Introduction to Instruction Scheduling

15-745 Optimizing Compilers
Spring 2006

Peter Lee

Reminders

* Read: Ch.17 (instruction scheduling)
* T3 due after Spring Break
* Watch the RSS feed for announcements about reading list and project suggestions
* Contact Mike about proposal dates

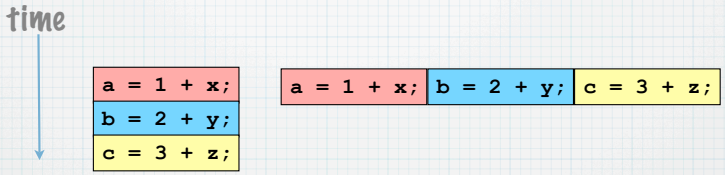
Optimizations

* IR-level optimizations
  * mostly machine independent
  * goal is to eliminate computations
* Instruction-level optimizations
  * register allocation: reduce data access cost
  * today: **instruction scheduling**

Instruction scheduling

* Assume all instructions are essential
  * i.e., we have finished optimizing the IR
* Instruction scheduling attempts to rewrite the code for maximum instruction-level parallelism (ILP)
* Instruction scheduling (IS) is NP-complete (and bad in practice), so heuristics must be used

Instruction scheduling



Since all three instructions are independent, we can execute them in parallel, assuming adequate hardware processing resources

Parallelism constraints

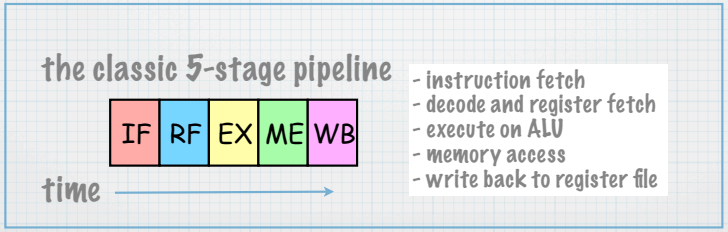
* **Data-dependence** constraints
  * If instruction A computes a value that is read by instruction B, then B can't execute before A is completed
* Resource **hazards**
  * Finiteness of hardware (e.g., how many add circuits) means limited parallelism

Hardware parallelism

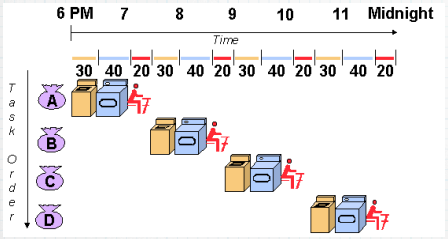
* Three forms of parallelism are found in modern hardware:
  * **pipelining**
  * **superscalar processing**
  * multiprocessing
* Of these, the first two forms are commonly exploited by instruction scheduling

Pipelining

* Basic idea is to decompose an instruction's execution into a sequence of **stages**, so that multiple instruction executions can be overlapped
  * same principle as the assembly line

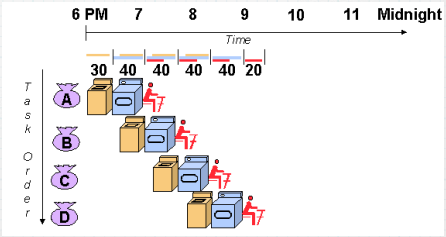


Why this might help



The classic laundry picture...

