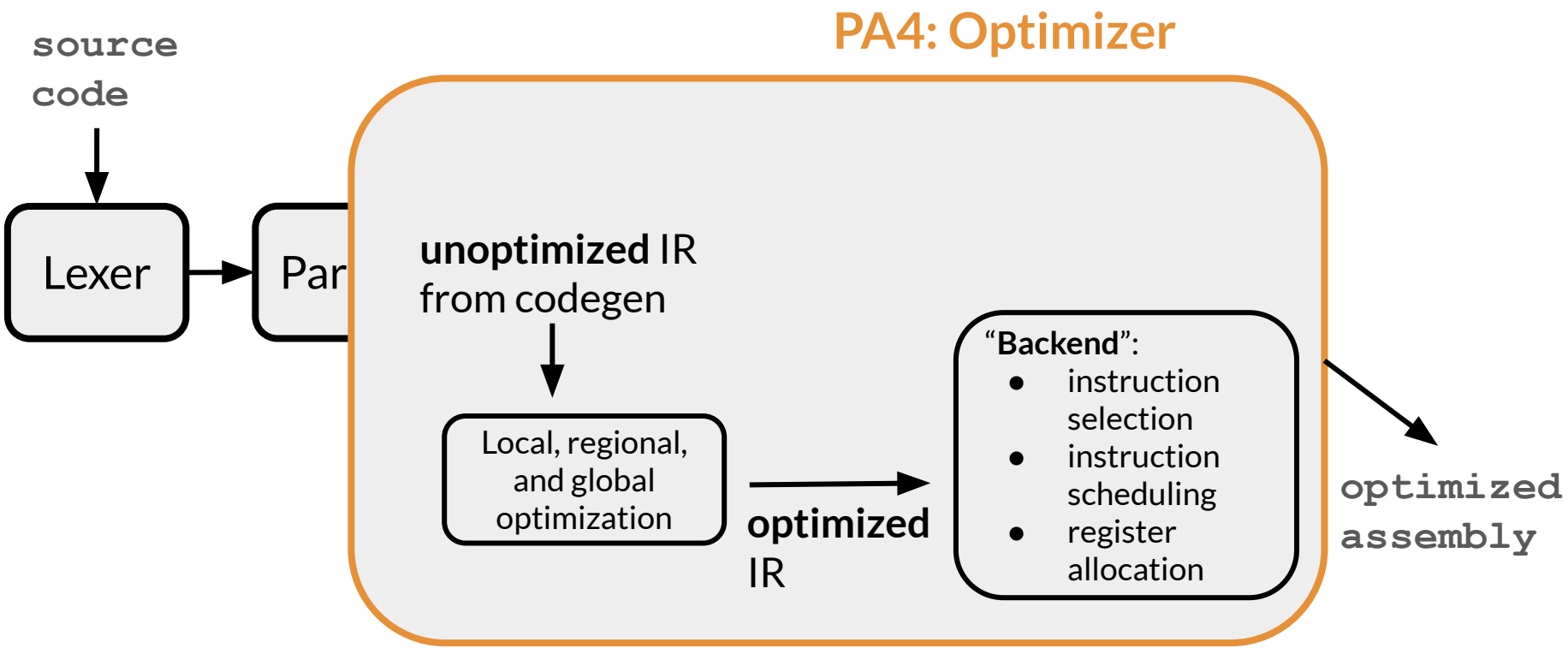
Traditional compiler/interpreter structure



Traditional compiler/interpreter structure



Traditional compiler/interpreter structure



Compiler Backend Overview

- The ***backend*** of an optimizing compiler converts from **optimized IR** into **optimized assembly** for the target machine

Compiler Backend Overview

- The ***backend*** of an optimizing compiler converts from **optimized IR** into **optimized assembly** for the target machine
- Three major subproblems:

Compiler Backend Overview

- The ***backend*** of an optimizing compiler converts from **optimized IR** into **optimized assembly** for the target machine
- Three major subproblems:
  - ***Instruction selection***: map IR into assembly code, combine low-level IR operations into machine instructions (take advantage of addressing modes, etc.)

Compiler Backend Overview

- The ***backend*** of an optimizing compiler converts from **optimized IR** into **optimized assembly** for the target machine
- Three major subproblems:
  - ***Instruction selection***: map IR into assembly code, combine low-level IR operations into machine instructions (take advantage of addressing modes, etc.)
  - ***Instruction scheduling***: Reorder instructions to minimize execution time, hide latencies from processor function units, memory/cache stalls

Compiler Backend Overview

- The ***backend*** of an optimizing compiler converts from **optimized IR** into **optimized assembly** for the target machine
- Three major subproblems:
  - ***Instruction selection***: map IR into assembly code, combine low-level IR operations into machine instructions (take advantage of addressing modes, etc.)
  - ***Instruction scheduling***: Reorder instructions to minimize execution time, hide latencies from processor function units, memory/cache stalls
  - ***Register allocation***: Map abstract registers to actual registers, add code to spill values to memory and reload as needed, etc

Compiler Backend Overview

- The ***backend*** of an optimizin
  IR into **optimized assembly**
- Three major subproblems:
  - ***Instruction selection***: ma
    low-level IR operations
    advantage of addressing modes, etc.)
  - ***Instruction scheduling***: Reorder instructions to minimize execution time, hide latencies from processor function units, memory/cache stalls
  - ***Register allocation***: Map abstract registers to actual registers, add code to spill values to memory and reload as needed, etc

Today we will briefly cover instruction selection + scheduling, and then start on register allocation. Wednesday, we'll cover register allocation in more detail.

Instruction Selection: Overview

Instruction Selection: Overview

- Goal: map **IR to assembly** code
  - assuming known storage layout and code shape

Instruction Selection: Overview

- Goal: map **IR to assembly** code
  - assuming known storage layout and code shape
- Problem to solve: given the low-level IR, there are **many possible code sequences** that implement it correctly

Instruction Selection: Overview

- Goal: map **IR to assembly** code
  - assuming known storage layout and code shape
- Problem to solve: given the low-level IR, there are **many possible code sequences** that implement it correctly
  - e.g., how many ways can we “set %rax to zero” in x86-64?

