CS 432
Fall 2018

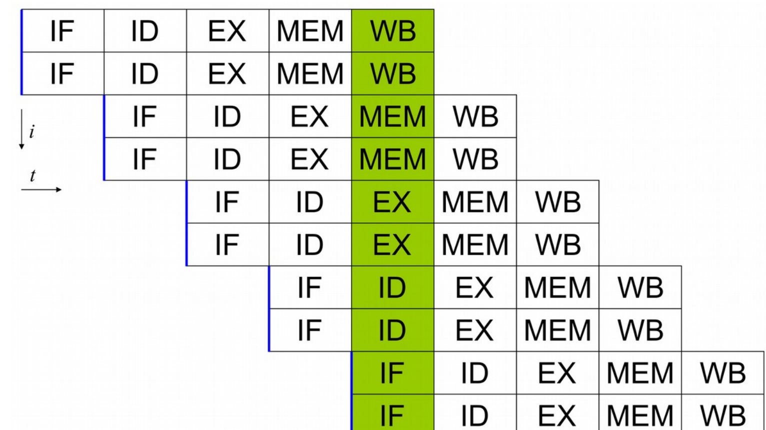
Mike Lam, Professor

https://xkcd.com/1542/

List Scheduling

Instruction Scheduling

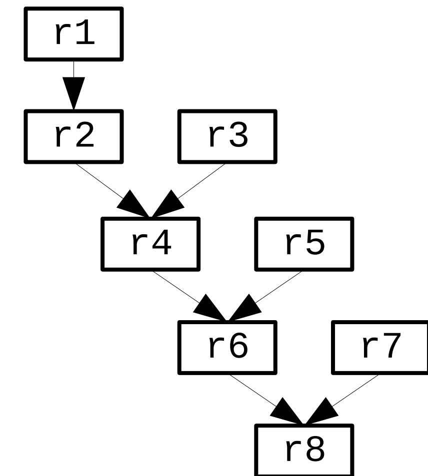
- Modern architectures expose many opportunities for optimization
  - Some instructions require fewer cycles
  - Superscalar processing (multiple functional units)
  - Instruction pipelining
  - Speculative execution
- Primary obstacle: data dependencies
  - A stall is a delay caused by having to wait 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

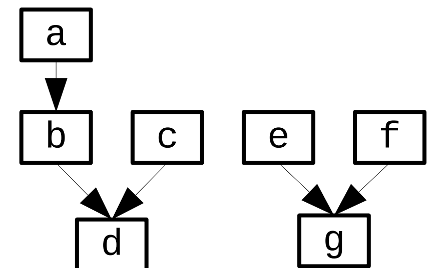



```
1    loadAI [BP-4] => r1
4    add r1, r1 => r2
5    loadAI [BP-8] => r3
8    mult r2, r3 => r4
9    loadAI [BP-12] => r5
12   mult r4, r5 => r6
13   loadAI [BP-16] => r7
16   mult r6, r7 => r8
18   store AI r8 => [BP-20]
```

```
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

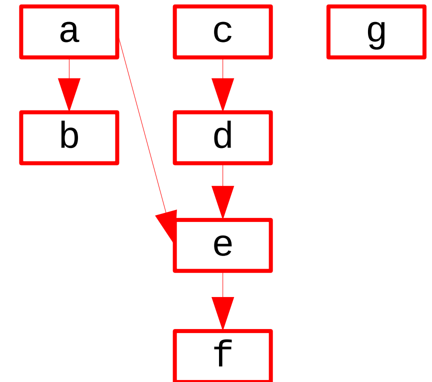


List Scheduling

- Prep work
  - Rename to avoid antidependencies
  - Build data dependence graph
  - Assign priority for each instruction
    - Usually based on node height and instruction latency
    - Goal: prioritize instructions on the critical path
- Iteratively build new schedule
  - 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

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

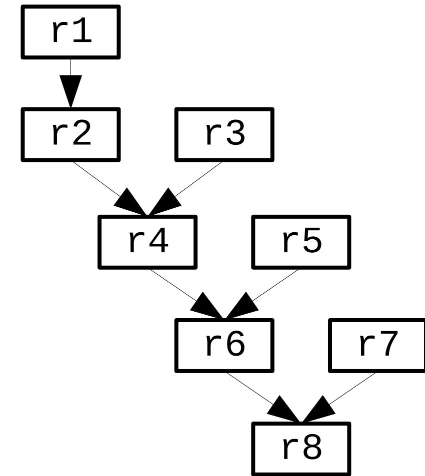



```
[1]     a) loadAI [BP-4] => r2
[4]     b) storeAI r2 => [BP-8]
[5]     c) loadAI [BP-12] => r3
[8]     d) add r3, r4 => r3
[9]     e) add r3, r2 => r3
[10]    f) storeAI r3 => [BP-16]
[11]    g) storeAI r7 => [BP-20]
```

```
[1] c) loadAI [BP-12] => r3
[2] a) loadAI [BP-4] => r2
[3] g) storeAI r7 => [BP-20]
[4] d) add r3, r4 => r3
[5] e) add r3, r2 => r3
[6] f) storeAI r3 => [BP-16]
[7] b) storeAI r2 => [BP-8]
```

Example

- 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




```
loadAI [BP-4] => r1
add r1, r1 => r2
loadAI [BP-8] => r3
mult r2, r3 => r4
loadAI [BP-12] => r5
mult r4, r5 => r6
loadAI [BP-16] => r7
mult r6, r7 => r8
store AI r8 => [BP-20]
```

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

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

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