

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 ≈ 1.0
 - Common instructions execute in one clock cycle

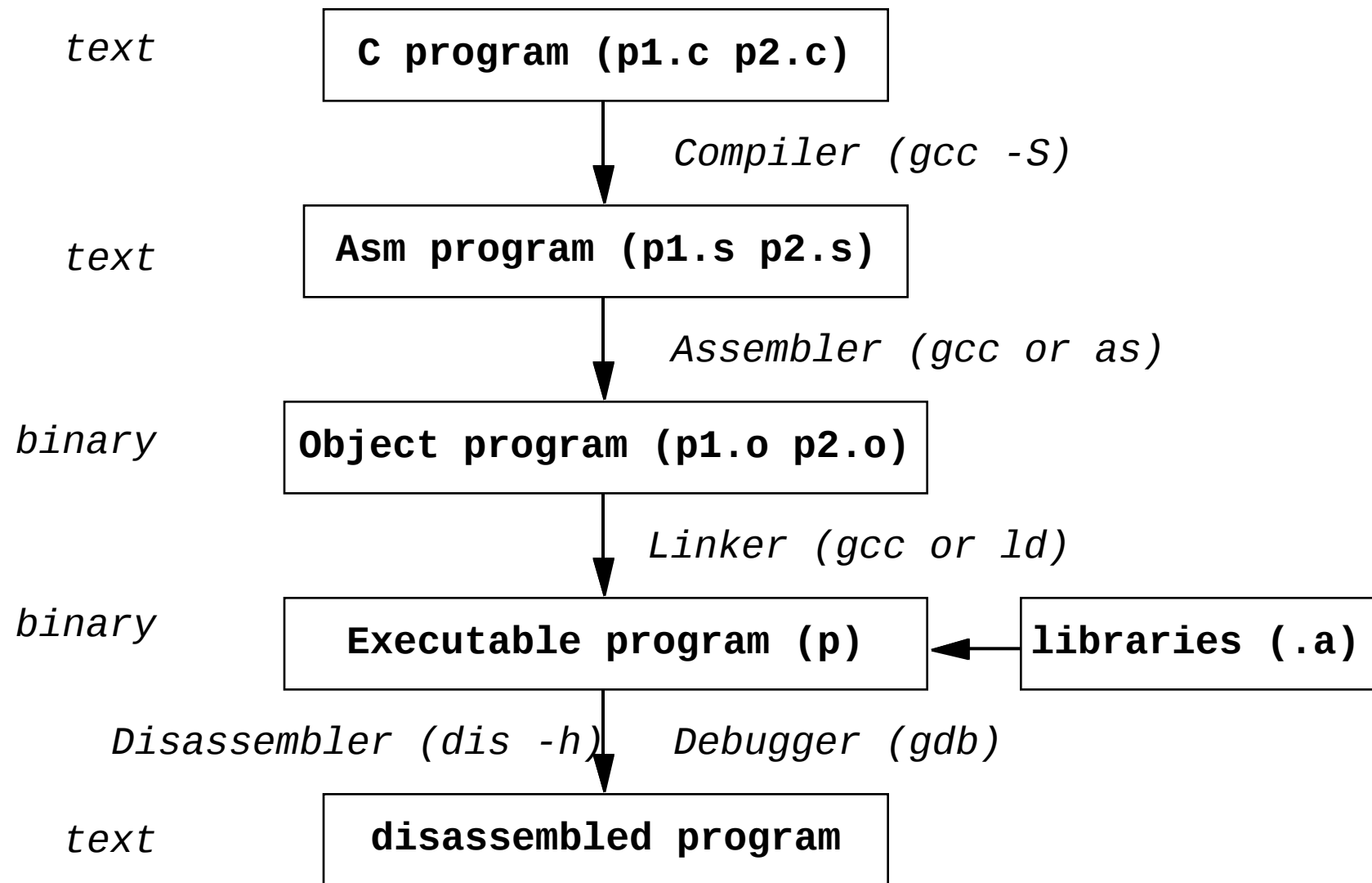
Assumes compiler will do optimizations

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

Widespread usage

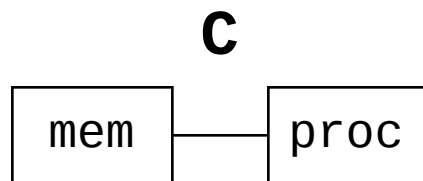
- Commercially
 - SGI, Sony Playstation, other embedded systems
- At CMU
 - DECStations (Including unix servers)
 - SGI Indigos

Translation Process



Abstract machines

Machine Model



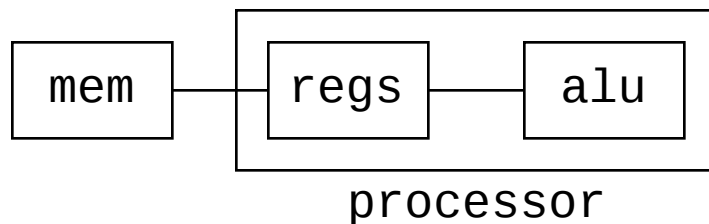
Data

- 1) char
- 2) int, float
- 3) double
- 4) struct, array
- 5) pointer

Control

- 1) loops
- 2) conditionals
- 3) goto
- 4) Proc. call
- 5) Proc. return

ASM



- 1) byte
 - 2) word
 - 3) doubleword
 - 4) contiguous word allocation
 - 5) address of initial byte
- 3) branch/jump
 - 4) jump & link

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

\$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	

Registers (cont.)

Important Ones for Now

- \$0** Constant 0
- \$2** Return Value
- \$3** Can use as temporary
- \$4** First argument
- \$5** Second argument
- \$31** Return address

\$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	Caller Save Temp
\$24	\$t8	
\$25	\$t9	Reserved for Operating Sys
\$26	\$k0	
\$27	\$k1	Global Pointer
\$28	\$gp	
\$29	\$sp	Stack Pointer
\$30	\$s8	Callee Save Temp
\$31	\$ra	Return Address

Program Representations

C Code

```
int gval;

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

Obtain with command

gcc -O -S code.c

Produces file **code.s**

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,$2,$5
                subu       $3,$3,$2
                sw          $3,gval
                j           $31
                .end      test1
```

Prog. Representation (Cont.)

