# Register Allocation

- In last chapter, we converted the AST into a low-level code format
  - To simplify low-level code generation and provide max flexibility for optimization, we assumed infinite virtual registers were available
  - Late in the compilation process, we need to map these virtual registers into physical registers, i.e., figure out where the data will really be stored by the hardware
  - This is the job of register allocation
    - It decides which virtual register will fit in which physical register
    - It decides what to do when all virtual registers won't fit
      - Typically called register spilling

# Register Allocation

- Two foundational papers on register allocation
  - Chaitin, et al – *Register Allocation via Coloring* (1980)
  - Chow/Hennessy – *Register Allocation by Priority-Based Coloring* (1984)

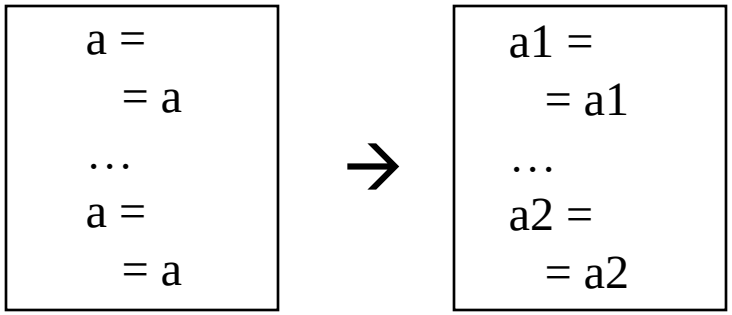
- There are 2 primary models to register allocation
  - All variables have a home location in memory
    - This is the method Chow/Hennessy, and what we saw in gcc
    - If the register allocator can put them in register, it will
      - Otherwise, they reside in memory
    - If the register allocator needs to spill the variable, it is just spilled to its home location

# Register Allocation

- 2 primary models to register allocation (cont)
  - If a local variable is "promoted" to register, no memory location is allocated for it – "promoted" globals are loaded into register at first use, and written back after last definition
    - Variables typically live and die in register
    - This is the method used by Chaitin, the IMPACT compiler, and what I have assumed in this course
    - If the register allocator needs to spill, it allocates a special spill space in the stack frame
      - Register allocator adds store/load pair
      - Globals can be spilled to their home location
    - Register allocation in this model is typically done late in the compilation process
      - Prevents false restrictions to optimization

# Register Allocation

- Register allocation is typically done on a function-by-function basis – assume all global variables are in their home location at beginning of function, and must store at end
- The basic process provides that a particular variable will be assigned to the same register for the entire function
  - Globals can be in different registers in different functions
  - Improvements on this approach
    - If a variable is set multiple times within a function, and each version is independent, treat as 2 separate variables (register renaming)
    - Live-range splitting – will look at later



# Register Allocation

- General register allocation process
  1. Perform detailed live range analysis
  2. Create interference graph of live ranges
  3. Perform classic graph coloring to assign registers

- Live range analysis
  - A virtual register is “live” when it contains something useful
    - After it has been written to, and before last use
  - The register’s “live range” is determined through def-use analysis
    - For every use of a register, determine which definition(s) of the register set the value
    - Complicated by branch structure and multiple defs
    - Granularity – either by operation (Chaitin) or by basic block (Chow/Hennessy)

# Register Allocation

- Create the interference graph
  - Nodes are live ranges – arcs are conflicts between them
  - Live range analysis has determined during which operations the variable is alive
  - If two variables are live during the same operation, then they conflict and an arc must be added to the graph
  - What does the implementation of this look like?

- Color the graph
  - The problem has been shown to be NP-Complete (exponential order)
  - Cannot solve for optimal solution – use heuristics to find a "good" solution
    - What's a heuristic?

# Register Allocation

- Graph coloring (cont)
  - Fortunately the problem is a bit different than classic coloring
    - The goal is not to determine the minimum number of colors required to color the graph
    - We have a fixed number of registers
      - The goal is to get the best subset of live ranges allocated to a physical register as possible
      - Thus, it is more an optimization problem of what is best solution if full coloring not doable
  - Heuristic approach
    - Assume we have n registers
    - If degree of all nodes <= n – 1, then problem can be readily solved – simply allocate higher degree nodes first
    - If some nodes have a degree > n – 1, may not be able to allocate all registers
      - Heuristics determine which are selected

# Register Allocation

- Graph coloring (cont)
  - If a live range has degree < n–1, it is unconstrained
    - An unconstrained node can always be colored, no matter what other coloring decisions were made prior
    - Unconstrained nodes removed from graph
  - Constrained nodes are the problem
    - Must heuristically decide how to prioritize and assign
    - Typically done by cost-benefit analysis
      - Costs – how many do you interfere with
      - Benefits
        - How many loads and stores are eliminated
  - Each live range assigned a priority based on the heuristic
    - Constrained nodes assigned registers based on this priority

# Register Allocation

- Graph coloring (cont)
  - Keep assigning highest priority registers till out of registers
  - Then assign unconstrained (guaranteed to work)
  - Chaitin stops there
- Chow splits live ranges – as soon as it becomes uncolorable
  - If no registers are left that are open throughout life
  - Perhaps, however, a subset of its life could be allocated a register
    - Maybe there's just one use way below other uses that is causing the problem
  - To split a live range, its value will have to be saved at end of 1st piece, and loaded back at beginning of next live range
  - Splitting may make live range unconstrained
  - May stay constrained, but not undoable yet
    - Need to re-compute priority

