Machine-Level Programming: Control Flow Floating Point

CS 347

Jan. 23, 1996

Topics

- Branches and loops
- Jump tables
- MIPS integer multiplication
- Instruction encoding

MIPS Register Convention

Register	Name	Use
$0	$0	Constant 0
$1	$at	Reserved Temp.
$2	$v0	Return Values
$3	$v1	
$4	$a0	Procedure arguments
$5	$a1	
$6	$a2	
$7	$a3	
$8	$t0	Caller Save Temporaries: May be overwritten by called procedures
$9	$t1	
$10	$t2	
$11	$t3	
$12	$t4	
$13	$t5	
$14	$t6	
$15	$t7	
$16	$s0	Callee Save Temporaries: May not be overwritten by called procedures
$17	$s1	
$18	$s2	
$19	$s3	
$20	$s4	
$21	$s5	
$22	$s6	
$23	$s7	
$24	$t8	Caller Save Temp
$25	$t9	
$26	$k0	Reserved for Operating Sys
$27	$k1	
$28	$gp	Global Pointer
$29	$sp	Stack Pointer
$30	$s8	Callee Save Temp
$31	$ra	Return Address

MIPS Implemented Branches

Formats

- **`b`*`COND`* `Rs, Rt,` *`label`***
  - *`Cond`* : Branch condition
- **`b`*`COND`* `Rs,` *`label`***
  - *`Cond`* : Branch condition, relative to zero

Conditions

Conditions		Reg. Arguments
`beq`	**EQual**	**2**
`bne`	**Not Equal**	**2**
`bgez`	**Greater or Equal to Zero**	**1**
`bgtz`	**Greater Than Zero**	**1**
`blez`	**Less or Equal to Zero**	**1**
`bltz`	**Less Than Zero**	**1**

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

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

Compiled

```
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

```
0x400230 <max>:        slt $v0,$a0,$a1
0x400234 <max+4>:      BEQ $v0,$zero,0x400240 <max+16>
0x400238 <max+8>:      nop
0x40023c <max+12>:     move $a0,$a1
0x400240 <max+16>:     JR $ra
0x400244 <max+20>:     move $v0,$a0
```

Loop Example

C Code

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

Handwritten 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`

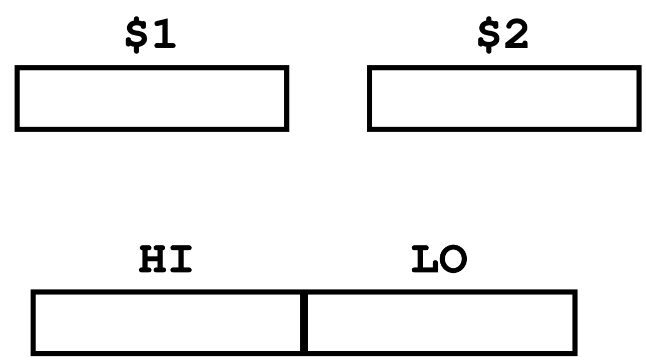
`mul $Rdest, $Rsrc`

`$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!**
  - Approx. 18 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
```

Disassembled

```
0x400ba0 <hfact>:       slti $at,$a0,2
0x400ba4 <hfact+4>:     BNE $at,$zero,0x400bc4 <hfact+36>
0x400ba8 <hfact+8>:     li $v0,1
0x400bac <hfact+12>:    multu $v0,$a0
0x400bb0 <hfact+16>:    addi $a0,$a0,-1
0x400bb4 <hfact+20>:    slti $at,$a0,2
0x400bb8 <hfact+24>:    mflo $v0
0x400bbc <hfact+28>:    BEQ $at,$zero,0x400bac <hfact+12>
0x400bc0 <hfact+32>:    nop
0x400bc4 <hfact+36>:    JR $ra
0x400bc8 <hfact+40>:    nop
```

Compiled Loop Code

C Code

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

Generated Assembly

```
fact:
    # x in $4, result in $3
    slt     $2,$4,2          # if x < 2
    BNE     $2,$0,$L3        # then goto exit
    li      $3,0x00000001    # with $3 = 1
    mult    $3,$4            # mult. result * x
$L5:
    mflo    $3               # Update result
    subu    $4,$4,1          # x--
    slt     $2,$4,2          # if x >= 2
    BEQ     $2,$0,$L5        # then goto loop
    mult    $3,$4            # *'ing result *x
$L3:
    J       $31              # return
    move    $2,$3            # with result
```

Compiled Loop Code

Disassembled

```
0x400230 <fact>:        slti $v0,$a0,2
0x400234 <fact+4>:      BNE $v0,$zero,0x400254 <fact+36>
0x400238 <fact+8>:      li $v1,1
0x40023c <fact+12>:     mult $v1,$a0
0x400240 <fact+16>:     mflo $v1
0x400244 <fact+20>:     addiu $a0,$a0,-1
0x400248 <fact+24>:     slti $v0,$a0,2
0x40024c <fact+28>:     BEQ $v0,$zero,0x400240 <fact+16>
0x400250 <fact+32>:     mult $v1,$a0
0x400254 <fact+36>:     JR $ra
0x400258 <fact+40>:     move $v0,$v1
```

