**Alpha Programming**

**CS 740**  
Sept. 15, 2000

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

---

**Abstract Machines**

**Machine Model**

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) char</td>
</tr>
<tr>
<td>2) int, float</td>
</tr>
<tr>
<td>3) double</td>
</tr>
<tr>
<td>4) struct, array</td>
</tr>
<tr>
<td>5) pointer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) loops</td>
</tr>
<tr>
<td>2) conditionals</td>
</tr>
<tr>
<td>3) goto</td>
</tr>
<tr>
<td>4) Proc. call</td>
</tr>
<tr>
<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 |
| 2) word |
| 3) branch/jump |
| 4) jump & link |
| 5) address of initial byte |
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

Program Representations

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

Compiled to Assembly
.align 3
.globl test1
.text
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
- $0 Return Value
- $1 ... $8 Temporaries
- $16 First argument
- $17 Second argument
- $26 Return address
- $31 Constant 0

Integer arguments

Temporaries

Callee saved
- Frame pointer, or callee saved

Object
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>: subq $1, t0, t1
0x120001154 <test1+36>: ret zero, (ra), 1

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>: staq $1, t0, t1
0x120001154 <test1+36>: ret zero, (ra), 1

Run gdb on object code
x/10 0x120001130
- Print 10 words in hexadecimal starting at address
0x120001130
disassemble 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

```
text1:
0x0: 27ba0001 ldah gp, 1(t12)
0x4: 23b88760 lida gp, -30880(gp)
0x8: e7fd8010 ldq t2, -32768(gp)
0xc: 82100402 addq a0, a0, t1
0x10: 40500401 addq a1, a1, t0
0x14: 40320041 addq t0, a1, t0
0x18: 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

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

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

C Code

```
long int
test2(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 $0,$17,$0
subq $1,$0,$1
ret $31,($26),1
.end test2
```

C Code

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

Annotated Assembly

```
arefl:
    ldq $1,0($16) # $1 = a+i
    ldq $0,0($17) # $0 = a
    addq $1,0,$0 # return with a
    ret $31,($26),1 # value = a+i
```

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

Annotated Assembly

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

C Code

```
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 $0,0($17) # $0 = v
    stq $1,0($16) # *sum = $1
    ret $31,($26),1 # return
```

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

Annotated Assembly

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

C Code

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

Annotated Assembly

```assembly
.long garray,80
.gref:
    .comm garray,80
    .setgp $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
```

Disassembled:

```
0x80 <gref>: 23ba0001 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

C Code

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

Annotated Assembly

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

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

Disassembled:

```
:18 -
.cs 740 F00
```

```
0x00 <set_i>: 7bba0001 ldah gp, 65536(t12)
0x04 <set_i+4>: 23ba0000 lda gp, -31008(gp)
0x08 <set_i+8>: a43b8018 ldq t0, -32744(gp)
0x0c <set_i+12>: 42010650 s8addq a0, t0, a0
0x10 <set_i+16>: a4100000 ldq v0, 0(a0)
0x14 <set_i+20>: 6bfa8001 ret zero, (ra), 1
```

Structures & Pointers (Cont.)

C Code

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

```
0x18 -
.cs 740 F00
```

```
:14 -
.cs 740 F00
```

```
0x00 <set_i+20>: 6bfa8001 ret zero, (ra), 1
```

```
0x0c <set_i+12>: 42010650 s8addq a0, t0, a0
```

```
0x10 <set_i+16>: a4100000 ldq v0, 0(a0)
```

```
0x14 <set_i+20>: 6bfa8001 ret zero, (ra), 1
```

```
0x18 <set_i>: 7bba0001 ldah gp, 65536(t12)
```

```
0x20 <set_i+4>: 23ba0000 lda gp, -31008(gp)
```

```
0x24 <set_i+8>: a43b8018 ldq t0, -32744(gp)
```

```
0x28 <set_i+12>: 42010650 s8addq a0, t0, a0
```

```
0x2c <set_i+16>: a4100000 ldq v0, 0(a0)
```

```
0x30 <set_i+20>: 6bfa8001 ret zero, (ra), 1
```

Structures & Pointers (Cont.)

