“I have traveled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won't last out the year.“

The editor in charge of business books for Prentice Hall, 1957.
Before We Get Started…

- Last time
  - Wrap up quick overview of C Programming
  - Super quick intro to gdb (debugging tool under Linux)
  - Learn how to login and use Euler, the CPU/GPU cluster

- Today
  - Basic tidbits about how computers are organized and how they work

- Reading Assignment
  - Read first 27 pages of the primer available on the website
Basic Elements Related to the Hardware/Software Interplay
Elements of Processor Architecture
Today’s Computer

- Follows paradigm formalized by Von Neumann in late 1940s
- The von Neumann model:
  - There is no distinction between data and instructions
  - Data and instructions are stored in memory as a string of 0 and 1 bits
    - Instructions are fetched + decoded + executed
    - Data is used to produce results according to rules specified by the instructions
There is a difference between a line of code and a processor instruction. Example:

- Line of C code:
  
  ```c
  a[4] = delta + a[3]; //line of C code
  ```

- MIPS assembly code generated by the compiler:
  ```asm
  lw $t0, 12($s2)  # reg $t0 gets value stored 12 bytes from address in $s2
  add$t0, $s4, $t0 # reg $t0 gets the sum of values stored in $s4 and $t0
  sw $t0, 16($s2) # a[4] gets the sum delta + a[3]
  ```

- Set of three corresponding MIPS instructions produced by the compiler:
  ```text
  10001110010010000000000000001100
  0000001010001000100001000000010000
  10101110010010000000000000010000
  ```
From Code to Instructions

- C code – what you write to implement an algorithm
- Assembly code – what your code gets translated into by the compiler
- Instructions – what the assembly code gets translated into by the compiler

Observations:
- The compiler typically goes from C code directly to machine instructions
- Machine instructions: what you see in an editor like notepad or vim or emacs if you open up an executable file
- There is a one-to-one correspondence between an assembly line of code and an instruction (most of the time)
- Assembly line of code can be regarded as an instruction that is expressed in a way that humans can relatively easy figure out what happens
- Back in the day people wrote assembly code
- Today coding in assembly done only for the super critical parts of a program if you want to optimize and don’t trust the compiler
Instruction Set Architecture (ISA)

- The same line a C code can lead to a different set of instructions on two different computers.

- This is so because two CPUs might draw on two different Instruction Set Architectures (ISA).

- ISA: defines the “language” that expresses at a very low level the actions of a processor.

- Example:
  - Microsoft’s Surface Tablet
    - RT version: uses a Tegra chip, which implements an ARM Instruction Set
    - Pro version: uses an Intel Atom chip, which implements x86 Instruction Set
Example: the same C code leads to different assembly code (and different set of machine instructions, not shown here)

```c
int main(){
    const double fctr = 3.14/180.0;
    double a = 60.0;
    double b = 120.0;
    double c;
    c = fctr*(a + b);
    return 0;
}
```

---

**x86 ISA**

call ___main
fldl LC0
fstpl -40(%ebp)
fldl LC1
fstpl -32(%ebp)
fldl LC2
fstpl -24(%ebp)
faddl -24(%ebp)
fldl LC0
fmulp %st, %st(1)
fstpl -16(%ebp)
movl $0, %eax
addl $36, %esp
popl %ecx
popl %ebp
leal -4(%ecx), %esp
ret

LC0:
 .long 387883269
 .long 1066524452
 .align 8
LC1:
 .long 0
 .long 1078853632
 .align 8
LC2:
 .long 0
 .long 1079902208

---

**MIPS ISA**

main:
 .frame $fp,48,$31 # vars= 32, regs= 1/0, args= 0, gp= 8
 .mask 0x40000000,-4
 .fmask 0x00000000,0
 .set noreorder
 .set nomacro
 addiu $sp,$sp,-48
 sw $fp,44($sp)
mov $f0,$f1
 mul.d $f0,$f2,$f0
 swc1 $f0,32($fp)
 swc1 $f1,36($fp)
mov $2,$0
mov $sp,$fp
lw $fp,44($sp)
addiu $sp,$sp,48
j $31

$LC0:
 .word 3649767765
 .word 1066523892
 .align3
$LC1:
 .word 0
 .word 1078853632
 .align3
$LC2:
 .word 0
 .word 1079902208
 .ident "GCC: (Gentoo 4.6.3 p1.6, pie-0.5.2) 4.6.3"
Instruction Set Architecture vs. Chip Microarchitecture

- ISA – can be regarded as a standard
  - Specifies what a processor should be able to do
    - Load, store, jump on less than, etc.

