

# Converting Intermediate Code to Assembly Code Using Declarative Machine Descriptions

**João Dias and Norman Ramsey**

`{dias,nr}@eecs.harvard.edu`

**Division of Engineering and Applied Sciences  
Harvard University**

# Compilers are expensive

Academic research compiler: **10 years** or more

- Standard ML of New Jersey
- Vortex
- Objective Caml
- lcc
- SUIF
- VPO

Industrial research compiler: **50 person-years**

- Jikes RVM
- Phoenix

Therefore: **reuse**

# Architectural change drives reuse

## Evolution is rapid

- Graphics processors
- Sensor networks, cell phones, ...
- Multicore chips

# Reuse compiler by retargeting

**Isolate machine-specific components**

**Retargeting**

- **Rewrite machine-specific components  
(clone and modify)**

# Retargeting by rewriting

**Front end**

**IR**

**Optimizer**

**IR**

**Instruction Selector**

**Machine instructions**

**Register Allocator**

**Assembly code**



# Retargeting by rewriting

Front end

IR

Optimizer

IR

Instruction Selector

Machine instructions

Register Allocator

Assembly code

Machine-dependent components replaced:

- instruction selector
- machine registers
- calling conventions
- stack layout

# Porting requires compiler, machine expertise

Front end

IR

Optimizer

IR

Instruction Selector

Machine instructions

→ Map IR to machine instructions  
(Twig, BURG, etc.)

Register Allocator

→ Specify register interference

Assembly code

# Reuse compiler by retargeting

Isolate machine-specific components

Retargeting

- Rewrite machine-specific components (clone and modify) – but: slow
- **Faster way: write description of target, automatically generate compiler components**

# **Code-generator generators create new problems**

**How do we describe the machine?**

# **Code-generator generators create new problems**

**How do we describe the machine?**

**How do we generate the components?**

# **Classic compiler-generation techniques limit reuse**

## **Domain-specific languages**

- languages for syntax, e.g. lex, yacc
- language for code generator, e.g. BURG

## **Each design is limited**

- **yacc**
  - generates a parser
  - requires both compiler, grammar knowledge
  - cannot be reused
- **BURG**
  - generates an instruction selector
  - requires both compiler, machine knowledge
  - cannot be reused

# To enable reuse, go beyond DSL

## Declarative machine description

- describes only instruction set
- independent of compiler IR
- independent of implementation language
- can be used for other tools  
(binary rewriting, software fault isolation, FPCC)

## But

- requires more sophisticated **analysis** to generate components (our contribution)

# $\lambda$ -RTL semantic description language is simple

## Three elements:

- State
- Instructions
- Effect of instruction on state

# Machine state is modeled by arrays

## Named storage spaces

- Indexed by integers
- Containing mutable cells

**Location (cell)** is  $\$space[offset]$  pair

Describe registers, memory, status words, ...

## Examples:

`$r[3]`

`$m[$sp + 12]`

`$c[0]`

# **An instruction is a state transformer**

**Meaning of instruction is its effects:**

- **Relation on input and output states**

**Simple model, but good enough for uniprocessor**

# An instruction is a state transformer

Meaning of instruction is its **effects**:

- Relation on input and output states

Simple model, but good enough for uniprocessor

Instructions:

- function of operands
- state transformation described by register-transfer lists

`ADD(reg, i8) is $r[reg] := $r[reg] + sx i8`

# Outline

- Introduction
- Describing semantics of instructions
- **Restructuring the compiler**
- Generating components
- Results

# Generating efficient backend is hard

Front end

IR

Optimizer

IR

Instruction Selector

Machine instructions

Register Allocator

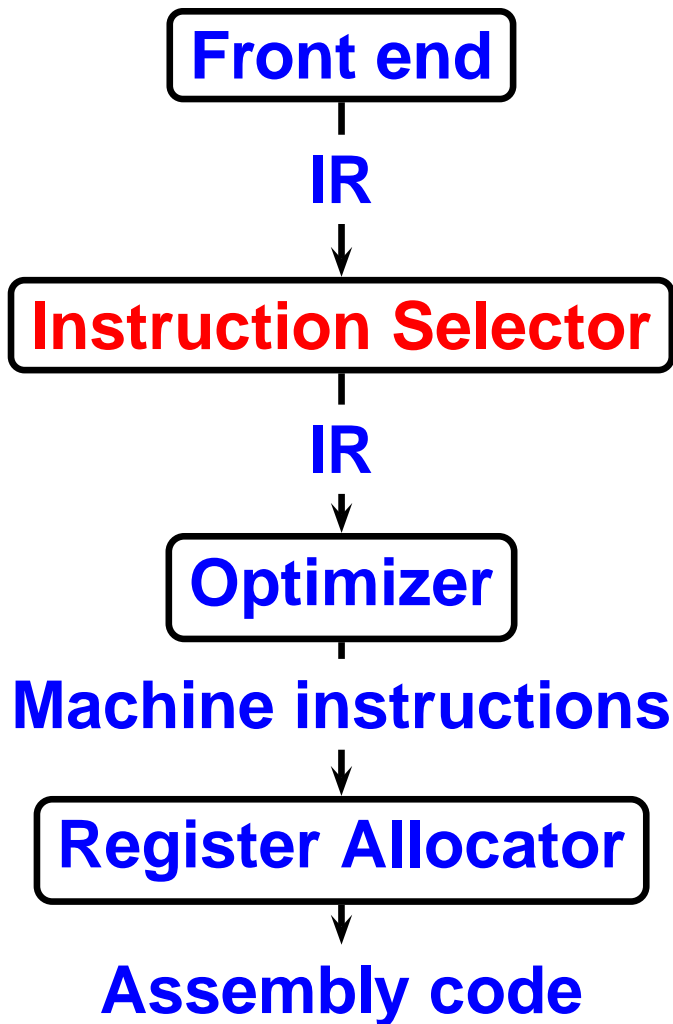
Assembly code

Machine-specific components run last

Problem:

