MIPS Programming
CS 740
Sept. 17 & 19
1997

Topics
- Basics
- Control Flow
- Procedures
- Data Representations
- Instruction Formats
MIPS Machines

Reduced Instruction Set Computer (RISC)

- Simple instructions with regular formats
- Targeted to implementation with CPI $\approx 1.0$
  - Common instructions execute in one clock cycle

Assumes compiler will do optimizations

- Register allocation
- Code scheduling
- Even assembler does some optimizations!

Widespread usage

- Commercially
  - SGI, Sony Playstation, other embedded systems
- At CMU
  - DECStations (Including unix servers)
  - SGI Indigos
Translation Process

- **Text**
  - **C program (p1.c p2.c)**
    - **Compiler (gcc -S)**
      - **Asm program (p1.s p2.s)**
        - **Assembler (gcc or as)**
          - **Object program (p1.o p2.o)**
            - **Linker (gcc or ld)**
              - **Executable program (p)**
                - **Disassembler (dis -h) Debugger (gdb)**
                  - **libraries (.a)**
                    - **Text**
                      - **disassembled program**
## Abstract machines

<table>
<thead>
<tr>
<th>Machine Model</th>
<th>Data</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C</strong></td>
<td>1) char</td>
<td>1) loops</td>
</tr>
<tr>
<td>mem</td>
<td>2) int, float</td>
<td>2) conditionals</td>
</tr>
<tr>
<td>proc</td>
<td>3) double</td>
<td>3) goto</td>
</tr>
<tr>
<td></td>
<td>4) struct, array</td>
<td>4) Proc. call</td>
</tr>
<tr>
<td></td>
<td>5) pointer</td>
<td>5) Proc. return</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASM</th>
<th>1) byte</th>
<th>3) branch/jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>2) word</td>
<td>4) jump &amp; link</td>
</tr>
<tr>
<td>regs</td>
<td>3) doubleword</td>
<td></td>
</tr>
<tr>
<td>alu</td>
<td>4) contiguous word allocation</td>
<td></td>
</tr>
<tr>
<td>processor</td>
<td>5) address of initial byte</td>
<td></td>
</tr>
</tbody>
</table>
MIPS Register Convention

General Purpose Registers
- 32 total
- Store integers and pointers
- Fast access: 2 reads, 1 write in single cycle

Usage Conventions
- Established as part of architecture
- Used by all compilers, programs, and libraries
- Assures object code compatibility
  - E.g., can mix Fortran and C

<table>
<thead>
<tr>
<th>General Purpose Registers</th>
<th>Return Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$1$</td>
<td>$at$</td>
</tr>
<tr>
<td>$2$</td>
<td>$v0$</td>
</tr>
<tr>
<td>$3$</td>
<td>$v1$</td>
</tr>
<tr>
<td>$4$</td>
<td>$a0$</td>
</tr>
<tr>
<td>$5$</td>
<td>$a1$</td>
</tr>
<tr>
<td>$6$</td>
<td>$a2$</td>
</tr>
<tr>
<td>$7$</td>
<td>$a3$</td>
</tr>
<tr>
<td>$8$</td>
<td>$t0$</td>
</tr>
<tr>
<td>$9$</td>
<td>$t1$</td>
</tr>
<tr>
<td>$10$</td>
<td>$t2$</td>
</tr>
<tr>
<td>$11$</td>
<td>$t3$</td>
</tr>
<tr>
<td>$12$</td>
<td>$t4$</td>
</tr>
<tr>
<td>$13$</td>
<td>$t5$</td>
</tr>
<tr>
<td>$14$</td>
<td>$t6$</td>
</tr>
<tr>
<td>$15$</td>
<td>$t7$</td>
</tr>
</tbody>
</table>

Constant 0
Reserved Temp.
Return Values
Procedure arguments
Caller Save
Temporaries:
May be overwritten by called procedures
## Registers (cont.)

### Important Ones for Now

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>$2</td>
<td>Return Value</td>
</tr>
<tr>
<td>$3</td>
<td>Can use as temporary</td>
</tr>
<tr>
<td>$4</td>
<td>First argument</td>
</tr>
<tr>
<td>$5</td>
<td>Second argument</td>
</tr>
<tr>
<td>$31</td>
<td>Return address</td>
</tr>
</tbody>
</table>

### Registers

<table>
<thead>
<tr>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16</td>
</tr>
<tr>
<td>$17</td>
</tr>
<tr>
<td>$18</td>
</tr>
<tr>
<td>$19</td>
</tr>
<tr>
<td>$20</td>
</tr>
<tr>
<td>$21</td>
</tr>
<tr>
<td>$22</td>
</tr>
<tr>
<td>$23</td>
</tr>
<tr>
<td>$24</td>
</tr>
<tr>
<td>$25</td>
</tr>
<tr>
<td>$26</td>
</tr>
<tr>
<td>$27</td>
</tr>
<tr>
<td>$28</td>
</tr>
<tr>
<td>$29</td>
</tr>
<tr>
<td>$30</td>
</tr>
<tr>
<td>$31</td>
</tr>
</tbody>
</table>

- **$s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7, $t8, $t9, $k0, $k1, $gp, $sp, $s8, $ra**: These registers are reserved for operating system use.
- **Callee Save Temporaries**: May not be overwritten by called procedures.
- **Caller Save Temp**: Used for temporary storage by the calling program.
- **Global Pointer**: Points to the beginning of the global data area.
- **Stack Pointer**: Points to the top of the stack.
- **Callee Save Temp**: Used for temporary storage by the called procedure.
- **Return Address**: Points to the next instruction after the current one.
Program Representations

C Code

```c
int gval;

void test1(int x, int y)
{
    gval = (x+x+x) - (y+y+y);
}
```

Compiled to Assembly

```
.comm   gval,8
.loc    1 7
.ent    test1

    test1:
        .frame  $sp,0,$31
        .mask   0x00000000,0
        .fmask  0x00000000,0
        addu    $3,$4,$4
        addu    $3,$3,$4
        addu    $2,$5,$5
        addu    $2,$3,$4
        subu    $3,$3,$2
        sw      $3,gval
        j       $31
    .end    test1
```

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`
### Prog. Representation (Cont.)

