Instruction Scheduling

Instruction Cycles

Not all instructions are equal.

- Load/Store: 3 cycles
- Add: 1 cycle
- Multiply: 2 cycles

Multi-cycle instructions can be interleaved in the processor pipeline

Examples with Cycles

Start		Operations
1	3	loadAI r_{arp}, @a ⇒ r_1
4	1	add r_1, r_1 ⇒ r_1
5	3	loadAI r_{arp}, @b ⇒ r_2
8	2	mult r_1, r_2 ⇒ r_1
10	3	loadAI r_{arp}, @c ⇒ r_2
13	2	mult r_1, r_2 ⇒ r_1
15	3	loadAI r_{arp}, @d ⇒ r_2
18	2	mult r_1, r_2 ⇒ r_1
20	3	storeAI r_1 ⇒ r_{arp}, @a

22 (a) Original Code

Start		Operations
1	1	loadAI r_{arp}, @a ⇒ r_1
2	1	loadAI r_{arp}, @b ⇒ r_2
3	1	loadAI r_{arp}, @c ⇒ r_3
4	1	add r_1, r_1 ⇒ r_1
5	1	mult r_1, r_2 ⇒ r_1
6	1	loadAI r_{arp}, @d ⇒ r_2
7	2	mult r_1, r_3 ⇒ r_1
9	2	mult r_1, r_2 ⇒ r_1
11	3	storeAI r_1 ⇒ r_{arp}, @a

13 (b) Scheduled Code

Constructing a Dependence Graph

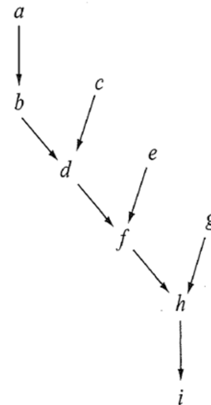
Start with the last instruction of a block of code.

Work up, adding arrows indicating which instructions depend on which data.

Dependence Graph (Finally a real tree!)

a:	loadAI	r_{arp}, @a	⇒	r_1
b:	add	r_1, r_1	⇒	r_1
c:	loadAI	r_{arp}, @b	⇒	r_2
d:	mult	r_1, r_2	⇒	r_1
e:	loadAI	r_{arp}, @c	⇒	~~r_3~~ r_2
f:	mult	r_1, r_2	⇒	r_1
g:	loadAI	r_{arp}, @d	⇒	r_2
h:	mult	r_1, r_2	⇒	r_1
i:	storeAI	r_1	⇒	r_{arp}, @a

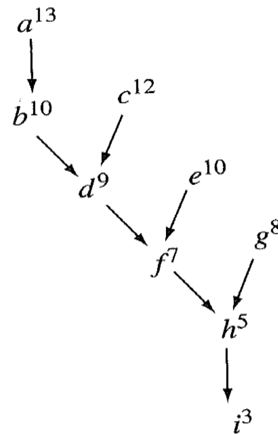
(a) Example Code



(b) Its Dependence Graph

Annotated Dependence Graph

Next, add sums of latencies to the nodes of the graph:
We will use this information later for inserting nodes into a (max) priority queue



List Scheduling Algorithm

A greedy algorithm.
Has been used since Babbage (I swear!)
Ready is a priority using the latency times as a criteria for insert,

```
Cycle ← 1
Ready ← leaves of D  // Max PriorityQueue
Active ← ∅           // Queue
while (Ready ∪ Active ≠ ∅)
   for each op ∈ Active
      if S(op) + delay(op) ≤ Cycle then
         remove op from Active
         for each successor s of op in D
            if s is ready
               then add s to Ready
   if Ready ≠ ∅ then
      op <- poll(Ready)
      S(op) ← Cycle
      add op to Active
   Cycle ← Cycle + 1
```