Notes for 9/14/09

Branch delay slots:

1. add $t2, $t3, $t3
2. sub $t0, $t1, $t2
3. bne $t0, $s0, SomePlace
[delay slot]

To fill branch delay slot, assembler wants to reorder instructions like this...

1. add $t2, $t3, $t3
3. bne $t0, $s0, SomePlace
2. sub $t0, $t1, $t2

But it can't, since the branch depends on the values of line 2.

So this is what the assembler can do instead...

2. sub $t0, $t1, $t2
3. bne $t0, $s0, SomePlace
1. add $t2, $t3

...since the branch doesn't have any data dependencies on line 1.

Compiler optimizations:

1. Loop unrolling—remove overhead of branches in loops

    for (i = 0; i < 10; ++i)
    
        A[i] += 1;

=> (partially unrolled)

    for (i = 0; i < 10; i += 2) {
        A[i] += 1;
        A[i + 1] += 1;
    }

=> (fully unrolled)

    A[0] += 1;
    A[1] += 1;
    ... A[9] += 1;
2. Common subexpression elimination—remove duplicated code

```c
int a = bar * baz * bletch;
int b = 2 * baz * bletch * bar;
```

=>

```c
int a = bar * baz * bletch;
int b = 2 * a;
```

3. Strength reduction—replace expensive instructions with equivalent, less expensive operations whenever possible

```c
i *= 4; => i << 2;
```

4. Constant folding—compute what you can at compile time instead of run time

```c
int i = 4*3; => int i = 12;
```

5. Code motion—move instructions outside of loops

```c
for (i = 0; i < 100; ++i) {
    j = k + z;
    A[i] += j;
}
```

=>

```c
j = k + z;
for (i = 0; i < 100; ++i) {
    A[i] += j;
}
```

6. Dead code elimination—remove code that never runs

```c
i = 10;
if (i < 5) {
    . . .
}
```

=>

```c
i = 10;
```

RISC vs. CISC

\[
\text{time to complete} = \frac{\text{seconds}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instructions}} \times \frac{\text{instructions}}{\text{program}}
\]
CISC wins in instructions per program.
RISC wins in cycles per instruction.
RISC wins in seconds per cycle.