Pipeline Processor

Hung-Wei Tseng
Summary: Performance Equation

Execution Time = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}

- ET = IC \times CPI \times \text{Cycle Time}
- IC (Instruction Count)
  - ISA, Compiler, algorithm, programming language, programmer
- CPI (Cycles Per Instruction)
  - Machine Implementation, microarchitecture, compiler, application, algorithm, programming language, programmer
- Cycle Time (Seconds Per Cycle)
  - Process Technology, microarchitecture, programmer
Corollaries of Amdahl’s Law

- Maximum possible speedup $S_{\text{max}}$
  
  $S_{\text{max}} = \frac{1}{(1-x)}$

- Make the common case fast (i.e., $x$ should be large)
  
  Common = most time consuming, not necessarily the most frequent
  
  Use profiling tools to figure out

- Estimate the potential of parallel processing
  
  $S_{\text{par}} = \frac{x}{S} + (1-x)$

- Estimate the effect of multiple optimizations
  
  $S = \frac{1}{(1 - X_{\text{Opt1Only}} - X_{\text{Opt2Only}} - X_{\text{Opt1&Opt2}}) + \frac{X_{\text{Opt1}}}{S_{\text{Opt1Only}}} + \frac{X_{\text{Opt2}}}{S_{\text{Opt2Only}}} + \frac{X_{\text{Opt1&Opt2}}}{S_{\text{Opt1&Opt2}}}}$
Power/energy

- **Power** — the direct contributor of “heat”
  - Affects —
    - Packaging of the chip
    - Heat dissipation cost
  - Sources —
    - Dynamic power
    - Static power
- **Energy** — aggregated power consumption over time —
  - Affects — the electricity bill and battery life is related to energy!
  - Lower power does not necessary means better battery life if the processor slow down the application too much
Is TFLOPS (Giga FLoating-point Operations Per Second) a good metric?

- Cannot compare different ISA/compiler
  - What if the compiler can generate code with fewer instructions?
  - What if new architecture has more IC but also lower CPI?
- Does not make sense if the application is not floating point intensive

\[
\text{TFLOPS} = \frac{\text{# of floating point instructions} \times 10^9}{\text{Execution Time}}
\]

\[
= \frac{\text{IC} \times \% \text{ of floating point instructions}}{\text{IC} \times \text{CPI} \times \text{Cycle Time} \times 10^9} = \frac{\text{Clock Rate} \times \% \text{ FP ins.}}{\text{CPI} \times 10^9}
\]

A fair performance metric should consider all aspects of IC, CT, CPI
Outline

- Single-cycle processor
- Pipeline processor
- Design a 5-stage pipeline ARMv8 processor
Single-cycle processor — the simplest form of processors
Single cycle processor
• Break up the logic with “pipeline registers” into pipeline stages
  • These registers only changes their output at the triggered edge cycle
• Each stage can act on different instruction/data
• States/Control signals of instructions are hold in pipeline registers
After the 5th cycle, the processor can do 5 instructions in parallel
Pipelining

cycle #6
cycle #7
cycle #8
cycle #9
cycle #10

The processor can complete 1 instruction each cycle
CPI == 1 if everything works perfectly!
Single-cycle v.s. pipeline
Cycle time of a pipeline processor

- Critical path is the longest possible delay between two registers in a design.
- The critical path sets the cycle time, since the cycle time must be long enough for a signal to traverse the critical path.
- Lengthening or shortening non-critical paths does not change performance.
- Ideally, all paths are about the same length.
Designing a 5-stage pipeline processor for MIPS
Pipeline an ARM processor

- Instruction Fetch
  - Read from instruction memory
- Decode
  - Figure out the incoming instruction?
  - Fetch the operands from the registers
- Execution
  - Perform ALU functions
- Memory access
  - Read/write data memory
- Write back results to registers
  - Write to the register file

Instruction Fetch (IF)
Instruction Decode (ID)
Execution (EXE)
Memory Access (MEM)
Write Back (WB)
Single cycle processor
5-stage pipeline processor
add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9, $10, $11
sw $1, 0($12)
5-stage pipeline processor

