http://tik.services
Day 1: Mostly the upper layers & software

Day 2: Mostly the lower layers & hardware
The upper layers (ctd)
Recap: main types of code

- Source code
- Object code
- Bytecode
- Machine code
- Microcode
Python

print("Hello, world!")
Interpreters

• An interpreter directly executes instructions written in a language, without requiring them having been compiled to machine language
• Often work “statement by statement”
• You only get an error when you reach its location
Python execution

• CPython
  – Lexes
  – Parses
  – Compiles Python to bytecode (.pyc)
  – Interprets the bytecode

• PyPy
  – Compile Python on the fly
  – Claims to be faster
  – Won’t help for short or C-heavy programs
Python internals

>>> def x():
    ...   print 1
    ...

>>> x.func_code
<code object x at 0x7fcf6d4dad50, file "<stdin>", line 1>

>>> x.func_code.co_code
'd\x01\x00GHd\x00\x00S'

>>>
Python internals

```python
>>> def x():
    ...   print 1
    ...

>>> import dis

>>> dis.dis(x)
2           0 LOAD_CONST               1 (1)
3 PRINT_ITEM
4 PRINT_NEWLINE
5 LOAD_CONST
8 RETURN_VALUE
```
alert("Hello World");
JavaScript execution

- A web browser has an execution “engine”
  - Often written in C/C++
  - Interpreter + JIT compiler + garbage collector
  - E.g., Google’s V8, Mozilla’s SpiderMonkey
- SpiderMonkey compiles JS to an intermediate language and interprets it
- V8 compiles JS to machine instructions
  - Contains two compilers
SpiderMonkey operation

• Compiler
  – Lexical scanner + Parser + Code generator
  – Consumes JS code, produces a script
  – The script contains bytecode, source annotations, literals, nested scripts

• Interpreter
  – Steps through the bytecode
  – Reentrant (JS-2-JS vs JS-2-C-2-JS)

• Garbage Collector
  – Triggers after a byte limit is passed

• Decompiler for debugging
Why must everything be in English?

What is ChinesePython?

“ChinesePython is a sort of translation work of the Python language into Chinese. The most notable changes to the Python language is that its keywords, variable names, built-in types and their methods are all translated into Chinese. That enables a programmer to write Chinese programs in Python’s style.”
Why must everything be in English?

```python
>>> 12 + 3
15
```

```python
>>> 15 * 2
30
```

```python
>>> 2 ** 4
16
```

```python
>>> 2 ** 64
OverflowError: integer exponentiation outcome too large
```
Why must everything be in English?

写 (print)
删除 (del)
定义/函数 (def)
忽略 (pass)
中断 (break)
下一個 (continue)
傳回 (return)
示警 (raise)
載入 ... 名 ... (import ... as ...)
從 ... 載入 ...
共用 (global)
執行 (exec)

斷言 (assert)
如 ... 不然 ... 否則 .. (if .. elif .. else ..)
只要 ... 否則 .. (while .. else ..)
取 .. 自 .. 然後 .. (for .. in .. else ..)
試 .. 失敗 .. 否則 .. 然後 .. (try .. except .. else .. finally ..)
概念/類別 (class)
來自/不來自 (in / not in)
是/不是 (is / is not)
或 or
且 and
不是 not
Why must everything be in English?

```python
number = 23
running = True

while running:
    guess = int(input('Enter an integer: '))

    if guess == number:
        print('Congratulations, you guessed it.')
        running = False
    elif guess < number:
        print('No, it is higher than that.')
    else:
        print('No, it is lower than that.')
```

數字 = 23
運行 = 真
當 運行:

猜測 = 數字(輸入('輸入一個數字: '))

如果 猜測 == 數字:
    印出 '恭喜，你猜對了.'
    運行 = 假
假使 猜測 < 數字:
    印出 '錯了，數字再大一點.'
否則:
    印出 '錯了，數字再小一點.'
The lower layers
Recap: main types of code

