Alpha Programming
CS 740
Sept. 17, 1998

Topics

- Basics
- Control Flow
- Procedures
- Instruction Formats
- Flavors of integers
- Floating point
- Data structures
- Byte ordering

Alpha Processors

Reduced Instruction Set Computer (RISC)

- **Simple instructions with regular formats**
- **<u>Key Idea:</u>** ***make the common case fast!***
  - infrequent operations can be synthesized using multiple instructions

Assumes compiler will do optimizations

- **e.g., scalar optimization, register allocation, scheduling, etc.**
- **ISA designed for *compilers*, not assembly language programmers**

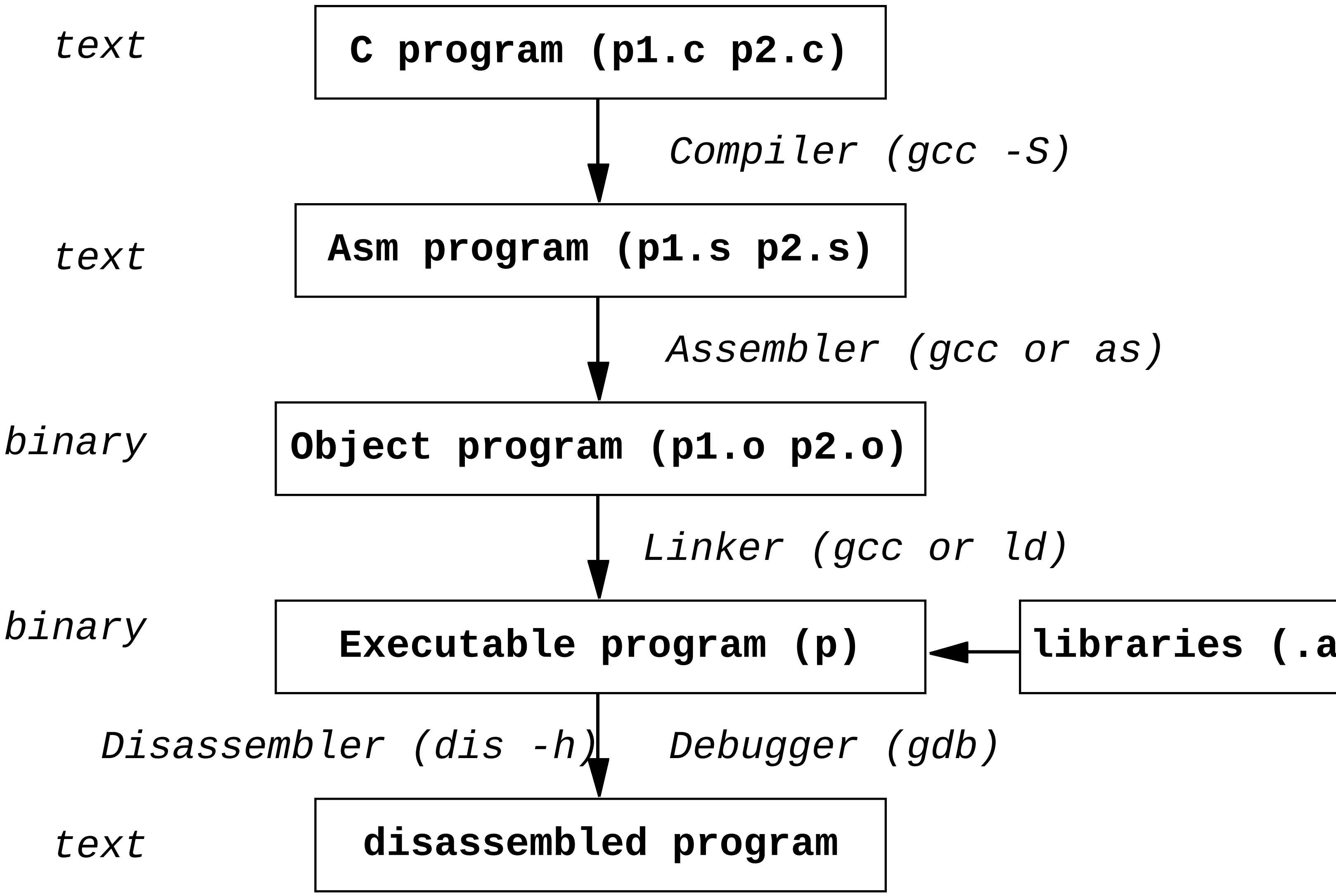
A 2nd Generation RISC Instruction Set Architecture

- **Designed for superscalar processors (i.e. >1 inst per cycle)**
  - avoids some of the pitfalls of earlier RISC ISAs (e.g., delay slots)
- **Designed as a 64-bit ISA from the start**

Very High Performance Machines

- **Alpha has been the clear performance leader for many years now**

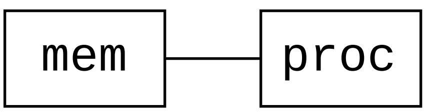
Translation Process



Abstract Machines

Machine Model | **Data** | **Control**

C



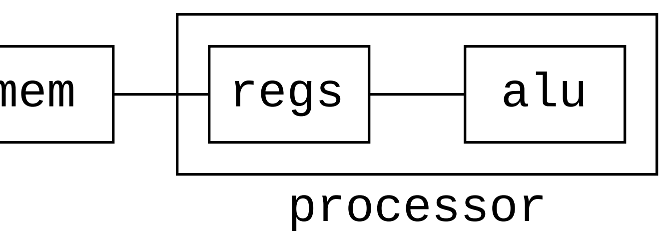
Data

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

Control

1) loops
2) conditiona
3) goto
4) Proc. call
5) Proc. retu

ASM



Data

1) byte
2) word
3) doubleword
4) contiguous word allocation
5) address of initial byte

Control

3) branch/jum
4) jump & lin

Alpha 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

Name	Register	Convention
v0	$0	Return value fro integer functions
t0	$1	Temporaries (not preserved across procedur calls)
t1	$2	
t2	$3	
t3	$4	
t4	$5	
t5	$6	
t6	$7	
t7	$8	
s0	$9	Callee saved
s1	$10	
s2	$11	
s3	$12	
s4	$13	
s5	$14	
s6,fp	$15	Frame pointer, o callee saved

Registers (cont.)

mportant Ones for Now

- $0 Return Value
- $1..$8 Temporaries
- $16 First argument
- $17 Second argument
- $26 Return address
- $31 Constant 0

a0	$16	Integer arguments
a1	$17	
a2	$18	
a3	$19	
a4	$20	
a5	$21	
t8	$22	Temporaries
t9	$23	
t10	$24	
t11	$25	
ra	$26	Return address
pv,t12	$27	Current proc addr or Te
AT	$28	Reserved for assemble
gp	$29	Global pointer
sp	$30	Stack pointer
zero	$31	Always zero

Program Representations

C Code

```
ng int gval;

id test1(long int x, long int y)

gval = (x+x+x) - (y+y+y);
```

btain with command

`gcc -O -S code.c`

roduces file `code.s`

Compiled to Assembly

```
        .align 3
        .globl test1
        .ent test1
test1:
        ldgp $29,0($27)
        .frame $30,0,$26,0
        .prologue 1
        lda $3,gval
        addq $16,$16,$2
        addq $2,$16,$2
        addq $17,$17,$1
        addq $1,$17,$1
        subq $2,$1,$2
        stq $2,0($3)
        ret $31,($26),1
        .end test1
```

Prog. Representation (Cont.)

Object

```
0x120001130 <test1>:
        0x27bb2000
        0x23bd6f30
        0xa47d8098
        0x42100402
        0x40500402
        0x42310401
        0x40310401
        0x40410522
        0xb4430000
        0x6bfa8001
```

Disassembled

```
0x120001130 <test1>:      ldah gp,536870912(t12
0x120001134 <test1+4>:    lda  gp, 28464(gp)
0x120001138 <test1+8>:    ldq  t2, -32616(gp)
0x12000113c <test1+12>:   addq a0, a0, t1
0x120001140 <test1+16>:   addq t1, a0, t1
0x120001144 <test1+20>:   addq a1, a1, t0
0x120001148 <test1+24>:   addq t0, a1, t0
0x12000114c <test1+28>:   subq t1, t0, t1
0x120001150 <test1+32>:   stq  t1, 0(t2)
0x120001154 <test1+36>:   ret  zero, (ra), 1
```

Run gdb on object code

`x/10 0x120001130`

- **Print 10 words in hexadecimal starting at address 0x120001130**

`dissassemble test1`

- **Print disassembled version of procedure**

Alternate Disassembly

Alpha program "dis"

`dis file.o`

- **Prints disassembled version of object code file**
- **The "-h" option prints hardware register names (`r0–r31`)**
- **Code not yet linked**
  - Addresses of procedures and global data not yet resolved

```
test1:
  0x0:   27bb0001  ldah    gp, 1(t12)
  0x4:   23bd8760  lda     gp, -30880(gp)
  0x8:   a47d8010  ldq     t2, -32752(gp)
  0xc:   42100402  addq    a0, a0, t1
  0x10:  40500402  addq    t1, a0, t1
  0x14:  42310401  addq    a1, a1, t0
  0x18:  40310401  addq    t0, a1, t0
  0x1c:  40410522  subq    t1, t0, t1
  0x20:  b4430000  stq     t1, 0(t2)
  0x24:  6bfa8001  ret     zero, (ra), 1
```

Returning a Value from a Procedure

C Code

```
ng int
st2(long int x, long int y)

return (x+x+x) - (y+y+y);
```

Compiled to Assembly

```
        .align 3
        .globl test2
        .ent test2
test2:
        .frame $30,0,$26,0
        .prologue 0
        addq $16,$16,$1
        addq $1,$16,$1
        addq $17,$17,$0
        addq $0,$17,$0
        subq $1,$0,$0
        ret $31,($26),1
        .end test2
```

Place result in $0

Pointer Examples

C Code

```
ng int
ddp(long int *xp,long int *yp)

int x = *xp;
int y = *yp;
return x + y;
```

```
id
cr(long int *sum, long int v)

long int old = *sum;
long int new = old+val;
*sum = new;
```

Annotated Assembly

```
iaddp:
        ldq $1,0($16)      # $1 = *xp
        ldq $0,0($17)      # $0 = *yp
        addq $1,$0,$0      # return with
        ret $31,($26),1    #   value x +
```

```
incr:
        ldq $1,0($16)      # $1 = *sum
        addq $1,$17,$1     # $1 += v
        stq $1,0($16)      # *sum = $1
        ret $31,($26),1    # return
```

Array Indexing

C Code

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

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

Annotated Assembly

```
arefl:
        s8addq $17,$16,$17 # $17 = 8*i + &a[0]
        ldq $0,0($17)      # return val = a[i]
        ret $31,($26),1    # return
```

```
arefi:
        s4addq $17,$16,$17 # $17 = 4*i + &a[0]
        ldl $0,0($17)      # return val = a[i]
        ret $31,($26),1    # return
```

Array Indexing (Cont.)