- instruction selector must not undo optimizer's work

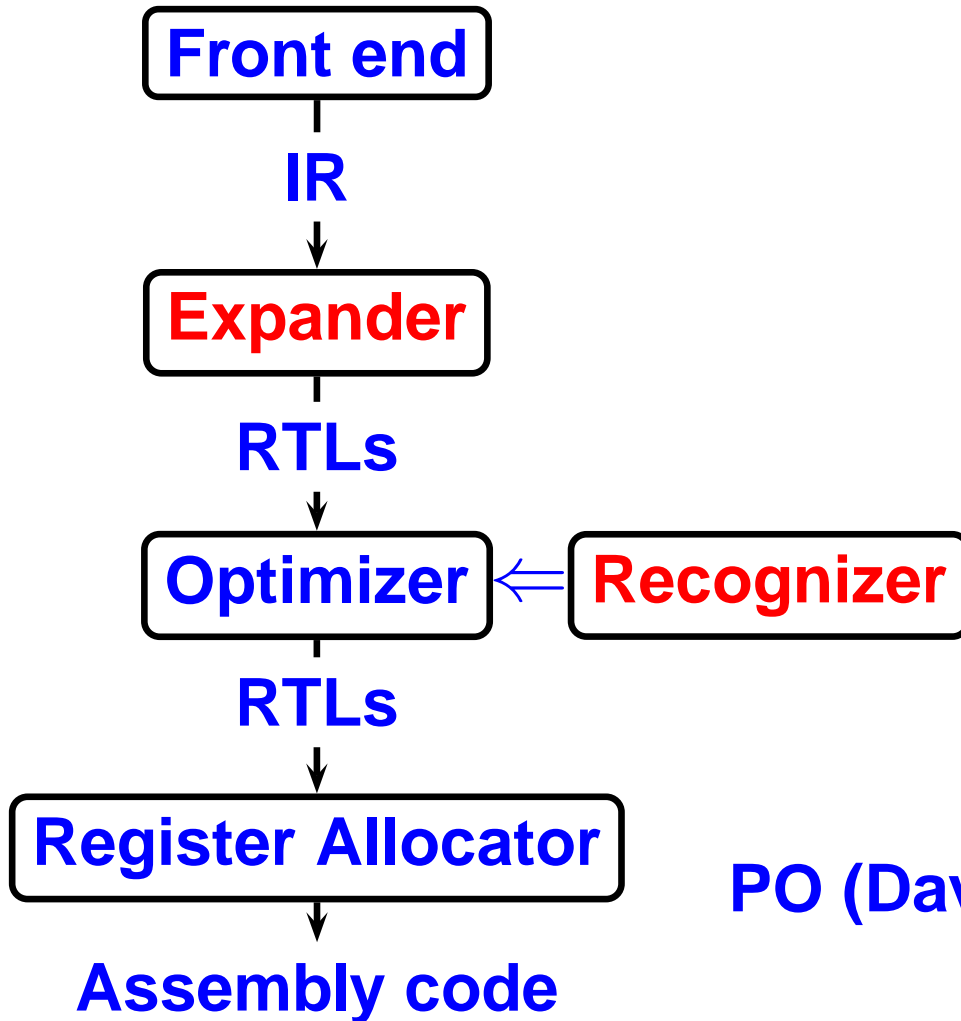
# Solution: do instruction selection first



Instruction selector can generate naive code:

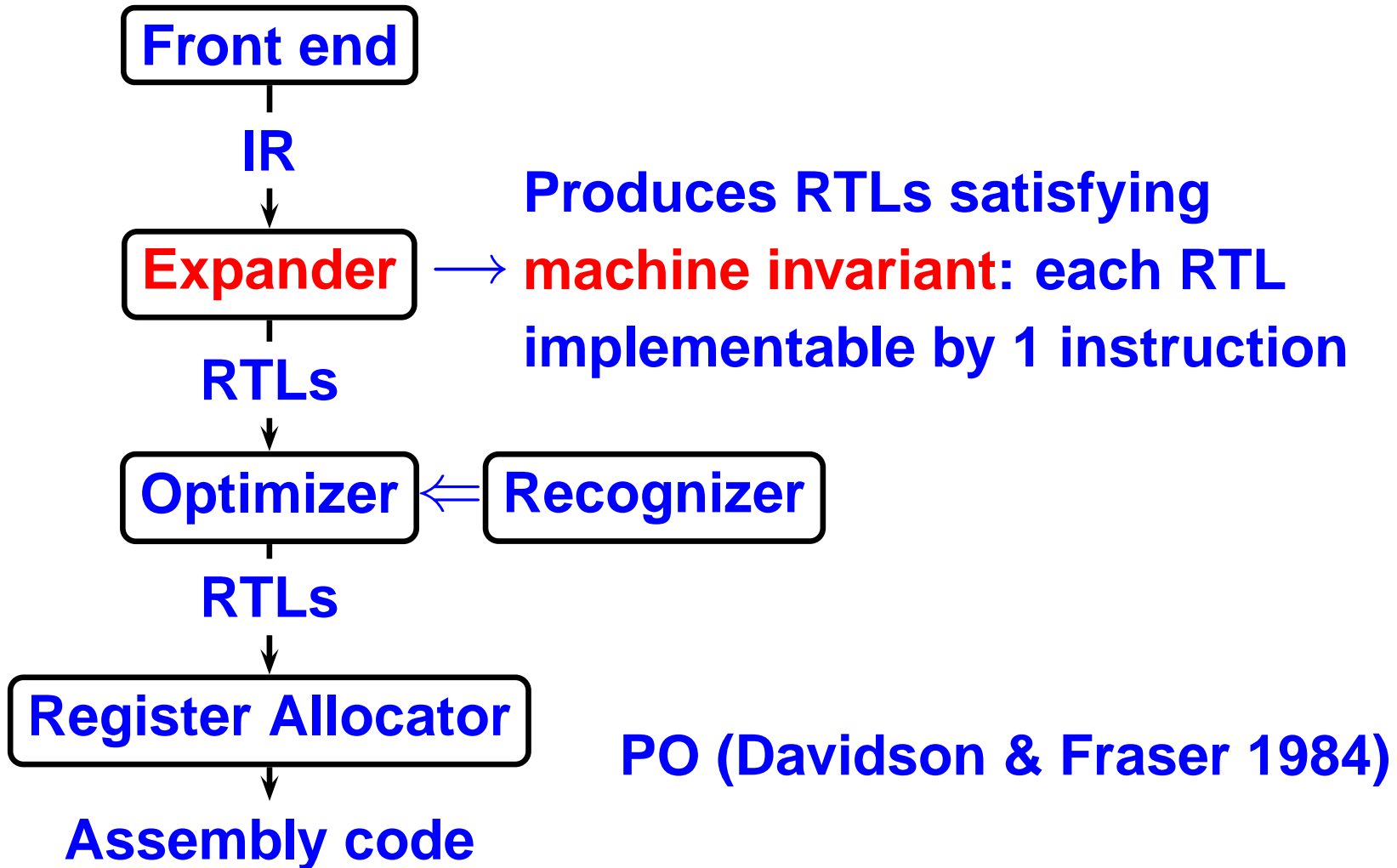
- easier to generate
- optimizer improves code

