Computer System Models

Topics to be covered:

- Layered view
- Structural view
- Operational view

Why?

- Provide a context for computing studies
- Gain a deeper understanding of how programs work
Computer Systems

Modern computer systems are layered

Applications: Word, IE, nedit, Firefox,…
Languages: C, C++, VisualC, Haskell, Java,…
O/S: Linux, Windows, MacOS,…
Machines: Intel 8086/Pentium, PowerPC, ARM, Motorola 68000, MIPS,…
Level-oriented view of Comp. Systems

Level 5
Problem-oriented language level
Translation (compiler)

Level 4
Assembly language level
Translation (assembler)

Level 3
Operating system machine level
Partial interpretation (operating system)

Level 2
Instruction set architecture level
Interpretation (microprogram) or direct execution

Level 1
Microarchitecture level
Hardware

Level 0
Digital logic level

COMP1911/1921
COMP3131 Prog. Lang & Compilers
COMP2121 Micros & Interfacing
COMP3232 Operating Systems

COMP3211 Computer Architecture
COMP3222 Dig. Circuits & Systems
## History of Computer Technology

<table>
<thead>
<tr>
<th>Decade</th>
<th>Impact</th>
<th>Hardware Technology</th>
<th>Software Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>1940’s</td>
<td>First prototypes</td>
<td>Vacuum tubes</td>
<td>Machine language</td>
</tr>
<tr>
<td>1950’s</td>
<td>First commercial computers</td>
<td>Transistors</td>
<td>FORTRAN, COBOL, LISP</td>
</tr>
<tr>
<td>1960’s</td>
<td>Widespread use in Business/defence</td>
<td>Integrated circuits</td>
<td>Multi-user operating systems</td>
</tr>
<tr>
<td>1970’s</td>
<td>Minicomputers</td>
<td>LSI, VLSI</td>
<td>UNIX, C</td>
</tr>
<tr>
<td>1980’s</td>
<td>Microcomputers</td>
<td>RISC</td>
<td>windows, mouse, menus</td>
</tr>
<tr>
<td>1990’s</td>
<td>Global network</td>
<td></td>
<td>Haskell, Perl, Tcl, VB, OO</td>
</tr>
<tr>
<td>2000’s</td>
<td>Ubiquitous computing</td>
<td>SOC</td>
<td>HTML, Java, VRML</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>XML, application-specific, services-oriented</td>
</tr>
</tbody>
</table>

**One constant: the underlying machine model**
Reminder

- If you have passed COMP1917 in 2008 or you have passed COMP1911 prior to 2008
  - You should do 1927 if you are a Computing major and don’t want to extend your studies, or you are interested in keeping the widest options open for further studies in Computing.

- If you have passed COMP1911 prior to 2008 and you are a non-Computing major
  - You may stay in COMP1921
The von Neumann Model

- Processor: control, calculation
- Memory: data & program storage
- Input/Output: interface to world
Processor (CPU)
The Processor’s Task

```c
{  
  Register PC; /* program counter */

  forever {  
    fetch next instruction from Memory[PC++];
    determine what kind of instruction;
    fetch any necessary data;
    carry out the specified operation;
  }
}
```

Register: word-sized storage cell (a “word” is typically 32 bits large, but most recent systems have moved to 64 bits)

Some processors execute the steps within the loop above sequentially, either in one long or multiple smaller cycles, and others complete all these steps on multiple instructions in parallel
Example of von Neumann processing

- The processor may need to compute

\[ A = B + C; \]

- This could be translated at machine level to the instruction:

\[
\text{add } $t0, $t1, $t2
\]

- But before executing this instruction, register $t1 would need to have had B loaded into it (another instruction)
- And after this instruction, register $t0 would need to be saved into the main memory location for A (yet another instruction)
Processor operations

- CPUs typically provide operations for:
  - data movement (reg-to-reg, reg-to-mem)
  - arithmetic calculation (e.g. + - * /)
  - logical calculation (e.g. && || !)
  - comparison (e.g. ==, >, <, >=, <=)
  - bit manipulation (e.g. ~ & | ^ >> <<)
  - program control (goto)
  - input/output (read, write)
## Top 10 80x86 Instructions - running integer codes

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Integer</th>
<th>Average</th>
<th>Percent total executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td></td>
<td></td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td></td>
<td></td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td></td>
<td></td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td></td>
<td></td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td></td>
<td></td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td></td>
<td></td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td></td>
<td></td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td></td>
<td></td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td></td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td></td>
<td></td>
<td>1%</td>
</tr>
</tbody>
</table>

