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

Alpha Processors

Reduction 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

Translation Process

<table>
<thead>
<tr>
<th>C program (p1.c p2.c)</th>
<th>Compiler (gcc -S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asm program (p1.s p2.s)</td>
<td>Assembler (gcc or as)</td>
</tr>
<tr>
<td>Object program (p1.o p2.o)</td>
<td>Linker (gcc or ld)</td>
</tr>
<tr>
<td>Executable program (p)</td>
<td>Libraries (.a)</td>
</tr>
<tr>
<td>Disassembler (dis -h)</td>
<td>Debugger (gdb)</td>
</tr>
<tr>
<td>disassembled program</td>
<td></td>
</tr>
</tbody>
</table>

Abstract Machines

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

ASM

<table>
<thead>
<tr>
<th>mem</th>
<th>regs</th>
<th>alu</th>
</tr>
</thead>
<tbody>
<tr>
<td>processor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Temporary (not preserved across procedure calls)

Callee saved

Frame pointer, or callee saved

Register (cont.)

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

Temporaries

Integer arguments

Temporaries

Current proc addr or Temp

Reserved for assembler

Global pointer

Stack pointer

Always zero

Program Representations

C Code

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

Obtain with command

gcc -O -S code.c

Produces file code.s

Compiled to Assembly

```assembly
.align 3
.globl test1
.type test1, oe
test1:
    ldp   $29,0($27)
.frame $30,0,$26,0
.prologue 1
    ldp   $30,0,264($27)
    stp   $5,31,32($27)
.stop
.ste
```

Run gdb on object code

```bash
x/10 0x120001130
```

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

Disassembled

```assembly
0x1200001130 <test1>:

0x1200001130 <test1>:
    ldp   $29,0($27)
    lda   gp, 28464(gp)
    ldq   t2, -32616(gp)
    addq a0, a0, t1
    addq t1, a0, t1
    addq a1, a1, t0
    addq t0, a1, t0
    subq t1, t0, t1
    stq   t1, 0(t2)
    ret    zero, (ra), 1
```

Run gdb on object code

```bash
x/10 0x120001130
```

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

- Print disassembled version of procedure
- 0x120001130
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, t0
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

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

```
Compiled to Assembly
.align 3
.globl test2
.ent test2
test2:
.frame $30,0,$26,0
.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
```

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;
}
```

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

Array Indexing

```
C Code
long int
aref1(long int a[], long int i)
{
return a[i];
}
```

```
Annotated Assembly
aref1:
addq $17,16,17 # $17 = 8*i + &a[0]
ldq $0,0($17) # return val = a[i]
ret $31,($26),1 # return
```

```
int
aref2(int a[], long int i)
{
return a[i];
}
```

```
Annotated Assembly
aref2:
addq $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
long int garray[10];
long int gref(long int i)
{
    return garray[i];
}

Annotated Assembly
.long int 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[i]
    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, (a0)
0x94 <gref+20>: 6bfa8001 ret zero, (ra), 1

Structures & Pointers

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

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

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

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

Disassembled:
find_a:
    s8addq $17,8,0 # $0 = 8+idx = 8
    addq $16,0,$0 # $0 += r
    ret $31,(0x26),1

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

Disassembled:
set_p:
    stq $17,32($16) # *(r+32) = ptr
    ret $31,(0x26),1
Structures & Pointers (Cont.)

C Code

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

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
```

void 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;
}

Branches

Conditional Branches

- 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
- Register value is typically set by a comparison instruction

Unconditional Branches

```assembly
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:
```assembly
cmovCond Ra, Rb, Rc
```
- Cond: comparison condition, Ra is compared with zero
  - some conditions as a conditional branch (eq, gt, le)
  - if (Ra Cond zero), then copy Rb into Rc

Psuedo-code example:
```c
if (x > 0) z = y;
```
- `cmovgt x, y, z`
**Conditional Move Example**

**C Code**

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

**Annotated Assembly**

```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**

```assembly
fact:
    bis $31,1,$0 # result = 1
    cmple $16,1,$1 # if (x <= 1) then
    bne $1,$50 # 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
    $51:
        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**

```assembly
ifact:
    bis $31,1,$0 # result = 1
    cmple $16,1,$1 # if (x <= 1) then
    bne $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
    $51:
        ret $31,(26),1 # return result
```

**“For” Loops in C**

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

**Direct Translation**

- **Init**;
- **While (test)**
- **Body ; update**
"For" Loop Example

C Code

```c
// Find max element 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

```
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:
saddq $3,$16,$1 # $3 = 8*i + &a[0]
    ldq $2,0($1) # $2 = a[i]
    cmple $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
```

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

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

Assembly: Setup

```
# op in $216
zapnot $16,15,$16 # zero upper 32 bits
cmplx $16,5,31 # if (op > 5) then
    beq $1,566 # branch to return
lda $1,576 # $1 = 6+tab[op]
saddq $3,51,$1 # $1 = 6+tab[op]
ldc $1,0,31 # $1 = 0
    addq $1,529,$2 # $2 = 0+6+tab[op]
jmp $31,020,368 # jump to jtab code
```

Enumerated Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0</td>
</tr>
<tr>
<td>MINT</td>
<td>1</td>
</tr>
<tr>
<td>DIV</td>
<td>2</td>
</tr>
<tr>
<td>MOD</td>
<td>3</td>
</tr>
<tr>
<td>BAD</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>
Jump Table

Table Contents

Enumerated Values

- ADD 0
- MUL 1
- MINUS 2
- DIV 3
- MOD 4
- BAD 5

Jump Table

Targets & Completion

| 66: | bsr $31,45,00 | # return '+' |
| 67: | ret $31,$(28),1 |
| 68: | bsr $31,42,00 | # return '*' |
| 69: | ret $31,$(28),1 |
| 70: | bsr $31,45,00 | # return '-' |
| 71: | ret $31,$(28),1 |
| 72: | bsr $31,47,00 | # return '/' |
| 73: | ret $31,$(28),1 |
| 66: | bsr $31,63,00 | # return '?' |

Procedural Calls & Returns

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

Procedure call:
- BAR $26, Label
- JAR $26, Rx

Procedure return:
- RET $31, ($26)

C Code

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

caller:
    ...}

callee:
    ...}
```

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

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
Register Saving Examples

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

```assembly
you:
  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)

who:
  bis $31, 6, $1 # overwrite $1
  ...
  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

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

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

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

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**

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

**Procedure Prologue**

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

Stack Frame Example

```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 ret addr
  stq $9,8($30) # save $9
  mask @4000200,-16
  prologue 1
```

**Procedure Epilogue**

- Stack Pointer
  - Frame Pointer
- Argument Build
- Save & pop
- Local & Temporaries
- (Virtual)
- Increasing Addresses

- CS 740 F'01
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 epilogue
    .align 4
$80:
    bis $31,1,$0 # return val = 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 Pointer
Frame Pointer

Stack Frame Example #2 (Cont.)

**C Code**

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\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
    Procedure Body

Stack Frame Example #2 (Cont.)

**C Code**

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\n", i, vals[i]);
}

**Procedure Body**

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
    Procedure Body

Stack Pointer
Frame Pointer
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**
- **val stored at $sp + 32**
- **"$sp + 32" passed as second argument ($17) to recursive call of rfact2**

Stack Pointer

- $sp + 0$
- $sp + 8$
- $sp + 16$
- $sp + 24$
- $sp + 32$
- $sp + 40$
- $sp + 48$

Caller

- $sp + 0$
- $sp + 8$
- $sp + 16$
- $sp + 24$
- $sp + 32$
- $sp + 40$
- $sp + 48$

Frame Pointer

- $sp + 0$
- $sp + 8$
- $sp + 16$
- $sp + 24$
- $sp + 32$
- $sp + 40$
- $sp + 48$


dfa

- 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 + 32
- bsr $26,rfact2

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;
}
```

This code results in a segmentation fault on the Alpha!

