Agenda

- Branching IR
- Control flow graphs
- Routines
- Real machines
- Wrapping up
Quadruple Intermediate Representation

Express machine instructions with their input and output registers as 4-tuples

( opcode, left, right, result )

• Assume an infinite number of temporary registers $R_i$

  • Hence the term _intermediate_, these will have to be mapped onto a finite set of physically available machine registers

• Special opcode called _label_ (not an actual machine instruction)

• Results can include registers $R_i$, constant indices $T_i$ and _labels_

  • Constant $T_i$ refers to tuple $i$ in sequence

  • Symbolic label $L_{name}$ will be resolved to a specific target tuple index later
Branching IR Example

\[
( (A \times B) \leq (C + D) ) \text{ and } ( (P == Q) \text{ or } (Q \neq R) )
\]

\[
\begin{align*}
T_1 & \quad ( \text{mul}, A, B, R_1 ) & R_1 & \leftarrow A \times B \\
T_2 & \quad ( \text{add}, C, D, R_2 ) & R_2 & \leftarrow C + D \\
T_3 & \quad ( \text{lte}, R_1, R_2, R_3 ) & R_3 & \leftarrow R_1 \leq R_2 \\
T_4 & \quad ( \text{br}, R_3, T???), Lfalse ) & \text{PC} & \leftarrow T??? \text{ if } R_3 \text{ else } Lfalse \\
T_5 & \quad ( \text{eq}, P, Q, R_4 ) & R_4 & \leftarrow P == Q \\
T_6 & \quad ( \text{br}, R_4, Ltrue, T??? ) & \text{PC} & \leftarrow Ltrue \text{ if } R_4 \text{ else } T??? \\
T_7 & \quad ( \text{neq}, Q, R, R_5 ) & R_5 & \leftarrow Q \neq R \\
T_8 & \quad ( \text{br}, R_5, Ltrue, Lfalse ) & \text{PC} & \leftarrow Ltrue \text{ if } R_5 \text{ else } Lfalse
\end{align*}
\]
Branching IR Example

\[(A \times B) \leq (C + D)\) and \[(P == Q) \text{ or } (Q != R)\]

\begin{align*}
T_1 &: (\text{mul}, A, B, R_1) \quad \text{R}_1 \leftarrow A \times B \\
T_2 &: (\text{add}, C, D, R_2) \quad \text{R}_2 \leftarrow C + D \\
T_3 &: (\text{lte}, R_1, R_2, R_3) \quad \text{R}_3 \leftarrow R_1 \leq R_2 \\
T_4 &: (\text{br}, R_3, T_5, L_{\text{false}}) \quad \text{PC} \leftarrow T_5 \text{ if } R_3 \text{ else } L_{\text{false}} \\
T_5 &: (\text{eq}, P, Q, R_4) \quad \text{R}_4 \leftarrow P == Q \\
T_6 &: (\text{br}, R_4, L_{\text{true}}, T_7) \quad \text{PC} \leftarrow T_7 \text{ if } R_3 \text{ else } L_{\text{false}} \\
T_7 &: (\text{neq}, Q, R, R_5) \quad \text{R}_5 \leftarrow Q != R \\
T_8 &: (\text{br}, R_5, L_{\text{true}}, L_{\text{false}}) \quad \text{PC} \leftarrow L_{\text{true}} \text{ if } R_5 \text{ else } L_{\text{false}}
\end{align*}
Control Flow Graphs

unconditional

BB 1 → BB 2

true

BB 3 → BB 4

false

BB 3 → BB 5
result = not P

genBB_BoolNot(node, result, bbT, bbF):
    bb = genBB(node.expr, result, bbF, bbT)
    return bb
Control flow graph examples
if expression? { statements... }
if expression? { s1... } else { s2... }
if e1? { s1... } else if e2? { s2... }

true

false

true

false

unconditional

unconditional
while expression? { statements... }
repeat { statements... } until expression?
CFG Examples

for (init; cond?; step) statement

Contains both forwards and backwards jumps
Mutually recursive basic blocks

```python
def genBB_Rec(...):
    bb3 = genBB(...)  # Uh oh...
    bb1 = genBB(..., bb2, bb3)
    bb2 = genBB(..., bb1, bb3)
    return bb3
```
Mutually recursive basic blocks

genBB_Rec(...):
    bb3 = genBB(...) 
    
    bb2ref = bbRef() 
    bb1ref = bbRef() 
    
    bb1 = genBB(..., bb2ref, bb3.ref) 
    bb2 = genBB(..., bb1ref, bb3) 
    
    bb2ref.ref = bb2 
    bb1ref.ref = bb1 
    
    return bb3

class BB:
    __init__:
        self.ref = self
Control Flow Graphs

- A control flow graph of interconnected basic blocks can be assembled into a final linear list of instructions and labels.

