“Computer Architecture”?

- The field of Computer Architecture is about the fundamental structure of computer systems
  - What are the components
  - How are they interconnected?
  - How fast does the whole system operate?
  - How much power does it consume?
  - How much does it cost to mass-produce?
  - How to achieve desired speed/power/cost trade-offs?
- The conceptual model for computer architecture, that hasn’t fundamentally changed since 1965: the Von-Neumann architecture
Von-Neumann

- In 1944, John von Neumann joined ENIAC.
- He wrote a memo about computer architecture, formalizing ENIAC ideas.
  - Eckert and Mauchly have pretty much been forgotten (they were in the trenches).
- These ideas became the Von Neumann architecture model.
  - A processor that performs operations and controls all that happens.
  - A memory that contains code and data.
  - I/O of some kind.
Amazingly, it’s still possible to think of the computer this way at a conceptual level (model from ~70 years ago!!!)

But a computer today doesn’t look quite like this
Data Stored in Memory

- All “information” in the computer is in binary form
  - Since Claude Shannon’s M.S. thesis in the 30’s
  - 0: zero voltage, 1: positive voltage (e.g., 5V)
  - bit: the smallest unit of information (0 or 1)

- The basic unit of memory is a byte
  - 1 Byte = 8 bits, e.g., “0101 1101”
  - 1 KiB = $2^{10}$ byte = 1,024 bytes
  - 1 MiB = $2^{10}$ KiB = $2^{20}$ bytes (~ 1 Million)
  - 1 GiB = $2^{10}$ MiB = $2^{30}$ bytes (~ 1 Billion)
  - 1 TiB = $2^{10}$ GiB = $2^{40}$ bytes (~ 1 Trillion)
  - 1 PiB = $2^{10}$ TiB = $2^{50}$ bytes (~ 1000 Trillion)
  - 1 EiB = $2^{10}$ PiB = $2^{60}$ bytes (~ 1 Million Trillion)
  - ...

- Note the “i” in the notations above: means “power of 2”
Data Stored in Memory

- Each byte in memory is labeled by a unique address.
- An address is a number which identifies the memory location of each byte in memory.
  - e.g., the byte at address 3 is 00010010
  - e.g., the byte at address 241 is 10110101
- Typically, we write address in binary as well.
  - e.g., the byte at address 00000011 is 00010010
  - e.g., the byte at address 11110001 is 10110101
- All addresses in the machine have the same number of bits.
  - e.g., 8-bit addresses.
- The processor has instructions that say “Read the byte at address X and give me its value” and “Write some value into the byte at address X.”
## Conceptual View of Memory

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000</td>
<td>0110 1110</td>
</tr>
<tr>
<td>0000 0000 0000 0001</td>
<td>1111 0100</td>
</tr>
<tr>
<td>0000 0000 0000 0010</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0000 0000 0000 0011</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0000 0000 0000 0100</td>
<td>0101 1110</td>
</tr>
<tr>
<td>0000 0000 0000 0101</td>
<td>1010 1101</td>
</tr>
<tr>
<td>0000 0000 0000 0110</td>
<td>0000 0001</td>
</tr>
<tr>
<td>0000 0000 0000 0111</td>
<td>0100 0000</td>
</tr>
<tr>
<td>0000 0000 0000 1000</td>
<td>1111 0101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Conceptual View of Memory

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000</td>
<td>0110 1110</td>
</tr>
<tr>
<td>0000 0000 0000 0001</td>
<td>1111 0100</td>
</tr>
<tr>
<td><strong>0000 0000 0000 0010</strong></td>
<td><strong>0000 0000</strong></td>
</tr>
<tr>
<td>0000 0000 0000 0011</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0000 0000 0000 0100</td>
<td>0101 1110</td>
</tr>
<tr>
<td>0000 0000 0000 0101</td>
<td>1111 0101</td>
</tr>
<tr>
<td>0000 0000 0000 1000</td>
<td>...</td>
</tr>
</tbody>
</table>

At address **0000 0000 0000 0010**, the content is **0000 0000**.
### Conceptual View of Memory

