CS 480
Fall 2015

Mike Lam, Professor

List Scheduling

Instruction Scheduling

- Modern architectures expose many opportunities for optimization
  - Superscalar processing (multiple functional units)
  - Some instructions require fewer cycles
  - Instruction pipelining
  - Speculative execution
- ***Stall*** – delay caused by having to wait for an operand to load
- Scheduling: re-order instructions to improve speed without changing results
  - Must not modify program semantics
    - Issue: data dependencies
  - May re-order other statements to maximize utilization and prevent stalls
  - Main algorithm: **list scheduling**

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
  - Graph for each basic block
  - Nodes for each instruction
  - Edges represent data dependencies
    - Edge (n_1, n_2) means that n_2 uses a result of n_1

Example

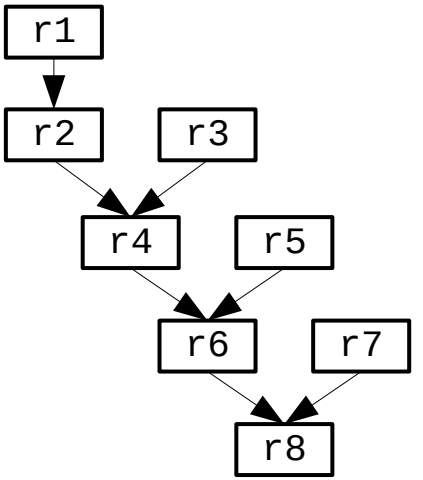
- Which program is preferable?
- Assumptions:
  - Loads and stores have a 3-cycle latency
  - Multiplications have a 2-cycle latency
  - All other instructions have a 1-cycle latency

```
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]
```

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

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

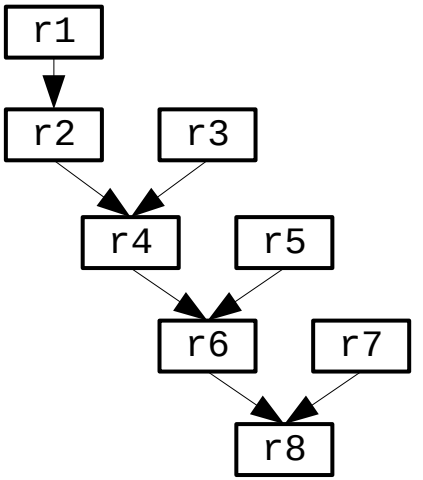



```
loadAI [BP-4] => r1
add r1, r1 => r2
loadAI [BP-8] => r3
mult r2, r3 => r4
loadAI [BP-12] => r5
mult r4, r5 => r6
loadAI [BP-16] => r7
mult r6, r7 => r8
store AI r8 => [BP-20]
```

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

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]
```

List Scheduling

- Prep work
  - Rename to avoid antidependencies
  - Build data dependence graph
  - Assign priority for each instruction
    - Usually based on node height (minimize critical path length)
- Iteratively build 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)
      - Choose the one with the highest priority

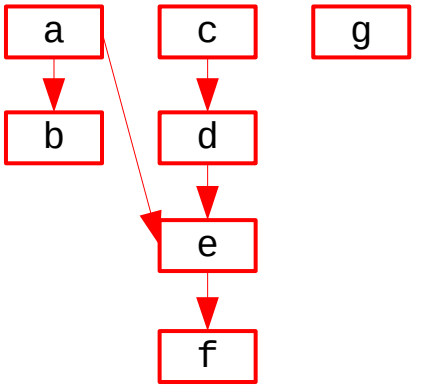
Example

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

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

Example

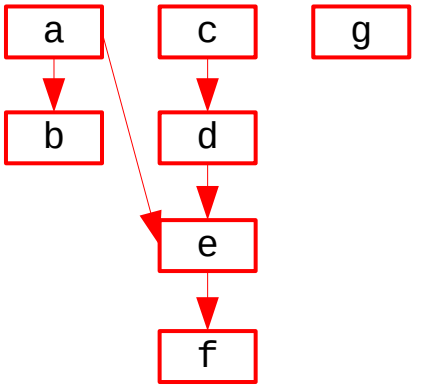
- Schedule the following code:
  - Loads and stores have a 3-cycle latency
  - Multiplications have a 2-cycle latecy
  - All other instructions have a 1-cycle latency




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

Example

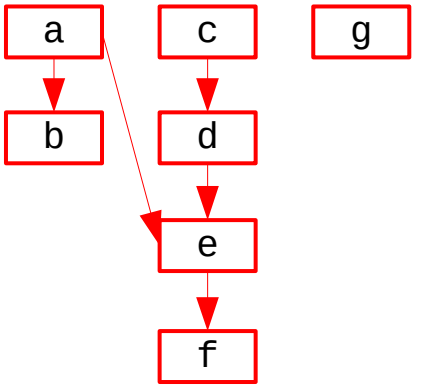
- Schedule the following code:
  - Loads and stores have a 3-cycle latency
  - Multiplications have a 2-cycle latecy
  - All other instructions have a 1-cycle latency




```
[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]
```

Example

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

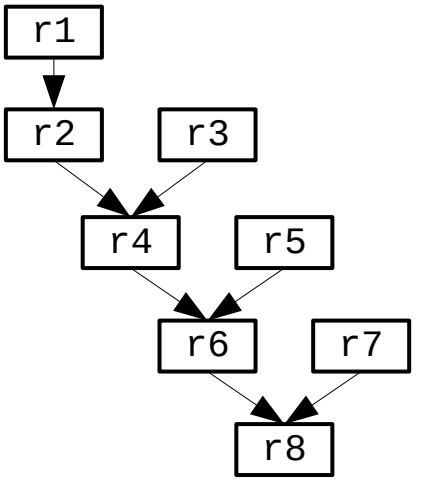



```
[1]     a) loadAI [BP-4] => r2
[4]     b) storeAI r2 => [BP-8]
[5]     c) loadAI [BP-12] => r3
[8]     d) add r3, r4 => r3
[9]     e) add r3, r2 => r3
[10]    f) storeAI r3 => [BP-16]
[11]    g) storeAI r7 => [BP-20]
```

```
[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 => r4
[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 latecy
  - 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 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
- Forward vs. backward list scheduling
  - List scheduling is cheap; just run several variants to see which works better for particular code segments