- Source code
- Object code
- Bytecode
- Machine code
- Microcode
Assembly/Machine code

- Assembly code – typically plain-text, readable by humans
- Machine code – binary code that can be directly fed to the CPU for execution
- Correspondence between the two is not guaranteed
- Correspondence between code on disk and in memory is not guaranteed
# Side by side

## Machine code

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>4018c0:</td>
<td>55</td>
<td>push rbp</td>
</tr>
<tr>
<td>4018c1:</td>
<td>48 89 e5</td>
<td>mov rbp, rsp</td>
</tr>
<tr>
<td>4018c4:</td>
<td><strong>53</strong></td>
<td>push rbx</td>
</tr>
<tr>
<td>4018c5:</td>
<td>48 83 ec 08</td>
<td>sub rsp, 0x8</td>
</tr>
<tr>
<td>4018c9:</td>
<td>80 3d 10 9a 20 00 00</td>
<td>cmp BYTE PTR [rip+0x209a10], 0x0</td>
</tr>
<tr>
<td>4018d0:</td>
<td>75 4b</td>
<td>jne 40191d &lt;time@plt+0xc5&gt;</td>
</tr>
<tr>
<td>4018d2:</td>
<td>bb 10 a4 60 00</td>
<td>mov ebx, 0x60a410</td>
</tr>
<tr>
<td>4018d7:</td>
<td>48 8b 05 0a 9a 20 00</td>
<td>mov rax, QWORD PTR [rip+0x209a0a]</td>
</tr>
<tr>
<td>4018de:</td>
<td>48 81 eb 08 a4 60 00</td>
<td>sub rbx, 0x60a408</td>
</tr>
<tr>
<td>4018e5:</td>
<td>48 c1 fb 03</td>
<td>sar rbx, 0x3</td>
</tr>
<tr>
<td>4018e9:</td>
<td>48 83 eb 01</td>
<td>sub rbx, 0x1</td>
</tr>
<tr>
<td>4018ed:</td>
<td>48 39 d8</td>
<td>cmp rax, rbx</td>
</tr>
<tr>
<td>4018f0:</td>
<td>73 24</td>
<td>jae 401916 &lt;time@plt+0xbe&gt;</td>
</tr>
<tr>
<td>4018f2:</td>
<td>66 0f 1f 44 00 00</td>
<td>nop WORD PTR [rax+rax*1+0x0]</td>
</tr>
<tr>
<td>4018f8:</td>
<td>48 83 c0 01</td>
<td>add rax, 0x1</td>
</tr>
</tbody>
</table>
RISC

• Reduced Instruction Set Computer
• Designed for speed
  – Few, simple instructions, mostly 1 cycle execution
  – Thus less logic transistors are required
  – Typically lots of registers
• Examples:
  – MIPS (Stanford)
  – SPARC (Berkeley)
  – Power (IBM)
• ARM – “Advanced Risc Machine”
  – RISC or CISC?
MIPS instruction set composition
111 instructions

- 21 arithmetic instructions (+, -, *, /, %)
- 8 logic instructions (&, |, ~)
- 8 bit manipulation instructions
- 12 comparison instructions (>, <, =, >=, =<, ¬)
- 25 branch/jump instructions
- 15 load instructions
- 10 store instructions
- 8 move instructions
- 4 miscellaneous instructions
Example MIPS assembly

Source
void swap (int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

Assembly
swap: sll $t1, $a1, 2
        add $t1, $a0, $t1
        lw $t0, 0($t1)
        lw $t2, 4($t1)
        sw $t2, 0($t1)
        sw $t0, 4($t1)
        jr $ra
CISC

• Complex Instruction Set Computer
• Designed for convenience
  – Internal microcode used to execute complex instructions
  – E.g., M-2-M instructions
  – Instructions need more clock cycles
  – Lots of logic, fewer registers
• Example: x86
## x86 assembly – how big?