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000</td>
<td>0110 1110</td>
</tr>
<tr>
<td>0000 0000 0000 0001</td>
<td>1111 0100</td>
</tr>
<tr>
<td>0000 0000 0000 0010</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0000 0000 0000 0011</td>
<td>0000 0000</td>
</tr>
<tr>
<td><strong>0000 0000 0000 0100</strong></td>
<td><strong>0101 1110</strong></td>
</tr>
<tr>
<td>0000 0000 0000 0110</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0000 0000 0000 0111</td>
<td>0100 0000</td>
</tr>
<tr>
<td>0000 0000 0000 1000</td>
<td>1111 0101</td>
</tr>
</tbody>
</table>

**At address 0000 0000 0000 0100, the content is 0101 1110**
Both Code and Data in Memory

- Once a program is loaded in memory, its address space contains both code and data.
- To the CPU those are not really different, but the programmer knows which bytes are data and which are code.
  - Always conveniently hidden from you if you’ve never written assembly.
  - But we’ll have to keep code/data straight in these lecture notes.

Example Address Space

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1100</td>
<td>0110 1011</td>
</tr>
<tr>
<td>0000 1101</td>
<td>1111 0010</td>
</tr>
<tr>
<td>0000 1110</td>
<td>0010 0001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1000 0000</td>
<td>1111 0000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111 0010</td>
<td>0101 1111</td>
</tr>
</tbody>
</table>

Memory
We need a CPU

- So now we have a memory in which we can store/retrieve bytes at precise location
- These bytes presumably have some useful meaning to us
  - e.g., integers, ASCII codes of characters, floating points numbers, RGB values
  - e.g., instructions that specify what to do with the data; when you buy a processor, the vendor defines the instruction set (e.g., instruction “0010 1101” means “increment some useful counter”)
- The CPU is the piece of hardware that modifies the content of memory
  - In fact, one can really think of the CPU as a device that takes use from on memory state (i.e, all the stored content) to another memory state (some new, desired stored content)
What's in the CPU?

CPU

Memory

I/O System
What’s in the CPU?

- Memory
- I/O
- System
- Control Unit
- ALU
- Program counter
- Current instruction
- Registers
- Current instruction
- System
- Memory
What’s in the CPU?

Registers: the “variables” that hardware instructions work with

Data can be loaded from memory into a register
Data can be stored from a register back into memory
Operands and results of computations are in registers
Accessing a register is really fast
There is a limited number of registers
What's in the CPU?

Arithmetic and Logic Unit: what you do computation with

used to compute a value based on current register values and store the result back into a register

+, *, /, -, OR, AND, XOR, etc.
What’s in the CPU?

**Program Counter**: Points to the next instruction

Special register that contains the address in memory of the next instruction that should be executed (gets incremented after each instruction, or can be set to whatever value whenever there is a change of control flow)
What’s in the CPU?

Current Instruction: Holds the instruction that’s currently being executed
What’s in the CPU?

**Control Unit:** Decodes instructions and make them happen

Logic hardware that decodes instructions (i.e., based on their bits) and sends the appropriate (electrical) signals to hardware components in the CPU.
The CPU in its “Glory”
Fetch-Decode-Execute Cycle

- The Fetch-Decode-Execute cycle
  - The control unit fetches the next program instruction from memory
    - Using the program counter to figure out where that instruction is located in the memory
  - The instruction is decoded and signals are send to hardware components
    - e.g., is the instruction loading something from memory? is it adding two register values together?
  - Operands are fetched from memory and put in registers, if needed
  - The ALU executes computation, if any, and store results in the registers
  - Register values are stored back to memory, if needed
  - Repeat

- Computers today implement MANY variations on this model
- But one can still program with the above model in mind
  - but certainly without (fully) understanding performance issues
## Fetch-Decode-Execute

### Diagram Description
- **Program Counter**: Holds the address of the current instruction.
- **Current Instruction**: Contains the address of the instruction to be executed.
- **ALU**: Performs arithmetic and logical operations.
- **Control Unit**: Controls the flow of data and operations.

### Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1100</td>
<td>0110 1011</td>
</tr>
<tr>
<td>0000 1101</td>
<td>1111 0010</td>
</tr>
<tr>
<td>0000 1110</td>
<td>0010 0001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111 0010</td>
<td>0101 1111</td>
</tr>
</tbody>
</table>

### Control Unit

- **Register**: Temporary storage for data and instructions.
- **Program Counter**: Current address of instruction to be executed.
- **Current Instruction**: Instruction to be fetched.