Instruction Selection: Overview

- Goal: map **IR to assembly** code
  - assuming known storage layout and code shape
- Problem to solve: given the low-level IR, there are **many possible code sequences** that implement it correctly
  - e.g., how many ways can we “set %rax to zero” in x86-64?

```
movq   $0, %rax          salq   64, %rax
subq   %rax, %rax        shrq   64, %rax
xorq   %rax, %rax        imulq  $0, %rax
```

Instruction Selection: Overview

- Goal: map **IR to assembly** code
  - assuming known storage layout and code shape
- Problem to solve: given the low-level IR, there are **many possible code sequences** that implement it correctly
  - e.g., how many ways can we “set %rax to zero” in x86-64?

```
movq   $0, %rax          salq   64, %rax
subq   %rax, %rax        shrq   64, %rax
xorq   %rax, %rax        imulq  $0, %rax
```

- Many machine instructions also do **several things at once** (e.g., register arithmetic and effective address calculation in a movq)

Instruction Selection: Criteria

Instruction Selection: Criteria

- Several possibilities: **fastest**, smallest, minimize power consumption (e.g., don't use a functional unit if leaving it powered-down is a win), reduce memory traffic, etc.

Instruction Selection: Criteria

- Several possibilities: **fastest**, smallest, minimize power consumption (e.g., don't use a functional unit if leaving it powered-down is a win), reduce memory traffic, etc.
  - Typically we want "fastest", but depends on opt. target

Instruction Selection: Criteria

- Several possibilities: **fastest**, smallest, minimize power consumption (e.g., don't use a functional unit if leaving it powered-down is a win), reduce memory traffic, etc.
  - Typically we want "fastest", but depends on opt. target
- Sometimes **not obvious**
  - e.g., if one of the function units in the processor is idle and we can select an instruction that uses that unit, it effectively executes for free, even if that instruction wouldn't be chosen normally
    - (Some interaction with scheduling here...)
    - (and it might consume extra power, so bad if that matters)

Instruction Selection: Tree Pattern Matching

- One algorithm for instruction selection is **tree pattern matching**

Instruction Selection: Tree Pattern Matching

- One algorithm for instruction selection is **tree pattern matching**
- Goal: find a sequence of machine instructions that perform the computation described by the program's IR code

Instruction Selection: Tree Pattern Matching

- One algorithm for instruction selection is **tree pattern matching**
- Goal: find a sequence of machine instructions that perform the computation described by the program's IR code
- Algorithm:

Instruction Selection: Tree Pattern Matching

- One algorithm for instruction selection is **tree pattern matching**
- Goal: find a sequence of machine instructions that perform the computation described by the program's IR code
- Algorithm:
  - Describe each machine instruction we want to consider using **same low-level IR** used for program

Instruction Selection: Tree Pattern Matching

- One algorithm for instruction selection is **tree pattern matching**
- Goal: find a sequence of machine instructions that perform the computation described by the program's IR code
- Algorithm:
  - Describe each machine instruction we want to consider using **same low-level IR** used for program
  - ***Tile*** the low-level IR tree with operation (instruction) trees

Instruction Selection: Tree Pattern Matching

- One algorithm for instruction selection is **tree pattern matching**
- Goal: find a sequence of machine instructions that perform the computation described by the program's IR code
- Algorithm:
  - Describe each machine instruction we want to consider using **same low-level IR** used for program
  - ***Tile*** the low-level IR tree with operation (instruction) trees
    - A tiling "***implements***" a tree if it covers every node in the tree and the overlap between any two tiles (trees) is limited to a single compatible node

Instruction Selection: Tiling Generation

- Two common algorithms to generate tilings:

Instruction Selection: Tiling Generation

- Two common algorithms to generate tilings:
  - **Maximal munch**:

Instruction Selection: Tiling Generation

- Two common algorithms to generate tilings:
  - **Maximal munch**:
    - Top-down tree walk

Instruction Selection: Tiling Generation

- Two common algorithms to generate tilings:
  - **Maximal munch**:
    - Top-down tree walk
    - Find largest tile that fits each node

Instruction Selection: Tiling Generation

- Two common algorithms to generate tilings:
  - **Maximal munch**:
    - Top-down tree walk
    - Find largest tile that fits each node
      - Why largest? Heuristic: One instruction that “does more” is likely cheaper than several that do less

Instruction Selection: Tiling Generation

- Two common algorithms to generate tilings:
  - **Maximal munch**:
    - Top-down tree walk
    - Find largest tile that fits each node
      - Why largest? Heuristic: One instruction that “does more” is likely cheaper than several that do less
  - **Dynamic programming**:
    - Assign costs to each node in the tree using a cost model

Instruction Selection: Tiling Generation

- Two common algorithms to generate tilings:
  - **Maximal munch**:
    - Top-down tree walk
    - Find largest tile that fits each node
      - Why largest? Heuristic: One instruction that “does more” is likely cheaper than several that do less
  - **Dynamic programming**:
    - Assign costs to each node in the tree using a cost model
      - cost = cost of individual node + subtree costs

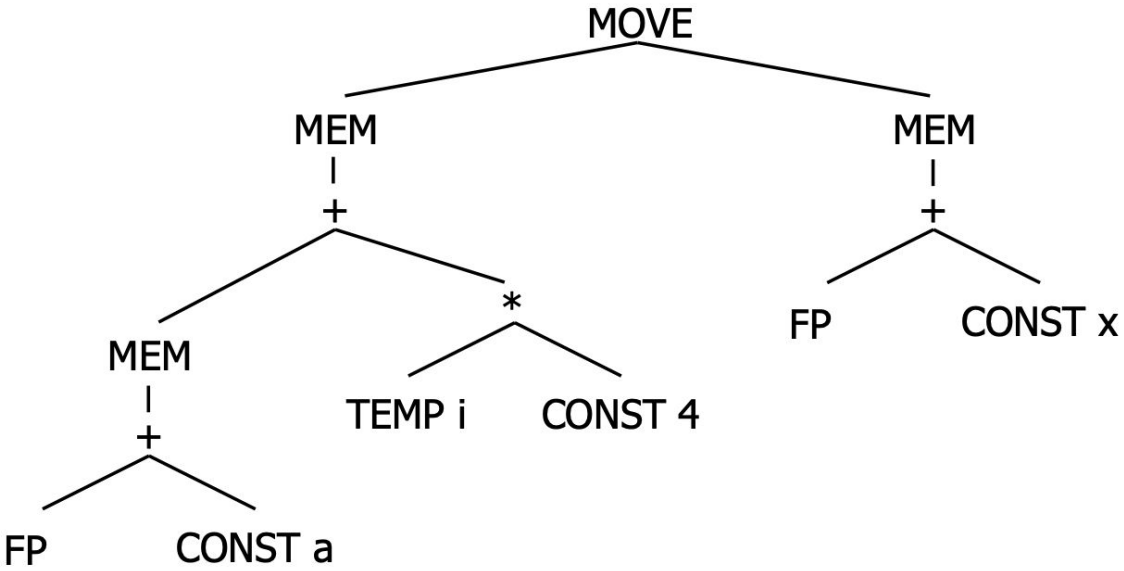
Instruction Selection: Tiling Generation