C Code

```
ng int garray[10];

ng int gref(long int i)

return garray[i];
```

Annotated Assembly

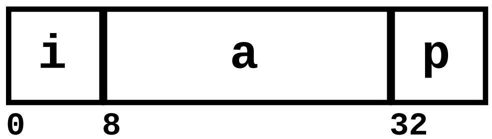
```
        .comm   garray,80

gref:
        ldgp $29,0($27)   # setup the gp
        lda $1,garray     # $1 = &garray[0]
        s8addq $16,$1,$16 # $16 = 8*i + $1
        ldq $0,0($16)     # ret val = garray[
        ret $31,($26),1   # return
```

Disassembled:

```
0x80 <gref>:     27bb0001  ldah   gp, 65536(t12
0x84 <gref+4>:   23bd86e0  lda    gp, -31008(gp
0x88 <gref+8>:   a43d8018  ldq    t0, -32744(gp
0x8c <gref+12>:  42010650  s8addq a0, t0, a0
0x90 <gref+16>:  a4100000  ldq    v0, 0(a0)
0x94 <gref+20>:  6bfa8001  ret    zero, (ra), 1
```

Structures & Pointers

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



C Code

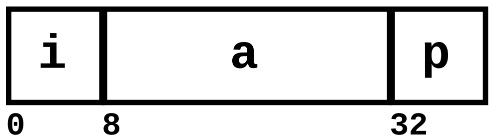
```
void
set_i(struct rec *r,
      long int val)
{
  r->i = val;
}
```

Annotated Assembly

```
set_i:
        stq $17,0($16)   # r->i = val
        ret $31,($26),1
```

Structures & Pointers (Cont.)

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



C Code

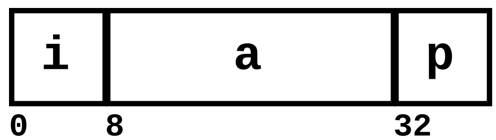
```
long int *
find_a(struct rec *r,
       long int idx)
{
  return &r->a[idx];
}
```

Annotated Assembly

```
find_a:
        s8addq $17,8,$0   # $0 = 8*idx + 8
        addq $16,$0,$0    # $0 += r
        ret $31,($26),1
```

Structures & Pointers (Cont.)

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



C Code

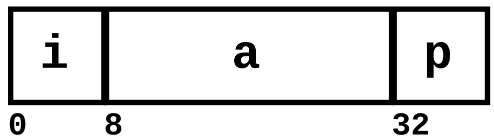
```
void
set_p(struct rec *r,
      long int *ptr)
{
  r->p = ptr;
}
```

Annotated Assembly

```
set_p:
        stq $17,32($16)  # *(r+32) = ptr
        ret $31,($26),1
```

Structures & Pointers (Cont.)

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



C Code

```
id addr(struct rec *r)

 long int *loc;
 r->i = 1;
 loc = &r->a[r->i];
 r->p = loc;
 *(r->p) = 2;
 r->a[0] = 4;
 *(r->p+1) = 8;
```

"bis" = bitwise OR

Annotated Assembly

```
addr:
        bis $31,1,$1       # $1 = 1
        stq $1,0($16)      # r->i = 1
        bis $31,8,$2       # $2 = 8
        addq $16,16,$1     # $1(loc) = &r->a[1
        stq $1,32($16)     # r->p = loc
        bis $31,2,$1       # $1 = 2
        stq $1,16($16)     # r->a[1] = 2
        bis $31,4,$1       # $1 = 4
        stq $1,8($16)      # r->a[0] = 4
        ldq $1,32($16)     # $1 = r->p
        stq $2,8($1)       # *(r->p+1) = 8
        ret $31,($26),1    # return
```

Branches

Conditional Branches

`b`*`Cond`* `Ra,` *`label`*

– *Cond* : branch condition, relative to zero

`bne`	**Equal**	`Ra == 0`
`bne`	**Not Equal**	`Ra != 0`
`bgt`	**Greater Than**	`Ra > 0`
`bge`	**Greater Than or Equal**	`Ra >= 0`
`blt`	**Less Than**	`Ra < 0`
`ble`	**Less Than or Equal**	`Ra <= 0`

- **Register value is typically set by a *comparison* instruction**

Unconditional Branches

`br` *`label`*

Conditional Branches

Comparison Instructions

- Format: `cmpCond Ra, Rb, Rc`
  - *Cond:* **comparison condition, Ra relative to Rb**

`cmpeq`	**Equal**	Rc = (Ra == Rb)
`cmplt`	**Less Than**	Rc = (Ra < Rb)
`cmple`	**Less Than or Equal**	Rc = (Ra <= Rb)
`cmpult`	**Unsigned Less Than**	Rc = (uRa < uRb)
`cmpule`	**Unsigned Less Than or Equal**	Rc = (uRa <= uRb)

C Code

```
ng int
ndbr(long int x, long int y)

long int v = 0;
if (x > y)
  v =  x+x+x+y;
return v;
```

Annotated Assembly

```
condbr:
        bis $31,$31,$0    # v = 0
        cmple $16,$17,$1  # (x <= y)?
        bne $1,$45        # if so, branch
        addq $16,$16,$0   # v = x+x
        addq $0,$16,$0    # v += x
        addq $0,$17,$0    # v += y
$45:
        ret $31,($26),1   # return v
```

Conditional Move Instructions

Motivation:

- **conditional branches tend to disrupt pipelining & hurt performance**

Basic Idea:

- **conditional moves can replace branches in some cases**
  - avoids disrupting the flow of control

Mechanism:

```
cmovCond Ra, Rb, Rc
```

- ***Cond:* comparison condition, `Ra` is compared with *zero***
  - same conditions as a conditional branch (**eq, ne, gt, ge, lt, le**)
- **if (`Ra` *`Cond`* `zero`), then copy `Rb` into `Rc`**

Psuedo-code example:

```
if (x > 0) z = y;   =>    cmovgt x, y, z
```

Conditional Move Example

C Code

```
ng int
x(long int x, long int y)

return (x < y) ? y : x;
```

Annotated Assembly

```
max:
        cmple $17,$16,$1 # $1 = (y <= x)?
        bis $16,$16,$0   # $0 = x
        cmoveq $1,$17,$0 # if $1 = 0, $0 = y
        ret $31,($26),1  # return
```

“Do-While” Loop Example

C Code

```
ng int fact(long int x)

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

Annotated Assembly

```
fact:
        bis $31,1,$0      # result = 1
$50:
        mulq $0,$16,$0    # result *= x
        subq $16,1,$16    # x--
        cmple $16,1,$1    # if (x > 1) then
        beq $1,$50        #  continue loopin
        ret $31,($26),1   # return result
```

“While” Loop Example

C Code

```
ng int ifact(long int x)

long int result = 1;
while (x > 1)
  result *= x--;
return result;
```

Annotated Assembly

```
ifact:
        bis $31,1,$0       # result = 1
        cmple $16,1,$1     # if (x <= 1) then
        bne $1,$51         #  branch to retur
$52:
        mulq $0,$16,$0     # result *= x
        subq $16,1,$16     # x--
        cmple $16,1,$1     # if (x > 1) then
        beq $1,$52         #  continue loopin
$51:
        ret $31,($26),1    # return result
```

“For” Loops in C

```
for (init; test; update )
    body
```

direct translation

```
init;
while(test )
   { body ;update }
```

“For” Loop Example

C Code

```
Find max ele. in array */
ng int amax(long int a[],
              long int count)

long int i;
long int result = a[0];
for (i = 1; i < count; i++)
  if (a[i] > result)
    result = a[i];
return result;
```

r (*init*; *test*; *update*)
body

it;
ile(*test*)
{ *body* ;*update* }

Annotated Assembly

```
amax:
        ldq $0,0($16)     # result = a[0]
        bis $31,1,$3      # i = 1
        cmplt $3,$17,$1   # if (i >= count),
        beq $1,$61        #  branch to retur
$63:
        s8addq $3,$16,$1 # $1 = 8*i + &a[0]
        ldq $2,0($1)      # $2 = a[i]
        cmple $2,$0,$1    # if (a[i] <= res)
        bne $1,$62        #  skip “then” par
        bis $2,$2,$0      # result = a[i]
$62:
        addq $3,1,$3      # i++
        cmplt $3,$17,$1   # if (i < count),
        bne $1,$63        #  continue loopin
$61:
        ret $31,($26),1   # return result
```

Jumps

Characteristics:

- transfer of control is unconditional
- target address is specified by a *register*

Format:

`jmp Ra,(Rb),Hint`

- Rb contains the target address
- for now, don't worry about the meaning of Ra or *"Hint"*
- synonyms for `jmp`: `jsr`, `ret`

Compiling Switch Statements

C Code

```
pedef enum
{ADD, MULT, MINUS, DIV, MOD, BAD}
_type;

ar 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

Code

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

ar 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 $16
zapnot $16,15,$16   # zero upper 32 bits
cmpule $16,5,$1     # if (op > 5) then
beq $1,$66          #  branch to return
lda $1,$74          # $1 = &jtab[0]
s4addq $16,$1,$1    # $1 = &jtab[op]
ldl $1,0($1)        # $1 = jtab[op]
addq $1,$29,$2      # $2 = $gp + jtab[op
jmp $31,($2),$68    # jump to jtab code
```

Jump Table

able Contents

```
4:
    .gprel32 $68
    .gprel32 $69
    .gprel32 $70
    .gprel32 $71
    .gprel32 $72
    .gprel32 $73
```

numerated Values

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

Targets & Completion

```
$68:
        bis $31,43,$0    # return '+'
        ret $31,($26),1
$69:
        bis $31,42,$0    # return '*'
        ret $31,($26),1
$70:
        bis $31,45,$0    # return '-'
        ret $31,($26),1
$71:
        bis $31,47,$0    # return '/'
        ret $31,($26),1
$72:
        bis $31,37,$0    # return '%'
        ret $31,($26),1
$73:
        bis $31,63,$0    # return '?'
$66:
        ret $31,($26),1
```

Procedure Calls & Returns

aintain the return address in a special register ($26)

rocedure call:

- **`bsr $26, `*`label`*** Save return addr in $26, branch to ***`label`***
- **`jsr $26, (Ra)`** Save return addr in $26, jump to address in **`Ra`**

rocedure return:

- **`ret $31, ($26)`** Jump to address in **`$26`**

Code

```
g int caller()
eturn callee(); }

g int callee()
eturn 5L; }
```

Annotated Assembly

```
caller:
       ...
 0x800 bsr $26,callee    # save return addr (0x804)
 0x804 ...               #  $26, branch to callee
       ...
callee:
 0x918 bis $31,5,$0      # return value = 5
 0x91c ret $31,($26),1   # jump to addr in $26
```

Stack-Based Languages

Languages that support recursion

- **e.g., C, Pascal**

Stack Allocated in *Frames*

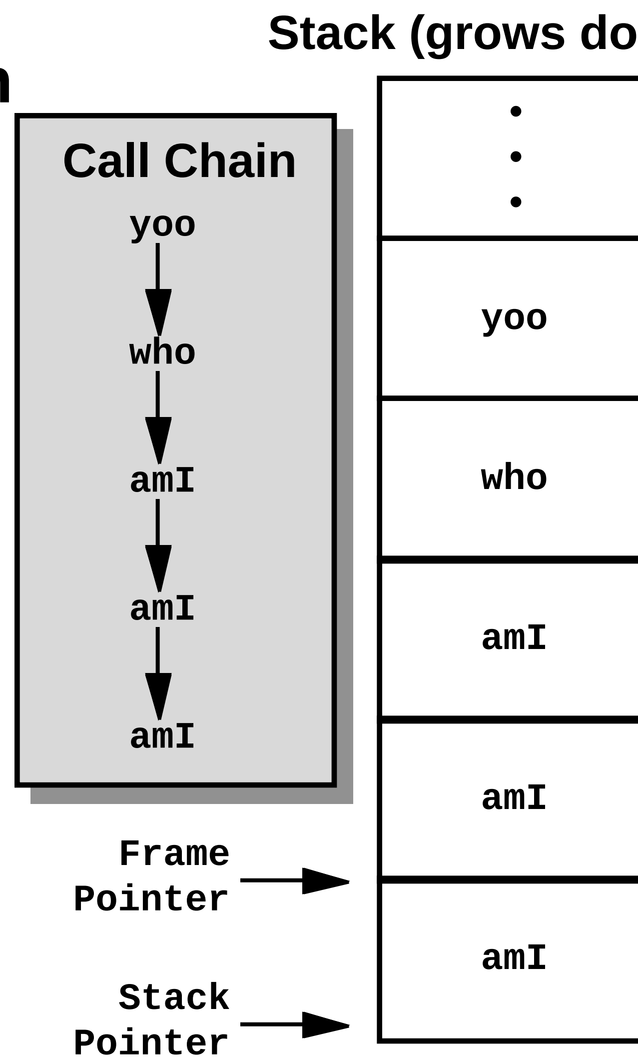
- **state for procedure invocation**
  - – return point, arguments, locals

Code Example

```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

```
who(…)
{
    •
    •
    amI();
    •
    •
}
```

```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```



Register Saving Conventions

Vhen procedure yoo calls `who`:

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

Caller Save" Registers:

- **not guaranteed to be preserved across procedure calls**
- **can be immediately overwritten by a procedure without first saving**
  – useful for storing local temporary values within a procedure
- **if yoo wants to preserve a caller-save register across a call to `who`:**
  – save it on the stack before calling `who`
  – restore after `who` returns

Callee Save" Registers:

- **must be preserved across procedure calls**
- **if `who` wants to use a callee-save register:**
  – save current register value on stack upon procedure entry
  – restore when returning

Register Saving Examples

Caller Save

- **Caller must save / restore if live across procedure call**

```
o:
   bis $31, 17, $1
      • • •
   stq $1, 8($sp) # save $1
   bsr $26, who
   ldq $1, 8($sp) # restore $1
      • • •
   addq $1, 1, $0
   ret $31, ($26)
```

```
o:
   bis $31, 6, $1 # overwrite $1
      • • •
   ret $31, ($26)
```

Callee Save

- **Callee must save / restore if overwriting**

```
yoo:
    bis $31, 17, $9
      • • •
    bsr $26, who
      • • •
    addq $9, 1, $0
    ret $31, ($26)
```

```
who:
    stq $9, 8($sp)   # save $9
    bis $31, 6, $9   # overwrite $
      • • •
    ldq $9, 8($sp)   # restore $9
    ret $31, ($26)
```

Alpha has both types of registers -> choose type based on usage

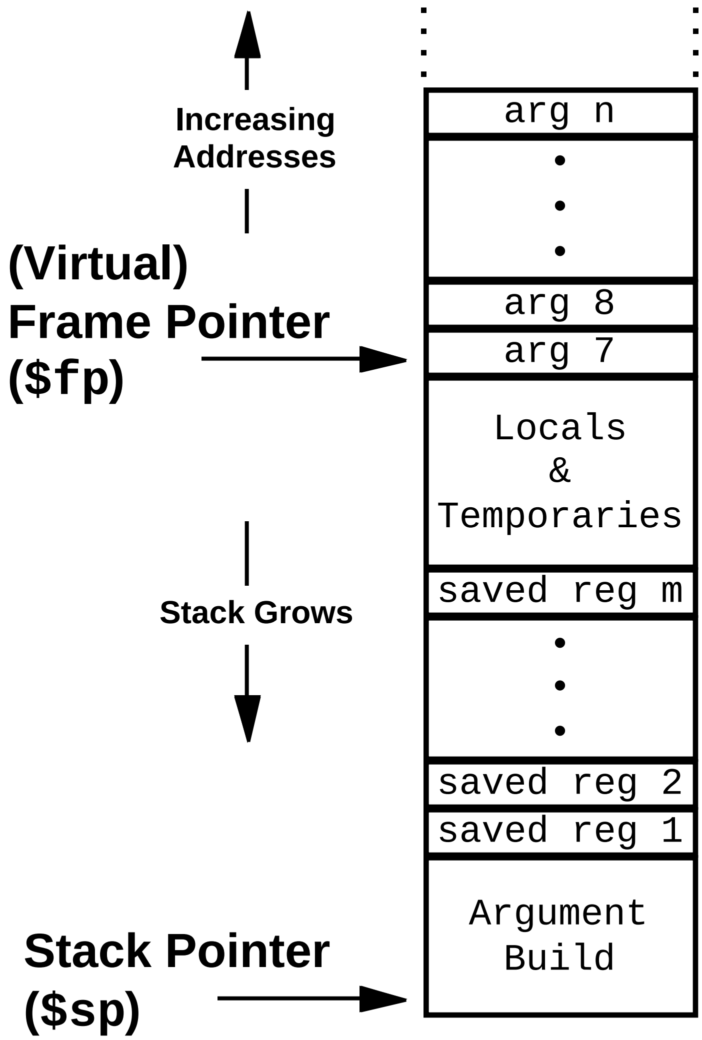
Alpha Stack Frame

Conventions

- **Agreed upon by all program/ compiler writers**
  - Allows linking between different compilers
  - Enables 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 (at the very least, to save the return address ($26))

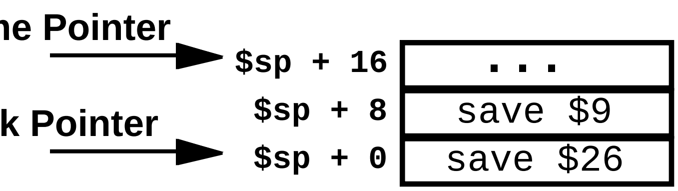
Stack Frame Structure

- **Must be a multiple of 16 bytes**
  - pad the region for locals and temporaries as needed

Stack Frame Example

C Code

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



Stack frame: 16 bytes

Virtual frame ptr @ $sp + 16

Save registers $26 and $9

No floating pt. regs. used

Procedure Prologue

```
rfact:
       ldgp $29,0($27)     # setup gp
rfact..ng:
       lda $30,-16($30)    # $sp -= 16
       .frame $30,16,$26,0
       stq $26,0($30)      # save ret addr
       stq $9,8($30)       # save $9
       .mask 0x4000200,-16
       .prologue 1
```

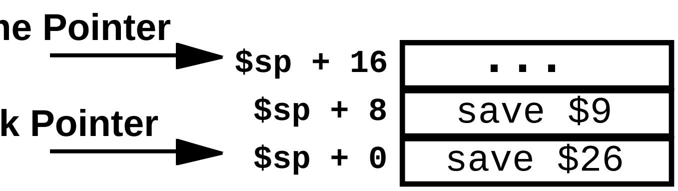
Procedure Epilogue

```
       ldq $26,0($30)      # restore ret ad
       ldq $9,8($30)       # restore $9
       addq $30,16,$30     # $sp += 16
       ret $31,($26),1
```

Stack Frame Example (Cont.)

C Code

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



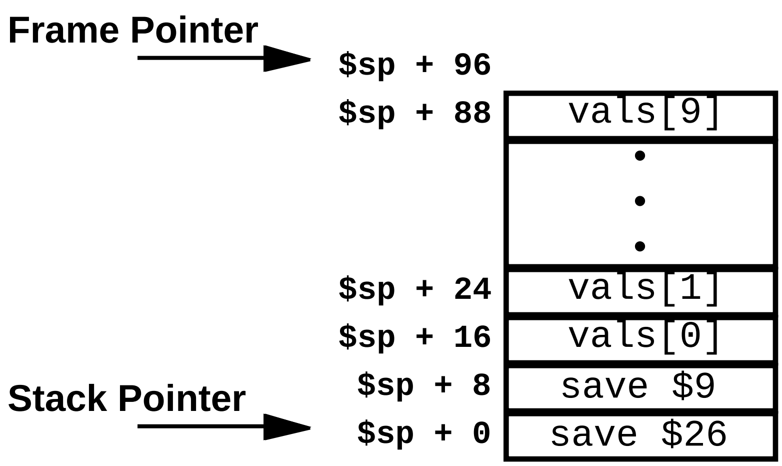
Annotated Assembly

```
rfact:
          ldgp $29,0($27)      # setup gp
rfact..ng:
          lda $30,-16($30)     # $sp -= 16
          .frame $30,16,$26,0
          stq $26,0($30)       # save return addr
          stq $9,8($30)        # save $9
          .mask 0x4000200,-16
          .prologue 1
          bis $16,$16,$9       # $9 = x
          cmple $9,1,$1        # if (x <= 1) then
          bne $1,$80           #  branch to $80
          subq $9,1,$16        # $16 = x - 1
          bsr $26,rfact..ng    # recursive call
          mulq $9,$0,$0        # $0 = x*rfact(x-1
          br $31,$81           # branch to epilog
          .align 4
$80:
          bis $31,1,$0         # return val = 1
$81:
          ldq $26,0($30)       # restore retrn ad
          ldq $9,8($30)        # restore $9
          addq $30,16,$30      # $sp += 16
          ret $31,($26),1
```

Stack Frame Example #2

C Code

```
ld show_facts(void) {
int i;
long int vals[10];
vals[0] = 1L;
for (i = 1; i < 10; i++)
  vals[i] = vals[i-1] * i;
for (i = 9; i >= 0; i--)
  printf("Fact(%d) = %ld\n",
          i, vals[i]);
```



Procedure Prologue

```
show_facts:
       ldgp $29,0($27)
       lda $30,-96($30)     # $sp -= 96
       .frame $30,96,$26,0
       stq $26,0($30)       # save ret addr
       stq $9,8($30)        # save $9
       .mask 0x4000200,-96
       .prologue 1
       bis $31,1,$1         # $1 = 1
       stq $1,16($30)       # vals[0] = 1L
```

- **Stack frame: 96 bytes**
- **Virtual frame ptr @ $sp + 96**
- **Save registers $26 and $9**
- **Local storage for `vals[]`**

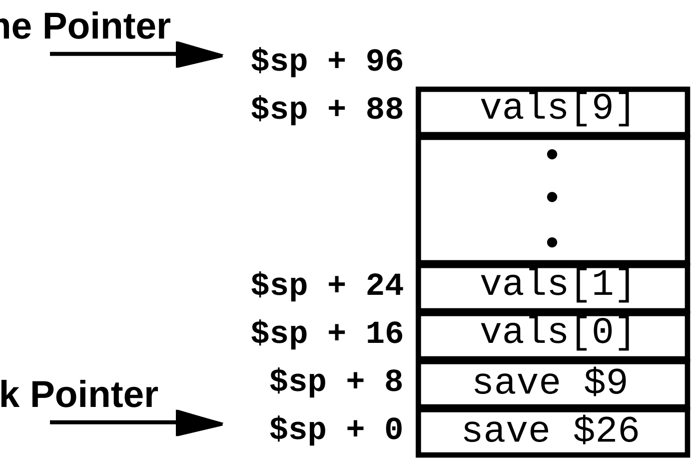
Stack Frame Example #2 (Cont.)

C Code

```
ld show_facts(void) {
nt i;
ong int vals[10];
als[0] = 1L;
or (i = 1; i < 10; i++)
  vals[i] = vals[i-1] * i;
or (i = 9; i >= 0; i--)
  printf("Fact(%d) = %ld\n",
          i, vals[i]);
```

Procedure Prologue

```
show_facts:
       ldgp $29,0($27)
       lda $30,-96($30)    # $sp -= 96
       .frame $30,96,$26,0
       stq $26,0($30)      # save ret add
       stq $9,8($30)       # save $9
       .mask 0x4000200,-96
       .prologue 1
       bis $31,1,$1        # $1 = 1
       stq $1,16($30)      # vals[0] = 1L
```



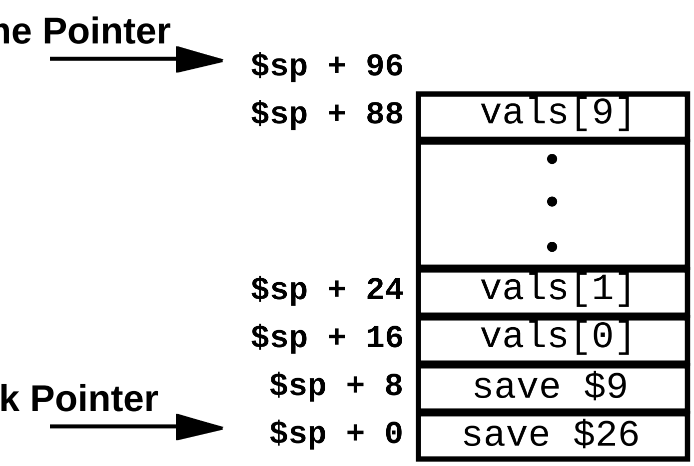
Procedure Epilogue

```
    ldq $26,0($30)      # restore ret addr
    ldq $9,8($30)       # restore $9
    addq $30,96,$30     # sp += 96
    ret $31,($26),1
```

Stack Frame Example #2 (Cont.)

C Code

```
ld show_facts(void) {
nt i;
ong int vals[10];
vals[0] = 1L;
for (i = 1; i < 10; i++)
  vals[i] = vals[i-1] * i;
for (i = 9; i >= 0; i--)
  printf("Fact(%d) = %ld\n",
           i, vals[i]);
```



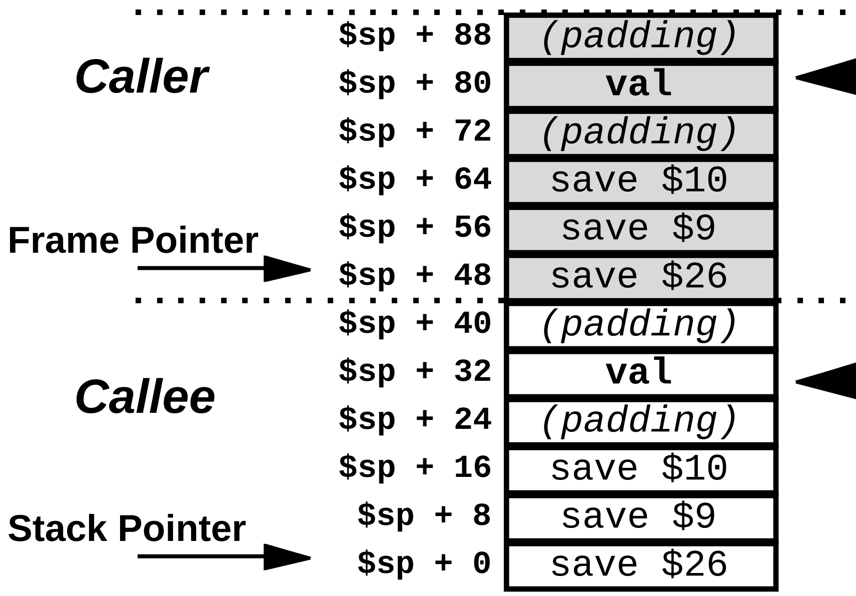
Procedure Body

```
        bis $31,1,$9          # i = 1
$86:
        s8addq $9,$30,$2      # $2 = 8*i + $sp
        addq $2,16,$2         # $2 = &vals[i]
        subl $9,1,$1          # $1 = i - 1
        s8addq $1,$30,$3      # $3 = 8*(i-1) + $sp
        addq $3,16,$3         # $3 = &vals[i-1]
        bis $3,$3,$1          # $1 = &vals[i-1]
        ldq $1,0($1)          # $1 = vals[i-1]
        mulq $9,$1,$1         # $1 = vals[i-1]*i
        stq $1,0($2)          # vals[i] = $1
        addl $9,1,$9          # i++
        cmple $9,9,$1         # if (i <= 9) then
        bne $1,$86            #  continue looping
        bis $31,9,$9          # i = 9
$91:
        s8addq $9,$30,$1      # $1 = 8*i + $sp
        addq $1,16,$1         # $1 = &vals[i]
        lda $16,$C32          # arg1 = &”Fact(%d}.
        bis $9,$9,$17         # arg2 = i
        ldq $18,0($1)         # arg3 = vals[i]
        jsr $26,printf        # call printf
        ldgp $29,0($26)       # reset gp
        subl $9,1,$9          # i--
        cmplt $9,0,$1         # if (i >= 0) then
        beq $1,$91            #  continue looping
```

Stack Addrs as Procedure Args

C Code

```
ld rfact2(long int x,
          long int *result)

if (x <= 1)
  *result = 1;
else {
  long int val;
  rfact2(x-1,&val);
  *result = x * val;
}
return;
```



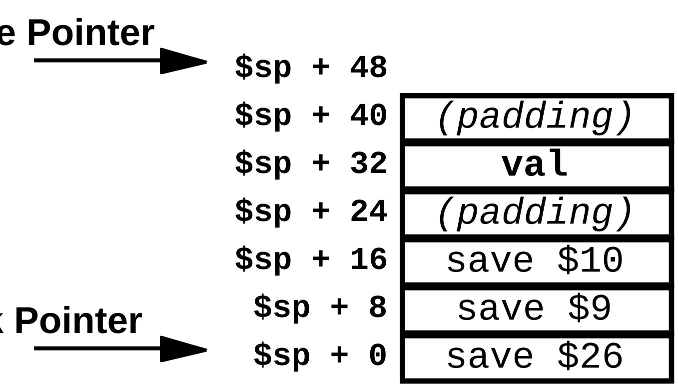

```
rfact2:
        lda $30,-48($30)   # $sp -= 48
        stq $26,0($30)     # save $26
        stq $9,8($30)      # save $9
        stq $10,16($30)    # save $10
        bis $16,$16,$9     # $9 = x
        ...
        subq $9,1,$16      # arg1 = x - 1
        addq $30,32,$17    # arg2 = $sp + 3
        bsr $26,rfact2
```

- Stack frame: 48 bytes
- Padded to 16B alignment
- `val` stored at $sp + 32
- "$sp + 32" passed as second argument ($17) to recursive call of `rfact2`

tack Addrs as Procedure Args (Cont

C Code

```
ld rfact2(long int x,
          long int *result)

if (x <= 1)
  *result = 1;
else {
  long int val;
  rfact2(x-1,&val);
  *result = x * val;
}
return;
```




