Topics

- Basics
- Control Flow
- Procedures
- Instruction Formats
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

- **text**
  - C program (p1.c p2.c)
    - Compiler (gcc -S)
  - Asm program (p1.s p2.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

- **binary**
  - libraries (.a)
Abstract Machines

Machine Model

C

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

ASM

<table>
<thead>
<tr>
<th>Data</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) byte</td>
<td>3) branch/jump</td>
</tr>
<tr>
<td>2) 4-byte word</td>
<td>4) jump &amp; link</td>
</tr>
<tr>
<td>3) 8-byte word</td>
<td></td>
</tr>
<tr>
<td>4) contiguous word allocation</td>
<td></td>
</tr>
<tr>
<td>5) address of initial byte</td>
<td></td>
</tr>
</tbody>
</table>
### 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

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>$0</td>
</tr>
<tr>
<td>t0</td>
<td>$1</td>
</tr>
<tr>
<td>t1</td>
<td>$2</td>
</tr>
<tr>
<td>t2</td>
<td>$3</td>
</tr>
<tr>
<td>t3</td>
<td>$4</td>
</tr>
<tr>
<td>t4</td>
<td>$5</td>
</tr>
<tr>
<td>t5</td>
<td>$6</td>
</tr>
<tr>
<td>t6</td>
<td>$7</td>
</tr>
<tr>
<td>t7</td>
<td>$8</td>
</tr>
<tr>
<td>s0</td>
<td>$9</td>
</tr>
<tr>
<td>s1</td>
<td>$10</td>
</tr>
<tr>
<td>s2</td>
<td>$11</td>
</tr>
<tr>
<td>s3</td>
<td>$12</td>
</tr>
<tr>
<td>s4</td>
<td>$13</td>
</tr>
<tr>
<td>s5</td>
<td>$14</td>
</tr>
<tr>
<td>s6, fp</td>
<td>$15</td>
</tr>
</tbody>
</table>

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

### Important Ones for Now

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>Return Value</td>
</tr>
<tr>
<td>$1..$8</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$16</td>
<td>First argument</td>
</tr>
<tr>
<td>$17</td>
<td>Second argument</td>
</tr>
<tr>
<td>$26</td>
<td>Return address</td>
</tr>
<tr>
<td>$31</td>
<td>Constant 0</td>
</tr>
</tbody>
</table>

### Integer arguments

- $a0, $a1, $a2, $a3, $a4, $a5
- $t8, $t9, $t10, $t11
- $ra
- $pv, $t12

### Temporaries

- $at
- $gp
- $sp
- $zero

### Return address

- $at
- $gp
- $sp
- $zero

### Current proc addr or Temp

- $at
- $gp
- $sp
- $zero

### Reserved for assembler

- $at
- $gp
- $sp
- $zero

### Global pointer

- $at
- $gp
- $sp
- $zero

### Stack pointer

- $at
- $gp
- $sp
- $zero

### Always zero

- $at
- $gp
- $sp
- $zero
Program Representations

C Code

```c
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
.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`
Run gdb on object code

\[\text{x/10 0x120001130}\]

- Print 10 words in hexadecimal starting at address 0x120001130

\text{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
```

test1:

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

## C Code

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

## Compiled to Assembly

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

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

*Place result in $0*
Pointer Examples

C Code

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

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

Annotated Assembly

```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
```

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

C Code

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

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

Annotated Assembly

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

arefi:
    s4addq $17,$16,$17 # $17 = 4*i + &a[0]
    ld1 $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
.comm garray,80

gref:
    ldgp $29,0($27)  # setup the gp
    lda $1,garray   # $1 = &garray[0]
    s8addq $16,$1,$16 # $16 = 8*i + $1
    ldq $0,0($16)   # ret val = garray[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, 0(a0)
0x94 <gref+20>: 6bfa8001 ret zero, (ra), 1
```
```c
struct rec {
    long int i;
    long int a[3];
    long int *p;
};

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

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

**C Code**

**Annotated Assembly**
Structures & Pointers (Cont.)

C Code

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

Annotated Assembly

```assembly
find_a:
    s8addq $17,8,$0  # $0 = 8*idx + 8
    addq $16,$0,$0   # $0 += r
    ret $31,($26),1
```
struct rec {
    long int i;
    long int a[3];
    long int *p;
};

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

set_p:
    stq $17, 32($16) # *(r+32) = ptr
    ret $31, ($26), 1
Structures & Pointers (Cont.)