# Structure of a PO-style compiler

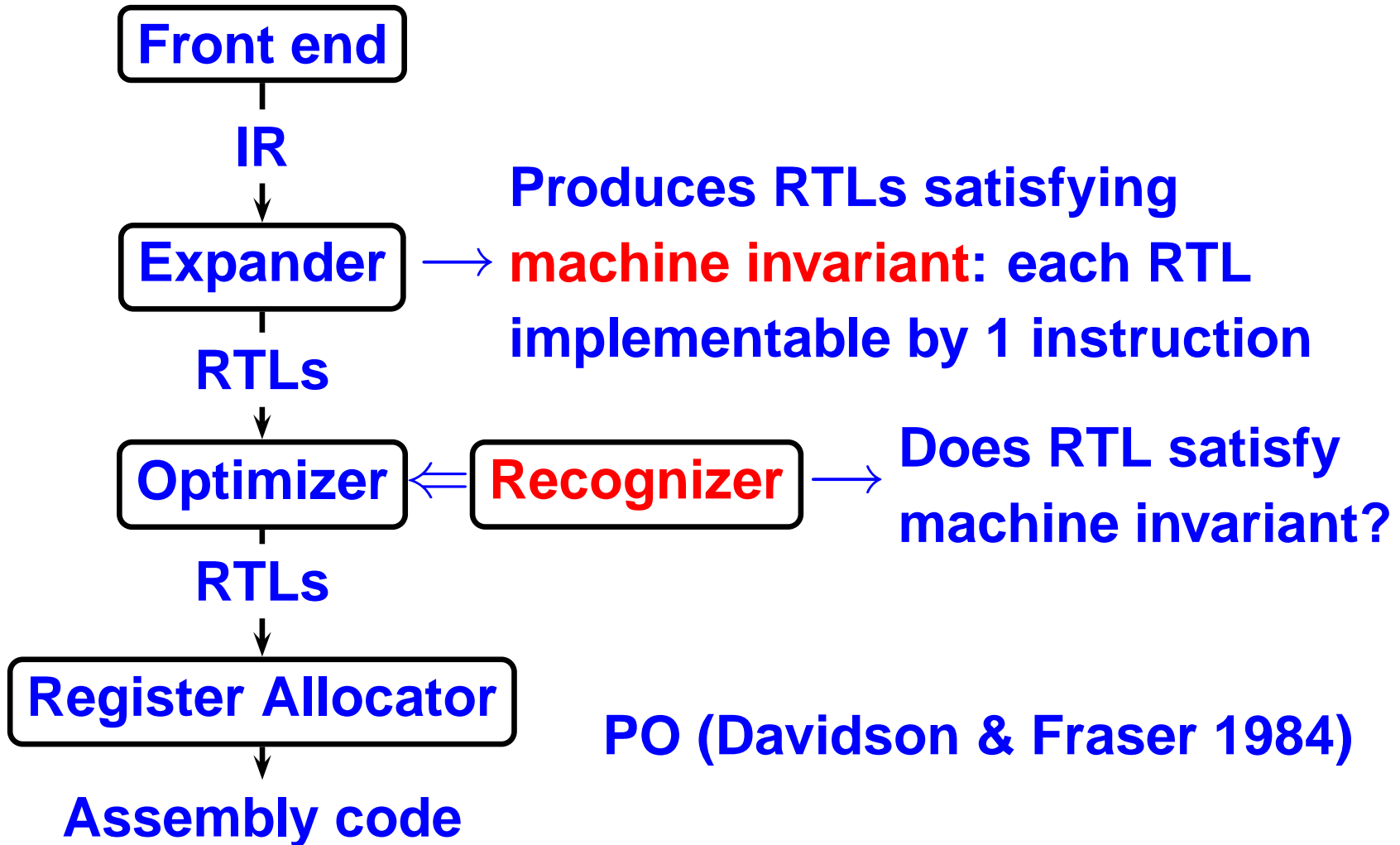


PO (Davidson & Fraser 1984)

# Structure of a PO-style compiler



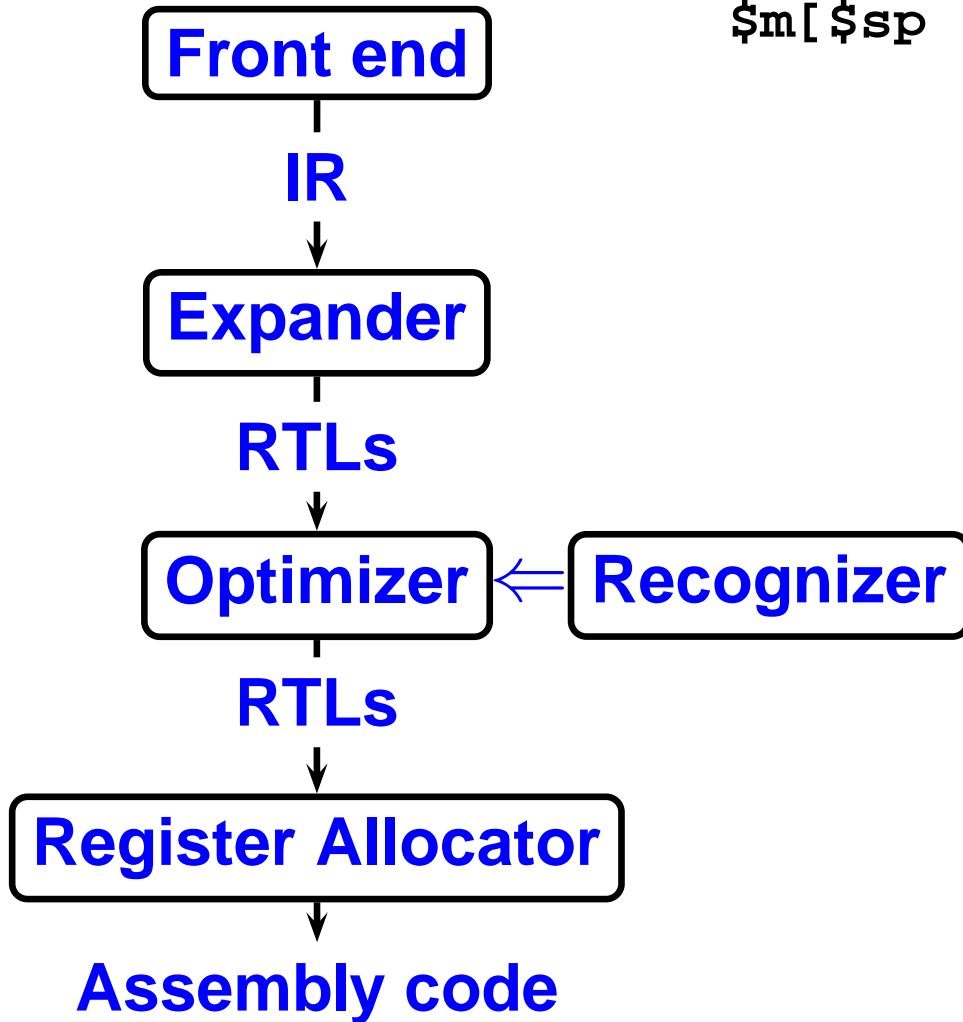
# Structure of a PO-style compiler



# A PO-style compiler in action

Stack-to-stack move:

```
$m[$sp + 8] := $m[$sp + 12];
```



# Expander chooses naive instructions

Instruction split into small steps:

```
$m[$sp + 8] := $m[$sp + 12];
```



```
$t[1] := $sp + 12;
```

```
$t[2] := $m[$t[1]];
```

```
$t[3] := $sp + 8;
```

```
$m[$t3] := $t[2];
```

Front end

IR

Expander

RTLs

Optimizer

Recognizer

RTLs

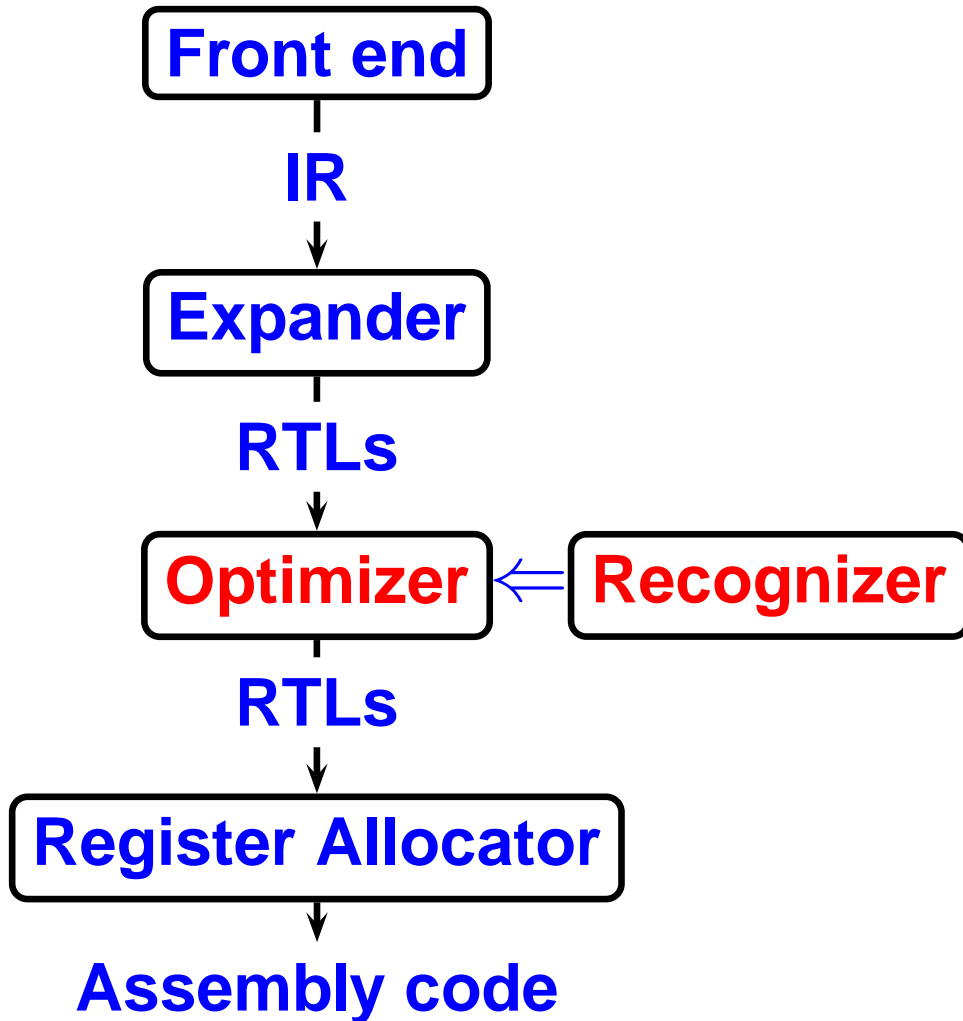
Register Allocator

Assembly code

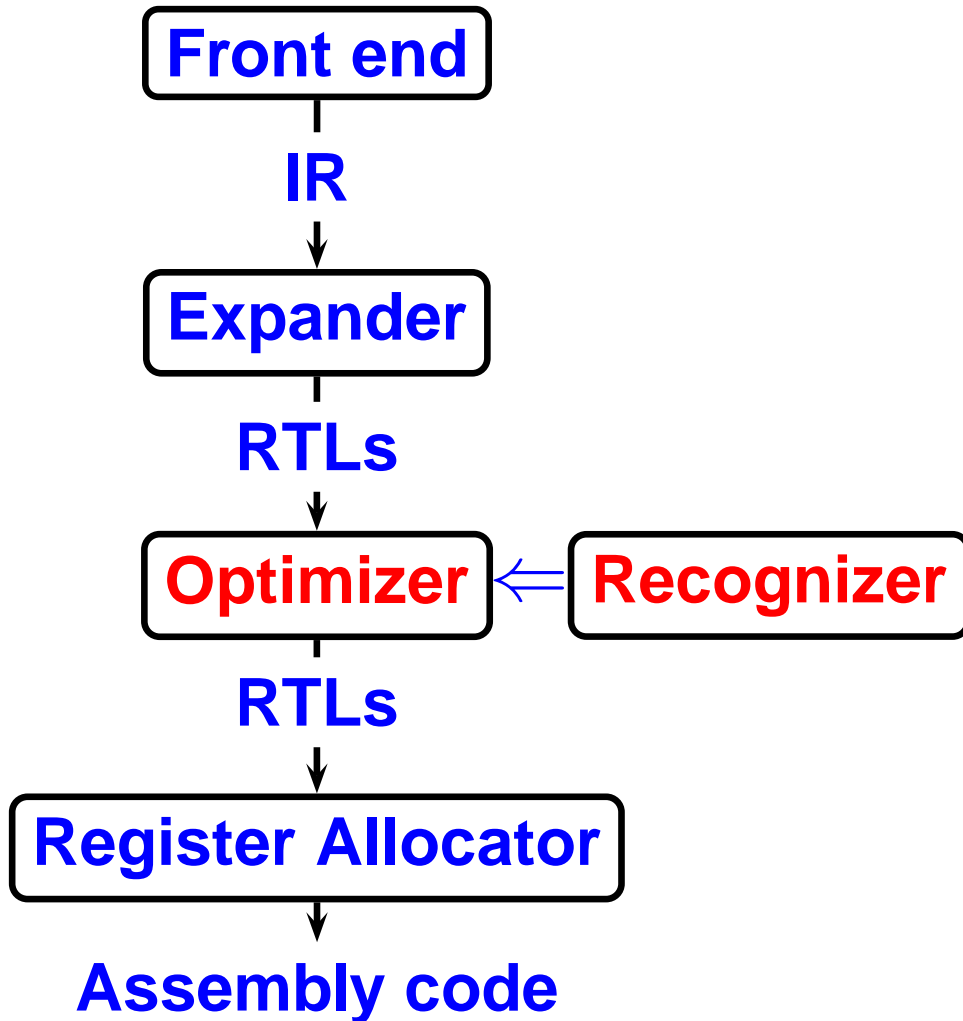
# Optimizer improves the code

## Peephole optimization:

```
$t[1] := $sp + 12;  
$t[2] := $m[$t[1]];  
$t[3] := $sp + 8;  
$m[$t[3]] := $t[2];
```



# Optimizer improves the code



## Peephole optimization:

```
$t[1] := $sp + 12;
```

```
$t[2] := $m[$t[1]];
```

```
$t[3] := $sp + 8;
```

```
$m[$t[3]] := $t[2];
```