<table>
<thead>
<tr>
<th>Object</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400230 &lt;test1&gt;:</td>
<td>0x400230 &lt;test1&gt;:</td>
</tr>
<tr>
<td>0x00841821</td>
<td>addu $v1,$a0,$a0</td>
</tr>
<tr>
<td>0x00a51021</td>
<td>addu $v0,$a1,$a1</td>
</tr>
<tr>
<td>0x00451021</td>
<td>addu $v0,$v0,$a1</td>
</tr>
<tr>
<td>0x00641821</td>
<td>addu $v1,$v1,$a0</td>
</tr>
<tr>
<td>0x0061823</td>
<td>subu $v1,$v1,$v0</td>
</tr>
<tr>
<td>0x03e00008</td>
<td>jr $ra</td>
</tr>
<tr>
<td>0xaf838108</td>
<td>sw $v1,-32504($gp)</td>
</tr>
</tbody>
</table>

---

Run gdb on object code

```
x/7 0x400230
```

- Print 7 word in hexadecimal starting at address 0x400230

```
dissasemlble test1
```

- Print disassembled version of procedure
Alternate Disassembly

MIPS program “dis”

```
dis -h file.o
```
- Prints disassembled version of object code file
- Using register names r1–r31
- Code not yet linked
  - Addresses of procedures and global data not yet resolved

```
test1:
  0x0: 00841821 addu r3,r4,r4
  0x4: 00a51021 addu r2,r5,r5
  0x8: 00451021 addu r2,r2,r5
  0xc: 00641821 addu r3,r3,r4
  0x10: 00621823 subu r3,r3,r2
  0x14: 03e00008 jr r31
  0x18: af830000 sw r3,0(gp)
```
Delayed Jumps & Branches

C Code

```c
int test2(int x, int y)
{
    return (x+x+x) - (y+y+y);
}
```

Normal jump

- No code after jump executed

Delayed jump

- Execute instruction following jump
- Strange semantics, efficient implementation

Compiled to Assembly

```
.loc 1 12
.ent test2
test2:
    .frame $sp,0,$31
    .mask 0x00000000,0
    .fmask 0x00000000,0
    addu $2,$4,$4
    addu $2,$2,$4
    addu $3,$5,$5
    addu $3,$3,$5
    .set noreorder
    .set nomacro
    j $31
    subu $2,$2,$3
    .set macro
    .set reorder
.end test2
```
Annotated Delayed Branch Code

C Code

```c
int test2(int x, int y)
{
    return (x+x+x) - (y+y+y);
}
```

Annotated Assembly

```assembly
test2:
    # x in $4, y in $5
    # Return result in $2
    addu    $2,$4,$4   # $2 = x+x
    addu    $2,$2,$4   # $2 += x
    addu    $3,$5,$5   # $3 = y+y
    addu    $3,$3,$5   # $3 += y
    J       $31        # (Delayed) return
    subu    $2,$2,$3   # with value $2-$3
```

Notation Convention

- Upper case denotes delayed branch
- Lower case denotes normal branch

For Exposition Only

- Actual machine supports only delayed branches
- Regular ones converted by assembler
### Pointer Examples

#### C Code

```c
int iaddp(int *xp, int *yp)
{
    int x = *xp;
    int y = *yp;
    return x + y;
}
```

```c
void incr(int *sum, int v)
{
    int old = *sum;
    int new = old + val;
    *sum = new;
}
```

#### Annotated Assembly

**iaddp:**

- `lw $2,0($4)` # $2 = *xp
- `lw $3,0($5)` # $3 = *yp
- `J $31` # (delayed) return
- `addu $2,$2,$3` # with value $2+$3

**incr:**

- `lw $2,0($4)` # $2 = *sum
- `addu $2,$2,$5` # $2+= v
- `J $31` # (delayed) return
- `sw $2,0($4)` # with *sum = $2
Array Indexing

C Code

```c
int aref(int a[], int i) {
    return a[i];
}

int garray[10];

int gref(int i) {
    return garray[i];
}
```

Annotated Assembly

```assembly
aref:
    sll $5,$5,2  # $5 = 4*i
    addu $5,$5,$4 # $5 = &a[i]
    lw $2,0($5) # return val. = a[i]
    j $31       # Return

.comm garray,40

gref:
    sll $4,$4,2  # $4 = 4*i
    lw $2,garray($4) #Return val = garray[i]
    j $31 # Return
```

Disassembled:

```
0x40034c <gref>: sll r4,r4,0x2
0x400350 <gref+4>: lui r2,4096
0x400354 <gref+8>: addu r2,r2,r4
0x400358 <gref+12>: lw r2,4304(r2)
0x40035c <gref+16>: JR r31
0x400360 <gref+20>: nop
```
C Code

```c
struct rec {
    int i;
    int a[3];
    int *p;
};

void addr(struct rec *r) {
    int *loc;
    r->i = 1;
    loc = &r->a[r->i];
    r->p = loc;
    *(r->p) = 2;
    r->a[0] = 4;
    *(r->p+1) = 8;
}
```

Annotated Assembly

```
addr:
  li   $2,0x00000001  # $2 = 1
  sw   $2,0($4)      # r->i = 1
  addu $2,$4,8       # $2 = &r->a[1]
  sw   $2,16($4)     # r->p = $2
  li   $2,0x00000002  # $2 = 1
  sw   $2,16($4)     # r->a[1] = 2
  lw   $2,16($4)     # $2 = r->p
  li   $3,0x00000004  # $3 = 4
  li   $5,0x00000008  # $4 = 8
  sw   $3,4($4)      # r->a[0] = 4
  J    $31           # Return
  sw   $5,4($2)      # setting
    # *(r->p+1) = 8
```
MIPS Implemented Branches

