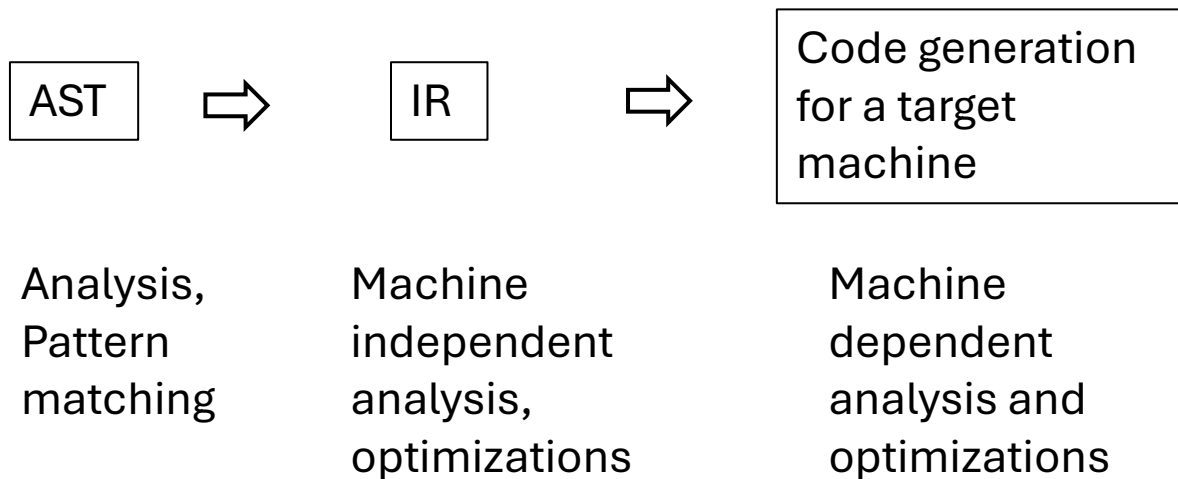


Code Generation – I

Nikhil Hegde


Compiler Optimizations in LLVM
Lecture series @ QUALCOMM Inc.


Compiler phases so far..



Naïve approach

- “Macro-expansion”
- Treat each 3AC instruction separately, generate code in isolation

ADD A, B, C  LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C

MUL A, 4, B  LD A, R1
MOV 4, R2
MUL R1, R2, R3
ST R3, B

Why is this bad? (I)

