MACHINE CODE GENERATION

- Instruction Selection
- Register Allocation and Assignment
- Optimization

Issues:
- Complexity of Target Machine
- Level of Translation: expression, statement, basic block, routine, program?
- Management of Scarce Resources
APPRAOCHES TO INSTRUCTION SELECTION

For **RISC** targets, translate one IR instruction to one or more target instructions.

For **CISC** targets, translate several IR instructions to one target instruction.
Example Source: \( a := b \) (assuming \( a,b \) in frame)

3-addr IR: 
\[
t_1 = fp-12 \\
t_2 = *t_1 \\
t_3 = fp+8 \\
* t_3 = t_2
\]

Typical Tree IR: 
```
    mov \
   / \ \\
  /   \ \\
 mem   mem
 |   |  \\
+   +  \\
/ \  / \ \\
fp  8 fp -12
```

extreme RISC: 
```
add %fp,-12,%r3  
ld [%r3], %r7  
add %fp,8,%r4  
st %r7,[%r4]
```

moderate RISC: 
```
ld [%fp-12], %r7  
st %r7, [%fp + 8]
```

CISC: 
```
move [%fp-12],[%fp+8]
```
Task: Manage scarce resources (registers) in environment with imperfect information (static program text) about dynamic program behavior.

General aim is to keep frequently-used values in registers as much as possible, to lower memory traffic. Can have a large effect on program performance.

Variety of approaches are possible, differing in sophistication and in scope of analysis used.

Allocator may be unable to keep every “live” variable in registers; must then “spill” variables to memory. Spilling adds new instructions, which often affects the allocation analysis, requiring a new iteration.

If spilling is necessary, what should we spill? Some heuristics:

- Don’t spill variables used in inner loops.
- Spill variables not used again for “longest” time.
- Spill variables which haven’t been updated since last read from memory.
To determine how long to keep a given variable (or temporary) in a register, need to know the range of instructions for which the variable is live.

A variable or temporary is live immediately following an instruction if its current value will be needed in the future (i.e., it will be used again, and it won’t be changed before that use).

Example:

```
! temps live after instruction:
movI 3, $T2   ! $T2
movI $T2, $T3 ! $T2 $T3
addI $T3, 4, $T4 ! $T2 $T4
addI $T2, $T4, $T4 ! $T4
movI $T4, $RET ! (nothing)
```

It’s easy to calculate liveness for a consecutive series of instructions without branches, just by working backwards.
But if a value can stay in a register over a jump, things get harder, e.g.:

```
! temps live after instruction:
1 movI 0, $T1 ! $T1 $T3
2 L1: addI $T1, 1, $T2 ! $T2 $T3
3 addI $T3, $T2, $T3 ! $T2 $T3
4 mulI $T2, 2, $T1 ! $T1 $T3
5 cmpI $T1, 1000 ! $T1 $T3
6 jl L1 ! $T1 $T3
7 movI $T3, $RET ! (nothing)
```

To calculate liveness in this case requires iterative flow analysis and the result is only conservative approximation to true liveness.

The live range of a variable is the set of instructions which leave it live. E.g. here live range of $T1$ is \{1, 4, 5, 6\}; live range of $T3$ is \{1, \ldots, 6\}.

Basic idea: If two variables have disjoint live ranges, they can occupy the same physical register.

So in both examples, 2 physical registers suffice to allocate all temporaries without spilling.
LINEAR SCAN ALLOCATION

Using live ranges turns out to be computationally expensive (more later).

A simple alternative is to approximate each live range by a live interval. This is the consecutive interval of instructions between the first and last use of each temporary. Example:

```plaintext
! temps live after instruction:
1  movI 0, $T1  ! $T1 $T3
2 L1: addI $T1, 1, $T2  ! $T2 $T3
3    addI $T3, $T2, $T3  ! $T2 $T3
4    mulI $T2, 2, $T1  ! $T1 $T3
5    cmpI $T1, 1000  ! $T1 $T3
6    jl L1  ! $T1 $T3
7    movI $T3, $RET  ! (nothing)

Live ranges: $T1: 1,4,5,6  $T2:2,3  $T3:1,2,3,4,5,6

Live intervals: $T1: [1,6]  $T2: [2,3]  $T3: [1,6]
```

(Revised) Basic idea: if two temporaries have non-overlapping live intervals, they can occupy the same physical register.
LINEAR SCAN ALLOCATION ALGORITHM DETAILS

1. Compute \( \text{startpoint}[i] \) and \( \text{endpoint}[i] \) of live interval \( i \) for each temporary. Store the intervals in a list in order of increasing start point.

2. Initialize set \( \text{active} := \emptyset \) and pool of free registers = all usable registers.

3. For each live interval \( i \) in order of increasing start point:
   (a) For each interval \( j \) in \( \text{active} \), in order of increasing end point
       - If \( \text{endpoint}[j] \geq \text{startpoint}[i] \) break to step 3b.
       - Remove \( j \) from \( \text{active} \).
       - Add \( \text{register}[j] \) to pool of free registers.
   (b) Set \( \text{register}[i] := \) next register from pool of free registers, and remove it from pool. (If pool is already empty, need to spill.)
   (c) Add \( i \) to \( \text{active} \), sorted by increasing end point.
L I N E A R  S C A N  A L L O C A T I O N  F O R  F A B

- For HW4, the live interval computation code is already provided for you. It maps each Name, Temp, Arg and RetReg operand to its live interval.
- Note that if an operand is defined but not used, its live interval is empty, and it does not appear in the map.
- When allocating a register in step 3(b), need to consider different register classes (callee-save vs. caller-save). Simple strategy: use callee-save register if operand’s live range includes a call instruction; otherwise use caller-save register if available. (This way, we never have to worry about performing caller saves at all.)
- The Arg and RetReg operands must be pre-allocated to fixed registers. These registers can still be used to store other operands with non-overlapping ranges, but at step 3(b) we must look ahead to make sure that the interval doesn’t overlap a pre-allocated interval.
- Once a spill is necessary, simply commit the spilled operand to a stack slot once and for all.