Signatures and Silicon Art

- Just as the great architects, place their hidden signature, so too do computer designers.

The “Pentium Killer” Macintosh G3 chips were code-named "Arthur" as in Camelot, and the sword represents Excalibur.

Motorola/IBM PowerPC 750

MIPS R10000 Processor
Review: Function calling

- Follow **calling conventions** & nobody gets hurt.

- Function Call Bookkeeping:
  - **Caller:**
    - Arguments  $a0, a1, a2, a3, (sp)
    - Return address  $ra
    - Call function  jal label  # $ra=pc+4; pc=label
  - **Callee:**
    - Not restored  $t0 - t9
    - Restore caller’s  $s0 - $s7, $sp, $fp
    - Return value  $v0, $v1
    - Return  jr $ra  # pc = $ra
Compiling nested C func into MIPS

int sumSquare(int x, int y) { return mult(x,x)+ y; }

SumSquare:

**Prologue**

- addi $sp, $sp,-4
- sw $ra, 0($sp)
- addi $sp, $sp,-4
- sw $a1, 0($sp)
- addi $sp, $sp,-4
- sw $a0, 0($sp)

**Body**

- addi $a1, $a0,$0
- jal mult

**Epilogue**

- lw $a0,0($sp)
- addi $sp, $sp,4
- lw $a1,4($sp)
- addi $sp, $sp,4
- lw $ra, 8($sp)
- addi $sp, $sp,4
- add $v0,$v0,$a1
- jr $ra
int sumSquare(int x, int y) {
  return mult(x,x)+ y;
}

Compiling nested C func into MIPS: combine addi’s

Prologue

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>$sp, $sp,−12</td>
</tr>
<tr>
<td>sw</td>
<td>$ra, 8($sp)</td>
</tr>
<tr>
<td>sw</td>
<td>$a1, 4($sp)</td>
</tr>
<tr>
<td>sw</td>
<td>$a0, 0($sp)</td>
</tr>
</tbody>
</table>

Body

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>$a1, $a0,$0</td>
</tr>
<tr>
<td>jal</td>
<td>mult</td>
</tr>
<tr>
<td>lw</td>
<td>$a0,0($sp)</td>
</tr>
<tr>
<td>lw</td>
<td>$a1,4($sp)</td>
</tr>
</tbody>
</table>

Epilogue

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>$ra, 8($sp)</td>
</tr>
<tr>
<td>add</td>
<td>$v0,$v0,$a1</td>
</tr>
<tr>
<td>addi</td>
<td>$sp,$sp,12</td>
</tr>
<tr>
<td>jr</td>
<td>$ra</td>
</tr>
</tbody>
</table>

# push stack stack
# push return addr
# push y
# push x
# mult(x,x)
# call mult
# pop x
# pop y
# pop return addr
# mult()+y
# pop stack space
Argument Passing greater than 4

- C code fragment
  
  ```
  g = f(a, b, c, d, e);
  ```

- MIPS assembler

  ```
  addi $sp, $sp, -4
  sw $s4, 0($sp)    # push e
  add $a3, $s3, $0  # register push d
  add $a2, $s2, $0  # register push c
  add $a1, $s1, $0  # register push b
  add $a0, $s0, $0  # register push a
  jal f             # $ra = pc + 4
  add $s5, $v0, $0  # g=return value
  ```
### Review: MIPS registers and conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Conventional usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>0</td>
<td>Constant 0 (writing $0 will not change $0)</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Reserved by assembler for pseudo-instructions</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Function “return value;” &amp; temporary usage</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>“function($a0, $a1, $a2 $a3);” calling arguments</td>
</tr>
<tr>
<td>$t0-$t9</td>
<td>8-15,24,25</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>Saved Temporary (preserved across call)</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Global pointer (global static variables)</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack pointer (args &gt; 4 &amp; local function variables)</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Function return address (see jal instruction)</td>
</tr>
</tbody>
</table>
Review: Program memory layout