Object

```
0x400230 <test1>:  
    0x00841821  
    0x00a51021  
    0x00451021  
    0x00641821  
    0x00621823  
    0x03e00008  
    0xaf838108
```

Disassembled

```
0x400230 <test1>:      addu $v1,$a0,$a0  
0x400234 <test1+4>:    addu $v0,$a1,$a1  
0x400238 <test1+8>:    addu $v0,$v0,$a1  
0x40023c <test1+12>:   addu $v1,$v1,$a0  
0x400240 <test1+16>:   subu $v1,$v1,$v0  
0x400244 <test1+20>:   jr $ra  
0x400248 <test1+24>:   sw $v1,-32504($gp)
```

Run gdb on object code

x/7 0x400230

– Print 7 word in hexadecimal starting at address 0x400230

disassemble 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

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

Normal jump

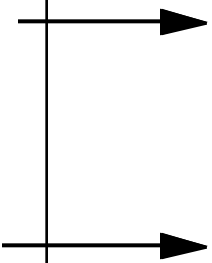
- No code after jump executed

Delayed jump

- Execute instruction following jump
- Strange semantics, efficient implementation

Compiled to Assembly

```
test2:      .loc      1 12
            .ent      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

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

Annotated 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

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

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

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

```
int garray[10];

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

Disassembled:

Annotated 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

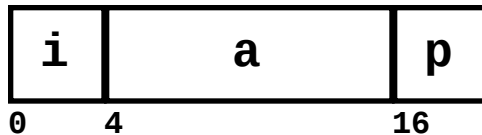
gref:
    sll    $4,$4,2      # $4 = 4*i
    lw     $2,garray($4) #Return val = garray[i]
    j      $31         # Return
```

```
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
```

Structures & Pointers

C Code

```
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    $2,8($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

- **bCOND Rs, Rt, *label***
 - *Cond* : Branch condition
- **bCOND Rs, *label***
 - *Cond* : Branch condition, relative to zero

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

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

```
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

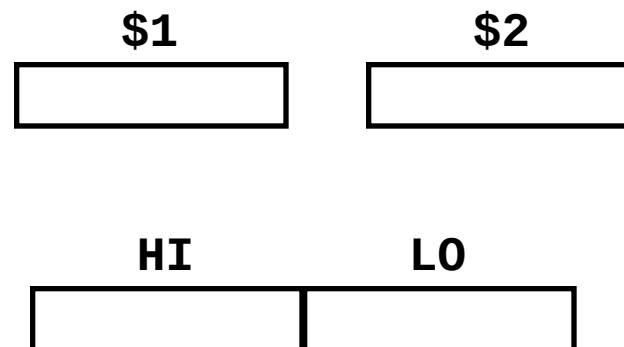
mul \$Rdest, \$Rsrc

\$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!**
 - 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:	1020ffffb	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

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
```

Stack-Based Languages

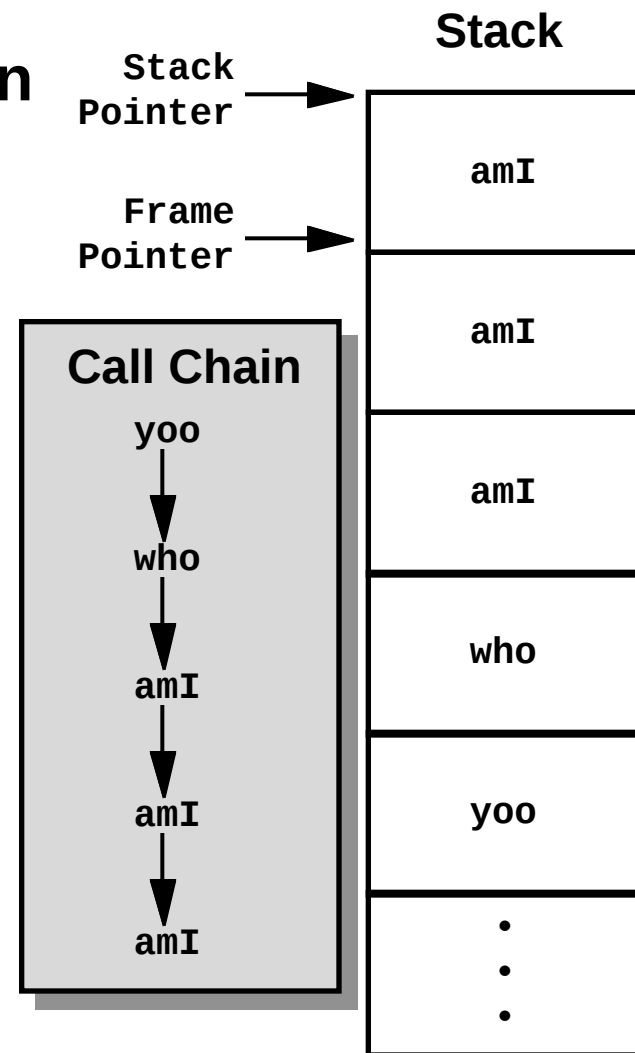
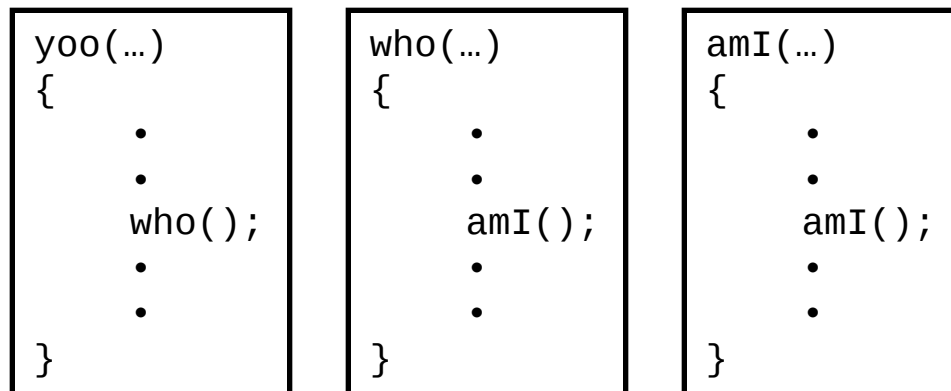
Languages that support recursion

- E.g.: C, Pascal

Stack Allocated in *Frames*

- State for procedure invocation
 - Return point, arguments, locals

Code Example



Saving Conventions

When procedure yoo calls who

- yoo is the *caller*
- who is the *callee*

Caller Save

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

Callee Save

- If 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

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

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

What's wrong with above code?

