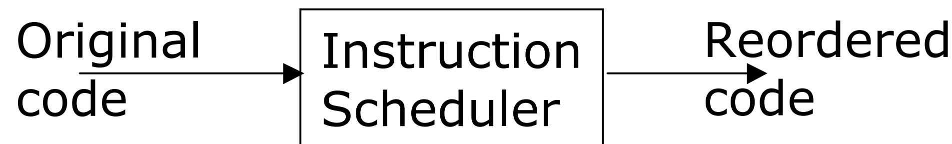
Instruction Scheduling

Instruction scheduling



- Reorder operations to reduce running time
  - Different operations take different number of cycles
    - Referencing values not yet ready causes operation pipeline to stall
  - Processors can issue multiple instructions every cycle
    - VLIW processors: can issue one operation per functional unit in each cycle
    - Superscalar processors: tries to issue the next k instructions if possible

Instruction Scheduling Example

Assumptions: memory load: 3 cycles; mult: 2 cycles; other: 1 cycle

start	
1	loadAI rarp, @w ➔ r1
4	add r1, r1 ➔ r1
5	loadAI rarp, @x ➔ r2
8	mult r1, r2 ➔ r1
9	loadAi rarp, @y ➔ r2
12	mult r1, r2 ➔ r1
13	loadAI rarp, @z ➔ r2
16	mult r1, r2 ➔ r1
18	storeAI r1 ➔ rarp, 0

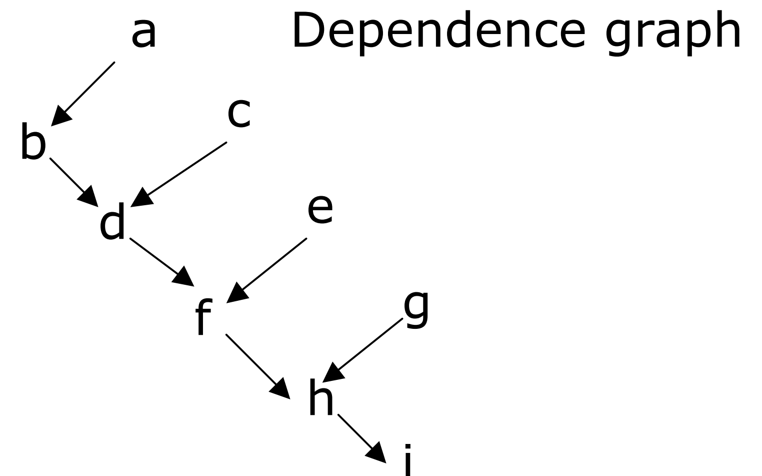
start	
1	loadAI rarp, @w ➔ r1
2	loadAI rarp, @x ➔ r2
3	loadAi rarp, @y ➔ r3
4	add r1, r1 ➔ r1
5	mult r1, r2 ➔ r1
6	loadAI rarp, @z ➔ r2
7	mult r1, r3 ➔ r1
9	mult r1, r2 ➔ r1
11	storeAI r1 ➔ rarp, 0

- Instruction level parallelism (ILP)
  - Independent operations can be evaluated in parallel
- Given enough ILP, a scheduler can hide memory and functional-unit latency
  - Must not violate original semantics of input code

Dependence Graph

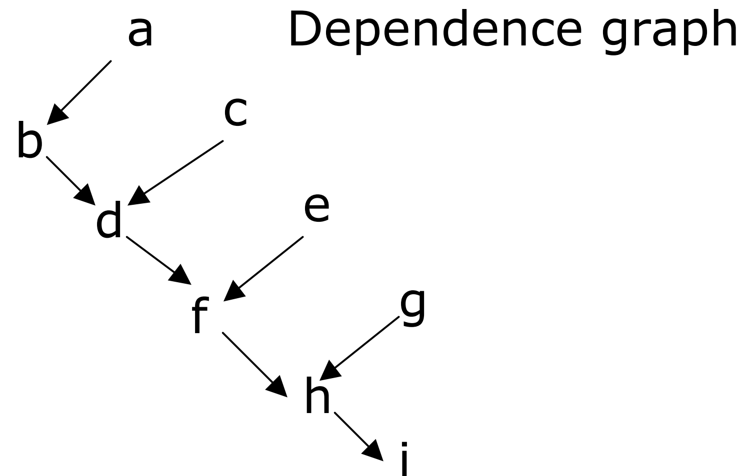
- Dependence/precedence graph G = (N,E)
  - Each node n ∈ N is a single operation
    - type(n) : type of functional-unit that can execute n
    - delay(n): number of cycles required to complete n
  - Edge (n1,n2) ∈ N indicates n2 uses result of n1 as operand
  - G is acyclic within each basic block

```
a: loadAI   rarp, @w  → r1
b: add      r1, r1    → r1
c: loadAI   rarp, @x  → r2
d: mult     r1, r2    → r1
e: loadAi   rarp, @y  → r2
f: mult     r1, r2    → r1
g: loadAI   rarp, @z  → r2
h: mult     r1, r2    → r1
i: storeAI r1         → rarp, 0
```