- Microarchitecture – how the silicon is organized to implement the functionality promised by ISA

- Example:
  - Intel and AMD both use the x86 ISA
  - Nonetheless, they have different microarchitectures
RISC vs. CISC

- **RISC Architecture – Reduced Instruction Set Computing Architecture**
  - Usually each instruction is coded into a set of 32 bits
  - Recently a move to 64 bits
  - Each executable has fixed length instruction be it 32 or 64 (no mixing)
  - The key attribute: the length of the instruction is fixed
  - Promoted by: ARM Holding, company that started as ARM (Advanced RISC Machines)
    - Use in: embedded systems, smart phones – Intel, NVIDIA, Samsung, Qualcomm, Texas Instruments
    - Somewhere between 8 and 10 billion chips based on ARM manufactured annually

- **CISC Architecture – Complex Instruction Set Computing Architecture**
  - Instructions have various lengths
    - Examples: 32 bit instruction followed by 256 bit instruction followed later on by 128 bit instruction, etc.
  - Intel’s X86 is the most common example
  - Promoted by: Intel, AMD
    - Used in: laptops, desktops, workstations, supercomputers
RISC vs. CISC

- RISC is simpler to comprehend, provision for, and work with

- Decoding CISC leads to extra power consumption and makes things more complicated

- A CISC instruction is usually broken down into several micro-operations (uops)

- CISC Architectures invite spaghetti type evolution of the ISA and require complex microarchitecture
  - Provide the freedom to do as you wish
The CPU’s Control Unit (CU)

- Think of a CPU as a big kitchen
  - A work order comes in (this is an instruction)
  - The cook (this is the ALU) starts to cook a meal
  - Some ingredients are needed: meat, spinach, potatoes, etc. (this is the data)
  - Some ready to eat product goes out the kitchen: a soup (this is the result)

- The cook, the passing of meat, passing of pasta, the movement of the sautéed meat to chopping board, boiling of pasta, etc. – they happen in a coordinated fashion (based on a kitchen clock) and is managed by the CU

- The CU manages/coordinates/controls based on information in the work order (the instruction)
The FDX Cycle

- FDX stands for Fetch-Decode-Execute
- This is what the CPU keeps doing to execute a sequence of instructions that combine to make up a program

- Fetch: an instruction is fetched from memory
  - Recall that it will look like this (on 32 bits, MIPS, `lw $t0, 12($s2)`):
  