Credit: Stefan Heule, Stanford

<table>
<thead>
<tr>
<th>Measure</th>
<th>Count</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT&amp;T mnemonic (e.g., addl)</td>
<td>1,279</td>
<td>Counts the number of unique mnemonics in AT&amp;T syntax.</td>
</tr>
<tr>
<td>Intel mnemonic (e.g., add)</td>
<td>981</td>
<td>Rough estimate of the number of different kinds of operations the x86 instruction set can perform, ignoring the operand type and size.</td>
</tr>
<tr>
<td>Mnemonic and operand types (e.g., add_r32_imm32)</td>
<td>3,683</td>
<td>Upper bound on the number of instructions (all distinctions considered)</td>
</tr>
<tr>
<td>Mnemonic and operand width (e.g., add_32_32)</td>
<td>2,034</td>
<td>This is an estimate of the number of different kinds of instructions, with varying operand sizes. Does not distinguish instructions that operate on registers vs. constants vs. memory locations.</td>
</tr>
</tbody>
</table>
The twist - Micro-ops
a.k.a. μops

- Many instructions get translated into micro-ops.
  - Secret, little operations
- Decoded uops are normally stored in a cache (1-2k uops)
- Micro-code: micro-ops and data structures in internal, high-speed CPU memory
  - Can be sequenced internally by the CPU (for instructions >4 uops long)
  - Examples: transcendentals (sin/cos etc), floating-point assists, AVX to SSE transitions, string operations (e.g., REP MOVSB)
Reverse engineering uops
Credit: Benjamin Kollenda and Philipp Koppe, CCC 2017
Second twist - ARM

• ARM uses micro-ops internally, too!
• But not micro-code

• What is RISC, what is CISC?
Westmere core diagram
A. Nowak / D. Levinthal
Caching

Approximate memory Latencies on Intel Haswell CPUs

Processor Core (Registers)
- **L1I** (32 KB)  
  - 64 B/1c, 4 c latency
- **L1D** (32 KB)  
  - 64 B/1c, 12 c latency

L2 (256 KB)  
- 64 B/1c for all cores ~36 c latency

Shared L3 (8192 KB)  
- ~24 B/c for all cores 200-450 c latency

Local memory (rather large, might be persistent)

\[ c = \text{cycle} \]  

Adapted from S. Jarp

08-Feb-18
In-order vs. Out-of-order

**In-order**
- Execute instructions sequentially as they come.

**Out-of-order**
- Look ahead in the instruction stream for instructions that are not dependent and execute them out of order.
Speculation
(hardware)

• Speculation does things before it’s certain they need or can be done. CPUs speculate all the time.

• Control speculation
  – E.g., predicting a path

• Data speculation
  – E.g., predicting needed data
The TAQ
Trollingly Asked Questions
Digression: exploiting hardware

Meltdown

- You’ve heard of Meltdown and Spectre

- Meltdown
  - The kernel’s Page Table mappings are shared between processes
  - CPUs (AMD, ARM, Intel) speculate on memory accesses
  - Intel chips allow speculative usage of kernel data
  - Cache effects of this speculation can be timed and used to infer kernel data
  - Reading is very slow

- Solution
  - Kernel Page Table Isolation
  - Performance penalties for heavy kernel use
Digression: exploiting hardware

Spectre

- Spectre
  - The Branch Prediction Unit collects statistics on branches (taken or not) to better predict outcomes
  - It does not gather metadata, such as which process the branch came from
  - An attacker can “train” the BPU, enabling it to control speculative execution and cache of the target process
  - Only the victim process can “scan” the cache
  - Caveats: requires victim source knowledge, slow reading

- Solution
  - Serializing instructions – force reads and writes to flush
Parallelism
Physical CPU layout
Intel “Broadwell” Core i7 example

Image: Intel
Physical CPU layout
Intel “Skylake” Xeon example
IC design phases

- System Specification
- Architectural Design
- Functional Design and Logic Design
- Circuit Design
- Physical Design
  - Physical Verification and Signoff
  - Fabrication
  - Packaging and Testing
  - Chip