- Two common algorithms to generate tilings:
  - **Maximal munch**:
    - Top-down tree walk
    - Find largest tile that fits each node
      - Why largest? Heuristic: One instruction that “does more” is likely cheaper than several that do less
  - **Dynamic programming**:
    - Assign costs to each node in the tree using a cost model
      - cost = cost of individual node + subtree costs
    - Try all possible combinations bottom-up, pick cheapest

Instruction Selection: Tiling Generation

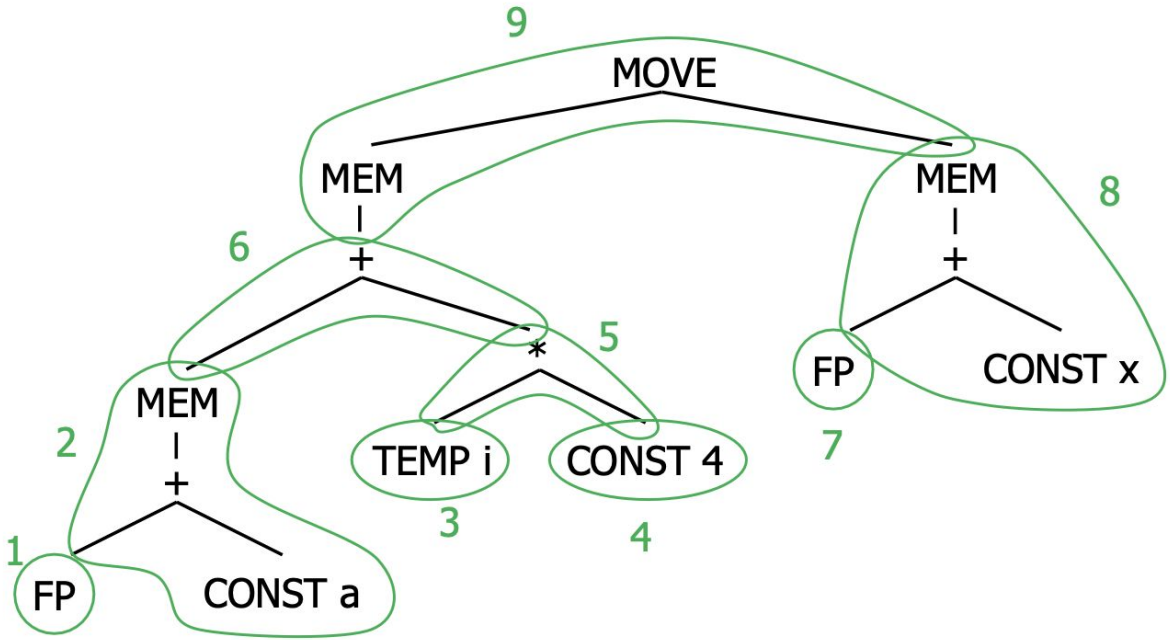
- Two common algorithms to generate tilings:
  - **Maximal munch**:
    - Top-down tree walk
    - Find largest tile that fits each node
      - Why largest? Heuristic: One instruction that “does more” is likely cheaper than several that do less
  - **Dynamic programming**:
    - Assign costs to each node in the tree using a cost model
      - cost = cost of individual node + subtree costs
    - Try all possible combinations bottom-up, pick cheapest
    - Slower, but optimal for a given cost model

Instruction Selection: T.P.M. Example



Tree for `a[i] := x`

Instruction Selection: T.P.M. Example



Tree for `a[i] := x`

Instruction Selection: Generating Code

Given a tiled tree, to generate code:

Instruction Selection: Generating Code

Given a tiled tree, to generate code:

- Do a postorder tree walk with node-dependant order for children

Instruction Selection: Generating Code

Given a tiled tree, to generate code:

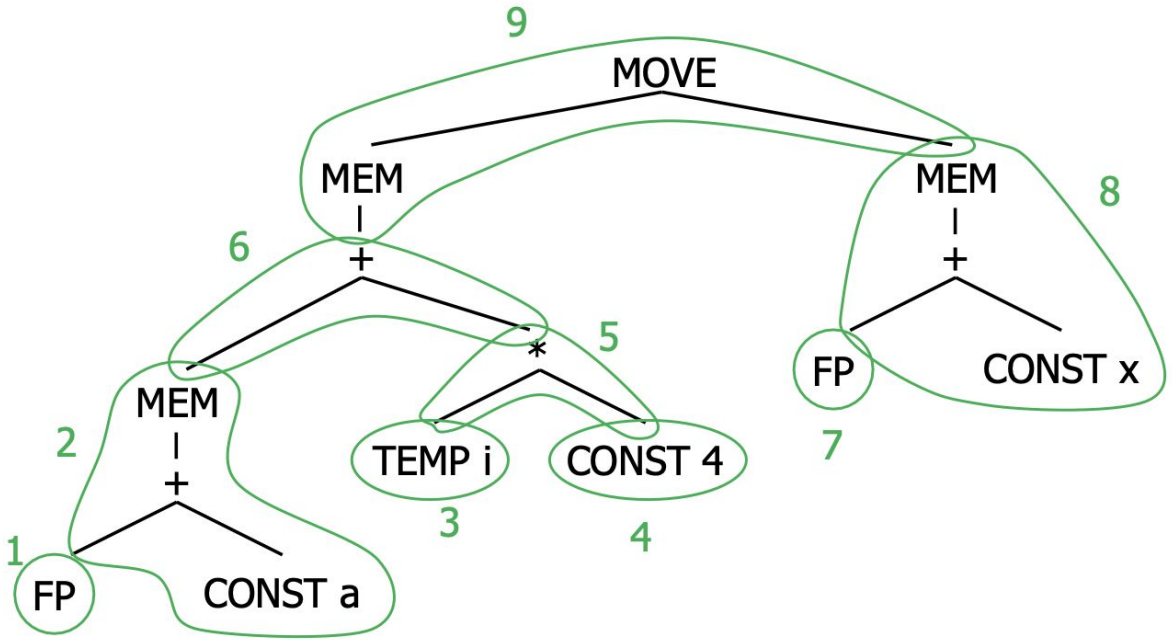
- Do a postorder tree walk with node-dependant order for children
- Each tile corresponds to a code sequence; emit code sequences in order