C Code

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

Annotated Assembly

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

C Code

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

Annotated Assembly

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

Disassembled:

```
:15 -
.cs 740 F00
```

```
0x00 <find_a>: 21ba0001 ldah gp, 65536(t12)
0x04 <find_a+4>: 21bd86e0 lda gp, -31008(gp)
0x08 <find_a+8>: a15b8018 ldq t0, -32744(gp)
0x0c <find_a+12>: 42010650 s8addq a0, t0, a0
0x10 <find_a+16>: a1a00000 ldq v0, 0(a0)
0x14 <find_a+20>: 6bfa8001 ret zero, (ra), 1
```

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

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

```
0x18 -
.cs 740 F00
```

```
0x1c <set_p>: 21ba0001 ldah gp, 65536(t12)
0x20 <set_p+4>: 21bd86e0 lda gp, -31008(gp)
0x24 <set_p+8>: a15b8018 ldq t0, -32744(gp)
0x28 <set_p+12>: 42010650 s8addq a0, t0, a0
0x2c <set_p+16>: a1a00000 ldq v0, 0(a0)
0x30 <set_p+20>: 6bfa8001 ret zero, (ra), 1
```
### Structures & Pointers (Cont.)

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

### C Code Annotated Assembly

```asm
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

- `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:**

```asm
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+y
    addq $0,$16,$0 # v = x+y
    addq $0,$17,$0 # v = y
    s45: ret $31,($24),1 # return v
```

**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**

```
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
    cmple $16,1,$1 # if (x <= 1) then
    bne $1,$51 #  branch to return
    $50:
    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**

```
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 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]
  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
```

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

Switch Statement Example

C Code

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

Enumerated Values

```assembly
# op in $26
zapnot $26,15,$26 # zero upper 16 bits
cmpule $26,5,$26 # if (op > 5) then
  bne $26,8,$26 # branch to return
  lda $26.5,01 # if (op > 5) then
  bne $26,8,$26 # branch to return
  lda $1,574 # $1 = 8*tab[$]
  s4addq $26,5,01 # $1 = 8*tab[$op]
  lda $26,0,01 # $1 = 8*tab[$op]
  addq $1,529,42 # $2 = 8*pr + 8*tab[$op]
  jmp $26.5,31,168 # jump to jtab code
```

Assembly: Setup

```assembly
$26:
  zanp $26,15,$26 # zero upper 16 bits
  cmpule $26,5,$26 # if (op > 5) then
  bne $26,8,$26 # branch to return
  lda $26.5,01 # if (op > 5) then
  bne $26,8,$26 # branch to return
  lda $1,574 # $1 = 8*tab[$]
  s4addq $26,5,01 # $1 = 8*tab[$op]
  lda $26,0,01 # $1 = 8*tab[$op]
  addq $1,529,42 # $2 = 8*pr + 8*tab[$op]
  jmp $26.5,31,168 # jump to jtab code
```
Jump Table

Table Contents

574:
.gprel32 068
   .gprel32 069
   .gprel32 070
   .gprel32 071
   .gprel32 072
   .gprel32 073

Targets & Completion

668:
   .gprel32 068
   .gprel32 069
   .gprel32 070
   .gprel32 071
   .gprel32 072
   .gprel32 073

Enumerated Values

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

Procedure Calls & Returns

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

Procedure call:
- bar $26, label
  Save return addr in $26, branch to label
- jar $26, (Ra)
  Save return addr in $26, jump to address in Ra

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

C Code

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

long int callee()
{
  return 5L;
}

caller:
...
0x800 bsr callee
  # save return addr (0x804) in $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();
  ...
}

Stack (grows down)

Call Chain

yoo

who

ami

ami

ami

ami

Frame Pointer

Stack Pointer

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
  ```
  bis $31, 17, $1
  stq $9, 8($sp) # save $9
  ```