Address $\infty$

Stack
- $\texttt{sp}$: stack pointer
- $\texttt{fp}$: frame pointer

Heap
- $\texttt{gp}$: global pointer

Static

Code

Memory Management: 
- R: read only
- W: write only
- X: execute only

- (.sdata) Stack data for saved procedure information: return address, dynamic variables, >4 arguments
- Explicitly created space, e.g., malloc(); C pointers
- (.data) Global static vars (.bss initialized to 0)
- (.rdata) const variables and const strings
- (.text) Program: machine instructions
Alternate Static allocation (scope: public to everyone)

- Static declaration
  ```c
  int c[100];
  int *sumarray(int a[], int b[]) {
    int i;
    static int c[100];
    for (i = 0; i < 100; i = i + 1)
      c[i] = a[i] + b[i];
    return c;
  }
  ```

- The variable scope of `c` is very public and is accessible to everyone outside the function
Review: memory allocation model

```
int e[100];
int *sumarray(int a[], int b[]) {
    register int x;
    int i; int d[32]; int *p;
    static int c[100];
    const int f = 3;
    p = (int *)malloc(sizeof(int));
    ...
}
```

- public scope: variable `e`
- private scope:
  - `x`, `i`, `d`, `p`, `c`, `f`
Performance of memory model

speed performance: fastest to slowest

- no setup time, fast access:
  - register int x;
- no setup time, fast access:
  - const int f = 3;
- no setup time($gp), slow access (lw,sw):
  - static int c;
- fast setup time(addi $sp,$sp,-size),
  - slow access (lw,sw):
  - int i; int d[32];
- high setup time(search loop),
  - slow access (lw,sw):
  - malloc(); free();

storage performance: reuse

- unrestricted:
  - malloc(); free();
- unrestricted but cannot free:
  - static int c;
- nested within scope:
  - int i; int d[32];
- limited resource:
  - register int x;
- restricted:
  - const int f = 3;
Global/Static storage examples

- Global C code fragment
  (outside a function)

  char a;
  char b[ ] = “hello\n”
  char c[6] = { -1, 20, 0xf };
  short d;
  int e;
  int f = 0x1f;
  int *g;
  int ********h;
  int *i[5];
  int (*j)(int x, int y);

- MIPS assembler

  .data
  a: .byte 0
  b: .asciiz “hello\n”
  c: .byte -1, 20, 0xf, 0, 0, 0
  d: .half 0
  e: .word 0
  f: .word 0x1f
  g: .word 0
  h: .word 0
  i: .word 0
  j: .word 0
Global variables

- **Global C code fragment**
  (outside a function)
  
  `char  a;`
  `char  *b;`
  `char  *c = &a;`
  `char  ***d;`
  `short e;`
  `short *f;`
  `short ***g;`
  `float h;`
  `float  *i;`
  `double **j`

- **MIPS assembler**
  `.data`
  
  `a:   .byte 0`
  `b:   .word 0`
  `c:   .word a`
  `d:   .word 0`
  `e:   .half 0`
  `f:   .word 0`
  `g:   .word 0`
  `h:   .float 0`
  `i:   .word 0`
  `j:   .word 0`
Dynamic Allocation and access

- **C code**
  
  ```c
  funcion( ) {
    char a;
    char *b;
    char *c=&a;
    char ***d;
    short e;
    short *f;
    short ***g;
    float h;
    float *i;
    double **j
  }
  ```

  - add $fp,$sp,$0
  - add $sp,$sp,-67

- **Stack offset**
  
  - add $sp,$sp,-67
  - 31($sp)
  - 30($sp)
  - 26($sp)
  - 22($sp)
  - 16($sp)
  - 12($sp)
  - 8($sp)
  - 4($sp)
  - 0($sp)
Dynamic initialization of variables

- C code
  function( ) {
    char *c=&a;
  }

- add $fp,$sp,$0
- add $sp,$sp,-67

# -5($fp)  #c:  .word

...  
addi  $t1,$fp,0  #address of a
sw  $t1,5($fp)  #initialize c
Static/Global Struct (by default a public class)

