Section 3.1 – Principles of Pipeline Design

**Pipelining:** Parallelism is achieved and performance is improved by starting to execute one instruction before the previous one is finished.

- The simplest kind of pipeline overlaps the *execution* of one instruction with the *fetch* of the next instruction.
- Because two instructions can be processed simultaneously, we say that the pipeline has two *stages*.

![Diagram of pipeline stages](image)

Another, idealized way of looking at it:

![Diagram of instruction execution](image)

However, things aren’t quite that neat. For example:

- On a cache miss, instruction fetch may have to wait for many cycles.
- Some instructions take more than one cycle to execute (e.g. load and store instructions).
- A branch instruction changes where the next instruction is fetched.

So we don’t really get one instruction every cycle with a 2-stage pipeline.
A pipeline may have more than two stages. Stone describes an instruction pipeline with 7 stages (page 145):

This sequence shows that the complete instruction execution requires 70 ns. If each instruction takes this long and we do not start the next instruction before completing the current one, we have a processing rate of 1 instruction per 0.07 microseconds = 14.3 MIPS.
Synchronization and Pipelines

We can improve on this processing rate by beginning subsequent instructions prior to completion of current ones. But to do this, we need to synchronize the passing of results from one stage to the next.

We use a fixed clock rate and synchronize all processing with this fixed clock. Each stage:

♦ Performs its function within the clock period.
♦ Shifts (intermediate) results to the next stage on the clock edge.

Clock Rate

The clock rate can be chosen such that the longest stage finishes in one clock period. An alternative would be to choose a faster clock rate and insert wait states (integral number of delay periods) to allow the longer stages to finish their processing.

With the 7 stage example, we could choose a 20 ns clock period (longest stage is 20 ns), which is a 50 MHz clock. If we can sustain an instruction stream through this pipeline at this rate (a new instruction into stage 1 and a completed instruction out of stage 7 every 20 ns), we get a processing rate of 1 completed instruction every 0.02 microseconds = 50 MIPS.

An alternative clock period of 5 ns (implied by Stone in figure 3.5) would yield the same processing rate, because we could start a new instruction only every 4 clock periods.

Following is the pipeline with a 200 MHz clock:
Synchronous Pipeline

With this example, the 20 ns stages use four clock periods and the 5 ns stages complete their work in one clock period, but must wait 3 additional periods before shifting results to the next stage. Stone’s figure 3.5 shows these delay periods being inserted.
Instruction pipeline organization for DEC Alpha 21164

**Figure 2 Instruction Pipeline Stages**

- **Integer Operate Pipeline**
  - IC 0
  - IB 1
  - SL 2
  - AC 3
  - 4
  - 5
  - 6
  - First Integer Operate Stage
  - If Needed, Second Integer Operate Stage
  - Write Integer Register File
  - Instruction Cache Read
  - Instruction Buffer, Branch Decode, Determine Next PC
  - Slot by Function Unit
  - Register File Access Checks, Integer Register File Access

- **Floating-Point Pipeline**
  - IC 0
  - IB 1
  - SL 2
  - AC 3
  - 4
  - 5
  - 6
  - 7
  - 8
  - Floating-Point Register File Access
  - First Floating-Point Operate Stage
  - Write Floating-Point Register File, Last Floating-Point Operate Stage

- **Memory Reference Pipeline**
  - IC 0
  - IB 1
  - SL 2
  - AC 3
  - 4
  - 5
  - 6
  - 7
  - 8
  - 9
  - 10
  - 11
  - 12
  - Dcache Read Begins
  - Dcache Read Ends
  - Use Dcache Data, Store Writes
  - Dcache, Scache, Tag Access
  - Scache Data Access Begins
  - Scache Data Access Ends
  - Fill Dcache
  - Use Scache Data

Arithmetic, logical, shift, and compare instructions complete in pipeline stage 4 (1-cycle latency). CMOV completes in stage 5 (2-cycle latency). IMULL has an 8-cycle or 9-cycle latency. CMOV or BR can issue in parallel (0-cycle latency) with a dependent CMP instruction.
An Arithmetic Pipeline

\[ A = a \times 2^p \]
\[ B = b \times 2^q \]

\[ r = \max(p, q) \]

\[ t = |p - q| \]

\[ c \]

\[ L \]

\[ C = d \times 2^s = A + B \]
Latency

Even though we can complete one instruction every 20 $ns$ in the example, each instruction takes 80 $ns$ to get thorough the pipeline from initiation to completion. This is called the latency of the pipeline.

