Introduction to Instruction Scheduling

- Scheduling Constraints
- Basic Block Scheduling (list scheduling)

Instruction Scheduling

- Scheduling or reordering of instructions to improve CPU performance
  - Important impact to performance
  - Hottest issues in 90’s
  - CPU optimization is less important than memory optimization these days
  - Optimization for multi-threaded CPUs is hotter than multi-issue CPUs these days

History of Instruction Scheduling

- Early pipelined machines before RISC
  - Overlapping the execution of instructions requires microcode compaction
- Early single-issue RISC machines
  - Filling branch/load delay slot requires reordering
- Superscalar (multi-issue) RISC, VLIW, CISC
  - Elaborate code scheduling to fill multiple issue slots for parallel execution

Instruction Scheduling Techniques

Categories of instruction scheduling

- Local, basic block scheduling
- Global, cross-block scheduling
- Software pipelining

Instruction scheduling is done via code motion or code placement

Instruction scheduling is constrained by resource and dependence constraints

Scheduling Constraints

- Resource Constraints
  - Caused by mutual exclusion of functional units
  - Machine constraints of real CPUs are complex
- Program Constraints
  - Caused by precedence dependences in program
  - Control dependences
  - Data dependences
  - Techniques to overcome these constraints

Control Dependences

- An example

```
if ( a > t ) then {
        b = a*a;
}
c=a+d;
```

- Control Dependence
  - b=a*a is control dependent on a>t
    - Cannot move b=a*a ahead of the branch (a>t)
  - c=a+d is not control dependent on a>t although it is located after the branch (a>t)
    - Can move c=a+d before the branch with no problem

Overcoming Control Dependences

Speculative code motion

- Move control-dependent instruction before conditional branch so that they are executed speculatively

```
b = a*a;
if ( a > t ) then {

}
c=a+d;
```

- Effectiveness of speculative execution
  - If the original path is taken: successful
  - If the original path is not taken: nothing to lose
  - Speculative code motion should be made to use otherwise idle resources

Speculation

- Correctness of speculative code motion
  - Should not affect the correctness

```
/* if b is live after c=a+d */
b' = a*a;
if ( a > t ) then {
        b = b';
}
c=a+d;
```

  - Should not cause an exception
    - Speculative loads
  - Should not cause a permanent update
    - No speculative stores

Useful code motion

Non-speculative code motion

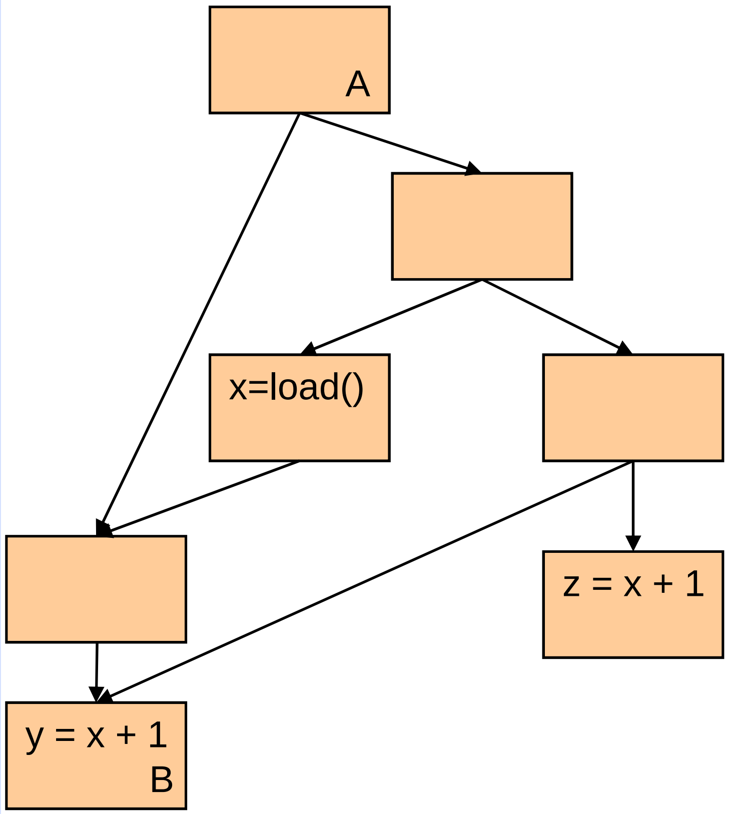
```
c=a+d;
if ( a > t ) then {
        b = a*a;
}
```

