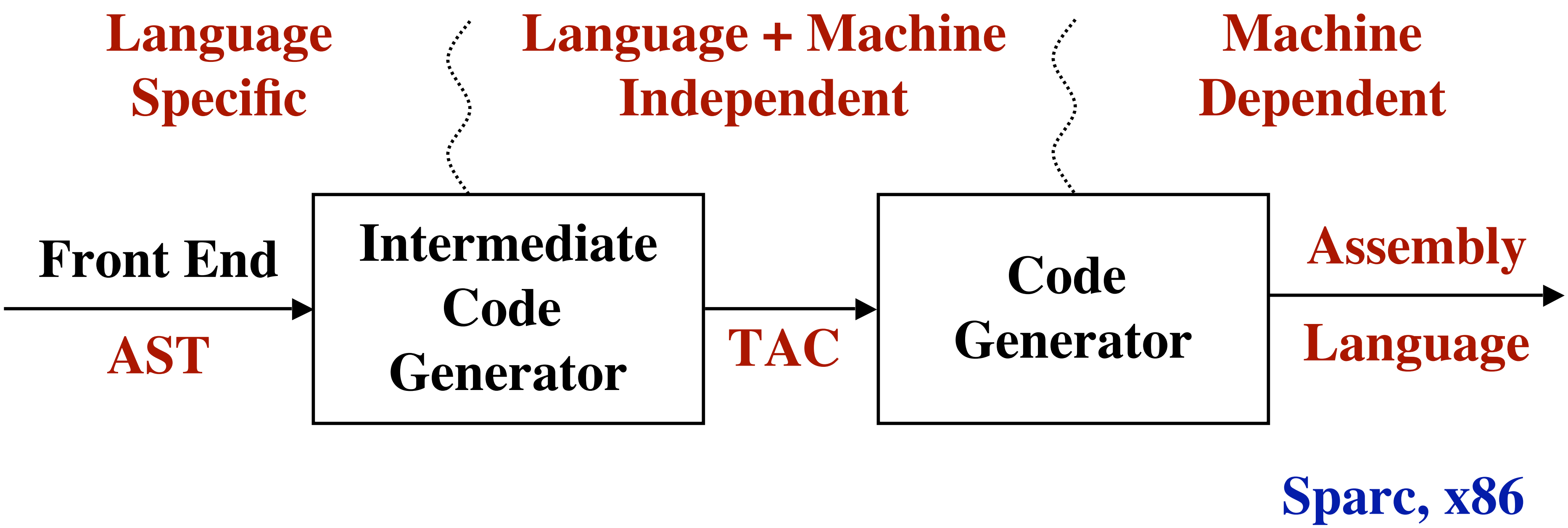
CMPT 379
Compilers

Anoop Sarkar

http://www.cs.sfu.ca/~anoop

TAC: Intermediate Representation

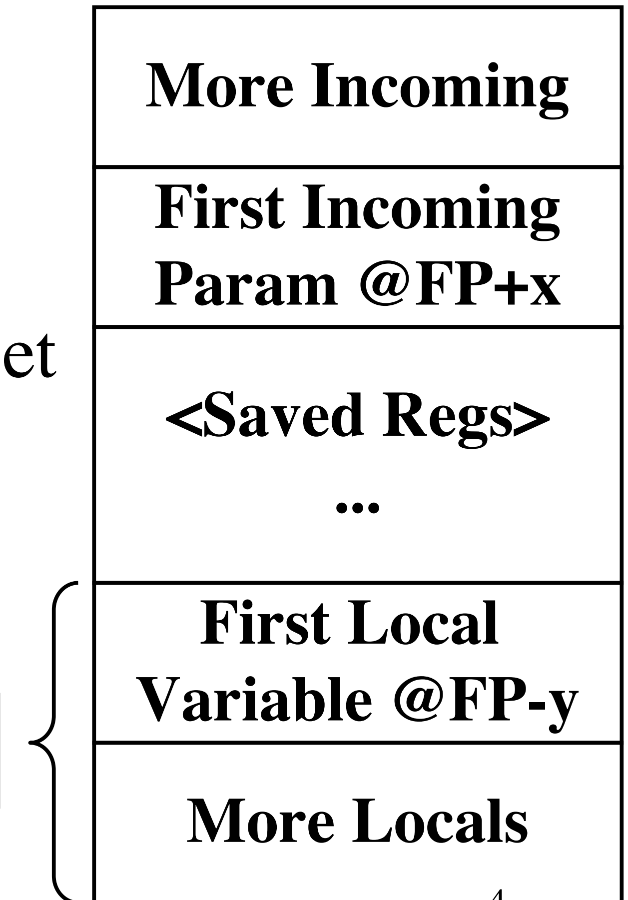


TAC: 3-Address Code

- Instructions that operate on named locations and labels
  - Mini-ISA or “generic assembly”
- Locations
  - Every location is some place to store 4 bytes
    - Pretend we can make infinitely many of them
  - Either on stack frame:
    - You assign offset (plus other information possibly)
  - Or global variable
    - Referred to by global name
- Labels (you generate as needed)

Function arguments

- Compute offsets for all incoming arguments, local variables and temporaries
  - Incoming arguments are at offset x, x+4, x+8, …
  - Locals+Temps are at –y, -y-4, -y-8,…
- Compute → **Frame Size**



Computing Location Offsets

```
class A {
 void f (int a /* @x+4 */,
            int b /* @x+8 */,
            int c /* @ x+12 */) {
   int s; // @-y-4
   if (c > 0) {
            int t; … // @-y-8
   } else {
            int u;     // @-y-12
            int t; … // @-y-16
   }
 }
}
```

Location offsets for temporaries are ignored on this slide