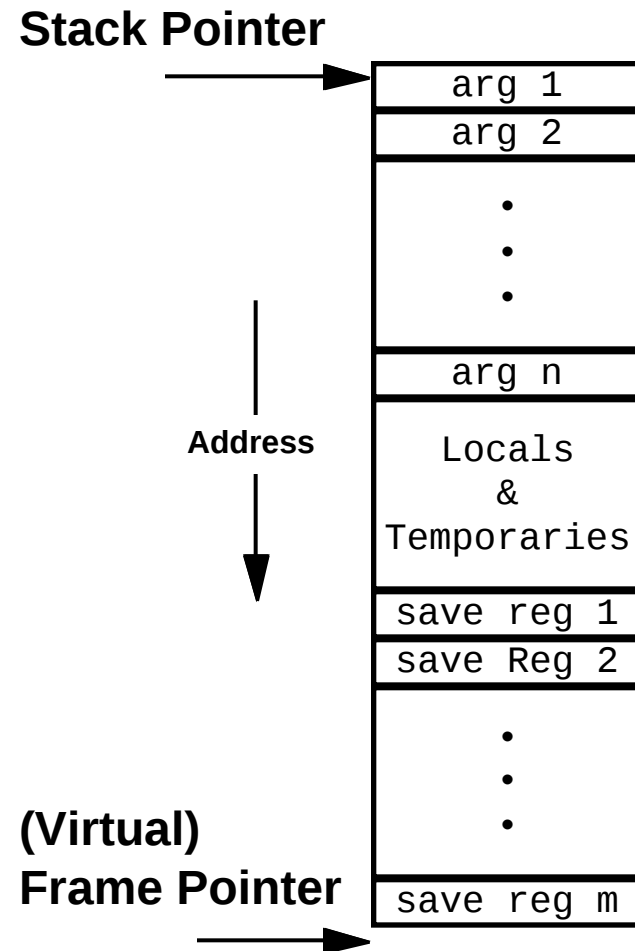
MIPS Stack Frame

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 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, ..., \$7**
 - Even if don't store anything there
 - Used by symbolic debuggers

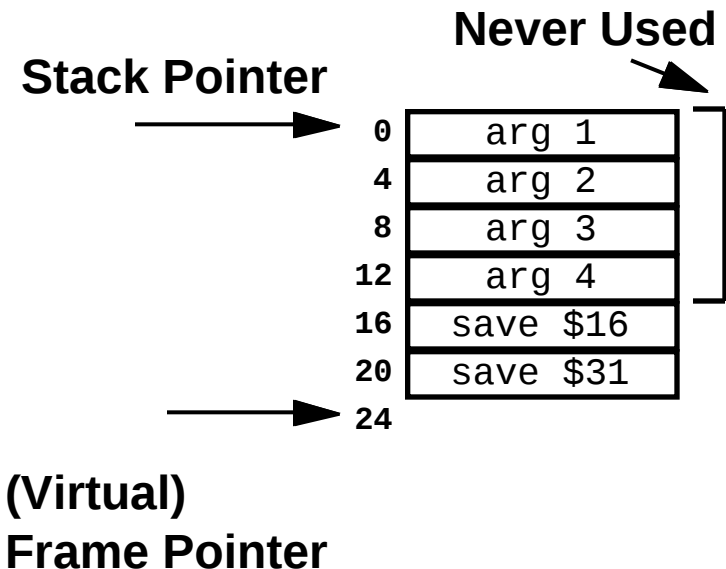
Optimized Stack Frame Example

C Code

```
/* 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)
```

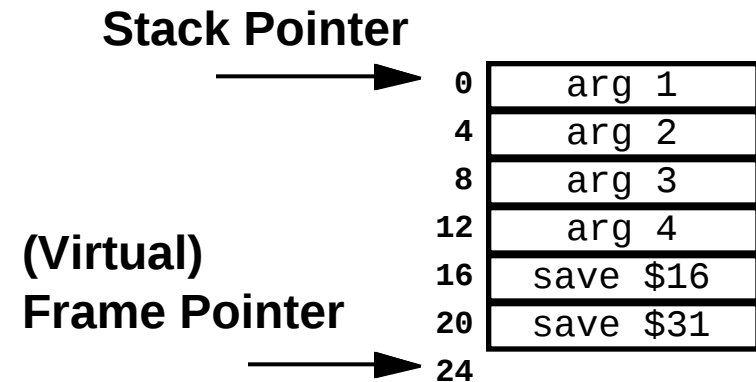


- Stack frame 24 bytes
- Virtual frame ptr at $sp + 24$
- Save registers \$31 and \$16
- Starting at $sp + 24 - 4$
- No floating pt. regs. used

Opt. Stack Frame Example (Cont.)

