Language of the Machine
Machine Instructions
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
Argument Passing greater than 4

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

- MIPS assembler

```assembly
addi     $sp, $sp, -4   # push e
sw       $s4, 0($sp)    # register push d
add       $a3, $s3, $0  # register push c
add       $a2, $s2, $0  # register push b
add       $a1, $s1, $0  # register push a
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>$0</td>
<td>0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Expression evaluation &amp; function return</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Arguments 1 to 4</td>
</tr>
<tr>
<td>$t0-$t9</td>
<td>8-15,24,35</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>$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>Pointer to global area</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 (used by function call)</td>
</tr>
</tbody>
</table>
Review: Program memory layout

Address $\infty$

Stack

- $\textit{sp}$: Stack pointer
- $\textit{fp}$: Frame pointer

Heap

- $\textit{gp}$: 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
- (.rdata) Const variables and const strings
- (.data) Global static vars (.bss initialized to 0)
- (.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
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;
  - 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
  }
  ```

- **Stack offset**
  ```assembly
  add $fp,$sp,$0
  add $sp,$sp,-67
  # 0($fp) #a: .byte
  # -1($fp) #b: .word
  # -5($fp) #c: .word
  # -9($fp) #d: .word
  # -13($fp) #e: .half
  # -15($fp) #f: .word
  # -19($fp) #g: .word
  # -23($fp) #h: .float
  # -27($fp) # i: .word
  # -31($fp) # j: .word
  ```

- **Stack offset**
  ```assembly
  add $sp,$sp,-67
  ```
Dynamic initialization of variables

- **C code**
  
  ```c
  funcion( ) {
    char *c=&a;

    # -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;
  };
  ```

  ```c
  struct point *p;
  ```

  ```c
  struct point  g;
  ```

  ```c
  struct point h={7,8};
  ```

- Same as C++ code
  ```cpp
  class point {
    public:
      float x, y;
  };
  ```

- MIPS assembler
  ```mips
  p: .word 0
  ```

  ```mips
  g: .float 0
  ```

  ```mips
  h: .float 7,8
  ```
Static Classes: inheritance

• C++ code

```cpp
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;
```
Instruction as Number Example (decimal)

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>17</th>
<th>18</th>
<th>16</th>
<th>0</th>
<th>32</th>
</tr>
</thead>
</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: \( d_4d_3d_2d_1d_0 \)**

  \[
  \begin{align*}
  d_4 \times B^4 + d_3 \times B^3 + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0 \\
  = 10010_{\text{ten}} = 1 \times 10^4 + 0 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 0 \times 10^0 \\
  = 1 \times 10000 + 0 \times 1000 + 0 \times 100 + 1 \times 10 + 0 \times 1 \\
  = 10000 + 0 + 0 + 10 + 0 \\
  = 10010_{\text{ten}} \\
  \\
  = 10010_{\text{two}} = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\
  = 1 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\
  = 16_{\text{ten}} + 0_{\text{ten}} + 0_{\text{ten}} + 2_{\text{ten}} + 0_{\text{ten}} \\
  = 18_{\text{ten}}
  \end{align*}
\]
### Numbers: Decimal, Binary, Octal, Hex

<table>
<thead>
<tr>
<th>base 10: Decimal</th>
<th>base 2: Binary</th>
<th>base 8: Octal</th>
<th>base 16: Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00000 00</td>
<td>00</td>
<td>16 10000 10</td>
</tr>
<tr>
<td>01</td>
<td>00001 01</td>
<td>01</td>
<td>17 10001 11</td>
</tr>
<tr>
<td>02</td>
<td>00010 02</td>
<td>02</td>
<td>18 10010 12</td>
</tr>
<tr>
<td>03</td>
<td>00011 03</td>
<td>03</td>
<td>19 10011 13</td>
</tr>
<tr>
<td>04</td>
<td>00100 04</td>
<td>04</td>
<td>20 10100 14</td>
</tr>
<tr>
<td>05</td>
<td>00101 05</td>
<td>05</td>
<td>21 10101 15</td>
</tr>
<tr>
<td>06</td>
<td>00110 06</td>
<td>06</td>
<td>22 10110 16</td>
</tr>
<tr>
<td>07</td>
<td>00111 07</td>
<td>07</td>
<td>23 10111 17</td>
</tr>
<tr>
<td>08</td>
<td>01000 08</td>
<td>08</td>
<td>24 11000 18</td>
</tr>
<tr>
<td>09</td>
<td>01001 09</td>
<td>09</td>
<td>25 11001 19</td>
</tr>
<tr>
<td>10</td>
<td>01010 0A</td>
<td>10</td>
<td>26 11010 1A</td>
</tr>
<tr>
<td>11</td>
<td>01011 0B</td>
<td>11</td>
<td>27 11011 1B</td>
</tr>
<tr>
<td>12</td>
<td>01100 0C</td>
<td>12</td>
<td>28 11100 1C</td>
</tr>
<tr>
<td>13</td>
<td>01101 0D</td>
<td>13</td>
<td>29 11101 1D</td>
</tr>
<tr>
<td>14</td>
<td>01110 0E</td>
<td>14</td>
<td>30 11110 1E</td>
</tr>
<tr>
<td>15</td>
<td>01111 0F</td>
<td>15</td>
<td>31 11111 1F</td>
</tr>
</tbody>
</table>

**Octal example:**
```
01111101
=175
```

**Hex example:**
```
01111101
=7d
```
Instruction as Number Example (binary)

- C code: \( i = j + k; \ /* i-k:\$s0-$s2 */ \)
- Assembly: \texttt{add \$s0,\$s1,\$s2}  \#\$s0=s1+s2
- Decimal representation:
  \[ \begin{array}{cccccc}
  0 & 17 & 18 & 16 & 0 & 32 \\
  \end{array} \]
- Binary representation:
  \[ \begin{array}{cccccc}
  000000 & 10001 & 10010 & 10000 & 00000 & 100000 \\
  \end{array} \]
  