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
  - jst $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”
  - 0x0000000010000002L --> 4294967298
  - 0x0000000008000001L --> 2147483649
  - 0x0000000008000001 --> -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

0x0000000000000002 --> 2 (truncated)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2

0x0000000000000001 --> -2147483647 (extended)
F F F F F F F F 8 0 0 0 0 0 0 1

Page 12
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

-byte code-

Integer Conversion Examples

<table>
<thead>
<tr>
<th>C Code</th>
<th>Return Value Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int longint(int int li)</td>
<td>addl $16,$31,0 # sign extend (#2)</td>
</tr>
<tr>
<td></td>
<td>[Replace high order bits with sign]</td>
</tr>
<tr>
<td>long int2long(int i)</td>
<td>bis $16,$16,0 # Verbatim copy (#2)</td>
</tr>
<tr>
<td></td>
<td>[Already in proper form]</td>
</tr>
<tr>
<td>unsigned uint2uint(unsigned int u)</td>
<td>addl $16,$31,0 # sign extend (#2)</td>
</tr>
<tr>
<td></td>
<td>[Replace high order bits with sign. Even</td>
</tr>
<tr>
<td></td>
<td>though really want 0's]</td>
</tr>
<tr>
<td>long unsigned uint2ulong(unsigned int u)</td>
<td>zapnot $16,15,0 # zero high bytes (#2)</td>
</tr>
<tr>
<td></td>
<td>[Clear high order bits]</td>
</tr>
</tbody>
</table>

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

-byte code-

$1 = 0x0F0F0F0F0F0F0F0F

zap $1, 37, $2
37_{10} = 0000101012
zapnot $1, 15, $2
15_{10} = 000011112
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

### 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;
}
```

```assembly
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

```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;
}
```

```assembly
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]
    ld $f1,0($2) # $f1 = x[i]
    ld $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: cvtq, cvtst, cvtts, cvttp, …
- 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
double long2double(long i)
{
    return (double) i;
}
```

```assembly
stq $16,0($30)
ldt $f1,0($30)
cvtqt $f1,$f0 # Convert T_Floating to S_Floating
```

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

```assembly
cvtst $f16,$f0 # Convert S_Floating to T_Floating
```
**Structure Allocation**

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

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

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

**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**

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

**Accessing Byte in Structure**

**C Code**

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

**Result Computation**

```c
ldq_u $2,0($1) # unaligned load
extb $0,$16,$0 # extract byte p%8
sll $0,56,$0
sra $0,56,$0 # Sign extend char
```

**Retrieving Single Byte From Memory**

|$1 = 0xa103$ | $0xa107$ | $0xa100$
|---|---|---|
| $01$ | $23$ | $67$ | $89$ | $AB$ | $CD$ | $EF$

- ldq_u $2$, $0$(|$1$) loads quad word at address $0xa100$
  - Aligned quad word containing address $0xa103$
**Byte Retrieval (Cont)**

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

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

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

**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;
}
```

**Accessing Through Pointer**

**C Code**

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

**Result Computation**

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

**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_ele a_index
    (struct_ele a[1], int idx)
{
    return &a[idx];
}
```

**Address Computation**

```c
/* quad word at 8th byte from p */
ldq $0,8($16)
```

```c
/* quad word at 8th byte */
ldq $0,8($16)
```

```c
/* Retriev e i[1] */
ldl $0,4($1)
```

```c
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

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

![Alignment OK](attachment:image)

**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;
```

![Alignment OK](attachment:image)

**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**

- $l1 = 0, 8, 16$  
- $l2 = ur.x$  
- $l3 = ll.x$  
- $s1 = ur.y$  
- $s2 = ll.y$  
- $area = l1 * s1 * s2$  

![Alignment OK](attachment:image)

**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;
```

![Alignment OK](attachment:image)
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;

#include <stdio.h>

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
- NOT the same as (float) u
- show_parts extracts different components of float

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

#include <stdio.h>

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

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

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

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

#include <stdio.h>

u
0 4
f

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**

```
```

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 Sun

**Big Endian**

```
```

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]
```

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 == [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