**Total** 96%

° Simple instructions dominate instruction frequency

One fifth of all instructions are branches!
Main (Primary) Memory
- Program view of memory

- Effectively: a large (slow) array of registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>1008</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>100C</td>
<td>F F F F F F F F</td>
</tr>
<tr>
<td>1010</td>
<td></td>
</tr>
</tbody>
</table>

- Accessed by giving an address (location, offset) and reading/writing data there.
Memory hierarchy
- a more complete picture of memory

- To make memory fast enough to keep up with processors it has to be small
  - But programs have large appetites for memory

- To cope with the competing requirements for fast and large memory, computer architects have developed a hierarchy in which memory is accessed:
  - Few, very fast registers within the processor are at the top & provide access to a few 100B every clock cycle
  - On-chip cache provides access to MB in 10s cycles
  - Main or primary memory of GB is accessed in 100s of cycles
  - Disk or secondary memory provides access to TB in 1000’s of cycles
Memory system

Processor

- cache
  (several 100KB – several MB)

Memory/system bus

- 1000’s MB/s
- 100’s MB/s

Memory Controller

I/O Controller

Primary Memory

- (several 100MB – several GB)

ATA/PCI/SCSI Peripheral/IO bus

- 100’s MB/s
- 10’s MB/s
- 100’s MB/s
- 10’s MB/s

Tape

- (10’s GB – 100’s GB)

Hard disk

- (several 10’s GB)

CD/DVD

- (several 100’s MB – several GB)
Primary Memory

- Programs and data are stored in secondary memory such as disks until they are required for execution, when they are “paged” into main memory from where they can be more readily accessed as execution proceeds.
- Program code and data are moved from primary memory into cache when accessed.
- Data is fetched from cache into registers as determined by data “load” instructions.
- Results are written back to cache and ultimately main memory by “store” instructions.
Primary Memory Types

- RAM — Random Access Memory
  - Static: SRAM – fast, expensive (6 transistors per cell), e.g. registers & cache
  - Dynamic: DRAM – slower, cheaper (1 transistor per cell), denser, e.g. main memory

- ROM — Read Only Memory

- EEPROM — Electrically Erasable Programmable ROM

- NVRAM — Non-volatile RAM
  - e.g. flash memory
Secondary Memory

- Primary memory allows faster access but is often too small for computing requirements
- Contents of primary memory are lost when power source removed
- Secondary memory, e.g. disks, CD, tape etc. is usually:
  - cheaper per GB
  - slower to access
  - has larger storage capacity
  - persistent
  - removable
Examples of Secondary Memory

- Hard disk (IDE/SCSI/USB)
- Flash memory
- DVD
- CD-ROM/CD-R/CD-RW
- Floppy disk
- Magnetic tape
Cache Memory

- Faster access than main memory but usually much smaller storage capacity

- *Locality principle* — neighbouring memory locations loaded next — why small cache loading blocks of contiguous addresses works…
  - **Spatial locality:**
    - access array elements or fields in records one after another
  - **Temporal locality:**
    - access instructions in a loop over and over again during a small period of time
**Input/Output Devices**

Vast range of devices are interfaced:

<table>
<thead>
<tr>
<th>Device</th>
<th>Read/Write</th>
<th>Speed</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>disks</td>
<td>r/w</td>
<td>high</td>
<td>high-volume storage</td>
</tr>
<tr>
<td>tape</td>
<td>r/w</td>
<td>low</td>
<td>archiving</td>
</tr>
<tr>
<td>cd-rom</td>
<td>r/o</td>
<td>medium</td>
<td>storage</td>
</tr>
<tr>
<td>display</td>
<td>w/o</td>
<td>medium</td>
<td>CRT, LC, ...</td>
</tr>
<tr>
<td>keyboard</td>
<td>r/o</td>
<td>low</td>
<td></td>
</tr>
<tr>
<td>mouse</td>
<td>r/o</td>
<td>low</td>
<td>1,3-button</td>
</tr>
<tr>
<td>other computers</td>
<td>r/w</td>
<td>varying</td>
<td>networks</td>
</tr>
<tr>
<td>VR-helmet</td>
<td>r/w</td>
<td>high</td>
<td>games</td>
</tr>
<tr>
<td>mechanical</td>
<td>r/w</td>
<td>low</td>
<td>embedded systems</td>
</tr>
<tr>
<td>equipment</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
From high-level to low-level languages