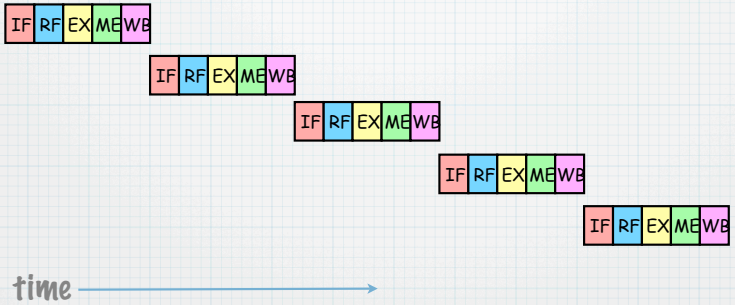
Works best if each stage takes the same amount of time (e.g., one clock cycle)



Pipelining speedup

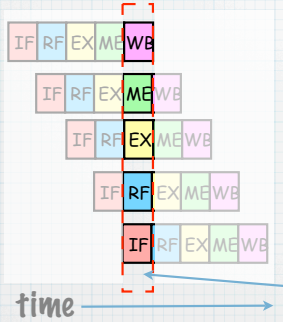
* Suppose a non-pipelined machine can execute an instruction in 5ns
  * this is a **completion rate** of 0.20inst/ns
* If each stage of the pipelined machine completes in 1ns, then it executes at a completion rate of 1.0inst/ns (after 5ns to fill the pipeline)
  * an N-fold improvement (where N is the number of stages)!

Pipelining illustration



The standard von Neumann model

Pipelining illustration



In a given cycle, each instruction is in a different stage, but every stage is active

The pipeline is "full" here

The pipeline concept was first developed for the MIPS RISC processor, by Hennessy, et al. 1981

Pipelining speedup

* In the ideal case, a pipelined processor will complete one instruction every cycle
  * this is the **instruction throughput**, which ideally is **1 IPC** (instructions per cycle)
* Alternatively, we can talk in terms of **CPI** (cycles per instruction)

Is more better?

* If 5 stages is good, 8 should be better, right?
  * and 20 even better than that!
* Not only do we get (potentially) more parallelism this way, but with smaller/simpler stages, the clock rate can be even higher
  * so we win on both parallelism and cycle time

More and more stages

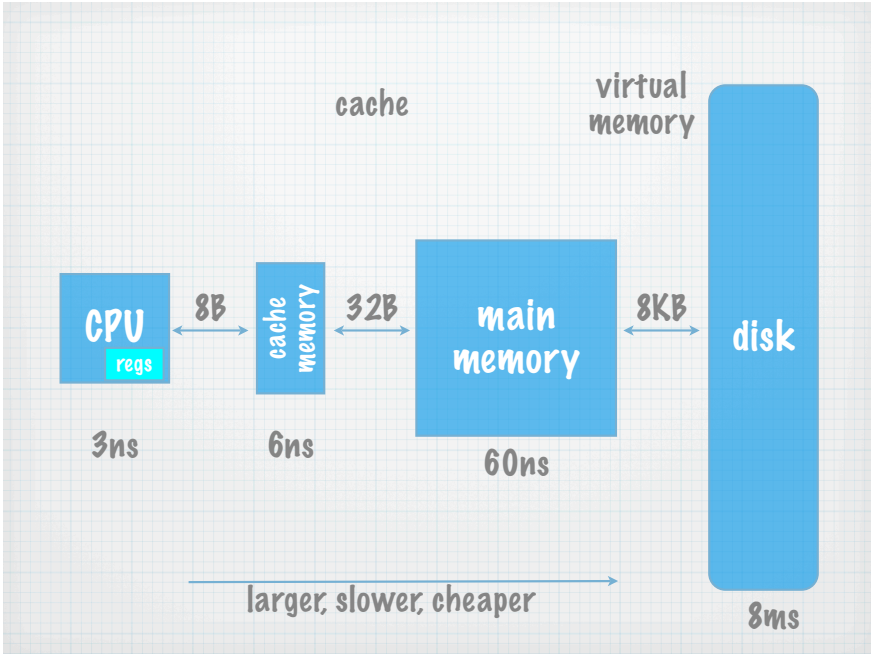
* Indeed, the MIPS R4000 used an 8-stage pipeline, and the Pentium 4 has 20 stages!
  * most consumers equate performance with clock rate, so this also sells more chips
* However, the value of a stage is unclear if it is extremely simple, e.g., less than the time of an integer add
* Amdahl's Law: a limit to available parallelism