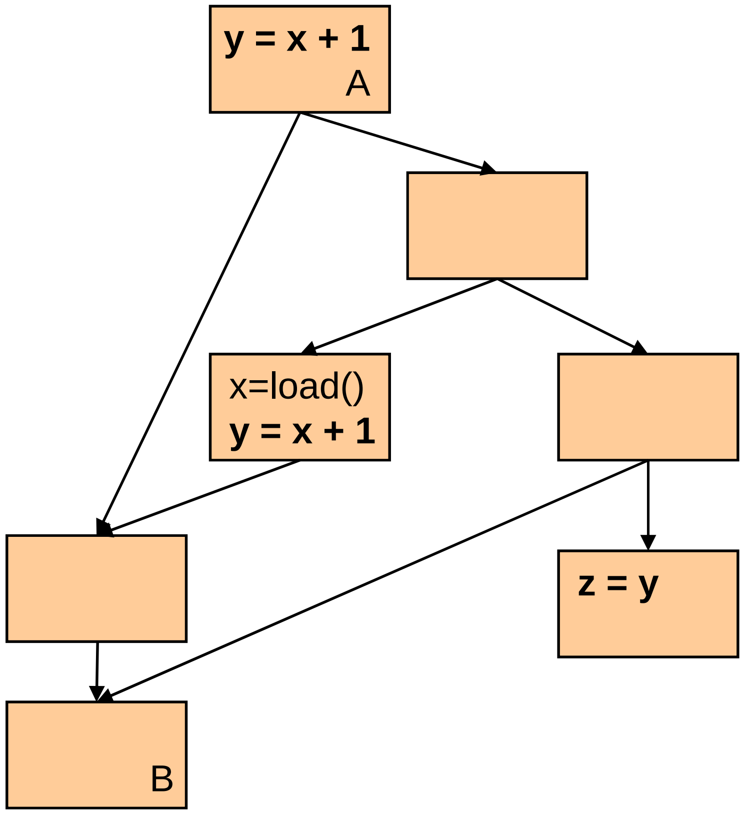
Should perform useful code motion as many as possible

- Non-speculative code motion
- Partially-speculative code motion
  - Unification
- Speculative code motion with higher hit ratio
  - Based on profiling

Unification

- Simplest form: moving an instruction below a hammock to above the hammock
- More sophisticated form of unification





Data Dependences

Must maintain the order of accesses to the potentially same locations

- True dependence: write → read
- Anti dependence: read → write
- Output dependence: write → write