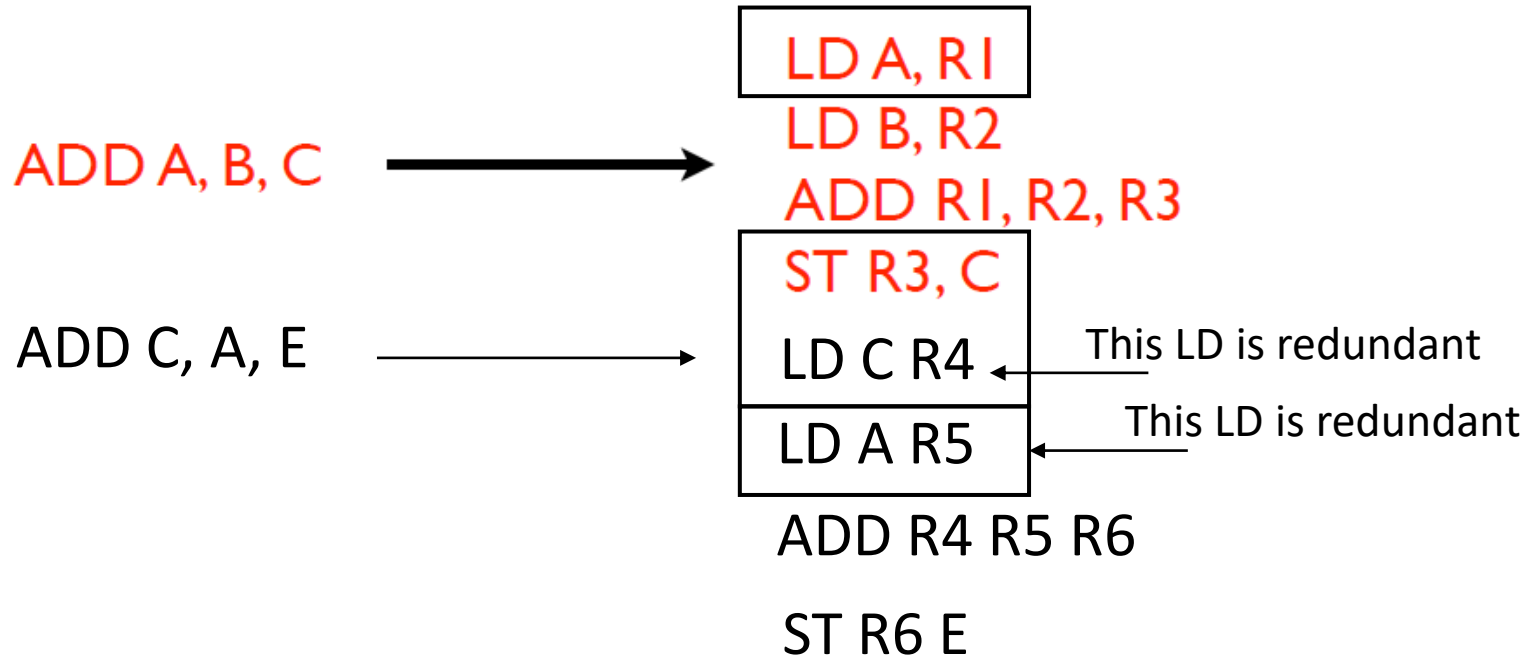
MUL A, 4, B →
LDA A, R1
MOV 4, R2
MUL R1, R2, R3
ST R3, B

MUL A, 4, B →
LDA A, R1
MULI R1, 4, R3
ST R3, B

There is a better instruction available!

Too many instructions
Should use a different instruction type

Why is this bad? (II)



Why is this bad? (III)

ADD A, B, C → LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C

ADD A, B, C
ADD A, B, D → LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C
LD A, R4
LD B, R5
ADD R4, R5, R6
ST R6, D

Wasting instructions recomputing $A + B$

How do we address this?

- Several techniques to improve performance of generated code
 - *Instruction selection* to choose better instructions
 - *Peephole optimizations* to remove redundant instructions
 - *Common subexpression elimination* to remove redundant computation
 - *Register allocation* to reduce number of registers used