**Callee Save**
- Callee must save / restore if overwriting
  ```
  bis $31, 17, $1
  ```

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

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

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 Example

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

**Procedure Prologue**
```asm
ldq $26, 0($26) # setup sp
rfact .reg:
1da $20, -16($10) # Eip <- 16
.frame $20, 16, $26, 0
stq $26, 0($10) # save ret addr
stq $9, 8($10) # save $9
```

**Procedure Epilogue**
```asm
1dq $26, 0($10) # restore ret addr
1dq $9, 8($10) # restore $9
add $30, 16, $30
ret $31, ($26)
```
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

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

### Stack Pointer
- $sp + 8
- $sp + 0
- $sp + 16
  - ...
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`

Frame Pointer

```
<table>
<thead>
<tr>
<th>Caller</th>
<th>Stack Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>$sp + 16</code></td>
</tr>
<tr>
<td></td>
<td><code>$sp + 24</code></td>
</tr>
<tr>
<td></td>
<td><code>$sp + 32</code></td>
</tr>
</tbody>
</table>
```

Stack Pointer

```
<table>
<thead>
<tr>
<th>Call not</th>
<th>Frame Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>$sp + 48</code></td>
</tr>
<tr>
<td></td>
<td><code>$sp + 64</code></td>
</tr>
<tr>
<td></td>
<td><code>$sp + 80</code></td>
</tr>
</tbody>
</table>
```

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 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;
}
```

Annotated Assembly

```
<table>
<thead>
<tr>
<th>Frame Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$sp + 16</code></td>
</tr>
<tr>
<td><code>$sp + 24</code></td>
</tr>
<tr>
<td><code>$sp + 32</code></td>
</tr>
</tbody>
</table>

Stack Pointer

```
<table>
<thead>
<tr>
<th>Call not</th>
<th>Frame Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>$sp + 48</code></td>
</tr>
<tr>
<td></td>
<td><code>$sp + 64</code></td>
</tr>
<tr>
<td></td>
<td><code>$sp + 80</code></td>
</tr>
</tbody>
</table>
```

Annotated Assembly

```
<table>
<thead>
<tr>
<th>Overwrite</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$sp + 16</code></td>
</tr>
</tbody>
</table>
```

Virtual frame ptr @ `$sp + 16`

-> overwrites callee stack!

Stack frame: 16 bytes

- Virtual frame ptr @ `$sp + 16`
- Padded to 16B alignment
- `buf[0]` stored at `$sp + 4`
- "$sp + 4" passed as first argument ($26) to `overwrite`
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)
- 0x0000000100000001L → 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

0x0000000010000002 → 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

$1 = 0x0FOFOFOFOFOFOFL

\[
\begin{array}{cccccccccc}
0 & F & 0 & F & 0 & F & 0 & F & 0 & F \\
0000 & 1111 & 0000 & 1111 & 0000 & 1111 & 0000 & 1111 & 0000 & 1111
\end{array}
\]

\[
\begin{array}{cccccccccc}
& & & & & & & & & \\
\text{addl} & \text{\$1, \$31, \$1} & & & & & & & & \\
& & & & & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

Integer Conversion Examples

C Code

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

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

unsigned longunsigned(long unsigned ul)
{
    return (unsigned long) ul;
}
```

Return Value Computation

- longint $16, $31, $0 # sign extend [Replace high order bits with sign]
- int2long $16, $31, $0 # Verbatim copy [Already in proper form]
- unsigned longunsigned $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
- zapnot a, b, c

<table>
<thead>
<tr>
<th>$1 = 0x0123456789abcdefL</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 23 45 00 09 89 AB CD EF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$1 = 0x0123456789abcdefL</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 23 45 00 09 89 AB CD EF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$1 = 0x0123456789abcdefL</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$1 = 0x0123456789abcdefL</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$1 = 0x0123456789abcdefL</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$1 = 0x0123456789abcdefL</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
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

- CallStack Save
- CallStack 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;
}
```

```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:
gr $3,0,$3 # $1 = 4 * i
add $31,$16,$3 # $3 = x[i]
add $31,$17,$1 # $1 = y[i]
ldt $f1,$f1($3) # $f1 = x[i]
ldt $f10,$f10($1) # $f10 = y[i]
mult $f1,$f10,$f1 # $f1 = x[i] * y[i]
addt $f0,$f1,$f0 # result += $f1
add $31,1,$3 # i++
cmp $31,$31,$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:
gr $3,0,$3 # $1 = 4 * i
add $31,$16,$3 # $3 = x[i]
add $31,$17,$1 # $1 = y[i]
ld $f1,$f1($3) # $f1 = x[i]
ld $f10,$f10($1) # $f10 = y[i]
mult $f1,$f10,$f1 # $f1 = x[i] * y[i]
add $f0,$f1,$f0 # result += $f1
add $31,1,$3 # i++
cmp $31,$31,$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, cvtq, 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
cvtts $f16,$f0
```

