“the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation”


• The visible interface between software and hardware
• What the user (OS, compiler, ...) needs to know to reason about how the machine behaves
• Abstracted from the details of how it may accomplish its task
Architecture (2)

Computer “Architecture” defines

- Instruction set architecture (ISA)
  - Instruction set
  - Operand types
  - Data types (integers, FPs, ...)
  - Memory addressing modes, ...
- Registers and other state
- The interrupt/exception model
- Memory management and protection
- Virtual and physical address layout
- I/O model
- ...
Architecture (3)

- **Microarchitecture**
  - Organization of the machine below the visible level
    - Number/location of functional units
    - Pipeline/cache configurations
    - Programmer transparent techniques: prefetching, ...
  - Must provide same meaning (semantics) as the visible architecture model

- **Implementation**
  - Hardware realization
  - Logic circuits, VLSI technology, process, ...
Architecture (4)

Architecture (4)

- XScale (IXA)
- IA-32 (IA-32e, EM64T, x86-64)
- Intel 64
- Itanium (IA-64)

Microarchitecture

<table>
<thead>
<tr>
<th>Processor Brand Name</th>
<th>P5</th>
<th>P6</th>
<th>NetBurst</th>
<th>Mobile</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Code Name</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P5, P54C, P54CS, P55C, Tillamook</td>
<td>Pentium</td>
<td>Pentium Pro</td>
<td>Pentium 4</td>
<td>Pentium M</td>
<td>Core 2 Quad/Duo/Solo</td>
</tr>
<tr>
<td>P5, P54C, P54CS, P55C, Tillamook</td>
<td>Pentium Pro</td>
<td>Pentium II</td>
<td>Pentium D</td>
<td>Willamette, Northwood, Prescott, Cedar Mill, Smithfield, Presler</td>
<td>Banias, Dothan</td>
</tr>
<tr>
<td>P5, P54C, P55C, Tillamook</td>
<td>Pentium Pro</td>
<td>Pentium II</td>
<td>Pentium III</td>
<td>Willamette, Northwood, Prescott, Cedar Mill, Smithfield, Presler</td>
<td>Yonah</td>
</tr>
</tbody>
</table>
Instruction Set (1)

- **Instruction set**
  - The repertoire of instructions of a computer
  - Different computers have different instruction sets
    - But with many aspects in common
  - Early computers had very simple instruction sets
    - Simplified implementation
  - Many modern computers also have simple instruction sets
    - “RISC (Reduced Instruction Set Computer)”
Instruction Set (2)

- **MIPS instruction set**
  - Similar to other architectures developed since 1980’s
  - Stanford MIPS commercialized by MIPS Technologies ([www.mips.com](http://www.mips.com))
  - Large share of embedded core market
    - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
    - Almost 100 million MIPS processors manufactured in 2002
    - Used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, ...
  - Typical of many modern ISAs
    - See MIPS Reference Data tear-out card (“green card”), and Appendixes B and E
**Arithmetic Operations (1)**

- **Arithmetic operations**
  - Add and subtract, three operands
    - Two sources and one destination

```
add a, b, c  # a ← b + c
```

- All arithmetic operations have this form

- **Design Principle 1**: Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost
Arithmetic Operations (2)

- Arithmetic example
  - C code:

\[
f = (g + h) - (i + j);
\]

- Compiled MIPS code:

```
add t0, g, h  # temp t0 = g + h  
add t1, i, j  # temp t1 = i + j  
sub f, t0, t1  # f = t0 - t1
```
Operands (1)

- **Register operands**
  - Arithmetic instructions use register operands
  - MIPS has a 32 x 32-bit register file
    - Use for frequently accessed data
    - Numbered 0 to 31
    - 32-bit data called a “word”
  - **Assembler names**
    - $t0, t1, …, t9$ for temporary values
    - $s0, s1, …, s7$ for saved variables

- **Design Principle 2: Smaller is faster**
  - (cf.) Main memory: millions of locations
Operands (2)

- Register operand example
  - C code:
    
    ```c
    f = (g + h) - (i + j);
    ```
    
    - f, g, h, i, j in $s0, $s1, $s2, $s3, $s4

  - Compiled MIPS code:
    
    ```mips
    add $t0, $s1, $s2
    add $t1, $s3, $s4
    sub $s0, $t0, $t1
    ```
## MIPS registers

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$zero</td>
<td>The constant value 0</td>
</tr>
<tr>
<td>1</td>
<td>$at</td>
<td>Assembler temporary</td>
</tr>
<tr>
<td>2</td>
<td>$v0</td>
<td>Values for results and expression evaluation</td>
</tr>
<tr>
<td>3</td>
<td>$v1</td>
<td>Arguments</td>
</tr>
<tr>
<td>4</td>
<td>$a0</td>
<td>Temporaries (Caller-save registers)</td>
</tr>
<tr>
<td>5</td>
<td>$a1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$a2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$a3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$t0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$t1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$t2</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>$t3</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>$t4</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>$t5</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>$t6</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>$t7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>$s0</td>
<td>Saved temporaries (Callee-save registers)</td>
</tr>
<tr>
<td>17</td>
<td>$s1</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>$s2</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>$s3</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>$s4</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>$s5</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>$s6</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>$s7</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>$t8</td>
<td>More temporaries (Caller-save registers)</td>
</tr>
<tr>
<td>25</td>
<td>$t9</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>$k0</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>27</td>
<td>$k1</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>Global pointer</td>
</tr>
<tr>
<td>29</td>
<td>$sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>$fp</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>Return address</td>
</tr>
</tbody>
</table>
Operands (4)

- Memory operands
  - Main memory used for composite data
    - Arrays, structures, dynamic data
  - To apply arithmetic operations
    - Load values from memory into registers
    - Store result from register to memory
  - Memory is byte addressed
    - Each address identifies an 8-bit byte
  - Words are aligned in memory
    - Address must be a multiple of 4
  - MIPS is Big Endian
    - Most-significant byte at least address of a word
Operands (5)

- Memory operand example 1
  - C code:
    \[
    g = h + A[8];
    \]
    - g in $s1$, \( h \) in $s2$, base address of \( A \) in $s3$
  - Compiled MIPS code:
    - 4 bytes per word
    - Index 8 requires offset of 32
    \[
    \begin{align*}
    &\text{lw} \quad \text{\$t0, 32($s3)} \quad \# \text{load word} \\
    &\text{add} \quad \text{\$s1, \$s2, \$t0}
    \end{align*}
    \]
Operands (6)

- Memory operand example 2
  - C code:
    
    \[
    \]
  
    - h in $s2$, base address of A in $s3$

  - Compiled MIPS code:
    - Index 8 requires offset of 32
    
    \[
    \begin{align*}
    & \text{lw} \ $t0, 32($s3) \ # \text{load word} \\
    & \text{add} \ $t0, $s2, $t0 \\
    & \text{sw} \ $t0, 48($s3) \ # \text{store word}
    \end{align*}
    \]
Operands (7)

Register vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important
Operands (8)

- **Immediate operands**
  - Constant data specified in an instruction
    
    \[
    \text{addi } \$s3, \$s3, 4
    \]
  
  - No subtract immediate instruction
    - Just use a negative constant
    
    \[
    \text{addi } \$s2, \$s1, -1
    \]

  - **Design Principle 3**: Make the common case fast
    - Small constants are common
    - Immediate operand avoids a load instruction
Operands (9)

- **The constant zero**
  - MIPS register 0 ($zero) is the constant 0
    - Cannot be overwritten
  - Useful for common operations
    - E.g., move between registers

```
add $t2, $s1, $zero
```
Representing Data (1)

- **Unsigned binary integers**
  - Given an n-bit number

\[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0 \]

- Range: 0 to +2^n – 1
- Example
  - 0000 0000 0000 0000 0000 0000 0000 1011_2
    - = 0 + ... + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0
    - = 0 + ... + 8 + 0 + 2 + 1 = 11_{10}

- Using 32 bits
  - 0 to +4,294,967,295
Representing Data (2)

- 2’s-complement signed integers
  - Given an n-bit number
    \[ x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0 \]
  - Range: \(-2^{n-1}\) to \(+2^{n-1} - 1\)
  - Example
    - 1111 1111 1111 1111 1111 1111 1111 1100₂
    - \(-1 \times 2^{31} + 1 \times 2^{30} + \cdots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0\)
    - \(-2,147,483,648 + 2,147,483,644 = -4_{10}\)
  - Using 32 bits
    - \(-2,147,483,648\) to \(+2,147,483,647\)
Representing Data (3)

- 2’s-complement signed integers (cont’d)
  - Bit 31 is sign bit
    - 1 for negative numbers
    - 0 for non-negative numbers
  - \((-2^n - 1)\) can’t be represented
  - Non-negative numbers have the same unsigned and 2’s-complement representation
  - Some specific numbers:

<table>
<thead>
<tr>
<th>Number</th>
<th>2’s-Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 0000 ... 0000</td>
</tr>
<tr>
<td>-1</td>
<td>1111 1111 ... 1111</td>
</tr>
<tr>
<td>Most-negative</td>
<td>1000 0000 ... 0000</td>
</tr>
<tr>
<td>Most-positive</td>
<td>0111 1111 ... 1111</td>
</tr>
</tbody>
</table>
Representing Data (4)

Signed negation

- Complement and add 1
  - Complement means $1 \rightarrow 0$, $0 \rightarrow 1$

\[ \overline{x} + x = 1111\ldots111_2 = -1 \]
\[ \overline{x} + 1 = -x \]

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots \ 0010_2$
  - $-2 = 1111\ 1111\ \ldots \ 1101_2 + 1$
    \[= 1111\ 1111\ \ldots \ 1110_2\]
Representing Data (5)