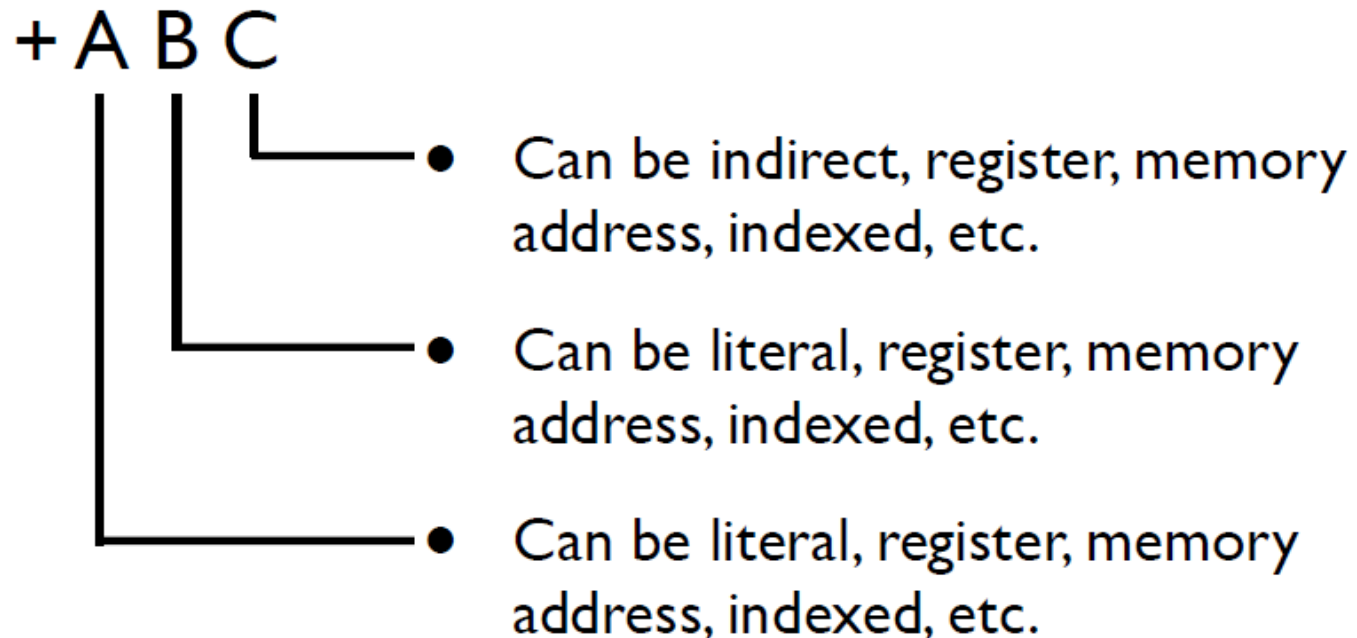
LLVM Code Generator Steps

- Instruction Selection
 - LLVM IR -> DAG
- Scheduling and Formation
 - determine a schedule for DAG nodes
- SSA-based Machine Code Optimization
 - e.g. peephole optimizations
- Register Allocation
 - Unlimited virtual registers to limited machine registers. Introduce spill code if required.
- Prolog/Epilog Code generation
 - Function call conventions, frame pointer elimination
- Late Machine Code Optimization
 - Spill code scheduling, peephole optimizations
- Code Emission
 - Assembler code or direct machine code

Challenges

Instruction selection

- Even a simple instruction may have a large set of possible address modes and combinations