  ```
  10001110010010000000000000001100
  ```

- Decode: this strings of 1s and 0s are decoded by the CU
  - Example: here’s an “I” (eye) type instruction, made up of four fields

```
<table>
<thead>
<tr>
<th>op (6 bits)</th>
<th>rs (5 bits)</th>
<th>rt (5 bits)</th>
<th>constant or address (16 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

32 bits
Decoding: Instructions Types

- Three types of instructions in MIPS ISA
  - Type I
  - Type R
  - Type J
**Type I (MIPS ISA)**

- The first six bits encode the basic operation; i.e., the opcode, that needs to be completed
  - Example adding two numbers (000000), subtracting two numbers (000001), dividing two numbers (000011), etc.
- The next group of five bits indicates in which register the first operand is stored
- The subsequent group of five bits indicates the register where the second operand is stored.
- Some instructions require an address or some constant offset. This information is stored in the last 16 bits

<table>
<thead>
<tr>
<th>op (6 bits)</th>
<th>rs (5 bits)</th>
<th>rt (5 bits)</th>
<th>constant or address (16 bits)</th>
</tr>
</thead>
</table>
Type R (MIPS ISA)

- Type R has the same first three fields `op`, `rs`, `rt` like I-type

- Packs three additional fields:
  - Five bit `rd` field (register destination)
  - Five bit `shamt` field (shift amount)
  - Six bit `funct` field, which is a function code that further qualifies the opcode
FDX Cycle: The Execution Part
It All Boils Down to Transistors…

- Why are transistors important?

- Transistors can be organized to produce complex logical units that have the ability to execute instructions

- More transistors increase opportunities for building/implementing in silicon functional units that can operate at the same time towards a shared goal
Transistors at Work: AND, OR, NOT

- NOT logical operation is implemented using one transistor
- AND and OR logical ops requires two transistors

Truth tables for AND, OR, and NOT

<table>
<thead>
<tr>
<th>AND</th>
<th>in₁=0</th>
<th>in₂=0</th>
<th>in₁=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>in₂=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>in₂=1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>in₁=0</th>
<th>in₁=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>in₂=0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>in₂=1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOT</th>
<th>in₁=0</th>
<th>in₁=1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

- Design a digital logic block that receives three inputs via three bus wires and produces one signal that is 0 (low voltage) as soon as one of the three input signals is low voltage.
  - In other words, it should return 1 if and only if all three inputs are 1.

<table>
<thead>
<tr>
<th>$i_{in_1}$</th>
<th>$i_{in_2}$</th>
<th>$i_{in_3}$</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Logic Equation:

\[
out = \overline{i_{in_3}} + i_{in_2} \cdot i_{in_1}
\]

- Solution: digital logic block is a combination of AND, OR, and NOT gates.
  - The NOT is represented as a circle O applied to signals moving down the bus.
Example

- Implement a digital circuit that produces the Carry-out digit in a one bit summation operation

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>in₁</td>
<td>in₂</td>
<td>CarryIn</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Logic Equation:

CarryOut = (in₁·CarryIn)+(in₂·CarryIn)+(in₁·in₂)
Integrated Circuits-A One Bit Combo: OR, AND, 1 Bit Adder

- 1 Bit Adder, the Sum part
- Combo: OR, AND, 1 Bit Sum
  - Controlled by the input “Operation”
Integrated Circuits: Ripple Design of 32 Bit Combo

- Combine 32 of the 1 bit combos in an array of logic elements
  - Get one 32 bit unit that can do OR, AND, +
Integrated Circuits: From Transistors to CPU

- Transistor
- Gate
- Mux (selector)
- Complex Combinational Block
- Array of Logic Elements

From simple to complex...

Examples:
- AND, OR, NOT
- $out = in_1 + in_2$
- Example: One bit adder
- Example: 32 bit adder

Requires a "Control Signal" as input
Combine a couple of them
Place instances in an array
Every 18 months, the number of transistors per unit area doubles (Moore’s Law)
- Current technology (2013): feature length is 22 nm
- Next wave (2014): feature length is 14 nm

Example
- NVIDIA Fermi architecture of 2010:
  - 40 nm technology
  - Chips w/ 3 billion transistors → more than 500 scalar processors, 0.5 TFlops
- NVIDIA Fermi architecture 2012:
  - 28 nm technology
  - Chips w/ 7 billion transistors → more than 2000 scalar processors, 1.5 TFlops
Registers
Registers

- Instruction cycle: fetch-decode-execute (FDX)
- CU – responsible for controlling the process that will deliver the request baked into the instruction
- ALU – does the busy work to fulfill the request put forward by the instruction
- The instruction that is being executed should be stored somewhere
- Fulfilling the requests baked into an instruction usually involves handling input values and generates output values
  - This data needs to be stored somewhere
Registers

- Registers, quick facts:
  - A register is an entity whose role is that of storing information
  - A register is the type of storage with shortest latency – it’s closest to the ALU
  - Typically, one cannot control what gets kept in registers (with a few exceptions)

- The number AND size of registers used are specific to a ISA
  - Prime example of how ISA decides on something and the microarchitecture has to do what it takes to implement this design decision

- In MPIS ISA: there are 32 registers of 32 bits that are used to store critical information
Register Types

- Discussion herein covers only several register types typically encountered in a CPU (abbreviation in parenthesis)
  - List not comprehensive, showing only the more important ones

- Instruction register (IR) – a register that holds the instruction that is executed
  - Sometimes known as “current instruction register” CIR

- Program Counter (PC) – a register that holds the address of the next instruction that will be executed
  - NOTE: unlike IR, PC contains an *address* of an instruction, not the actual instruction
Memory Data Register (MDR) – register that holds data that has been read in from main memory or, alternatively, produced by the CPU and waiting to be stored in main memory.

Memory Address Register (MAR) – the address of the memory location in main memory (RAM) where input/output data is supposed to be read in/written out.

- NOTE: unlike MDR, MAR contains an *address* of a location in memory, not actual data.

Return Address (RA) – the address where upon finishing a sequence of instructions, the execution should return and commence with the execution of subsequent instruction.
Registers on previous two slides are a staple in most chip designs.

There are several other registers that are common to many chip designs yet they are encountered in different numbers. Since they come in larger numbers they don’t have an acronym.

- Registers for Subroutine Arguments (4) – a0 through a3
- Registers for temporary variables (10) – t0 through t9
- Registers for saved temporary variables (8) – s0 through s7
  - Saved between function calls
Register Types [Cntd.]

- Several other registers are involved in handling function calls

- Summarized below, but their meaning is only apparent in conjunction with the organization of the virtual memory

  - Global Pointer (gp) – a register that holds an address that points to the middle of a block of memory in the static data segment

  - Stack Pointer (sp) – a register that holds an address that points to the last location on the stack (top of the stack)

  - Frame Pointer (fp) - a register that holds an address that points to the beginning of the procedure frame (for instance, the previous sp before this function changed it’s value)
Register, Departing Thoughts

- Examples:
  - In 32 bit MIPS ISA, there are 32 registers
  - On a GTX580 NVIDIA card there are more than 500,000 32 bit temporary variable registers to keep busy 512 Scalar Processors (SPs) that made up 16 Stream Multiprocessors (SMs)

- Registers are very precious resources

- Increasing their number is not straightforward
  - Need to change the design of the chip (the microarchitecture)
  - Need to work out the control flow
Reading Assignment

- Read the primer document to learn about
  - The FDX (fetch-decode-execute) cycle
  - The bus

- Read first 27 pages of the document
  - Post comments on the forum, there is a discussion thread dedicated to the primer

- URL:
  http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf