Pipelining

• One way to reduce the CPI term of the time equation is to perform more than one operation at a time.

• In general this is known as instruction-level parallelism.

• A particular form of parallelism is pipelining
Example: Assembly line manufacturing automobiles

- Many different cares are being built at the same time.

- Each car is at a different stage of completion

- For each car there is a particular order of steps needed to produce a working car – axles must be contructed and put on before wheels.
• In a processor this type of idea can be applied at different levels

• pipelining within the execution portion of the instruction – the ALU or floating point unit can be pipelined

• pipelining of the entire process from fetching to completion of an instruction
Floating point addition

Suppose \( x = 2^p \times t_1 \) and \( y = 2^q \times t_2 \) then, \( x + y \) implies

1. compare exponents (c-stage)
2. shift mantissa (s-stage)
3. add mantissas (a-stage)
4. normalize (n-stage)
4-digit floating point decimal arithmetic

- **Example 1**
  - \(x = 10^4(0.6314)\) and \(y = 10^1(0.3865)\)
  - \(fl(x + y) = 10^4(0.6314) + 10^4(0.0003865) = 10^4(0.6318)\)
  - (no normalization needed)

- **Example 2**
  - \(x = 10^4(0.6314)\) and \(y = 10^4(0.6065)\)
  - \(fl(x - y) = 10^4(0.6314) + 10^4(0.6065) = 10^4(0.0249)\)
  - (no shift needed) \(fl(x - y) = 10^4(0.0249) = 10^3(0.249)\)
input $x_i, y_i \rightarrow \boxed{C} \rightarrow \boxed{S} \rightarrow \boxed{A} \rightarrow \boxed{N} \rightarrow \text{output}$

$$z_i = x_i + y_i \quad i = 1, \ldots, n$$

Cycle by cycle profile of pipelined add. ($O_i$ indicates subcomputations for the $i$-th component operation.)

<table>
<thead>
<tr>
<th>cycle</th>
<th>in</th>
<th>C</th>
<th>S</th>
<th>A</th>
<th>N</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x_1$, $y_1$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>$x_2$, $y_2$</td>
<td>$O_1$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>$x_3$, $y_3$</td>
<td>$O_2$</td>
<td>$O_1$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>$x_4$, $y_4$</td>
<td>$O_3$</td>
<td>$O_2$</td>
<td>$O_1$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>$x_5$, $y_5$</td>
<td>$O_4$</td>
<td>$O_3$</td>
<td>$O_2$</td>
<td>$O_1$</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>$x_6$, $y_6$</td>
<td>$O_5$</td>
<td>$O_4$</td>
<td>$O_3$</td>
<td>$O_2$</td>
<td>$z_1$</td>
</tr>
<tr>
<td>7</td>
<td>$x_7$, $y_7$</td>
<td>$O_6$</td>
<td>$O_5$</td>
<td>$O_4$</td>
<td>$O_3$</td>
<td>$z_2$</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
</tbody>
</table>
Note that we have assumed that we have several computations of the same type to perform on essentially independent data. ($n$ is usually referred to as the vector length.)

This is known as vector processing and was one of the earliest techniques used to develop so-called supercomputers.

It typically imposes restrictions on where the $x_i$, $y_i$ and $z_i$ are located in memory.
Simple Vector Timing Model

A simple model can be derived using 3 basic architectural parameters and a vector length $n$. Let $\lambda =$ stages in pipe (one clock cycle per stage), $\tau =$ clock cycle in seconds, and $\sigma =$ the pipeline set-up time and operand memory latency in cycles. In a very loose sense $\lambda$ and $\sigma$ are related to the architectural design and $\tau$ is technology dependent.