- Many opportunities for optimization at the level of the intermediate representation:
  - Block layout order can take advantage of branch fall-through instructions.
  - Remove unconditional jumps by merging two basic blocks into a single sequence of instructions (possibly with a label if the second block was the target of >1 branches).
  - When $bbLeft=bbRight$, transform conditional branch into unconditional jump (then apply above).
  - Delete instructions that produce values that are never used (requires more elaborate usage analysis to prove).
  - Propagate constant values, unroll short loops, hoist out common subexpressions, convert expensive instructions to cheaper ones.
Code generation for routines
Routine *declaration* code generation

- For each routine (procedure or function), lay out the activation record offsets for all key data: parameters, local variables, control

- When generating code for each routine, the code generated should contain three pieces:
  
  - Prologue: sets up the runtime environment for the routine, such as allocating local storage and setting up the display
  
  - Body scope code: generated statement by statement
  
  - Epilogue: cleans up the runtime environment, restores the display and jumps back to the caller *return site*
Routine *call*

code generation

- When calling a routine (function or procedure), the compiler should statically know the address where the routine declared starts (the prologue)

  - If not knowable statically (a *virtual method table* in C++ terms), at least have a method for where to find the correct destination branch address

- A routine *call* involves 3 parts:

  - **Setup**: evaluate arguments and make the results available as parameters, deal with return site and return value allocation

  - **Make the jump**: unconditionally branch to routine prologue

  - **Return site**: placed immediately after the jump, performs any necessary setup or return value extraction
Caller vs Callee

• The *caller* is the one making the call to the calle, and the *callee* is the thing that is being called by the caller

• To effect a call, the caller and callee must agree on certain convention and divisions of labour

• Who will handle each piece?
  • Register save and restore
  • Display manipulation
  • Return value preparation
  • Allocating local storage space
  • Argument evaluation
Possible activation record layout

**Callee Push**

1. Temporary storage
2. ...
3. Local storage...
4. Old Display Save
5. Argument N
6. ...
7. Argument 1
8. Return site address
9. Return value

**Caller Push**

**Callee Pop**

**Callee Pop**
Returning

- Routine return calls can be implemented as a branch to the local epilogue

- In the case of function return, first evaluate and save the return value and then branch

- Typically the ISA allocates a specific register for returning small scalar values (machine word sized)

- For larger return values (such as struct’s), the caller may allocate storage and pass a pointer to the callee through which it can write return values
Argument passing methods

- The caller passes in arguments which the callee receives as formally named parameters

- What does it mean to pass a parameter? What does $A[i]$ actually mean?
  - **Call by value**: pass value of $A[i]$
  - **Call by reference**: pass the address of $A[i]$
  - **Call by name**: pass *something* that calculates the address of $A[i]$
  - **Call by value-result**: pass the value of $A[i]$ as the named formal parameter on entrance, and copy the value of that formal parameter back to the original parameter on exit
Argument passing methods

- **By value**: formal parameter can be treated like a local variable that is pre-initialized with the value of the passed argument.
  - Some languages will dictate that all formal parameters are constant read-only.

- **By result**: the formal parameter acts like a kind of return value; it is uninitialized at entrance, but its value is copied back out to argument at exit.

- **By value-result**: like by result, but argument is initially passed by value into formal parameter.

- **By reference (address)**: formal parameter is actually passed the *address* of the argument, and any scalar assignment of that parameter name acts like a pointer-dereferenced assignment.

- **By name**: like by reference, except that address is recalculated on each use (lazy evaluation).
Passing interpretations

```go
func F(p integer) {
    p = p + 1
    print p
}
var x integer
x = 1
F(x)
print x
```

Output:

By value:

2 1

By reference, value-result:

2 2
Passing interpretations

```plaintext
func F(i, E integer) {
    i = i + 1
    E = E + 1
}
var i integer
i = 0
A[0] = 0  A[1] = 1
F(i, A[i])
print i, A[0], A[1]
```

Output:

**By value:**

```
0 0 1
```

**By reference, value-result:**

```
1 1 1
```

**By name:**

```
1 0 2
```
func F() {
    func A() {
        func P() { … }
        func Q() { … }
    }
    func B() { … }
}
Quad IR vs Real Machines

• Not all instruction opcodes we want to use in the IR are available as actual machine instructions.

• The machine code specifics for real architectures sometimes offer compound instructions that can express multiple steps in one instruction:

  \[ \text{mov rax, } [rbx + 4*rcx] \quad rax = *(rbx + 4*rcx) \]

• Templates can be used to identify and express multiple quad IR opcodes as single machine instructions using these addressing modes.

• The tuple-based IR assumed an infinite number of pseudo-registers \( R_k \), whereas real machines have a finite number.

• \textit{Register allocation} is the process of mapping the pseudo-registers onto real machine registers, taking into account overlapping use and the finite limit.
Register Allocation

- Perform a Live Variable Analysis on all pseudo registers used
  - A register is *live* inclusively between the point where it is given a value and where it is last used
- Build an interference graph:
  - Register X *interferes* with register Y if X is *live* at the point of *definition* for Y
  - The graph contains registers $R_k$ as nodes, and edges between nodes $R_a$ and $R_b$ if the registers interfere with one another
  - If there are $N$ physical registers available, any node with less than $N$ edges can be assigned to a physical register
Register Allocation

- load A, R₁
- add R₁, D, R₁
- load B, R₂
- load C, R₃
- ... uses of R₂, R₃ ...

R₁ live
Register Allocation

- Attempt to *colour* the interference graph with \( N \) unique colours (where \( N \) is the number of available physical registers)

- Nodes connected in the graph cannot share the same colour (because they are live at the same time)

- A successful graph colouring corresponds to a valid mapping of pseudo registers to machine registers

- If colouring succeeds, all pseudo registers have been mapped

- If it fails, find the region with highest register *pressure* (most registers live at same time) and *spill* some pseudo registers into temporary storage memory

- Graph colouring is NP hard, although some linear approximations and alternate approaches to allocation exist
In practise

• Libraries like LLVM provide users with a convenient, higher level IR abstraction with complete optimizing code generation backends targeting multiple machine architectures

• Compiler suites like GCC support multiple frontend languages (C, C++, Fortran, Java, Ada, Go) and provide multiple intermediate tree representations for the purposes of optimization and target machine abstraction
Conclusion
Recognize  Analyze  Transform

Frontend  Backend
Thank you!

Good luck!