C Code Conversion Code

- Convert T_Floating to S_Floating

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

C Code Conversion Code

- Convert S_Floating to T_Floating

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

C Code Conversion Code

- Convert T_Floating to S_Floating

```c
cvtst $f16,$f0
```

C Code Conversion Code

- Convert S_Floating to T_Floating

```c
double long2double(long i)
```
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
ldl $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
ldl $0,8($16)
```

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

Retrieving Single Byte From Memory

<table>
<thead>
<tr>
<th>$1 = 0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x107</td>
</tr>
</tbody>
</table>

- \$1 = 0x103
  - \$1 = 0x103, \$2 = 0x0
    - loads byte at address 0x107
      - Data at 0x107: 01 23 45 67 89 AB CD EF
      - 0x107 = 0x103
      - Sign extend char
    - 1dq_u $2, $1(8) loads quad word at address 0x100
      - Aligned quad word containing address 0x103
      - \$1 = 0x103, \$2 = 0x0
        - \$1 = 0x103
```

```
# Unaligned load
addtb $0,$16,$0
```

```
# Extract byte p%8
setb $0,$16,$0
```

```
# Sign extend char
```

```
# Double at 16th byte
ldt $f0,16($16)
```
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
# quad word at 8th byte
ldq $0,8($16)
```

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[3](struct_ele a[], int idx)
{
    return &a[idx];
}
```

Address Computation

```c
a[0] a[1] a[2] ... a[72]
a+0 a+24 a+48 a+72
```
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;
```

```
0 8 16 24
```

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

```
a[0].d a[1].d
```

Alignment OK

```
a must be multiple of 8
```

```
Alignment OK
```

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;
```

```
0 8 16
```

```
ll.x ll.y ur.x ur.y
```

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;
}
```

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

```
ll.x ll.y ur.x ur.y
```

```
r+0 r+8 r+16
```

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;
```

```
0 4 8
```

```
| i[0] | i[1] |
```

```
d
```

Page 17
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;
```
 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**

```
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;0</td>
<td>&amp;1</td>
<td>&amp;2</td>
<td>&amp;3</td>
<td>&amp;4</td>
<td>&amp;5</td>
<td>&amp;7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
LSB | MSB | LSB | MSB | LSB | MSB | LSB | MSB |
----|-----|-----|-----|-----|-----|-----|-----|
&0  | &1  | &2  | &3  | &4  | &5  | &7  |
```

- **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]

Output on Alpha:

Print

```

Byte Ordering on x86

**Little Endian**

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;0</td>
<td>&amp;1</td>
<td>&amp;2</td>
<td>&amp;3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
LSB | MSB | LSB | MSB |
----|-----|-----|-----|
&0  | &1  | &2  | &3  |
```

- **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**  
  [0xf3f2f1f0]

Output on Pentium:

Print

```

Byte Ordering on Sun

**Big Endian**

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>@0</td>
<td>@1</td>
<td>@2</td>
<td>@3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
LSB | MSB | LSB | MSB |
----|-----|-----|-----|
@0  | @1  | @2  | @3  |
```

- **Characters 0-7**  
  [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- **Shorts 0-3**  
  [0xf0f1f2f3, 0xf4f5f6f7]
- **Ints 0-1**  
  [0xf0f1f2f3, 0xf4f5f6f7]
- **Long 0**  
  [0xf0f1f2f3]
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