- C code
  ```c
  struct point {
    float x, y;
  };
  
  struct point *p;
  
  struct point  g;
  
  struct point h={7,8};
  ```

- Same as C++ code
  ```cpp
  class point {
    public:
      float x, y;
  };
  
  class point p;
  
  class point g;
  
  class point h={7,8};
  ```

- MIPS assembler
  ```assembly
  p: .word 0
  
  g: .float 0
  
  h: .float 7,8
  ```
Static Classes: inheritance

- C++ code
  ```
  class point { /* base */
  public:
    float x, y;
  }
  
  class rectangle:
    public point { /* derived */
    public:
      float x2, y2;
    }
  
  /* create an instance */
  class point a;
  class rectangle b;
  ```

- MIPS Assembler
  ```
  a: .float 0  #x
  .float 0  #y
  b: .float 0  #x
  .float 0  #y
  .float 0  #x2
  .float 0  #y2
  ```
Unaligned structures: performance issue

```c
struct {
    char c;
    int x, y;
} t;

register int f;

f = t.x;
```

• MIPS Assembler

```
0x4000   t: .byte 0 #c
0x4001   .word 0 #x unaligned
0x4005   .word 0 #y
```

```assembly
# $s1=f
la       $s0,t     #$s1=&t
lw       $s1,0($s0) #$s1=cx_3x_2x_1
sll      $s1,$s1,8  #$s1=x_3x_2x_10
lbu      $at,4($s0) #$t1=000x_0
or       $s1,$s1,$at #$s1=x_3x_2x_1x_0
```

Unaligned words increase the code size and time by a factor of 4 at least!
Instruction as Number Example (decimal)

- **C code:** `i = j + k; /* i-k:$s0-$s2 */`
- **Assembly:** `add $s0,$s1,$s2  #s0=s1+s2`
- **Decimal representation:**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>16</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

- Segments called **fields**
- 1st and last tell MIPS computer to add
- 2nd is 1st source operand (17 = $s1)
- 3rd is 2nd source operand (18 = $s2)
- 4th is destination operand (16 = $s0)
- 5th unused, so set to 0

**Order differs:**
- destination 1st v.last! (common error)
Numbers: Review

- **Number Base B => B symbols per digit:**
  - Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Base 2 (Binary): 0, 1

- **Number representation: \( d_4d_3d_2d_1d_0 \)**
  - \( d_4 \times B^4 + d_3 \times B^3 + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0 \)

- **10010_{\text{ten}} =\right= 1 \times 10^4 + 0 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 0 \times 10^0  
  = 1 \times 10000 + 0 \times 1000 + 0 \times 100 + 1 \times 10 + 0 \times 1  
  = 10000 + 0 + 0 + 10 + 0  
  = 10010_{\text{ten}}**

- **10010_{\text{two}} = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0  
  = 1 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1  
  = 16_{\text{ten}} + 0_{\text{ten}} + 0_{\text{ten}} + 2_{\text{ten}} + 0_{\text{ten}}  
  = 18_{\text{ten}}**
# Numbers: Decimal, Binary, Octal, Hex

<table>
<thead>
<tr>
<th>base 2: Binary</th>
<th>Dec</th>
<th>Bin</th>
<th>Hex</th>
<th>Dec</th>
<th>Bin</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>base 8: Octal</td>
<td>00</td>
<td>0000</td>
<td>00</td>
<td>16</td>
<td>10000</td>
<td>10</td>
</tr>
<tr>
<td>base 10: Decimal</td>
<td>01</td>
<td>00001</td>
<td>01</td>
<td>17</td>
<td>10001</td>
<td>11</td>
</tr>
<tr>
<td>base 16: Hex</td>
<td>02</td>
<td>00010</td>
<td>02</td>
<td>18</td>
<td>10010</td>
<td>12</td>
</tr>
<tr>
<td>Octal (every 3 bits):</td>
<td>03</td>
<td>00011</td>
<td>03</td>
<td>19</td>
<td>10011</td>
<td>13</td>
</tr>
<tr>
<td>= 1001111101</td>
<td>04</td>
<td>00100</td>
<td>04</td>
<td>20</td>
<td>10100</td>
<td>14</td>
</tr>
<tr>
<td>= 1, 001, 111, 101</td>
<td>05</td>
<td>00101</td>
<td>05</td>
<td>21</td>
<td>10101</td>
<td>15</td>
</tr>
<tr>
<td>= 1,1,1,7,5</td>
<td>06</td>
<td>00110</td>
<td>06</td>
<td>22</td>
<td>10110</td>
<td>16</td>
</tr>
<tr>
<td>= 011175 (in C/C++)</td>
<td>07</td>
<td>00111</td>
<td>07</td>
<td>23</td>
<td>10111</td>
<td>17</td>
</tr>
<tr>
<td>Hex (every 4 bits):</td>
<td>08</td>
<td>01000</td>
<td>08</td>
<td>24</td>
<td>11000</td>
<td>18</td>
</tr>
<tr>
<td>= 1001111101</td>
<td>09</td>
<td>01001</td>
<td>09</td>
<td>25</td>
<td>11001</td>
<td>19</td>
</tr>
<tr>
<td>= 10, 0111, 1101</td>
<td>10</td>
<td>01010</td>
<td>0a</td>
<td>26</td>
<td>11010</td>
<td>1a</td>
</tr>
<tr>
<td>= 2, 7, d</td>
<td>11</td>
<td>01011</td>
<td>0b</td>
<td>27</td>
<td>11011</td>
<td>1b</td>
</tr>
<tr>
<td>= 0x27d (in C/C++)</td>
<td>12</td>
<td>01100</td>
<td>0c</td>
<td>28</td>
<td>11100</td>
<td>1c</td>
</tr>
<tr>
<td>Decimal:</td>
<td>13</td>
<td>01101</td>
<td>0d</td>
<td>29</td>
<td>11101</td>
<td>1d</td>
</tr>
<tr>
<td>= 1001111101 = 0x27d</td>
<td>14</td>
<td>01110</td>
<td>0e</td>
<td>30</td>
<td>11110</td>
<td>1e</td>
</tr>
<tr>
<td>= 2<em>16² + 7</em>16¹ + 13*16⁰</td>
<td>15</td>
<td>01111</td>
<td>0f</td>
<td>31</td>
<td>11111</td>
<td>1f</td>
</tr>
</tbody>
</table>
Instruction as Number Example (binary)

- **C code:**
  
  ```c
  i = j + k; /* i-k:$s0-$s2 */
  ```

- **Assembly:**
  
  ```assembly
  add $s0,$s1,$s2 #s0=s1+s2
  ```

- **Decimal representation:**

  | 0 | 17 | 18 | 16 | 0 | 32 |

- **Binary representation:**

  | 000000 | 10001 | 10010 | 10000 | 00000 | 100000 |

  - 6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

  - Called **Machine Language Instruction**

  - Layout called **Instruction Format**

  - All MIPS instructions 32 bits (word): simple!
Big Idea: Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as numbers
  2) Programs can be stored in memory to be read or written just like numbers

- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs
  - Compilers can translate HLL (data) into machine code (instructions)
Big Consequence #1: Everything addressed

- Since all instructions and data are stored in memory as numbers, **everything has a memory address**: instructions, data words
  - branches use memory address of instruction
- **C pointers are just memory addresses**: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: **“Program Counter” (PC)**
  - Better name is Instruction Address Register, but PC is traditional name
Big Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to instruction set architecture
  - Different version for Macintosh and IBM PC
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
- Leads to instruction set evolving over time
- Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium II); could still run program from 1981 PC today
Instruction Format Field Names

- Fields have names:

<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>

- **op**: basic operation of instruction, “opcode”
- **rs**: 1st register source operand
- **rt**: 2nd register source operand
- **rd**: register destination operand, gets the result
- **shamt**: shift amount (used later, so 0 for now)
- **funct**: function; selects the specific variant of the operation in the op field; sometimes called the function code
Instruction Formats

