Computer Architecture?

- Instruction Set Architecture
- Processor Organization
- VLSI Implementations

Software layers:
- Operating systems
- Compiler
- Other applications, e.g., games

“Application pull”

“Technology push”

Semiconductor technologies

“Architecture”

“Microarchitecture”

“Physical hardware”
High level system architecture

Memory stores:
- Program code
- Data (static or dynamic)
- Execution stack

Important registers:
- Program counter (PC)
- Instruction register (IR)
- General purpose registers

Chapter 1 Review: Computer Performance

- **Response Time (latency)**
  - How long does it take for my job to run?

- **Throughput**
  - How much work is getting done?

- **CPU execution time (our focus)**
  - doesn’t count I/O or time spent running other programs

- **Performance = 1 / Execution time**

- **Important architecture evaluation metrics**
  - Last century: TIME, TIME, TIME, COST
  - Now: TIME, COST, POWER, TEMPERATURE
  - In this course, we will mainly talk about time
Clock Cycles (from Section 1.6)

- Instead of using execution time in seconds, we can use number of cycles
  - Clock “ticks” indicate when to start activities:

```
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+
  time
```

- cycle time = time between ticks = seconds per cycle

- clock rate (frequency) = cycles per second (1 Hz. = 1 cycle/sec)
  
  EX: A 2 GHz. clock has a cycle time of \( \frac{1}{2 \times 10^9} = 0.5 \times 10^{-9} \) seconds.

- Time per program = cycles per program \( \times \) time per cycle

\[
\text{seconds} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}} = \frac{\text{cycles}}{\text{Instruction}} \times \frac{\text{Instructions}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}
\]

Non-pipelined CPUs

- Different instructions may take different number of cycles to execute (Why?)

- Cycles per instruction (CPI):
  Average number of cycles for executing an instruction (depends instruction mix)

Example: 20%, 30% and 50% of the instructions in a program are of type A, B and C, respectively. The execution of a type A, B and C instruction needs 1, 2 and 3 cycles, respectively.

What is the CPI for the program?
Machine performance

- Suppose we have two implementations of the same instruction set architecture (ISA).

For some program,

Machine A has a clock cycle time of 10 ns. and a CPI of 2.0
Machine B has a clock cycle time of 20 ns. and a CPI of 1.2

What machine is faster for this program?

**Execution time** = \( \text{CPI} \times \# \text{ of instructions} \times \text{Cycle time} \)

**Definitions:**
- \# of instructions per cycle (IPC) = inverse of the CPI
- MIPS = \# of million instructions per second

MIPS ISA (from chapter 2)

**MIPS assumes 32 CPU registers (\$0, \ldots, \$31)**

- All arithmetic instructions have 3 operands
- Operand order is fixed (destination first in assembly instruction)
- Operands of arithmetic operations are in registers
- Simple memory addressing mechanism

C code:
\[
A = B + C + D;  \\
E = F - A;
\]

MIPS code:
\[
\text{add } \$t0, \$s1, \$s2  \\
\text{add } \$s0, \$t0, \$s3  \\
\text{sub } \$s4, \$s5, \$s0
\]

\( t0, s1, s2, \ldots \) are symbolic names of registers (translated to the corresponding numbers by the assembler).
### Register usage Conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved (correspond to program variables)</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

Note: registers 26 and 27 are reserved for kernel use.