Formats
- \texttt{bCOND Rs, Rt, label}
  - \textit{Cond} : Branch condition
- \texttt{bCOND Rs, label}
  - \textit{Cond} : Branch condition, relative to zero

Conditions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Reg. Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{beq}</td>
<td>\texttt{EEqual}</td>
<td>2</td>
</tr>
<tr>
<td>\texttt{bne}</td>
<td>\texttt{Not Equal}</td>
<td>2</td>
</tr>
<tr>
<td>\texttt{bgez}</td>
<td>\texttt{Greater or Equal to Zero}</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{bgtz}</td>
<td>\texttt{Greater Than Zero}</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{blez}</td>
<td>\texttt{Less or Equal to Zero}</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{bltz}</td>
<td>\texttt{Less Than Zero}</td>
<td>1</td>
</tr>
</tbody>
</table>

Implementation
- All have single cycle delay slot
Branch Notation Convention

Caveat

• For exposition only
• Not machine-readable assembly code

Conventional Branches

• Lower Case
  – j, jr, beq, bltz, etc.
• If branch taken, following instruction not executed
• Allowed in assembly

Delayed Branches

• Upper Case
  – J, JR, BEQ, BLTZ, etc.
• Following instruction executed whether or not branch taken
• Actually implemented
• Use assembler directives to enable/disable.
Conditional Example

C Code

```c
int max(int x, int y)
{
    return (x < y) ? y : x;
}
```

Compiled

```assembly
max:
    # x in $4, y in $5
    slt  $2,$4,$5   # $4 < $5 ?
    beq  $2,$0,$L2 # if not, goto $L2
    move $4,$5     # else $4 = y
$L2:
    J     $31        # Return
    move $2,$4       # with $4
```

Disassembled

```
0xdc: 0085102a    slt     r2,r4,r5
0xe0: 10400002    BEQ     r2,r0,0xec
0xe4: 00000000    nop
0xe8: 00a02021    move    r4,r5
0xec: 03e00008    JR       r31
0xf0: 00801021    move    r2,r4
```
Loop Example

C Code

```c
int fact(int x) {
    int result = 1;
    while (x >= 2) {
        result *= x--;
    }
    return result;
}
```

Handwritten Assembly

```assembly
fact:
    li      $2,0x1
    blt     $4,2,$L5  # Skip if x < 2
$L4:
    mul     $2,$4    # result *= x
    addi    $4,$4,-1  # x--
    bge     $4,2,$L4  # loop if x >= 2
$L5:
    j       $31      # return
.end    fact
```

Macro-Expanded Operations

```
blt $Rs, $Rt, label
Branch if $Rs < $Rt
mul $Rdest, $Rsrc
$Rdest *= $Rsrc
```
MIPS Integer Multiplication

Multiplication Operation

- **Arguments:** 32-bits each
  - Unsigned: `multu $1, $2`
  - Signed: `mult $1, $2`

- **Product:** 64 bits
  - Stored in special register pair
  - High & Low order words

- **Retrieve result with special instructions**
  - `mfhi $3`
  - `mflo $2`

- **SLOW!**
  - Many clock cycles
Assembled Loop Code

Handwritten Assembly

```
li      $2,0x1
blt     $4,2,$L5 # Skip if x < 2
$L4:
  mul     $2,$4    # result *= x
  addi    $4,$4,-1 # x--
  bge     $4,2,$L4 # loop if x >= 2
$L5:
  j       $31      # return
```

Macro expansion uses register $1 ($at) as temporary

Disassembled

```
0x0:   28810002        slti    r1,r4,2
0x4:   14200007        BNE     r1,r0,0x24
0x8:   24020001        li      r2,1
0xc:   00440019        multu   r2,r4
0x10:  2084ffff        addi    r4,r4,-1
0x14:  28810002        slti    r1,r4,2
0x18:  00001012        mflo    r2
0x1c:  1020ffff        BEQ     r1,r0,0xc
0x20:  00000000        nop
0x24:  03e00008        JR      r31
0x28:  00000000        nop
```
Translation of for Loop

C Code

/* Find max ele. in array */
int amax(int a[], int count)
{
    int i;
    int result = a[0];
    for (i = 1; i < count; i++)
        if (a[i] > result)
            result = a[i];
    return result;
}

for (init; test; update)
    body

init;
while(test)
    { body ; update }

Annotated Assembly

  # *a in $4, count in $5
  # Use $6 for i, $7 for result
  lw     $7,0($4)       # result = a[0]
  li     $6,0x00000001  # i = 1
  slt    $2,$6,$5       # if !(i < count)
  beq    $2,$0,$L6      # then goto $L6
$L9:
    sll     $2,$6,2
    addu   $2,$2,$4       # $R2 = &a[i]
    lw     $3,0($2)       # $R3 = a[i]
    slt    $2,$7,$3       # if !(result < $R3)
    BEQ    $2,$0,$L7      # then goto $L7
    addu   $6,$6,1        # with count++
    move   $7,$3          # else result = $R3
$L7:
    slt    $2,$6,$5       # if i < count
    bne    $2,$0,$L9      # then goto $L9
$L6:
    J       $31           # return
    move   $2,$7          # with result
Compiling Switch Statements

C Code

typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    case ADD :
      return '+';
    case MULT:
      return '*';
    case MINUS:
      return '-';
    case DIV:
      return '/';
    case MOD:
      return '%';
    case BAD:
      return '?';
  }
}

Implementation Options

- **Series of conditionals**
  - Good if few cases
  - Slow if many

- **Jump Table**
  - Lookup branch target
  - Avoids conditionals
  - Possible when cases are small integer constants

- **GCC**
  - Picks one based on case structure
Switch Statement Example

C Code

typedef enum {
    ADD, MULT, MINUS, DIV, MOD, BAD
} op_type;

char unparse_symbol(op_type op) {
    switch (op) {
    case ADD :
        return '+';
    case MULT:
        return '*';
    case MINUS: 
        return '-';
    case DIV:
        return '/';
    case MOD:
        return '%';
    case BAD:
        return '?';
    }
}

Enumerated Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0</td>
</tr>
<tr>
<td>MULT</td>
<td>1</td>
</tr>
<tr>
<td>MINUS</td>
<td>2</td>
</tr>
<tr>
<td>DIV</td>
<td>3</td>
</tr>
<tr>
<td>MOD</td>
<td>4</td>
</tr>
<tr>
<td>BAD</td>
<td>5</td>
</tr>
</tbody>
</table>

Assembly: Setup

```
# op in $R4
li $2,0x00000004
sltu $2,$2,$4       #if (4 < op)
BNE $2,$0,$L144     #then goto $L144
li $2,0x0000003f    # with ‘?’
sll $2,$4,2
lw $2,$L143($2)     # $2 = &tab[op]
j $2                 # goto $2
```
Jump Table

Table Contents

Enumerated Values

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0</td>
</tr>
<tr>
<td>MULT</td>
<td>1</td>
</tr>
<tr>
<td>MINUS</td>
<td>2</td>
</tr>
<tr>
<td>DIV</td>
<td>3</td>
</tr>
<tr>
<td>MOD</td>
<td>4</td>
</tr>
<tr>
<td>BAD</td>
<td>5</td>
</tr>
</tbody>
</table>

Targets

<table>
<thead>
<tr>
<th>$L137:</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
</tr>
<tr>
<td>$L144</td>
</tr>
<tr>
<td>li</td>
</tr>
<tr>
<td>$2,0x0000002b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$L138:</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
</tr>
<tr>
<td>$L144</td>
</tr>
<tr>
<td>li</td>
</tr>
<tr>
<td>$2,0x0000002a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$L139:</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
</tr>
<tr>
<td>$L144</td>
</tr>
<tr>
<td>li</td>
</tr>
<tr>
<td>$2,0x0000002d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$L140:</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
</tr>
<tr>
<td>$L144</td>
</tr>
<tr>
<td>li</td>
</tr>
<tr>
<td>$2,0x0000002f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$L141:</th>
</tr>
</thead>
<tbody>
<tr>
<td>li</td>
</tr>
<tr>
<td>$2,0x00000025</td>
</tr>
</tbody>
</table>

Completion

<table>
<thead>
<tr>
<th>$L144:</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
</tr>
<tr>
<td>$31</td>
</tr>
<tr>
<td># return</td>
</tr>
</tbody>
</table>


Languages that support recursion
  • E.g.: C, Pascal

Stack Allocated in *Frames*
  • State for procedure invocation
    – Return point, arguments, locals

**Code Example**

```plaintext
yoo(…)
{
  ...
  who();
  ...
}

who(…)
{
  ...
  amI();
  ...
}

amI(…)
{
  ...
  amI();
  ...
}
```

**Call Chain**

```
yoo
  ↓
who
  ↓
amI
  ↓
amI
  ↓
amI
  ↓
  ...
  ...
  ...
  ...
  ...
```
Saving Conventions

When procedure \texttt{yoo} calls \texttt{who}

- \texttt{yoo} is the \textit{caller}
- \texttt{who} is the \textit{callee}

\textbf{Caller Save}

- If \texttt{yoo} wants value in register before & after \texttt{who} returns
  - Save it on the stack before calling \texttt{who}
  - Restore after \texttt{who} returns

\textbf{Callee Save}

- If \texttt{who} wants to use register
  - Save current register value on stack on entry
  - Restore when returning
Saving Examples

**Caller Save**
- Caller must save/restore
- Callee can clobber

```
yoo:
    li $8, 17
    ...
    sw $8, 20($sp)
    jal who
    lw $8, 20($sp)
    ...
    mov $2, $8
    j $31

who:
    li $8, 57
    ...
    j $31
```

**Callee Save**
- Callee must save/restore

```
who:
    sw $16, 20($sp)
    li $16, 57
    ...
    lw $16, 20($sp)
    j $31
```

```
yoo:
    li $16, 17
    ...
    jal who
    ...
    mov $2, $16
    j $31
```

What’s wrong with above code?
MIPS Stack Frame

Conventions

• Agreed by all program/compiler writers
  – Allows linking between different compilers
  – Allows symbolic debugging tools

Run Time Stack

• Save context
  – Registers
• Storage for local variables
• Parameters to called functions
• Required to support recursion

Stack Pointer

 arg 1
 arg 2

 arg n

 Locals & Temporaries
 save reg 1
 save Reg 2

 (Virtual) Frame Pointer

 save reg m
Stack Frame Requirements

Procedure Categories

- **Leaf procedures that do not use stack**
  - Do not call other procedures
  - Can fit all temporaries in caller-save registers

- **Leaf procedures that use stack**
  - Do not call other procedures
  - Need stack for temporaries

- **Non-leaf procedures**
  - Must use stack (to save $31 at the least)

Stack Frame Structure

- **Must be multiple of 8 bytes**
- **Must have space for saving argument registers $4, \ldots, 7**
  - Even if don’t store anything there
  - Used by symbolic debuggers
**Optimized Stack Frame Example**

**C Code**

```c
/* Recursive factorial */
int rfact(int x)
{
    if (x <= 1)
        return 1;
    return x * rfact(x-1);
}
```

**Compiled -O: Frame Setup**

```
rfact:
    .frame $sp,24,$31
    .mask 0x80010000,-4
    .fmask 0x00000000,0
    subu $sp,$sp,24
    sw $31,20($sp)
    sw $16,16($sp)
```

- Stack frame 24 bytes
- Virtual frame ptr at sp + 24
- Save registers $31 and $16
- Starting at sp + 24 - 4
- No floating pt. regs. used
Opt. Stack Frame Example (Cont.)