<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>

• What if want longer fields? e.g., `lw $2 32($5)`
  – 5 bits => address of $2^5$ or 32 => too small
  – But want all instructions same length!
• Principle: Good design demands good compromises
  – Add 2nd format with larger address

- 1st format **R (register)**; 2nd format **I (immediate)**
Notes about Register and Imm. Formats

To make it easier for hardware (HW), 1st 3 fields same in R-format and I-format

Alas, \( r_t \) field meaning changed
- R-format: \( r_t \) is 2nd source operand
- I-format: \( r_t \) can be register destination operand

How HW know which format is which?
- Distinct values in 1st field (op) tell whether last 16 bits are 3 fields (R-format) or 1 field (I-format)
### Instructions, Formats, “opcodes”

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>op</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>–add</td>
<td>Register</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>–sub</td>
<td>Register</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>–slt</td>
<td>Register</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>–jr</td>
<td>Register</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>–lw</td>
<td>Immediate</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>–sw</td>
<td>Immediate</td>
<td></td>
<td>43</td>
</tr>
<tr>
<td>–addi</td>
<td>Immediate</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>–beq</td>
<td>Immediate</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>–bne</td>
<td>Immediate</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>–slti</td>
<td>Immediate</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

Register Format if op field = 0
Immediate Instruction in Machine Code

- **C code:** `i = j + 4; /* i,j:$s0,$s1 */`
- **Assembly:** `addi $s0,$s1,4  #$s0=$s1+4`

**Format:**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>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>

**Decimal representation:**

<table>
<thead>
<tr>
<th>8</th>
<th>17</th>
<th>16</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

**Binary representation:**

<table>
<thead>
<tr>
<th>001000</th>
<th>10001</th>
<th>10000</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
<th>0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# MIPS instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALU</strong></td>
<td><code>alu $rd,$rs,$rt</code></td>
<td>$rd = $rs &lt;alu&gt; $rt</td>
</tr>
<tr>
<td><strong>ALUi</strong></td>
<td><code>alui $rd,$rs,const16</code></td>
<td>$rd = $rs &lt;alu&gt; const16</td>
</tr>
<tr>
<td><strong>Data</strong></td>
<td><code>lw $rt,byteoffset($rs)</code></td>
<td>$rt = Mem[$rs + offset]</td>
</tr>
<tr>
<td></td>
<td><code>sw $rt,byteoffset($rs)</code></td>
<td>Mem[$rs + offset] = $rt</td>
</tr>
<tr>
<td><strong>Transfer</strong></td>
<td><code>beq $rs,$rt,wordoffset</code></td>
<td>$pc = ($rd == $rs)? (pc+4+offset&lt;&lt;2):(pc+4);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All instructions occur on word aligned (i.e. 0,4,8, ...) don’t need last 2 bits.</td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td><code>j wordaddress</code></td>
<td>pc = wordaddress &lt;&lt; 2</td>
</tr>
</tbody>
</table>
# MIPS fixed sized instruction formats

## R - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
</table>

**ALU**

\[
\text{alu } \text{rd}, \text{rs}, \text{rt}
\]

## I - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>const16 or byteoffset or wordoffset</th>
</tr>
</thead>
</table>

**ALUi**

\[
\text{alui } \text{rt}, \text{rs}, \text{const16}
\]

**Data**

\[
\text{lw } \text{rt}, \text{byteoffset}(\text{rs})
\]

**Transfer**

\[
\text{sw } \text{rt}, \text{byteoffset}(\text{rs})
\]

**Branch**

\[
\text{beq } \text{rs}, \text{rt}, \text{wordoffset}
\]

## J - Format

<table>
<thead>
<tr>
<th>op</th>
<th>absolute address</th>
</tr>
</thead>
</table>