```
rfact2:
      lda $30,-48($30) # $sp -= 48
      stq $26,0($30)   # save $26
      stq $9,8($30)    # save $9
      stq $10,16($30)  # save $10
      bis $16,$16,$9   # $9 = x
      bis $17,$17,$10  # $10 = result
      cmple $9,1,$1    # if (x > 1) then
      beq $1,$83       #  branch to $83
      bis $31,1,$1     # $1 = 1
      br $31,$85       # go to epilogue
$83:
      subq $9,1,$16    # arg1 = x - 1
      addq $30,32,$17  # arg2 = $sp + 32
      bsr $26,rfact2   # rfact2(x-1,&val
      ldq $1,32($30)   # $1 = val
      mulq $9,$1,$1    # $1 = x * val
$85:
      stq $1,0($10)    # store to *resul
      ldq $26,0($30)   # restore $26
      ldq $9,8($30)    # restore $9
      ldq $10,16($30)  # restore $10
      addq $30,48,$30  # $sp += 48
      ret $31,($26),1  # return
```

Stack Corruption Example

C Code

```
oid overwrite(int a0, int a1,
      int a2, int a3, int a4,
      int a5, int a6)

 long int buf[1]; /* Not enough! */
 long int i = 0;
 buf[i++] = a0;
 buf[i++] = a1;
 buf[i++] = a2;
 buf[i++] = a3;
 buf[i++] = a4;
 buf[i++] = a5;
 buf[i++] = a6;
 buf[i++] = 0;
 return;
```

```
void crash()
{
  overwrite(0,0,0,0,0,0,0);
}
```

This code results in a segmentation fault on the Alpha!

Stack Corruption Example (Cont.)

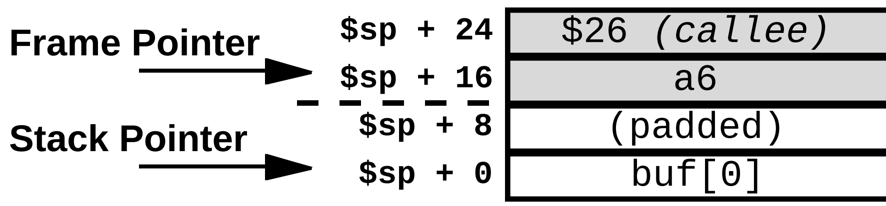
C Code

```
id overwrite(int a0, int a1,
      int a2, int a3, int a4,
      int a5, int a6)

long int buf[1];
long int i = 0;
buf[i++] = a0;
buf[i++] = a1;
buf[i++] = a2;
buf[i++] = a3;
buf[i++] = a4;
buf[i++] = a5;
buf[i++] = a6;
buf[i++] = 0;
return;
```

Stack frame: 16 bytes

Virtual frame ptr @ $sp + 16

-> overwrites callee stack!



Annotated Assembly

```
overwrite:
        lda $30,-16($30) # $sp -= 16
        ldl $1,16($30)   # $1 = a6
        stq $16,0($30)   # buf[0] = a0
        stq $17,8($30)   # buf[1] = a1
        stq $18,16($30)  # buf[2] = a2
        stq $19,24($30)  # buf[3] = a3
        stq $20,32($30)  # buf[4] = a4
        stq $21,40($30)  # buf[5] = a5
        stq $1, 48($30)  # buf[6] = a6
        stq $31,56($30)  # buf[7] = 0
        addq $30,16,$30  # $sp += 16
        ret $31,($26),1
```

Instruction Formats

Arithmetic Operations:

- **all register operands**
  - `addq $1, $7, $5`
- **with a literal operand**
  - `addq $1, 15, $5`

6	5	5	3	1	7	5
Opcode	Ra	Rb	SBZ	0	Func	Rc

6	5	8	1	7	5
Opcode	Ra	Lit	1	Func	Rc

Branches:

- **a single source register**
  - `bne $1,` *`label`*

6	5	21
Opcode	Ra	Displacement

Jumps:

- **one source, one dest reg**
  - `jsr $26, $1,` *`hint`*

6	5	5	16
Opcode	Ra	Rb	Hint

Loads & Stores:

- `ldq $1, 16($30)`

6	5	5	16
Opcode	Ra	Rb	Offset

Basic Data Types

ntegral

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

Alpha	Bytes	C
byte	1	[unsigned] `char`
word	2	[unsigned] `short`
long word	4	[unsigned] `int`
quad word	8	[unsigned] `long int`, pointers

loating Point

- Stored & operated on in floating point registers
- Special instructions for four different formats (only 2 we care about)

Alpha	Bytes	C
S_floating	4	`float`
T_floating	8	`double`

Int vs. Long Int

Difference Data Types

- **Long int uses quad (8-byte) word**
- **Int uses long (4-byte) word**

Visible to C Programmer

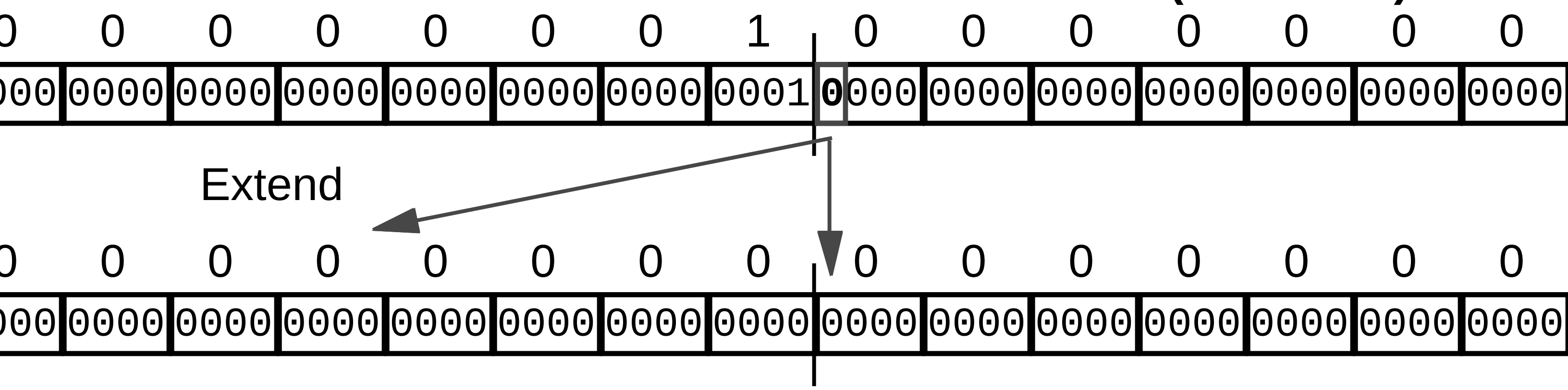
- **Long constants should be suffixed with "L"**

```
0x0000000100000002L    -->     4294967298
0x0000000100000002     -->              2     (truncated)
0x0000000080000001L    -->     2147483649
0x0000000080000001     -->    -2147483647     (extended)
```

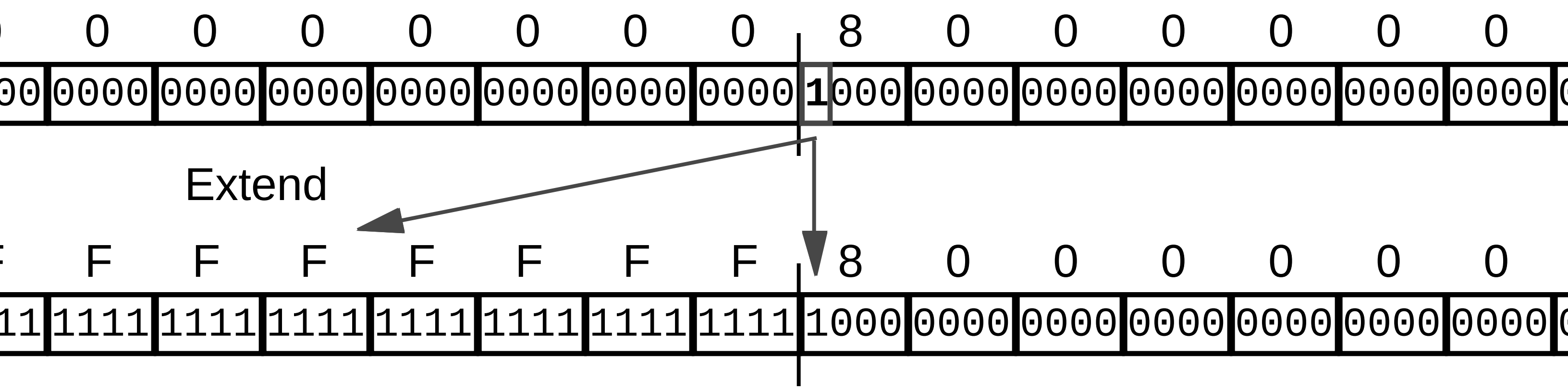
- **Printf format string should use `%ld` and `%lu`**
- **Don't try to pack pointers into space declared for integer**
  - Pointer will be corrupted
  - Seen in code that manipulates low-level data structures

A Closer Look at Quad --> Long

`0x0000000100000002` `-->` `2` (truncated)



`0x0000000080000001` `-->` `-2147483647` (extended)



Internal Representation

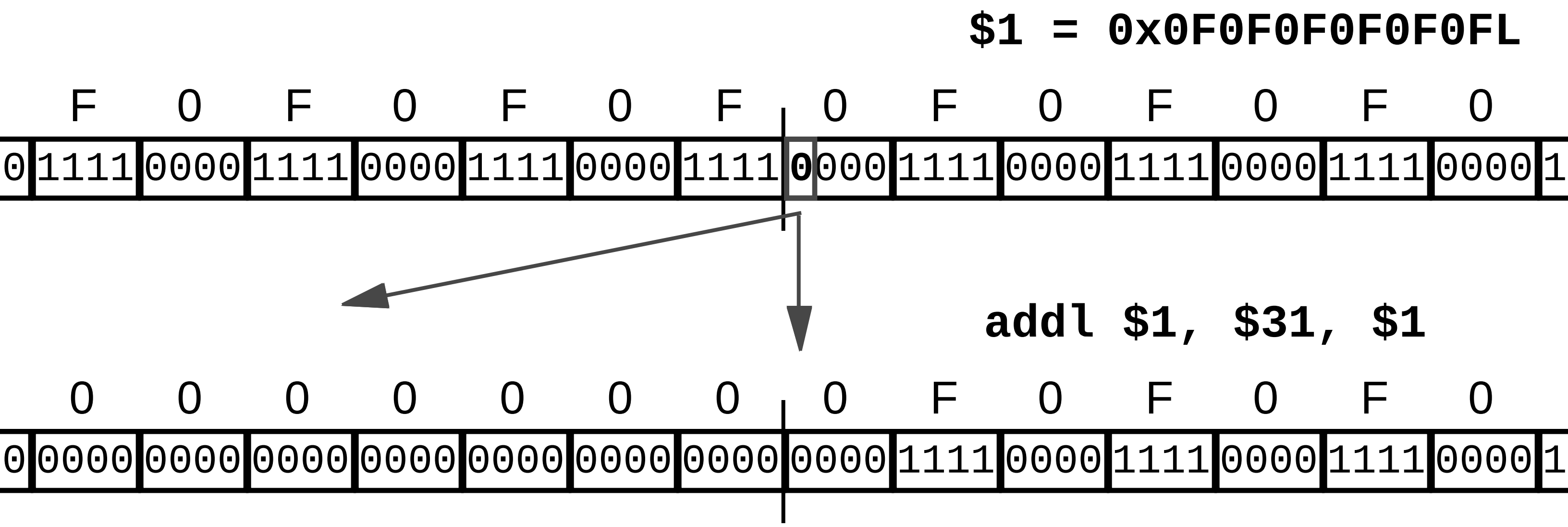
All General Purpose Registers 8 bytes

- **Long (unsigned) int's stored in full precision form**
- **Int's stored in signed-extended form**
  - High order 33 bits all match sign bit
- **Unsigned's also stored in sign-extended form**
  - Even though really want high order 32 bits to be zero
  - Special care taken with these values

Separate Quad and Long Word Arithmetic Instructions

- **`addq` computes sum of 8-byte arguments**
- **`addl` computes sign-extended sum of 4-byte arguments**
  - **`addl` $16, $31, $16** handy way to sign extend int in register $16
- **`ldq` reads 8 bytes from memory into register**
- **`ldl` reads 4 bytes from memory and sign extends into register**

ADDL Example



Integer Conversion Examples

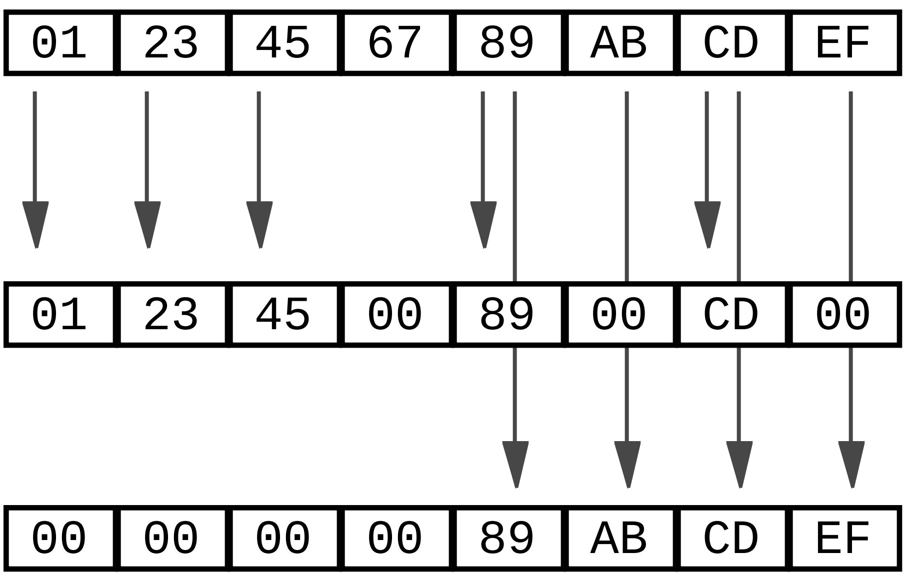
C Code	Return Value Computation
`int long2int(long int li) { return (int) li; }`	`addl $16,$31,$0  # sign extend` [Replace high order bits with sign]
`long int2long(int i) { return (long) i; }`	`bis $16,$16,$0 # Verbatim copy` [Already in proper form]
`unsigned ulong2uint(long unsigned ul) { return (unsigned) ul; }`	`addl $16,$31,$0 # sign extend` [Replace high order bits with sign. Even though really want 0's]
`long unsigned uint2ulong(unsigned int u) { return (unsigned long) u; }`	`zapnot $16,15,$0 # zero high byt` [Clear high order bits]

Byte Zapping

Set selected bytes to zero

- **`zap a, b, c`**
  - Low order 8 bits of b acts as mask
  - Copy nonmasked bytes from a to c
- **`zapnot a, b, c`**

`$1 = 0x0123456789abcdefL`



`zap $1, 37, $2`

$37_{10} = 00010101_2$

`zapnot $1, 15, $2`

$15_{10} = 00001111_2$

Floating Point Unit

mplemented as Separate Unit

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

loating Point Formats

- S_Floating (C `float`): 32 bits
- T_Floating (C `double`): 64 bits

loating Point Data Registers

- 32 registers, each 8 bytes
- Labeled $f0 to $f31
- $f31 is always 0.0

$f0	$f1	Return Values
$f2	$f3	Callee Save Temporaries:
$f4	$f5	
$f6	$f7	
$f8	$f9	
$f10	$f11	Caller Save Temporaries:
$f12	$f13	
$f14	$f15	
$f16	$f17	Procedure argum
$f18	$f19	
$f20	$f21	
$f22	$f23	Caller Save Temporaries:
$f24	$f25	
$f26	$f27	
$f28	$f29	
$f30		
$f31		Always 0.0

Floating Point Code Example

Compute Inner Product of Two Vectors

- Single precision

```
at inner_prodF
float x[], float y[],
int n)

nt i;
loat result = 0.0;
or (i = 0; i < n; i++) {
 result += x[i] * y[i];

eturn result;
```

```
        cpys $f31,$f31,$f0 # result = 0.0
        bis $31,$31,$3      # i = 0
        cmplt $31,$18,$1    # 0 < n?
        beq $1,$102         # if not, skip loop
        .align 5
$104:
        s4addq $3,0,$1      # $1 = 4 * i
        addq $1,$16,$2      # $2 = &x[i]
        addq $1,$17,$1      # $1 = &y[i]
        lds $f1,0($2)       # $f1 = x[i]
        lds $f10,0($1)      # $f10 = y[i]
        muls $f1,$f10,$f1   # $f1 = x[i] * y[i]
        adds $f0,$f1,$f0    # result += $f1
        addl $3,1,$3        # i++
        cmplt $3,$18,$1     # i < n?
        bne $1,$104         # if so, loop
$102:
        ret $31,($26),1     # return
```

Double Precision

```
ble inner_prodD
double x[],
double y[], int n)

nt i;
ouble result = 0.0;
or (i = 0; i < n; i++) {
 result += x[i] * y[i];

eturn result;
```

```
     cpys $f31,$f31,$f0 # result = 0.0
     bis $31,$31,$3     # i = 0
     cmplt $31,$18,$1   # 0 < n?
     beq $1,$102        # if not, skip loop
     .align 5
$104:
     s8addq $3,0,$1     # $1 = 4 * i
     addq $1,$16,$2     # $2 = &x[i]
     addq $1,$17,$1     # $1 = &y[i]
     ldt $f1,0($2)      # $f1 = x[i]
     ldt $f10,0($1)     # $f10 = y[i]
     mult $f1,$f10,$f1  # $f1 = x[i] * y[i]
     addt $f0,$f1,$f0   # result += $f1
     addl $3,1,$3       # i++
     cmplt $3,$18,$1    # i < n?
     bne $1,$104        # if so, loop
$102:
     ret $31,($26),1    # return
```

Numeric Format Conversion

Between Floating Point and Integer Formats

- **Special conversion instructions `cvttq`, `cvtqt`, `cvtts`, `cvtst`, …**
- **Convert source operand in one format to destination in other**
- **Both source & destination must be FP register**
  - Transfer to & from GP registers via stack store/load

C Code

Conversion Code

```
float double2float(double d)
{
  return (float) d;
}
```

```
cvtts $f16,$f0
```

[Convert T_Floating to S_Floating]

```
double long2double(long i)
{
  return (double) i;
}
```

```
stq $16,0($30)
ldt $f1,0($30)
cvtqt $f1,$f0
```

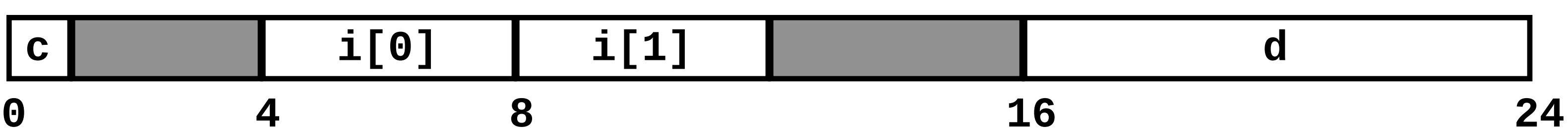
[Pass through stack and convert]

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

- **Primitive data type requires K bytes**
- **Address must be multiple of K**

Specific Cases

- **Long word data address must be multiple of 4**
- **Quad word data address must be multiple of 8**

Reason

- **Memory accessed by (aligned) quadwords**
  - Inefficient to load or store data that spans quad 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

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

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

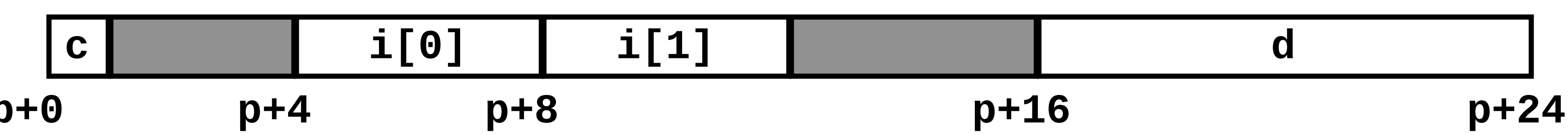
```
double struct_d(struct_ptr p)
{
  return p->d;
}
```

Result Computation