Shorter clock periods reduce latency. For example, if we set the clock period to 20 $ns$ instead of 5 $ns$ for the example pipeline, the pipeline latency would be 140 $ns$ instead of 80 $ns$.

Speedup

The speedup of a pipeline measures how much more quickly a workload is completed by the pipeline processor than by a non-pipeline processor. Stone defines speedup as:

$$\text{Speedup} = \frac{\text{Best Serial Time}}{\text{Parallel Execution Time}}$$

In the example, Stone’s best serial time is the asynchronous time of 70 $ns$ from the first diagram. The parallel execution time (per instruction) is 20 $ns$, so the speedup for this example is $70/20 = 3.5$.

In reality, we would never use the asynchronous time, and so I would prefer to use the clocked serial time to calculate speedup. The example pipeline would then give a speedup of $80/20 = 4.0$.

Once again, this assumes that we can keep the pipeline full and complete an instruction every 20 $ns$ (which is almost never the case), which leads to the concept of efficiency.
**Efficiency**

Stone also deals with the *efficiency* of parallel computers (§3.3) – how well all of the processing units are utilized.

He also states *Amdahl’s Law*, which says that as we add more parallel components, we can effectively utilize them less and less.

Three things that prevent 100% utilization of processing units:

- We must spend some serial time to set up the parallel units. The parallel units are idle during this setup time.

- For pipelined processors, it takes some time to fill the pipe at the start of processing and to empty the pipe at the end of the processing. During this startup and finish up time, not all stages are working.

- The pipeline has hazards that prevent a steady flow of data through the pipe. For example, the instruction pipeline may have branches and data dependencies that prevent an instruction from proceeding until an earlier instruction completes processing and exits the pipe.

Amdahl’s law was really formulated for *vector computers* and *array computers* (discussed later) but we can also see that the concept also applies to any pipeline. We will deal with the first two items when we discuss vector and array computers. We deal with hazards and the ability to keep the pipeline full here.

We pick up the instruction pipeline example again in order to discuss these concepts.
Collisions in the pipeline

We can’t really keep the example instruction pipeline full if we have a single cache for accessing memory. For example, we would be trying to:

- store results of instruction 1 while we are
- fetching an operand for instruction 3 and
- fetching instruction 6 all at the same time.

These are called collisions in the pipeline.

The reservation table

A reservation table is a very useful tool for analyzing pipelines.

A Reservation Table for Stone’s Instruction Pipeline

<table>
<thead>
<tr>
<th>Clock</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst Fetch</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst Dec.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addr Gen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Op Fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Op Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update PC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This would indicate that there are no collisions in the pipeline, and that we could initiate a new instruction every 4 clock periods.

However, this isn’t really possible.
We can see this by showing subsequent instruction initiations in the reservation table. The shaded cells show collisions for use of the cache if we initiate an instruction every 4 cycles:

<table>
<thead>
<tr>
<th>Clock → Operation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst Fetch</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Inst Dec.</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addr Gen</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Op Fetch</td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Op Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update PC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

If we change the table to reflect that the memory operations are really using the same functional unit (cache), we get the following table:

<table>
<thead>
<tr>
<th>Clock → Operation</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem Op</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Inst Dec.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addr Gen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update PC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Given this view of the pipeline, how often can we initiate an instruction?
Suppose we implemented a harvard cache so we could fetch an instruction concurrent with loading or storing a data item? The new reservation table would look like:

<table>
<thead>
<tr>
<th>Clock</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst Fetch</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst Dec.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addr Gen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Op</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update PC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Now how often can we initiate an instruction?

One way to determine how frequently we can initiate an instruction and ensure no collisions is to overlay one copy of the reservation table with another, and slide one to the right until none of the X’s overlap.

<table>
<thead>
<tr>
<th>Clock</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst Fetch</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst Dec.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addr Gen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Update PC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

This shows that we can initiate an instruction only every 9 clocks. With a cycle time of 5 $\text{ns}$, we get a speedup of $\frac{80}{45} = 1.8$ (instead of 4 in the ideal case).