### ALU

- Performs operations based on the current instruction.
Somehow, the program counter is initialized to some content, which is an address (we’ll see how that happens much later)
Fetch the content (instruction) at address 0000 1100, which is “0110 1011”, and store it in the “current instruction” register.
### Fetch-Decode-Execute

**Diagram:**
- **ALU**
- **Control Unit**
- **Program Counter:** 0000 1101
- **Current Instruction:** 0110 1011
- **Register:**
  - Content: 0000 1100
  - Value: 0110 1011
  - Content: 0000 1101
  - Value: 1111 0010
  - Content: 0000 1110
  - Value: 0010 0001
  - Content: 1000 0000
  - Value: 1111 0000
  - Content: 1111 0010
  - Value: 0101 1111

**Address**
- 0000 1100
- 0000 1101
- 0000 1110
- 1000 0000
- 1111 0010
- 1111 0010

**Value**
- 0110 1011
- 1111 0010
- 0010 0001
- 1111 0000
- 0101 1111

**Memory**

**Increment the program counter**
Decode instruction “0110 1011”. Say it means: “Load the value at address 1000 0000 and store it in the second register”
Fetch-Decode-Execute

Send signals to all hardware components to execute the instruction: load the value at address 1000 0000, which is "1111 0000" and store it in the second register.
Fetch the content (instruction) at address 0000 1101, which is “1111 0010”, and store it in the “current instruction” register.
Fetch-Decode-Execute

- Program counter: 0000 1110
- Current instruction: 1111 0010
- Register: 0000 1100
- Memory:
  - Address: 0000 1100, Value: 0110 1011
  - Address: 0000 1101, Value: 1111 0010
  - Address: 0000 1110, Value: 0010 0001
  - Address: 1000 0000, Value: 1111 0000
  - Address: 1111 0010, Value: 0101 1111

- ALU
- Control Unit
- Increment the program counter
Decode instruction “1111 0010”. Say it means: “Do a logical NOT on the second register”

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1100</td>
<td>0110 1011</td>
</tr>
<tr>
<td>0000 1101</td>
<td>1111 0010</td>
</tr>
<tr>
<td>0000 1110</td>
<td>0010 0001</td>
</tr>
<tr>
<td>1000 0000</td>
<td>1111 0000</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>1111 0010</td>
<td>0101 1111</td>
</tr>
</tbody>
</table>

Memory
Send signals to all hardware components to execute the instruction: do a logical NOT on the second register.
Fetch the content (instruction) at address 0000 1110, which is “0010 0001”, and store it in the “current instruction” register.
Fetch-Decide-Execute

Increment the program counter
Fetch-Decode-Execute

Decode instruction “0010 0001”. Say it means: “Store the value in the second register to memory at address 1111 0010”
Send signals to all hardware components to execute the instruction: store the value in the second register, which is 0000 1111, to memory at address 1111 0010.
Fetch-Decode-Execute

- This is only a simplified view of the way things work
- The “control unit” is not a single thing
  - Control and data paths are implemented by several complex hardware components
- There are multiple ALUs, there are caches, there are multiple CPUs in fact (“cores”)
- Execution is pipelined: e.g., while one instruction is fetched, another is executed
- Decades of computer architecture research have gone into improving performance, thus often leading to staggering hardware complexity
  - Doing smart things in hardware requires more logic gates and wires, thus increasing processor cost
- But conceptually, fetch-decode-execute is it
In-Class Exercise

With the following instruction set definition and machine state, what is the new memory state after execution completes?

<table>
<thead>
<tr>
<th>code</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0000</td>
<td>Increment the register</td>
</tr>
<tr>
<td>1111 0010</td>
<td>Decrement the register</td>
</tr>
<tr>
<td>0101 1111</td>
<td>Save register to address not(register)</td>
</tr>
</tbody>
</table>

![Diagram showing program counter, register, ALU, Control Unit, and memory state transitions]
- Fetch the instruction: “1111 0000”
- Execute it: increment register to value “1111 0011”
- Fetch the next instruction: “1111 0001”
- Execute it: save value “1111 0011” to address “0000 1100”
- Fetch the instruction: “1111 0000”
- Execute it: increment register to value “1111 0011”

<table>
<thead>
<tr>
<th>code</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0000</td>
<td>Increment the register</td>
</tr>
<tr>
<td>1111 0010</td>
<td>Decrement the register</td>
</tr>
<tr>
<td>0101 1111</td>
<td>Save register to address not(register)</td>
</tr>
</tbody>
</table>