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

Real machines can’t execute C (or Haskell or…).

lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)

<table>
<thead>
<tr>
<th>0000</th>
<th>1001</th>
<th>1100</th>
<th>0110</th>
<th>1010</th>
<th>1111</th>
<th>0101</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>1111</td>
<td>0101</td>
<td>1000</td>
<td>0000</td>
<td>1001</td>
<td>1100</td>
<td>0110</td>
</tr>
<tr>
<td>1100</td>
<td>0110</td>
<td>1010</td>
<td>1111</td>
<td>0101</td>
<td>1000</td>
<td>0000</td>
<td>1001</td>
</tr>
<tr>
<td>0101</td>
<td>1000</td>
<td>0000</td>
<td>1001</td>
<td>1100</td>
<td>0110</td>
<td>1010</td>
<td>1111</td>
</tr>
</tbody>
</table>

ALUOP[0:3] <= InstReg[9:11] & MASK

(Register Transfer Level language)
Why don’t we write machine code?

Two programs to compute $\sum_{i=0}^{100} i^2$

C version:

```c
int main(void)
{
    int i, sum = 0;
    for (i = 0; i <= 100; ++i)
        sum += i*i;
    return 0;
}
```
Why don’t we write machine code?

MIPS machine code:

0010011110111101111111111111110000
1010111111011111001111001110100
101011111010011111000011100000
101011111010101111000011111110
101011111101000100011100001000
101011111010001000111000010000
100011111010111110000000001100
100011111010111110000000001110
000000001110011111000111111101
....
Assembly Code = symbolic machine code

MIPS assembler program for $\sum_{i=0}^{100} i^2$

```assembly
main:
    li $t0, 0          # i = 0;
    li $t1, 0          # sum = 0;
    li $t2, 100        # if (i>100)
loop:
    bgt $t0, $t2, end  # goto end;
    mul $t3, $t0, $t0  # tmp = i*i;
    add $t1, $t1, $t3  # sum += tmp;
    add $t0, $t0, 1    # ++i;
    j loop             # goto loop;
end:
    ...
```

Assembly code should be viewed as a compromise: to be used only if we really need to work close to the machine e.g. for performance sake, as in OS kernel/device drivers.
When do we write assembly code?

- If we have an application where
  - speed of program is critical \textit{and/or}
  - size of program is critical
we might write part of the program in assembler.

- Try a high-level language first!

- But if the compiler (even with optimisation) doesn’t do a good enough job:
  - find the critical parts of the program
  - re-write them in assembler
Assembler to Machine Code

MIPS assembly code looks like:

```
add $t0, $t1, $t2
```

MIPS machine code looks like:

```
00000001000010010101000000100000
```

Relationship between them:

```
<table>
<thead>
<tr>
<th></th>
<th>$t0</th>
<th>$t1</th>
<th>$t2</th>
<th>unused</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>000000</td>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>00000</td>
</tr>
</tbody>
</table>
```

where $t0$ is register #8, $t1$ is register #9, ...
Execution Cycle

**Instruction Fetch**
- Obtain instruction from program storage

**Instruction Decode**
- Determine required actions and instruction size

**Operand Fetch**
- Locate and obtain operand data

**Execute**
- Compute result value or status

**Result Store**
- Deposit results in storage for later use

**Next Instruction**
- Determine successor instruction
Basic ISA Classes

Accumulator (1 register):

1 address add A acc ← acc + mem[A]
1+x address addx A acc ← acc + mem[A + x]

Stack:

0 address add tos ← tos + next

General Purpose Register:

2 address add A B EA(A) ← EA(A) + EA(B)
3 address add A B C EA(A) ← EA(B) + EA(C)

Load/Store:

3 address add Ra Rb Rc Ra ← Rb + Rc
2 address load Ra Rb Ra ← mem[Rb]
store Ra Rb mem[Rb] ← Ra

Comparison:

Bytes per instruction? Number of Instructions? Cycles per instruction?
Comparing Number of Instructions

Code sequence for $C = A + B$ for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1,A</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1,B</td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store R1, C</td>
<td>Add R3,R1,R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store R3, C</td>
</tr>
</tbody>
</table>
Did you know?

- With \textit{move, cmp, goto, add}, can program anything.

- What’s the minimum number of operations needed?
  - One \ldots \textit{sbn} (subtract and branch if negative)
To be continued…

Some useful references for the material on Computer Systems: