Alpha Programming
CS 740
Sept. 8, 2003

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

Translation Process

C program (p1.c p2.c)
Compiler (gcc -S)

Assembler (gcc or as)

Object program (p1.o p2.o)
Linker (gcc or ld)

Executable program (p)
Disassembler (dis -h)
Debugger (gdb)

Disassembled program

Abstract Machines

Machine Model

Data

Control

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

mem

asm

proc

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

ASM

mem

regs

alu

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

Alpha Processors

Reduced Instruction Set Computer (RISC)
• Simple instructions with regular formats
  • Key Idea: 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
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

Return value from integer functions
Temporaries (not preserved across procedure calls)
Callee saved
Frame pointer, or callee saved

Important Ones for Now
$0 Return Value
$1...$8 Temporaries
$16 First argument
$17 Second argument
$26 Return address
$31 Constant 0

Program Representations

C Code
long int gval;
void test1(long int x, long int y)
{
  gval = (x+x+x) - (y+y+y);
}

Compilied 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

Obtain with command
gcc -O -S code.c
Produces file code.s

Registers (cont.)

Important Ones for Now
a0 $16
a1 $17
a2 $18
a3 $19
a4 $20
a5 $21
t8 $22
t9 $23
t10 $24
t11 $25
ra $26
pv, t12 $27
AT $28
gp $29
sp $30
zero $31

Integer arguments
Temporaries
Return address
Current proc addr or Temp
Reserved for assembler
Global pointer
Stack pointer
Always zero

Object
Disassembled

Ox120001130 <test1>:
  0x27bb2000
  0x23bd6f30
  0xa47d8098
  0x42100402
  0x40500402
  0x42310401
  0x40310401
  0x40410522
  0xb4430000
  0x6bfa8001

Obtain with command
x/10 Ox120001130
- Print 10 words in hexadecimal starting at address Ox120001130
dissassemble test1
- Print disassembled version of procedure

Run gdb on object code
x/10 Ox120001130
- Print 10 words in hexadecimal starting at address Ox120001130
**Alternate Disassembly**

**Alpha program “dis”**

- 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: 21bd8760 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**

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

**Compiled to Assembly**

```
.test2:
.align 3
.globl test2
.ent test2

addq $16,$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**

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

**Annotated Assembly**

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

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

**Array Indexing**

**C Code**

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

**Annotated Assembly**

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

```
long int
arefl(long 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
```

```
int
arefi(int a[], long int i)
{
  return a[i];
}
```
Array Indexing (Cont.)

C Code

```c
long int garray[10];
long int gref(long int i) {
    return garray[i];
}
```

Annotated Assembly

```assembly
.glabel gref
    lidgp $29,0($27)  # setup the gp
    lda $1,garray   # $1 = &garray[0]
    sadq $16,$1,$16  # $16 = &garray[0] + 8*i
    ldq $0,$(16)    # ret val = garray[i]
    ret $31,$(16),1 # return
```

Disassembled:

```
gref:
    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 sadq 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

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

Annotated Assembly

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

Structures & Pointers (Cont.)

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

C Code

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

Annotated Assembly

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

Structures & Pointers (Cont.)

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

C Code

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

Annotated Assembly

```assembly
set_p:
    stq $17,32($16)  # *r+32 = ptr
    ret $31,$(16),1
```
Structures & Pointers (Cont.)

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

C Code

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

Annotated Assembly

```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

- `bCond Ra, label`
  - Cond: branch condition, relative to zero
    - `beq` 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`

Unconditional Branches

- `br label`

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:

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

```
long int condbr(long int x, long int y) {
    long int v = 0;
    if (x > y) {
        v = x^x^x^y;
    }
    return v;
}
```

```
condbr:
    bis $31,$31,$0  # v = 0
cmp $16,$17,$1  # x <= y?
    bne $1,345,$0  # if so, branch
addq $16,$16,$0  # v = x^x
addq $0,$16,$0  # v = x^x
addq $0,$17,$0  # v = y
$45:
    ret $31,($26),1  # return v
```
**Conditional Move Example**

**C Code**

```c
long int max(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**

```c
long 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 looping
    ret $31,($26),1 # return result
```

**“While” Loop Example**

**C Code**

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

**Annotated Assembly**

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

**“For” Loops in C**

**Direct Translation**

```c
for (init; test; update )
{
    body
}
```

```
init;
while(test )
{
    body;update
}
```
"For" Loop Example

**C Code**

```c
/* Find max ele. in array */
long 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;
}
```

**Annotated Assembly**

```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 return
$63:
s8addq $3,$16,$1  # $1 = 8*i + &a[0]
ldq $2,0($1)  # $2 = a[i]
cmp $2,$0,$1  # if (a[i] <= res),
    bne $1,$62  # skip "then" part
    bis $2,$2,$0  # result = a[i]
$62:
    addq $3,1,$3  # i++
    cmplt $3,$17,$1  # if (i < count),
        bne $1,$63  # continue looping
$61:
    ret $31,($26),1  # return result
```

for (init; test; update )
body

init;
while(test)
{ body ; update }

Jumps

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

**Format:**

```assembly
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

**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

**C Code**

```c
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 '?';
    }
}
```

**Switch Statement Example**

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

**Enumerated Values**

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

**Assembly: Setup**

```assembly
# 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]
s8addq $16,$1,$1  # $1 = &jtab[0]
ld $1,0($1)  # $1 = jtab[0]
addq $1,$29,$2  # $2 = $gp + jtab[0]
jmp $31,($2),$68  # jump to jtab code
```
Jump Table

Table Contents

<table>
<thead>
<tr>
<th>Enumerated Values</th>
<th>Targets &amp; Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD 0  MULT 1  MINUS 2  DIV 3  MOD 4  BAD 5</td>
<td>$74: .gprel32 $68 .gprel32 $69 .gprel32 $70 .gprel32 $71 .gprel32 $72 .gprel32 $73</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.gprel32 $68</td>
<td>.gprel32 $69</td>
<td>.gprel32 $70</td>
<td>.gprel32 $71</td>
<td>.gprel32 $72</td>
<td>.gprel32 $73</td>
<td>.gprel32 $74</td>
<td>.gprel32 $75</td>
</tr>
</tbody>
</table>

Procedure Calls & Returns

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

Procedure 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

Procedure return:
- ret $31, ($26)  Jump to address in $26

C Code

```c
long int caller()
{ return callee(); }

long int callee()
{ return 5L; }
```

Annotated Assembly

```asm
long int caller();
{ return callee(); }

0x800 bsr $26, callee  # save return addr (0x804) in
0x804 ...  # $26, branch to callee
...

callee:
0x918 bis $31,5,$0  # return value = 5
0x91c ret $31,($26),1  # jump to addr in $26
```

Register Saving Conventions

When 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

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();
}
```

Stack (grows down)
Register Saving Examples

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

```assembly
yoo:
  bis $31, 17, $1
  stq $1, 8($sp) # save $1
  ber $26, who
  ldq $1, 8($sp) # restore $1
  addq $1, 1, $0
  ret $31, ($26)
```

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

```assembly
who:
  stq $9, 8($sp) # save $9
  bis $31, 6, $9 # overwrite $9
  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
/* Recursive factorial */
long int rfact(long int x)
{
  if (x <= 1)
    return 1;
  return x * rfact(x-1);
}
```

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

  frame: 16 bytes
  Virtual frame ptr @ $sp + 16
  Save registers $26 and $9
  No floating pt. regs. used
```

Stack Frame Example

```assembly
Procedure Prologue
rfact:
  1dgp $29,0($27) # setup gp
  rfact.ng:
    1da $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
```

```assembly
Procedure Epilogue
  1dq $26,0($30) # restore ret addr
  1dq $9,8($30) # restore $9
  addq $30,16,$30 # $sp += 16
  ret $31,($26),1
```

Stack Pointer
- $sp + 16
- save $9
- save $26

Frame Pointer
- $sp + 16
Stack Frame Example (Cont.)

C Code

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

Annotated Assembly

```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 epilogue.align 4
$80:
    bis $31,1,$9 # i = 1
$81:
    ldq $26,0($30) # restore ret addr
    ldq $9,8($30) # restore $9
    addq $30,16,$30 # $sp += 16
    ret $31,$(26),1
```

Stack Frame Example #2

C Code

```c
void 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
", i, vals[i]);
}
```

Procedure Prologue

```assembly
# show_facts:
ldgp $29,0($27) # setup gp
1da $30,-96($30) # $sp -= 96
.frame $30,96,$26,0 # branch to sp
stq $26,0($30) # save ret addr
stq $9,8($30) # save $9
.mask 0x4000200,-96
.prologue 1
bis $31,1,$9 # i = 1
```

Stack Pointer

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

Procedure Body

```assembly
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) # $2 = i
    addl $9,1,$9 # i++
    cmplt $9,0,$1 # if (i >= 0) then
    beq $1,$91 #  continue looping
```

Stack Pointer

- $sp + 96
- $sp + 88
- $sp + 24
- $sp + 16
- $sp + 8
- $sp + 0
- save $9
- save $26

Procedure Epilogue

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

Stack Pointer

- $sp + 96
- $sp + 88
- $sp + 24
- $sp + 16
- $sp + 8
- $sp + 0
- save $9
- save $26

Stack Frame Example #2 (Cont.)

C Code

```c
void 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
", i, vals[i]);
}
```

Procedure Prologue

```assembly
# show_facts:
ldgp $29,0($27) # setup gp
1da $30,-96($30) # $sp -= 96
.frame $30,96,$26,0 # branch to sp
stq $26,0($30) # save ret addr
stq $9,8($30) # save $9
.mask 0x4000200,-96
.prologue 1
bis $31,1,$9 # i = 1
```

Stack Pointer

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

Procedure Body

```assembly
bis $31,1,$9 # i = 1
$86:
    # $2 = 8*i + $sp
    addq $2,16,$2 # $2 = &vals[i]
    subl $9,1,$1 # $1 = i - 1
    # $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) # $2 = i
    addl $9,1,$9 # i++
    cmplt $9,0,$1 # if (i >= 0) then
    beq $1,$91 #  continue looping
```

Stack Pointer

- $sp + 96
- $sp + 88
- $sp + 24
- $sp + 16
- $sp + 8
- $sp + 0
- save $9
- save $26
Stack Addrs as Procedure Args

**C Code**

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

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

**Stack Addrs as Procedure Args (Cont.)

**C Code**

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

- **Stack frame**: 16 bytes
- Virtual frame ptr @ $sp + 16

**Stack Corruption Example**

**C Code**

```c
void 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;
}
```

- **Stack frame**: 16 bytes
- Virtual frame ptr @ $sp + 16

**Stack Corruption Example (Cont.)

**C Code**

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

- This code results in a segmentation fault on the Alpha!
Instruction Formats

Arithmetic Operations:
- all register operands
  - addq $1, $7, $5
- with a literal operand
  - addq $1, 15, $5

Branches:
- a single source register
  - bne $1, label

Jumps:
- one source, one dest reg
  - jsr $26, $1, hint

Loads & Stores:
- ldq $1, 16($30)

Basic Data Types

Integral
- 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

Floating 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

Different 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)
  - 0x0000000800000001L --> 2147483649
  - 0x0000000800000001 --> -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

0x0000000000000002L --> 4294967298
0x0000000000000002 --> 2 (truncated)
0x0000000000000001L --> 2147483649
0x0000000000000001 --> -2147483647 (extended)

0x0000000000000001L --> -2147483647 (extended)

0x0000000000000001 --> 2 (truncated)
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
• ld1 reads 4 bytes from memory and sign extends into register

ADDL Example

$1 = 0x0F0F0F0F0F0F

Integer Conversion Examples

C Code

```c
int long2int(long int li)
{
    return (int) li;
}

long int2long(int i)
{
    return (long) i;
}

unsigned ulong2uint(unsigned int u)
{
    return (unsigned) u;
}

long unsigned uint2ulong(unsigned int u)
{
    return (unsigned long) u;
}
```

Return Value Computation

```c
addl $16, $31, $0 # sign extend

[Replace high order bits with sign]

bis $16, $16, $0 # Verbatim copy

[Already in proper form]

addl $16, $31, $0 # sign extend

[Replace high order bits with sign. Even though really want 0's]

zapnot $16, $31, $0 # zero high bytes

[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
  - $1 = 0x0123456789abcdef

```
zap $1, 37, $2
37_{10} = 00010101_2
```

```
zapnot $1, 15, $2
15_{10} = 00001111_2
```
Floating Point Unit

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

Floating Point Formats
- S_Floating (C float): 32 bits
- T_Floating (C double): 64 bits

Floating Point Data Registers
- 32 registers, each 8 bytes
- Labeled $f0$ to $f31$
- $f31$ is always 0.0

Return Values
- Copys
- Call
- Call
- Save
- Save
- (Temporaries)

Floating Point Code Example

Compute Inner Product of Two Vectors
- Single precision

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

```
cpys $f31,$f31,$f0 # result = 0.0
bis $31,$31,$3 # i = 0
cmp $31,$18,$1 # 0 < n?
beq $1,$102 # if not, skip loop
.align 5
$104:
s4add $3,0,$1 # $1 = 4 * i
add $1,$16,$2 # $2 = &x[i]
add $1,$17,$1 # $1 = &y[i]
lsl $f1,0($2) # $f1 = x[i]
lsl $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

```c
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;
}
```

```
cpys $f31,$f31,$f0 # result = 0.0
bis $31,$31,$3 # i = 0
cmp $31,$18,$1 # 0 < n?
beq $1,$102 # if not, skip loop
.align 5
$104:
s8add $3,0,$1 # $1 = 4 * i
add $1,$16,$2 # $2 = &x[i]
add $1,$17,$1 # $1 = &y[i]
lsl $f1,0($2) # $f1 = x[i]
lsl $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,$318,$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

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

```
cvtt $f16,$f0
[Convert T_Floating to S_Floating]
```

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

```
addi $3,1,$3
add $1,$16,$2
add $1,$17,$1
ldt $f1,0($2)
```

```
cvtqt $f1,$f0
[cvtst]
```
Structure Allocation

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

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

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

Result Computation
- # address of 4th byte
- addq $16,4,$0

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

Result Computation
- # Long word at 8th byte
- ld1 $0,8($16)

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

Result Computation
- # Double at 16th byte
- ldt $f0,16($16)

Accessing Byte in Structure

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

Result Computation
- # address of 4th byte
- addq $16,4,$0

- # Long word at 8th byte
- ld1 $0,8($16)

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

Retrieving Single Byte From Memory

\( \$1 = 0x103 \)

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

```c
$2
```

0x100

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>23</th>
<th>45</th>
<th>67</th>
<th>89</th>
<th>AB</th>
<th>CD</th>
<th>EF</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$1</td>
<td>$2</td>
<td>$3</td>
<td>$4</td>
<td>$5</td>
<td>$6</td>
<td>$7</td>
<td>$8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x107</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 23 45 67 89 AB CD EF</td>
<td>01 23 45 67 89 AB CD EF</td>
</tr>
</tbody>
</table>

```c
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
```
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

```c
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;
}
```

Arrays of Structures

Principles

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

C Code

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

Address Computation

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

Accessing Through Pointer

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

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

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

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

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

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

```c
da[0].d    a[1].d    a+24
```

Alignment OK

da must be multiple of 8

a+24

## Nested Allocations

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

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

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

<table>
<thead>
<tr>
<th>ll.x</th>
<th>ll.y</th>
<th>ur.x</th>
<th>ur.y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

## Nested Allocation (cont.)

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

### Computation

```c
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

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

<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>
### Example Use of Union

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

```c
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);
        break;
    }
}
```

### Using Union to Access Bit Patterns

- Get direct access to bit representation of float
- `bit2float` generates float with given bit pattern
- `show_parts` extracts different components of float

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

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;
    
}

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

### Byte Ordering

**Idea**
- Bytes in long 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

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

### Byte Ordering Example

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

```plaintext

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>i[0]</td>
<td></td>
</tr>
</tbody>
</table>
```
Byte Ordering Example (Cont).

```c
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**

```
\begin{array}{cccccccc}
f0 & f1 & f2 & f3 & f4 & f5 & f6 & f7 \\
\mid \hline \\
LSB & MSB & LSB & MSB & LSB & MSB & LSB & MSB \\
\mid \hline \\
LSB & LSB & MSB & LSB & MSB & LSB & MSB & LSB \\
\mid \hline \\
LSB & MSB & LSB & MSB & LSB & MSB & LSB & MSB \\
\mid \hline \\
LSB & LSB & MSB & LSB & MSB & LSB & MSB & LSB \\
\mid \hline \\
LSB & LSB & MSB & LSB & MSB & LSB & MSB & LSB \\
\mid \hline \\
\end{array}
```

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**

```
\begin{array}{cccccccc}
f0 & f1 & f2 & f3 & f4 & f5 & f6 & f7 \\
\mid \hline \\
LSB & MSB & LSB & MSB & LSB & MSB & LSB & MSB \\
\mid \hline \\
\end{array}
```

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**

```
\begin{array}{cccccccc}
f0 & f1 & f2 & f3 & f4 & f5 & f6 & f7 \\
\mid \hline \\
\end{array}
```

Output on Sun:

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 == \[0xf7f6f5f4f3f2f1f0\]
Alpha Memory Layout

Segments
- **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