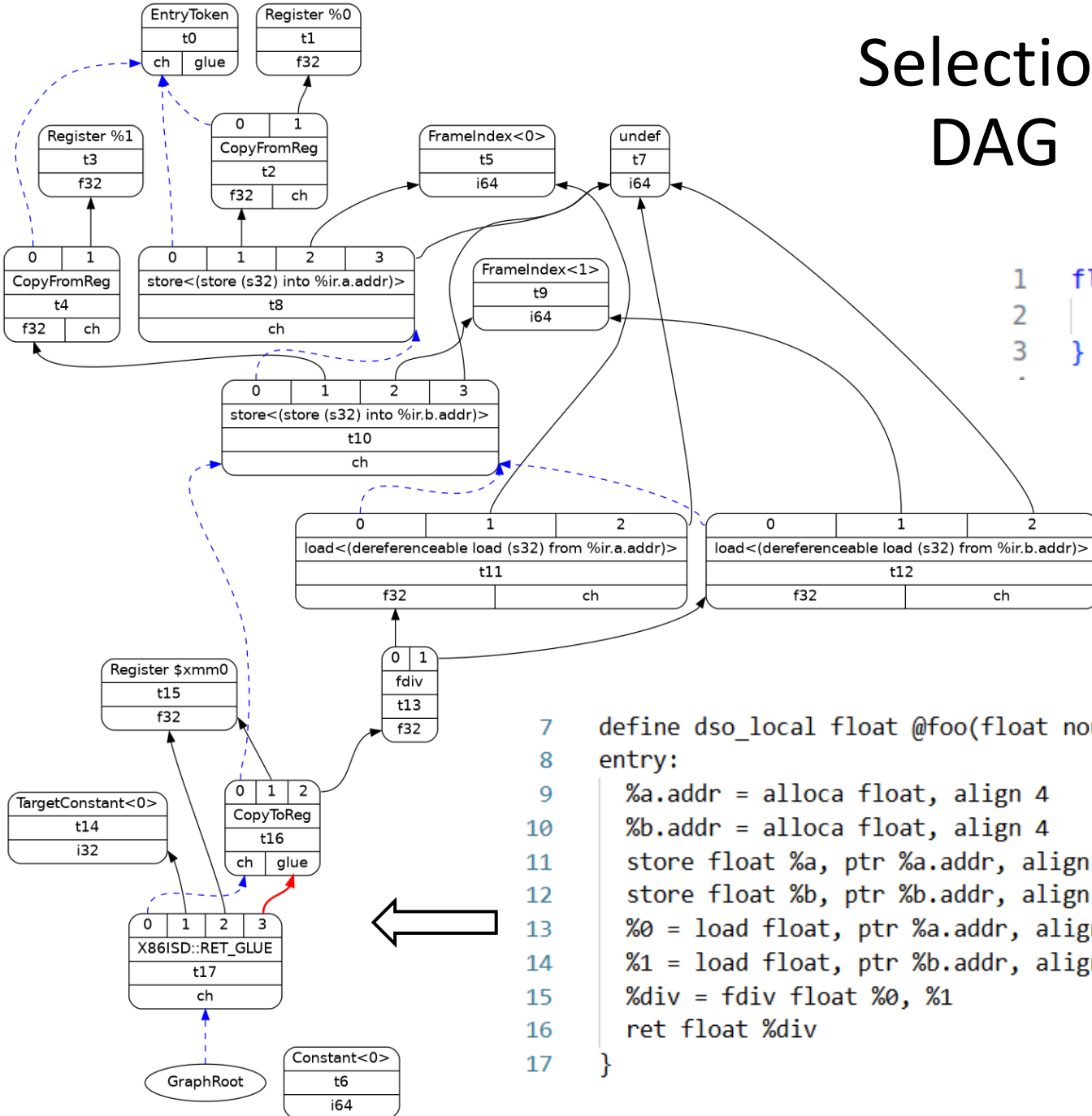


- Dozens of potential combinations!

Instruction Selection in LLVM

- **Objective:** LLVM IR to target-specific machine instruction
- LLVM implementations: SelectionDAG, GlobalISel
- Considering LLVM's codegen support for a variety of target platforms, the design of Instruction Selection module has reusable and target specific components.
- SelectionDAG - why DAG representation?
 - After few iterations, the DAG we would have makes it easy to schedule instructions. (schedulingDAG).
 - Enable automation

Selection DAG



```

1 float foo(float a, float b){
2     return a/b;
3 }

```

```

7 define dso_local float @foo(float noundef %a, float noundef %b) #0 {
8   entry:
9     %a.addr = alloca float, align 4
10    %b.addr = alloca float, align 4
11    store float %a, ptr %a.addr, align 4
12    store float %b, ptr %b.addr, align 4
13    %0 = load float, ptr %a.addr, align 4
14    %1 = load float, ptr %b.addr, align 4
15    %div = fdiv float %0, %1
16    ret float %div
17 }

```

Selection DAG

- Node
 - Opcode, Operand, Value(s), Machine Value Type,
 - Values are of two kinds: representing data- and control- flow
- Edges
 - Simple: represent dataflow from e.g., type int to another int
 - Chained: define ordering between nodes that have side effect. E.g. stores, loads, call, return.
- Can have legal and illegal DAGs
 - Illegal DAG: a node with value, whose type is not supported on the target machine. E.g. Native support for `i1` type on most modern processors.

Instruction Selection

1. Build Initial DAG
2. Optimize Initial DAG
3. Legalize SelectionDAG Types
4. Optimize Selection DAG
5. Legalize SelectionDAG Operations
6. Optimize SelectionDAG
7. Select Instructions from SelectionDAG (also called DAG to DAG)
8. SelectionDAG scheduling and Formation

Demo

Legalize Type

1. Optimized lowered selection DAG: %bb.0 'foo:'
 2. SelectionDAG has 15 nodes:
 3. t0: ch,glue = EntryToken
 4. t2: i32,ch = CopyFromReg t0, Register:i32 %0
 5. t3: i16 = truncate t2
 6. t5: i16 = add t3, Constant:i16<5>
 7. t8: i1 = setcc t5, Constant:i16<6>, setugt:ch
 8. t10: i32 = select t8, t2, Constant:i32<9>
 9. t13: ch,glue = CopyToReg t0, Register:i32 \$eax, t10
 10. t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 \$eax, t13:1
-