**Diagrm:**
- Program counter: 1000 0001
- Register: 1111 0011
- ALU
- Control Unit
- Instructions (code): 0000 1100, 1111 0010, 0000 1101, 0010 0001, ..., 0000 0000, 1111 0000, 1000 0001, 0101 1111
- Fetch the instruction: “1111 0000”
- Execute it: increment register to value “1111 0011”
- Fetch the next instruction: “0101 1111”

### Code and Operation Table

<table>
<thead>
<tr>
<th>code</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0000</td>
<td>Increment the register</td>
</tr>
<tr>
<td>1111 0010</td>
<td>Decrement the register</td>
</tr>
<tr>
<td>0101 1111</td>
<td>Save register to address not(register)</td>
</tr>
</tbody>
</table>

---

![Diagram showing program counter, register, ALU, and Control Unit with code and operation values]
- Fetch the instruction: “1111 0000”
- Execute it: increment register to value “1111 0011”
- Fetch the next instruction: “0101 1111”
- Execute it: save value “1111 0011” to address “0000 1100”

<table>
<thead>
<tr>
<th>code</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0000</td>
<td>Increment the register</td>
</tr>
<tr>
<td>1111 0010</td>
<td>Decrement the register</td>
</tr>
<tr>
<td>0101 1111</td>
<td>Save register to address not(register)</td>
</tr>
</tbody>
</table>

Diagram:

- Program counter
- Register
- ALU
- Control Unit
- Instruction memory

- Instruction Memory:
  - 0000 1100
  - 0000 1101
  - 0000 0000
  - 1111 0011
  - 0101 1111
  - ...
The Clock

- Every computer maintains an internal clock that regulates how quickly instructions can be executed, and is used to synchronize system components
  - Just like a metronome
- In the previous example, each “event” happens at a different “tick” of the clock
- The frequency of the clock is called the *clock rate*
- The time in between two clock ticks is called a clock cycle or *cycle* for short
- Clock cycle = 1 / Clock Rate
  - Clock rate = 2.4 GHz
  - Clock cycle = 1 / (2.4*1000*1000*1000)
    - = 0.416 e⁻⁹ sec
    - = 0.416 ns (nanosec)
Faster/slower Clock Rate

- The higher the clock rate, the shorter the clock cycle
- It’s tempting to think that a faster clock rate means a faster computer
- But it all depends on what amount of work is done in a clock cycle!
  - Computer A: clock rate of 2GHz and a multiplication requires 10 cycles
  - Computer B: clock rate of 1.5GHz and a multiplication requires 5 cycles
  - Computer B is faster than Computer A to run a program that performs a lot of multiplications
- Therefore, clock rates should not be used to compare computers in different families
  - A 1.4GHz Pentium 4 is most likely slower than a 1.5GHz Pentium 4
  - A 2.4GHz Pentium 4 may be slower than a 2.0GHz AMD Athlon64
- Furthermore, comparisons depend on the type of applications
  - Computer A faster than Computer B for some applications
  - Computer B faster than Computer A for some others
Instructions

- Instructions are encoded in binary machine code
  - e.g.: 01000110101101 may mean “perform an addition of two registers and store the results in another register”

- The CPU is built using gates (OR, AND, etc.) which themselves use transistors
  - See ICS331

- These gates implement instruction decoding
  - Based on the bits of the instruction code, several signals are sent to different electronic components, which in turn perform useful tasks

- Typically, an instruction consists of two parts
  - The opcode: what the instruction computes
  - The operands: the input to the computation

<table>
<thead>
<tr>
<th>opcode</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>11010110101101</td>
</tr>
</tbody>
</table>
Instruction Set Architecture (ISA)

- When designing a CPU, one must define the set of all the instructions it understands
  - This is one thing that Intel engineers do
- This is called the ISA: Instruction Set Architecture
- Typical ISA include instructions for
  - Performing arithmetic operations on register values
  - Load values from memory into registers
  - Store values from registers into memory
  - Test register values to decide what instruction to execute next
  - ...
- Envision a loooong specification manual that lists all the possible instructions...
### ISA specification Example: x86

Let's look at the Web site [http://ref.x86asm.net/](http://ref.x86asm.net/)