C Code

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



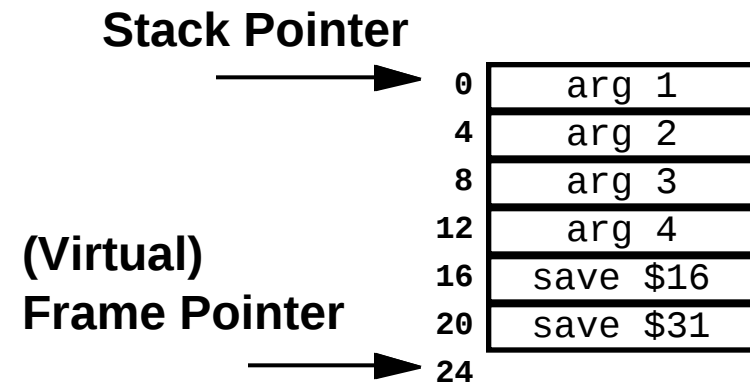
Compiled -O: Body

```
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

```
/* 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
```

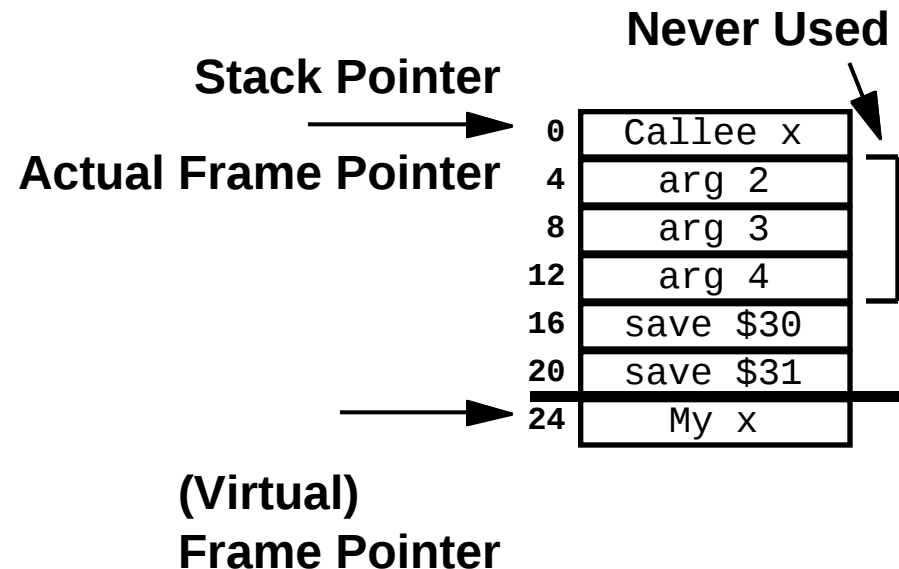
Unopt. Stack Frame Example:

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 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

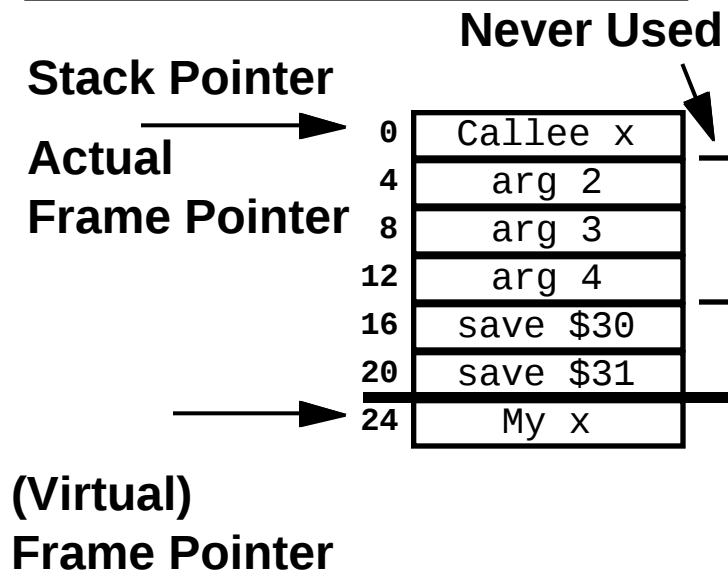
Unopt. Stack Frame Example (Cont.)

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
```



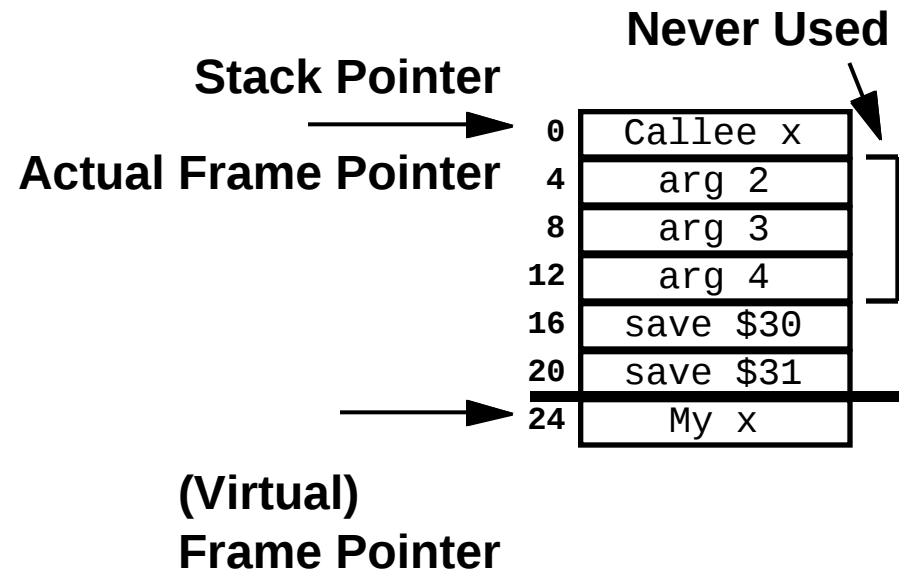
Unopt. Stack Frame Example (Cont.)