- Partitioning
- Floorplanning
- Placement
- Clock Tree Synthesis
- Signal Routing
- Timing Closure

Designing a CPU

main steps

1. Hardware Description Language (HDL) design and verification
   – Essentially the CPU is designed in code
   – Simulated intensively on compute farms

2. Synthesis – HDL becomes simple logic gates (“netlist”)

3. Post-synthesis verification
   – Are all assumed timings correct?

4. Device layout – all components “physically” placed on a surface
   – Following physical constraints
   – Wiring, locations of subsystems, cache sizes, etc
Physical design flow

1. Design entry
2. Initial synthesis
3. Initial floorplan
4. Synthesis with load constraints
5. Timing-driven placement
6. Synthesis with in-place optimization
7. Detailed placement

Increasing accuracy of wire-load estimates

EDA software
Controlling electrons

- Despite all those steps, not all CPUs turn out the same, even if they look the same
- Binning is the process of checking what a CPU can do and where it fits
  - Varying #cores, HT, clock speed, VT, cache
Physical process, 2 layers

1. Get wafer
2. Grow oxide
3. Deposit Si3N4 using LPCVD
4. Apply photoreist
5. Expose using Mask 1
6. Cure resist
7. Remove uncured resist
8. Etch
9. Remove other resist
10. Grow oxide using LOCOS
11. Apply resist
12. Expose using Mask 2
13. Cure resist
14. Remove uncured resist
15. Ion implant for P-wells (Boron)
16. Remove other resist
17. Apply resist
18. Expose using Mask 3
19. Cure resist
20. Remove uncured resist
21. Ion implant for N-wells (Phosphorus)
22. Remove other resist
23. High temp drive-in to fix implant damage/grow wells
24. Apply resist
25. Expose using Mask 4
26. Cure resist
27. Remove uncured resist
28. Ion implant for P-type gate (Boron)
29. Remove other resist
30. Apply resist
31. Expose using Mask 5
32. Cure resist
33. Remove uncured resist
34. Ion implant for N-type gate (Phosphorus)
35. Remove other resist
36. Strip and regrow thin oxide for gate
37. Deposit polysilicon
38. Apply resist
39. Expose using Mask 6 (Critical litho step!)
40. Cure resist
41. Remove uncured resist
42. Etch poly (Critical etch step!)
43. Remove other resist
44. Apply resist
45. Expose using Mask 7
46. Cure resist
47. Remove uncured resist
48. Ion implant for N-type source/drain (Phosphorus)
49. Remove other resist
50. Apply resist
51. Expose using Mask 8
52. Cure resist
53. Remove uncured resist
54. Ion implant for P-type source/drain (Boron)
55. Remove other resist
56. Apply resist
57. Expose using Mask 9
58. Cure resist
59. Remove uncured resist
60. Ion implant for N+ source/drain (Arsenic)
61. Remove other resist
62. High temp drive in to fix implant damage/grow wells
63. Apply resist
64. Expose using Mask 10
65. Cure resist
66. Remove uncured resist
67. Ion implant for P+ source/drain (Boron)
68. Remove other resist
69. High temp drive in to repair implant damage
70. Unmasked etch
71. Ti deposited by sputtering
72. React in N2 ambient to form TiSi2 and TiN
73. Apply resist
74. Expose using Mask 11
75. Cure resist
76. Remove uncured resist
77. Etch TiN
78. Remove other resist
79. Deposit conformal layer of SiO2
80. Planarize using Chemical/Mechanical Polish (CMP)
81. Apply resist
82. Expose using Mask 12
83. Cure resist
84. Remove uncured resist
85. Etch SiO2
86. Remove other resist
87. Deposit TiN
88. Deposit W
89. Planarize using CMP
90. Remove uncured resist
91. Expose using Mask 13
92. Cure resist
93. Apply resist
94. Expose using Mask 14