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

ADD	0
MULT	1
MINUS	2
DIV	3
MOD	4
BAD	5

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

```
$L143:
        .word   $L137
        .word   $L138
        .word   $L139
        .word   $L140
        .word   $L141
```

Enumerated Values

```
ADD      0
MULT     1
MINUS    2
DIV      3
MOD      4
BAD      5
```

Targets

```
$L137:
   J    $L144            # goto $L144
   li   $2,0x0000002b    # with '+'
$L138:
   J    $L144            # goto $L144
   li   $2,0x0000002a    # with '*'
$L139:
   J    $L144            # goto $L144
   li   $2,0x0000002d    # with '-'
$L140:
   J    $L144            # goto $L144
   li   $2,0x0000002f    # with '/'
$L141:
   li   $2,0x00000025    # $2 = '%'
```

Completion

```
$L144:
        j       $31   # return
```

Instruction Formats

Arithmetic

- `addu Rd, Rs, Rt`
- Similar for shift, Booleans, and comparisons

6	5	5	5	5	6
0	Rs	Rt	Rd	0	33

Constant-Manipulating

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

6	5	5	16
8	Rs	Rt	imm

Jump / Branch

- `j offset`

6	26
2	offset

- `beq Rs, Rt, offset`
- PC relative

6	5	5	16
4	Rs	Rt	Offset

Load/Store

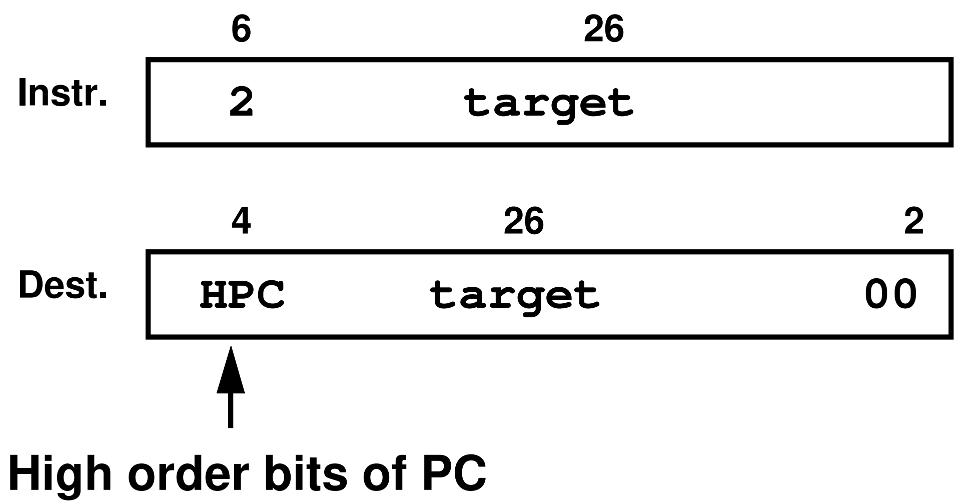
- `lw Rt, Offset(Rs)`

6	5	5	16
35	Rs	Rt	Offset

Specifying Jump Targets

Jump (and link)

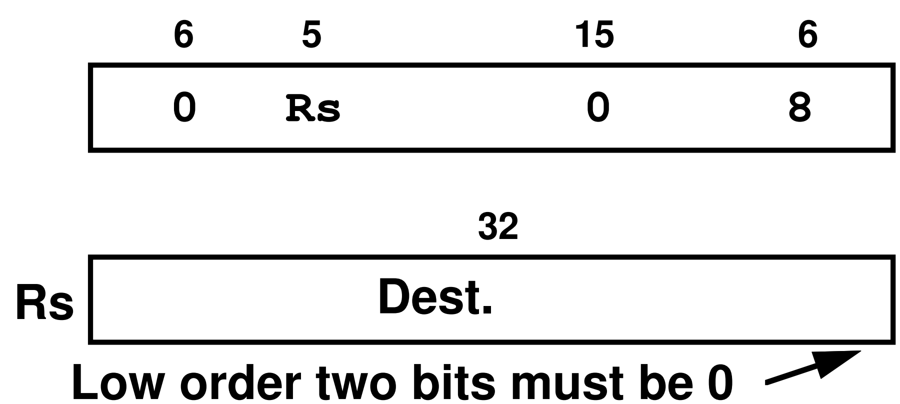
```
j target
```



Jump (and link) Register

```
jr Rs
```

- Rs holds dest.



Specifying Branch Targets

Branch

```
beq Rs, Rt, offset
```

6	5	5	16
`4`	`Rs`	`Rt`	`Offset`

14	16	2
`Offset Sign`	`Offset`	`00`

\+ Delay Slot PC

Dest.