Basic Pipeline Scheduling and Loop Unrolling

To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline.

For these examples we will assume:
- The function units are fully pipelined, i.e. an instruction can be issued every cycle,
- there are no structural hazards,
- 1 branch delay slot, and
- the latencies are:

<table>
<thead>
<tr>
<th>Instruction Producing Result</th>
<th>Instruction Using Result</th>
<th>Latency in cycles (# of nops between)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store Double</td>
<td>2</td>
</tr>
<tr>
<td>Load Double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load Double</td>
<td>Store Double</td>
<td>0</td>
</tr>
</tbody>
</table>

Suppose we want to execute:

\[
\text{for}(i=1; \ i<100; \ i++) \\
\text{x}[i] = \text{x}[i] + s;
\]

Each iteration is independent.

In assembly:

```
loop:
  ld  $f0, 0($t1)
  add  $f4, $f0, $f2
  sd  $f4, 0($t1)
  subi $t1, $t1, 8
  bneq $t1, loop
```

Unscheduled on MIPS

```
loop:
  ld  $f0, 0($t1)

  add  $f4, $f0, $f2

  sd  $f4, 0($t1)

  subi $t1, $t1, 8

  bneq $t1, loop
```

Clock Cycle Issued

```
loop:
  ld  $f0, 0($t1)  1

  add  $f4, $f0, $f2

  sd  $f4, 0($t1)

  subi $t1, $t1, 8

  bneq $t1, loop
```

________ cycles required.
Scheduled

Clock Cycle Issued

loop:

```
ld $f0, 0($t1)      1

add $f4, $f0, $f2

subi $t1, $t1, 8

bneq $t1, loop

sd $f4, ______($t1)
```

______ cycles required.

Much faster!

This is a non-trivial change. Most compilers would see that the sd depends on the subi and wouldn’t exchange them. A smarter compiler/assembler would do it. One load-add-store loop requires 6 clock cycles, but load-add-store only requires 3 cycles. The other 3 cycles are subi, bneq, and stall.

Use loop unrolling to do more per loop.

Unroll the loop 4 times:

```
loop:
```

```
ld $f0, 0($t1)
add $f4, $f0, $f2
sd $f4, 0($t1)

```

```
ld ______, ______ ($t1)
addd ______, $f6, $f2
sd ______, ______ ($t1)
```

```
ld $f10, ______ ($t1)
add $f12, $f10, $f2
sd $f12, ______ ($t1)
```

```
ld $f14, ______ ($t1)
add $f16, $f14, $f2
sd $f16, ______ ($t1)
```

```
subi $t1, $t1, ______
bneq $t1, loop
```
Note: Saved 2*3=6 cycles
Used different registers so we can schedule better.

1 loop takes 27 cycles or 6.8 clock cycles per element.

This unrolled version is slower than the standard.

In real programs we don’t always know the number of loops. Suppose we have \( n \) loops, we can unroll to have \( k \) copies. Then we would have 2 copies of code. The first would execute \( n \mod k \) times, then second would execute \( n/k \) times.

**Scheduled – unrolled loop**

```
loop:
    ld $f0, 0($t1)
    ld $f6, -8($t1)
    ld $f10, -16($t1)
    ld $f14, -24($t1)
    
    add $f4, $f0, $f2
    add $f8, $f6, $f2
    add $f12, $f10, $f2
    add $f16, $f14, $f2
    
    sd $f4, 0($t1)
    sd $f8, -8($t1)
    sd $f12, -16($t1)
    
    sub $t1, $t1, ______
    bneq $t1, loop
    
    sd $f16, ______ ($t1)
```

Unrolling exposes more computation that can be scheduled to minimize stalls.