95. Cure resist
96. Remove uncured resist
97. Etch SiO2
98. Anisotropic deposit of SiO2
99. Apply resist
100. Expose using Mask 16
101. Cure resist
102. Remove uncured resist
103. Etch SiO2
104. Remove other resist
105. Deposit TiN
106. Deposit W
107. Planarize using CMP
108. Deposit Al
109. Apply resist
110. Expose using Mask 15
111. Cure resist
112. Remove uncured resist
113. Etch SiO2
114. Plasma etch Al to form Metal 1
115. Deposit conformal layer of SiO2
116. Planarize using Chemical/Mechanical Polish (CMP)
117. Apply resist
118. Expose using Mask 14
119. Cure resist
120. Remove uncured resist
121. Plasma etch Al to form Metal 1
122. Deposit conformal layer of SiO2
123. Planarize using Chemical/Mechanical Polish (CMP)
124. Apply resist
125. Expose using Mask 13
126. Cure resist
127. Remove uncured resist
128. Etch SiO2
129. Deposit W
130. Planarize using CMP
131. Deposit Al
132. Apply resist
133. Expose using Mask 12
134. Cure resist
135. Remove uncured resist
136. Etch SiO2
137. Remove other resist
138. Ion implant for P+ source/drain (Boron)
139. Remove other resist
140. Conformal deposit of SiO2
141. Anisotropic etch to create sidewall spacers around gate
142. Apply resist
143. Expose using Mask 10
144. Cure resist
145. Remove uncured resist
146. Ion implant for P-type source/drain (Boron)
147. Remove other resist
148. High temp drive in to repair implant damage
149. Unmasked etch
150. Ti deposited by sputtering
151. React in N2 ambient to form TiSi2 and TiN
152. Apply resist
153. Expose using Mask 11
154. Cure resist
155. Remove uncured resist
156. Etch TiN
157. Remove other resist
158. Deposit conformal layer of SiO2
159. Planarize using Chemical/Mechanical Polish (CMP)
160. Apply resist
161. Expose using Mask 12
162. Cure resist
163. Remove uncured resist
164. Etch SiO2
165. Remove other resist
166. Deposit TiN
167. Deposit W
168. Planarize using CMP
169. Remove uncured resist
170. Expose using Mask 13
171. Cure resist
172. Remove uncured resist
173. Etch SiO2
174. Anisotropic deposit of SiO2
175. Apply resist
176. Expose using Mask 10
177. Cure resist
178. Remove uncured resist
179. Etch SiO2
180. Remove other resist
181. Deposit W
Thank you
e-mail: an@tik.services
http://tik.services

All content which is original in this work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.
RISC vs. CISC study at UWM

Little difference!

Table 2: Findings of VRG RISC vs CISC study

<table>
<thead>
<tr>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Large performance gaps exist.</td>
</tr>
</tbody>
</table>
| 2. Cycle-count gaps are less than 3×
  (A8 to Atom OoO Processors to i7) |
| 3. Cycles per instruction (CPI)
can be less on x86 implementation |
| 4. ISA performance effects
  indistinguishable between x86 and ARM |
| 5. Microarchitecture, not the ISA,
  responsible for performance differences |
| 6. Beyond micro-op translation, x86 ISA
  introduces no overheads over ARM and MIPS ISA |

<table>
<thead>
<tr>
<th>Power</th>
</tr>
</thead>
</table>
| 7. Power consumption does not have
  a direct correlation to ISA |
| 8. Choice of power or perf. optimization
  impacts power use more than does ISA |
| 9. Energy use primarily a design choice;
  ISA's impact insignificant |

<table>
<thead>
<tr>
<th>Tradeoffs</th>
</tr>
</thead>
</table>
| 10. High-performance processors require more
  power than lower-performance processors |
| 11. It is the microarchitecture
  and design methodology that really matters |