Limitations of pipelining

* Since all stages execute in parallel, the cycle time is limited by the slowest stage
* Also, the deeper the pipeline the longer it takes to fill
* Worst of all, pipeline hazards can cause the processor to suspend, or **stall**, progress temporarily
* Stalls can have a devastating effect on performance

Pipeline hazards

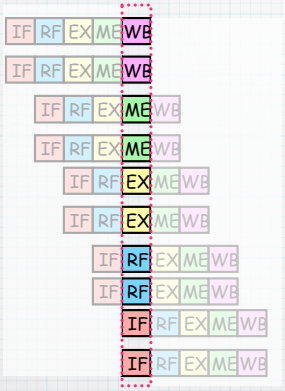
* One instruction may need the results of a previous instruction
* A branch/jump target might not be known until later in the pipeline
* An instruction may be waiting on a fetch from main memory
* Such inconsistencies in the pipeline that causes stalls are called **hazards**



Superscalar processing

* To get even more parallelism, we can go **superscalar**
* The basic idea:
  * multiple instructions proceed simultaneously through the same pipeline stages
  * this is accomplished by adding more hardware, for parallel execution of stages and for dispatching instructions to them

Superscalar illustration



Multiple instructions in the same pipeline stage at the same time

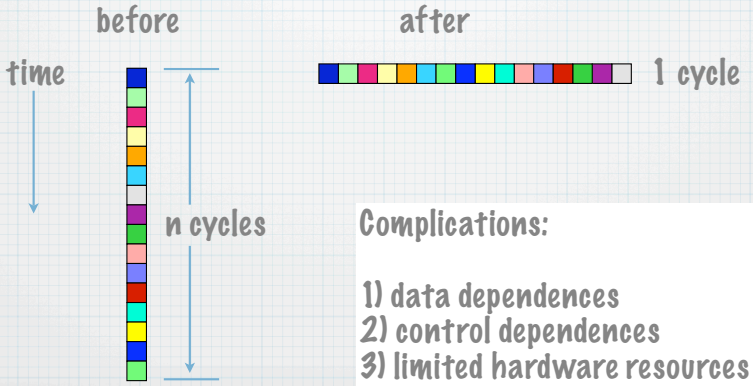
Some versions of the PowerPC, for example, have 4 ALUs and 2 FPUs

The Intel i960 (1988) was the first commercial superscalar microprocessor

Before that, 1960's-era Cray supercomputers also used the superscalar concept

Scheduling complications

We would like to reorder/rewrite instructions so as to maximize parallelism, i.e., keep all hardware resources busy



Complications:

1) data dependences
2) control dependences
3) limited hardware resources

Complication #1: Data dependences

* Must be careful not to read or write a data location "too early"

```
x = 1
y = x
```

True dependence: read-after-write

sometimes fixed via renaming

```
x = 1
x = 2
```

Output dependence: write-after-write

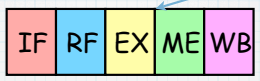
```
y = x
x = 1
```

Anti-dependence: write-after-read

Data hazards

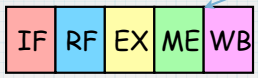
```
r1 = r2 + r3
r4 = r1 + r1
```

r2+r3 available here




```
r1 = [r2]
r4 = r1 + r1
```

[r2] available here



Data dependences

* In practice, data dependences are extremely difficult to reason about
* Considerable research effort on alias analysis, pointer analysis, with limited success

```
x = a[i];
*p = 1;
y = *q;
*r = z;
```

Register renaming

* Sometimes data hazards are not "real", in the sense that a simple renaming of registers can eliminate them
  * for output dependence, A and B write
  * for anti dependence, A reads & B writes
* The hardware can sometimes rename registers, thereby allowing reordering

Renaming example

```
r1      = r2 + 1
[fp+8]  = r1
r1      = r3 + 2
[fp+12] = r1
```

```
r7      = r2 + 1
[fp+8]  = r7
r1      = r3 + 2
[fp+12] = r1
```

```
r7      = r2 + 1
r1      = r3 + 2
[fp+8]  = r7
[fp+12] = r1
```

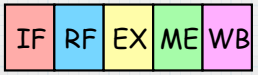
Some register allocators avoid quick re-use of registers, to get some of the benefit of renaming.

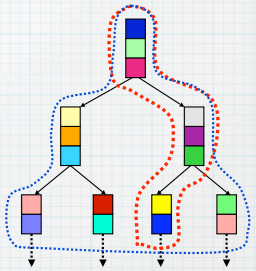
Note that there is a **phase-ordering problem**:

- scheduling before or after register allocation?

Complication #2: Control dependences

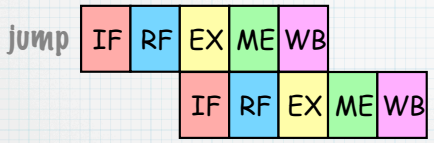
* What do we do when we reach a conditional branch?
  * unconditional target is available only after decode stage
  * conditional target is available only after execute stage



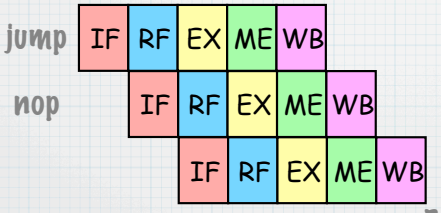


Can try to schedule for multiple paths, but this is largely impractical for real programs

Branch delay slots



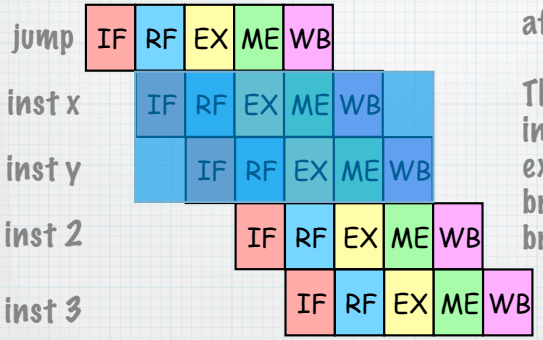
Option 1: hw solution
stall the pipeline



Option 2: sw solution
insert nop instructions

Both make the code slower

Branch delay slots



Option 3:
branch takes effect after the delay slots.

This means some instructions get executed after the branch but before the branching takes effect

Branch prediction

* Some current processors will speculatively execute at conditional branches
  * if correct guess, then great
  * if not, then flush the pipelines before the WB stage
* Average number of instructions per basic block (in C code) is 5
  * what happens with a 20-stage pipeline?

Complication #3: Hardware

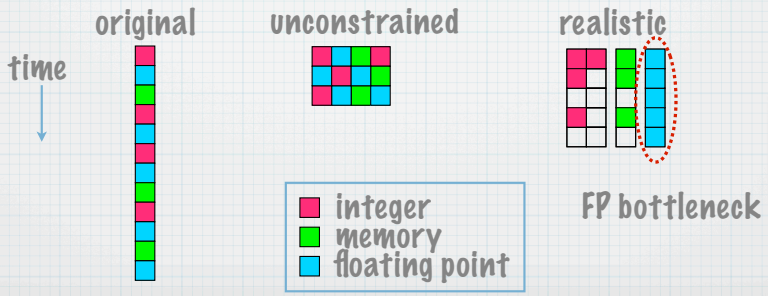
* The hardware is finite (obviously)
* Also, for engineering reasons, there may be constraints on how the hardware resources may be used

HW complication: Issue width

* Superscalar allows more than one instruction to be "issued" to the processor in each cycle
* However, there is a finite limit

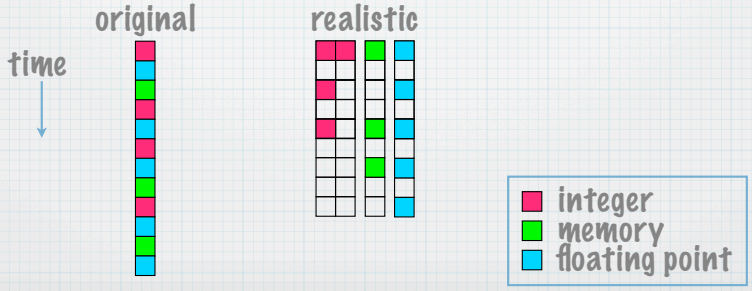
HW Complication: Functional units

* A 4-way superscalar might be limited to issuing, e.g., at most 2 integer, 1 memory, and 1 FP instruction per cycle



HW complication: Pipeline limits

* Often, there are restrictions on the pipelining in a functional unit
  * e.g., some FPUs allow a new FP division only once every 2 cycles



Compiler or hardware?

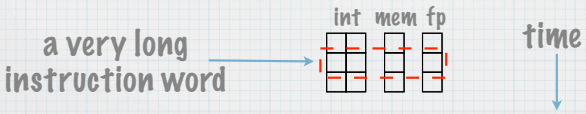
* It is possible to do some "on-the-fly" instruction re-ordering in hardware
* This raises the question of whether it is better to do scheduling in the compiler or in hardware
* There is general agreement, however, that further improvements in superscalar control will be quite limited

Compiler or hardware?

* In reality, a combination of compiler and hardware scheduling support will be used

VLIW processors

* Idea: Give full control of scheduling to the compiler
* Why: The hardware can be simpler (and thus faster) if it isn't complicated by scheduling
* How: Expose all functional units via a "very long instruction word"

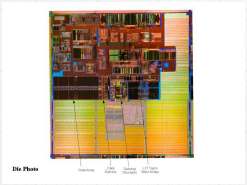


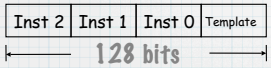
Compiling for VLIW

* Except for memory references, execution latencies for each functional unit are fixed
  * so, the hardware typically checks data dependences on memory references dynamically
* The compiler is responsible for taking data dependences into account
  * must insert NOPs for empty slots

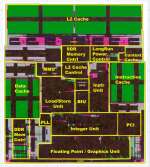
```
a = b + 1;
c = a - d;
e = c / 3;
f = g - e;
```

Intel/HP Itanium2





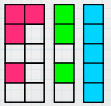
Transmeta Crusoe 5400



VLIW code generated at run-time

In-order superscalar processors

* Special dispatching hardware performs full data-dependence checking at run-time
* If no dependences are blocking the execution of the next instruction, it is dispatched to the available functional unit(s)
* Unlike VLIW, scheduling is not needed for correctness, only for performance



Out-of-order superscalar processors

* Basic idea:
  * when an instruction is stuck (due to a data dependence or lack of hardware resources), look for a later instruction that can be executed

```
x = *p;
y = x + 1;
z = a + 2;
b = c / 3;
```

this is slow, waiting on memory

and this is stuck, due to true dependence

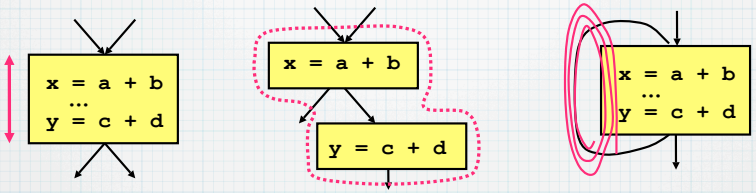
but these can execute immediately

Note: complexity of dependence checking increases exponentially with issue width

Compiler or hardware?

* For the forseeable future, high-end processors will likely be out-of-order
  * so, moving instructions small distances in the compiler is probably not worthwhile
* Low-end processors may be in-order or VLIW
  * so, instruction scheduling will be essential

Major scheduling approaches



list scheduling (within a block)

trace scheduling (across blocks)

software pipelining (across iterations)