Instruction Selection: Generating Code

Given a tiled tree, to generate code:

- Do a postorder tree walk with node-dependant order for children
- Each tile corresponds to a code sequence; emit code sequences in order
- Connect tiles by using same register (or temporary) name to tie boundaries together

Instruction Selection: T.P.M. Example



2. LOAD r1 <- M[fp+a_{off}]
4. ADDI r2 <- 4 + r0
5. MUL r2 <- r2 * r_i
6. ADD r1 <- r1 + r2
8. LOAD r2 <- M[fp+x_{off}]
9. STORE M[r1+0] <- r2

Tree for `a[i] := x`

Trivia Break: Computer Science

This American engineer, inventor and science administrator joined MIT in 1919. One of his major scientific works was a differential analyzer, a mechanical analog computer with some digital components that could solve differential equations with as many as 18 independent variables. However, he is best-known for his work in scientific administration: he was vice-president and dean of MIT's School of Engineering, and then became the head of the U.S. Office of Scientific Research and Development during the second world war. He was also instrumental in the founding of the National Science Foundation, and he founded the company that eventually became Raytheon while he was at MIT.

Trivia Break: Computer Science

Vannevar Bush wrote a 1945 article about this hypothetical electromechanical device for interacting with microform documents. Bush envisioned that individuals would compress and store all of their books, records, and communications in this device, "mechanized so that it may be consulted with exceeding speed and flexibility". The individual was supposed to use it as an automatic personal filing system, making the it "an enlarged intimate supplement to his memory." The concept influenced the development of early hypertext systems, eventually leading to the creation of the World Wide Web, and personal knowledge base software.

Instruction Scheduling

Instruction Scheduling

- Goal: **reorder** instructions to minimize execution time given instruction and operand latencies
  - Assume fixed program code at this point

Instruction Scheduling

- Goal: **reorder** instructions to minimize execution time given instruction and operand latencies
  - Assume fixed program code at this point
- Why?

Instruction Scheduling

- Goal: **reorder** instructions to minimize execution time given instruction and operand latencies
  - Assume fixed program code at this point
- Why?
  - Many operations have non-zero latencies
  - Modern machines can issue several operations per cycle
    - Want to take advantage of multiple function units on chip
  - Loads and stores may or may not block
  - Branch costs vary
    - Want to help out the branch predictor, if we can

Instruction Scheduling: Example

Instruction Scheduling: Example: Latencies

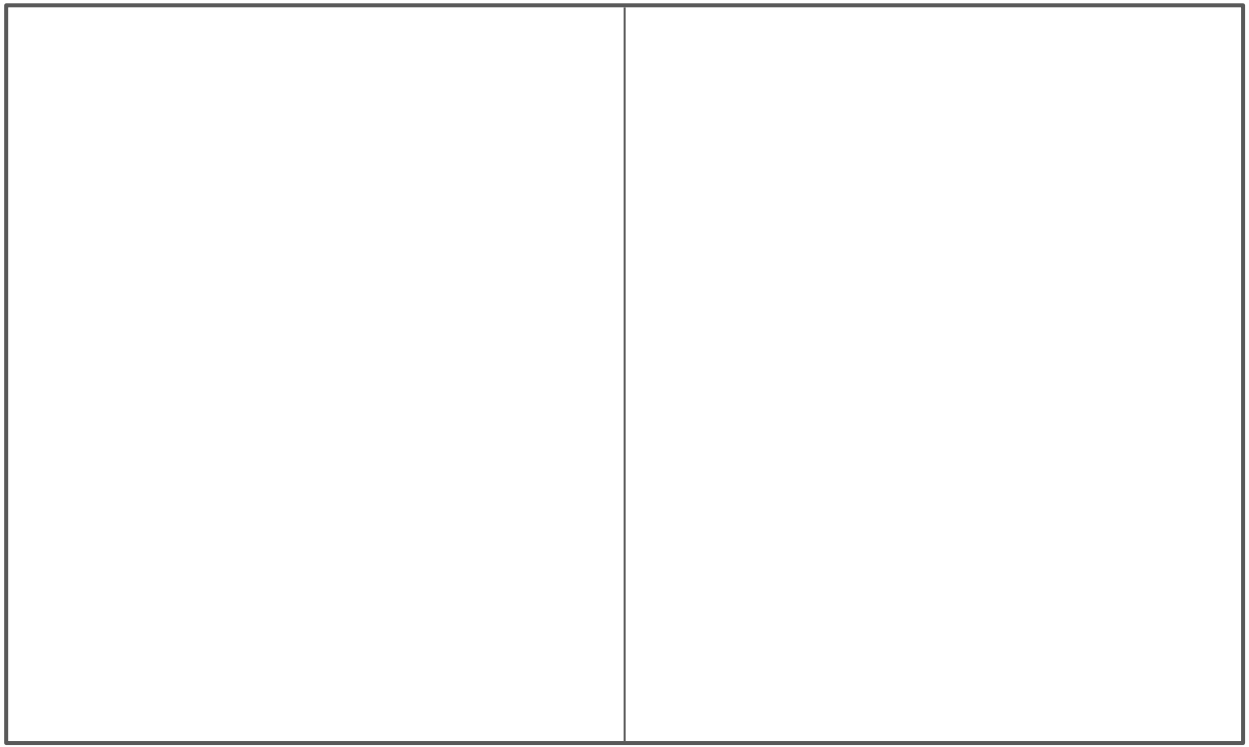
Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1
BRANCH	0 TO 8

(Note that these are just simplified examples; a real machine's behavior will be much more complex!)

Instruction Scheduling: Example

Consider two schedules for the expression `w*2*x*y*z`:

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1
BRANCH	0 TO 8



Instruction Scheduling: Example

Consider two schedules for the expression w*2*x*y*z:

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1
BRANCH	0 TO 8

Simple schedule

```
1  LOAD    r1 <- w
4  ADD     r1 <- r1,r1
5  LOAD    r2 <- x
8  MULT    r1 <- r1,r2
9  LOAD    r2 <- y
12 MULT    r1 <- r1,r2
13 LOAD    r2 <- z
16 MULT    r1 <- r1,r2
18 STORE   w <- r1
21 r1 free
```

2 registers, 20 cycles

Instruction

Quick in-class exercise: turn to someone near you and use your big human brain to come up with a **better schedule**.