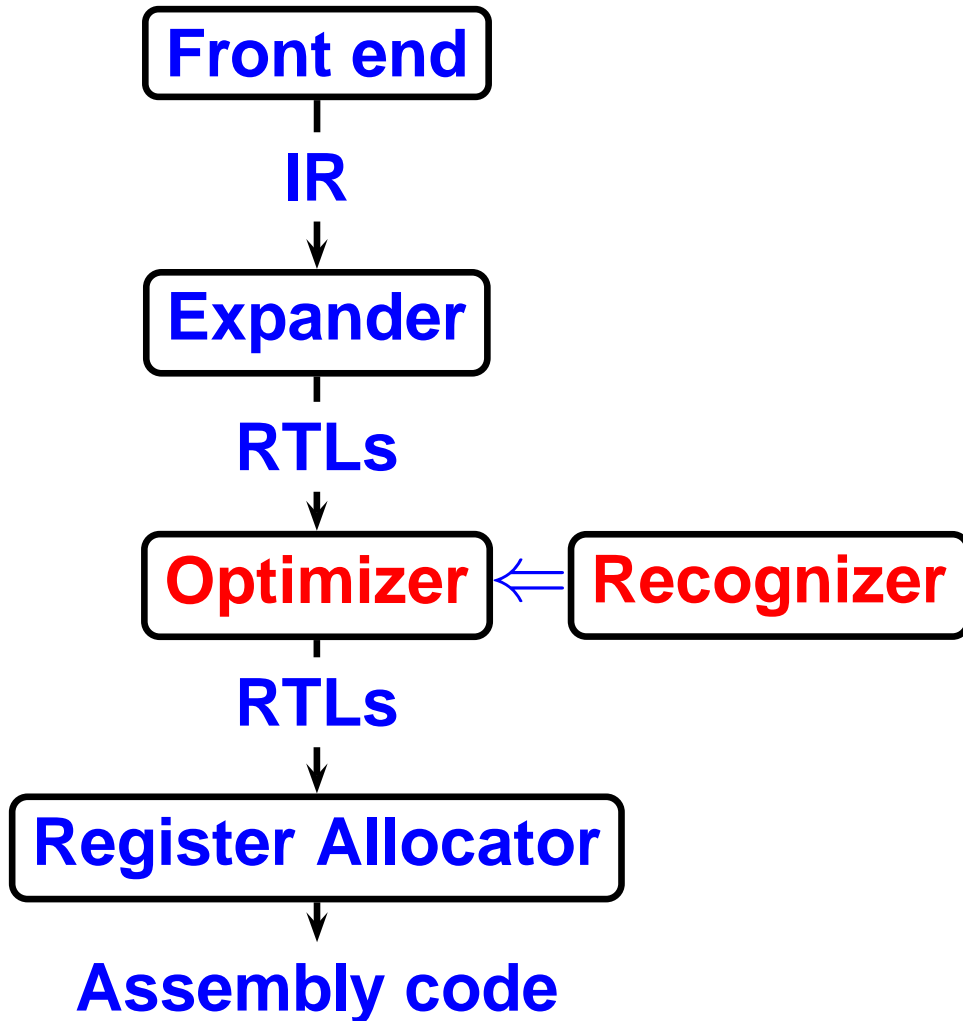


```
$t[2] := $m[$sp + 12];
```

```
$t[3] := $sp + 8;
```

```
$m[$t[3]] := $t[2];
```

# Optimizer improves the code



## Peephole optimization:

```
$t[1] := $sp + 12;
```

```
$t[2] := $m[$t[1]];
```

```
$t[3] := $sp + 8;
```

```
$m[$t[3]] := $t[2];
```



```
$t[2] := $m[$sp + 12];
```

```
$m[$sp + 8] := $t[2];
```

# Recognizer rejects invalid RTLs

Invalid transformation:

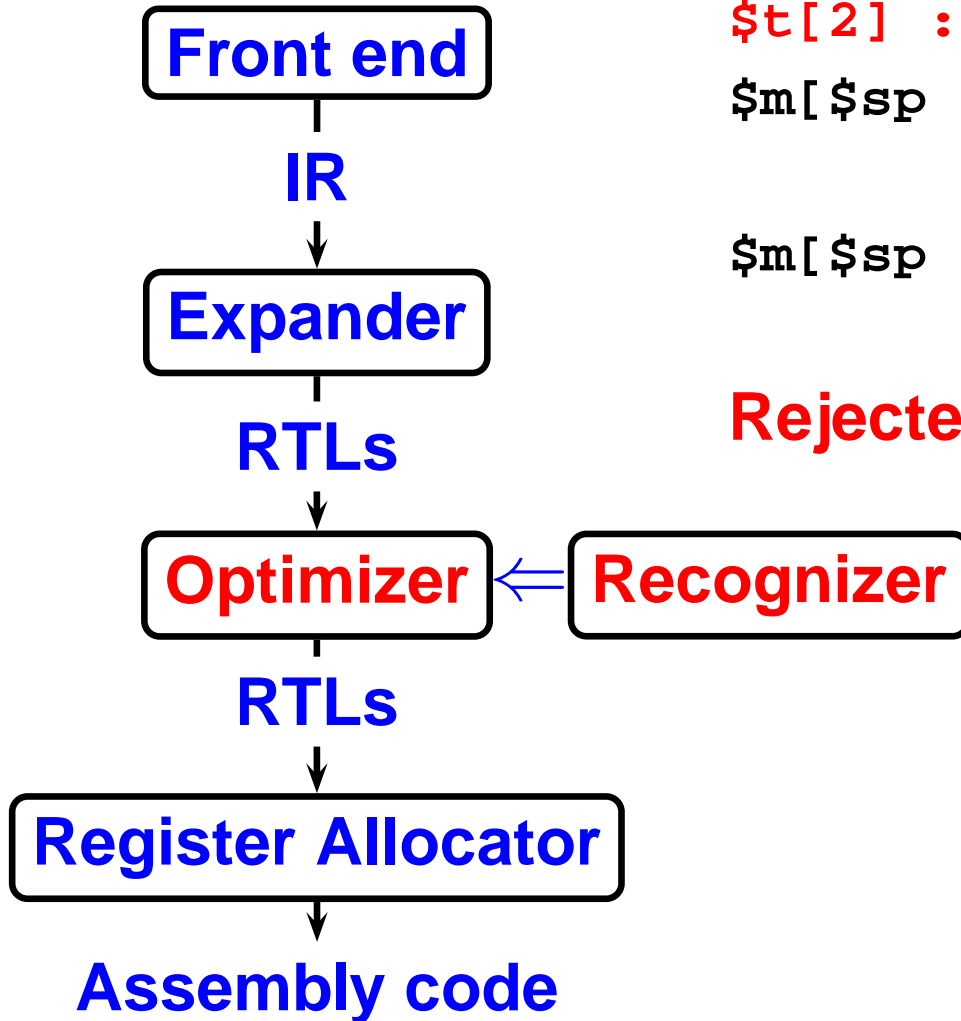
```
$t[2] := $m[$sp + 12];
```

```
$m[$sp + 8] := $t[2];
```