Anti Dependences

```
a: loadAI  rarp, @w ➔ r1
b: add     r1, r1   ➔ r1
c: loadAI  rarp, @x ➔ r2
d: mult    r1, r2   ➔ r1
e: loadAi  rarp, @y ➔ r2
f: mult    r1, r2   ➔ r1
g: loadAI  rarp, @z ➔ r2
h: mult    r1, r2   ➔ r1
i: storeAI r1       ➔ rarp, 0
```



- e cannot be issued before d even if e does not use result of d
  - e overwrites the value of r2 that d uses
  - There is an anti-dependence from d to e
- To handle anti-dependences, schedulers can
  - Add anti-dependences as new edges in dependence graph; or
  - Rename registers to eliminate anti-dependences
    - Each definition receives a unique name

The scheduling problem

- Given a dependence graph D = (N,E), a schedule S
  - maps each node $n \in N$ to a cycle number to issue n
- Each schedule S must satisfy three constraints
  - Well-formed: for each node $n \in N$, S(n) >= 1;
    there is at least one node $n \in N$ such that S(n) = 1
  - Correctness: if $(n1,n2) \in E$, then S(n1) + delay(n1) <= S(n2)
  - Feasibility:
    for each cycle i >= 1 and each functional-unit type t,
    number of node n where type(n)=t and S(n)=i
    ≤ number of functional-unit t on the target machine

Quality of Scheduling

- Given a well-formed schedule S that is both correct and feasible, the length of the schedule is

$$\mathbf{L(s) = \max_{n \in N}(S(n) + delay(n))}$$

- A schedule S is time-optimal if it is the shortest
  - For all other schedules Sj (which contain the same set of operations),

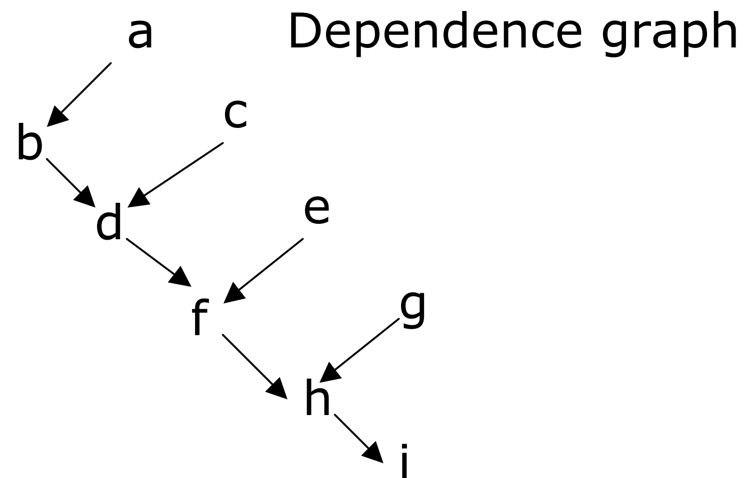
    L(S) <= L(Sj) (S has shorter length than Sj)

Instruction Scheduling

- Measures of schedule quality
  - Execution time
  - Demands for registers
    - Try to minimize the number of live values at any point
  - Number of resulting instructions from combining operations into VLIW
  - Demands for power --- efficiency in using functional units
- Difficulty of instruction scheduling
  - Balancing multiple requirements while searching for time-optimality
    - Register pressure, readiness of operands, combining multiple operations to form a single instruction
- Local instruction scheduling (scheduling on a single basic block) is NP complete for all but the most simplistic architectures
  - Compilers produce approximate solutions using greedy heuristics