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```
5-stage pipeline processor

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```
5-stage pipeline processor

- Instruction memory
- Read Address
- Instruction memory
- Instruction [31:0]
- Read Address
- Instruction [25:21]
- Instruction [20:16]
- Instruction [15:11]
- Instruction [16:11]
- Instruction [15:0]
- Read Register 1
- Read Register 2
- Read Data 1
- Read Data 2
- Write Register
- Write Data
- RegDst
- ALUOp
- ALUSrc
- RegWrite
- PC Src
- PC
- IF/ID
- ID/EX
- EX/MEM
- MEM/WB
- Adder
- Shift Left 2
- ALU
- Zero
- ALU Ctrl.
- Sign-extend
- Write Data
- Data memory
- Address
- Read Data
- 1 m
- 0 x
- 1 m
- 0 x
- add $1, $2, $3
- lw $4, 0($5)
- sub $6, $7, $8
- sub $9,$10,$11
- sw $1, 0($12)
5-stage pipeline processor

add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw $1, 0($12)
add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw $1, 0($12)
Simplified pipeline diagram

- Use symbols to represent the physical resources with the abbreviations for pipeline stages.
  - IF, ID, EXE, MEM, WB
- The horizontal axis represents the timeline, and the vertical axis represents the instruction stream.
- Example:

  ```
  add $1, $2, $3
  lw  $4, 0($5)
  sub $6, $7, $8
  sub $9,$10,$11
  sw  $1, 0($12)
  ```
Pipeline hazards
Pipeline hazards

• Even though we perfectly divide pipeline stages, it’s still hard to achieve CPI == 1.

• Pipeline hazards:
  • Structural hazard
    • The hardware does not allow two pipeline stages to work concurrently
  • Data hazard
    • A later instruction in a pipeline stage depends on the outcome of an earlier instruction in the pipeline
  • Control hazard
    • The processor is not clear about what’s the next instruction to fetch
Can we get the right result?

• Given the current 5-stage pipeline,

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EXE</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
</tr>
<tr>
<td></td>
<td>lw $4, 0($1)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
</tr>
<tr>
<td></td>
<td>sub $6, $7, $8</td>
<td>sub $6, $7, $8</td>
<td>sub $9, $1, $10</td>
<td>sub $9, $10, $11</td>
<td>sub $6, $7, $8</td>
</tr>
<tr>
<td></td>
<td>sw $1, 0($12)</td>
<td>sw $11, 0($12)</td>
<td>sw $1, 0($12)</td>
<td>sw $1, 0($12)</td>
<td>sw $1, 0($12)</td>
</tr>
</tbody>
</table>

b cannot get $1 produced by a before WB
both a and d are accessing $1 at 5th cycle
We don’t know if d & e will be executed or not

Data hazard  Structural hazard  Control hazard
Structural hazard
Structural hazard

- The hardware cannot support the combination of instructions that we want to execute at the same cycle
- The original pipeline incurs structural hazard when two instructions competing the same register.
- Solution: write early, read late
  - Writes occur at the clock edge and complete long enough before the end of the clock cycle.
  - This leaves enough time for outputs to settle for reads
  - The revised register file is the default one from now!

```plaintext
add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9,$10, $1
sw $1, 0($12)
```
Structural hazard

- The design of hardware causes structural hazard
- We need to modify the hardware design to avoid structural hazard
Data hazard
Data hazard

- When an instruction in the pipeline needs a value that is not available
- Data dependences
  - The output of an instruction is the input of a later instruction
  - May result in data hazard if the later instruction that consumes the result is still in the pipeline
Sol. of data hazard I: Stall

- When the source operand of an instruction is not ready, stall the pipeline
  - Suspend the instruction and the following instruction
  - Allow the previous instructions to proceed
  - This introduces a pipeline bubble: a bubble does nothing, propagate through the pipeline like a nop instruction
- How to stall the pipeline?
  - Disable the PC update
  - Disable the pipeline registers on the earlier pipeline stages
  - When the stall is over, re-enable the pipeline registers, PC updates
Performance of stall