Analysis on register operands is easy

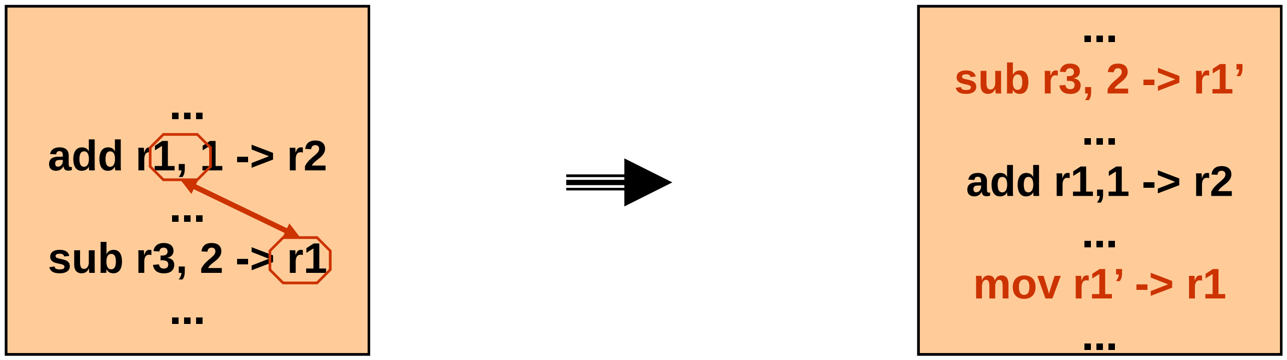
Overcoming Data Dependences

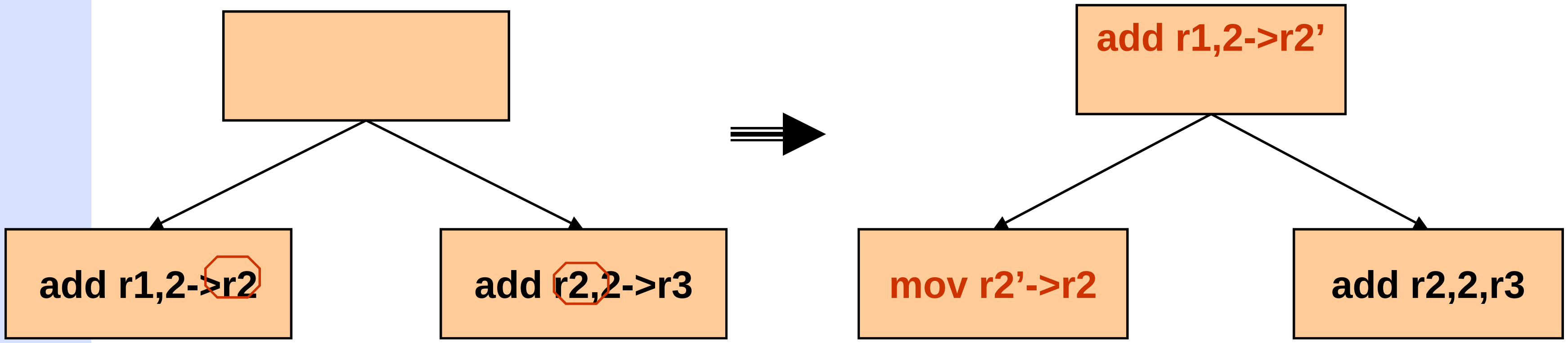
Anti and output data dependences are non-true dependences

- Caused by conserving registers
- Speculative code motion can cause violation of data dependences
- Can be overcome by (partial) renaming

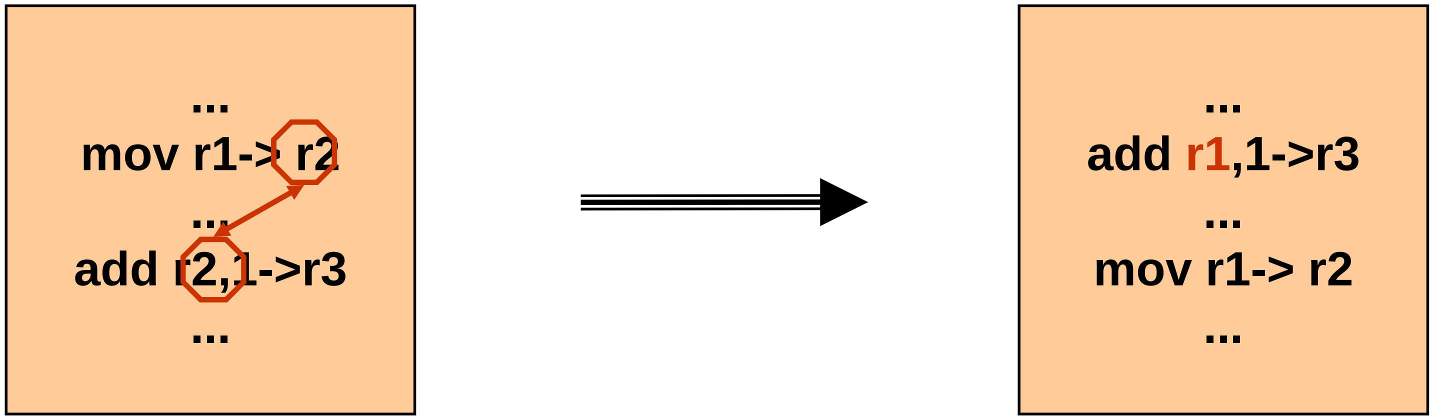
True data dependences on copies can also be overcome by forward substitution

Renaming


...
add r1, 1 -> r2
...
sub r3, 2 -> r1
...
...
sub r3, 2 -> r1’
...
add r1,1 -> r2
...
mov r1’ -> r1
...


add r1,2->r2
add r2,2->r3
add r1,2->r2’
mov r2’->r2
add r2,2,r3

Forward-substitution



Parallelism and Registers

- Register Allocation: more register < - > more parallelism

```
r1 = r2 + r3                r1 = r2 + r3
 a = r1                      a = r1
r1 = r2 + r5                r4 = r2 + r5
 b = r1                      b = r4
```

Analysis of Memory Data Dependences

Analysis on Memory Variables

- Simple: base+offset1 = base+offset2 ?
- Data dependence analysis: A[2i] = A[2i+1] ?
- Interprocedural analysis: global = parameter ?
- Pointer analysis: p1 = p2 ?