Critical Path of Dependence

```
a: loadAI  rarp, @w → r1
b: add     r1, r1   → r1
c: loadAI  rarp, @x → r2
d: mult    r1, r2   → r1
e: loadAi  rarp, @y → r2
f: mult    r1, r2   → r1
g: loadAI  rarp, @z → r2
h: mult    r1, r2   → r1
i: storeAI r1       → rarp, 0
```



- Given a dependence graph D
  - Each node ni can start only if all other nodes that ni depend on have finished
  - Length of any dependence path n1n2…ni (any path in D) is
    delay(n1)+delay(n2)+…+delay(ni)
- Critical path: the longest path in the dependence graph
  - should schedule nodes on critical path as early as possible

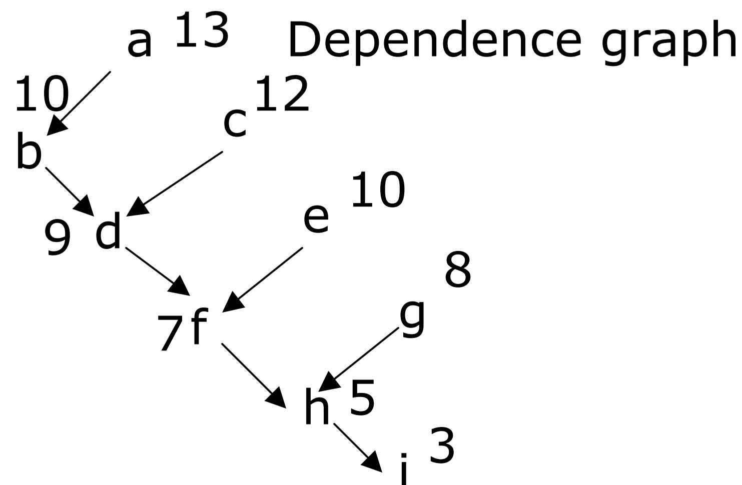
List Scheduling

- A greedy heuristic to scheduling operations in a single basic block
  - The most dominating approach since 1970s
  - Find reasonable scheduling and adapts easily to different processor architectures
- List scheduling steps
  - Build a dependence graph
  - Assign priorities to each operation n
    - Eg., the length of longest latency path from n to end
  - Iteratively select an operation and schedule it
    - Keep a ready list of operations with operands available

List scheduling algorithm

Example:

```
a: loadAI  rarp, @w → r1
b: add     r1, r1   → r2
c: loadAI  rarp, @x → r3
d: mult    r2, r3   → r4
e: loadAi  rarp, @y → r5
f: mult    r4, r5   → r6
g: loadAI  rarp, @z → r7
h: mult    r6, r7   → r8
i: storeAI r8       → rarp, 0
```



Dependence graph

```
Cycle := 1
Ready := leaves of D
Active := ∅
While (Ready ∪ Active ≠ ∅)
   if Ready ≠ ∅ then
      remove top priority i from Ready
      S(i) := Cycle
      add i to Active
   Cycle ++
   for each i ∈ Active
      if S(i) + delay(i) <= Cycle then
         remove i from Active
         for each successor j of i in D
            Mark edge (i,j) ready
            if all edges to j are ready
               then add j to Ready
```

Example: list scheduling

cycle	Ready	active	integer	memory
1	ceg	a		a
2	eg	c		c
3	g	e		e
4	g	b	b	
5	g	d	d	
6		g		g
7		f	f	
8				
9		h	h	
10				
11		i	i	

start	
1	loadAI rarp, @w ➔ r1
2	loadAI rarp, @x ➔ r2
3	loadAi rarp, @y ➔ r3
4	add r11, r1 ➔ r1
5	mult r1, r2 ➔ r1
6	loadAI rarp, @z ➔ r2
7	mult r1, r3 ➔ r1
9	mult r1, r2 ➔ r1
11	storeAI r1 ➔ rarp, 0