Consider two schedules for the expression w*2*x*y*z:

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1
BRANCH	0 TO 8

```
1  LOAD    r1 <- w
4  ADD     r1 <- r1,r1
5  LOAD    r2 <- x
8  MULT    r1 <- r1,r2
9  LOAD    r2 <- y
12 MULT    r1 <- r1,r2
13 LOAD    r2 <- z
16 MULT    r1 <- r1,r2
18 STORE   w <- r1
21 r1 free
     2 registers, 20 cycles
```

Instruction Scheduling: Example

Consider two schedules for the expression w*2*x*y*z:

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1
BRANCH	0 TO 8

Simple schedule

```
1  LOAD    r1 <- w
4  ADD     r1 <- r1,r1
5  LOAD    r2 <- x
8  MULT    r1 <- r1,r2
9  LOAD    r2 <- y
12 MULT    r1 <- r1,r2
13 LOAD    r2 <- z
16 MULT    r1 <- r1,r2
18 STORE   w <- r1
21 r1 free
```

2 registers, 20 cycles

Loads early

```
1  LOAD    r1 <- w
2  LOAD    r2 <- x
3  LOAD    r3 <- y
4  ADD     r1 <- r1,r1
5  MULT    r1 <- r1,r2
6  LOAD    r2 <- z
7  MULT    r1 <- r1,r3
9  MULT    r1 <- r1,r2
11 STORE   w <- r1
14 r1 is free
```

3 registers, 13 cycles

List Scheduling

List Scheduling: Algorithm

- ***List scheduling*** constructs a schedule, one cycle at a time, by doing a (mildly modified) **topological sort** of a ***precedence graph***

List Scheduling: Algorithm

- ***List scheduling*** constructs a schedule, one cycle at a time, by doing a (mildly modified) **topological sort** of a ***precedence graph***
  - in the precedence graph:
    - nodes are operations
    - edges are **dependencies**: if the operation at node n2 uses the result of the operation at n1, then (n1, n2) is an edge

List Scheduling: Algorithm

- ***List scheduling*** constructs a schedule, one cycle at a time, by doing a (mildly modified) **topological sort** of a ***precedence graph***
  - in the precedence graph:
    - nodes are operations
    - edges are **dependencies**: if the operation at node n2 uses the result of the operation at n1, then (n1, n2) is an edge
  - use ***priority*** to choose among ready (=in-degree 0) operations
    - priority = number of cycles on critical path to the end (usually)

List Scheduling: Algorithm

- ***List scheduling*** constructs a schedule, one cycle at a time, by doing a (mildly modified) **topological sort** of a ***precedence graph***
  - in the precedence graph:
    - nodes are operations
    - edges are **dependencies**: if the operation at node n2 uses the result of the operation at n1, then (n1, n2) is an edge
  - use ***priority*** to choose among ready (=in-degree 0) operations
    - priority = number of cycles on critical path to the end (usually)
- Note: may need to rename registers to avoid false dependencies and conflicts

List Scheduling: Algorithm

- ***List scheduling*** constructs a (mildly modified) **topolog**
  - in the precedence gra
    - nodes are operati
    - edges are **depend** the result of the c
  - use ***priority*** to choose
    - priority = number (usually)
- Note: may need to renam and conflicts

Full algorithm if you want to implement it yourself:

```
P = precedence graph;
Cycle = 1; Ready = leaves of P; Active = empty;
while (Ready and/or Active are not empty)
 if (Ready is not empty)
  remove an op from Ready;
  S(op) = Cycle;
  Active = Active È op;
 Cycle++;
 for each op in Active
  if (S(op) + delay(op) <= Cycle)
   remove op from Active;
   for each successor s of op in P
    if (s is ready – i.e., all operands available)
     add s to Ready
```

List Scheduling: Example

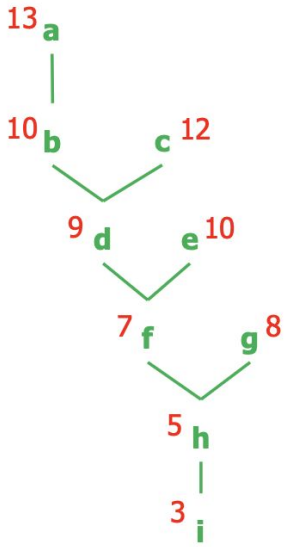
Code

```
a  LOAD     r1 <- w
b  ADD      r1 <- r1,r1
c  LOAD     r2 <- x
d  MULT     r1 <- r1,r2
e  LOAD     r2 <- y
f  MULT     r1 <- r1,r2
g  LOAD     r2 <- z
h  MULT     r1 <- r1,r2
i  STORE    w <- r1
```

List Scheduling: Example

Code

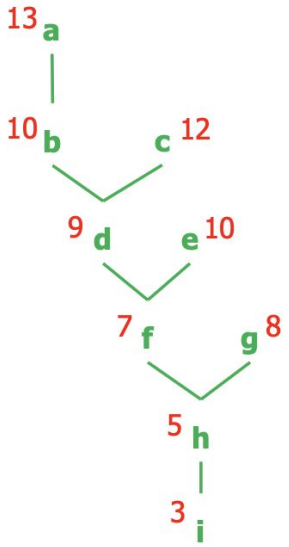
a	LOAD	r1 <- w
b	ADD	r1 <- r1,r1
c	LOAD	r2 <- x
d	MULT	r1 <- r1,r2
e	LOAD	r2 <- y
f	MULT	r1 <- r1,r2
g	LOAD	r2 <- z
h	MULT	r1 <- r1,r2
i	STORE	w <- r1



List Scheduling: Example

Code

```
a  LOAD     r1 <- w
b  ADD      r1 <- r1,r1
c  LOAD     r2 <- x
d  MULT     r1 <- r1,r2
e  LOAD     r2 <- y
f  MULT     r1 <- r1,r2
g  LOAD     r2 <- z
h  MULT     r1 <- r1,r2
i  STORE    w <- r1
```



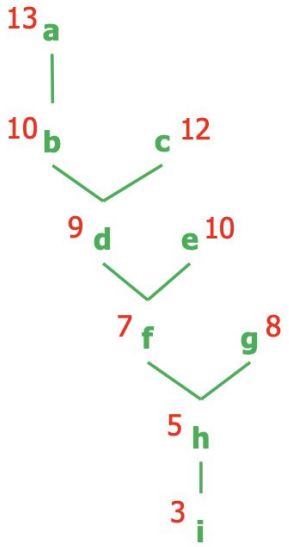
cycle: 1
ready: a c e g
active: –

List Scheduling: Example

#	instr	done
1	a LOAD	4

Code

a	LOAD	r1 <- w
b	ADD	r1 <- r1,r1
c	LOAD	r2 <- x
d	MULT	r1 <- r1,r2
e	LOAD	r2 <- y
f	MULT	r1 <- r1,r2
g	LOAD	r2 <- z
h	MULT	r1 <- r1,r2
i	STORE	w <- r1



