Code Scheduling

Modern processors are pipelined.

They give the impression that all instructions take unit time by executing instructions in *stages* (steps), as if on an assembly line.

Certain instructions though (loads, floating point divides and square roots, delayed branches) take more than one cycle to execute.

These instructions may *stall* (halt the processor) or require a nop (null operation) to execute properly.

A *Code Scheduling* phase may be needed in a compiler to avoid stalls or eliminate nops.

Scheduling Expression DAGs

After generating code for a DAG or basic block, we may wish to schedule (reorder) instructions to reduce or eliminate stalls.

A *Postpass Scheduler* is run after code selection and register allocation.

Postpass schedulers are very general and flexible, since they can be used with code generated by any compiler with any degree of optimization

But, since they can't modify register allocations, they can't always avoid stalls.

Dependency DAGs

Obviously, not all reorderings of generated instructions are acceptable.

Computation of a register value must precede all uses of that value.

A store of a value must precede all loads that might read that value.

A *Dependency Dag* reflects essential ordering constraints among instructions:

- Nodes are Instructions to be scheduled.
- An arc from Instruction i to Instruction j indicates that i must be executed before j may be executed.

Kinds of Dependencies

We can identify several kinds of dependencies:

- True Dependence:

  An operation that uses a value has a true dependence (also called a flow dependence) upon an earlier operation that computes the value. For example:

```
mov  1, %l2
add  %l2, 1, %l2
```

- Anti Dependence:

  An operation that writes a value has a anti dependence upon an earlier operation that reads the value. For example:

```
add  %l2, 1, %l0
mov  1, %l2
```

- Output Dependence:

  An operation that writes a value has a output dependence upon an earlier operation that writes the value. For example:

```
mov  1, %l2
mov  2, %l2
```

  Collectively, true, anti and output dependencies are called data dependencies. Data dependencies constrain the order in which instructions may be executed.

Example

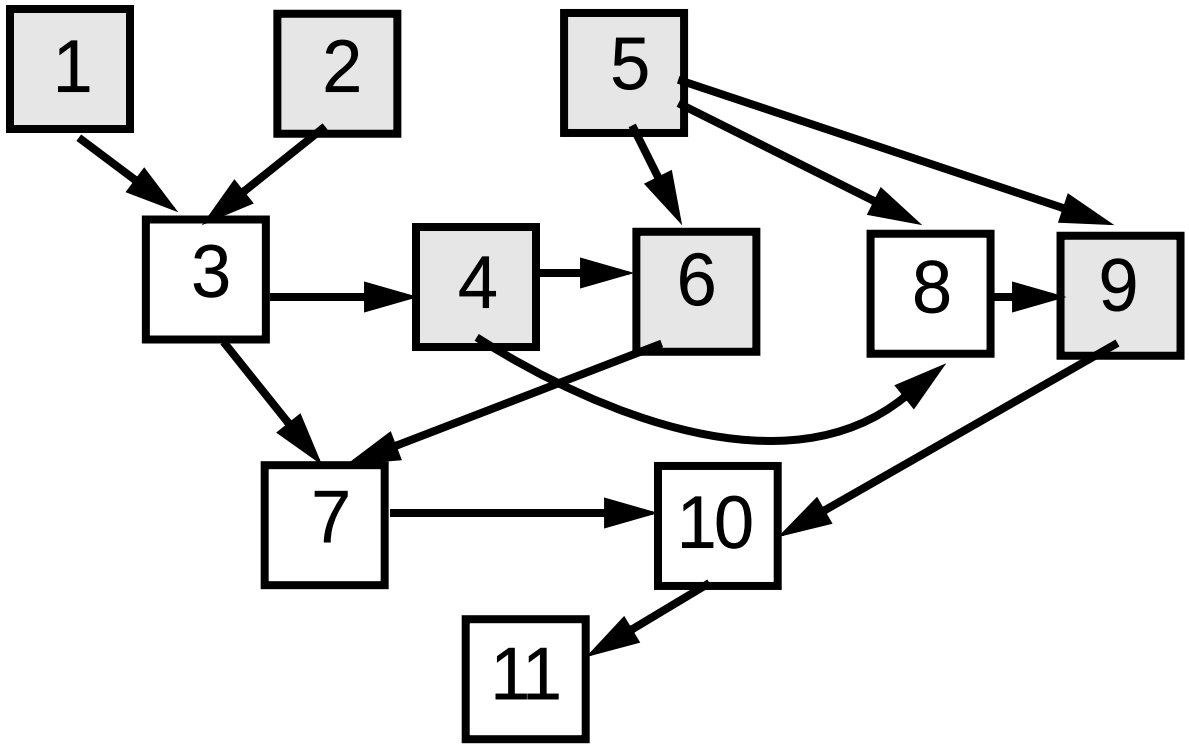
Consider the code that might be generated for

```
a = ((a+b) + (c*d)) + ((c+d) * d);
```

We'll assume 4 registers, the minimum possible, and we'll reuse already loaded values.

Assume a 1 cycle stall between a load and use of the loaded value and a 2 cycle stall between a multiplication and first use of the product.

```
1.  ld    [a], %r1
2.  ld    [b], %r2          <-- Stall
3.  add   %r1,%r2,%r1
4.  ld    [c], %r2
5.  ld    [d], %r3
6.  smul  %r2,%r3,%r4       <-- Stall
7.  add   %r1,%r4,%r1       <-- Stall*2
8.  add   %r2,%r3,%r2
9.  smul  %r2,%r3,%r2
10. add   %r1,%r2,%r1       <-- Stall*2
11. st    %r1,[a]           (6 Stalls Total)
```



Scheduling Requires Topological Traversal

Any valid code schedule is a *Topological Sort* of the dependency dag.

To create a code schedule you

(1) Pick any root of the Dag.

(2) Remove it from the Dag and schedule it.

(3) Iterate!

Choosing a *Minimum Delay* schedule is NP-Complete:

"Computers and Intractability,"
M. Garey and D. Johnson,
W.H. Freeman, 1979.