  \(6\text{ bits} \quad 5\text{ bits} \quad 5\text{ bits} \quad 5\text{ bits} \quad 5\text{ bits} \quad 6\text{ bits}\)

- Called \textbf{Machine Language Instruction}
- Layout called \textbf{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., \( \text{lw } \$2 \ 32(\$5) \)
  - 5 bits \( \Rightarrow \) address of \( 2^5 \) or 32 \( \Rightarrow \) too small
  - But want all instructions same length!
- Principle: Good design demands good compromises
  - Add 2nd format with larger address

- 1st format \( \text{R (register)} \); 2nd format \( \text{l (immediate)} \)
Notes about Register and Imm. Formats

<table>
<thead>
<tr>
<th>R:</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I:</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>address</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</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>
</tr>
</tbody>
</table>

- To **make it easier for hardware** (HW), 1st 3 fields same in R-format and I-format
- **Alas, \( rt \) field meaning changed**
  - R-format: \( rt \) is 2nd source operand
  - I-format: \( rt \) 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></td>
</tr>
<tr>
<td>–sw</td>
<td>Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–addi</td>
<td>Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–beq</td>
<td>Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–bne</td>
<td>Immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–slti</td>
<td>Immediate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Format if op field = 0

CWRU EECS 322  26
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>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>
<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>alu $rd,$rs,$rt</td>
<td>$rd = $rs &lt;alu&gt; $rt</td>
</tr>
<tr>
<td><strong>ALUi</strong></td>
<td>alui $rd,$rs,value</td>
<td>$rd = $rs &lt;alu&gt; value</td>
</tr>
<tr>
<td><strong>Data</strong></td>
<td>lw $rt,offset($rs)</td>
<td>$rt = Mem[$rs + offset]</td>
</tr>
<tr>
<td><strong>Transfer</strong></td>
<td>sw $rt,offset($rs)</td>
<td>Mem[$rs + offset] = $rt</td>
</tr>
<tr>
<td><strong>Branch</strong></td>
<td>beq $rs,$rt,offset</td>
<td>$pc = ($rd == $rs)? (pc+4+offset):(pc+4);</td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>j address</td>
<td>pc = address</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>

**I - Format**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>value or offset</th>
</tr>
</thead>
</table>

**J - Format**

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

**ALU**

| alu $rd,$rs,$rt |

**ALUi**

| alui $rt,$rs,value |

**Data Transfer**

| lw $rt,offset($rs) |
| sw $rt,offset($rs) |

**Branch**

| beq $rs,$rt,offset |

**Jump**

| j address |
Suppose there are 32 registers, `addu` opcode=001001, `addi` op=001000

- **`addu`**
  - Opcode: 001001
  - 32 registers
  - `alu $rd,$rs,$rt`
  - Example: `addu $23, $0, $31`
  - Binary: 01000000001011100000000000000000

- **`addi`**
  - Opcode: 001000
  - `alui $rt,$rs,value`
  - Example: `addi $17, $0, 5`
  - Binary: 01000000001010000000000000000101
MIPS instruction formats

Arithmetic
- addi $rt, $rs, value
- add $rd,$rs,$rt

Data Transfer
- lw $rt,offset($rs)
- sw $rt,offset($rs)

Conditional branch
- beq $rs,$rt,offset

Unconditional jump
- j address
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;
}
```

**Assemble.s**

```
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>Opcode</th>
<th>Operands</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400020</td>
<td>addi</td>
<td>$8, $0, 1</td>
<td># addi $t0, $0, 1</td>
</tr>
<tr>
<td>0x00400024</td>
<td>addu</td>
<td>$9, $0, $0</td>
<td># addu $t1, $0, $0</td>
</tr>
<tr>
<td>0x00400028</td>
<td>bge</td>
<td>$9, $4, 2</td>
<td># bge $t1, $a0, w2</td>
</tr>
<tr>
<td>0x0040002c</td>
<td>addu</td>
<td>$8, $8, $8</td>
<td># addi $t0, $t0, $t0</td>
</tr>
<tr>
<td>0x00400030</td>
<td>addi</td>
<td>$9, $9, 1</td>
<td># addi $t1, $t1, 1</td>
</tr>
<tr>
<td>0x00400034</td>
<td>beq</td>
<td>$0, $0, -3</td>
<td># beq $0, $0, w1</td>
</tr>
<tr>
<td>0x00400038</td>
<td>addu</td>
<td>$2, $0, $8</td>
<td># addu $v0, $0, $t0</td>
</tr>
<tr>
<td>0x0040003c</td>
<td>jr</td>
<td>$31</td>
<td># jr $ra</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>$pc</th>
<th>$v0</th>
<th>$a0</th>
<th>$t0</th>
<th>$t1</th>
<th>$ra</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400020</td>
<td>?</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>700018</td>
</tr>
<tr>
<td></td>
<td>$2</td>
<td>$4</td>
<td>$8</td>
<td>$9</td>
<td>$31</td>
</tr>
<tr>
<td>00400024</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>?</td>
<td>700018</td>
</tr>
<tr>
<td></td>
<td>addi</td>
<td>$t0, $0, 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00400028</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>700018</td>
</tr>
<tr>
<td></td>
<td>addu</td>
<td>$t1, $0, $0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00400038</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>700018</td>
</tr>
<tr>
<td></td>
<td>bge</td>
<td>$t1,$a0,w2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0040003c</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>700018</td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>$v0,$0,$t0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00700018</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>700018</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>$ra</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>