15 cycles! CPI == 3
(If there is no stall, CPI should be just 1!)

add $1, $2, $3
lw $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw $1, 0($5)
Sol. of data hazard II: Forwarding

- The result is available after EXE and MEM stage, but publicized in WB!
- The data is already there, we should use it right away!
- Also called bypassing

```
add $1, $2, $3
lw $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw $1, 0($5)
```

We can obtain the result here!
Sol. of data hazard II: Forwarding

• Take the values, where ever they are!

add $1, $2, $3
lw $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw $1, 0($5)

10 cycles! CPI == 2 (Not optimal, but much better!)
When can/should we forward data?

- If the instruction entering the EXE stage consumes a result from a previous instruction that is entering MEM stage or WB stage
  - A source of the instruction entering EXE stage is the destination of an instruction entering MEM/WB stage
  - The previous instruction must be an instruction that updates register file
5-stage pipeline processor
5-stage pipeline processor
There is still a case that we have to stall...

- Revisit the following code:

  ```assembly
  add $1, $2, $3
  lw  $4, 0($1)
  sub $5, $2, $4
  sub $1, $3, $1
  sw  $1, 0($5)
  ```

  If the instruction entering EXE stage depends on a load instruction that does not finish its MEM stage yet, we have to stall!

  lw generates result at MEM stage, we have to stall
5-stage pipeline processor
if (option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
Control hazard
Control hazard

• The processor cannot determine the next PC to fetch

```
LOOP: lw  $t3, 0($s0)
     addi $t0, $t0, 1
     add  $v0, $v0, $t3
     addi $s0, $s0, 4
     bne $t1, $t0, LOOP
     sw  $v0, 0($s1)
```

7 cycles per loop
Branch prediction to reduce the overhead of control hazards
• The processor needs a “cheat sheet” for where the branch is going without calculating it
Dynamic branch prediction

- A 2-bit counter for each branch
- Predict taken if the counter value >= 2
- If the prediction in taken states, fetch from target PC, otherwise, use PC+4
  - If we guess right — **no penalty**
  - If we guess wrong — **flush** (clear pipeline registers) for mis-predicted instructions that are currently in IF and ID stages and reset the PC
Dynamic branch prediction

- A 2-bit counter for each branch
- Predict taken if the counter value >= 2
- If the prediction in taken states, fetch from target PC, otherwise, use PC+4

```
<table>
<thead>
<tr>
<th>PC</th>
<th>Branch Target Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400420</td>
<td>0x8048324 11</td>
</tr>
<tr>
<td>0x400464</td>
<td>0x8048392 10</td>
</tr>
<tr>
<td>0x400578</td>
<td>0x804850a 00</td>
</tr>
<tr>
<td>0x41000C</td>
<td>0x8049624 01</td>
</tr>
</tbody>
</table>
```
Performance of 2-bit counter

- 2-bit state machine for each branch

```
for(i = 0; i < 10; i++) {
    sum += a[i];
}
```

90% accuracy!

<table>
<thead>
<tr>
<th>i</th>
<th>state</th>
<th>predict</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4-9</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>T</td>
<td>NT</td>
</tr>
</tbody>
</table>

- Application: 80% ALU, 20% Branch, and branch resolved in EX stage, average CPI?
- \(1+20\% \times (1-90\%) \times 2 = 1.04\)
Local 2-bit predictor

```c
i = 0;
do {
    if( i % 3 != 0 ) // Branch Y, taken if i % 3 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100 )// Branch X
```

For branch X, almost 100%,
For branch Y, only 50%

```
<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>branch?</th>
<th>state</th>
<th>prediction</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>X</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>X</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>X</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
</tbody>
</table>
```
Branch prediction using global history
Recap: local 2-bit predictor

\[
i = 0;
\text{do} \{ \\
\quad \text{if}( i \% 3 \neq 0) \quad \text{// Branch Y, taken if } i \% 3 = 0 \\
\qquad a[i] *= 2; \\
\qquad a[i] += i; \\
\} \text{ while ( ++i < 100) // Branch X}
\]

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>branch?</th>
<th>state</th>
<th>prediction</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
</tbody>
</table>

For branch X, almost 100%, For branch Y, only 50%

Can we capture the pattern?
Instead of using the PC to choose the predictor, use a bit vector (global history register, GHR) made up of the previous branch outcomes.