← You could reuse @-y-8 here, but okay if you don't

TAC Instructions (I)

- Assignment
- rhs can be
  - Location
  - String Constant
  - Integer Constant
  - Label

- Example:

  t2 := t1;

  t3 := “Hello”

  t5 := 42;

  t7 := L1;

```
Code.Append(
   new LoadStringConstant(
         /*t3=*/GenTempVar(), “Hello”));
```

TAC Instructions (II)

- Arithmetic
  - Binary add, sub, multiply, divide, modulo
- Equality (eq)
- Relational (lt)
- Logical (and, or)

- Labels and branches:
  - Insert label in TAC stream

    L4:
  - Unconditional branch

    goto L4
  - Conditional branch

    ifz t1 **goto** L3

TAC Instructions (III)

- Preparing function calls

  param t1;

  (eval left to right)

  (push right to left)

  pop n

- Calling methods
- Label vs. Address

  call

- Void vs. nonvoid

  t1 = call L3

  call t3 (akin to jump return)

  return t3 (t3 is the return **value**)

TAC Instructions (IV)

- Defining functions
  - BeginFunc <n>
    - Enter function, specify or forward-declare stack frame size
  - EndFunc
  - Return
  - Return t3

- Loads and Stores
  - Optional integer offset
  - Examples:

    t2 = *(t4)