```
# address of 4th byte
  addq $16,4,$0
```

```
# Long word at 8th byte
  ldl $0,8($16)
```

```
# Double at 16th byte
   ldt $f0,16($16)
```



Accessing Byte in Structure

C Code

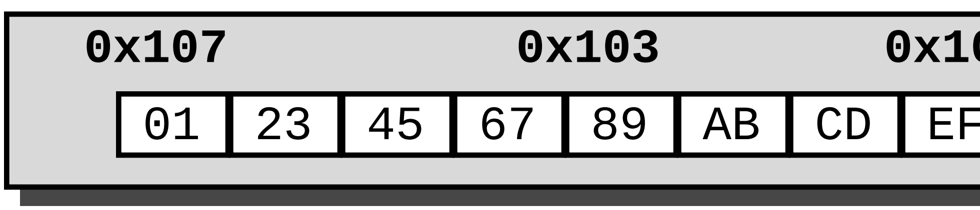
```
ar struct_c(struct_ptr p)

return p->c;
```

Result Computation

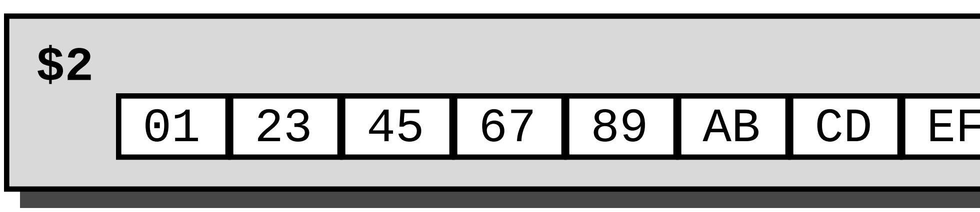
```
ldq_u $0,0($16)  # unaligned load
extbl $0,$16,$0  # extract byte p%8
sll $0,56,$0
sra $0,56,$0     # Sign extend char
```

Retrieving Single Byte From Memory

`$1 = 0x103`



- **`ldq_u $2, 0($1)` loads quad word at address `0x100`**
  – Aligned quad word containing address **`0x103`**



Byte Retrieval (Cont)

$2							
01	23	45	67	89	AB	CD	EF

- **`extbl $2, $1, $6` extracts byte 3 and copies into $6**
  – Uses low order 3 bits of $1 as byte number

$6							
00	00	00	00	00	00	00	89

- **`sll $6, 56, $6` moves low order byte to high position**

$6							
89	00	00	00	00	00	00	00

- **`sra $6, 56, $6` completes sign extension of selected byte**

$6							
FF	FF	FF	FF	FF	FF	FF	89

Arrays vs. Pointers

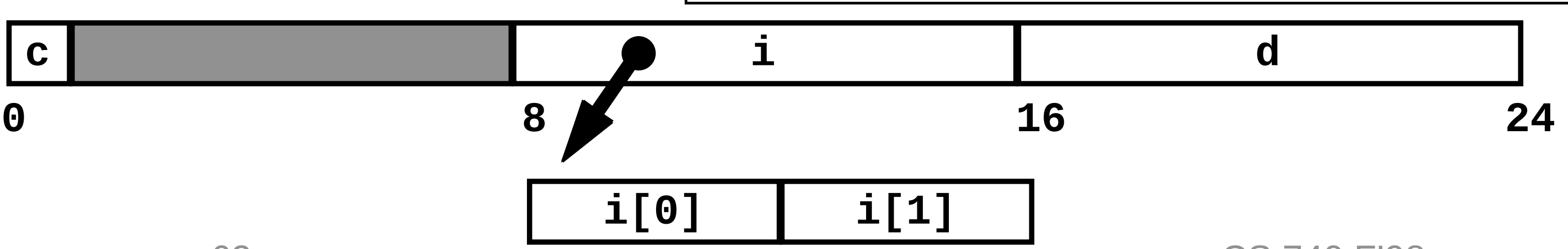
Recall

- **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 Code for Allocation

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

```
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

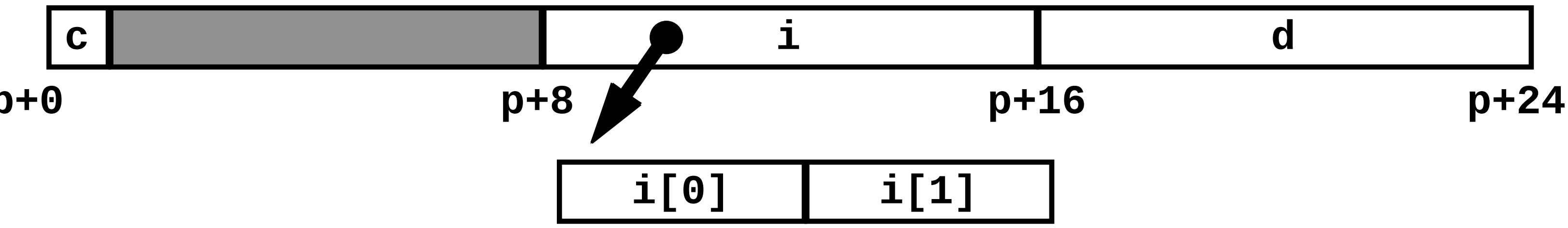
```
nt *pstruct_i(pstruct_ptr p)

 return p->i;
```

```
nt pstruct_i1(pstruct_ptr p)

 return p->i[1];
```

Result Computation

```
# quad word at 8th byte
  ldq $0,8($16)
```

```
# i = quad word at 8th byte from
  ldq $1,8($16)
# Retrieve i[1]
  ldl $0,4($1)
```



Arrays of Structures

rinciples

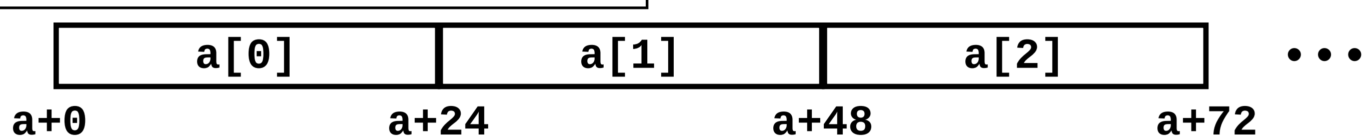
- **Allocated by repeating allocation for array type**
- **Accessed by computing address of element**
  - – Attempt to optimize
    - » Minimize use of multiplication
    - » Exploit values determined at compile time

C Code

```
* Index into array of
  struct_ele's */
truct_ptr a_index
   (struct_ele a[], int idx)

 return &a[idx];
```

Address Computation

```
s4subq $17,$17,$0   # 3 * idx
s8addq $0,$16,$0    # 24*idx + a
```

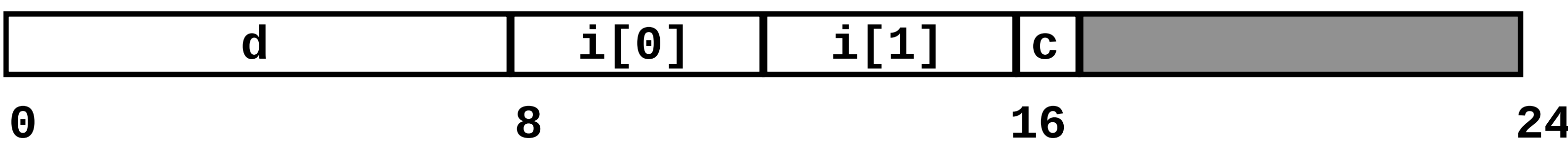


Aligning Array Elements

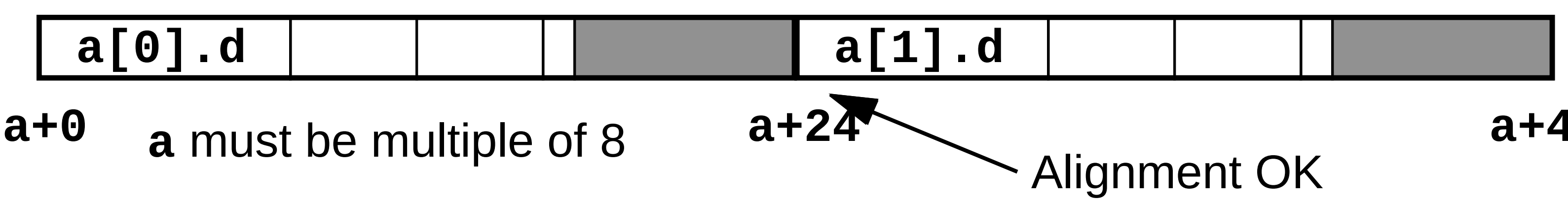
Requirement

- **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;
```




```
rev_ele a[2];
```

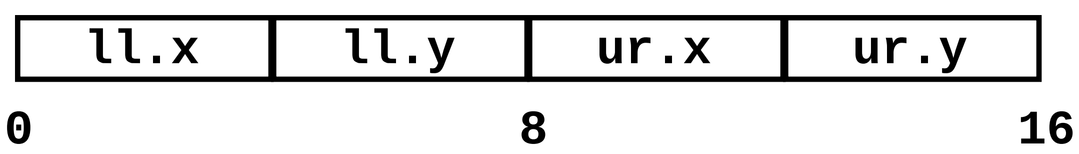


Nested Allocations

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;
```



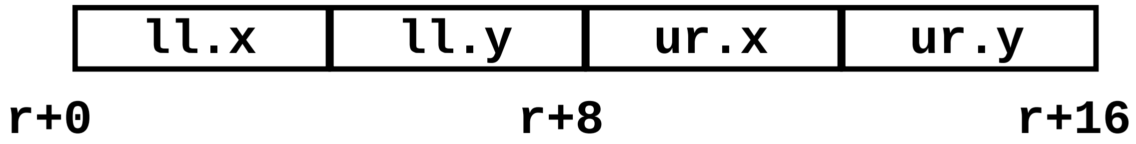
Nested Allocation (cont.)