- Global predictor: predictor using results from all branches
- Local predictor: predictor tracking states/history for each branch
- Each entry in the history table has its own counter.

First level

3-bit GHR: $\begin{bmatrix} 01 & 11 & 11 & 10 & 11 & 00 & 11 & 11 & 10 \end{bmatrix}$

$= 101$ (T, NT, T)

Pentium Pro uses this predictor
Performance of the 2-bit global predictor

```
i = 0;
do {
    if( i % 3 != 0) // Branch Y, taken if i % 3 == 0
        a[i] *= 2;
    a[i] += i;
    // Branch Y
} while ( ++i < 100) // Branch X
```

Nearly perfect after this
Branch prediction & your code
• Why the sorting the array speed up the code despite the increased instruction count?

```cpp
if (option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```
Demo: popcount

- How many 1s in binary representations
- Applications
  - Hamming weight
  - Encryption/decryption

```c
int main(int argc, char *argv[]) {
    uint64_t key = 0xdeadbeef;
    int count = 1000000000;
    uint64_t sum = 0;

    for (int i=0; i < count; i++)
    {
        sum += popcount (RandLFSR(key));
    }
    printf("Result: %lu\n", sum);
    return sum;
}
```
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 3, 2, 3, 3, 4};
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    return c;
}
Because popcount is important, both Intel and AMD added a POPCNT instruction in their processors with SSE4.2 and SSE4a.

In C/C++, you may use the intrinsic "_mm_popcnt_u64" to get # of "1"s in an unsigned 64-bit number.

You need to compile the program with `-m64 -msse4.2` flags to enable these new features.

```c
#include <smmintrin.h>
inline int popcount(uint64_t x) {
    int c = _mm_popcnt_u64(x);
    return c;
}
```
The pipeline of modern processors
The pipeline of modern processors

• Way deeper than 5 stages
• Shortening the pipeline stages helps improve the “cycle time”
• Higher marketing values since consumers usually link performance with frequencies
• Potentially higher power consumption as dynamic/active power = aCV²f
• If the execution time is better, still consume less energy
Intel Pentium 4 Microarch.
• Very deep pipeline: in order to achieve high frequency! (start from 1.5GHz)
  • 20 stages in Netburst
  • 31 stages in Prescott
• 103W (3.6GHz, 65nm)
• Reference
  • The Microarchitecture of the Pentium 4 Processor
AMD Athlon 64
AMD Athlon 64

- 12 stage pipeline

- 89W TDP (Opteron 2.2GHz 90nm)
Case Study
2.1 THE SKYLAKE MICROARCHITECTURE

The Skylake microarchitecture builds on the successes of the Haswell and Broadwell microarchitectures. The basic pipeline functionality of the Skylake microarchitecture is depicted in Figure 2-1.

The Skylake microarchitecture offers the following enhancements:

• Larger internal buffers to enable deeper OOO execution and higher cache bandwidth.
• Improved front end throughput.
• Improved branch predictor.
• Improved divider throughput and latency.
• Lower power consumption.
• Improved SMT performance with Hyper-Threading Technology.
• Balanced floating-point ADD, MUL, FMA throughput and latency.

The microarchitecture supports flexible integration of multiple processor cores with a shared uncore subsystem consisting of a number of components including a ring interconnect to multiple slices of L3 (an off-die L4 is optional), processor graphics, integrated memory controller, interconnect fabrics, etc. A four-core configuration can be supported similar to the arrangement shown in Figure 2-3.

Good reference for intel microarchitectures:
Demo revisited

- Why the sorting the array speed up the code despite the increased instruction count?

```cpp
if (option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```
Thank you!
Pipeline
Hung-Wei Tseng