# Register Allocation

- Those local variables allocated to register will live and die in the register
- Without live-range splitting, those variables not allocated to register will be spilled
  - Register allocator allocates spill space in local stack frame
  - Must insert stores and loads at each define and use
- With live-range splitting, live ranges keep getting split until either the partial range is allocated, or live range priority is negative (no benefit to allocating to register)
  - If partial range allocated, patch-up code inserted and then treated like allocated variable
  - Live range pieces not allocated must have stores and loads inserted

# Register Allocation

- Register allocation is usually quite effective
  - With 32 registers, most functions will have little trouble allocating registers, and little spill code required
    - The performance difference between a basic non-splitting scheme and more elegant scheme may be minor
  - With 8 (actually 6 or 7) registers, there may be significant spill code required
    - Quality of register allocator more critical
    - We found that with X86 the register allocator still usually did a pretty decent job

# Scheduling

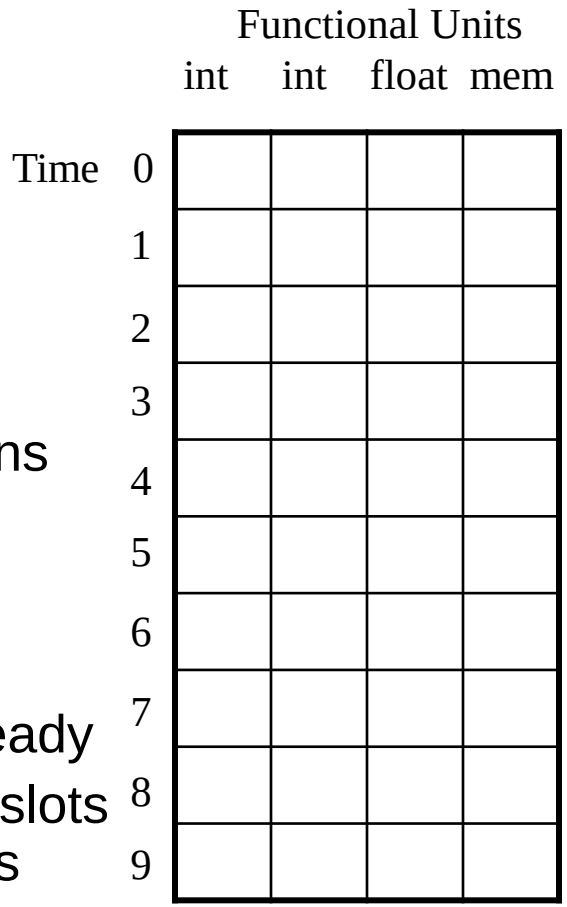
- Code Scheduling
  - Reordering instructions to maximize instruction-level parallelism (ILP)
    - Allow code to flow through pipelined architecture smoothly
    - Reduces effects of dependences within the code
  - Must maintain original program semantics
  - May be done one basic block at a time

- In microcode world, called microcode compaction

- NP complete problem, so develop heuristic methods
  - We will look at two
    - Non-backtracking: List Scheduling
    - Backtracking: Slack Scheduling

# List Scheduling

- Data Structures
  - Queues
    - Ready
    - Pending Ready
    - Not Ready
  - Scheduling Array – resources vs time
- Algorithm
  1. Create dependence graph of operations
  2. Assign priority to each operation
  3. While unscheduled operation remain
     - Start a new scheduling cycle
     - Move any poss from pending to ready
     - Place ready operations into cycle slots according to priority and resources
     - Update status of unscheduled ops
- Only real complexity is in assigning priority

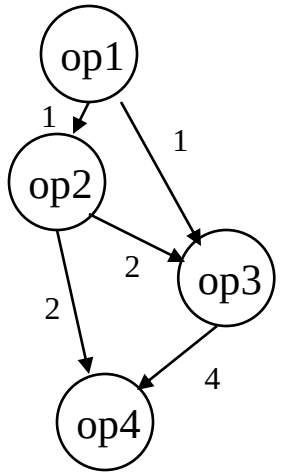


# List Scheduling

- Need to create a dependence graph of operations, with arcs representing operations which are dependent
  - Requires dependence analysis

```
op1:  r1 = r1 + 1
op2:  r2 = r1 * 2
op3:  r3 = r1 / r2
op4:  r4 = r2 + r3
```

- Scheduling an instruction
  - Highest priority instruction in ready queue which matches available resources should be scheduled
  - Once scheduled, update dependent instructions
    - Remove one incoming arc
      - When zero incoming, it is ready/pending
    - Update ready time



# List Scheduling

- Assigning priorities
  - Start with dependence graph
  - Compute earliest ready time
  - Compute deadline for issue

- Use heuristic based upon ready time and deadline time to assign priority
  - Deadline - Ready is important measure
    - This is what slack scheduling uses

# Slack Scheduling

- Backtracking approach, came from realm of software pipelining
- Again, create dependence graph and compute earliest / deadline times
- An instruction's slack = deadline - earliest
  - A slack of zero means there is no tolerance in where this instruction can be scheduled, without "breaking schedule"
- Instructions are prioritized and scheduled by order of increasing slack

# Slack Scheduling

- Big difference here is that we have a target schedule we are trying to achieve
  - The dependence height of the dependence graph gives the minimum possible schedule length
  - So, try to create a schedule of this length, filling instructions into slots in the schedule
  - If schedule impossible, backtrack and try again
    - Increase target schedule length and keep trying
      - Just start over totally with new target
      - Insert extra cycle into existing schedule and continue

# Slack Scheduling

- What has to be done when instruction scheduled?
  - Mark resources as used
  - All other instructions in its dependence chain must be updated
    - If an instruction could go in cycle 4 or 5, scheduling it in 5 will affect the earliest time of dependent operations
    - Big computation overhead

- What does backtracking entail?
  - Remove inst and free resources
  - Update all dependent operations to see if their slack changed