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

```
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', '!');
}
```

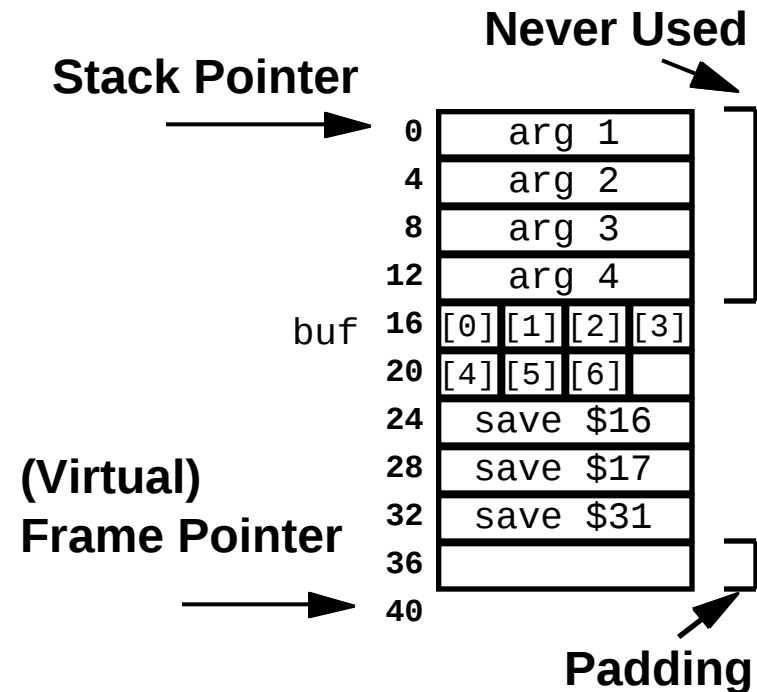
Local Storage Example

C Code: Initial

```
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.)

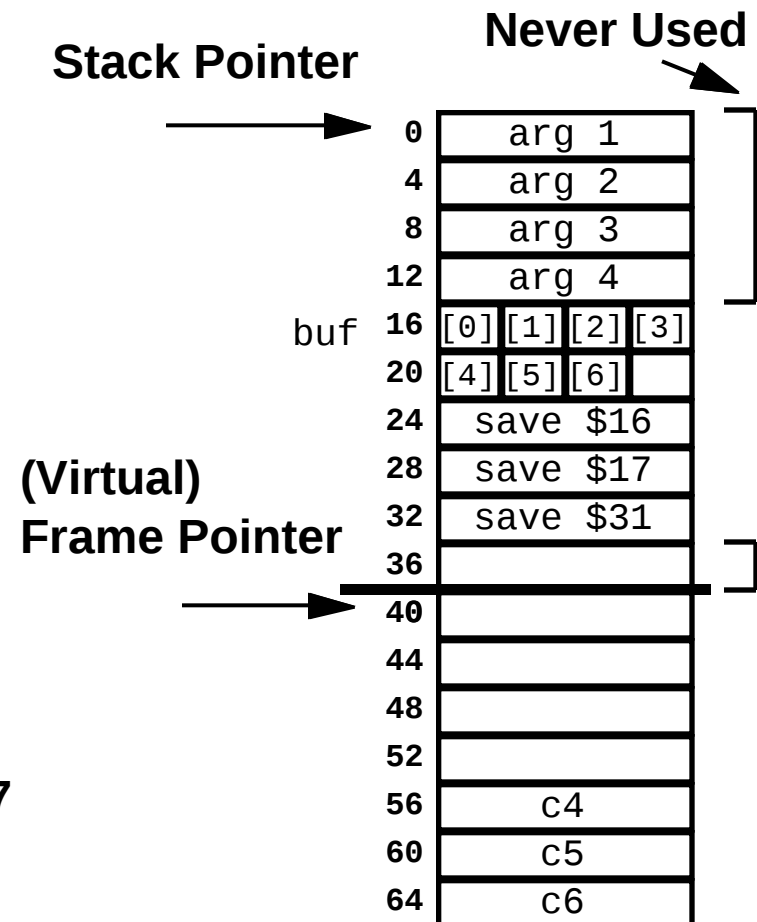
C Code: Argument Retrieval

```
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



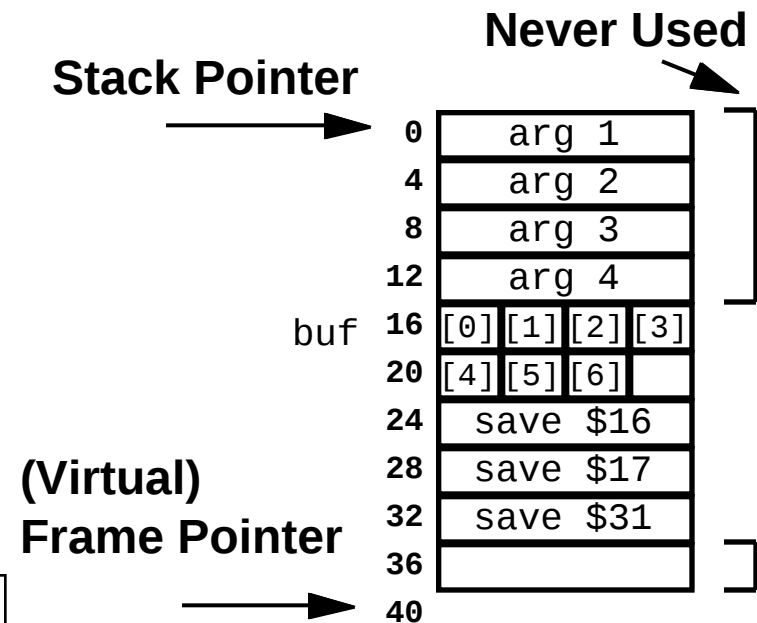
Local Storage Example (Cont.)

C Code: Local Setup

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

Compiled -O

```
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
```



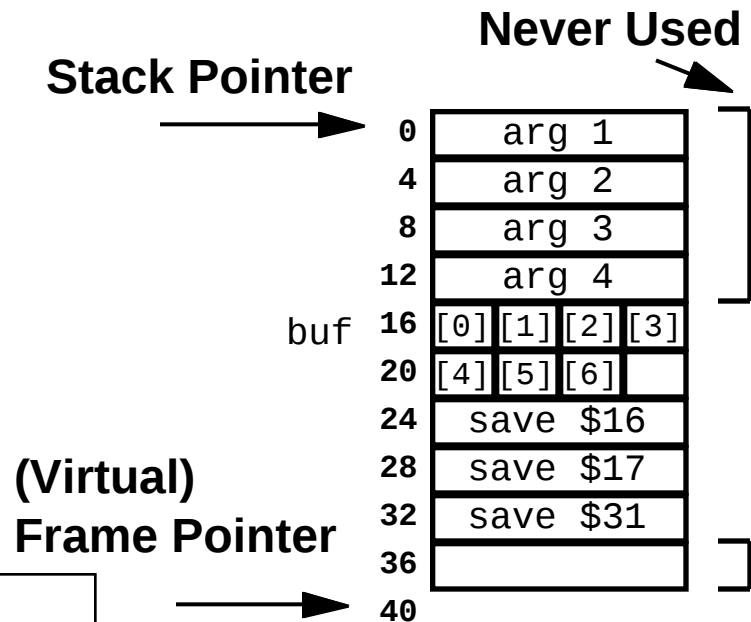
Local Storage Example (Cont.)