    *(t5+4) = t6

- Unary minus, logical not

  t2 := not t3

What TAC doesn't give you

- Array indexing (bounds check)
- Two or n-dimensional arrays
- Relational <=, >=, >, …
- Conditional branches other than **ifz**
- Field names in records/structures
  - Use base+offset load/store
- Object data and method access

```c
int gcd(int x, int y)
{
   int d;
   d = x - y;
   if (d > 0)
      return gcd(d, y);
   else if (d < 0)
      return gcd(x, -d);
   else
      return x;
}
```

```
gcd:
      BeginFunc 32 ;
      tmp0 := x - y ;
      d := tmp0 ;
      tmp1 := 0 ;
      tmp2 := tmp1 < d ;
      ifz tmp2 goto L0 ;
      param y #1 ;
      param d #0 ;
      tmp3 := call gcd ;
      pop 8 ;
      return tmp3 ;
      goto L1 ;
L0:
      tmp4 := 0 ;
          ….
L1:
      EndFunc ;
```

```
int factorial(int n)
{
  if (n <=1 ) return 1;
  return n*factorial(n-1);
}

void main()
{
   print(factorial(6));
}
```

```
factorial:
    BeginFunc 32 ;
    tmp0 := 1 ;
    tmp1 := n lt tmp0 ;
    tmp2 := n eq tmp0 ;
    tmp3 := tmp1 or tmp2 ;
    ifz tmp3 goto L0 ;
    tmp4 := 1 ;
    Return tmp4 ;
L0:
    tmp5 := 1 ;
    tmp6 := n minus tmp5 ;
    param tmp6 #0 ;
    tmp7 := call factorial ;
    pop 4 ;
    tmp8 := n * tmp7 ;
    return tmp8 ;
    EndFunc ;
```

Short-circuiting Booleans

- More complex if statements:
  - if (a or b and not c) { … }
- Typical sequence:

  t1 := not c

  t2 := b and t1

  t3 := a or t2

- Short-circuit is possible in this case:
  - if (a and b and c) { … }
- Short-circuit sequence:

  t1 := a

  ifz t1 goto L0 /* sckt */

  goto L4

  L0: t2 := b

  ifz t2 goto L1

```
void main() {
    int i;
    for (i = 0; i < 10; i = i + 1)
        print(i);
}
```

```
main:
        BeginFunc 24 ;
        tmp0 := 0 ;
        i := tmp0 ;
L0:
        tmp1 := 10 ;
        tmp2 := i < tmp1 ;
        ifz tmp2 goto L1 ;
        param i #0 ;
        call PrintInt ;
        pop 4 ;
        tmp3 := 1 ;
        tmp4 := i + tmp3 ;
        i := tmp4 ;
        goto L0 ;
L1:
        EndFunc ;
```

```
void foo(int[] arr)
      { arr[1] = arr[0] * 2; }
```

```
foo:
  BeginFunc 48 ;
    tmp0 := 1 ;
    tmp1 := 4 ;
    tmp2 := tmp1 * tmp0 ;
    tmp3 := arr + tmp2 ;
    tmp4 := *(tmp3) ;
    tmp5 := 0 ;
    tmp6 := 4 ;
    tmp7 := tmp6 * tmp5 ;
    tmp8 := arr + tmp7 ;
    tmp9 := *(tmp8) ;
    tmp10 := 2 ;
    tmp11 := tmp9 * tmp10 ;
    tmp4 := tmp11 ;
  EndFunc ;
```

Wrong

```
foo:
  BeginFunc 44;
    t0 := 1;
    t1 := 4;
    t2 := t1 * t0;
    t3 := arr + t2;
    t4 := 0;
    t5 := 4;
    t6 := t5 * t4;
    t7 := arr + t6;
    t8 := *(t7);
    t9 := 2;
    t10 := t8 * t9;
    *(t3) := t10;
  EndFunc;
```

Correct

Backpatching

- Easiest way to implement the translations is to use two passes
- In one pass we may not know the target label for a jump statement
- *Backpatching* allows us to do it in one pass
- Generate branching statements with the targets of the jumps temporarily unspecified
- Put each of these statements into a list which is then filled in when the proper label is determined

Correctness vs. Optimizations

- When writing backend, correctness is paramount
  - Efficiency and optimizations are secondary concerns at this point
- Don’t try optimizations at this stage

Basic Blocks

- Functions transfer control from one place (the caller) to another (the called function)
- Other examples include any place where there are branch instructions
- A *basic block* is a sequence of statements that enters at the start and ends with a branch at the end
- Remaining task of code generation is to create code for basic blocks and branch them together

Summary

- TAC is one example of an intermediate representation (IR)
- An IR should be close enough to existing machine code instructions so that subsequent translation into assembly is trivial
- In an IR we ignore some complexities and differences in computer architectures, such as limited registers, multiple instructions, branch delays, load delays, etc.