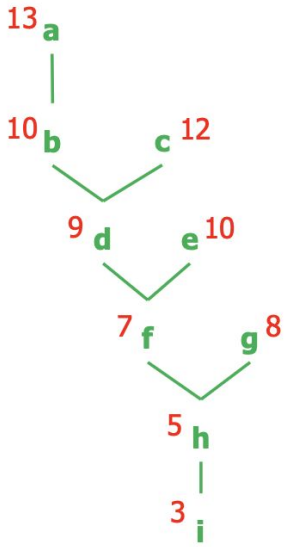
cycle: 1
ready: a c e g
active: –

List Scheduling: Example

#	instr	done
1	a LOAD	4
2	c LOAD	5

Code

a	LOAD	r1 <- w
b	ADD	r1 <- r1,r1
c	LOAD	r2 <- x
d	MULT	r1 <- r1,r2
e	LOAD	r2 <- y
f	MULT	r1 <- r1,r2
g	LOAD	r2 <- z
h	MULT	r1 <- r1,r2
i	STORE	w <- r1

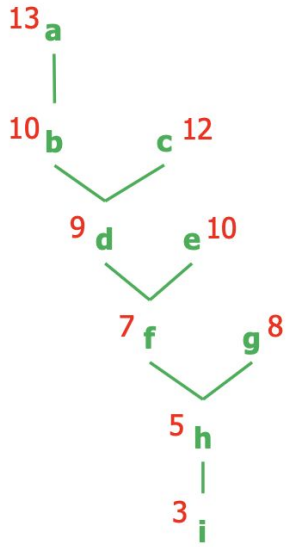


cycle: ~~1~~ 2
ready: ~~a~~ c e g
active: a

List Scheduling: Example

Code

a	LOAD	r1 <- w
b	ADD	r1 <- r1,r1
c	LOAD	r2 <- x
d	MULT	r1 <- r1,r2
e	LOAD	r2 <- y
f	MULT	r1 <- r1,r2
g	LOAD	r2 <- z
h	MULT	r1 <- r1,r2
i	STORE	w <- r1



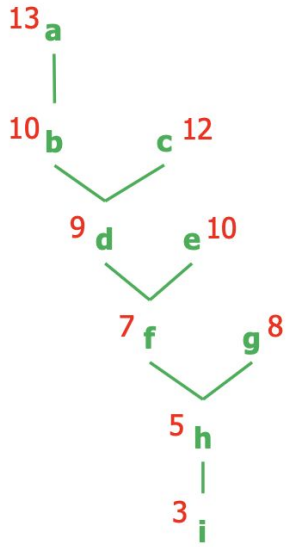
cycle: ~~1~~ ~~2~~ 3
ready: ~~a~~ ~~c~~ e g
active: a c

#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6

List Scheduling: Example

Code

a	LOAD	r1 <- w
b	ADD	r1 <- r1,r1
c	LOAD	r2 <- x
d	MULT	r1 <- r1,r2
e	LOAD	r2 <- y
f	MULT	r1 <- r1,r2
g	LOAD	r2 <- z
h	MULT	r1 <- r1,r2
i	STORE	w <- r1



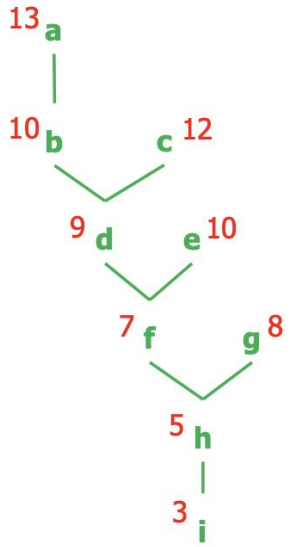
cycle: ~~1~~ ~~2~~ ~~3~~ 4
ready: ~~a~~ ~~c~~ ~~e~~ g b
active: ~~a~~ c e

#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5

List Scheduling: Example

Code

a	LOAD	r1 <- w
b	ADD	r1 <- r1,r1
c	LOAD	r2 <- x
d	MULT	r1 <- r1,r2
e	LOAD	r2 <- y
f	MULT	r1 <- r1,r2
g	LOAD	r2 <- z
h	MULT	r1 <- r1,r2
i	STORE	w <- r1



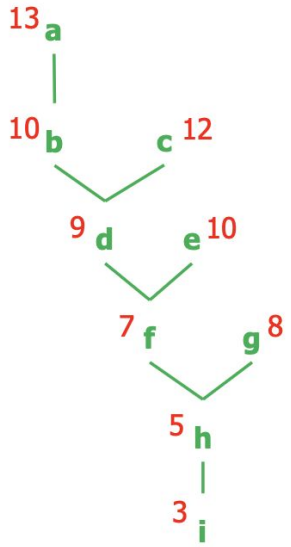
cycle: ~~1~~ ~~2~~ ~~3~~ ~~4~~ 5
ready: ~~a~~ ~~c~~ ~~e~~ g ~~b~~ d
active: ~~a~~ ~~c~~ e ~~b~~

#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7

List Scheduling: Example

Code

a	LOAD	r1 <- w
b	ADD	r1 <- r1,r1
c	LOAD	r2 <- x
d	MULT	r1 <- r1,r2
e	LOAD	r2 <- y
f	MULT	r1 <- r1,r2
g	LOAD	r2 <- z
h	MULT	r1 <- r1,r2
i	STORE	w <- r1



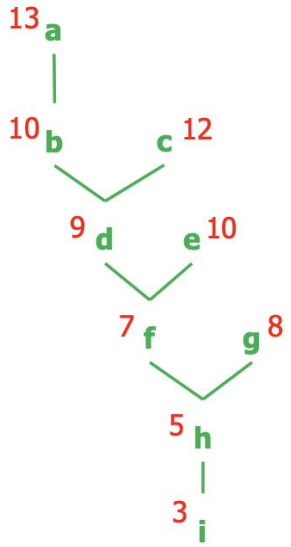
cycle: ~~1 2 3 4 5~~ 6
ready: ~~a c~~ e g ~~b d~~
active: ~~a c~~ e ~~b~~ d