- **Sign extension**
  - Representing a number using more bits
    - Preserve the numeric value
  - In MIPS instruction set
    - `addi`: extend immediate value
    - `lb, lh`: extend loaded byte/halfword
    - `beq, bne`: extend the displacement
  - Replicate the sign bit to the left
    - (cf.) unsigned values: extend with 0s
  - Examples: 8-bit to 16-bit
    - +2: 0000 0010 \(\rightarrow\) 0000 0000 0000 0010
    - -2: 1111 1110 \(\rightarrow\) 1111 1111 1111 1110
Representing Instructions (1)

- Representing instructions in MIPS
  - Instructions are encoded in binary
    - Called machine code
  - MIPS instructions
    -Encoded as 32-bit instruction words
    - Small number of formats encoding operation code (opcode), register numbers, ...
    - Regularity!
  - Register numbers
    - $t0 - t7$ are registers 8 – 15
    - $t8 - t9$ are registers 24 – 25
    - $s0 - s7$ are registers 16 – 23
Representing Instructions (2)

- **MIPS R-format instructions**

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **Instruction fields**
  - **op**: operation code (opcode)
  - **rs**: first source register number
  - **rt**: second source register number
  - **rd**: destination register number
  - **shamt**: shift amount (00000 for now)
  - **funct**: function code (extends opcode)
Representing Instructions (3)

- MIPS R-format example

<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>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

```
add $t0, $s1, $s2
```

<table>
<thead>
<tr>
<th>special</th>
<th>$s1</th>
<th>$s2</th>
<th>$t0</th>
<th>0</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

00000010001100100100000000010000002 = 0232402016
Representing Instructions (4)

- **MIPS I-format instructions**

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

- Immediate arithmetic and load/store instructions
  - \textbf{rt}: destination or source register number
  - \textbf{Constant}: $-2^{15}$ to $+2^{15} - 1$
  - \textbf{Address}: offset added to base address in \textbf{rs}

- **Design Principle 4**: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible
Representing Instructions (5)

- **MIPS J-format instructions**

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- Jump instructions (j and jal)
  - **Address**: encodes 26-bit target address
Representing Instructions (6)

- **Stored program computers**
  - Instructions represented in binary, just like data
  - Instructions and data stored in memory
  - Programs can operate on programs
    - e.g., compilers, linkers, ...
  - Binary compatibility allows compiled programs to work on different computers
    - Standardized ISAs

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting program (machine code)</td>
</tr>
<tr>
<td>Editor program (machine code)</td>
</tr>
<tr>
<td>C compiler (machine code)</td>
</tr>
<tr>
<td>Payroll data</td>
</tr>
<tr>
<td>Book text</td>
</tr>
<tr>
<td>Source code in C for editor program</td>
</tr>
</tbody>
</table>
Logical Operations (1)

- **Logical operations**
  - Instructions for bitwise manipulation

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

- Useful for extracting and inserting groups of bits in a word
Logical Operations (2)

- **Shift operations**

<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>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

  - **shamt**: how many positions to shift
  - **Shift left logical**
    - Shift left and fill with 0 bits
    - `sll` by \( i \) bits multiples by \( 2^i \)
  - **Shift right logical**
    - Shift right and fill with 0 bits
    - `srl` by \( i \) bits divides by \( 2^i \) (unsigned only)
    - (cf.) `sra`: shift right arithmetic
Logical Operations (3)

- **AND operations**
  - Useful to mask bits in a word
    - Select some bits, clear others to 0

<table>
<thead>
<tr>
<th></th>
<th>$t0$</th>
<th>$t1$</th>
<th>$t2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t0$</td>
<td>0000 0000 0000 0000 0000 0000</td>
<td>1101 1100 0000</td>
<td></td>
</tr>
<tr>
<td>$t1$</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t2$</td>
<td>0000 0000 0000 0000 0000 0000 1100 0000 0000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Logical Operations (4)

- **OR operations**
  - Useful to include bits in a word
    - Set some bits to 1, leave others unchanged

<table>
<thead>
<tr>
<th></th>
<th>$t0</th>
<th>$t1</th>
<th>$t2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0011</td>
<td>1100</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0011</td>
<td>1100</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

\text{or} \ $t0$, \ $t1$, \ $t2$
Logical Operations (5)

- **NOT operations**
  - Useful to invert bits in a word
    - Change 0 to 1, and 1 to 0
  - MIPS has NOR 3-operand instruction
    - $a \text{ NOR } b == NOT (a \text{ OR } b)$

```plaintext
nor $t0, $t1, $zero
```

<table>
<thead>
<tr>
<th>$t1</th>
<th>0000 0000 0000 0000 0011 1100 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t0</td>
<td>1111 1111 1111 1111 1100 0011 1111 1111</td>
</tr>
</tbody>
</table>
```