With today’s announcement, Kim joins the ranks of The Onion’s prior “Sexiest Man Alive” winners, including:

2011: Bashar al-Assad
2010: Bernie Madoff
2009: Charles and David Koch (co-winners)
2008: Ted Kaczynski
2007: T. Herman Zweibel
Brass tacks

In colloquial English, brass tacks (often "get down to brass tacks") refers to the fundamental or essential elements of a topic. As in the pieces of brass inserted into the edges of the paper creating the binding of a document. The points which hold an idea together.

Brass tacks are, of course, real as well as figurative items and two of the most commonly repeated supposed derivations refer to actual tacks. Firstly, there's the use of brass-headed nails as fabric fixings in the furniture trade, chosen on account of their decorative appearance and imperviousness to rust. Such brass tacks were commonly used in Tudor furniture and long predate the use of the phrase, which would tend to argue against that usage as the origin - why wait hundreds of years and then coin the phrase from that source? The supporters of that idea say that, in order to re-upholster a chair, the upholsterer would need first to remove all the tacks and fabric coverings, thus getting down to the basic frame of the chair. While that is true, it hardly seems to match the meaning of the expression, as the tacks would be the first thing to be removed rather than the last.

Earliest attestation in 1863 US, specifically Texas. A theory is that it comes from the brass tacks in the counter of a hardware store or draper's shop used to measure cloth in precise units (rather than holding one end to the nose and stretching out the arm to approximately one yard).
Hits vs misses

\[ t_a = h t_p + (1 - h) t_s \]

- \( t_a \): actual access time
- \( t_p \): primary memory access time
- \( t_s \): secondary memory access time
- \( h \): hit ratio, or hits per memory reference
### 3⁰ Example

4 sets, one line per block,
2 bytes per block, 4-bit addresses

<table>
<thead>
<tr>
<th>Address (decimal)</th>
<th>Tag bit (t=1)</th>
<th>Index bits (s=2)</th>
<th>Offset bits (b=1)</th>
<th>Block number (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>00</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>11</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

Read at address 0
Read 1
Read 13
Read 8
Read 0
Set-Associative Cache (2-way)

<table>
<thead>
<tr>
<th>Tag Memory</th>
<th>Valid Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Tag field, 5 bits

### Memory Address:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Group</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>
• At this point, should be able to do 6.13-6.17
• Practice problem: 6.18
Fully Associative Cache

- Primary cache memory has 1 region ($S=1$)
- What does this imply?
Mapping Function

- **Placement Strategies** -- where to place an incoming block in the cache.
- **Replacement Strategies** -- Which block to replace when a cache miss occurs.
- **Read and Write Policies** -- how to handle reads and writes upon cache hits and misses.
Hits and Misses

- What happens when there is a:
  - Cache Hit on a read or write?
  - Cache Miss on a read or write?
- Are the same things done for both the direct-mapped and set-associative caches?
Cache Hits Write Policy

- There are basically two methods:
  - *write through*: updates both the cache and main memory.
  - *write back* -- updates only the cache, and uses a *dirty bit* to indicate that it should be stored in main memory when purged from the cache.
Cache Miss Read Policy

- **direct-mapped cache:**
  - the valid bit is checked
    - **if not valid,** the block is read in, possibly forwarding the requested word;
    - **if valid,** the current block is eliminated (with consideration for the dirty bit) then the new block is loaded.
Cache Miss Read Policy

- set-associative cache:
  - if an empty line exists load block there
  - If no lines are empty, select a line to eliminate from the cache by one of two methods:
    - Least Recently Used (LRU)
    - Random Replacement
Cache Miss Write Policy

• There are two methods:

• **write allocate** -- the block is loaded into the cache and updated (commonly associated with write-back caches).

• **write-no allocate** -- the data is written directly to main memory and commonly associated with write-through caches.
Thrashing

- Thrashing can occur in many situations and results from a system being so overloaded that it spends all its time moving resources around so they can be used.

- In caches, thrashing results when the same few blocks are being moved in and out of a common cache location.
Performance Impact

• Cache Size
  – Hit rate vs. Hit Time

• Block Size
  – Hit rate vs. fewer cache lines & higher miss penalty

• Associatively
  – Larger E: reduce risk of thrashing
  – But slower hit time, more tag bits, higher miss penalty

• Write strategy
  – Write through: simpler, can use write buffer, read misses less expensive
  – Write back: fewer transfers (more important in lower part of hierarchy)
Build a Cache

• Create two caches:
  – A direct mapped cache
  – A 4-way set associative cache

• Assume that the main memory is 4 MB.

• What are:
  – How would you breakup the main memory address?
  – What would the cache address look like?
  – How much cache memory would you need?
Performance Realities

• *There’s more to performance than asymptotic complexity*

• Constant factors matter too!
  – Easily see 10:1 performance range depending on how code is written
  – Must optimize at multiple levels:
    • algorithm, data representations, procedures, and loops

• Must understand system to optimize performance
  – How programs are compiled and executed
  – How to measure program performance and identify bottlenecks
  – How to improve performance without destroying code modularity and generality
Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies

- Don’t (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter

- Have difficulty overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects
Limitations of Optimizing Compilers

• Operate under fundamental constraint
  – Must not cause any change in program behavior
  – Often prevents it from making optimizations when would only affect behavior under pathological conditions.

• Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  – e.g., Data ranges may be more limited than variable types suggest

• Most analysis is performed only within procedures
  – Whole-program analysis is too expensive in most cases

• Most analysis is based only on static information
  – Compiler has difficulty anticipating run-time inputs

• When in doubt, the compiler must be conservative
Generally Useful Optimizations

• Optimizations that you or the compiler should do regardless of processor / compiler

• Code Motion
  – Reduce frequency with which computation performed
    • If it will always produce same result
    • Especially moving code out of loop

```c
void set_row(double *a, double *b, long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```
void set_row(double *a, double *b, long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}

long j;
long ni = n*i;
double *rowp = a+ni;
for (j = 0; j < n; j++)
    *rowp++ = b[j];

set_row:
    testq %rcx, %rcx
    jle .L4
    movq %rcx, %rax
    imulq %rdx, %rax
    leaq (%rdi,%rax,8), %rdx
    movl $0, %r8d
    .L3:
    movq (%rsi,%r8,8), %rax
    movq %rax, (%rdx)
    addq $1, %r8
    addq $8, %rdx
    cmpq %r8, %rcx
    jg .L3
    .L4:
    rep ; ret

long j;
long ni = n*i;
double *rowp = a+ni;
for (j = 0; j < n; j++)
    *rowp++ = b[j];

set_row:
    testq %rcx, %rcx
    jle .L4
    movq %rcx, %rax
    imulq %rdx, %rax
    leaq (%rdi,%rax,8), %rdx
    movl $0, %r8d
    .L3:
    movq (%rsi,%r8,8), %rax
    movq %rax, (%rdx)
    addq $1, %r8
    addq $8, %rdx
    cmpq %r8, %rcx
    jg .L3
    .L4:
    rep ; ret
Reduction in Strength

– Replace costly operation with simpler one
– Shift, add instead of multiply or divide
  \[16 \times x \rightarrow x \ll 4\]
  
  • Utility machine dependent
  • Depends on cost of multiply or divide instruction
    – On Intel Nehalem, integer multiply requires 3 CPU cycles

– Recognize sequence of products

```c
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```