#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7
6	g LOAD	9

List Scheduling: Example

Code

a	LOAD	r1 <- w
b	ADD	r1 <- r1,r1
c	LOAD	r2 <- x
d	MULT	r1 <- r1,r2
e	LOAD	r2 <- y
f	MULT	r1 <- r1,r2
g	LOAD	r2 <- z
h	MULT	r1 <- r1,r2
i	STORE	w <- r1



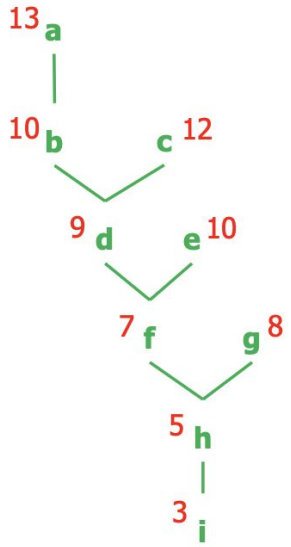
cycle: ~~1 2 3 4 5 6~~ 7
ready: ~~a c e g b d~~ f
active: ~~a c e b d~~ g

#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7
6	g LOAD	9
7	f MULT	9

List Scheduling: Example

Code

a	LOAD	r1 <- w
b	ADD	r1 <- r1,r1
c	LOAD	r2 <- x
d	MULT	r1 <- r1,r2
e	LOAD	r2 <- y
f	MULT	r1 <- r1,r2
g	LOAD	r2 <- z
h	MULT	r1 <- r1,r2
i	STORE	w <- r1



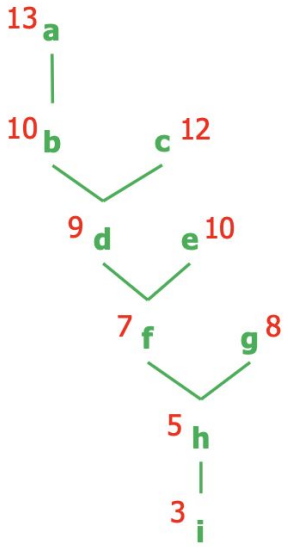
cycle: ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ ~~7~~ 8
ready: ~~a~~ ~~c~~ ~~e~~ ~~g~~ ~~b~~ ~~d~~ ~~f~~
active: ~~a~~ ~~c~~ e ~~b~~ ~~d~~ g f

#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7
6	g LOAD	9
7	f MULT	9
8	– (stall)	

List Scheduling: Example

Code

a	LOAD	r1 <- w
b	ADD	r1 <- r1,r1
c	LOAD	r2 <- x
d	MULT	r1 <- r1,r2
e	LOAD	r2 <- y
f	MULT	r1 <- r1,r2
g	LOAD	r2 <- z
h	MULT	r1 <- r1,r2
i	STORE	w <- r1



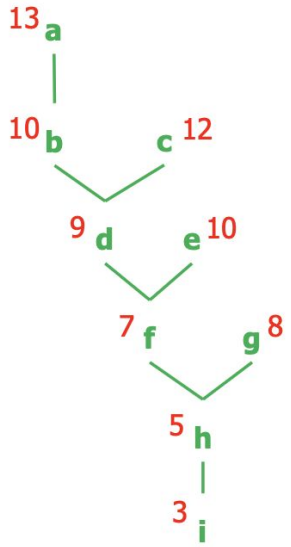
cycle: ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ ~~7~~ ~~8~~ 9
ready: ~~a~~ ~~c~~ ~~e~~ ~~g~~ ~~b~~ ~~d~~ ~~f~~ h
active: ~~a~~ ~~c~~ ~~e~~ ~~b~~ ~~d~~ ~~g~~ ~~f~~

#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7
6	g LOAD	9
7	f MULT	9
8	– (stall)	
9	h MULT	11

List Scheduling: Example

Code

a	LOAD	r1 <- w
b	ADD	r1 <- r1,r1
c	LOAD	r2 <- x
d	MULT	r1 <- r1,r2
e	LOAD	r2 <- y
f	MULT	r1 <- r1,r2
g	LOAD	r2 <- z
h	MULT	r1 <- r1,r2
i	STORE	w <- r1



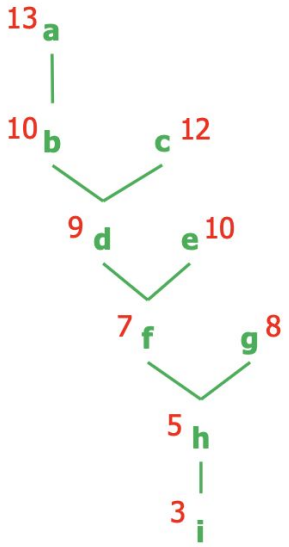
cycle: ~~1 2 3 4 5 6 7 8 9~~ 10
ready: ~~a c e g b d f h~~
active: ~~a c e b d g f~~ h

#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7
6	g LOAD	9
7	f MULT	9
8	– (stall)	
9	h MULT	11
10	– (stall)	

List Scheduling: Example

Code

a	LOAD	r1 <- w
b	ADD	r1 <- r1,r1
c	LOAD	r2 <- x
d	MULT	r1 <- r1,r2
e	LOAD	r2 <- y
f	MULT	r1 <- r1,r2
g	LOAD	r2 <- z
h	MULT	r1 <- r1,r2
i	STORE	w <- r1



cycle: ~~1 2 3 4 5 6 7 8 9 10~~ 11
ready: ~~a c e g b d f h~~ i
active: ~~a c e b d g f h~~

#	instr	done
1	a LOAD	4
2	c LOAD	5
3	e LOAD	6
4	b ADD	5
5	d MULT	7
6	g LOAD	9
7	f MULT	9
8	– (stall)	
9	h MULT	11
10	– (stall)	
11	i STORE	14

List Scheduling: Forwards vs Backwards

- Alternative: **backwards** list scheduling

List Scheduling: Forwards vs Backwards

- Alternative: **backwards** list scheduling
  - Work from the root to the leaves

List Scheduling: Forwards vs Backwards

- Alternative: **backwards** list scheduling
  - Work from the root to the leaves
  - Schedules instructions from end to beginning of the block

List Scheduling: Forwards vs Backwards

- Alternative: **backwards** list scheduling
  - Work from the root to the leaves
  - Schedules instructions from end to beginning of the block
- In practice, production compilers typically **try both** and pick the result that minimizes costs