$\not\Rightarrow$

```
$m[$sp + 8] := $m[$sp + 12];
```

**Rejected!**



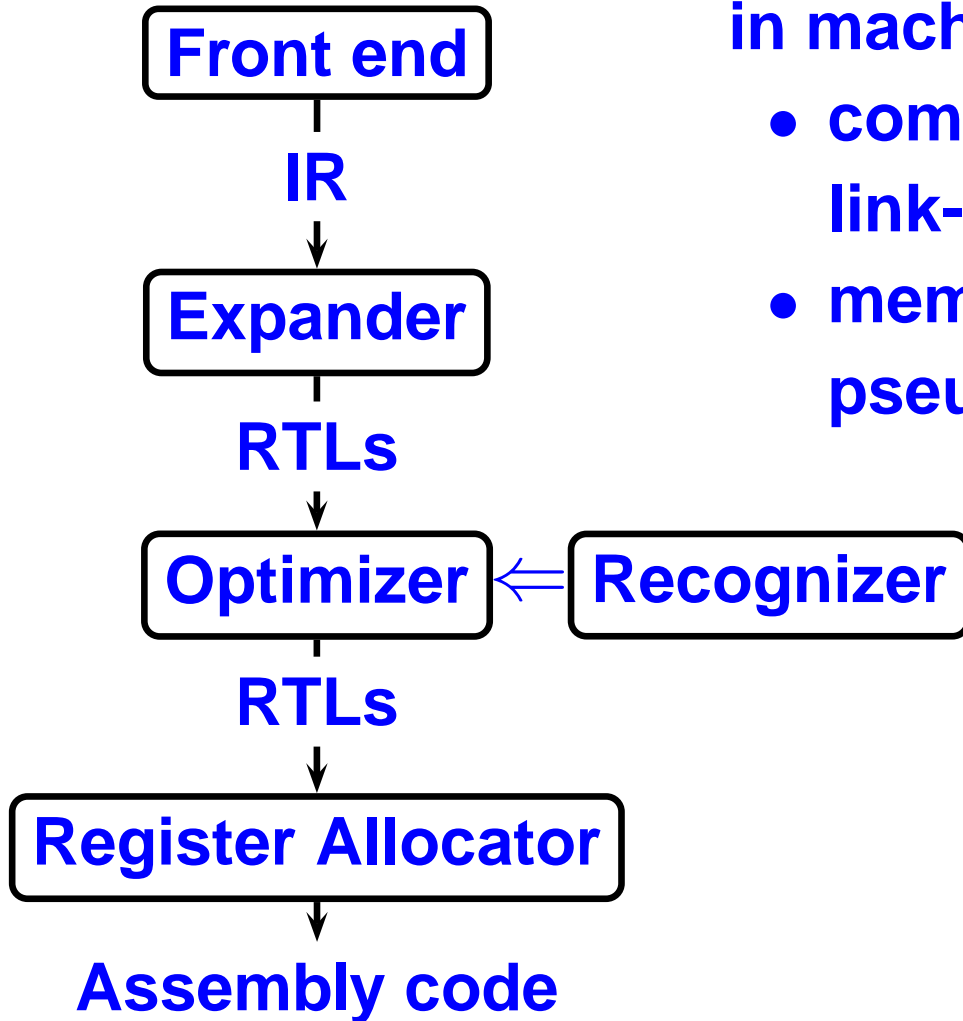
# Outline

- Introduction
- Describing semantics of instructions
- Restructuring the compiler
- **Generating the recognizer**
- Results

# The semantic gap

Compiler-level abstractions aren't in machine description:

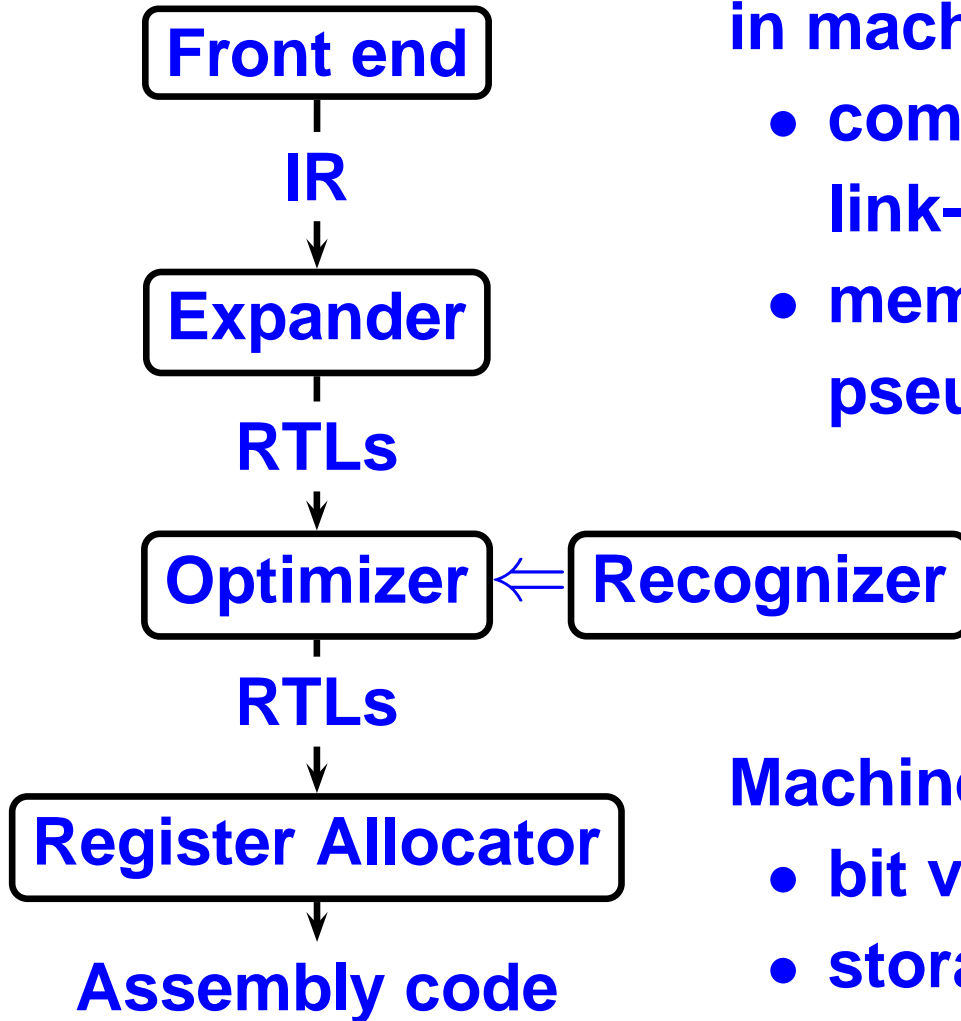
- compile-time constants, link-time constants
- memory, registers, pseudoregisters



# The semantic gap

Compiler-level abstractions aren't in machine description:

- compile-time constants, link-time constants
- memory, registers, pseudoregisters



Machine description provides:

- bit vectors
- storage spaces

# Bridging the semantic gap

Recognizer must accept RTLs with compiler abstractions

# Bridging the semantic gap

Recognizer must accept RTLs with compiler abstractions

Focus of talk: locations

- distinguish registers and memory
- identify pseudoregisters

How? Analysis on instruction set

# Binding time is the key

**Binding time:** when is value of expression known?  
(Feigenbaum 2001)

# Binding time is the key

**Binding time:** when is value of expression known?  
(Feigenbaum 2001)

- **Specification time:** depends only on literals;  
known from  $\lambda$ -RTL specification

```
inc(reg) is $r[reg] := $r[reg] + 1;
```

# Binding time is the key

**Binding time:** when is value of expression known?

(Feigenbaum 2001)

- **Specification time:** depends only on literals; known from  $\lambda$ -RTL specification
- **Instruction-creation time:** depends on operands, literals; known when instruction created

```
inc(reg) is $r[reg] := $r[reg] + 1;
```

# Binding time is the key

**Binding time:** when is value of expression known?  
(Feigenbaum 2001)

- **Specification time:** depends only on literals; known from  $\lambda$ -RTL specification
- **Instruction-creation time:** depends on operands, literals; known when instruction created
- **Run time:** depends on machine state, known at run time

```
inc(reg) is $r[reg] := $r[reg] + 1;
```

# Binding times distinguish registers

Binding times of addressing expressions determine how spaces are used:

# Binding times distinguish registers

Binding times of addressing expressions determine how spaces are used:

- **Fixed space:** each addressing expression known at specification time (condition codes)

# Binding times distinguish registers

Binding times of addressing expressions determine how spaces are used:

- **Fixed space:** each addressing expression known at specification time (condition codes)
- **Register-like space:** addressing expression known at instruction-creation time (registers)

# Binding times distinguish registers

Binding times of addressing expressions determine how spaces are used:

- **Fixed space:** each addressing expression known at specification time (condition codes)
- **Register-like space:** addressing expression known at instruction-creation time (registers)
- **Memory-like space:** addressing expressions not known until run time (memory, x86 floating-point stack)

# Finding classes of pseudoregisters

## Pseudoregister space

- infinite collection of imaginary registers
- stands for set of hardware registers

## Example register sets:

- 32-bit registers on x86
- 16-bit registers on x86
- \$r[1]-\$r[31] on Sparc

**Problem:** Find these sets by analysis of  $\lambda$ -RTL

# Location-set analysis

Identify interchangeable set of hardware registers

For each register-like location in instruction, determine range of addressing expression

- **Generally unconstrained**

```
inc(reg) is $r[reg] := $r[reg] + 1;
```

- **Constrained**

```
add(rd, rs1, i13) is
```

```
rd != 0 && rs1 != 0 ->
```

```
$r[rd] := $r[rs1] + sx i13;
```

Each register-like location set has corresponding pseudoregister space (accepted by recognizer)

# Generating the recognizer

Recognizer must accept RTLs corresponding to instructions

Generate a matcher using  $\lambda$ -RTL analysis

- Distinguish registers and memory
- Accept pseudoregisters

# The matcher

## Bottom-up BURG-style tree matcher for RTLs

- Syntactic matching under simple normal form
- Also generates assembly for instructions (used to emit assembly code)

# Generated recognizer is efficient

## Comparing hand-written and generated x86 recognizer

- Recognize same set of RTLs
- Total compilation time for compiler test suite

Recognizer	Compilation Time	Fraction of compilation taken by recognizer
Hand-written	69.29 s	3.99%
Generated	64.23 s	0.69%

# Contributions

**Instruction selection before optimization to generate efficient backend**

**Even compiler backend is surprisingly high level**

- **Analyses (binding times) recover high-level abstractions used by compilers**

**Automatic generation of efficient recognizer from machine descriptions**

# Future work