C Code

/* Recursive factorial */
int rfact(int x)
{
    if (x <= 1)
        return 1;
    return x * rfact(x-1);
}

Compiled -O: Body

move $16,$4  # Save x
slt $2,$16,2  # if x < 2
BNE $2,$0,$L3  # then goto $L3
li $2,0x00000001  # with return val 1
JAL rfact  # Call rfact
subu $4,$16,1  # on x-1
mult $16,$2  # x * return val
mflo $2  # as return val
Opt. Stack Frame Example (Cont.)

C Code

```c
/* Recursive factorial */
int rfact(int x)
{
    if (x <= 1)
        return 1;
    return x * rfact(x-1);
}
```

Compiled -O: Completion

```
$L3:  
    lw  $31,20($sp)
    lw  $16,16($sp)
    addu $sp,$sp,24
    j   $31
```

Stack Pointer

(Virtual)
Frame Pointer

<table>
<thead>
<tr>
<th>Frame Pointer</th>
<th>Stack Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>arg 1</td>
</tr>
<tr>
<td>4</td>
<td>arg 2</td>
</tr>
<tr>
<td>8</td>
<td>arg 3</td>
</tr>
<tr>
<td>12</td>
<td>arg 4</td>
</tr>
<tr>
<td>16</td>
<td>save $16</td>
</tr>
<tr>
<td>20</td>
<td>save $31</td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>
Unopt. Stack Frame Example:

C Code

/* Recursive factorial */
int rfact(int x)
{
    if (x <= 1)
        return 1;
    return x * rfact(x-1);
}

Compiled: Frame Setup

rfact:
    .frame $fp,24,$31
    .mask 0xc0000000,-4
    .fmask 0x00000000,0
    subu $sp,$sp,24
    sw $31,20($sp)
    sw $fp,16($sp)
    move $fp,$sp
    sw $4,24($fp)

Stack Pointer

0  Callee x
4  arg 2
8  arg 3
12 arg 4
16 save $30
20 save $31
24 My x

Never Used

Actual Frame Pointer

(Virtual) Frame Pointer

- Stack frame 24 bytes
- Actual frame pointer $fp uses $30
- Virtual frame ptr at $fp + 24
- Save registers $31 and $30
- Starting at $fp + 24 - 4
- No floating pt. regs. used
Unopt. Stack Frame Example (Cont.)

C Code

/* Recursive factorial */
int rfact(int x)
{
  if (x <= 1)
    return 1;
  return x * rfact(x-1);
}

Compiled: Body

lw $2,24($fp)    # Get argument x
slt $3,$2,2       # If !(x < 2)
beq $3,$0,$L2     # then goto $L2
li $2,0x00000001 # return val 1
j $L1            # goto $L1
$L2:
lw $3,24($fp)    # Get argument x
subu $2,$3,1     # Subtract 1
move $4,$2       # assign to arg1
jal rfact        # Call rfact
lw $3,24($fp)    # Get argument x
mult $2,$3       # x * return val
mflo $2          # is return val
j $L1

Stack Pointer

Actual Frame Pointer

(Virtual) Frame Pointer

Callee x
arg 2
arg 3
arg 4
save $30
save $31
My x

Never Used

0
4
8
12
16
20
24

Frame Pointer
Unopt. Stack Frame Example (Cont.)

C Code

/* Recursive factorial */
int rfact(int x)
{
    if (x <= 1)
        return 1;
    return x * rfact(x-1);
}

Compiled: Completion

move       $sp,$fp
lw         $31,20($sp)
lw         $fp,16($sp)
addu       $sp,$sp,24
j           $31
Local Storage Example

C Code

```c
char *stuff_it
    (char c0, char c1, char c2,
     char c3, char c4, char c5,
     char c6)
{
    char buf[7]; /* Not enough! */
    int i = 0;
    char *result;
    buf[i++] = c0;
    buf[i++] = c1;
    buf[i++] = c2;
    buf[i++] = c3;
    buf[i++] = c4;
    buf[i++] = c5;
    buf[i++] = c6;
    buf[i++] = 0;
    result = (char *) malloc(i);
    strcpy(result, buf);
    return result;
}

char *do_it()
{
    return stuff_it
        ('1','5','-','3','4','7','!');
}
```
Local Storage Example

C Code: Initial

```c
char *stuff_it
    (char c0, char c1, char c2,
     char c3, char c4, char c5,
     char c6)
{
    char buf[7]; /* Not enough! */
    int i = 0;
    char *result;
}
```

Compiled -O: Frame Setup

```
stuff_it:
    .frame  $sp,40,$31
    .mask   0x80030000,-8
    .fmask  0x00000000,0
    .subu   $sp,$sp,40
    sw      $31,32($sp)
    sw      $17,28($sp)
    sw      $16,24($sp)
```

- Stack frame 40 bytes
- Virtual frame ptr at $sp + 40
- Save registers $31, $17, $16
  - Starting at $sp + 40 - 8
- No floating pt. regs. used
Local Storage Example (Cont.)

C Code: Argument Retrieval

char *stuff_it
    (char c0, char c1, char c2,
     char c3, char c4, char c5,
     char c6)

Compiled -O: Args 5–7

lbu   $2,56($sp)
lbu   $3,60($sp)
lbu   $8,64($sp)

• c0 … c3 passed in registers $4 … $7
Local Storage Example (Cont.)