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;
}
```

Computation

```
ldl $2,8($16)  # $2 = ur.x
ldl $1,0($16)  # $1 = ll.x
subl $2,$1,$2  # $2 = width
ldl $0,12($16) # $0 = ur.y
ldl $1,4($16)  # $1 = ll.y
subl $0,$1,$0  # $0 = height
mull $2,$0,$0  # $0 = area
```

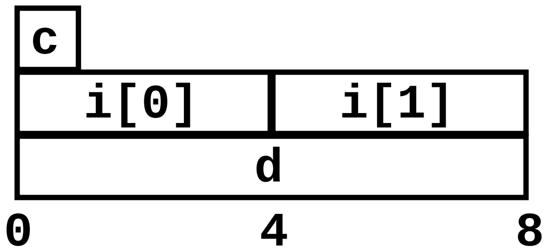


Union Allocation

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;
```



Example Use of Union

- Structure can hold 3 kinds of data
- Never use 2 forms simultaneously
- Identify particular kind with flag `type`

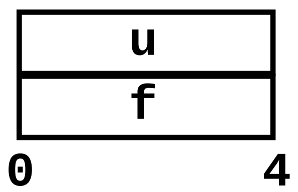
```
typedef enum { CHAR, INT, DOUBLE } utype;

typedef struct {
  utype type;
  union_ele e;
} store_ele, *store_ptr;
```

```
void print_store(store_ptr p)
{
  switch (p->type) {
  case CHAR:
    printf("Char = %c\n", p->e.c);
    break;
  case INT:
    printf("Int[0] = %d, Int[1] = %d\n",
           p->e.i[0], p->e.i[1]);
    break;
  case DOUBLE:
    printf("Double = %g\n", p->e.d);
  }
}
```

Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```

```
float bit2float(unsigned u) {
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```



- et direct access to bit epresentation of `float`
- `it2float` generates float with iven bit pattern
  - –NOT the same as `(float) u`
- `how_parts` extracts different omponents of float

```
void show_parts(float f) {
  int sign, exp, significand;
  bit_float_t arg;
  arg.f = f;
  /* Get bit 31 */
  sign = (arg.u >> 31) & 0x1;
  /* Get bits 30 .. 23 */
  exp = (arg.u >> 23) & 0xFF;
  /* Get bits 22 .. 0 */
  significand = arg.u & 0x7FFFFF;
  • • •
}
```

Byte Ordering

dea

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

ig Endian

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

ittle Endian

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

lpha

- Chip can be configured to operate either way
- Our’s are little endian
- Cray T3E Alpha’s are big endian

Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

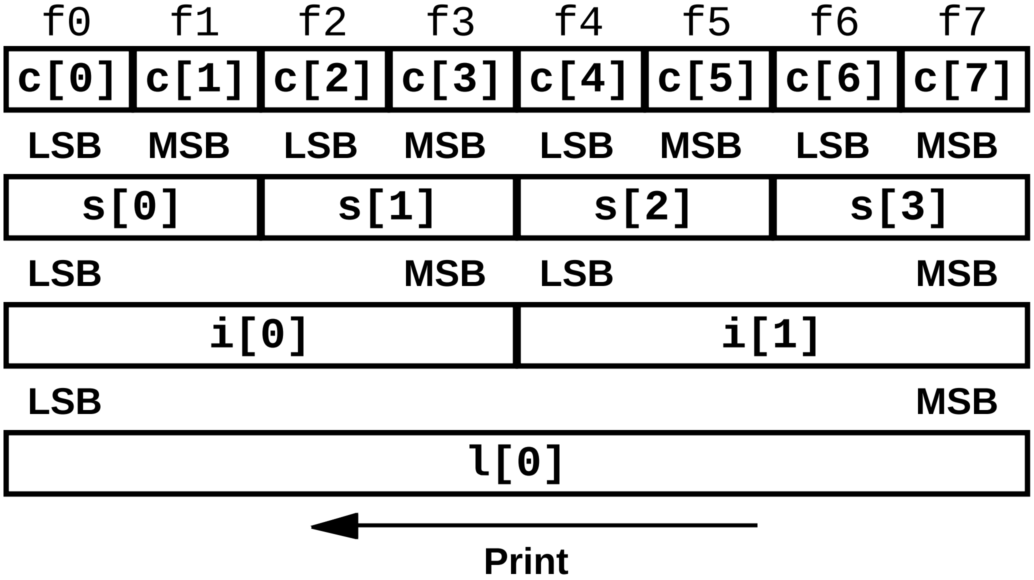
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
dw.c[j] = 0xf0 + j;
printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);
printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);
printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);
printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte Ordering on Alpha

Little Endian

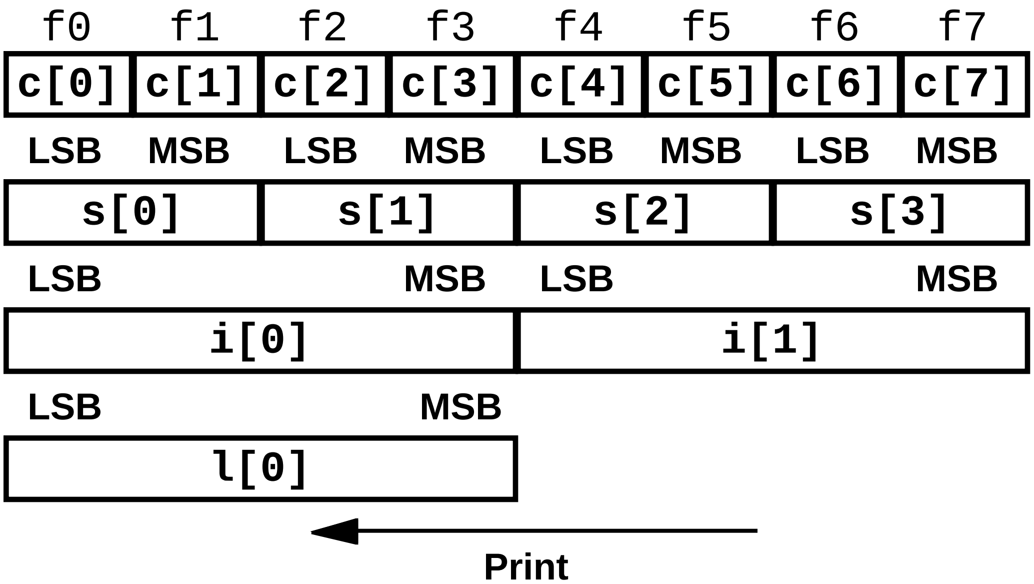


Output on Alpha:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

Byte Ordering on x86

Little Endian

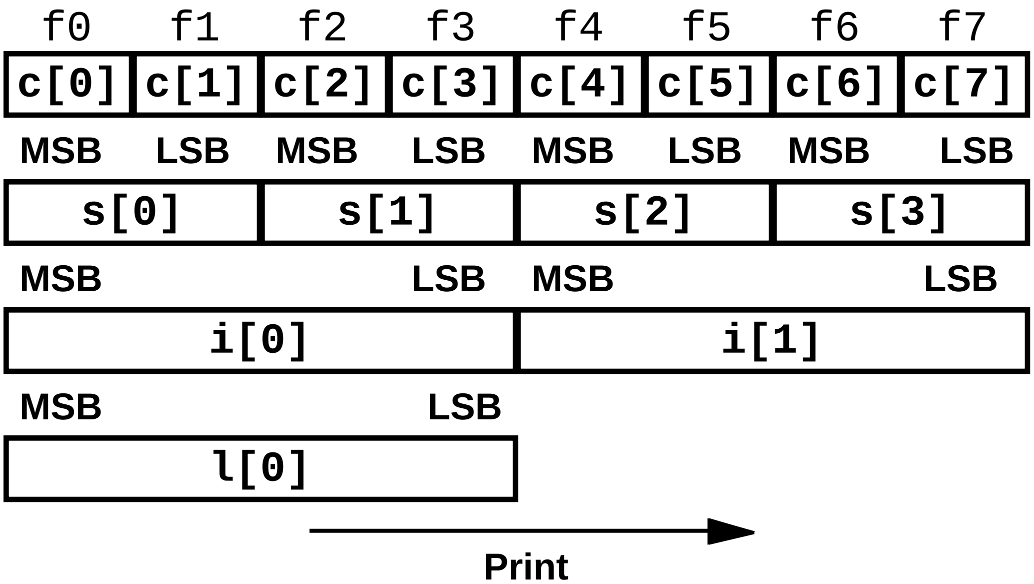


Output on Pentium:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [f3f2f1f0]
```

Byte Ordering on Sun

Big Endian



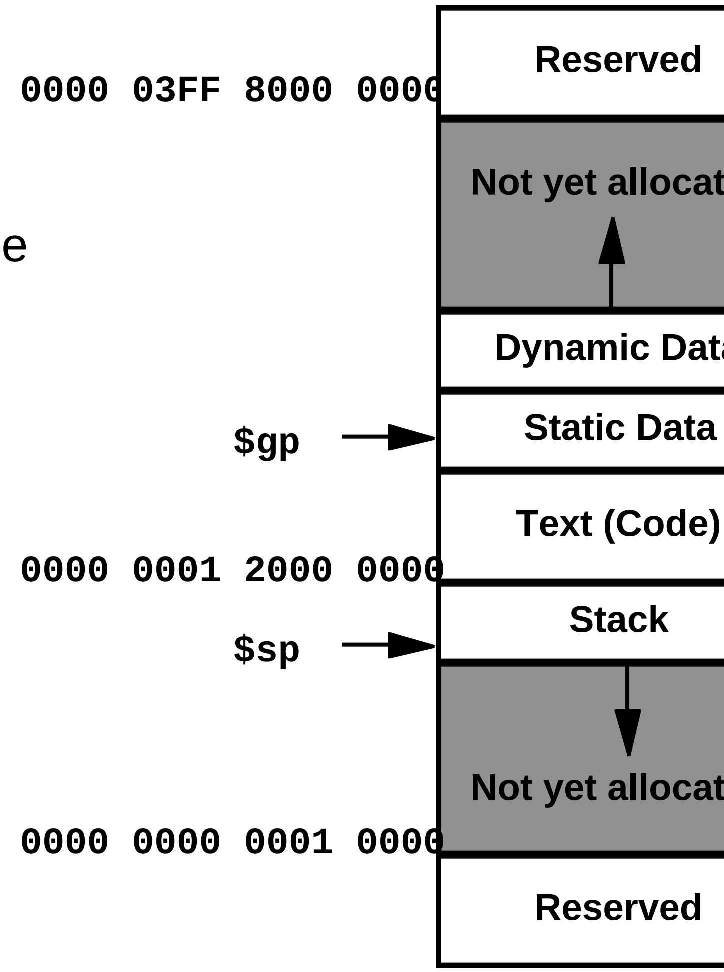
Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Alpha Memory Layout

gments

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



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**