1. Type-legalized selection DAG: %bb.0 'foo:'
2. SelectionDAG has 17 nodes:
3. t0: ch,glue = EntryToken
4. t2: i32,ch = CopyFromReg t0, Register:i32 %0
5. t3: i16 = truncate t2
6. t5: i16 = add t3, Constant:i16<5>
7. t15: i8 = setcc t5, Constant:i16<6>, setugt:ch
8. t18: i8 = and t15, Constant:i8<1>
9. t10: i32 = select t18, t2, Constant:i32<9>
10. t13: ch,glue = CopyToReg t0, Register:i32 \$eax, t10
11. t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 \$eax, t13:1

Legalize Type

- Promotion
- Expansion `__int128_t` a
- Soften
- Split vector
- Widen vector

Legalize Operation

1. Type-legalized selection DAG: %bb.0 'foo:'
 2. SelectionDAG has 17 nodes:
 3. t0: ch,glue = EntryToken
 4. t2: i32,ch = CopyFromReg t0, Register:i32 %0
 5. t3: i16 = truncate t2
 6. t5: i16 = add t3, Constant:i16<5>
 7. t15: i8 = setcc t5, Constant:i16<6>, setugt:ch
 8. t18: i8 = and t15, Constant:i8<1>
 9. t10: i32 = select t18, t2, Constant:i32<9>
 10. t13: ch,glue = CopyToReg t0, Register:i32 \$eax, t10
 11. t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 \$eax, t13:1
-

1. Legalized selection DAG: %bb.0 'foo:'
2. SelectionDAG has 15 nodes:
3. t0: ch,glue = EntryToken
4. t2: i32,ch = CopyFromReg t0, Register:i32 %0
5. t3: i16 = truncate t2
6. t5: i16 = add t3, Constant:i16<5>
7. t20: i16,i32 = X86ISD::SUB t5, Constant:i16<7>
8. t23: i32 = X86ISD::CMOV Constant:i32<9>, t2, TargetConstant:i8<3>, t20:1
9. t13: ch,glue = CopyToReg t0, Register:i32 \$eax, t23
10. t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 \$eax, t13:1

Legalize Operation

- Expand
- Libcall
- Promote
- Custom

Instruction Scheduling

Instruction Scheduling

- Code generation has created a sequence of assembly instructions
- But that is not the only valid order in which instructions could be executed!

LD A, R1
LD B, R2
R3 = R1 + R2
LD C, R4
R5 = R4 * R2
R6 = R3 + R5
ST R6, D



LD C, R4
LD B, R2
LD A, R1
R5 = R4 * R2
R3 = R1 + R2
R6 = R3 + R5
ST R6, D

- Different orders can give you better performance, more instruction level parallelism, etc.

Why do Instruction Scheduling?

- Not all instructions are the same
 - Loads tend to take longer than stores, multiplies tend to take longer than adds
- Hardware can overlap execution of instructions (pipelining)
 - Can do some work while waiting for a load to complete
- Hardware can execute multiple instructions at the same time (superscalar)
- Hardware has multiple functional units

Why do Instruction Scheduling? Contd..

- VLIW (very long instruction word)
 - Popular in the 1990s, still common in some DSPs
 - Relies on compiler to find best schedule for instructions, manage instruction-level parallelism
 - Instruction scheduling is vital
- Out-of-order superscalar
 - Standard design for most CPUs (some low energy chips, like in phones, may be in-order)
 - Hardware does scheduling, but in limited window of instructions
 - Compiler scheduling still useful to make hardware's life easier

Instruction Scheduling - Considerations

- Gather constraints on schedule:
 - Data dependences between instructions
 - Resource constraints
- Schedule instructions while respecting constraints
 - List scheduling
 - Height-based heuristic