C Code

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

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;
}
```

Annotated Assembly

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

“bis” = bitwise OR
Branches

Conditional Branches

\[ \text{bCond Ra, label} \]

- \text{Cond}: branch condition, relative to zero

- \text{bne}: Equal \quad \text{Ra} == 0
- \text{bne}: Not Equal \quad \text{Ra} != 0
- \text{bgt}: Greater Than \quad \text{Ra} > 0
- \text{bge}: Greater Than or Equal \quad \text{Ra} >= 0
- \text{blt}: Less Than \quad \text{Ra} < 0
- \text{ble}: Less Than or Equal \quad \text{Ra} <= 0

- Register value is typically set by a \textit{comparison} instruction

Unconditional Branches

\[ \text{br label} \]
Conditional Branches

Comparison Instructions

- Format: \texttt{cmpCond Ra, Rb, Rc}
  
  - \textit{Cond}: comparison condition, Ra relative to Rb

  \begin{itemize}
  \item \texttt{cmpeq} Equal \quad Rc = (Ra == Rb)
  \item \texttt{cmplt} Less Than \quad Rc = (Ra < Rb)
  \item \texttt{cmple} Less Than or Equal \quad Rc = (Ra <= Rb)
  \item \texttt{cmpult} Unsigned Less Than \quad Rc = (uRa < uRb)
  \item \texttt{cmpule} Unsigned Less Than or Equal \quad Rc = (uRa <= uRb)
  \end{itemize}

C Code

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

Annotated Assembly

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

$45:
    ret $31,($26),1  # return v
```
Conditional Move Instructions

Motivation:
• conditional branches tend to disrupt pipelining & hurt performance

Basic Idea:
• conditional moves can replace branches in some cases
  – avoids disrupting the flow of control

Mechanism:

\[ \text{cmovCond Ra, Rb, Rc} \]

• \textbf{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:

\[
\text{if (x > 0) z = y; => cmovgt x, y, z}
\]
## 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

```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
    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:
    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
```

```c
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

```
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
$s63:
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
$62:
    bis $2,$2,$0  # result = a[i]
$61:
    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:

\[
jmp \text{ Ra, (Rb), Hint}\]

• \(\text{Rb}\) contains the target address
• for now, don’t worry about the meaning of \(\text{Ra}\) or “\(\text{Hint}\)”
• synonyms for \(\text{jmp}\): \text{jsr, ret}
Compiling Switch Statements

### 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 '?';
    }
}
```

### Implementation Options

- **Series of conditionals**
  - Good if few cases
  - Slow if many
- **Jump Table**
  - Lookup branch target
  - Avoids conditionals
  - Possible when cases are small integer constants
- **GCC**
  - Picks one based on case structure
Switch Statement Example

C Code

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

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

Enumerated Values

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

Assembly: Setup

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

## Table Contents

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

## Enumerated Values

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

##Targets & Completion

<table>
<thead>
<tr>
<th>$68:</th>
</tr>
</thead>
<tbody>
<tr>
<td>bis $31,43,$0 # return <code>+</code></td>
</tr>
<tr>
<td>ret $31,($26),1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$69:</th>
</tr>
</thead>
<tbody>
<tr>
<td>bis $31,42,$0 # return <code>*</code></td>
</tr>
<tr>
<td>ret $31,($26),1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$70:</th>
</tr>
</thead>
<tbody>
<tr>
<td>bis $31,45,$0 # return <code>-</code></td>
</tr>
<tr>
<td>ret $31,($26),1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$71:</th>
</tr>
</thead>
<tbody>
<tr>
<td>bis $31,47,$0 # return <code>/</code></td>
</tr>
<tr>
<td>ret $31,($26),1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$72:</th>
</tr>
</thead>
<tbody>
<tr>
<td>bis $31,37,$0 # return <code>%</code></td>
</tr>
<tr>
<td>ret $31,($26),1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$73:</th>
</tr>
</thead>
<tbody>
<tr>
<td>bis $31,63,$0 # return <code>?</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$66:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret $31,($26),1</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

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

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

Languages that support recursion
  • e.g., C, Pascal

Stack Allocated in *Frames*
  • state for procedure invocation
    – return point, arguments, locals

Code Example