### Memory Organization

- Viewed as a large, single-dimension array of bytes.
- A memory address is an index into the array.
- "Byte addressing" means that the index points to a byte of memory.
- A word in MIPS is 32 bits long or 4 bytes (will not consider 64-bit versions)

\[
\begin{array}{cccc}
\ldots000000 &=& 0 & \text{8 bits of data} \\
\ldots000001 &=& 1 & \text{8 bits of data} \\
\ldots000010 &=& 2 & \text{8 bits of data} \\
\ldots000011 &=& 3 & \text{8 bits of data} \\
\ldots000100 &=& 4 & \text{8 bits of data} \\
\ldots000101 &=& 5 & \text{8 bits of data} \\
\ldots000110 &=& 6 & \text{8 bits of data} \\
\ldots000111 &=& 7 & \text{8 bits of data} \\
\ldots001000 &=& 8 & \text{8 bits of data} \\
\ldots001001 &=& 9 & \text{8 bits of data} \\
\ldots001010 &=& 10 & \text{8 bits of data} \\
\ldots001011 &=& 11 & \text{8 bits of data} \\
\ldots001100 &=& 12 & \text{8 bits of data} \\
\ldots001101 &=& 13 & \text{8 bits of data} \\
\ldots001110 &=& 14 & \text{8 bits of data} \\
\ldots001111 &=& 15 & \text{8 bits of data} \\
\ldots010000 &=& 16 & \text{8 bits of data} \\
\ldots010001 &=& 17 & \text{8 bits of data} \\
\ldots010010 &=& 18 & \text{8 bits of data} \\
\ldots010011 &=& 19 & \text{8 bits of data} \\
\ldots010100 &=& 20 & \text{8 bits of data} \\
\ldots010101 &=& 21 & \text{8 bits of data} \\
\ldots010110 &=& 22 & \text{8 bits of data} \\
\ldots010111 &=& 23 & \text{8 bits of data} \\
\ldots011000 &=& 24 & \text{8 bits of data} \\
\ldots011001 &=& 25 & \text{8 bits of data} \\
\ldots011010 &=& 26 & \text{8 bits of data} \\
\ldots011011 &=& 27 & \text{8 bits of data} \\
\ldots011100 &=& 28 & \text{32 bits of data} \\
\ldots011101 &=& 29 & \text{32 bits of data} \\
\ldots011110 &=& 30 & \text{32 bits of data} \\
\ldots011111 &=& 31 & \text{32 bits of data} \\
\ldots100000 &=& 32 & \text{32 bits of data} \\
\ldots100001 &=& 33 & \text{32 bits of data} \\
\ldots100010 &=& 34 & \text{32 bits of data} \\
\ldots100011 &=& 35 & \text{32 bits of data} \\
\ldots100100 &=& 36 & \text{32 bits of data} \\
\ldots100101 &=& 37 & \text{32 bits of data} \\
\ldots100110 &=& 38 & \text{32 bits of data} \\
\ldots100111 &=& 39 & \text{32 bits of data} \\
\ldots101000 &=& 40 & \text{32 bits of data} \\
\ldots101001 &=& 41 & \text{32 bits of data} \\
\ldots101010 &=& 42 & \text{32 bits of data} \\
\ldots101011 &=& 43 & \text{32 bits of data} \\
\ldots101100 &=& 44 & \text{32 bits of data} \\
\ldots101101 &=& 45 & \text{32 bits of data} \\
\ldots101110 &=& 46 & \text{32 bits of data} \\
\ldots101111 &=& 47 & \text{32 bits of data} \\
\ldots110000 &=& 48 & \text{32 bits of data} \\
\ldots110001 &=& 49 & \text{32 bits of data} \\
\ldots110010 &=& 50 & \text{32 bits of data} \\
\ldots110011 &=& 51 & \text{32 bits of data} \\
\ldots110100 &=& 52 & \text{32 bits of data} \\
\ldots110101 &=& 53 & \text{32 bits of data} \\
\ldots110110 &=& 54 & \text{32 bits of data} \\
\ldots110111 &=& 55 & \text{32 bits of data} \\
\ldots111000 &=& 56 & \text{32 bits of data} \\
\ldots111001 &=& 57 & \text{32 bits of data} \\
\ldots111010 &=& 58 & \text{32 bits of data} \\
\ldots111011 &=& 59 & \text{32 bits of data} \\
\ldots111100 &=& 60 & \text{32 bits of data} \\
\ldots111101 &=& 61 & \text{32 bits of data} \\
\ldots111110 &=& 62 & \text{32 bits of data} \\
\ldots111111 &=& 63 & \text{32 bits of data} \\
\end{array}
\]

- Address space is \(2^{32}\) bytes with byte addresses from 0 to \(2^{232}-1\)
- \(2^{32}\) words – the address of a word is the address of its first byte
  - the least 2 significant bits of a word address are 00
Load and Store Instructions

- A memory address = content of a register + an immediate constant


MIPS code:
\[
\begin{align*}
\text{lw} & \quad \$t0, \enspace 32($s3) \quad /\!\!/ \text{load word} \\
\text{add} & \quad \$t0, \enspace \$s2, \enspace \$t0 \\
\text{sw} & \quad \$t0, \enspace 32($s3) \quad /\!\!/ \text{store word}
\end{align*}
\]

- Can you modify the above example to add \( h \) to all the elements \( A[0], \ldots, A[8] \)?
- Need additional types of instructions?

Control instructions

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:
  \[
  \begin{align*}
  \text{beq} & \quad \$t0, \enspace \$t1, \text{label} \quad /\!\!/ \text{branch if } \$t0 \text{ and } \$t1 \text{ contain identical data} \\
  \text{bne} & \quad \$t0, \enspace \$t1, \text{label} \quad /\!\!/ \text{branch if } \$t0 \text{ and } \$t1 \text{ contain different data}
\end{align*}
\]
  - one may assign a label to any instruction in assembly language

- MIPS unconditional branch instruction:
  \[ j \quad \text{label} \]


To summarize (short version of Figure 2.1):

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>$s1, s2, s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands, data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>$s1, s2, s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands, data in registers</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>$s1, s2, 100</td>
<td>$s1 = $s2 + 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw</td>
<td>$s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>lb</td>
<td>$s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td></td>
<td>lw</td>
<td>$s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td></td>
<td>lb</td>
<td>$s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>$s1, 100</td>
<td>$s1 = 100 * 2^31</td>
<td>Loads constant in upper 31-bits</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>beq</td>
<td>$s1, $s2, 25</td>
<td>If ($s1 == $s2) go to PC + 4 + 100</td>
<td>Equal test, PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>$s1, $s2, 25</td>
<td>If ($s1 != $s2) go to PC + 4 + 100</td>
<td>Not equal test, PC-relative</td>
</tr>
<tr>
<td></td>
<td>slt</td>
<td>$s1, $s2, $s3</td>
<td>If ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than, for switch, base</td>
</tr>
<tr>
<td></td>
<td>slti</td>
<td>$s1, $s2, 100</td>
<td>If ($s2 &lt; 100) $s1 = 1; else $s1 = 0</td>
<td>Compare less than constant</td>
</tr>
<tr>
<td></td>
<td>bge</td>
<td>$s1, $s2, 25</td>
<td>If ($s1 + 100) $s1 = 1; else $s1 = 0</td>
<td>Compare greater than or equal, PC-relative</td>
</tr>
<tr>
<td></td>
<td>bgt</td>
<td>$s1, $s2, 25</td>
<td>If ($s1 + 100) $s1 = 1; else $s1 = 0</td>
<td>Compare greater than, PC-relative</td>
</tr>
<tr>
<td></td>
<td>bneq</td>
<td>$s1, $s2, 25</td>
<td>If ($s1 != $s2) $s1 = 1; else $s1 = 0</td>
<td>Compare not equal, PC-relative</td>
</tr>
<tr>
<td></td>
<td>jmp</td>
<td>2500</td>
<td>go to 10000 in current segment</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>j</td>
<td>14</td>
<td>go to 2514</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jal</td>
<td>2500, $ra</td>
<td>go to 10000; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>

- Instruction operands:
  - Some instructions need 3 registers
  - Some instructions need 2 registers and a constant
  - Some instructions need one register
  - Some instructions need a constant
Machine Language instructions