Complexity of List Scheduling

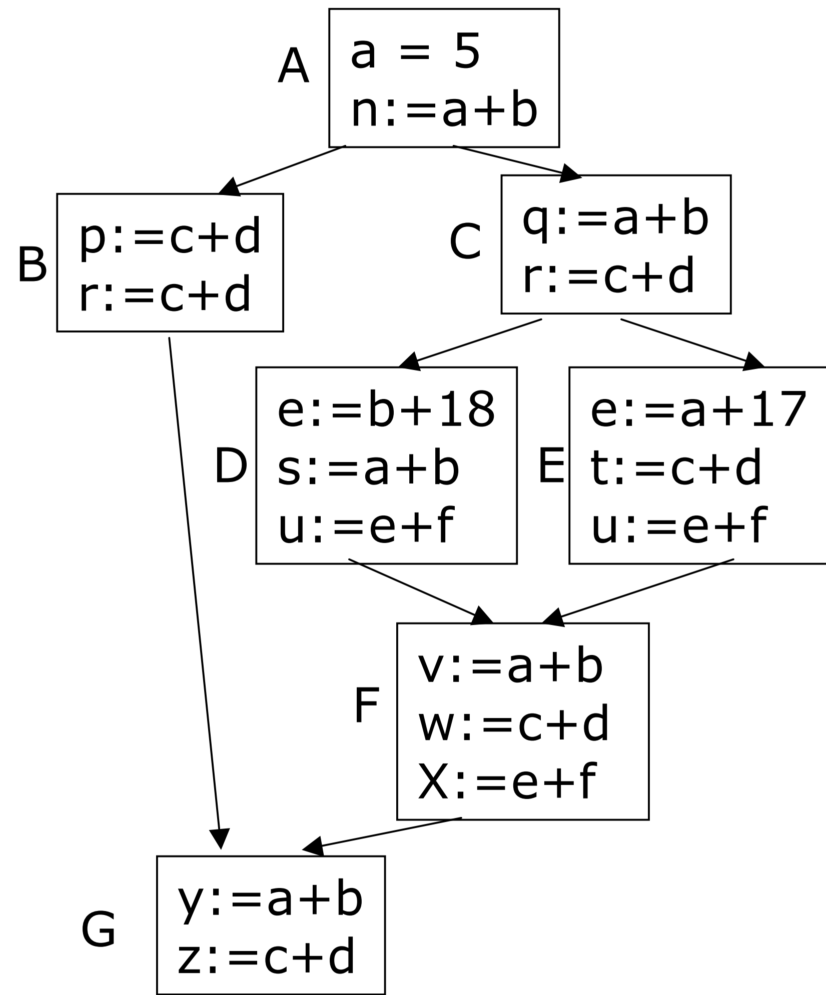
- Asymptotic complexity
  - O(NlogN + E) assuming D=(N,E)
  - Assume for each $n \in N$, delay(n) is a small constant
- When making each scheduling decision
  - Scan Ready list to find the top-priority op
    - O(logN) if using priority queue
  - Scan Active list to modify Ready list
    - Separate ops in Active list according to their complete cycles
    - Each edge must be marked as ready once: O(E)

The list-scheduling algorithm

- How good is the solution?
  - Optimal if a single op is ready at any point
  - If multiple ops are ready,
    - Results depend on assignment of priority ranking
    - Not stable in tie breaking of same-ranking operations
- Complications
  - Wait time at basic block boundaries
    - Wait for all ops in the previous basic block to complete
  - Improvement: trace scheduling (across block boundaries)
- Scheduling functional units in VLIW instructions
  - Must allocate operations on specific functional units
- Uncertainty of memory operations
  - Memory access may take different number of cycles
    - depending on whether it is in the cache

Scheduling Larger Regions

- Superlocal scheduling
  - Work on one EBB at a time
- Three EBBs: AB, ACD, ACE
- Block A appears in two EBBs
  - Moving operations to A may lengthen other EBBs
  - May need compensation code in less frequently run EBBs
    - Make other EBBs even longer
- More aggressive superlocal scheduling
  - Clone blocks to create longer EBBs
  - Apply loop unrolling



Trace Scheduling

- Start with execution counts for control-flow edges
  - Obtained by profiling with representative data
- A “trace” is a maximal length acyclic path through the CFG
  - Pick the “hot” path to optimize
  - At the cost of possibly lengthening less frequently executed paths
- Trace Scheduling Entire CFG
  - Pick & schedule hot path
  - Insert compensation code
  - Remove hot path from CFG
  - Repeat the process until CFG is empty

Summary

- Instruction scheduling
  - Reordering of instructions to enhance fine-grained parallelism within CPU
  - Dependence based approach
- List scheduling
  - Heuristic to scheduling operations in a single basic block
- Trace scheduling
  - Extending list scheduling to go beyond single basic blocks