```
yoo(...) {
  ...
  who();
  ...
}

who(...) {
  ...
  amI();
  ...
}

amI(...) {
  ...
  amI();
  ...
}
```
Register Saving Conventions

When procedure you calls who:
  • you 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 you 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

```
yoo:
  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
  
  ret $31, ($26)
```

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

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

who:
  stq $9, 8($sp)  # save $9
  bis $31, 6, $9  # overwrite $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 Pointer
($sp$)

Frame Pointer
($fp$)

Increasing Addresses

Stack Grows

Argument Build

Locals & Temporaries

saved reg m

saved reg 2

saved reg 1

arg 8
arg 7

arg n
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

```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 0x4000200,-16
    .prologue 1
```

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

Procedure Epilogue

```assembly
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 (Cont.)

C Code

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

Stack Pointer

- $sp + 16$
- $sp + 8$
- $sp + 0$

Frame Pointer

- $sp + 16$
- $sp + 8$
- $sp + 0$

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

- $sp + 16$
- $sp + 8$
- $sp + 0$

- save $9$
- save $26$
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\n", i, vals[i]);
}
```

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

Procedure Prologue

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

Procedure Prologue

```assembly
show_facts:
    ldgp $29,0($27)    # restore ret addr
    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 Epilogue

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

Frame Pointer

- $sp + 96
- $sp + 88
- vals[9]
- vals[9]
- vals[9]
- vals[9]
- vals[9]
- vals[9]
- vals[9]
- vals[9]
- vals[9]

Stack Pointer

- $sp + 24
- vals[1]
- vals[1]
- vals[1]
- vals[1]
- vals[1]
- vals[1]
- vals[1]
- vals[1]
- vals[1]
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\n", i, vals[i]);
}
```

Procedure Body

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

Frame Pointer

\[
\begin{array}{c}
\text{vals[9]} \\
\text{vals[1]} \\
\text{vals[0]} \\
\text{save $9$} \\
\text{save $26$}
\end{array}
\]

Stack Pointer

\[
\begin{array}{c}
\text{$sp + 96$} \\
\text{$sp + 88$} \\
\text{$sp + 24$} \\
\text{$sp + 16$} \\
\text{$sp + 8$} \\
\text{$sp + 0$}
\end{array}
\]
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`
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;
}
```

### rfact2:

```
lda $30,-48($30)  # $sp -= 48
stq $26,0($30)  # save $26
stq $9,8($30)  # save $9
stq $10,16($30)  # save $10
bis $16,$16,$9  # $9 = x
bis $17,$17,$10  # $10 = result
cmple $9,1,$10  # if (x > 1) then
beq $1,$83  # branch to $83
bis $31,1,$1  # $1 = 1
br $31,$85  # go to epilogue
```

```
$83:
subq $9,1,$16  # arg1 = x - 1
addq $30,32,$17  # arg2 = $sp + 32
bsr $26,rfact2  # rfact2(x-1,&val)
ldq $1,32($30)  # $1 = val
mulq $9,$1,$1  # $1 = x * val
```

```
$85:
stq $1,0($10)  # store to *result
ldq $26,0($30)  # restore $26
ldq $9,8($30)  # restore $9
ldq $10,16($30)  # restore $10
addq $30,48,$30  # $sp += 48
ret $31,($26),1  # return
```
Stack Corruption Example

C Code

```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;
}
```

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

This code results in a segmentation fault on the Alpha!
Stack Corruption Example (Cont.)

C Code

```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;
}
```

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

`-> overwrites caller stack frame!`

Annotated Assembly

```
overwrite:
    lda $30,-16($30)  # $sp -= 16
    ldl $1,16($30)    # $1 = a6
    stq $16,0($30)    # buf[0] = a0
    stq $17,8($30)    # buf[1] = a1
    stq $18,16($30)   # buf[2] = a2
    stq $19,24($30)   # buf[3] = a3
    stq $20,32($30)   # buf[4] = a4
    stq $21,40($30)   # buf[5] = a5
    stq $22,48($30)   # buf[6] = a6
    stq $31,56($30)   # buf[7] = 0
    addq $30,16,$30   # $sp += 16
    ret $31,($26),1
```

Stack Pointer

```
$sp + 0
$sp + 8
$sp + 16
$sp + 24
```

Frame Pointer

```
$26 (caller)
a6
(padded)
buf[0]
```

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

`-> overwrites caller stack frame!`
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)