**Jump**

\[
\text{j } \text{wordoffset}
\]
Assembling Instructions

Suppose there are 32 registers, addu opcode=001001, addi op=001000

\[
\begin{array}{cccccc}
\text{op} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{func} \\
0x001001 & 00000 & 11111 & 10111 & 00000 & 0000000000000000
\end{array}
\]

\[
\text{alu$ \text{rd}, \text{rs}, \text{rt}$}
\]

\[
\begin{array}{cccccc}
\text{op} & \text{rs} & \text{rt} & \text{value} \text{ or offset} \\
0x001000 & 00000 & 00101 & 0000000000000101
\end{array}
\]

\[
\text{alui$ \text{rt}, \text{rs}, \text{value}$}
\]

\[
\begin{array}{cccccc}
\text{op} & \text{rs} & \text{rt} & \text{value} \text{ or offset} \\
0x001000 & 00000 & 00101 & 0000000000000101
\end{array}
\]

\[
\text{addi$ \text{rt}, \text{rs}, 5$}
\]
MIPS instruction formats

**Arithmetic**
- addi $rt, $rs, const16
- add $rd, $rs, $rt

**Data Transfer**
- lw $rt, byteoffset($rs)
- sw $rt, byteoffset($rs)

**Conditional branch**
- beq $rs, $rt, wordoffset

**Unconditional jump**
- j wordaddress
C function to MIPS Assembly Language

```c
int power_2(int y) { /* compute x=2^y; */
    register int x, i; x=1; i=0; while(i<y) { x=x*2; i=i+1; }
    return x;
}
```

Assembly:
```
addi $t0, $0, 1  # x=1;
addu $t1, $0, $0  # i=0;
w1:  bge $t1,$a0,w2  # while(i<y) { /* bge= greater or equal */
    addu $t0, $t0, $t0  # x = x * 2; /* same as x=x+x; */
    addi $t1,$t1,1  # i = i + 1;
    beq $0,$0,w1  # }
w2:  addu $v0,$0,$t0  # return x;
    jr $ra  # jump on register ( pc = ra; )
```

Comments:
- Exit condition of a while loop is if ( i >= y ) then goto w2
Power_2.s: MIPS storage assignment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400020</td>
<td>addi $8, $0, 1 # addi $t0, $0, 1</td>
<td></td>
</tr>
<tr>
<td>0x00400024</td>
<td>addu $9, $0, $0 # addu $t1, $0, $0</td>
<td></td>
</tr>
<tr>
<td>0x00400028</td>
<td>bge $9, $4, 2 # bge $t1, $a0, w2</td>
<td></td>
</tr>
<tr>
<td>0x0040002c</td>
<td>addu $8, $8, $8 # addi $t0, $t0, $t0</td>
<td></td>
</tr>
<tr>
<td>0x00400030</td>
<td>addi $9, $9, 1 # addi $t1, $t1, 1</td>
<td></td>
</tr>
<tr>
<td>0x00400034</td>
<td>beq $0, $0, -3 # beq $0, $0, w1</td>
<td></td>
</tr>
<tr>
<td>0x00400038</td>
<td>addu $2, $0, $8 # addu $v0, $0, $t0</td>
<td></td>
</tr>
<tr>
<td>0x0040003c</td>
<td>jr $31 # jr $ra</td>
<td></td>
</tr>
</tbody>
</table>

2 words after pc fetch after bge fetch pc is 0x00400030 plus 2 words is 0x00400038

Byte address, not word address
Machine Language Single Stepping

Assume `power2(0)` is called; then $a0=0$ and $ra=700018$

<table>
<thead>
<tr>
<th></th>
<th>$v0$</th>
<th>$a0$</th>
<th>$t0$</th>
<th>$t1$</th>
<th>$ra$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pc$</td>
<td>$2$</td>
<td>$4$</td>
<td>$8$</td>
<td>$9$</td>
<td>$31$</td>
</tr>
<tr>
<td>00400020</td>
<td>?</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>700018</td>
</tr>
<tr>
<td>00400024</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>?</td>
<td>700018</td>
</tr>
<tr>
<td>00400028</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>700018</td>
</tr>
<tr>
<td>00400038</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>700018</td>
</tr>
<tr>
<td>0040003c</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>700018</td>
</tr>
<tr>
<td>00700018</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>700018</td>
</tr>
</tbody>
</table>

Values change after the instruction!