It follows that for a vector of length $n$:

- $t_s = \tau \lambda n$ is the sequential execution time (it is actually less than that due to latches in pipe);

- $t_v = \tau (\sigma + \lambda) + \tau (n - 1) = t_0 + \tau n$ is the vector execution where time $t_0$ is called the startup time;

- $r^s_{\infty} = 1/\tau \lambda$ is the rate of sequential execution in operations per second;

- $r^v_{\infty} = \lim_{n \to \infty} n/t_v = 1/\tau$ is the asymptotic vector execution rate.
Define \( n_{1/2} \equiv t_0/\tau = (\sigma + \lambda - 1) \). Clearly, \( n_{1/2} \) is the number of elements that could be processed at the asymptotic vector rate during the startup time. Time can be conveniently written as the linear function of \( n \) (Hockney)

\[
t_v = t_0 + \tau n = \tau n_{1/2} + \tau n = \tau (n_{1/2} + n)
\]

\[
t_v = \frac{n + n_{1/2}}{r_v^\infty}
\]

- \( \tau = 1/r_v^\infty \) is the slope of the time function
- \( t_0 \) is the y–intercept
- \( -n_{1/2} \) is the x–intercept
- \( n_{1/2}, r_v^\infty, \) and \( r_s^\infty \) can be measured easily empirically, so you do not have to have \( \lambda \) and \( \sigma \).
• Vector break-even length, $n_b$, is the vector length for which $t_s = t_v$. We have

$$n_b/r^s_\infty = (n_b + n_{1/2})/r^v_\infty \Rightarrow n_b = n_{1/2}/(R_\infty - 1)$$

where $R_\infty = r^v_\infty/r^s_\infty > 1$.

• Performance (operations per second) for vector length $n$:

$$r^v_n = \frac{n}{t_v} = \frac{n r^v_\infty}{(n + n_{1/2})} = \frac{r^v_\infty}{1 + \rho}$$

where $\rho = n_{1/2}/n$. 

The ratio of $n$ to $n_{1/2}$ can then be related to the fraction of the asymptotic vector performance achieved.

where $R = r_n^v/r_\infty^v$ and $\mu = \rho^{-1} = n/n_{1/2}$. Note $R = .5$ for $n = n_{1/2}$ and $R = .9$ for $n = 9n_{1/2}$. 
This model also applies to other forms of pipelining.

Note that Amdahl’s law can be applied to the vector processing case.

The fast mode corresponds to pipelined or vector processing and the slow mode corresponds to sequential or nonpipelined processing.

The same conclusions follows:

If there is substantial speedup possible due to large computational rate improvements it takes a large fraction of “fast” computations to achieve a significant fraction of the that speedup.
The entire process from fetching to completion of an instruction can also be pipelined (in addition to perhaps pipelining the functional units).

Recall the MIPS stages from Chapter 5:

- Fetch Instruction (IF)
- Decode instruction and read register file (ID)
- Execute operation or compute address (EX)
- Access memory (MEM)
- Write register file (WB)
Question:

Suppose you have a sequence of instructions, can you have different instructions performing each of the stages above simultaneously?

Answer:

Maybe.
There are three classes of barriers to the successful application of pipelining.

- **Structural hazards**

- **Control hazards**

- **Data hazards**

Mitigating the degradation due to these effects is what takes the simple idea of pipelining and makes it a very complicated and subtle implementation problem for both HW and SW.
Structural Hazards

Enough HW must be available so that the stages of the pipe do not require the same resources at the same time, e.g., two stages can not both require the use of the same ALU at the same time.

Consider the HW for the 5 stages:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Action</th>
<th>HW</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Fetch Instruction</td>
<td>Memory</td>
</tr>
<tr>
<td>ID</td>
<td>Decode and read registers</td>
<td>register file</td>
</tr>
<tr>
<td>EX</td>
<td>Execute or compute address</td>
<td>ALU</td>
</tr>
<tr>
<td>MEM</td>
<td>Access memory</td>
<td>Memory</td>
</tr>
<tr>
<td>WB</td>
<td>Write registers</td>
<td>register file</td>
</tr>
</tbody>
</table>
• Memory
  – Stages 1 and 4 both access memory so we have a structural hazard.
  – We use to memories - instruction and data – to remove hazard.
  – This is possible because stages 1 and 4 access completely independent portions of memory.

• Registers
  – Stages 2 and 5 both access the register file.
  – We cannot use two different register files since both stages can require access to any of the registers.
  – Timing of register file must be changed.
  – New register file is written in first half of the cycle and read in the second half.
  – Step 5 uses the register file in the first half and Step 2 uses it in the second half – access not simultaneous therfore no structural hazard.
Control Hazards

- The ability to fetch an instruction every cycle implies that the address of the next instruction is known.

- When processing instructions sequentially in terms of addresses this is simply $PC + 4$ and fetching of the next instruction can be overlapped with completing the previous instruction.

- Branches and jumps cause problems for this assumed mode of operation.
• Suppose the pipeline is set up so that the branch target address and condition is evaluated by the end of Stage 2.

• Note the text varies from one section and chapter to another the assumption about when the branch address and condition have been evaluated so read carefully.

• There must be some cost in performance to the delay in knowing where the next instruction is located.

• There is a one cycle stall, i.e., one extra cycle is added to the execution time relative to the time it would have taken if the branch address had been known immediately.
Program execution order (in instructions)

- add $4, $5, $6
- beq $1, $2, 40
- lw $3, 300($0)

Time: 2 4 6 8 10 12 14 16

Instruction fetch
Reg
ALU
Data access
Reg
Instruction fetch
Reg
ALU
Data access
Reg
Instruction fetch
Reg
ALU
Data access
Reg

2ns 4ns 2ns
• As the pipelines get deeper the point at which the branch condition and target address are known often is delayed much later than the end of Stage 2.

• The stalls increase accordingly.

• We must design HW to deal with delays in target address availability, e.g. generate stalls when needed.

• Mitigation is possible.
  – Branch prediction (very complicated area)
  – Branch delay slots (HW and SW mixture)
Branch Prediction

- guess what the next instruction is and proceed with it.

- Easiest in HW discussed so far – predict not taken, so next instruction is expected at $PC + 4$.

- No penalty if correct prediction.

- Stall added if wrong.

- **MUST PREVENT CONTAMINATION OF THE STATE BY ERRONEOUSLY STARTED INSTRUCTION**

- This can be done, e.g., by supressing register or memory writes.
Program execution order (in instructions)

add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)

Program execution order (in instructions)

add $4, $5, $6
beq $1, $2, 40
or $7, $8, $9
Branch Delay Slots

• If the target and condition are available at the end of Stage 2 there is a 1 cycle stall, i.e., there is a slot for an instruction to be fetched and started into the pipe.

• Rather than worry about prediction, the compiler can be required to find an instruction, that is always executed, i.e., whose execution
  – does not affect branch condition
  – does not depend on branch condition
• The instruction will be in $PC + 4$ and therefore the pipeline can proceed without worrying about incorrect predictions or suppressing state updates.

• typically the instruction comes from the set before the branch in the original code.
beq $1, $2, 10
add $4, $5, $6  (Delayed branch slot)
lw $3, 300($0)
Data Hazards

- Data hazards are caused by data dependences between instructions.

- There are many types of data dependences, we are concerned here with true dependences or flow dependences.

- Instruction 1 produces data that Instruction 2 consumes.
  - add $1,$2,$3
  - mul $4,$1,$5

- Forwarding or bypassing hardware can solve some but not all such dependence problems.
INSERT figures 6.8 and 6.9 here
Pipelined Datapath

• In order to have multiple instructions running simultaneously, the data required by each must be available to the appropriate combinational logic segment.

• We start with the single cycle data path of Chapter 5.

• An initial attempt is made to pipeline the data flow.

• An error is identified by considering a LW instruction and fixed.
• Note that the branch target and condition are not used to update the PC until the MEM stage in this data path.

• This entails a three cycle stall.

• Or three branch delay slots.

• The data path does not satisfy the earlier assumption.