Requires memory disambiguation

- Front-ends may help
- Back-ends can also analyze by itself by following the roots of memory addresses

Memory Disambiguation

Memory disambiguation results

- Yes: store-load can be replaced by store-copy
- Partially yes
- No: no data dependence
- Maybe: speculative code motion
  - Exploit unsafe load and address compare buffer

Overview of Scheduling

- Local, basic block (BB) scheduling
  - List scheduling
  - Interaction between register allocation and scheduling
- Global Scheduling
  - Cross-Block code scheduling
- Software Pipelining

Machine Model

- Machine model used in our scheduling examples
- Parallel operations
  - One integer operation:
    - ALUop dest = src1, src2 (1 cycle)
  - One memory operation:
    - LD dest = addr (2 cycle), ST src, addr (1 cycle)

- All registers are read at the beginning of a cycle, and are written at the end of a cycle
  - LD r2=0(r1) followed by ALU r1=r3, r4 can execute in parallel
  - load uses the old value of r1
  - Anti-dependence has a delay of zero cycle

Precedence Constraints

- Data dependences among instructions in a BB form a directed acyclic graph (DAG)
  - Nodes: instructions
  - Edges: data dependence constraints, labeled by its delay cycles
  - An example BB

```
i1 : LD      r2 = 9(r1)
i2 : ST            r4,  0(r3)
i3 : ADD     r4 = r4, r3
i4 : ADD     r1 = r5, r4
i5 : ADD     r6 = r2, r4
```

Scheduling without Resource Constraints

- Topological Sort

```
Ready = nodes with zero predecessors
Loop until READY is empty
    Schedule each node in READY as early as possible
    Update predecessor count of successor nodes
    Update READY
```

- Length of the schedule = critical path length

Scheduling with Finite Resources

- Optimal scheduling is NP-complete in general
- Need a heuristic solution

- List scheduling

```
READY = nodes with zero predecessors
Loop until READY is empty
    Let n be the node in READY with highest priority
    Schedule n in the earliest slot
        that satisfies precedence + resource constraints
    Update predecessor count of n's successor nodes
    Update READY
```

List Scheduling

- Scope: DAGs
  - Schedule operations in topological order
  - Never backtracks
- Consider “best” among those in the critical path, branch-and-bound for microinstruction scheduling [Fisher]