List Scheduling: Forwards vs Backwards

- Alternative: **backwards** list scheduling
  - Work from the root to the leaves
  - Schedules instructions from end to beginning of the block
- In practice, production compilers typically **try both** and pick the result that minimizes costs
  - **Little extra expense** since the precedence graph and other information can be reused

List Scheduling: Forwards vs Backwards

- Alternative: **backwards** list scheduling
  - Work from the root to the leaves
  - Schedules instructions from end to beginning of the block
- In practice, production compilers typically **try both** and pick the result that minimizes costs
  - **Little extra expense** since the precedence graph and other information can be reused
  - Different directions win in different cases

List Scheduling: Beyond Basic Blocks

- It’s possible to do **regional or even global** list scheduling

List Scheduling: Beyond Basic Blocks

- It's possible to do **regional or even global** list scheduling
  - However, it's much more complicated and gains are often small

List Scheduling: Beyond Basic Blocks

- It's possible to do **regional or even global** list scheduling
  - However, it's much more complicated and gains are often small
  - e.g., if you schedule an entire EBB, you need a heuristic to estimate the cost of "infinite" paths around loops
    - could e.g., assume all loops execute 10 times

List Scheduling: Beyond Basic Blocks

- It's possible to do **regional or even global** list scheduling
  - However, it's much more complicated and gains are often small
  - e.g., if you schedule an entire EBB, you need a heuristic to estimate the cost of "infinite" paths around loops
    - could e.g., assume all loops execute 10 times
- Some compilers use **profiling information** for scheduling
  - i.e., run the code and see how many cycles different schedules actually take

List Scheduling: Beyond Basic Blocks

- It's possible to do **regional or even global** list scheduling
  - However, it's much more complicated and gains are often small
  - e.g., if you schedule an entire EBB, you need a heuristic to estimate the cost of "infinite" paths around loops
    - could e.g., assume all loops execute 10 times
- Some compilers use **profiling information** for scheduling
  - i.e., run the code and see how many cycles different schedules actually take
  - downside: need to actually run the code
    - e.g., where do you get inputs?

Register Allocation

Register Allocation

- IR code typically assumes **infinitely-many registers** are available
  - but real machines only have a small number of registers :(

Register Allocation

- IR code typically assumes **infinitely-many registers** are available
  - but real machines only have a small number of registers :(
- Task of the ***register allocator***: create a mapping from IR's abstract registers to physical registers

Register Allocation

- IR code typically assumes **infinitely-many registers** are available
  - but real machines only have a small number of registers :(
- Task of the ***register allocator***: create a mapping from IR's abstract registers to physical registers
  - Or, if that's not possible, to memory locations
    - this happens when there aren't enough physical registers

Register Allocation

- IR code typically assumes **infinitely-many registers** are available
  - but real machines only have a small number of registers :(
- Task of the ***register allocator***: create a mapping from IR's abstract registers to physical registers
  - Or, if that's not possible, to memory locations
    - this happens when there aren't enough physical registers
  - Insert code to move values between registers and memory if needed ("***spill code***")

Register Allocation

- IR code typically assumes **infinitely-many registers** are available
  - but real machines only have a small number of registers :(
- Task of the ***register allocator***: create a mapping from IR's abstract registers to physical registers
  - Or, if that's not possible, to memory locations
    - this happens when there aren't enough physical registers
  - Insert code to move values between registers and memory if needed ("***spill code***")
    - Typically we will re-run the instruction scheduler if we ever have to spill a register

Register Allocation: PA3 Version

- Even your PA3 implementation should probably have a simple “register allocator” somewhere

Register Allocation: PA3 Version

- Even your PA3 implementation should probably have a simple “register allocator” somewhere
  - my suggestion: “**everything goes in memory**”

Register Allocation: PA3 Version

- Even your PA3 implementation should probably have a simple "register allocator" somewhere
  - my suggestion: "**everything goes in memory**"
  - then you can insert load code before any operation that requires its operands in registers without worrying about what's in those registers when you do

Register Allocation: PA3 Version

- Even your PA3 implementation should probably have a simple "register allocator" somewhere
  - my suggestion: "**everything goes in memory**"
  - then you can insert load code before any operation that requires its operands in registers without worrying about what's in those registers when you do
- However, one of the **first optimizations** that I suggest you implement for PA4 is a simple register allocator

Register Allocation: PA3 Version

- Even your PA3 implementation should probably have a simple "register allocator" somewhere
  - my suggestion: "**everything goes in memory**"
  - then you can insert load code before any operation that requires its operands in registers without worrying about what's in those registers when you do
- However, one of the **first optimizations** that I suggest you implement for PA4 is a simple register allocator
  - avoiding memory operations is **extremely profitable** 🤑🤑🤑

Register Allocation: PA3 Version

- Even your PA3 implementation should probably have a simple "register allocator" somewhere
  - my suggestion: "**everything goes in memory**"
  - then you can insert load code before any operation that requires its operands in registers without worrying about what's in those registers when you do
- However, one of the **first optimizations** that I suggest you implement for PA4 is a simple register allocator
  - avoiding memory operations is **extremely profitable** 🤑🤑🤑
  - I recommend waiting until later in PA4 to try more complex schemes

Register Allocation: Overview

Register Allocation: Overview

- Register allocation: correctness or optimization
  - depends on **memory model**

Register Allocation: Overview

- Register allocation: correctness or optimization
  - depends on **memory model**
- Local register allocation
  - two approaches: top-down and bottom-up

Register Allocation: Overview

- Register allocation: correctness or optimization
  - depends on **memory model**
- Local register allocation
  - two approaches: top-down and bottom-up
- Regional register allocation + complications
  - **liveness analysis** strikes again!

Register Allocation: Overview

- Register allocation: correctness or optimization
  - depends on **memory model**
- Local register allocation
  - two approaches: top-down and bottom-up
- Regional register allocation + complications
  - **liveness analysis** strikes again!
- Global register allocation via **reduction to graph coloring**
  - including a **union-find** algorithm for fun

Course Announcements

- PA3 deadline is **today** (AoE)
  - How is it going?
- I will hold an extra office hour today **11:30-12:30** for those who would like to see a PA3 test case
  - You may also be able to catch me either between 2 and 2:30 in my office or at the CS seminar this afternoon, but no promises
- PA4 is still Coming Soon™ (I'm actually trying to get this autograder right the first time…)
- Note that PA4c1's specification is TAC -> TAC
  - that is, the input is also a `.cl-tac` file
  - PA4c1 is due April 28, and is mostly optional