- Instructions, like registers and data words, are also 32 bits long
- **R-type instruction format:**
  - Example: `add $t0, $s1, $s2` → `add $8, $17, $18`

  \[ \begin{array}{cccccc}
  0 & 17 & 18 & 8 & 0 & 32 \\
  \end{array} \]
  
  \[ \begin{array}{ccccccc}
  000000 & 10001 & 10010 & 01000 & 00000 & 100000 \\
  \end{array} \]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

- **I-type instruction format:**
  - Example: `lw $t0, 36($s2)`

  \[ \begin{array}{cccccc}
  100011 & 10010 & 01000 & 0000000000100100 \\
  \end{array} \]
  
<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16-bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>18</td>
<td>8</td>
<td>36</td>
</tr>
</tbody>
</table>

The address stored in the 16-bit address of a branch instruction is the address of the target instruction relative to the next instruction (offset).

**Branch instructions (I-type)**

`beq $4, $5, Label` ===> put target address in PC if $4 == $5

The address stored in the 16-bit address of a branch instruction is the address of the target instruction relative to the next instruction (offset).

**Diagram:**

```
L1: beq $4, $5, L2
  | sub ... | ...010000
  | 4 instr. | ...010010
  |        | ...
  |        | ...
L2: add ...
  | memory address | ...
```

35 instructions = (100)₂ words = (10000)₂ bytes
Jump instructions

- In Jump instructions, address is relative to the 4 high order bits of PC+4
  - Memory is logically divided into 16 segments of 256 MB each.
  - address in Jump instruction = offset from the start of the current segment

![Diagram of Jump instructions]

Overview of MIPS

- only three instruction formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>op, rs, rt, rd, shamt, funct</td>
</tr>
<tr>
<td>I</td>
<td>op, rs, rt, 16 bit constant/offset</td>
</tr>
<tr>
<td>J</td>
<td>op, 26 bit relative address</td>
</tr>
</tbody>
</table>

- Arithmetic and logic instructions use the **R-type** format
- Memory and branch instructions (lw, sw, beq, bne) use the **I-type** format
- Jump instructions use the **J-type** format

- Refer to section A.10 for complete specifications of MIPS instructions
MIPS Addressing modes (operands’ specification)

1. Immediate addressing
   
2. Register addressing
   
3. Base addressing
   
4. PC-relative addressing
   
5. Pseudodirect addressing

x86 Addressing Modes (A contrast)

- 12 addressing modes available to compute the effective address
  - Immediate (operand is in instruction – as in MIPS)
  - Register operand (operand in register – as in MIPS)
  - Displacement (the address of the operand is in the instruction
    - similar to the jump instruction in MIPS (without the 4 PC bits)
  - Relative (same as PC-relative in MIPS)
  - Base + displacement (address = content of a base register +
    displacement )
    - similar to base addressing in MIPS
  - Base (same as above with displacement = 0)
  - Scaled index with displacement (use an index rather than a base register)
  - Base with index and displacement (use two registers; base and index)
  - Base scaled index with displacement (multiply the index by a scale)
- Actual address (linear address) = effective address + base address
  (the beginning of the current segment).
- Instructions other than load/store can access memory
x86 Address Calculation

Displacement (16 bits)

MIPS addressing

Segmenting:

Effective Address
Linear Address

x86 Instruction Format (variable length instructions)

<table>
<thead>
<tr>
<th>Instruction prefixes</th>
<th>Opcode</th>
<th>Mod/R/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 or 1</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>0, 1, 2, or 4</td>
<td>0, 1, 2, or 4</td>
</tr>
</tbody>
</table>

Mod 7 6 5 4 3 2 1 0
Reg Opcode 6 5 4
R/M 3 2 1 0

Scale 7 6 5 4 3 2 1 0
Index 6 5 4
Base 3 2 1 0
FIGURE 1.10.1 ENIAC, the world’s first general-purpose electronic computer.

Programming in the 40th.