Variations of List Scheduling

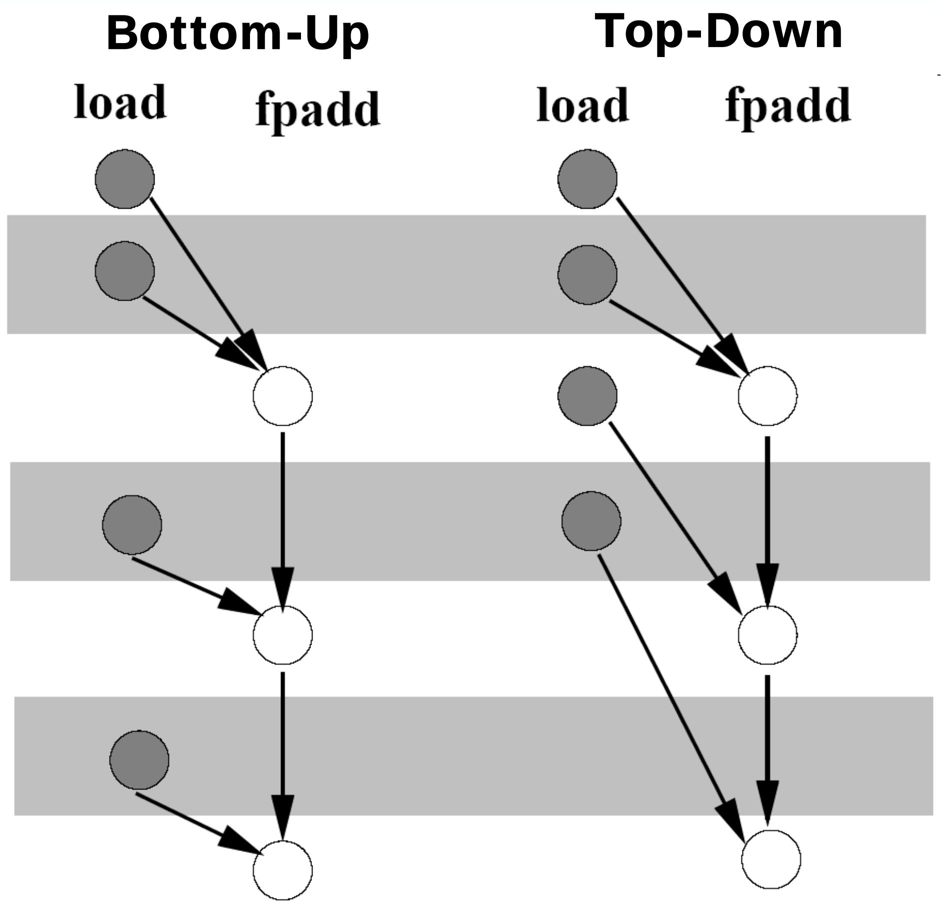
Priority function for node *n*

- delay: max delay slots from *n* to any node
- critical path: max cycles from *n* to any node
- resource requirements
- source order

Direction: top-down vs. bottom-up

Operation vs. cycle scheduling

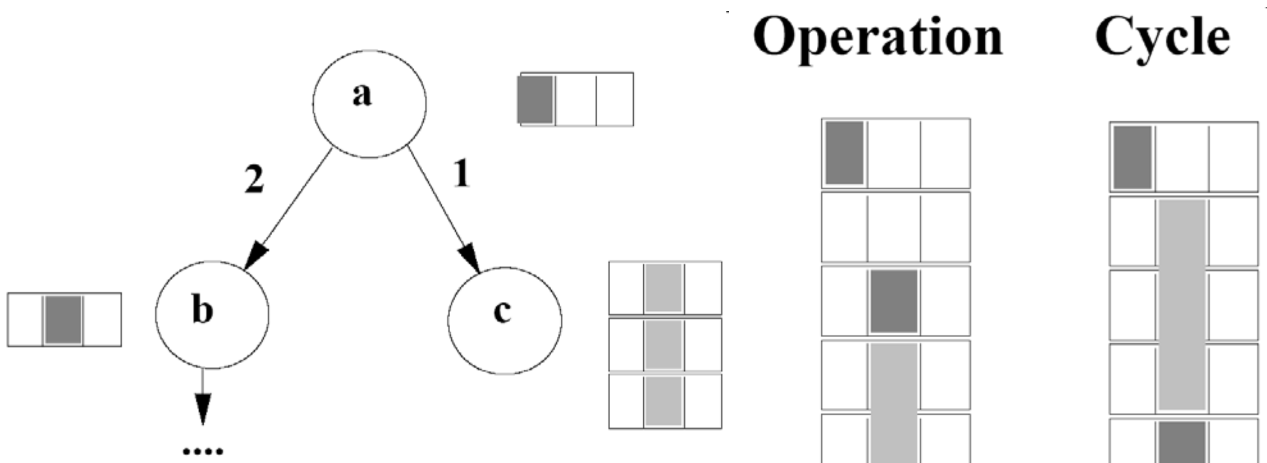
Direction: Top-down vs. Bottom-up



- Bottom-up (ALAP) shortens register lifetimes for expression trees than top-down (AEAP)

Cycle vs. Operation Scheduling

- Operation scheduling
  - Schedule critical operations first



- Cycle Scheduling
  - Concept of a current cycle
  - Only instructions ready in the current cycle are considered

Cycle vs. Operation Scheduling

- Advantages of operation scheduling
  - highest priority operation gets highest priority
- Advantages of cycle scheduling
  - Simpler to implement
    (no need to keep history of resource usage)
  - Easier to keep track of register lifetimes

Phase Ordering of RA and IS

- Allocation and assign registers → Schedule
  - round robin, partial renaming, forward substitution
- Allocate to infinite registers → Schedule → Assign registers
  - Add spill code later
- Allocate to infinite registers → Schedule → Assign registers → Schedule (spill code)
  - Current popular wisdom

Integrated Solution

- Integrated register allocation & scheduling [Goodman & Hsu]
  - List schedule (keep track of liveness of registers)
    - Priority 1: maximize parallelism
    - Priority 2: reduces registers
  - When remaining registers > threshold, use priority 1
  - Otherwise, use priority 2