<table>
<thead>
<tr>
<th>op</th>
<th>mnemonic</th>
<th>operands</th>
<th>description, notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ADD</td>
<td>r/m8</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>ADD</td>
<td>r/m16/32</td>
<td>Add</td>
</tr>
<tr>
<td>02</td>
<td>ADD</td>
<td>r/m8</td>
<td>Add</td>
</tr>
<tr>
<td>03</td>
<td>ADD</td>
<td>r/m16/32</td>
<td>Add</td>
</tr>
<tr>
<td>04</td>
<td>ADD</td>
<td>AL</td>
<td>Add</td>
</tr>
<tr>
<td>05</td>
<td>ADD</td>
<td>eAX</td>
<td>Add</td>
</tr>
<tr>
<td>06</td>
<td>PUSH</td>
<td>E8</td>
<td>Push Word, Doubleword or Quadword onto the Stack</td>
</tr>
<tr>
<td>07</td>
<td>POP</td>
<td>E8</td>
<td>Pop a Value from the Stack</td>
</tr>
<tr>
<td>08</td>
<td>L OR</td>
<td>r/m8</td>
<td>Logical Inclusive OR</td>
</tr>
<tr>
<td>09</td>
<td>L OR</td>
<td>r/m16/32</td>
<td>Logical Inclusive OR</td>
</tr>
<tr>
<td>0A</td>
<td>OR</td>
<td>r/m8</td>
<td>Logical Inclusive OR</td>
</tr>
<tr>
<td>0B</td>
<td>OR</td>
<td>r/m16/32</td>
<td>Logical Inclusive OR</td>
</tr>
<tr>
<td>0C</td>
<td>OR</td>
<td>AL</td>
<td>Logical Inclusive OR</td>
</tr>
<tr>
<td>0D</td>
<td>OR</td>
<td>eAX</td>
<td>Logical Inclusive OR</td>
</tr>
<tr>
<td>0E</td>
<td>PUSH</td>
<td>CS</td>
<td>Push Word, Doubleword or Quadword onto the Stack</td>
</tr>
</tbody>
</table>
Assembly language

- It’s really difficult for humans to read/remember binary instruction encodings
  - But people used to do it!
  - One would typically use hexadecimal encoding, but still it seems impossible to do in today’s world
- Therefore it is typical to use a set of mnemonics, which form the assembly language
  - It is often said that the CPU understands assembly language
  - This is not technically true, as the CPU understand machine code, which we, as humans, choose to represent using assembly language
- An assembler transforms assembly code into machine code (i.e., from a human readable format into a binary format that the CPU understands)
Assembly Language

- It used to be that *all* computer programmers did all day was to write assembly code
- This was difficult for many reasons
  - Difficult to read
  - Very difficult to debug
  - Different from one computer to another!
- The use of assembly language for all programming prevented the (sustainable) development of large software project involving many programmers
- This is the main motivation for the development of high-level languages
  - FORTRAN, Cobol, C, etc.
High-level Languages

- The first successful high-level language was FORTRAN
  - Developed by IBM in 1954 to run on their 704 series
  - Used for scientific computing
- The introduction of FORTRAN led people to believe that there would never be bugs again because it made programming so easy!
  - But high-level languages led to larger and more complex software systems, hence leading to bugs
- Another early programming language was COBOL
  - Developed in 1960, strongly supported by DoD
  - Used for business applications
- In the early 60s IBM had a simple marketing strategy
  - On the IBM 7090 you used FORTRAN to do science
  - On the IBM 7080 you used COBOL to do business
- Many high-level languages have been developed since then, and they are what most programmers use
  - Fascinating history (see ICS 313)
High-Level Languages

- Having high-level languages is good, but CPUs do not understand them
  - As we saw, they only understand very basic instructions to manipulate registers, etc.
- Therefore, there needs to be a translation from a high-level language to machine code
- The translation is done by a compiler
- Let’s see this on a picture....
char *tmpfilename;
int num_schedulers=0;
int num_request_submitters=0;
i,j;
if (!(f = fopen(filename,"r"))) {
    xbt_assert1(0,"Cannot open file %s",filename);
}
while(fgets(buffer,256,f)) {
    if (!strncmp(buffer,"SCHEDULER",9))
        num_schedulers++;
    if (!strncmp(buffer,"REQUESTSUBMITTER",16))
        num_request_submitters++;
} 
fclose(f);
tmpfilename = strdup("/tmp/jobsimulator_"
The Big (Simplified) Picture