C Code: Calls

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

Compiled -O

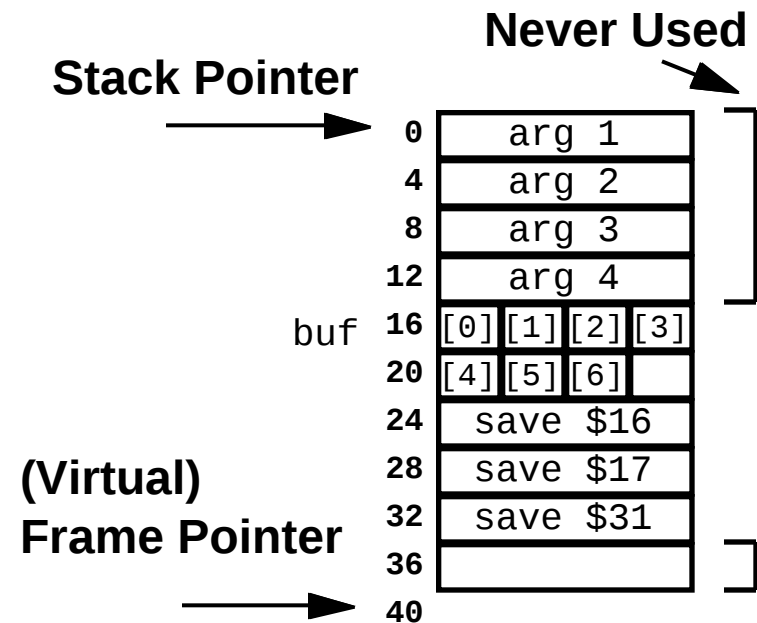
```
li      $4,0x00000008    # i = 8  
jal     malloc           # malloc(i)  
move    $16,$2           # $16 = result  
move    $4,$16  
JAL     strcpy           # strcpy(result,  
move    $5,$17           # , buf)  
move    $2,$16           # return val result
```



Local Storage Example (Cont.)

Completion

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



Basic Data Types

Integral

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

MIPS	Bytes	C
byte	1	[unsigned] char
half word	2	[unsigned] short
word	4	[unsigned] [long] int pointer

Floating Point

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

MIPS	Bytes	C
single precision	4	float
double precision	8	double

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

\$f0	\$f1	Return Values
\$f2	\$f3	
\$f4	\$f5	Caller Save Temporaries: May be overwritten by called procedures
\$f6	\$f7	
\$f8	\$f9	
\$f10	\$f11	
\$f12	\$f13	Procedure arguments
\$f14	\$f15	
\$f16	\$f17	Caller Save Temporaries:
\$f18	\$f19	
\$f20	\$f21	Callee Save Temporaries: May not be overwritten by called procedures
\$f22	\$f23	
\$f24	\$f25	
\$f26	\$f27	
\$f28	\$f29	
\$f30	\$f31	

Floating Point Code Example

Compute Inner Product of Two Vectors

- Single precision

```
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;
}
```

```
inner_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 Precision

```
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:
    l.d     $f2,0($4) # $f2,3 = x[i]
    l.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

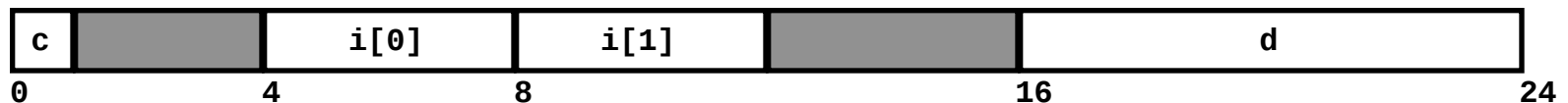
0x40:	44802000	mtc1	r0,\$f4
0x44:	44802800	mtc1	r0,\$f5
0x48:	27bdfbf8	addiu	sp,sp,-8
0x4c:	18c0000c	BLEZ	r6,0x80
0x50:	00001821	move	r3,r0
0x54:	c4820000	lwc1	\$f2,0(r4)
0x58:	c4830004	lwc1	\$f3,4(r4)
0x5c:	c4a00000	lwc1	\$f0,0(r5)
0x60:	c4a10004	lwc1	\$f1,4(r5)
0x64:	24630001	addiu	r3,r3,1
0x68:	46201082	mul.d	\$f2,\$f2,\$f0
0x6c:	0066102a	slt	r2,r3,r6
0x70:	24a50008	addiu	r5,r5,8
0x74:	46222100	add.d	\$f4,\$f4,\$f2
0x78:	1440fff6	BNE	r2,r0,0x54
0x7c:	24840008	addiu	r4,r4,8
0x80:	46202006	mov.d	\$f0,\$f4
0x84:	03e00008	JR	r31
0x88:	27bd0008	addiu	sp,sp,8

Structure Allocation

Principles

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

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



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

```
char struct_c(struct_ptr p)
{
    return p->c;
}
```

```
int *struct_i(struct_ptr p)
{
    return p->i;
}
```

```
int struct_i1(struct_ptr p)
{
    return p->i[1];
}
```

```
double struct_d(struct_ptr p)
{
    return p->d;
}
```

Assembler

```
lb    $2,0($4)    # 1st byte
j     $31
```

```
J     $31
addu  $2,$4,4      # addr of 4th byte
```

```
lw    $2,8($4)    # word at 8th byte
j     $31
```

```
l.d   $f0,16($4)  # dblewd at 16th by.
j     $31
```

