Optimization Overview
data types

• You may have noticed that the code you are generating in your project seems very inefficient

• Lots of redundant computation

```
LA T1 <address of x>
LW T2 0(T1)
LA T3 <address of x>
LW T4 0(T3)
ADD T5 T2 T4
```

instead of

```
LA T1 <address of x>
LW T2 0(T1)
LW T4 0(T1)
ADD T5 T2 T4
```
data types

• You may have noticed that the code you are generating in your project seems very inefficient

• Lots of redundant computation

```assembly
LA T1 <address of x>
LW T2 0(T1)
LA T3 <address of x>
LW T4 0(T3)
ADD T5 T2 T4
```

instead of

```assembly
LA T1 <address of x>
LW T2 0(T1)
LW T4 0(T1)
ADD T5 T2 T4
```
You may have noticed that the code you are generating in your project seems very inefficient.

Lots of instruction choice.

Instead of:
```
LI T1 10
LW T2 8(FP)
ADD T3 T1 T2
SW T3 -4(FP)
```

```
LW T2 8(FP)
ADDI T3 T2 10
SW T3 -4(FP)
```
You may have noticed that the code you are generating in your project seems very inefficient

Lots of unnecessary loads and stores

Instead of

```
LA T1 <address of x>
LI T2, 10
SW T2, 0(T1)
LW T3, 0(T1)
ADDI T4, T3, 20
SW T4, 0(T1)
```

```
But code inefficiency goes beyond just small sequences of instructions

• What about redundant computation happening inside a loop?

• What about expensive operations happening inside a loop?

• What about code that access memory in a way that is bad for caching?
optimization

• Compilers can generate correct code relatively easily
• But generating efficient code is much harder
  • This is where a lot of programming languages research happens

• Can happen at multiple levels
  • At the source code level or AST level (e.g., restructure loops for better performance)
  • At the assembly level (e.g., replace some sequences of instructions with more efficient sequences)
  • At the intermediate representation level (e.g., remove redundant instructions)
intermediate representation

- We have already worked with one intermediate representation: abstract syntax trees
- Many compilers have another, lower-level intermediate representation that facilitates optimization
  - Closer to assembly, but no machine specific instructions, registers, etc.
  - Examples: LLVM bitcode, C# CIL, Java bytecode
Intermediate Representation
why an intermediate representation?

• Want to represent code in a form that is:
  • Closer to assembly than ASTs — low level operations, branches, memory operations
  • Not machine specific — no registers, instructions more “abstract” than machine instructions
  • Makes it easier to perform certain kinds of optimizations

LA T1 <address of x>
LI T2, 10
SW T2, 0(T1)
LW T3, 0(T1)
ADDI T4, T3, 20
SW T4, 0(T1)

becomes

MV 10, $GX  //X = 10
ADD $GX, 20, $GX  //X = 20 + X
three-address code

• All operations take at most three operands: two source operands, one destination operand

• Almost the same as Risc-V assembly, except:
  • No registers, only temporaries
  • Operands can be literals, temporaries, or variables
    • Loads and stores are implicit
    • Encode address information in operand names for easy translation later
      • Temporaries: $T\text{x}$
      • Globals: $G<\text{name}>$
      • Locals: $L<\text{stack offset}>$
converting 3ac into assembly

• Simple approach: **macro expansion**
• Treat each 3AC instruction separately, generate code in isolation
• Can be clever about how we turn 3AC into assembly by selecting appropriate assembly instructions

  • If one source operand is a literal, generate an immediate instruction

  • If source operand is a local variable, generate a load with an offset, rather than an address computation and a load
converting 3ac into assembly

• Generating better code:

```
ADD $GC, $LP4, $LL8
LW r1, 4(fp) //parameter at +4
LW r2, -8(fp) //local at -8
ADD r3, r1, r2
LA r4 <addr of C> //global C
SW r3, 0(r4)
```