C Code: Local Setup

```c
buf[i++] = c0;
b buf[i++] = c1;
buf[i++] = c2;
b buf[i++] = c3;
buf[i++] = c4;
b buf[i++] = c5;
buf[i++] = c6;
b buf[i++] = 0;
```

Compiled -O

```assembly
addu    $17,$sp,16  # $17 = &buf
sb      $4,16($sp)  # buf[0] = c0
sb      $5,17($sp)  # buf[1] = c1
sb      $6,18($sp)  # buf[2] = c2
sb      $7,19($sp)  # buf[3] = c3
sb      $0,23($sp)  # buf[7] = 0  !!
sb      $2,20($sp)  # buf[4] = c4
sb      $3,21($sp)  # buf[5] = c5
sb      $8,22($sp)  # buf[6] = c6
```
Local Storage Example (Cont.)

C Code: Calls

```c
result = (char *) malloc(i);
strcpy(result, buf);
```

Compiled -O

```assembly
li     $4,0x00000008  # i = 8
jal    malloc         # malloc(i)
move   $16,$2         # $16 = result
move   $4,$16
JAL    strcpy         # strcpy(result, , buf)
move   $5,$17
move   $2,$16         # return val result
```
Local Storage Example (Cont.)

Completion

| lw $31,32($sp) |
| lw $17,28($sp) |
| lw $16,24($sp) |
| addu $sp,$sp,40 |
| j $31 |

Stack Pointer

Never Used

(Virtual)
Frame Pointer

arg 1
arg 2
arg 3
arg 4
buf
[0] [1] [2] [3]
save $16
save $17
save $31

[0x0] [2] [0x0] [3]
[0x0] [4] [0x0] [6]
Basic Data Types

Integral

- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

<table>
<thead>
<tr>
<th>MIPS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1</td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>half word</td>
<td>2</td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>word</td>
<td>4</td>
<td>[unsigned] [long] int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pointer</td>
</tr>
</tbody>
</table>

Floating Point

- Stored & operated on in floating point registers
- Special instructions for two different precisions

<table>
<thead>
<tr>
<th>MIPS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>double precision</td>
<td>8</td>
<td>double</td>
</tr>
</tbody>
</table>
Floating Point Unit

Implemented as Coprocessor 1
- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

Floating Point Values
- Single precision (C `float`): 32 bits
- Double precision (C `double`): 64 bits

Floating Point Data Registers
- 32 registers, each 4 bytes
- Allocated in pairs
  - Even for single precision

```
<table>
<thead>
<tr>
<th>$f0</th>
<th>$f1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f2</td>
<td>$f3</td>
</tr>
<tr>
<td>$f4</td>
<td>$f5</td>
</tr>
<tr>
<td>$f6</td>
<td>$f7</td>
</tr>
<tr>
<td>$f8</td>
<td>$f9</td>
</tr>
<tr>
<td>$f10</td>
<td>$f11</td>
</tr>
<tr>
<td>$f12</td>
<td>$f13</td>
</tr>
<tr>
<td>$f14</td>
<td>$f15</td>
</tr>
<tr>
<td>$f16</td>
<td>$f17</td>
</tr>
<tr>
<td>$f18</td>
<td>$f19</td>
</tr>
<tr>
<td>$f20</td>
<td>$f21</td>
</tr>
<tr>
<td>$f22</td>
<td>$f23</td>
</tr>
<tr>
<td>$f24</td>
<td>$f25</td>
</tr>
<tr>
<td>$f26</td>
<td>$f27</td>
</tr>
<tr>
<td>$f28</td>
<td>$f29</td>
</tr>
<tr>
<td>$f30</td>
<td>$f31</td>
</tr>
</tbody>
</table>
```

Return Values
Caller Save Temporaries: May be overwritten by called procedures
Procedure arguments
Caller Save Temporaries:
Callee Save Temporaries: May not be overwritten by called procedures

Temporaries:
- May be overwritten by called procedures
- May not be overwritten by called procedures
Floating Point Code Example

Compute Inner Product of Two Vectors

• Single precision

```c
float inner_prodF
    (float x[], float y[], int n)
{
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++) {
        result += x[i] * y[i];
    }
    return result;
}
```

```
intra_prodF:
    subu $sp,$sp,8     # Allocate stack frame
    mtc1 $0,$f4        # result = 0.0
    BLEZ $6,$L3        # if n <= 0 goto $L3
    move $3,$0          # i = 0
$L5:
    l.s $f0,0($4)       # $f0 = x[i]
    l.s $f2,0($5)       # $f2 = y[i]
    mul.s $f0,$f0,$f2   # $f0 = x[i] * y[i]
    addu $5,$5,4        # y+i++
    add.s $f4,$f4,$f0   # result += $f0
    addu $3,$3,1        # i++
    slt $2,$3,$6        # if i < n
    BNE $2,$0,$L5       # then loop
    addu $4,$4,4        # with x+i++
$L3:
    mov.s $f0,$f4       # return val = result
    addu $sp,$sp,8      # Deallocate frame
    j $31                # return
```
double inner_prodD
    (double x[], double y[],
     int n)
{
    int i;
    double result = 0.0;
    for (i = 0; i < n; i++) {
        result += x[i] * y[i];
    }
    return result;
}