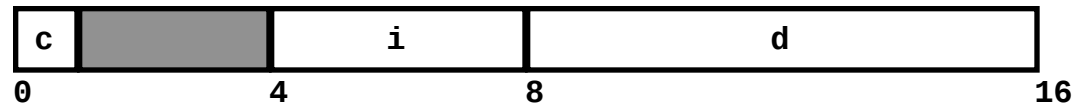
Arrays vs. Pointers

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

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

C Code for Allocation



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

Accessing Through Pointer

C Code

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

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

Assembler

```
lw    $2,4($4)    # word at 4th byte
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

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

Assembler

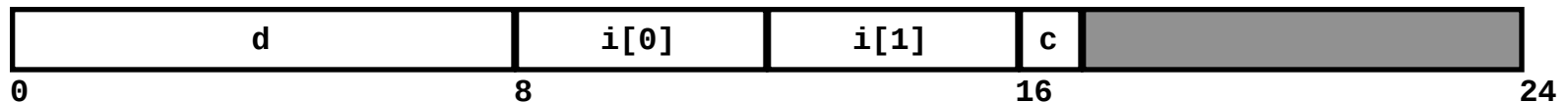
```
sll    $2,$5,1
addu   $2,$2,$5
sll    $2,$2,3    # $2 = 24 * idx
J      $31        # return
        # with a + (scaled) idx
addu   $2,$4,$2
```

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;
```

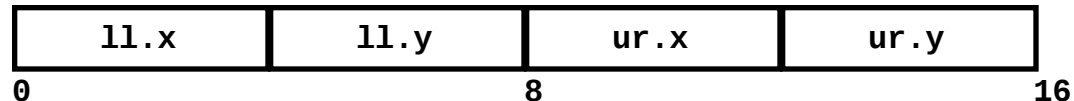


Nested Allocations

Principles

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

```
typedef struct {  
    int x;  
    int y;  
} point_ele, *point_ptr;  
  
typedef struct {  
    point_ele ll;  
    point_ele ur;  
} rect_ele, *rect_ptr;
```



Nested Allocation (cont.)

C Code

```
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

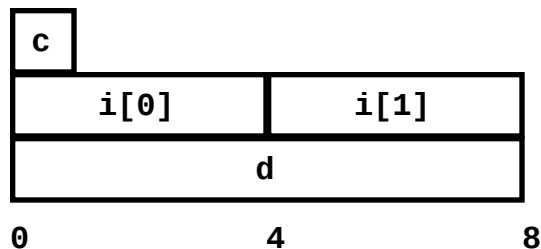
```
lw    $2,8($4)    # $2 = ur.x
lw    $3,0($4)    # $3 = ll.x
subu  $2,$2,$3    # $2 = width
lw    $3,12($4)   # $3 = ur.y
lw    $4,4($4)    # $4 = ll.y
subu  $3,$3,$4    # $3 = height
mult  $2,$3
mflo  $2          # width * height
j     $31
```

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

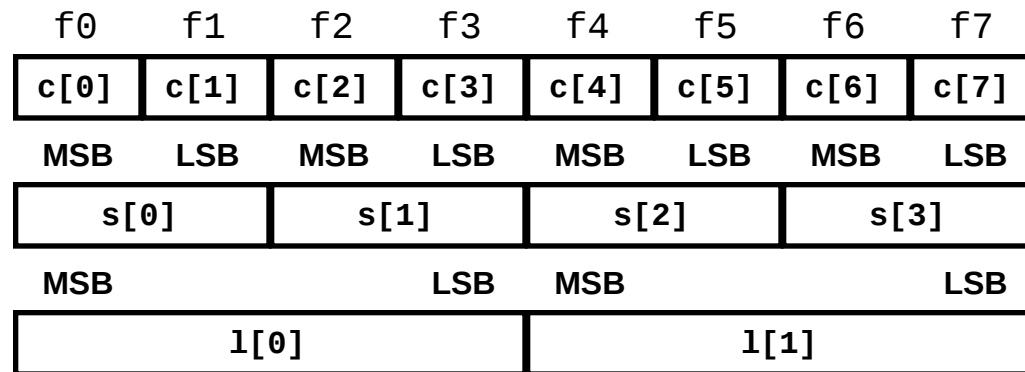
Byte Ordering Example

```
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]);
```

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
l[0]				l[1]			

Byte Ordering Example (cont.)

Big Endian

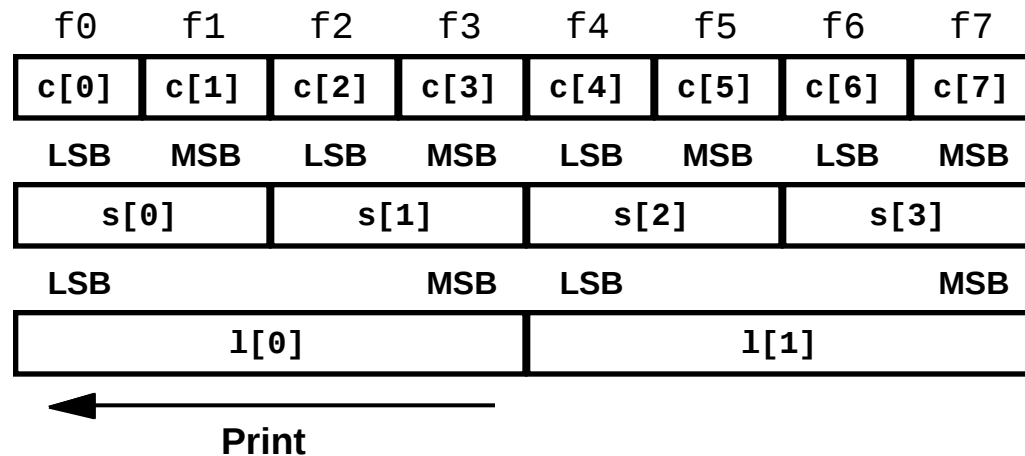


Output on Sun:

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



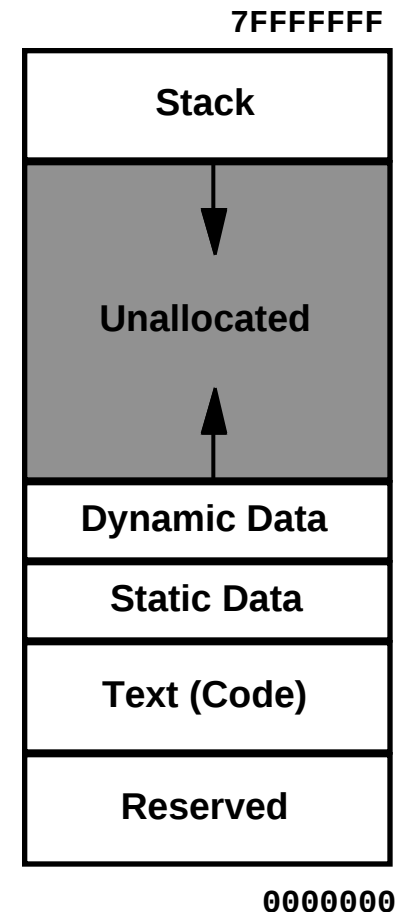
Output on DecStation:

