List Scheduling
int main() {
    int x = 4 + 5;
    return x;
}
Instruction Scheduling

- Modern architectures expose many opportunities for optimization
  - Some instructions require fewer cycles
  - Superscalar processing (multiple functional units)
  - Instruction pipelining
  - Speculative execution
- Primary obstacle: data dependencies
  - A stall is a delay caused by having to wait for an operand to load
- Scheduling: re-order instructions to improve performance
  - Maximize utilization and prevent stalls
  - Must not modify program semantics
  - Main algorithm: list scheduling

https://en.wikipedia.org/wiki/Superscalar_processor
Example

- Which program is preferable?
- Assumptions:
  - Loads and stores have a 3-cycle latency
  - Multiplications have a 2-cycle latency
  - All other instructions have a 1-cycle latency

```
1  loadAI [BP-4] => r1
4  add r1, r1 => r2
5  loadAI [BP-8] => r3
8  mult r2, r3 => r4
9  loadAI [BP-12] => r5
12 mult r4, r5 => r6
13 loadAI [BP-16] => r7
16 mult r6, r7 => r8
18 store AI r8 => [BP-20]
1  loadAI [BP-4] => r1
2  loadAI [BP-8] => r3
3  loadAI [BP-12] => r5
4  add r1, r1 => r2
5  mult r2, r3 => r4
6  loadAI [BP-16] => r7
7  mult r4, r5 => r6
9  mult r6, r7 => r8
11 store AI r8 => [BP-20]
```
Data Dependence

- **Data dependency** ($x = _; _ = x$)
  - Read after write
  - Hard constraint
- **Antidependency** ($_ = x; x = _$)
  - Write after read
  - Can rename to avoid (could require more register spills)
- **Dependency graph**
  - One for each basic block
    - Could have multiple roots; technically a forest of directed acyclic graphs (DAGs)
  - Nodes for each instruction
  - Edges represent data dependencies
    - Edge $(n_1, n_2)$ means that $n_2$ uses a result of $n_1$
List Scheduling

- Prep work
  - Rename to avoid antidependencies
  - Build data dependence graph
  - Assign priority for each instruction
    - Usually based on node height and instruction latency
      - Priority of a leaf node is its latency
      - Priority of a branch node is its latency plus the maximum priority of any immediate successor
    - Goal: prioritize instructions on the critical path
List Scheduling

- Iteratively build new schedule
  - A schedule is just a list of instructions in the order they are to be issued
    - Sometimes with “idle” cycles (no new instructions) marked with “-”
    - Example: “a, b, -, c, -, -” means “start instruction a on cycle one, b on cycle two, nothing on cycle three, c on cycle four, and then wait two more cycles for everything to finish”
  - Track a set of "ready" instructions
    - No remaining unresolved data dependencies; i.e., can be issued
  - For each cycle:
    - Check all currently executing instructions for any that have finished
      - Add any new "ready" dependents to set
    - Start executing a new "ready" instruction (if there are any)
      - Greedy algorithm: if multiple instructions are ready, choose the one with the highest priority
      - Helps to note the cycle where the instruction will finish
Example

- Schedule the following code:
  - Loads and stores have a 3-cycle latency
  - Multiplications have a 2-cycle latency
  - All other instructions have a 1-cycle latency

Original schedule:
  a, -, -, b, c, -, -, d, e, f, g, -, -
  (13 cycles)

New schedule:
  c, a, g, d, e, b, f, -, -
  (9 cycles)
Instruction Priorities

• Usually based on node height and latency first
  – Minimizes critical path
• Many methods for tie-breaking
  – Node's rank (# of successors; breadth-first search)
  – Node's descendant count
  – Latency (maximize resource efficiency)
  – Resource ordering (maximize resource efficiency)
  – Source code ordering (minimize reordering)
  – **No clear winner here!**
Tradeoffs

• **Forward vs. backward** list scheduling
  
  - Backward scheduling: build schedule in reverse
    • Choose last instruction on critical path first
    • Schedule from roots to leaves instead of leaves to roots
    • Similar to backward data flow analysis
  
  - List scheduling is cheap; just run several variants to see which works better for particular code segments
Tradeoffs

- Instruction scheduling vs. register allocation
  - Fewer registers → more sequential code
  - More registers → more possibilities for parallelism
  - Scheduling can also impact number of spills.loads

![Diagram showing tradeoffs between fewer and more registers]
Regional scheduling

- Usually based on local list scheduling
- Extended using various techniques
  - Analyze extended basic blocks (chains of basic blocks)
  - Detect hot traces or paths using profile information
  - Sometimes need to insert compensation code
  - Sometimes need to clone entire blocks
- Particularly important for loops
  - Focus on core kernel of the loop
  - Constrained by loop-carried dependencies
Exercise

• Schedule this program from earlier

• Assumptions:
  – Loads and stores have a 3-cycle latency
  – Multiplications have a 2-cycle latency
  – All other instructions have a 1-cycle latency

a) loadAI [BP-4] => r1
b) add r1, r1 => r2
c) loadAI [BP-8] => r3
d) mult r2, r3 => r4
e) loadAI [BP-12] => r5
f) mult r4, r5 => r6
g) loadAI [BP-16] => r7
h) mult r6, r7 => r8
i) storeAI r8 => [BP-20]
Exercise

- Schedule this program from earlier
- Assumptions:
  - Loads and stores have a 3-cycle latency
  - Multiplications have a 2-cycle latency
  - All other instructions have a 1-cycle latency

a) loadAI [BP-4] => r1
b) add r1, r1 => r2
c) loadAI [BP-8] => r3
d) mult r2, r3 => r4
e) loadAI [BP-12] => r5
f) mult r4, r5 => r6
g) loadAI [BP-16] => r7
h) mult r6, r7 => r8
i) storeAI r8 => [BP-20]