High-level code

```
char *tmpfilename;
int num_schedulers=0;
int num_request_submitters=0;
int i,j;
if (!(f = fopen(filename,"r"))) {
    xbt_assert1(0,"Cannot open file \"%s\",filename);
}
while(fgets(buffer,256,f)) {
    if (!strncmp(buffer,"SCHEDULER",9))
        num_schedulers ++;
    if (!strncmp(buffer,"REQUESTSUBMITTER",16))
        num_request_submitters ++;
}
fclose(f);
tmpfilename = strdup("/tmp/jobsimulator_
```

Assembly code

```
   sll $t3, $t1, 2
   add $t3, $s0, $t3
   sll $t4, $t0, 2
   add $t4, $s0, $t4
   lw $t5, 0($t3)
   lw $t6, 0($t4)
   slt $t2, $t5, $t6
   beq $t2, $zero, endif
   add $t0, $t1, $zero
   sll $t4, $t0, 2
   add $t4, $s0, $t4
   lw $t5, 0($t3)
   lw $t6, 0($t4)
   slt $t2, $t5, $t6
   beq $t2, $zero, endif
```

Machine code

```
010001010101110110
1010101110101001
1010101110101001
1111000110101001
00011011011111101
01000000001000100
0000100010000111
10100101010101011
0001010101001001
0101010101010101
10101011111010101
1010101101010101
11110001101010100
1
```

COMPILER

ASSEMBLER

CPU
This course’s topics:

High-level code

```c
char *tmpfilename;
int numSchedulers=0;
int num_request_submitters=0;
int i,j;
if (!(f = fopen(filename,"r"))) {
  xbt_assert1(0,"Cannot open file %s",filename);
}
while(fgets(buffer,256,f)) {
  if (!strncmp(buffer,"SCHEDULER",9))
    num_schedulers++;
  if (!strncmp(buffer,"REQUESTSUBMITTER",16))
    num_request_submitters++;
}
fclose(f);
tmpfilename = strdup("/tmp/jobsimulator_");
```

Hand-written Assembly code

```assembly
sll $t3, $t1, 2
add $t3, $s0, $t3
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
add $t0, $t1, $zero
sll $t4, $t0, 2
add $t4, $s0, $t4
lw  $t5, 0($t3)
lw  $t6, 0($t4)
slt $t2, $t5, $t6
beq $t2, $zero, endif
```

ASSEMBLER

Assembly code

Machine code

```
010000101010110110
101010101111010101
101001010101010001
111100001010110101
000101010111101111
010000000100000100
000100100100100011
101001010101010111
000101010101010101
010101010101010111
101010111110101101
1010101010101010101
```
What we do in this course

First part of the semester (bulk of the course)
- Learn how to write assembly code
  - For the x86 architecture
- Learn how to use an assembler and a compiler to run our assembly code

Second part of the semester (shorter)
- Learn about systems tools
  - loader, linker, compiler, debugger, etc.
Why should we learn all this?

- **Why should we learn how to write assembly code?**
  - Students: “We won’t write assembly code for a living!”
- **Reason #1:** Many of you will have to write *some* assembly
  - Write small piece of assembly for performance optimization as part of larger software projects
  - Write assembly code for embedded devices
- **Reason #2:** Learning assembly makes you a better programmer in high-level languages
  - Makes you keenly aware of what happens under the cover, which allows for easier debugging
  - Makes you understand “performance bugs”
  - Allows you to write more efficient high-level code
  - Allows you to read generated assembly to better understand what’s going on
Why should we learn all this?

- Why should we learn how compilers work?
  - Students: “We won’t develop compilers for a living!”

- Reason #1: Many of you will develop “some” compilers
  - Some of you may develop a compiler for a programming language
  - But often one has to write “compiler” for things that one doesn’t always think of as programming languages
    - E.g., configuration files for large software systems

- Reason #2: Knowing how a compiler works makes you a better programmer
  - You know understand the connection between high-level code and generated assembly code (see previous slide)
  - You understand what some high-level language constructs really entail under the cover, and thus understand their performance implications
Why should we learn all this?

- Meta-reason: this course should go a long way in giving you a **holistic understanding** of how a program goes from just a text file to a running code
  - You should be able to describe in low-level details how you go from “I wrote a piece of C code that calls a function that adds 2 and 2 together and prints the result” to “the computer prints 4”
  - The complexity of such a simple thing is actually quite stunning, and we’ll take a simplified view
  - There should be something satisfying in knowing how things work from top to bottom!

- This “holistic understanding” should be acquired with ICS312, ICS331, ICS431/EE461, and ICS332
Conclusion

- If you want to know more
  - Take a computer architecture course