```
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

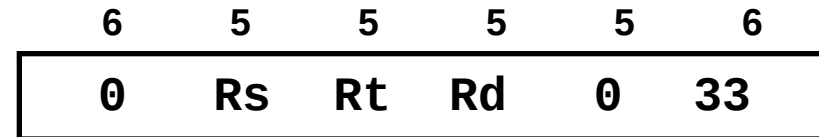
- **Stack**
 - Grows down from top
 - Implements runtime stack
- **Data**
 - Static space for global variables
 - » Allocation determined at compile time
 - Dynamic space for runtime allocation
 - » E.g., using malloc
- **Text**
 - Stores machine code for program
- **Reserved**
 - Used by operating system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



Instruction Formats

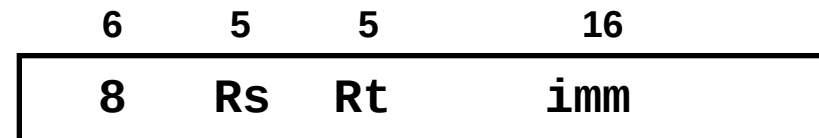
Arithmetic

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



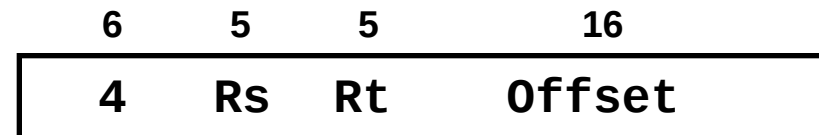
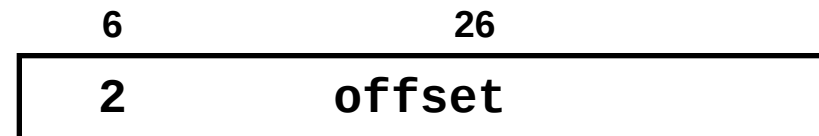
Constant-Manipulating

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



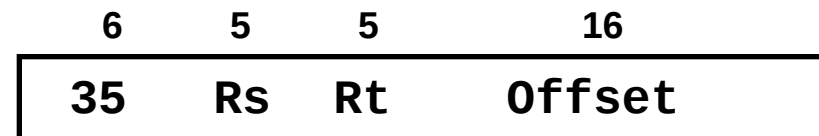
Jump

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



Load/Store

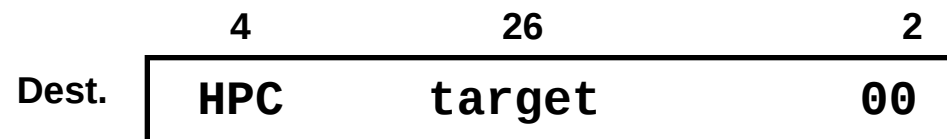
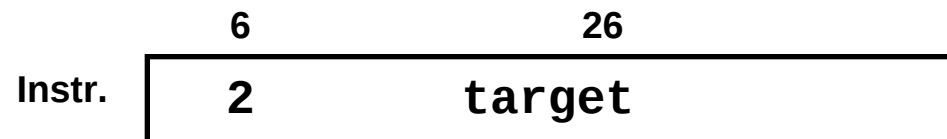
- `lw Rt, Offset(Rs)`



Specifying Jump Targets

Jump (and link)

j target

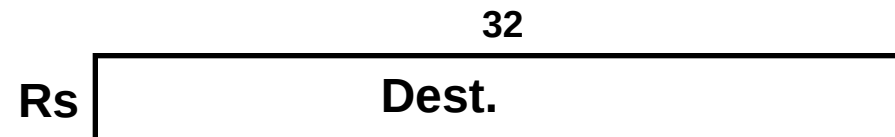
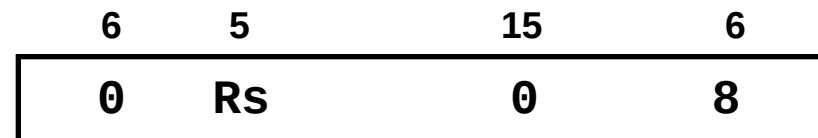


High order bits of slot PC

Jump (and link) Register

jr Rs

- Rs holds dest.



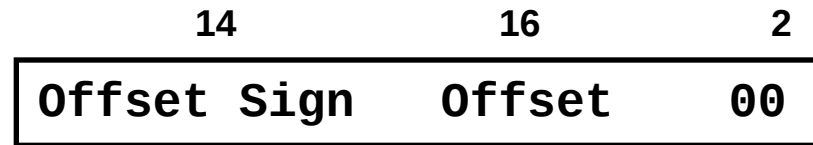
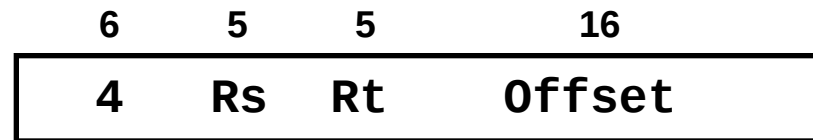
Low order two bits must be 0



Specifying Branch Targets

Branch

`beq Rs, Rt, offset`



+

Delay Slot PC

Dest.

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