1 Basic Block Scheduling

In this section, we’ll review how to schedule code within a basic block. Consider the following code:

```c
// assume Y initialized
1: X = Y + 1
2: *X = 0
3: Y = ...
4: *X = 1
5: ... = *X
6: ... = *Y
```

The machine has one ALU unit, and two MEM units. All instructions take 1 clock to return their result. ... are ALU operations.

1.1 Step 1: determine the data dependencies

The first step is to determine what data dependencies exist between instructions. This in general undecidable, and we’ll have to be conservative. There are three classes of data dependencies:

- **True dependency**: read-after-write: if the original code reads a value after the value is written, the scheduled code must too (or the read will see garbage)

- **Output dependency**: write-after-write: if the original writes the same value twice, the scheduled code must also write the same value twice in the same order (or the final value that sticks will be different)

- **Anti-dependency**: write-after-read: if the original code writes to a value after it is read, the scheduled code must too (or the read will see the new value instead of the old one)

Data dependencies occur between locations, which could be registers or memory addresses.

In our example, we have the following dependencies:
• 1 → 2: RAW (to dereference the pointer, you must compute it first)
• 1 → 3: RAW
• 1 → 5: RAW
• 1 → 4: WAR (we change the value of \(Y\))
• 2 → 4: WAW (memory writes to the same pointer must be ordered)
• 3 → 6: RAW (again, compute the pointer before deref)
• 4 → 5: RAW (the memory location was just changed)
• 4 → 6: RAW: after \(Y\) was changed, \(X\) and \(Y\) can potentially alias, so conservatively we must introduce a data dependency

As for 2 → 5, it is possible to write a RAW dependency, or not, depending on how precise your analysis is. It does not matter because there is a transitive dependency anyway. Similarly, a very conservative analysis concludes that \(X\) and \(Y\) alias, but a better analysis realizes they do not, because \(X \neq X + 1\).

Side note: In real life, output dependencies after often not respected. Compilers are free to reorder, or elide entirely, memory writes to non-atomic non-volatile locations, provided that the final result does not change. Anti-dependencies are also sometimes ignored, if the compiler can reuse an earlier cached read. True dependencies on the other hand, cannot be worked around (hence the name). In the assignments and the final, though, you must not ignore dependencies, unless it explicitly says otherwise.

1.2 Step 2: write the data dependency graph

This just means writing the previous result in graph form:
1.3 Step 3: annotate data dependency graph with delays

These are just the number of clocks that it takes for a dependency to be satisfied. On pipelined machines, RAW dependencies take the full length of an instruction (because they actually need the result), and other dependencies take one clock (because they only care about ordering) On non-pipelined machines, all dependencies take the full length of the instruction.

In this example, all instructions have a length of 1, so all dependencies are annotated with 1.

1.4 Step 4: schedule

Draw the reservation table. There are infinite rows (although hopefully we won’t use all of them), and as many columns as functional units. In this case, 3: one ALU and two MEM.

This is a basic block, so there are no cycles. Start in topological order, and pick a node that has no dependencies. Schedule it as early as possible, satisfying data dependencies and resources. Remove it from the graph. Continue.

The final result is:

1: X = Y + 1
2: *X = 0 3: Y = ...
4: *X = 1
5: ... = *X 6: ... = *Y

2 Software pipelining

Basic block scheduling only deals with basic blocks, and is appropriate for control intensive code, when coupled with global scheduling. For numeric code, on the other hand, a better technique is software pipelining, which attempts to pack instructions in tight loops.

Consider the following code:

```c
for (i = 0; i < n; i++) {
    X[i] = Y[i] + 1
    X[i] *= 2
    Y[i] = Y[i] + 1
}
```

Assume the machine has ALU instructions on registers, which take 2 clocks to complete. LD and ST access memory and take 1 clock. The machine is not pipelined. The machine has 4 ALU units and 2 MEM units.

If we are to apply software pipelining to this loop, the first thing is to bring the loop to use instructions that the machine support. In this pass, we assume infinite virtual registers.
for (i = 0; i < n; i++) {
1: x0 = Y[i]
2: x1 = x0 + 1
3: X[i] = x1
4: x2 = X[i]
5: x2 *= 2
6: X[i] = x2
7: x3 = Y[i]
8: x4 = x3 + 1
9: Y[i] = x4
}

There are a lot of dependencies in this code, and basic block scheduling is unlikely to help us. But we can use software pipelining instead.

2.1 Step 1: Compute lower bound on initiation interval

From the lecture, the lower bound for all resource is: max \( \frac{n_i}{R_i} \). We have two resources, MEM and ALU. One iteration of the loop requires 6 ALU units (we have 3 ALU instructions, each take 2 clocks), and 6 MEM units. The lower bound is thus max(6/4, 6/2) = 3.
2.2 Step 2: Draw the dependency graph

As before:

If the loop is a DO-ACROSS loop (has loop carried dependencies), you must also specify the number of loop iterations next to the delay of a dependency. If the loop is a DO-ALL loop, as in this one, all dependencies have iteration number 0.

2.3 Step 3: Draw the modulo reservation table

We draw a table of height 3, and width the number of units we have (4 ALU and 2 MEM). Then we proceed as before, scheduling the basic block, but we also pay attention to have no modulo reservation conflicts. If we do have a conflict, we can try adding a NOP to skip the
clock cycle. Or we can give up, increase the initiation interval and start over. If everything fails, we can always increase the initiation interval all the way the basic block schedule with no modulo at all.

For this example, we obtain:

<table>
<thead>
<tr>
<th>ALU1</th>
<th>ALU2</th>
<th>ALU3</th>
<th>ALU4</th>
<th>MEM1</th>
<th>MEM2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

### 2.4 Step 4: Codegen

Code generation then just proceeds by generating the basic block in order. After a number of clocks equal to the initiation interval, you start the second iteration in parallel. After twice the initiation interval, you start the third iteration in parallel, and so on, until you move past the end of the first iteration. You put a BL (branch loop) there and you’re done.

Note that unlike basic block scheduling, here you do not put two instructions for the same iteration in parallel.

The final code for a single iteration is:

```plaintext
1: x0 = Y[i]
2: x1 = x0 + 1
   nop
3: X[i] = x1
4: x2 = X[i]
5: x2 *= 2
   nop
6: X[i] = x2
7: x3 = Y[i]
8: x4 = x3 + 1
   nop
9: Y[i] = x4
```
The final code (before register allocation and modulo variable expansion) is thus:

```
x0 = Y[i]
x1 = x0 + 1
nop
X[i] = x1
x0 = Y[i]
x2 = X[i]
x1 = x0 + 1
x2 *= 2
nop
X[i] = x1
x0 = Y[i]
X[i] = x2
x2 = X[i]
x1 = x0 + 1
x3 = Y[i]
x2 *= 2
nop
L: x4 = x3 + 1
nop
X[i] = x1
x0 = Y[i]
x4 = x3 + 1
nop
X[i] = x1
x0 = Y[i]
Y[i] = x4
x3 = Y[i]
x2 *= 2
nop
Y[i] = x4
x4 = x3 + 1
nop
Y[i] = x4
x4 = x3 + 1
nop
Y[i] = x4
```