inner_prodD:
    # result in $f4,5
    mtc1 $0,$f4    # result = 0.0
    mtc1 $0,$f5
    subu $sp,$sp,8 # allocate stack frame
    BLEZ $6,$L8    # if n <= 0 goto $L8
    move $3,$0     # i = 0
$L10:
    1.d $f2,0($4) # $f2,3 = x[i]
    1.d $f0,0($5) # $f0,1 = y[i]
    mul.d $f2,$f2,$f0 # $f2,3 *= y[i]
    addu $5,$5,8   # y+i++
    add.d $f4,$f4,$f2 # result *= $f2,f3
    addu $3,$3,1    # i++
    slt $2,$3,$6    # if i < n
    BNE $2,$0,$L10 # then loop
    addu $4,$4,8    # with x+i++
$L8:
    mov.d $f0,$f4    # return val = result
    addu $sp,$sp,8   # deallocate frame
    j $31          # return
Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Operation</th>
<th>Destination, Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40</td>
<td>44802000</td>
<td>mtcl</td>
<td>r0,$f4</td>
</tr>
<tr>
<td>0x44</td>
<td>44802800</td>
<td>mtcl</td>
<td>r0,$f5</td>
</tr>
<tr>
<td>0x48</td>
<td>27bdff8</td>
<td>addiu</td>
<td>sp,sp,-8</td>
</tr>
<tr>
<td>0x4c</td>
<td>18c0000c</td>
<td>BLEZ</td>
<td>r6,0x80</td>
</tr>
<tr>
<td>0x50</td>
<td>00001821</td>
<td>move</td>
<td>r3,r0</td>
</tr>
<tr>
<td>0x54</td>
<td>c4820000</td>
<td>lwcl</td>
<td>$f2,0(r4)</td>
</tr>
<tr>
<td>0x58</td>
<td>c4830004</td>
<td>lwcl</td>
<td>$f3,4(r4)</td>
</tr>
<tr>
<td>0x5c</td>
<td>c4a00000</td>
<td>lwcl</td>
<td>$f0,0(r5)</td>
</tr>
<tr>
<td>0x60</td>
<td>c4a10004</td>
<td>lwcl</td>
<td>$f1,4(r5)</td>
</tr>
<tr>
<td>0x64</td>
<td>24630001</td>
<td>addiu</td>
<td>r3,r3,1</td>
</tr>
<tr>
<td>0x68</td>
<td>46201082</td>
<td>mul.d</td>
<td>$f2,$f2,$f0</td>
</tr>
<tr>
<td>0x6c</td>
<td>0066102a</td>
<td>slt</td>
<td>r2,r3,r6</td>
</tr>
<tr>
<td>0x70</td>
<td>24a50008</td>
<td>addiu</td>
<td>r5,r5,8</td>
</tr>
<tr>
<td>0x74</td>
<td>46222100</td>
<td>add.d</td>
<td>$f4,$f4,$f2</td>
</tr>
<tr>
<td>0x78</td>
<td>1440fff6</td>
<td>BNE</td>
<td>r2,r0,0x54</td>
</tr>
<tr>
<td>0x7c</td>
<td>24840008</td>
<td>addiu</td>
<td>r4,r4,8</td>
</tr>
<tr>
<td>0x80</td>
<td>46202006</td>
<td>mov.d</td>
<td>$f0,$f4</td>
</tr>
<tr>
<td>0x84</td>
<td>03e00008</td>
<td>JR</td>
<td>r31</td>
</tr>
<tr>
<td>0x88</td>
<td>27bd0008</td>
<td>addiu</td>
<td>sp,sp,8</td>
</tr>
</tbody>
</table>
Structure Allocation

Principles

- Allocate space for structure elements contiguously
- Access fields by offsets from initial location
  - Offsets determined by compiler

```c
typedef struct {
    char c;
    int i[2];
    double d;
} struct_ele, *struct_ptr;
```

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>i[0]</td>
<td>i[1]</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>
Alignment

Requirements

- Data type requires $K$ bytes
- Address must be multiple of $K$

Specific Cases

- Word data address must be multiple of 4
- Doubleword data address must be multiple of 8

Reason

- Memory accessed by (aligned) words
  - Inefficient to load or store data that spans word boundaries
  - Virtual memory very tricky when datum spans 2 pages

Compiler

- Inserts gaps within structure to ensure correct alignment of fields
## Structure Access

### C Code

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>char struct_c(struct_ptr p)</td>
<td>{ return p-&gt;c; }</td>
</tr>
<tr>
<td>int *struct_i(struct_ptr p)</td>
<td>{ return p-&gt;i; }</td>
</tr>
<tr>
<td>int struct_i1(struct_ptr p)</td>
<td>{ return p-&gt;i[1]; }</td>
</tr>
<tr>
<td>double struct_d(struct_ptr p)</td>
<td>{ return p-&gt;d; }</td>
</tr>
</tbody>
</table>

### Assembler

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lb $2,0($4)</td>
<td></td>
<td># 1st byte</td>
</tr>
<tr>
<td>j</td>
<td>$31</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>$31</td>
<td></td>
</tr>
<tr>
<td>addu $2,$4,4</td>
<td></td>
<td># addr of 4th byte</td>
</tr>
<tr>
<td>lw $2,8($4)</td>
<td></td>
<td># word at 8th byte</td>
</tr>
<tr>
<td>j</td>
<td>$31</td>
<td></td>
</tr>
<tr>
<td>l.d $f0,16($4)</td>
<td></td>
<td># dblewd at 16th by.</td>
</tr>
</tbody>
</table>
Arrays vs. Pointers

Recall

- Can access stored data either with pointer or array notation
- Differ in how storage allocated
  - Array declaration allocates space for array elements
  - Pointer declaration allocates space for pointer only

```c
typedef struct {
    char c;
    int *i;
    double d;
} pstruct_ele,
*pstruct_ptr;
```

```c
pstruct_ptr pstruct_alloc(void)
{
    pstruct_ptr result = (pstruct_ptr)
        malloc(sizeof(pstruct_ele));
    result->i = (int *)
        calloc(2, sizeof(int));
    return result;
}
```

C Code for Allocation
### Accessing Through Pointer

#### C Code

```c
int *pstruct_i(pstruct_ptr p)
{
    return p->i;
}
```

```c
int pstruct_i1(pstruct_ptr p)
{
    return p->i[1];
}
```

#### Assembler

```
lw     $2,4($4)    # word at 4th byte from p
j       $31
```

```
# i = word at 4th byte from p
lw      $2,4($4)
# Retrieve word at 4th byte from i
lw      $2,4($2)
j       $31
```
Arrays

Principles

• Allocated by repeating allocation for array type
• Accessed by computing address of element
  – Attempt to optimize
    » Minimize use of multiplication
    » Exploit values determined at compile time

C Code

```c
/* Index into array of struct_ele's */
struct_ptr a_index
  (struct_ele a[], int idx)
{
  return &a[idx];
}
```

Assembler

```
sll $2,$5,1
addu $2,$2,$5
sll $2,$2,3  # $2 = 24 * idx
J $31        # return
    # with a + (scaled) idx
addu $2,$4,$2
```
Aligning Array Elements

Requirement
- Must make sure alignment requirements met when allocate array of structures
- May require inserting unused space at end of structure

typedef struct {
    double d;
    int i[2];
    char c;
} rev_ele, *rev_ptr;
Nested Allocations

Principles

- Can nest declarations of arrays and structures
- Compiler keeps track of allocation and access requirements

```c
typedef struct {
    int x;
    int y;
} point_ele, *point_ptr;

typedef struct {
    point_ele ll;
    point_ele ur;
} rect_ele, *rect_ptr;
```

<table>
<thead>
<tr>
<th>ll.x</th>
<th>ll.y</th>
<th>ur.x</th>
<th>ur.y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Nested Allocation (cont.)

C Code

```c
int area(rect_ptr r)
{
    int width = r->ur.x - r->ll.x;
    int height = r->ur.y - r->ll.y;
    return width * height;
}
```

Assembler

```assembly
lw $2,8($4)   # $2 = ur.x
lw $3,0($4)   # $3 = ll.x
subu $2,$2,$3 # $2 = width
lw $3,12($4)  # $3 = ur.y
lw $4,4($4)   # $4 = ll.y
subu $3,$3,$4 # $3 = height
mult $2,$3
mflo $2      # width * height
j $31
```
Union Allocation

Principles

- Overlay union elements
- Allocate according to largest element
- Programmer responsible for collision avoidance

typedef union {
    char c;
    int i[2];
    double d;
} union_ele, *union_ptr;
Byte Ordering

Idea

- Bytes in word numbered 0 to 3
- Which is most (least) significant?
- Can cause problems when exchanging binary data between machines

Big Endian

- Byte 0 is most, 3 is least
- IBM 360/370, Motorola 68K, Sparc

Little Endian

- Byte 0 is least, 3 is most
- Intel x86, VAX

MIPS

- Can be configured either way
- DECStation’s are little endian
Byte Ordering Example

union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned long l[2];
} dw;
int i;
for (i = 0; i < 8; i++)
    dw.c[i] = 0xf0 + i;
printf("Characters 0-7 == [0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    (long) dw.c[0], (long) dw.c[1], (long) dw.c[2], (long) dw.c[3],
    (long) dw.c[4], (long) dw.c[5], (long) dw.c[6], (long) dw.c[7]);
printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    (long) dw.s[0], (long) dw.s[1], (long) dw.s[2], (long) dw.s[3]);
printf("Longs 0-1 == [0x%x,0x%x]\n",
    dw.l[0], dw.l[1]);
### Byte Ordering Example (cont.)

#### Big Endian

<table>
<thead>
<tr>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
<th>f7</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
<th>MSB</th>
<th>LSB</th>
<th>MSB</th>
<th>LSB</th>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>l[0]</td>
<td></td>
<td>l[1]</td>
<td></td>
</tr>
</tbody>
</table>

#### Output on Sun:

- **Characters 0–7 ==** [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- **Shorts 0–3 ==** [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7 ]
- **Longs 0–1 ==** [0xf0f1f2f3, 0xf4f5f6f7 ]
Byte Ordering Example (cont.)

Little Endian

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>l[0]</th>
<th>l[1]</th>
</tr>
</thead>
</table>

Output on DecStation:

Characters 0–7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0–3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
Longs 0–1 == [0xf3f2f1f0, 0xf7f6f5f4]
MIPS Memory Layout

Segments

- **Stack**
  - Grows down from top
  - Implements runtime stack
- **Data**
  - Static space for global variables
    » Allocation determined at compile time
  - Dynamic space for runtime allocation
    » E.g., using malloc
- **Text**
  - Stores machine code for program
- **Reserved**
  - Used by operating system
    » I/O devices, process info, etc.
Instruction Formats

Arithmetic

• addu Rd, Rs, Rt
• Similar for shift, Booleans, and comparisons

Constant-Manipulating

• addi Rs, Rt, imm
• Only have 16-bit constant

Jump

• j offset

• beq Rs, Rt, offset
• PC relative

Load/Store

• lw Rt, Offset(Rs)
Specifying Jump Targets

**Jump (and link)**

\[ j \ target \]

**Jump (and link) Register**

\[ jr \ Rs \]

- Rs holds dest.

High order bits of slot PC

Low order two bits must be 0
Specifying Branch Targets

Branch

\[ \text{beq Rs, Rt, offset} \]

\[
\begin{array}{cccc}
6 & 5 & 5 & 16 \\
4 & Rs & Rt & Offset \\
\end{array}
\]

\[
\begin{array}{ccc}
14 & 16 & 2 \\
Offset & Sign & Offset & 00 \\
\end{array}
\]

+ Delay Slot PC

\[
\begin{array}{c}
\text{Dest.} \\
\end{array}
\]
RISC Principles Summary

Simple & Regular Instructions
- Small number of uniform formats
- Each operation does just one thing
  - Memory access, computation, conditional, etc.

Encourage Register Usage over Memory
- Operate on register data
  - Load/store architecture
- Procedure linkage

Rely on Optimizing Compiler
- Data allocation & referencing
- Register allocation
- Improve efficiency of user’s code
- Dealing with implementation artifacts
  - e.g., branch delay slots